50376
2005
371

n° d'ordre : 3668

Modélisation Unifiée des Aspects Répétitifs dans la Conception Conjointe Logicielle/Matérielle des Systèmes sur Puce à Hautes Performances

Thèse

présentée à

L'Université des Sciences et Technologies de Lille

Pour obtenir le titre de

Docteur en Informatique

par

Arnaud Cuccuru

Soutenue le 29 novembre 2005
devant la commission d'examen composée de :

Jean-Marc Geib,	Président,	LIFL
Jean-Luc Dekeyser,	Directeur,	LIFL
Jean-Marc Jezequel,	Rapporteur,	IRISA
François Terrier,	Rapporteur,	CEA
Pierre Boulet,	Examinateur,	LIFL
Robert de Simone,	Examinateur,	INRIA

SCD LILLE 1
D 030 311585 5

UNIVERSITÉ DES SC. ET TECHN.
BIBLIOTHÈQUE UNIVERSITAIRE
LILLE

Remerciements [1]

Au professeur Jean-Luc Dekeyser, pour m'avoir fait confiance alors que j'étais prédestiné à une carrière de comptable.

Au professeur Jean-Marc Geib, pour m'avoir fait l'honneur d'accepter de présider mon jury de thèse.

Aux professeurs Jean-Marc Jezequel et François Terrier, pour avoir accepter d'évaluer mes travaux, avec l'investissement en temps et en énergie que cela implique, et pour leurs précieux conseils et leurs remarques avisées.

A Robert de Simone, directeur de recherche INRIA, et au professeur Pierre Boulet, pour leur soutien, leurs conseils, et leur présence en tant qu'examinateurs.

A tous les membres de l'équipe WEST, passés (au moins à ceux que j'ai connu), présents (pour leur soutien, leurs idées, ou à défaut leur bonne humeur), et futurs (au moins à ceux qui liront cette thèse).

A tous mes proches, qui m'ont soutenu aussi bien matériellement que moralement, et m'ont permis de me remettre régulièrement en cause, en me posant des questions fréquentes et pertinentes sur mes travaux : "En quoi ça consiste une thèse ?", "Bon, je sais que je te l'ai déjà demandé, mais tu travailles sur quoi exactement ?", ou encore "Et où est-ce que ça va t'ammener ?". Pour répondre à cette dernière question, il est difficile d'éviter les lieux communs. Je serai tenté de répondre : "L'avenir nous le dira". Plus prosaiquement, je répondrais que dans le pire des cas, cette thèse m'ammènera au bureau de l'ANPE le plus proche de chez moi. Avec un peu de chance, on m'y proposera une formation de boucher ou d'entretien des espaces verts [2]...

A Thomson, pour avoir inventé le MO5.

Pour conclure la section des remerciements, il est d'usage de faire une dédicace. Parfois, le travail est dédié *"à celui ou celle qui n'a pas encore vu le jour au moment où ces lignes sont écrites"*, ou encore *"à toutes les filles que j'ai aimé avant"*. Mais après tout, ça n'est qu'une modeste thèse de doctorat en informatique, parmi les centaines qui doivent être soutenues chaque année dans le monde, une goutte de contribution dans l'océan de la science [3]. Je vais donc dédier cette thèse aux arbres qui vont se sacrifier pour permettre son impression [4], et conclure cette section par une citation écologique, à méditer...

Si vous êtes mordus par une vipère, sucez-vous le genou, ça fera marrer les écureuils.
Pierre Desproges

---

[1] **remerciements** n. m. pl. : Note de reconnaissance d'un auteur envers ceux qui l'ont aidé dans la rédaction de son ouvrage. [Office de la langue française, 1982]

[2] Il n'y a pas de sous métiers, l'essentiel étant de gagner sa vie honnêtement.

[3] Après la comptabilité, la poésie était ma deuxième passion...

[4] ... l'écologie la troisième.

Table des matières

1 Introduction **9**
- 1 Contexte . . . 11
- 2 Contribution . . . 12
- 3 Plan . . . 12

2 Positionnement **15**
- 1 Tendances méthodologiques . . . 17
- 2 Tendances technologiques . . . 19
- 3 Le flot de conception Gaspard . . . 21
- 4 Programmation à parallèlisme de données . . . 25
  - 4.1 Fortran . . . 27
  - 4.2 High Performance Fortran . . . 27
    - 4.2.1 Directive de parallélisation . . . 28
    - 4.2.2 Directives de placement des données . . . 28
  - 4.3 ArrayOL . . . 29
    - 4.3.1 Expression des dépendances de données . . . 30
    - 4.3.2 Factorisation de l'expression des dépendances . . . 30
    - 4.3.3 Unification des dimensions temporelles et spatiales . . . 33
    - 4.3.4 Exemple : Produit de matrices . . . 34
  - 4.4 Bilan . . . 34
- 5 Modélisation des architectures matérielles . . . 36
  - 5.1 HDLs . . . 37
  - 5.2 Langages niveau système . . . 37
  - 5.3 ADLs . . . 38
  - 5.4 Méthodes orientées UML . . . 38
  - 5.5 Autres outils et langages . . . 39
  - 5.6 Bilan . . . 40

3 Contribution des standards OMG à la définition du profil Gaspard **41**
- 1 SPT . . . 44
  - 1.1 Modélisation : General Resource Modeling Framework . . . 44
  - 1.2 Analyse : Analysis Models . . . 46
  - 1.3 Classification des ressources . . . 47
  - 1.4 Bilan . . . 47
- 2 QoS . . . 48
  - 2.1 Framework de définition des QoS . . . 48
  - 2.2 Méthodologie d'annotation des modèles . . . 49

2.3 Bilan . . . 50
3 SysML . . . 51
3.1 SysML vs UML 2 . . . 52
3.2 Allocation . . . 52
3.3 Bilan . . . 54
4 Uml for SoC . . . 55
4.1 Extensions pour la modélisation structurelle des SoCs . . . 55
4.2 Bilan . . . 55

4 Le profil Gaspard **57**
1 Architecture du profil . . . 60
2 Package "component" . . . 61
2.1 Vue domaine . . . 61
2.1.1 Encapsulation . . . 62
2.1.2 Composition et Assemblage . . . 63
2.1.3 Paramétrages . . . 64
2.2 Vue UML . . . 65
2.2.1 Composants . . . 67
2.2.2 Interfaces . . . 68
2.2.3 Paramètres . . . 73
3 Package "factorization" . . . 76
3.1 Vue domaine . . . 76
3.1.1 Multiplicités multi-dimensionnelles . . . 76
3.1.2 Modélisation de la topologie des liens . . . 77
3.2 Vue UML . . . 81
3.2.1 Limitations des mécanismes structurels d'UML . . . 81
3.2.2 Multiplicités multi-dimensionnelles . . . 84
3.2.3 Modélisation de la topologie des liens . . . 85
4 Package "hardwareArchitecture" . . . 88
4.1 Vue domaine . . . 88
4.1.1 Classification des ressources . . . 89
4.1.2 Communications . . . 97
4.2 Vue UML . . . 99
4.2.1 Modélisation des composants . . . 99
4.2.2 Modélisation des interfaces . . . 100
4.2.3 Exemple d'architecture : Quadri-processeurs Mips . . . 101
4.3 Impact des mécanismes de factorisation . . . 101
5 Package "application" . . . 105
5.1 Vue domaine . . . 105
5.1.1 Composants applicatifs . . . 105
5.1.2 Communications . . . 105
5.2 Vue UML . . . 107
5.2.1 Modélisation des composants . . . 107
5.2.2 Modélisation des interfaces . . . 107
5.3 Impact des mécanismes de factorisation . . . 108
5.3.1 Banalisation des dimensions spatiales et temporelles . . . 108
5.3.2 Expression du parallélisme de données . . . 109

5.3.3 Dépendances inter-répétitions 109
6 Package "association" 112
6.1 Vue domaine 112
6.1.1 Caractérisation 112
6.1.2 Allocation 114
6.2 Vue UML 116
6.3 Impact des mécanismes de factorisation 116

5 Étude de cas **123**
1 Application : DownScaler 125
1.1 Filtre horizontal 125
1.2 Filtre vertical 128
1.3 Structure de l'application 129
1.4 Optimisation 129
2 Architecture matérielle : Cluster de multi-mips 132
2.1 Nœud multi-mips 132
2.2 Réseau d'interconnexion Ω N×N 135
2.3 Cluster de multi-mips 137
3 Association 139
3.1 Dimensionnement de l'architecture et transformation de l'application 139
3.2 Modélisation du placement 141

6 Conclusion et perspectives **145**
1 Bilan 147
2 Perspectives 148
2.1 Alignement avec MARTE 148
2.2 Introduction du contrôle dans les modèles 149
2.3 Extension du modèle d'association 150
2.4 Prise en compte de l'environnement 150
2.5 Extension des mécanismes de factorisation 151

CHAPITRE 1

Introduction

1 Contexte 11
2 Contribution 12
3 Plan 12

1 Contexte

Dans les années 1990, l'évolution des technologies d'intégration sur puce (les "submicrons", ou technologies de gravure inférieures à 0.35 micron) a permis l'apparition d'un nouveau paradigme dans le domaine des systèmes embarqués : Les "Systems on Chip" (SoC), définis comme étant l'intégration sur une même puce d'une plate-forme complexe, potentiellement composée de plusieurs processeurs, de bancs mémoires, de circuits spécialisés (ASIC) ou reconfigurables (FPGA),... Des contrôleurs embarqués d'autrefois aux systèmes sur puce multiprocesseurs d'aujourd'hui (MPSoC), il existe un saut de complexité que les outils actuels d'aide à la conception n'arrivent pas à franchir. Indépendemment des considérations scientifiques que nous aborderons par la suite, le marché des systèmes embarqués est régi par une loi économique relativement simple : la conception d'un système a un coût non négligeable, et un des seuls moyens d'obtenir un retour sur l'investissement consenti est d'être le premier à commercialiser LE produit qui fera la différence, l'absence de concurrence permettant de justifier des prix plus élevés et de garantir la rentabilité [6]. Or, si les progrès technologiques offrent théoriquement la puissance de calcul nécessaire aux prochaines générations d'applications embarquées, les concepteurs ne disposent pas d'outils leur permettant d'exploiter à un coût raisonnable les transistors qui sont potentiellement mis à leur disposition (fig.1.1) : Le coût de la conception croît de façon exponentielle avec la complexité du système, d'autant plus que les contraintes de temps de mise sur le marché sont impondérables (il est vital d'être "le premier"). Cette situation paradoxale, où l'on se donne des moyens que l'on n'est pas en mesure d'exploiter, conduit les analystes les plus pessimistes à penser que l'économie bridera l'industrie des semi-conducteurs bien avant que les limites technologiques de la loi de Moore n'aient été atteintes [30].

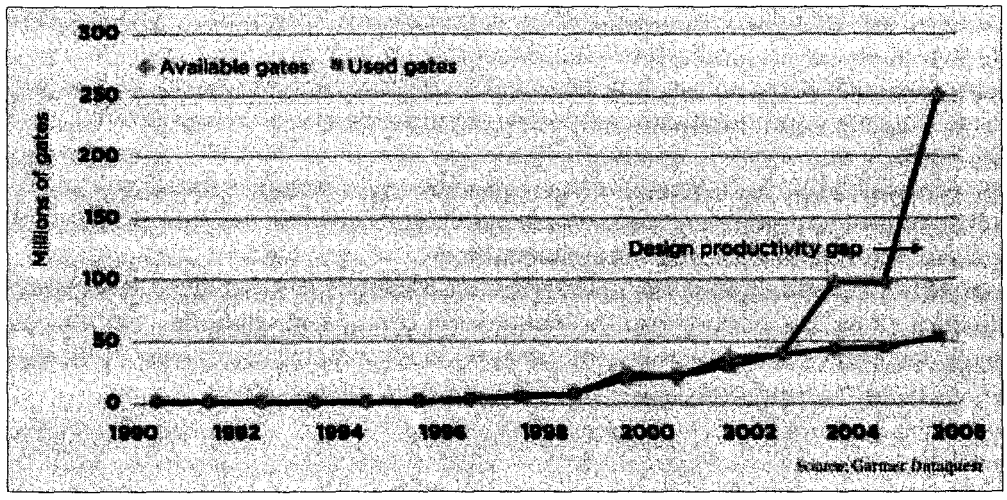


FIG. 1.1 – Portes disponibles vs Portes utilisées

Pour résoudre le problème qui leur est posé, les concepteurs doivent par ailleurs tenir compte d'une autre donnée, tout aussi préoccupante. D'un côté, la capacité de stockage d'énergie des batteries utilisées pour alimenter les systèmes embarqués croît très lentement. De l'autre côté, la complexité croissante des applications requiert toujours plus de capacités

de calcul, de stockage, et de communication. Par respect de contraintes d'embarquabilité évidentes, la taille et le poids des batteries ne peuvent être augmentés arbitrairement pour satisfaire les besoin d'énergie et d'autonomie du système. Il convient donc de concevoir des systèmes certes performants, mais économes en énergie, le tout dans un temps limité par les contraintes du marché.

2 Contribution

C'est dans ce contexte, où la conception des systèmes sur puce devient un problème de plus en plus délicat à gérer, que s'inscrivent les travaux présentés dans cette thèse. Ces travaux s'intègrent dans le développement d'un flot de conception en Y [1] nommé Gaspard, ayant pour ambition la réduction du temps de cycle de développement.

L'objectif de cette thèse est de fournir un support pour la modélisation à haut niveau d'abstraction (fonctionnel) des systèmes sur puce à hautes performances, avec une contribution à deux niveaux :

- la définition d'une syntaxe abstraite par des métamodèles exprimés en MOF (servant d'infrastructure pour la mise en œuvre du flot de conception),
- la définition d'une syntaxe concrète par un profil pour UML 2 (que l'on nommera par la suite le profil Gaspard).

Les lignes directrices ayant orienté ces travaux sont principalement :

- **la définition d'une méthodologie orientée composant,** afin d'offrir un support pour la définition et la réutilisation d'IPs logicielles ou matérielles.
- **la définition de mécanismes communs de factorisation,** pour l'expression sous une forme compacte du parallélisme potentiel des applications, du parallélisme disponible des architectures et de la topologie des connections entre composants, et du placement régulier des applications sur les architectures matérielles.
- **une séparation claire entre application et architecture matérielle,** afin de faciliter et d'encourager la réutilisation des modèles, mais aussi de garantir leur lisibilité.
- **la définition d'un modèle de description d'architectures matérielles,** permettant de dimensionner les ressources matérielles supportant l'exécution d'une application.
- **la définition d'un modèle d'association,** permettant de donner des directives de placement d'une application sur une architecture.
- **la définition de mécanismes de caractérisation,** visant à identifier des caractéristiques relatives à l'exécution d'une application (ou d'un composant applicatif) sur une architecture matérielle (ou un composant matériel), de façon à pouvoir orienter un placement manuel ou automatique de l'application sur l'architecture.

3 Plan

Dans le chapitre 2 (Positionnement), après avoir brièvement présenté les tendances méthodologiques et technologiques relatives à la conception des systèmes sur puce, nous proposerons une description du flot de conception Gaspard, en mettant en avant son originalité : l'utilisation de MDA pour sa mise en œuvre, l'apport du parallèlisme de données

[1] Spécification de l'application indépendemment d'une architecture matérielle, spécification d'une architecture matérielle indépendemment d'une application, et placement de l'application sur cette architecture

pour la conception des SoCs, et le dimensionnement à haut niveau d'abstraction des architectures matérielles. A ce titre, nous nous attarderons sur le modèle de programmation à parallèlisme de données, et donnerons quelques exemples représentatifs de langages proposés dans ce domaine. En particulier, nous décrirons le langage ArrayOL, qui servira de base à la définition de nos mécanismes communs de factorisation. Nous proposerons enfin un tour d'horizon des "outils" (langages, modèles, ou méthodologies) pour la modélisation des architectures matérielles, en essayant de montrer l'intérêt de s'abstraire (dans un premier temps) des détails d'implémentation de l'architecture.

Dans le chapitre 3 (Contribution des standards OMG à la définition du profil Gaspard), nous proposerons une étude détaillée des différents standards (adoptés ou en cours d'adoption) plus ou moins liés au domaine de l'embarqué. Nous allons voir que les travaux menés dans la communauté UML autour de la modélisation et de l'analyse des systèmes embarqués offrent globalement une réponse satisfaisante pour la majorité des objectifs que nous avons énoncé, à l'exception de l'objectif de définition de mécanismes de factorisation communs. Pour ce point précis, nous nous tournerons, comme nous l'avons déjà évoqué, du côté du langage ArrayOL.

Dans le chapitre 4 (Le profil Gaspard), nous donnerons une spécification du profil Gaspard, en réutilisant les éléments que nous aurons jugé comme pertinents dans les chapitres 2 et 3. Une vue simplifiée du contenu des métamodèles sera proposée sous la forme d'une vue domaine, mettant en avant les idées clé, tout en s'abstrayant des détails de mise en oeuvre en MOF.

Dans le chapitre 5 (Etude de cas), nous illustrerons l'utilisation du profil sur une étude de cas détaillée, en appliquant les différents mécanismes identifiés dans le profil, et en mettant en avant leur intérêt pour les différents aspects de modélisation du Y de Gaspard : Application, architecture matérielle, et placement.

Dans le chapitre 6 (Conclusion et perspectives), nous dresserons un bilan de notre proposition, et évoquerons des perspectives quant à l'évolution du profil et des métamodèles.

CHAPITRE 2

Positionnement

1 **Tendances méthodologiques** 17
2 **Tendances technologiques** 19
3 **Le flot de conception Gaspard** 21
4 **Programmation à parallèlisme de données** 25
4.1 Fortran 27
4.2 High Performance Fortran 27
4.3 ArrayOL 29
4.4 Bilan 34
5 **Modélisation des architectures matérielles** 36
5.1 HDLs 37
5.2 Langages niveau système 37
5.3 ADLs 38
5.4 Méthodes orientées UML 38
5.5 Autres outils et langages 39
5.6 Bilan 40

Dans la chapitre d'introduction, nous avons montré l'enjeu économique auquel étaient confrontés les concepteurs de systèmes sur puce. Cette enjeu explique en grande partie l'effervescence de la communauté scientifique, et la multitude des travaux de recherche menés autour de la problématique de la conception des systèmes sur puce. De ces différents travaux émergent clairement des tendances méthodologiques et technologiques. Après avoir brièvement présenté ces tendances, nous proposons une description du flot de conception Gaspard, en montrant en quoi il peut proposer une réponse interessante à cet enjeu, avec différentes propositions originales : l'utilisation de MDA pour sa mise en œuvre, l'apport du parallèlisme de données pour la conception des SoCs, et le dimensionnement à haut niveau d'abstraction des architectures matérielles. A ce titre, nous nous attarderons sur le modèle de programmation à parallèlisme de données, et donnerons quelques exemples représentatifs de langages proposés dans ce domaine. En particulier, nous décrirons le langage ArrayOL, qui servira de base à la définition de nos mécanismes communs de factorisation. Nous proposerons enfin un tour d'horizon des "outils" (langages, modèles, ou méthodologies) pour la modélisation des architectures matérielles, en essayant de montrer l'intérêt de s'abstraire (dans un premier temps) des détails d'implémentation de l'architecture.

1 Tendances méthodologiques

Pour réduire les temps de cycle de développement, deux aspects orthogonaux se révèlent indispensables : la montée dans les niveaux d'abstraction pour la spécification du système, et la réutilisation de blocs prédéfinis communément appelés IPs (Intellectual Properties). Alors que le premier aspect encourage à s'abstraire des détails d'implémentation, le second aspect encourage à capitaliser des développements qui ont abouti à la validation d'un bloc.

Dans un flot de conception traditionnel tel qu'illustré en fig.2.1 [60], les architectes systèmes travaillent en collaboration avec les développeurs pour définir les fonctionnalités attendues du système et dimensionner le support d'exécution. Sur la base d'une modélisation à "haut niveau" d'abstraction (à l'aide de langage de spécification niveau système tels que SystemC [54] ou SpecC [22]) et d'un ensemble de plans de test, une implantation au niveau transfert de registres (RTL, Register Transfer Level) est définie, servant elle-même de base pour la réalisation au niveau physique. Dans ce type de méthodologie, le niveau RTL est typiquement le pilier supportant le processus d'implantation du système, simplement parce que les outils d'analyse, d'estimation, ou encore de réalisation physique (tels que des outils de synthèse vers le niveau portes logiques) sont relativement matures. Dans une description au niveau RTL, la structure interne des IPs reflète précisément les registres et la logique combinatoire de l'architecture cible. Par ailleurs, les communications sont décrites en détail, tant au niveau des protocoles utilisés qu'au niveau du temps (précis au cycle près) : Le comportement de chaque module correspond précisément au comportement du module physique. Un tel niveau de précision nécessite par définition une somme considérable d'informations, source d'erreurs, et responsable d'un grand nombre d'itérations dans le cycle de développement d'un système (analyse, vérification et débogage). Il y a quelques années, l'utilisation de telles méthodologies pouvait encore être considérée comme raisonnable. Cependant, la complexité croissante des systèmes et la nécessité de maintenir des délais de développement compatibles avec les contraintes de temps de mise sur le marché rendent ces méthodologies de plus en plus inadaptées. Ce phénomène est communément nommé le "productivity gap" (déficit de productivité).

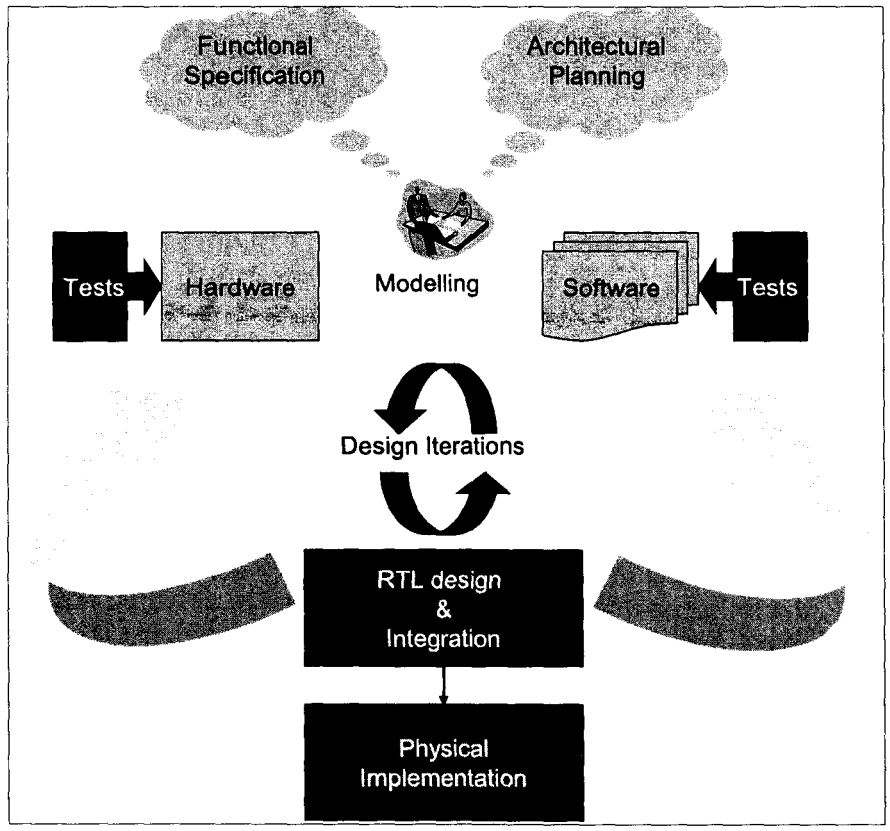


FIG. 2.1 - Flot traditionnel de conception de SoC

La nécessité de s'abstraire des détails du niveau RTL pour accélérer le processus de développement étant devenue une évidence, un nouveau paradigme de modélisation est apparu : la modélisation au niveau transactionnel (TLM, Transaction Level Modeling). Le TLM consiste à décrire les systèmes en respectant le niveau de précision relatifs à différents niveaux d'abstraction appelés niveaux TLM, dans lesquels les communications entre modules sont typiquement représentées par des appels de fonction (ex : `read(char* buf, int addr, int len)`). Fig.2.2 [21] propose un exemple de structuration des niveaux d'abstraction TLM. Il existe d'autres classifications, comme celle proposée dans [60]. Sans rentrer dans les détails de chaque niveau d'abstraction, l'idée globale est que chaque modèle pour un niveau donné permet la spécification et la vérification d'un ensemble de propriétés. Chaque niveau constitue un raffinement du niveau précédent, l'objectif étant de minimiser le nombre et la taille (distance entre deux niveaux) des "boucles" dans le processus de développement.

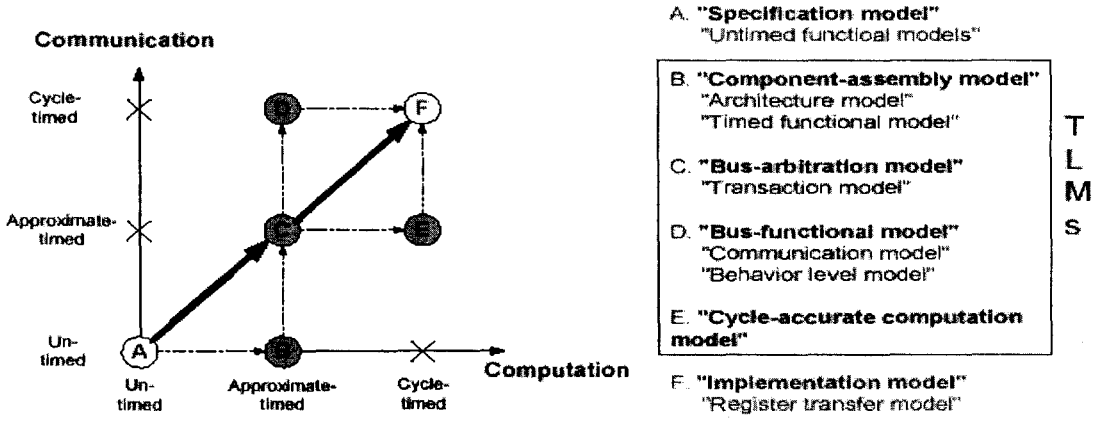


FIG. 2.2 – Structuration de la conception sous forme de niveaux d'abstraction

Récemment, un certain nombre d'actions ont été lancées dans le but d'encourager le développement du TLM : définition de lignes directrices pour la modélisation au niveau TLM, d'API TLM standard, d'IP spécifiées au niveau TLM, et mise en oeuvre d'un support logiciel. SystemC est l'un des premiers langages de description au niveau système à intégrer un support pour les différents niveaux TLM. Un effort de standardisation, initié par les principaux vendeurs d'outils pour l'EDA (Electronic Design Automation) et d'IP, est en effet en cours au niveau de l'OSCI [54], l'organisme en charge de la standardisation de SystemC.

2 Tendances technologiques

Pour satisfaire les besoins de puissance de calcul toujours croissant des applications embarquées, les concepteurs de SoC ont eu tendance à intégrer plus de ressources de calcul (programmables, configurables, ou spécialisées). Indépendemment du contexte des SoCs, le parallélisme constitue en effet un moyen naturel pour améliorer les performances d'un système. Sur les premiers SoCs, l'interconnexion des différents composants était principalement réalisée à base de bus et de connexions point à point. Les bus présentent l'intérêt de fournir des performances intéressantes, à un coût raisonnable, en offrant la possibilité

d'être partagés par différents acteurs. Néanmoins, la multiplication des coeurs de processeur sur une même puce a rapidement montré les limites de scalabilité de cette technologie, un bus pouvant être considéré comme efficace dans la limite de 3 à 10 composants interconnectés [31]. Au delà, le comportement du bus devient difficilement prévisible, et son utilisation se révèle inappropriée pour des domaines d'applications critiques, avec par exemple des contraintes de type temps réel dur. De plus, des problèmes liés à la physique des technologies submicron sont apparus, tels que des surcoûts en terme de consommation d'énergie et des phénomènes de bruit dont l'importance est directement liée à la longueur des fils. Les bus supportent ainsi difficilement la réutilisation d'un grand nombre d'IPs, rien ne garantissant que des IPs validées individuellement fourniront globalement un comportement cohérent, compte tenu des perturbations. Par ailleurs, ces problèmes physiques rendent de plus en plus complexes la synchronisation globale des différentes parties de la puce. Le signal d'horloge nécessitera bientôt plusieurs cycles d'horloge pour traverser la puce avec des problèmes de décalage incontrôlables, et l'arbre de distribution de l'horloge constitue déjà un facteur d'accroissement du coût et de la consommation d'énergie. L'hypothèse de l'utilisation d'un seul domaine d'horloge pour la synchronisation de toute la puce devient de moins en moins réaliste. Dès 1999, des travaux visant à trouver une alternative raisonnable aux bus et à résoudre le problème de synchronisation globale ont contribué à l'émergence d'un nouveau paradigme : Les "Network on Chips" (NoC) [2].

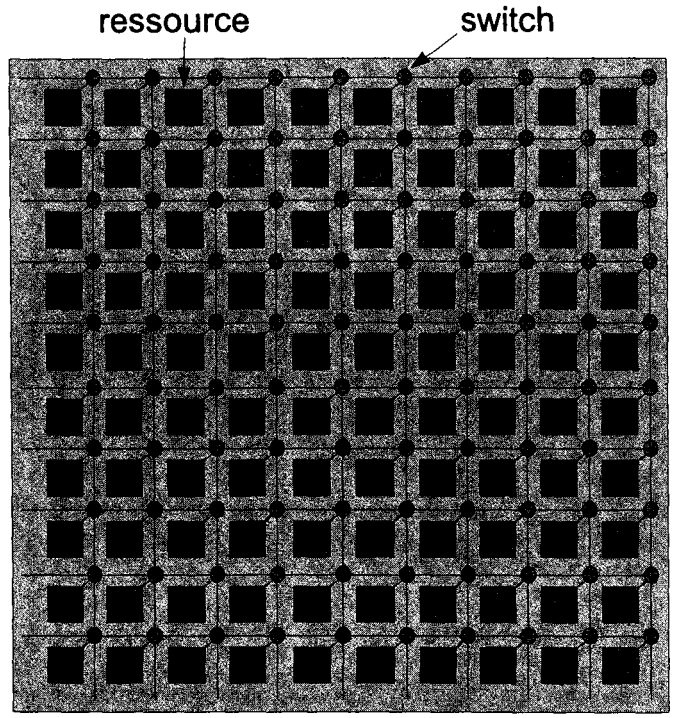


FIG. 2.3 – Exemple de NoC : Topologie en grille

La démarche relativement pragmatique sous-jacente au concept de NoC revient à considérer un SoC comme étant un système distribué, à l'échelle d'une puce. Le NoC

joue le rôle d'infrastructure pour le support des communications entre les différentes ressources, chaque ressource étant potentiellement cadencée par sa propre horloge. Le concept de NoC est ainsi intimement lié au paradigme GALS (Globally Asynchronous Locally Synchronous [40]). Un NoC est défini par un ensemble d'éléments d'interconnexion simples (généralement des switchs et des routeurs), interconnectés de façon régulière. A chaque élément d'interconnexion du réseau peut être attaché une ressource (fig.2.3). Par nature, un NoC propose donc de bonnes propriétés de scalabilité, et offre un support satisfaisant pour la réutilisation d'IPs. La topologie régulière des connexions rend par ailleurs prévisible la performance des communications, ainsi que les phénomènes électriques et physiques se produisant au niveau du réseau [32].

3 Le flot de conception Gaspard

Le flot de conception Gaspard [34], que nous proposons dans le contexte du projet INRIA DaRT [28], suit globalement les tendances que nous venons d'évoquer, avec quelques particularités. L'une des originalités de notre approche est d'utiliser une méthodologie orientée modèle pour la mise en oeuvre d'un flot de conception en Y (fig.2.4), partiellement inspiré du "Y chart" [21]. La méthodologie s'articule autour de trois niveaux d'abstractions : fonctionnel, transactionnel, et registre. Les concepts et la sémantique de chacun de ces niveaux sont capturés dans des métamodèles indépendants des plateformes d'exécution ou de simulation.

Gaspard se situe ainsi à la frontière de deux domaines de recherche : la conception des systèmes sur puce, et l'ingénierie dirigée par les modèles. Si la notion de "flot en Y" existe dans ces deux domaines, avec une articulation autour de trois points, il est important de noter que le "Y" de Gaspard correspond à un flot tel qu'on le définit généralement dans le domaine de la conception de systèmes sur puce (ou plus généralement dans le domaine de la conception de systèmes embarqués incluant des parties logicielles et matérielles) :

- Application : Spécification de la fonctionnalité attendue d'un système,
- Architecture matérielle : Spécification de l'architecture matérielle qui sera utilisée pour supporter l'exécution d'une application et réaliser ses fonctionnalités attendues,
- Association : Spécification du placement d'une application sur une architecture matérielle donnée.

Dans l'approche MDA, les trois points constituants une approche en Y (fig.2.5) font référence à des notions sensiblement différentes :

- Modèle indépendant des plateformes (PIM, Platform Independent Model) : Un modèle capturant de façon "abstraite" les éléments constituant un système. Le terme "abstrait" signifie simplement "indépendemment de toute technologies d'implémentation",
- Modèle de définition d'une plateforme (PDM, Platform Definition Model) : Un modèle définissant les éléments constituant une technologie d'implémentation,
- Modèle spécifique à une plateforme (PSM, Platform Specific Model) : Sans rentrer dans les détails méthodologiques, un PSM est le "produit" d'un PIM et d'un PDM. A partir d'une description indépendente de toute technologie d'implémentation et de la description d'une technologie d'implémentation, on produit une description spécifique à une technologie.

Il est par ailleurs important de noter que le terme "plateforme", qui est utilisé dans le domaine des systèmes embarqués pour désigner le support d'exécution (matériel, avec ou

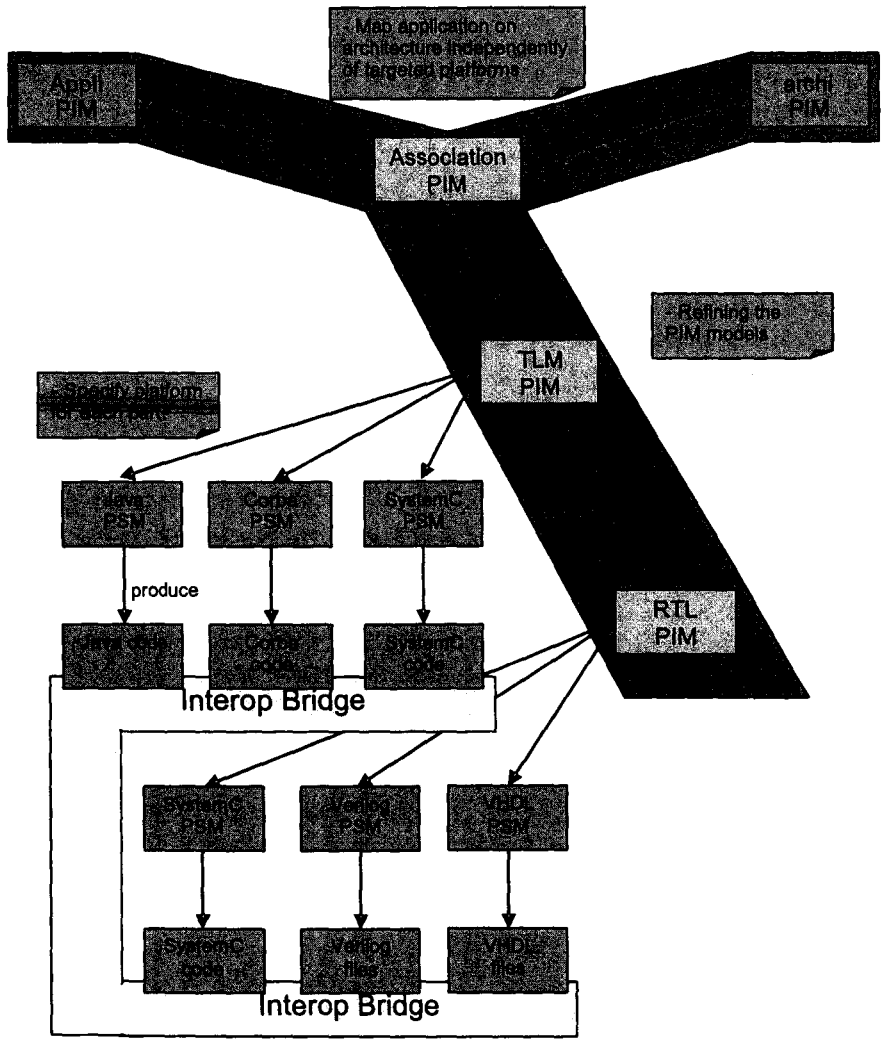


FIG. 2.4 – Métamodèles pour le flot de conception Gaspard

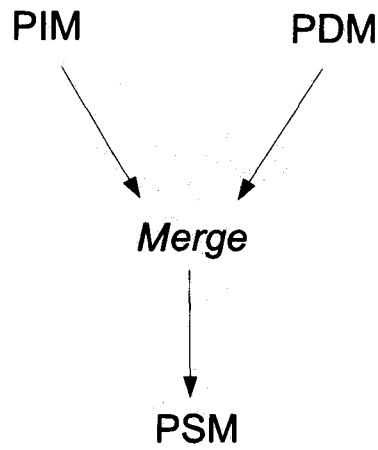


FIG. 2.5 – Flot en Y dans le MDA

sans système d'exploitation) d'une application, revêt ici une signification bien différente. Il s'agit, comme nous l'avons déjà évoqué, d'une technologie d'implémentation. On pourrait par exemple imaginer un modèle de composants indépendant, projeté vers un modèle de composants spécifique, tel que CCM ou EJB.

Ainsi, Gaspard ne partage pas la définition de flot en Y proposée dans le cadre du MDA. Le MDA intervient par contre dans la mise en œuvre de la méthodologie proposée par Gaspard. Cette méthodologie s'articule autour de trois niveaux d'abstractions : fonctionnel, transactionnel, et registre. Les concepts et la sémantique de chacun de ces niveaux sont capturés dans des métamodèles indépendants des plates-formes d'exécution ou de simulation. La motivation de l'utilisation de l'approche MDA [43] comme support de Gaspard est d'aider à gérer la complexité du flot, en mettant en œuvre différents types de raffinements automatiques ou semi-automatiques par des techniques de transformations de modèles, avec des raffinements PIM-PIM (c'est à dire entre niveaux d'abstractions) ou PIM-PSM (c'est à dire d'un modèle correspondant à un niveau d'abstraction vers un modèle d'exécution ou de simulation). Elle permet par ailleurs d'encourager la réutilisation des modèles. En effet, si les technologies évoluent rapidement, les modèles (spécifiés à un niveau PIM) et leur sémantique restent stables.

La contribution de DaRT au domaine, via la proposition du flot de conception Gaspard, s'articule autour de trois axes : la co-modélisation, les techniques d'optimisation, et la simulation, ayant pour objectif commun la réduction du temps de cycle de développement. Chacun des trois axes propose des solutions fortement influencées par l'aspect régulier et parallèle des systèmes visés.

► **Co-modélisation** Le terme co-modélisation fait référence à la modélisation conjointe des parties logicielles et matérielles d'un système. L'objectif est de proposer un environnement permettant la modélisation à haut niveau d'abstraction des systèmes à hautes performances, tant au niveau de l'application (parallélisme potentiel) qu'au niveau de l'architecture matérielle (parallélisme disponible). Nous reviendrons en détail sur ce point dans la section Contribution de ce chapitre. C'est en effet dans l'axe Co-modélisation que s'inscrit cette thèse.

► **Techniques d'optimisation** Les techniques visées sont principalement des transformations de constructions data parallèles. Ces transformations sont utilisés pour optimiser le placement et l'ordonnancement d'une application sur une architecture, et sont fortement inspirés des techniques utilisées dans la communauté du calcul haute performance. Par ailleurs, des heuristiques visant à minimiser la consommation d'énergie sont développées. Cet objectif implique la définition de techniques d'optimisation multi-critères.

► **Simulation** Dans cet axe de recherche, un modèle d'interopérabilité (identifié en fig.2.4 par "Interrop Bridge") est développé. L'objectif est de permettre la simulation d'un système composé d'IP spécifiées à différents niveaux d'abstraction, dans différents langages, éventuellement de façon distribuée. La simulation permet aussi bien de vérifier la fonctionnalité du système, que l'adéquation du placement et de l'ordonnancement (délais des communications, équilibrage de charge, allocations mémoire,...).

L'autre originalité de notre approche (suggérée dans les précédents paragraphes) est de proposer des solutions fortement orientées vers les systèmes multiprocesseurs réguliers,

éventuellement mis en oeuvre via des réseaux, pour l'exécution d'applications régulières offrant potentiellement beaucoup de parallèlisme de données. Cet aspect correspond à l'objectif (ennoncé dans l'introduction de cette thèse) de définir un mécanisme commun pour l'expression de cette régularité, du côté de l'application comme du côté de l'architecture. Dans les sections suivantes, nous allons dans un premier temps donner une description du modèle de programmation à parallèlisme de données, en mettant (notamment) en avant le processus qui a conduit à son élaboration : la nécessité de s'abstraire de l'architecture de la machine cible. Puis, nous proposerons un tour d'horizon des outils pour la modélisation des architectures matérielles, en rappelant la nécéssité de s'abstraire des détails d'implémentation de l'architecture.

4 Programmation à parallèlisme de données

Dans le domaine de la programmation parallèle, l'apparition du paradigme de "parallèlisme de données" résulte d'un processus similaire à celui qui a conduit les langages de programmation séquentiels à s'abstraire de plus en plus des modèles d'exécution [4]. En considérant comme modèle d'exécution séquentiel le modèle de Von Neumann (le contenu de différents emplacements mémoire est séquentiellement chargé dans les registres du processeur, consommé par différentes opérations, et de nouveau stocké dans la mémoire), on constate que les premiers langages "évolués", tels que Fortran, proposaient des constructions assez proches du modèle d'exécution (tel que l'instruction `GOTO`). Par la suite, des langages tels que Pascal ont introduits des moyens de plus haut niveau, pour permettre une programmation mieux structurée (avec les boucles `while`, `repeat` ou `for`). De la même manière, un détachement encore plus sensible vis à vis du modèle d'exécution est apparu, avec des paradigmes tels que la programmation orientée objet (C++, Effeil), logique (Prolog), ou encore fonctionnelle (Lisp, ML). L'abstraction par rapport au modèle d'exécution est motivée par la volonté de faire apparaître de plus en plus de sémantique dans les programmes. En prenant l'exemple extrême du `GOTO` de Fortran (en ayant en tête qu'un `GOTO` peut être réalisé de n'importe où vers n'importe où à l'intérieur d'un programme), seule une analyse sémantique peut permettre de découvrir quelle est la structure du flot de contrôle, ou encore quelles sont les limites du `GOTO`. A l'inverse, les mécanismes de programmation structurée offerts par la syntaxe de `Pascal` permettent d'exprimer de façon explicite la structure du flot de contrôle, offrant un gain sémantique évident (bien que ce gain soit fait au détriment du pouvoir d'expression du `GOTO`).

Un processus similaire s'est appliqué au domaine de la programmation parallèle, bien qu'il soit plus récent et forcément moins mature. Des années 60 aux années 90, la plupart des efforts de la communauté s'est concentrée sur la façon de proposer des architectures matérielles de plus en plus performantes, ce qui a principalement conduit à la définition d'une multitude de modèles d'exécution. La célèbre classification de Flynn [20] propose de regrouper ces modèles selon quatre catégories. Ces catégories diffèrent selon le nombre (unique ou multiple) de flots d'instructions et de données :

- SISD (Single Instruction Single Data) : PCs, stations de travail, ...
- MISD (Multiple Instruction Single Data) : processeurs vectoriels, architectures pipelinées, ...
- SIMD (Single Instruction Multiple Data) : grilles de processeurs, communément nommées "architectures massivement parallèles"

- MIMD (Multiple Instruction Multiple Data) : multi-processeurs, transputers,...

En comparaison, les efforts consentis pour proposer des langages adaptés aux domaines d'application visés par ce type d'architectures (typiquement des applications avec traitement numérique intensif) furent minimes. La tendance globalement suivie était de proposer des langages directement dérivés des modèles d'exécution ayant conduit à la conception d'une machine particulière. Au sein d'une même famille de machines, il a ainsi été de donner de constater que les langages proposés étaient si liés au modèle d'exécution d'une machine que l'évolution de cette machine entraînait une refonte ou un abandon du langage. Un exemple typique est celui de la Connection Machine CM-2 [7], et des langages C/ParIS et *Lisp. Le développement de la CM-5 [8] a conduit à l'abandon pur et simple de ces deux langages.

Ce constat étant établi, un certain nombre de travaux a été entrepris pour proposer des langages de programmation indépendants des modèles d'exécution des machines parallèles. Une des solutions proposées a été d'importer des modèles abstraits issus du monde de la programmation séquentielle, de les enrichir avec des structures de contrôle pour le parallélisme, et de définir une projection de ces modèles vers un maximum de modèles d'exécution parallèle. Ce processus a ainsi été appliqué à la programmation logique (Parallel Prolog), à la programmation fonctionnelle (avec les différentes variantes parallèles et/ou concurrentes de Lisp, ML, ou encore Haskell), et à la programmation orientée objet (avec des extensions parallèles ou distribuées de langages orientés objet, tels que CC++, ou encore pC++). Si la compilation efficace de ce type de modèles abstraits vers une machine séquentielle est un exercice délicat, elle s'avère être un challenge encore plus relevé lorsque l'on considère des machines parallèles, d'autant plus que la nécéssité d'obtenir de bonnes performance est la motivation principale pour l'utilisation de ce type de machine. Cette limitation a conduit à une autre voie de recherche, similaire, mais appliquée à des modèles de programmation moins abstraits, tels que le modèle de programmation de Pascal. Ces travaux ont particulièrement conduit à la définition du modèle CSP [24] (Communicating Sequential Processes), abstraction du modèle d'exécution MIMD. Ce modèle d'exécution a notemment servi de base à la définition du modèle de tâches de ADA, ou encore à la définition de librairies toujours fréquemment utilisées sur les machines MIMD, telles que PVM ou encore MPI. La principale limitation est que, étant finalement encore trop proche du modèle d'exécution, la manipulation explicite de structures de contrôle parallèles est relativement complexe et source d'erreurs, en particuliers lorsque le nombre de processus à considérer est important. Par ailleurs, la "proximité" par rapport au modèle d'exécution MIMD la rend difficilement projetable vers d'autres types d'architectures (et typiquement les architecture SIMD).

Cet état de fait a conduit à l'apparition d'un nouveau paradigme, communément désigné sous le nom de "programmation data parallèle", où l'expression du contrôle demeure séquentielle (donc relativement simple), alors que l'accès aux données est réalisé en parallèle. L'idée de base de la programmation dans le modèle du parallèlisme de données est de faire en sorte qu'un programme parallèle ait globalement l'apparence d'un programme séquentiel : Comme dans n'importe quel programme séquentiel, il n'y a qu'un seul thread de contrôle. Contrairement au parallèlisme de contrôle (le modèle CSP), il n'existe pas d'instructions de synchronisation. De la même manière, il n'existe pas de notion de variable privée, uniquement visible par le processeur qui la possède : Il n'y a qu'un seul espace de nommage, et toutes les variables déclarées dans un programme sont accessibles par tous les processeurs. Dans la plupart des langages offrant un support pour l'expression du parallèlisme de données, les données manipulées sont principalement des tableaux multidimensionnels. Ces langages offrent simplement des mécanismes permettant d'exprimer de façon compacte

qu'une répétition d'une seule et même tâche est appliquée sur différents motifs contenus dans ces tableaux, chaque répétition d'une tâche étant indépendante des autres répétitions.

Etant donné le nombre de contributeurs, qu'il s'agisse de grands constructeurs ou d'équipes de recherche plus modestes, il est difficile de fournir une liste exhaustive de tous les langages data parallèles existant ou ayant existé (Des classifications de ces langages existent, telles que celle proposée dans [38]). Nous retiendrons ici quelques langages représentatifs, qu'ils soient populaires, tels que Fortran et ses différentes évolutions jusque HPF, ou plus confidentiels, tels que ArrayOL.

4.1 Fortran

La première version du langage Fortran remonte à 1957. Fortran (pour FORmula TRANslator) a pour origine l'objectif de définir un langage permettant la traduction de formules scientifiques ou algébriques dans une forme utilisable par un ordinateur, à une époque où les ordinateurs ne peuvent être programmés (dans le meilleur des cas) qu'en assembleur [63]. De 1957 à 1978, un certain nombre d'évolution du langage, visant avant tout à accroître sa portabilité par la définition de différents standards, ont été proposées. Mais ce n'est qu'en 1992, avec l'apparition de la norme Fortran 90 [29], qu'apparaît dans Fortran la possibilité d'exprimer du parallèlisme de données. D'une certaine façon, Fortran 77 permettait également l'expression du parallèlisme de données, mais nécéssitait l'utilisation de bibliothèques spécifiques, peu confortables pour des non spécialistes.

Fortran 90 prend en compte l'expression du parallèlisme dans ses structures syntaxiques, en permettant d'accéder aux tableaux en tant qu'entités et d'effectuer des opérations globales sur ces tableaux, sans avoir à expliciter un par un (par le biais d'une boucle) les accès aux différents éléments du tableau. Ainsi, deux tableaux `A` et `B` de même forme et de même taille peuvent être additionnés terme à terme par la syntaxe : `A + B`. C'est au compilateur que revient la charge de déterminer l'implantation efficace de cette expression sur l'architecture cible.

Fortran 95, dernière version normalisée de Fortran, introduit des extensions permettant de palier le manque d'expressivité au niveau de la description des opérations entre tableaux. La principale innovation de Fortran 95 est l'ajout de l'instruction `forall`. Dans Fortran 90, si l'on veut affecter à un tableau bi-dimensionnel `A`, de taille `100×50`, une partie d'un tableau `B` de taille `100×100`, on peut utiliser une expression de la forme (exemple tiré de [36]) :

$$A=B(:,2:51) \tag{2.1}$$

Par contre, il est impossible d'affecter directement la transposée d'un bloc, ou une valeur dépendant de l'indice. Le `forall` offre cette possibilité, en permettant d'exprimer l'affectation avec des indices nommés :

$$\text{forall } (i=1:100,j=1:50)\ A(i,j)=B(j,i)+i+j \tag{2.2}$$

4.2 High Performance Fortran

High Performance Fortran (HPF) reprend la base posée par les spécifications du langage Fortran 95, et lui ajoute des directives de parallèlisation et de placement des données. La syntaxe proposée pour l'expression de ces directives, sous forme de commentaires, permet la compilation d'un source HPF par un compilateur Fortran classique. Il est important de

noter que HPF ne définit que des directives, c'est à dire une forme de conseils adressés au compilateur pour générer un code plus efficace. La liberté est laissée au compilateur de les exploiter s'il en est capable, ou s'il considère qu'il est en mesure de fournir une solution plus efficace. Sans remettre en cause l'intérêt du langage, c'est cette particularité qui fera qu'HPF n'aura jamais su s'imposer réellement chez les programmeurs de machines parallèles.

4.2.1 Directive de parallélisation

HPF introduit la directive `INDEPENDENT`, qui permet de spécifier explicitement au compilateur que toutes les itérations d'une boucle `do...enddo` ou d'un `forall` sont indépendantes, et qu'il peut donc parallèliser l'opération sans risques. Il est important de noter que si le programmeur assure à tort au compilateur qu'il n'existe pas de dépendences entre les différentes itérations, le programme sera faux, voir non déterministe.

4.2.2 Directives de placement des données

Les directives de placement introduites dans HPF (qui constituent sa réelle valeur ajoutée par rapport à Fortran 95) sont motivées par l'importance, dans le modèle de programmation à parallèlisme de données, de la répartition des données à travers les différents nœuds. Définir ce placement automatiquement lors de la compilation est un problème complexe, et pouvoir exploiter l'expertise du programmeur pour orienter le placement des données est un avantage considérable. Le placement des données influe à deux niveaux : la répartition de la charge des processeurs, et le volume de communication nécessaire à la récupération des opérandes. Globalement, les données doivent être organisées de façon à ce qu'un maximum d'opérandes soient présents sur le noeud où vont être réalisés les calculs. HPF introduit deux types de directives de placement des données :

▶ **Les directives d'alignement** Les directives d'alignement permettent de spécifier le placement des tableaux les uns par rapport aux autres, et donc de réduire les communications implicites. Pour spécifier la façon dont les données doivent être placées les unes par rapport aux autres, HPF introduit la notion de `template`. Un template est défini comme un tableau sans type, c'est à dire simplement par une forme (nombre de dimensions) et une taille (pour chacune de ces dimensions). Les templates servent de référentiel pour l'alignement des données. A noter que tout tableau déclaré dans un programme HPF est implicitement aligné sur un template de même géométrie.

Définir un alignement revient à déterminer comment les données contenues dans un ou plusieurs tableaux sont alignées par rapport à un template. Ces directives doivent être prises en compte pour garantir que toutes les données alignées sur le même élément d'un template soient présentes sur le même processeur physique.

Dans l'exemple suivant (toujours tiré de [36]), la directive `align` spécifie que le tableau `A` doit être aligné sur le tableau `B` (ou plus précisemment sur le template implicite associé au tableau `B`) de telle façon que pour tout `i` et `j`, `A(i,j)` soit aligné avec `B(*,j,i*2-1)` :

```
   dimension(10,10), real : : A
dimension(10,20,20), real : : B
!HPF align A(i,j) with B(*,j,i*2-1),
```

l'étoile (*) indiquant que chaque élément de `A` doit être répliqué sur la dimension correspondante de `B`.

► **Les directives de distribution** Les directives de distribution permettent de spécifier la répartition des données, et donc un équilibrage de la charge des différents processeurs. Pour permettre de spécifier cette distribution, HPF propose une structure abstraite, nommée `processors`, qui permet de définir une machine virtuelle cible. Sa structure est composée d'une grille multidimensionnelle et homogène de processeurs virtuels. A noter que rien ne définit des liens de communication entre ces processeurs. Les directives de placement que nous allons décrire par la suite ne permettent donc d'optimiser que le nombre d'échange de données, sans savoir si une communication avec un voisin directe est plus coûteuse qu'une communication à l'autre extrémité de la grille. Par ailleurs, le placement des processeurs abstraits sur les processeurs physiques est à la charge du compilateur.

Une distribution correspond à une répartition régulière des éléments d'un tableau ou d'un template sur les différents processeurs de la machine virtuelle. Ces répartitions sont faites pour chaque dimension du tableau ou template source, sur chaque dimension de la structure processors. HPF propose trois types de distribution :

- `block` : Le tableau ou template source est découpé en autant de motifs (ou blocs) de même taille qu'il y a de processeurs virtuels dans la structure processors. Chaque motif est ensuite alloué sur le processeur virtuel correspondant.
- `cyclic` : Chaque élément de la source est placé sur l'élément de la cible de même rang. Une fois le placement réalisé sur l'élément de dernier rang de la cible, les éléments de la source continuent d'être parcourus et le placement reprend sur l'élément de premier rang de la cible.
- `cyclic(k)` : Le processus de placement est le même que pour la distribution cyclique, sauf que l'on considère désormais des blocs de k élements dans le tableau ou template source.

4.3 ArrayOL

ArrayOL (Array Oriented Language) [14] propose une approche originale pour la modélisation d'applications de traitement de signal intensif : l'assignation unique et la modélisation des applications par l'unique expression des dépendances de données. ArrayOL se base sur le constat simple que toute optimisation réalisée par un compilateur ou par un quelconque outil d'optimisation nécessite une phase complexe d'analyse des dépendances de données. Par définition, ArrayOL permet de faire l'économie de cette phase. Toute technique de compilation ou d'optimisation appliquée à une application définie sur la base de ce langage doit simplement respecter les dépendances de données exprimées. Cet aspect offre un certain nombre de bénéfices :

- **Une description simple** des algorithmes,
- **Pas de phase d'analyse des dépendances** à effectuer par un compilateur ou un quelconque outil d'optimisation,
- **Tout le parallélisme potentiel de l'algorithme exprimé** dans le modèle, sous une forme facilement exploitable,
- **Spécification non biaisée** par des informations relatives à un modèle d'exécution.

Dans les sections suivantes, nous donnons une description du langage, et illustrons son utilisation par un exemple simple. La compréhension de cette section revêt une importance capitale pour la compréhension du reste de document. Les idées présentées ici ont en effet fortement orienté la définition du profil Gaspard présenté dans les chapitres suivants, telles que la modélisation d'applications par assemblage, mais aussi et surtout les mécanismes de

factorisation de l'expression des dépendances de données.

4.3.1 Expression des dépendances de données

ArrayOL propose une approche de construction des applications par "assemblage". Les "briques" utilisées pour la définition de ces assemblages sont simplement des fonctions produisant des sorties à partir de leurs entrées. Les données manipulées par ces fonctions sont uniquement des tableaux multidimensionnels. Ces fonctions, communément appelées tâches dans ArrayOL, sont reliées entre elles par des dépendances de données. L'expression de ces dépendances permet dans un premier temps de définir un ordre partiel minimal d'exécution des différentes tâches : une tâche ne peut produire ses résultats qu'à partir du moment où les données dont elle dépend ont été produites par une autre tâche, ou de manière plus générale par son environnement.

Ce mode d'expression supporte naturellement la modélisation du parallélisme de tâches. On parle alors de modèle global. Nous verrons par la suite qu'il supporte également l'expression du parallélisme de données, via des mécanismes de factorisation de l'expression des dépendances. On parlera alors de modèle local. Mais avant d'entrer plus en détail dans la description de ces mécanismes, il est important de noter que ce mode d'expression n'impose aucun modèle d'exécution. Une même spécification pourrait tout à fait donner lieu à une exécution séquentielle, ou à une exécution parallèle de type pipeline ou SPMD.

Les dépendances de données constituent le seul moyen d'interconnecter des tâches dans ArrayOL. Lorsqu'une dépendance de données est exprimée entre deux composants, cela signifie que l'un des composants a besoin de tout ou partie des données produites par l'autre composant pour pouvoir réaliser ses calculs. Deux types de dépendances sont identifiées : les dépendances dites "tableau" dans le cadre du modèle global, et les dépendances dites "motif" dans le cadre du modèle local.

▶ **Dépendances tableau** Les dépendances dites "tableau" sont exprimées dans le cadre du modèle global. Par modèle global, on entend un graphe de tâches orienté et acyclique. L'expression d'une dépendance tableau entre deux tâches signifie que l'une des tâches a besoin d'une partie ou de tous les éléments produits par l'autre tâche pour réaliser ses calculs (fig.2.6). Il s'agit d'une forme d'expression grossière des dépendances de données. Cette expression sera raffinée via l'utilisation de dépendances motif dans le cadre du modèle local.

▶ **Dépendances motif** Les dépendances dites "motif" sont exprimées dans le cadre du modèle local. Par modèle local, on entend la spécification de la structure d'une tâche, sous la forme d'une tâche répétée. Chaque répétition de cette tâche consomme un motif à l'intérieur de chaque tableau d'entrée, afin de produire un motif à l'intérieur de chaque tableau de sortie de la tâche la contenant (fig.2.7). Par motif, on entend simplement un ensemble de points contenus dans un tableau. Dans ArrayOL, les dépendances motifs sont indissociables des mécanismes de factorisation décrits dans la section suivante.

4.3.2 Factorisation de l'expression des dépendances

L'utilisation de dépendances motif dans le cadre du modèle local permet d'exprimer tout le parallélisme de données potentiel d'une tâche. Lorsque la structure d'une tâche est décrite à l'aide d'un modèle local, elle définit un espace de répétition pour la tâche

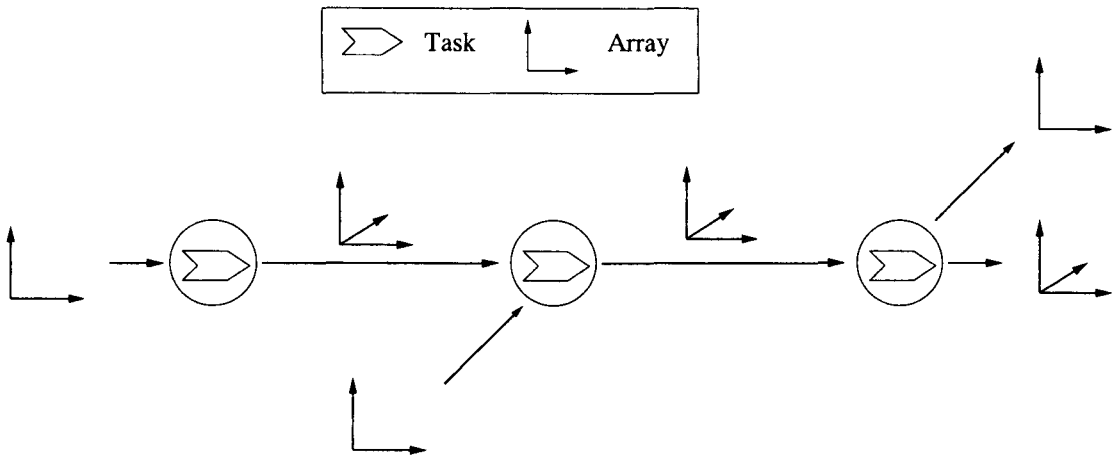
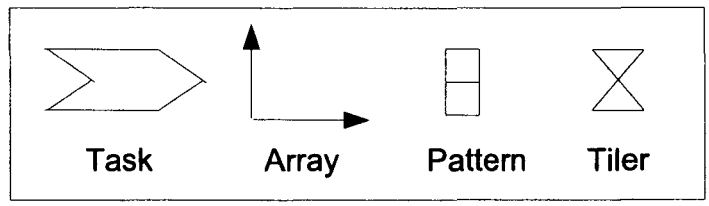


FIG. 2.6 – Dépendances tableau dans un modèle global



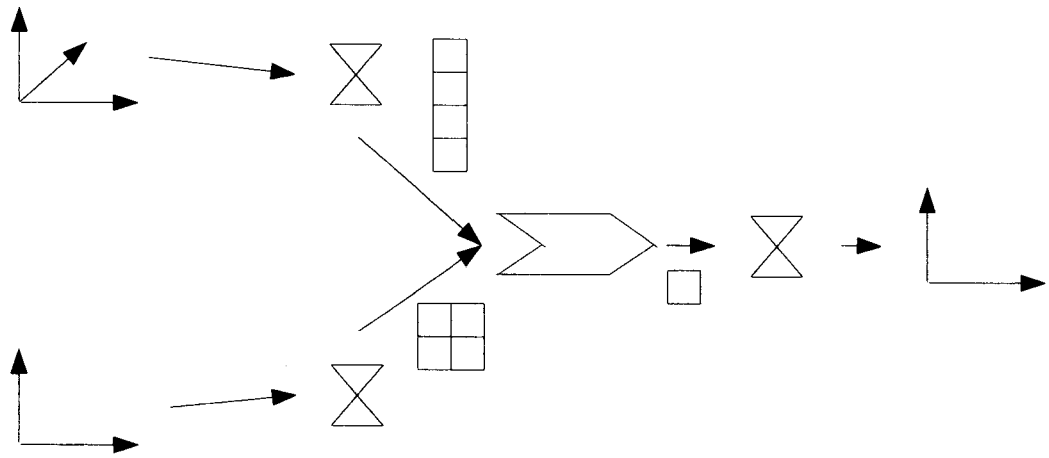


FIG. 2.7 – Dépendances motif dans un modèle local

contenue. L'expression "espace de répétition" (ou espace d'itération) est généralement utilisée pour représenter toutes les valeurs possibles des indices d'une boucle (ou d'un nid de boucle). Les limites de répétition, c'est à dire la valeur maximale pour chaque indice de boucle, peuvent être représentées par un vecteur d'entiers positifs, où chaque entier représente la limite de répétition de chaque indice de boucle. Pour chaque point de l'espace de répétition identifié par des coordonnées appartenant à l'espace de répétition, on associe une répétition de la tâche, et un motif d'entrée/sortie contenu dans chaque tableau d'entrée/sortie consommé/produit. En d'autres termes, chaque tâche connaît un sous-ensemble de points à consommer dans chacun des tableaux d'entrée, et un sous-ensemble à produire dans chacun des tableaux de sortie. Les coordonnées des motifs à l'intérieur des tableaux d'entrée/sortie sont basiquement identifiées par deux matrices : Les matrices de pavage et d'ajustage (en fig.2.7, ces vecteurs, bien qu'ils n'apparaissent pas, sont capturés par le concept de tiler. Dans la suite du document, et en particulier dans le chapitre décrivant le profil Gaspard, nous décrirons en détail ces mécanismes). La seule véritable contrainte liée à l'utilisation de ce mécanisme est que chaque point d'un tableau de sortie ne peut être produit qu'une et une seule fois. Cette contrainte, ou plutôt propriété, est appelée "assignation unique". Cela garantit notamment que l'exécution de chaque répétition de la tâche répétée peut être considérée comme indépendante de l'exécution des autres tâches : Il y a donc expression de tout le parallélisme potentiel du traitement relatif à la production des tableaux de sortie.

▶ **Ajustage** Les matrices d'ajustage (notées F en fig.2.8, pour Fitting) sont utilisées pour identifier les points appartenant à un motif, à partir d'une origine donnée à l'intérieur d'un tableau. Identifier les points appartenant à un motif peut être réalisé en appliquant un processus itératif : Chaque point appartenant à un motif est identifié en multipliant chaque vecteur composant la matrice d'ajustage par chaque point d'un espace de répétition donné. Les limites de cet espace de répétition sont déterminées par les dimensions du motif noté $\vec{d}$, définies au niveau de la tâche répétée (fig.2.8).

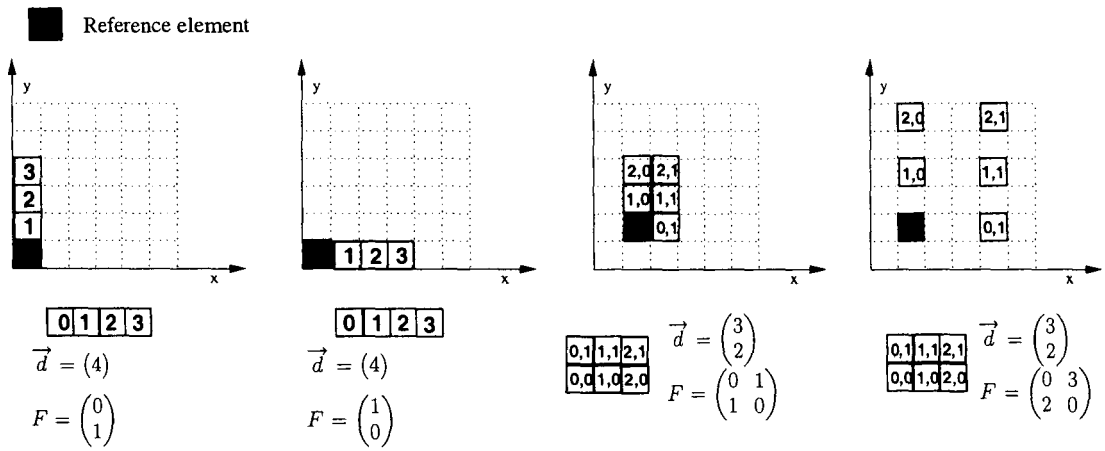


FIG. 2.8 – Exemples d'ajustage

Appliquer ce processus itératif peut amener à identifier des coordonnées de points en

dehors du tableau. Dans ce cas, les coordonnées des points sont considérées modulo les dimensions du tableau. A noter que cette remarque s'applique également au processus de pavage.

▶ **Pavage** Les matrices de pavage (notées P en fig.2.9) sont utilisées pour identifier les origines considérées lors du processus d'ajustage, ou en d'autres termes pour identifier l'origine de chaque motif à l'intérieur d'un tableau. Identifier ces origines peut également être réalisé par un processus itératif, similaire à celui du processus d'ajustage : Chaque origine est identifiée en multipliant chaque vecteur composant la matrice de pavage par chaque point de l'espace de répétition relatif au modèle local concerné (fig.2.9). Un vecteur nommé origine (noté $\vec{(o)}$) est utilisé pour définir l'origine du processus de pavage.

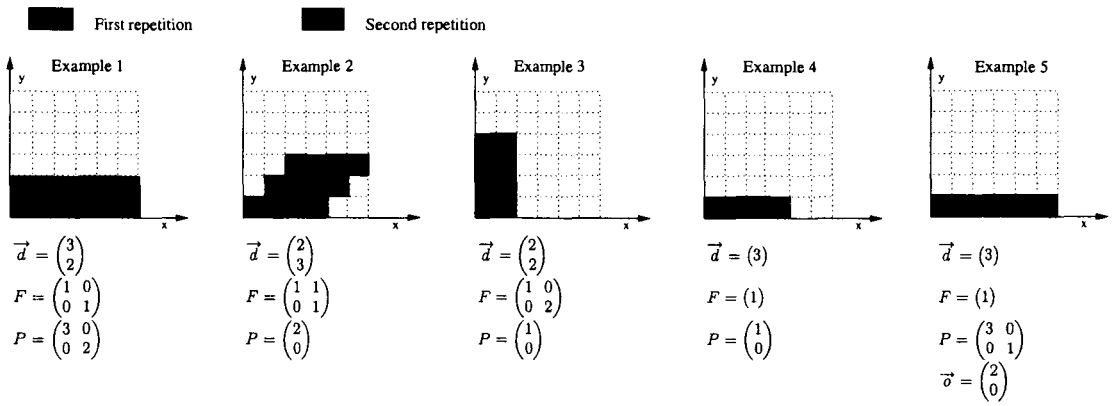


FIG. 2.9 – Exemples de pavages

Une des dépendances motif pour la production d'un tableau de sortie peut être qualifiée de "maître". Identifier qu'une dépendance joue le rôle de maître permet de déterminer automatiquement les limites de l'espace d'itération de pavage, sur la base des matrices de pavage et d'ajustage associés à cette dépendance. Les limites calculées garantissent que le tableau de sortie est complètement produit, tout en respectant la contrainte d'assignation unique précédemment évoquée. Calculer les limites de répétition de pavage revient à déterminer le nombre de motifs nécessaires, et a fortiori le nombre de tâches potentiellement exécutables en parallèle, pour produire complètement le tableau de sortie concerné.

4.3.3 Unification des dimensions temporelles et spatiales

Les tableaux multidimensionnels constituent les données de base manipulées par une application. L'une des propriétés clés de ces tableaux est que les dimensions temporelles et spatiales sont unifiées. Cela signifie que pour la spécification d'une application donnée, le temps n'est jamais explicitement différencié. Les dépendances de données sont exprimées via le même formalisme, qu'elles soient spatiales ou temporelles.

Les tableaux multidimensionnels d'ArrayOL sont caractérisés par une forme, c'est à dire le nombre de dimensions et la taille de chaque dimension, et par le type élémentaire des éléments qu'ils contiennent. Seule une dimension d'un tableau peut être de taille infinie.

Elle représente tous les échantillons pouvant être réalisés au cours du cycle de vie d'une application. Dans ce cas, le cycle de vie de l'application est potentiellement infini. Cela est par exemple représentatif pour une application de traitement de signal, où la quantité de données qui sera manipulée sur le temps ne peut pas être spécifiée au moment de la conception. A noter que la taille des tableaux est spécifiée de façon statique à la conception.

4.3.4 Exemple : Produit de matrices

Nous allons illustrer l'utilisation du langage ArrayOL par la modélisation d'un exemple académique : le produit de matrices. Cet exemple simple va nous permettre de montrer la puissance du modèle local pour l'expression du parallélisme de données d'un algorithme. Dans cet exemple, nous allons considérer un produit de deux matrices de dimensions 6×6. Calculer le produit de deux matrices revient à calculer le produit scalaire de chaque ligne d'une matrice par chaque colonne d'une autre matrice. Le calcul de chaque produit scalaire est indépendant des autres calculs, et tous les calculs peuvent donc être exécutés en parallèle. Fig.2.10 montre de façon explicite les différents produits scalaires pouvant être exécutés parallèlement. Pour produire le point d'indice `(i,j)` dans la matrice résultat, il suffit de réaliser le produit scalaire de la ligne d'indice `i` de la matrice `m1` par la colonne d'indice `j` de la matrice `m2`.

ArrayOL nous permet d'exprimer directement tout ce parallélisme, en identifiant les motifs nécessaires dans les tableaux d'entrée pour la production des motifs des tableaux de sortie. Considérons une tâche élémentaire `DotProduct` prenant en entrée deux vecteurs de taille 6, et produisant en sortie un scalaire, correspondant au produit scalaire des deux entrées. Dans le modèle local illustré en fig.2.11, l'espace de répétition de `DotProduct` est défini par le vecteur [6,6] (une tâche pour la production de chaque point du tableau de sortie). On constate par exemple en fig.2.11 que pour le calcul du point d'indice [4,2], l'origine de la ligne à considérer dans la matrice m1 est `[[0,0],[0,1]]×[4,2]+[0,0]=[0,2]`, soit les vecteurs de pavage multipliés par les coordonnées du point considéré dans l'espace de répétition de pavage, auquel on ajoute le vecteur d'origine. En généralisant ce processus pour chaque point de l'espace de répétition de pavage, on obtient l'origine de chaque ligne considérée pour le calcul de chaque produit scalaire. Pour connaître par exemple le troisième point appartenant à la ligne d'origine [0,2], on utilise un processus similaire : `[1,0]×[3]+[0,2]=[3,2]`, soit les vecteurs de pavage multipliés par les coordonnées du point considéré dans l'espace de répétition d'ajustage (c'est à dire la taille du motif), auquel on ajoute le vecteur déterminant l'origine du motif (qui est déterminé par le processus d'ajustage). En généralisant ce processus à l'ensemble des points de l'espace de répétition d'ajustage, on détermine l'ensemble des points appartenant à un motif.

Chaque répétition du produit scalaire étant indépendante des autres répétitions, et consommant chacune des motifs clairement identifiés à l'intérieur des tableaux d'entrée, le parallélisme potentiel de l'application est complètement exprimé, de façon compacte, via une expression en compréhension facilement interprétable par un outil de compilation ou d'optimisation.

4.4 Bilan

Le modèle de programmation à parallèlisme de données est un modèle de haut niveau, dans le sens où tous les aspects relatifs à la synchronisation du modèle à parallèlisme de

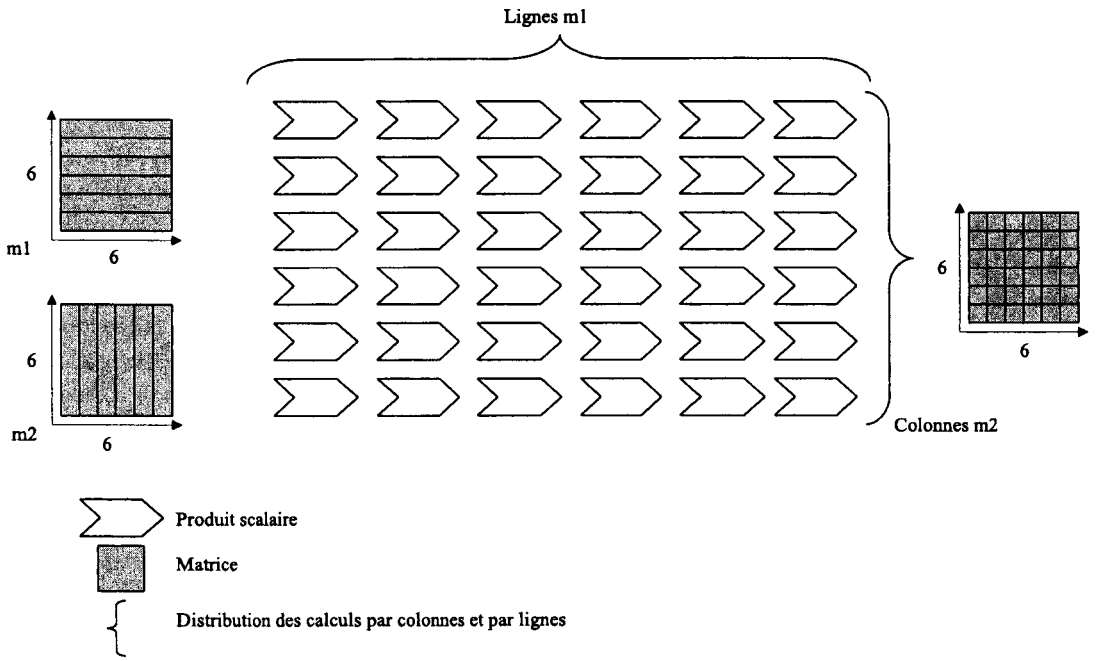


FIG. 2.10 – Parallélisme potentiel du produit de matrices

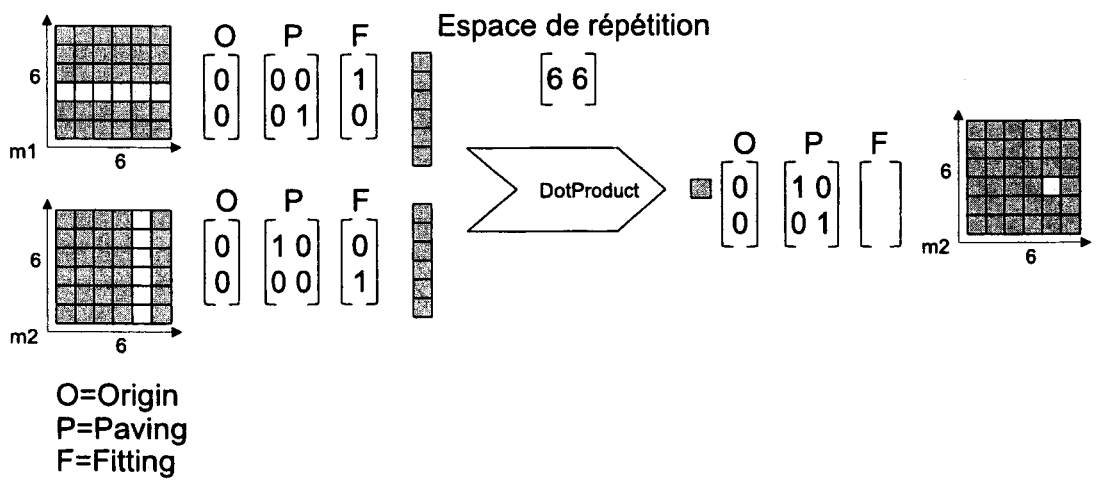


FIG. 2.11 – Expression du parallélisme de données

contrôle sont abstraits. S'il y a perte au niveau du pouvoir d'expression, il y a un gain sémantique, qui permet de minimiser les risques d'erreurs, les phases de débogage, et ainsi de minimiser le temps de conception. Dans ce sens, le modèle de programmation à parallèlisme de données peut fournir une réponse au problème de déficit de productivité évoqué en introduction de ce document. Au niveau des langages, notre attention s'est portée sur le langage ArrayOL. Même si nous n'en faisons pas la démonstration ici, ArrayOL propose un pouvoir d'expression du parallèlisme de données comparable à celui de HPF [1]. Par ailleurs, ArrayOL propose une solution élégante pour la modélisation des applications offrant potentiellement beaucoup de parallélisme : Tout le parallélisme de l'application est exprimé de façon compacte, sans ajout d'informations parasites. A ce titre, ArrayOL va constituer un bon point de départ pour la définition du profil Gaspard, puisque la classe d'applications visée par ArrayOL fait indiscutablement partie des cibles de Gaspard. Néanmoins, ArrayOL ne peut être utilisé seul pour la mise en oeuvre d'une méthodologie de co-design, puisqu'il ne prend en compte que la modélisation des applications, et pas la modélisation des architectures matérielles (et par conséquent la modélisation du placement d'une application sur une architecture). Par ailleurs, au niveau applicatif, le profil Gaspard a pour objectif d'aller plus loin que la modélisation d'applications purement data-parallèle, où toutes les répétitions sont totalement indépendantes. On souhaiterait pouvoir modéliser des applications où le résultat produit par une répétition est utilisé par une autre répétition. Par la suite, nous verrons comment ces lacunes seront comblées. Il est ici important de retenir que la principale contribution d'ArrayOL pour le profil Gaspard se situe au niveau de ses mécanismes de factorisation de l'expression des dépendances de données. Dans le chapitre consacré au profil Gaspard, nous verrons comment ces mécanismes sont extraits du contexte d'ArrayOL, et généralisés à l'ensemble des aspects couverts par Gaspard : Application, Architecture matérielle et Association.

5 Modélisation des architectures matérielles

La prise en compte de l'architecture matérielle est un élément essentiel du développement d'applications embarquées efficaces, particulièrement dans les cas où ces applications sont fortement contraintes (temps, consommation d'énergie, ...). Disposer d'un langage ou d'un modèle permettant d'exprimer les spécificités du matériel est une nécessité. Un grand nombre d'études ont été menées pour proposer un moyen de décrire ces aspects matériels, aboutissant à la définition d'une multitude de langages. Un flot de conception de systèmes sur puce est généralement basé (comme nous l'avons déjà évoqués) sur plusieurs niveaux d'abstractions. Bien qu'il n'existe pas à priori de langage couvrant efficacement tous ces niveux, il existe des propositions fournissant des réponses pour des problèmes relatifs à chacun de ces différents niveaux d'abstractions. Ainsi, pendant le processus de conception d'un système, les langages peuvent être utilisés à plusieurs fins :

- La spécification et la documentation des systèmes : dans ce cas, aucun modèle exécutable n'est nécessaire.
- La validation des systèmes : un modèle exécutable est requis.
- La vérification des systèmes : la vérification nécessite un modèle formel du langage.
- Le raffinement et la synthèse : une sémantique pour la réalisation est nécessaire.

[1] Cette démonstration sera faite dans la thèse de Philippe Dumont, LIFL, à venir

La proposition que nous réaliserons par la suite concerne plutôt le premier aspect, c'est à dire la spécification et la documentation du système. Notre objectif est en effet de fournir un moyen de dimensionner l'architecture matérielle, afin de pouvoir exprimer un placement "raisonnable" de l'application concernée sur l'architecture matérielle visée. Néanmoins, nous allons réaliser, à titre indicatif, un tour d'horizon des différentes solutions proposées sur le marché. Cette étude est partiellement basée sur [41], que nous avons complété et réactualisé.

5.1 HDLs

Les HDLs (Hardware Description Languages) constituent les moyens de plus bas niveau pour modéliser du matériel. Ces langages supportent des concepts spécifiques pour la modélisation du matériel, tels que les communications interprocessus (signaux, protocoles), la réactivité, ainsi que des types de données spécifiques (entiers signés et non signés, données en virgule fixe ou flottante, et structures de vecteurs de bits). Les deux principaux HDLs utilisés sur le marché sont VHDL et Verilog. Le niveau d'abstraction auxquels sont définis ces langages en fait des outils performants pour la modélisation au niveau RTL. Néanmoins, dans une méthodologie telles que Gaspard, où l'on commence la modélisation à des niveaux d'abstractions beaucoup plus élevés, ces langages ne sont pas utilisables. Il est nécessaire dans ce cas de s'abstraire des détails d'implémentation de l'architecture, afin d'en extraire les informations pertinentes pouvant orienter le placement de l'application. Néanmoins, comme nous l'avons évoqué dans le chapitre 2 (section décrivant le flot de conception Gaspard), VHDL pourrait être envisagé comme une cible des transformations de modèle.

5.2 Langages niveau système

Les langages de description niveau système sont généralement définis comme des extensions de langages de programmation classiques (typiquement C et C++), en y ajoutant des concepts orientés vers la description du matériel. Le choix d'étendre des langages de programmation classiques est principalement motivés par trois raisons :

- Des langages tels que C ou C++ fournissent les structures de contrôle et les types de données nécessaires
- La plupart des systèmes contiennent des parties logicielles et matérielles. Ces langages de programmation fournissent une solution naturelle pour la description de la partie logicielle.
- Ces langages et les environnements de programmation qui leurs sont associés sont familiers des concepteurs de système.

Les langages niveau système introduisent généralement des extensions aux langages de base sous la forme de bibliothèques de fonctions ou de classes, permettant notemment la prise en compte de la concurrence, du temps, ou encore des comportements réactifs. Les langages les plus connus sont SystemC, et SpecC.

A l'origine fortement orientés vers le niveau RTL, ces langages tendent à s'abstraire vers des niveaux de description plus élevés, les niveaux TLM, que nous avons présentés dans le chapitre 2 (section décrivant les tendances méthodologiques pour la conception des systèmes sur puce). La principale critique que l'on peut adresser à ce type de solutions est

que modèle d'application et modèle d'architecture ne sont pas clairement séparés. Le partitionnement logiciel/matériel intervient à des niveaux relativement bas, proches du RTL. Cela nous rammène à une problématique comparable à celle évoquée pour les HDLs : Il n'est pas possible d'extraire un dimensionnement de l'architecture permettant d'orienter le placement d'une application spécifiée à haut niveau d'abstraction.

5.3 ADLs

Les ADLs (Architecture Description Languages) fournissent des solutions plus adaptées aux besoins du flot de conception Gaspard. Ces langages permettent une définition plus ou moins formelle, à haut niveau d'abstraction, de l'architecture (logicielle et/ou matérielle) d'un système. Les ADLs introduisent principalement trois concepts de base pour la modélisation des systèmes [39] : les modules, les connecteurs, et les configurations (combinaisons particulières de modules et de connecteurs). Beaucoup d'ADLs sont disponibles sur le marché. Une liste exhaustive ainsi qu'une comparaison de ces différents ADLs est disponible dans [56] (Bien que datée, cette étude demeure représentative). Nous pouvons cependant nous attarder sur AADL, d'une part pour son modèle de description d'architectures matérielles, et d'autre part pour les liens existants entre AADL et l'OMG. A l'heure où ces lignes sont écrites, une RFC concernant un profile UML pour AADL est toujours d'actualité. Des discussions sont en cours pour déterminer si cette RFC a toujours lieu d'être, étant donné qu'il existe beaucoup d'intersections entre les requirements adressés par la RFP MARTE, et les solutions proposées dans AADL. Ces discussions sont d'autant plus justifiées que le profil en est toujours à un stade embryonnaire.

AADL [19] (autrefois Avionics and Architecture Description Language, et aujourd'hui Architecture Analysis and Design Language), lui même basé sur MetaH, permet de décrire la structure d'un système embarqué par un assemblage de composants logiciels et matériels. Il prend en compte à la fois les aspects fonctionnels (données en entrée et en sortie) et non fonctionnels (informations quantitatives, relatives au temps par exemple) de la spécification des composants. Définit à un haut niveau d'abstraction, le modèle d'architecture matérielle de AADL introduit naturellement des concepts permettant la description de ressources de calcul (processor), de stockage (memory), de communication (bus), ainsi qu'une catégorie générique incluant tout composant matériel n'appartenant pas aux trois précédentes catégories (device). Chacun de ces différents concepts possède un certain nombre d'attributs, permettant de caractériser une architecture matérielle, en vue d'appliquer différentes techniques d'analyse (principalement ordonnançabilité et tolérance aux pannes). Bien que ces différentes concepts aient une sémantique fortement liée au modèle de description d'application de AADL, et que les différentes caractérisations proposées ne correspondent pas réellement aux besoins de Gaspard, ce type de modèle, de plus haut niveau, se rapproche de nos objectifs (dimensionnement et caractérisation de l'architecture pour orienter le placement).

5.4 Méthodes orientées UML

Un certain nombre de proposition visant à enrichir UML pour permettre la description de matériel existent. Ces propositions se situent à un niveau d'abstraction comparable à celui de AADL, et permettent dans l'ensemble de dimensionner l'architecture matérielle d'un

système. Nous citons quelques exemples représentatifs de modèles et méthodologies proposés dans la communauté.

HaSoC [23] (Hardware and Software object on Chip) propose une méthodologie de type platform-based, basée sur l'utilisation de UML pour représenter la structure et le comportement d'une architecture matérielle à un haut niveau d'abstraction. Le modèle matériel de HaSoC permet basiquement la modélisation des ressources de calculs (généralistes, reconfigurables, ou spécialisées) et des élements d'interconnexion permettant d'établir des communications entre les différentes ressources. Ce modèle permet la génération de code SystemC pour la simulation du système, et éventuellement la synthèse. La principale limitation du modèle concerne l'absence de moyen pour représenter les éléments de stockage.

Le processus de conception SLOOP [64] (System Level design with Object-Oriented Process), ancêtre de UML for SoC (que nous présenterons dans le chapitre suivant), intégre une méthodologie basée sur UML, permettant à la fois la modélisation des systèmes sur puce, et l'évaluation des performance des systèmes modélisés. Le modèle de description d'architecture matérielle intègre des éléments similaires à HaSoC et AADL (processeurs, mémoires, interconnexions). Les architectures sont modélisées dans des diagrammes de déploiement (UML 1.x), par le biais d'un certain nombre de stéréotypes appliqués au concept de nœud (`Node`). La limitation du modèle se situe principalement au niveau de la description des interconnexions, qui ne laissent aucune liberté de choix et sont systématiquement considérées comme des FIFOs.

Enfin, Accord/UML [35] propose une méthodologie de conception des systèmes temps réel distribué basée sur UML. A l'origine orientée vers des aspects purement logiciels, Accord a été étendu dans le cadre du projet Acotris [1] pour prendre en compte également des aspects matériels. Le modèle d'architecture matérielle est directement dérivé de celui de SynDEx, que nous décrivons dans la section suivante.

5.5 Autres outils et langages

En dehors des différentes catégories que nous avons jusqu'à présent mentionné, certains langages ou outils, orientés ou non vers les systèmes embarqués, proposent des solutions pour dimensionner les architectures matérielles.

C'est le cas par exemple pour des langages orientés vers le parallèlisme de donnée, tels que HPF (que nous avons décrit dans la section consacrée au modèle de programmation à parallèlisme de données), via la structure nommée `processors`. Cette structure représente une grille de processeurs virtuels, et est utilisée pour définir une distribution des données et par extension des calculs (il s'agit du "owner computes rule", où les calculs "suivent" les données). Néanmoins, il est impossible de décrire la topologie des connections entre les différents processeurs virtuels de la grille. Parallaxis [5] reprend ce principe de machine virtuelle, mais enrichit le mécanisme en permettant de décrire la topologie des connections. Il est ainsi possible, à partir d'une description de l'application et de l'architecture virtuelle, de mettre en œuvre un placement tenant compte de la topologie des connexions. La limitation de ses approches est de ne pas tenir compte des architectures mémoires ou de la nature des connexions entre les processeurs virtuels.

Des outils, tels que SynDEx [58], proposent également des modèles d'architecture matérielle. SynDEx est un outil de conception basé sur la méthodologie AAA (Adéquation Algorithme Architecture), pour le prototypage rapide et l'implantation optimisée d'applications embarquées, temps-réel, et distribuées sur des architectures matérielles multipro-

cesseurs. Le modèle d'architecture de SynDEx prend en compte des processeurs (nommés `operator`) et des interconnexions (nommées `communication media`). Les médias de communication ont pour particularité de réprésenter une abstraction à la fois de l'architecture mémoire du système, et de l'architecture de communication (telle qu'on l'entend classiquement : bus, connexions point à point, réseau d'interconnexion, ...). Si cette abstraction est adaptée pour la mise en œuvre de la méthodologie AAA (définition du placement optimisé des calculs sur les différents processeurs et des échanges de données sur les chemins de communication entre processeurs), elle peut ne pas être satisfaisante dans une méthodologie de type IP based, où, si le concepteur d'un système possède une IP représentant un interconnect et une IP représentant une mémoire, il souhaite pouvoir les différencier.

5.6 Bilan

Outre les différentes limitations évoquées pour chacuns des langages, outils, ou méthodes, il est important de retenir ici que les solutions trop bas niveau telles que celles proposées par les HDL ou les langages niveau système ne sont pas adaptées aux besoins de Gaspard. L'objectif est d'être en mesure de dimensionner l'architecture, et de la caractériser, afin d'orienter le placement des applications à un haut niveau d'abstraction. Les solutions proposées par SynDEx, Parallaxis, ou encore les différentes méthodologies orientées vers UML proposent des solutions qui correspondent nettement plus à nos attentes. La solution que nous proposerons dans le chapitre consacré à la description du profil Gaspard suivra gloablement ces tendances, modulo quelques différences concernant la sémantique, la mise en œuvre (réalisée par extension du profil SPT) et les concepts introduits. Par ailleurs, les aspects concernant les architectures matérielles régulières (pris en compte par HPF et Parallaxis) seront pris en compte via nos mécanismes communs de factorisation.

CHAPITRE 3

Contribution des standards OMG à la définition du profil Gaspard

1 **SPT** **44**
1.1 Modélisation : General Resource Modeling Framework 44
1.2 Analyse : Analysis Models 46
1.3 Classification des ressources 47
1.4 Bilan 47
2 **QoS** **48**
2.1 Framework de définition des QoS 48
2.2 Méthodologie d'annotation des modèles 49
2.3 Bilan 50
3 **SysML** **51**
3.1 SysML vs UML 2 52
3.2 Allocation 52
3.3 Bilan 54
4 **Uml for SoC** **55**
4.1 Extensions pour la modélisation structurelle des SoCs 55
4.2 Bilan 55

Dans le chapitre précédent, nous avons positionné le flot de conception Gaspard par rapport aux tendances méthodologiques et technologiques pour la conception des systèmes sur puce, et nous avons montré quel pouvait être l'intérêt du parallèlisme de données (et en particulier du langage ArrayOL) et du dimensionnement à haut niveau d'abstraction des architectures matérielles dans la conception de tels systèmes. Nous allons maintenant voir que les standards définis à l'OMG (Object Management Group, organisme en charge notamment de la standardisation d'UML), qu'ils soient déjà adoptés ou en cours d'adoption, et étant plus ou moins orientés vers la modélisation des systèmes embarqués, fournissent globalement des solutions adaptées aux objectifs que nous avons évoqué dans la section Contribution du chapitre introductif de ce document (à l'exception des mécanismes de factorisation). Malheureusement, aucun de ces standards ne fournit une solution complète. Notre stratégie est donc de réutiliser, pour chacun de ces standards, les éléments qui proposeront une réponse (au moins partielle) à un de nos objectifs particuliers. Sur la base de ces éléments retenus, nous définirons nos propres extensions. Ainsi, nous allons voir que :

- **UML 2** fournit des solutions pour la mise en oeuvre de l'**approche orientée composant**[1],
- le profil **SPT** fournit des solutions pour la mise en oeuvre de notre **modèle d'architecture matérielle**,
- le profil **QoS** fournit des solutions pour la mise en oeuvre des **caractérisations**,
- **SysML** propose des mécanismes d'allocation logiciel/matériel adaptés à notre **modèle d'association**.

Nous étudierons ensuite une autre proposition standard (ou ayant vocation à le devenir) : Uml for SoC, en mettant en évidence le fait qu'elles ne proposent pas d'éléments susceptibles de nous intéresser.

[1]Pour faciliter la compréhension du profil Gaspard, les mécanismes offerts pas UML 2 ne seront pas décrits dans ce chapitre, mais directement intégrés dans la description du profil Gaspard

1 SPT

Le profil SPT (Schedulability, Performance, and Time) [53] fut soumis et adopté en 2001 en réponse à la RFP proposée par le Real-Time Analysis and Design Working Group (RTAD) de l'OMG. L'objectif du profil est de définir une façon commune pour représenter le temps (et d'autres propriétés qui lui sont relatives), le premier bénéfice étant de pouvoir échanger les propriétés quantitatives des modèles entre différents outils. SPT propose ainsi une façon standard d'annoter les modèles UML avec des informations temporelles, la standardisation permettant de faciliter l'interaction entre modeleurs UML et outils d'analyse de performance. Dans cette étude, le profil SPT présente essentiellement un intérêt historique, puisqu'il s'agit de la première tentative de l'OMG de définir un standard pour la modélisation des systèmes temps réel. Depuis l'adoption du profil, d'autres standards aux impacts non négligeables ont fait leur apparition. Plus précisément, SPT est basé sur les versions 1.x d'UML, alors qu'UML 2 [45] [48] [44] [46] a entre temps été standardisé. De la même manière, SPT introduisait la notion de "qualité de service" dans UML, alors que le profil QoS (Quality Of Service and Fault Tolerance [49]), entièrement dédié à cet objectif, a été standardisé. A cela s'ajoute un nombre significatif de remarques et de suggestions pour l'amélioration et la consolidation du profil, qui ont amené la communauté à considérer qu'une révision mineure du standard n'était pas envisageable, et qu'il était nécessaire de réaliser une révision de fond via la proposition d'une nouvelle RFP : MARTE (Modeling and Analysis of Real-Time and Embedded systems) [52], à laquelle nous avons participé via le consortium PROTES. Nous reviendrons en détail sur cette RFP dans la conclusion de ce document.

Le profil SPT s'articule autour de cinq sous packages, eux-mêmes regroupés en deux packages : Des packages génériques définissant des outils pour la modélisation des applications temps réel (regroupés dans le package `General Resource Modeling Framework`), et des packages spécialisés pour la mise en oeuvre d'activités d'analyse spécifiques aux applications temps réel (regroupés dans le package `Analysis Models`). Nous donnons dans les sections qui suivent un aperçu du contenu de ces différents packages, mais nous nous attarderons sur la classification des ressources proposée dans le profil SPT. Nous verrons dans le chapitre consacré à la définition du profil Gaspard que cette classification servira de point de départ à la définition de notre modèle d'architecture matérielle.

1.1 Modélisation : General Resource Modeling Framework

L'objectif du package `General Resource Modeling Framework` (GRM) est de définir l'ensemble des concepts de base nécessaires à la modélisation de systèmes temps réel. Le package se divise en trois sous-packages : `RT Resource`, `RT Time` et `RT Concurrency`. Le premier définit les concept de "ressource" et de "qualité de service", notions fondamentales pour le profil SPT. Le second définit un ensemble d'aspects relatifs au temps dans les systèmes temps réel : temps, durées, et sources d'évènements temporels (tels que timers et horloges). Le dernier raffine le modèle de concurrence défini dans le métamodèle UML (1.x).

Les deux concepts fondamentaux introduits dans le package `GRM` sont les concepts de ressources et de qualités de service (QoS). Une ressource est définie comme étant une entité offrant un ou plusieurs services à son environnement. Le concept de qualité de service permet de quantifier les limitations d'un ou plusieurs services offerts par une ressource. Le

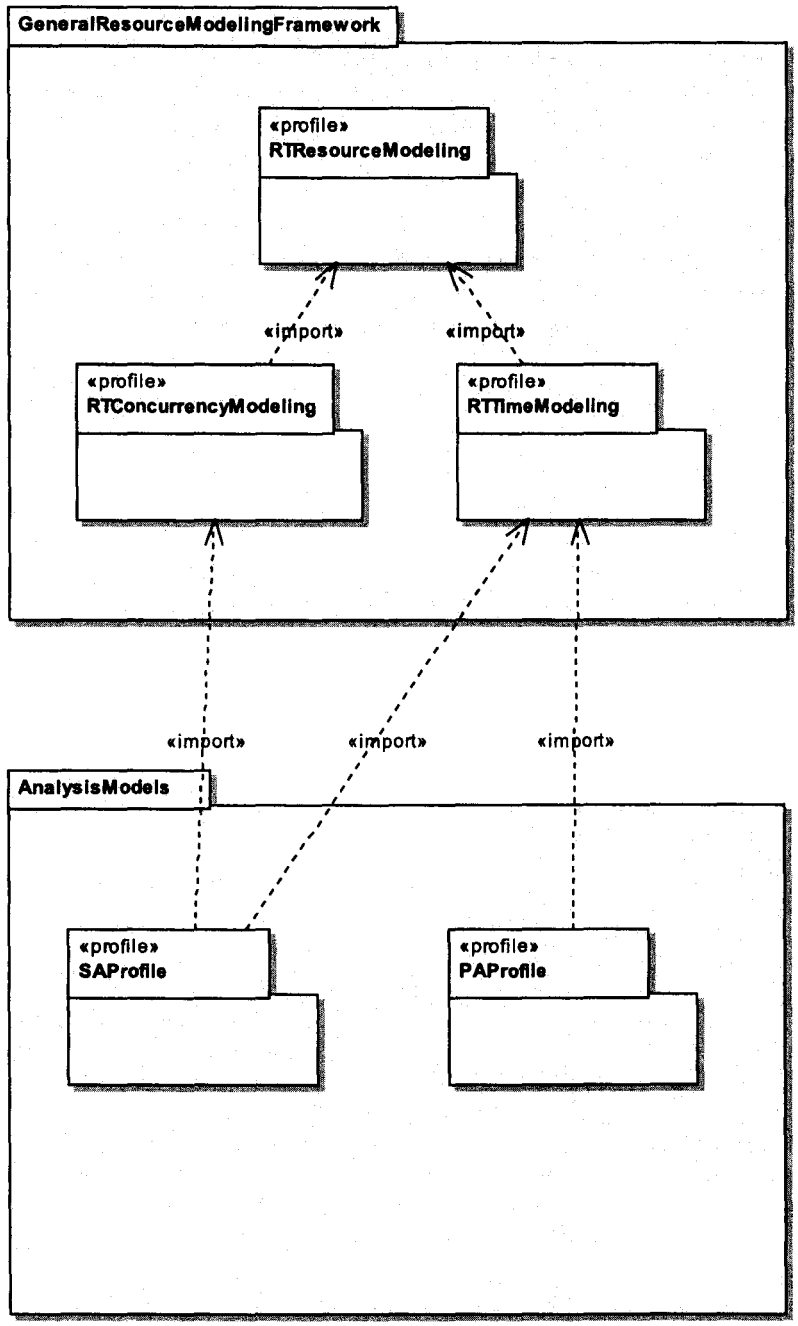


FIG. 3.1 – Structure du profil SPT

package RT Resource propose une classification intéressante des ressources. Dans la section 1.3, nous nous attarderons en détail sur cette classification. La définition du package et des éléments qu'il contient est basée sur un modèle classique de type client/serveur : Les clients requièrent des services aux serveurs qui se chargent de leur exécution. Ce type de requête de service est identifié par le terme "utilisation de ressource" (Resource Usage), et peut éventuellement être caractérisé par un ou plusieurs paramètres de qualité de service. On retrouve deux aspects orthogonaux pour les paramètres de QoS : QoS offerte, et QoS requise. Les paramètres de QoS associés à une requête émise par un client correspondent à la qualité de service attendue, ou requise, par le client pour l'exécution de ce service. Du côté du serveur, les paramètres de QoS spécifient la qualité qu'il est en mesure de fournir pour l'exécution d'un service. La cohérence des paramètres de QoS est vérifiée simplement si la qualité de service offerte est au moins aussi bonne que la qualité de service requise. En fig.3.2, nous illustrons ce mécanisme avec un exemple tiré de [15].

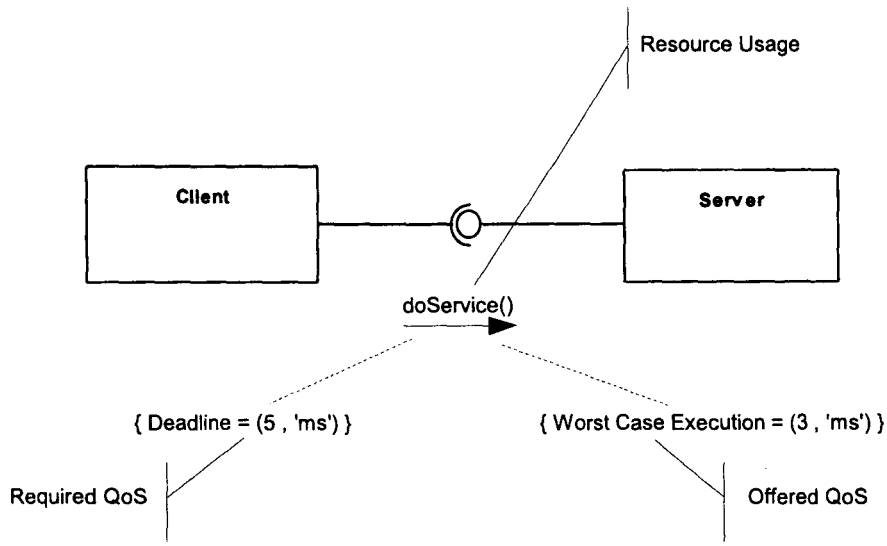


FIG. 3.2 – Modèle client/serveur et qualité de service dans le GRM

1.2 Analyse : Analysis Models

L'objectif du package Analysis Models est de définir un support pour différentes méthodes d'analyse. Ces différentes méthodes sont identifiées dans des sous-packages de Analysis Models, et il est d'ailleurs supposé que l'intégration de nouvelles méthodes spécifiques d'analyse soit réalisées via la définition de sous-packages de Analysis Models. Ainsi, le sous-package Schedulability Analysis (SA) définit les stéréotypes, tagged values et contraintes pour des formes communes d'analyse d'ordonnançabilité. De la même manière, le sous-package Performance Analysis fournit un support pour des méthodes communes d'analyse de performance. Ces deux packages réutilisent les concepts définis dans les packages RT Resource, RT Time et RT Concurrency du package GRM. Pour chacun des packages, un certain d'informations pour l'annotation des modèles sont définis. Néanmoins, ces différentes informations sont définies indépendamment du concept

de qualité de service défini dans le package GRM. C'est en partie ce problème que le profil QoS tente de corriger. Nous ne nous attarderons donc pas sur la description du contenu de ces packages.

1.3 Classification des ressources

Dans le package GRM, une classification des ressources selon trois critères est proposée : fonction (purpose), activité (activeness), et protection. Toute ressource peut être classée selon ces trois critères.

▶ **Fonction** Une ressource peut être caractérisée par l'un de ces trois types de fonction :
- Processor : Ce type de fonction permet d'identifier qu'une ressource est capable de stocker et d'exécuter le code d'un programme. Il peut bien évidemment s'agir d'une ressource de calcul matérielle telle qu'un CPU, mais aussi par exemple d'une ressource logicielle, telle qu'une machine virtuelle.
- Communication : Ce type de fonction permet d'identifier qu'une ressource a pour rôle de jouer un support pour la mise en œuvre des communications (typiquement des requêtes de services) entre d'autres ressources. D'un point de vue matérielle, un bus serait typiquement classé dans cette catégorie. D'un point de vue logiciel, une FIFO ou encore un sémaphore rentrerait dans cette catégorie.
- Device : Ce type de fonction englobe toutes les ressources n'entrant pas dans la catégorie Processor ou Communication, tels que capteurs, moteurs, disques,...

▶ **Activité** Une ressource peut être caractérisée par l'un de ces deux types d'activité :
- Active : La ressource est capable d'initier des communications avec son environnement sans avoir été "stimulée" par une autre ressource.
- Passive : La ressource n'est pas capable de générer des communications, et ne peut que réagir à des communications émises par son environnement.

▶ **Protection** Une ressource peut être caractérisée par l'un de ces deux types de protection :
- Protected : La ressource offre des services "exclusifs" à son environnement. Cela signifie que les accès concurrents aux services offerts par la ressource sont régis par une politique de gestion des accès.
- Unprotected : Les services offerts par la ressource ne sont soumis à aucune politique de gestion des accès.

1.4 Bilan

Si le mécanisme d'annotation des modèles semble obsolète par rapport à ce que propose le profil QoS, le profile SPT propose néanmoins une classification intéressante des ressources. Comme nous l'avons évoqué dans la section consacrée à la description de l'environnement Gaspard (chapitre introductif de ce document), la prise en compte du support d'exécution matériel est essentielle pour la conception d'un système embarqué. L'idée consiste donc à s'inspirer de cette classification pour la définition de notre modèle d'architecture matérielle, en apportant quelques évolutions, notamment pour la prise en compte de

la hiérarchie liée à l'utilisation d'une méthodologie orientée composant. Par ailleurs, l'approche de type client/serveur pourra être conservée comme base du modèle de communication des éléments matériels composant une architecture.

2 QoS

Le profil QoS (UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms) [49], adopté en 2004, définit un ensemble d'extensions permettant la modélisation de concepts de qualité de service et de tolérance aux pannes, et permet d'enrichir les modèles via un mécanisme d'annotation. La définition du concept de qualité de service est héritée du profil SPT. Comme nous l'avons évoqué dans la section précédente, le concept de QoS reste dans le profil SPT un concept abstrait. Les différents types de qualité de service introduits dans les packages d'analyse d'ordonnançabilité et de performance n'ont pas de lien direct avec les concepts de qualité de service. Le profil QoS va d'une certaine façon au bout de la démarche initiée dans le profil SPT, en proposant un framework de définition des qualités de service, permettant d'instancier et donc d'utiliser concrètement les concepts définissant la notion de qualité de service. Ce framework est illustré en fig.3.3.

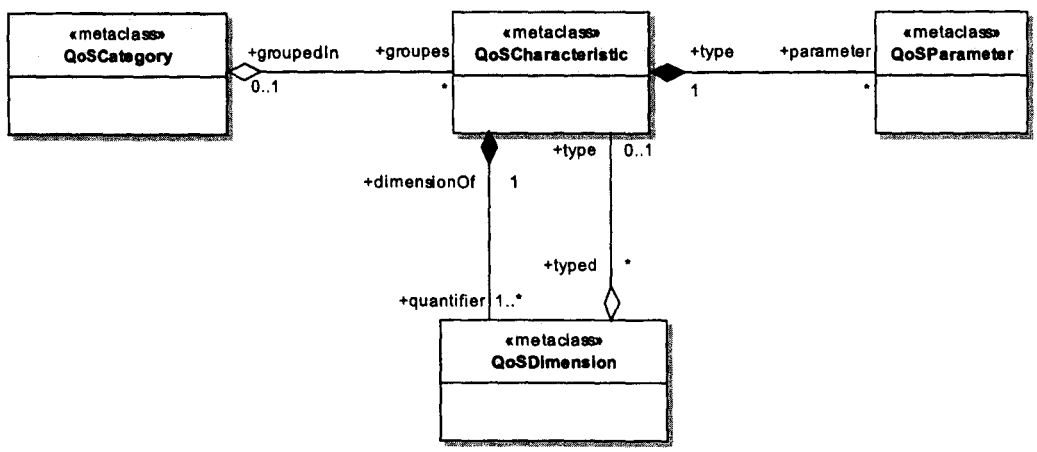


FIG. 3.3 – Framework de définition de qualités de service (diagramme simplifié)

2.1 Framework de définition des QoS

Les caractéristiques QoS (`QoSChareristic`) représentent des caractéristiques quantifiables des services. Elles sont définies indépendemment des éléments qu'elles vont caractériser, et constituent en ce sens des entités réutilisables. La spécification du profil QoS propose d'ailleurs un certain nombre de caractéristiques QoS, définies sous forme de catalogues. Le concept de `QoSCharacteristic` est une forme de constructeur pour la description d'aspects non fonctionnels d'un système, tels que latence, disponibilité, ou en-

core fiabilité. Ce concept supporte des mécanismes d'extensions, simplement par la possibilité de définir des liens d'héritage entre caractéristiques. Ce mécanisme encourage la réutilisation de caractéristiques existantes, et permet de factoriser la définition de nouvelles caractéristiques. Les caractéristiques peuvent par ailleurs être paramétrées, via un mécanisme de templates. Les unités ou les types des valeurs considérées dans une caractéristique sont des exemples typiques de paramètres. Le concept de `QoSDimension` permet quant à lui de spécifier des attributs pour les `QoSCharacteristics`, pour la définition de caractéristiques complexes. Comme illustré en fig.3.3, une caractéristique possède au moins une dimension, mais peut en posséder plusieurs. A noter que l'attribut `direction : DirectionKind` (énumération proposant les valeurs '<', '>' ou `'undefined'`) du concept `QoSDimension` permet de définir une relation de qualité pour la comparaison de deux valeurs différentes d'une dimension. Par exemple, pour la relation '<', la qualité d'une valeur x pour une dimension par rapport à une valeur y est meilleure lorsque x est inférieur à y. Enfin, le concept de `QoSCategory` permet de regrouper la définition des caractéristiques QoS par domaines de caractérisation. Dans le catalogue de caractéristiques QoS proposé dans la spécification, un certain nombre de catégories générales sont proposées, telles que `Performance`, `Dependability`, ou encore `Integrity`.

2.2 Méthodologie d'annotation des modèles

Dans [57], une méthodologie en trois étapes (extraite de la spécification du profil QoS) est proposée. Les exemples présentés par la suite sont tirés de cette référence, où les mécanismes d'annotation QoS sont utilisés pour caractériser le modèle d'un système robotisé. La première étape consiste à définir les caractéristiques de qualité de service spécifiques à un domaine et/ou à un type d'analyse, en les groupant par catégories. Ces nouvelles caractéristiques sont typiquement définies sur la base des caractéristiques proposées dans le catalogue général de la spécification du profil QoS, mais peuvent également être définies indépendemment des caractéristiques de base (fig.3.4).

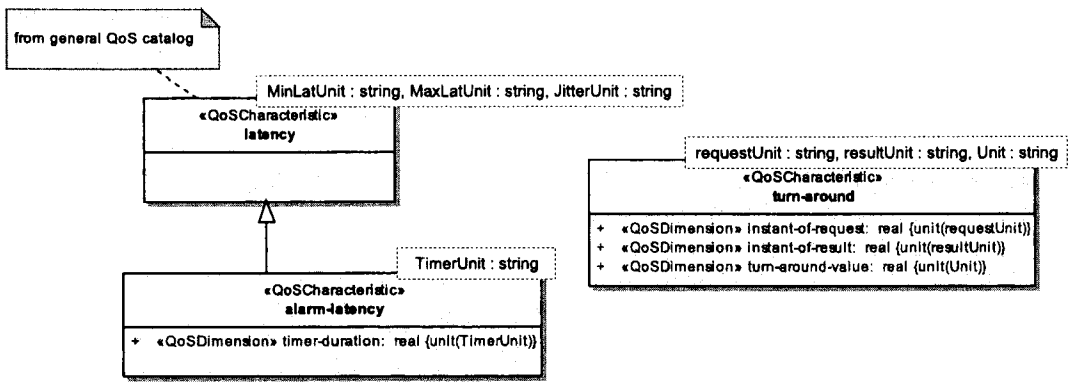


FIG. 3.4 – Définitions de caractéristiques QoS

La deuxième étape consiste en la définition d'un "modèle de qualité" plus ou moins

spécifique à l'étude de cas en cours, dans lequel tous les paramètres des classes templates définissant les caractéristiques QoS sont fixés (fig.3.5). Le préfixe "PA4AS" est lié à l'exemple, et signifie "Performance Analysis for Automation System".

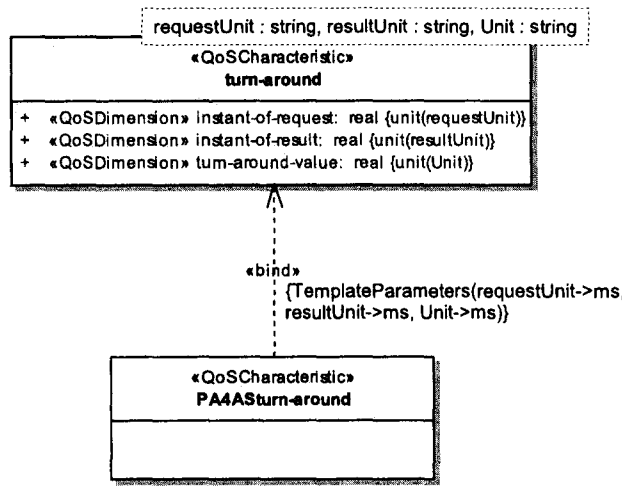


FIG. 3.5 – Définitions d'un "modèle de qualité" spécifique

Enfin, les modèles UML sont annotés avec les contraintes et les valeurs de QoS relatives aux caractéristiques QoS définies dans les étapes précédentes. Différentes approches sont proposées dans la spécification pour la mise en oeuvre de cette étape. Une approche consiste à attacher des notes aux éléments de modèles caractérisés, en y décrivant les valeurs de QoS correspondant à la caractéristique QoS visée. Une autre approche consiste à connecter l'élément caractérisé à une instance de classe stéréotypée «QoSValue». La classe instanciée est elle-même stéréotypée «QoSCharacteristic», et ses attributs sont stéréotypés «QoSDimension». L'instance de la classe spécifie les valeurs d'instance des attributs stéréotypés «QoSDimension».

2.3 Bilan

Le profil QoS offre un mécanisme flexible pour la définition de qualités de service et l'annotation des modèles. Dans l'environnement Gaspard, du point de vue de l'utilisateur, l'architecture matérielle peut être vue comme un ensemble de ressources offrant des services pour supporter l'exécution d'une application : services de calcul (processeurs), services de stockage de données (mémoires),... Pour définir un placement pertinent d'une application sur une architecture, ou encore pour dimensionner correctement une architecture afin de supporter l'exécution d'une application, il est important de connaître les caractéristiques requises par l'application (par exemple un délai de 5 ms pour l'exécution d'une tâche), et les caractéristiques fournies par l'architecture pour le support de cette application (tel processeur est capable de réaliser cette même tâche en 6ms alors que tel autre est capable de la réaliser en 3 ms). Pour la mise en oeuvre de ce processus de caractérisation, et bien que le catalogue de qualité de service fourni dans la spécification du profile ne corresponde pas

vraiment à nos attentes, l'idée consiste à réutiliser les mécanismes offerts par le profil QoS, l'objectif étant d'orienter le processus de placement d'une application sur une architecture matérielle.

3 SysML

SysML (System Modeling Language) [50] est une proposition de langage généraliste en cours de standardisation à l'OMG offrant un support de modélisation pour l'ingénierie des systèmes, et proposée en réponse à la RFP UML pour l'ingénierie des systèmes (UML for System Engineering [47]). Le spectre de systèmes couvert par SysML se veut très large, les systèmes concernés pouvant inclure à la fois du matériel, du logiciel, des personnes, des processus,... L'objectif est de proposer des mécanismes permettant de modéliser les caractéristiques attendues d'un système, sa structure, son comportement fonctionnel, ou encore des processus d'allocations. La définition de SysML repose sur la superstructure d'UML 2, l'objectif étant de minimiser les modifications apportées au métamodèle UML, tout en répondant de façon pertinente aux requêtes de la RFP. L'architecture de SysML est illustrée en fig.3.6.

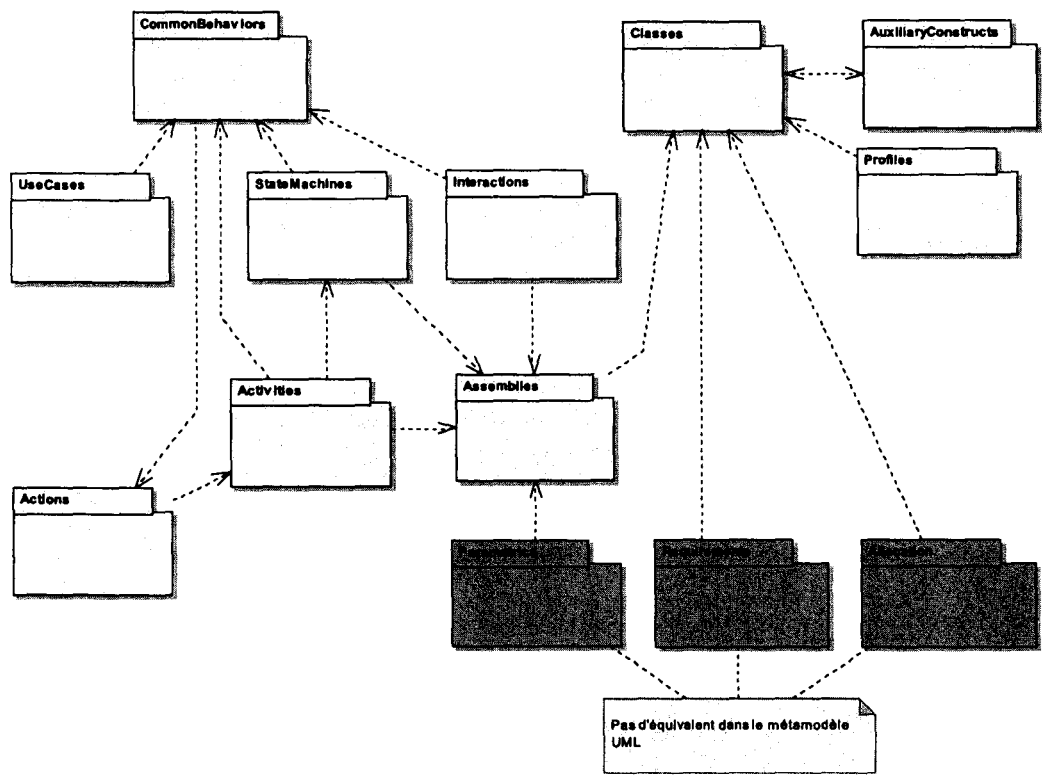


FIG. 3.6 – Architecture de SysML

3.1 SysML vs UML 2

Comme nous l'avons déjà évoqué, l'approche de conception de SysML est de réutiliser un sous-ensemble d'UML 2, et de définir des extensions pour les requêtes spécifiques de la RFP UML for SE. En fig.3.7, on peut constater que l'architecture de SysML est globalement alignée par rapport à l'architecture d'UML. Certains packages, tels que `Components` et `Deployment`, ne sont pas réutilisés dans le sens où ils ne sont pas indispensables pour répondre aux requêtes de la RFP. En effet, la RFP ne mentionne pas la nécessité de manipuler des artefacts (qui constituent basiquement la seule différence entre classes structurées et composants. Nous reviendrons sur ce point dans la section consacrée à la description du profil Gaspard.), et les mécanismes de déploiement du package `Deployment` sont inclus dans le package `Allocation`. D'autres packages, tels que `StateMachines`,`Activities`, ou encore `Interactions`, sont réutilisés sans modifications. Quelques extensions, plus (au niveau des activités) ou moins (au niveau des classes) importantes, sont introduites dans les packages `Activities` (par exemple la possibilité de manipuler les tokens de contrôle comme des tokens de données, ou encore la possibilité de modéliser des systèmes continus), `Classes` (par exemple la possibilité de regrouper les dépendances par ensembles, ou encore la possibilité de visualiser au niveau d'une classe toutes les classes dont elle hérite directement ou indirectement afin de mettre en valeur les différents niveaux de spécialisation qui ont conduit à la définition de cette classe) et `AuxiliaryConstructs`. Le package `Assemblies` réutilise les classes structurées du package `CompositeStructures` d'UML, avec quelques extensions mineures. Les packages `Requirements` (pour la modélisation des caractéristiques attendues d'un système), `Parametrics` (pour la modélisation de "relations paramétriques" entre éléments d'un système), et `Allocation` (nous nous attardons sur la description de ce package dans la section suivante) sont introduits.

3.2 Allocation

Dans SysML, `Allocation` désigne à la fois un concept très générique, et un mécanisme permettant de définir des liens entre des éléments de différents modèles, de différents types, ou encore de différents niveaux de hiérarchies, à un niveau très abstrait. L'allocation est vue comme une forme d'étape intermédiaire dans la conception d'un système, utilisée pour garantir la cohérence d'un modèle ou d'un ensemble de modèles en cours de définition, et pour orienter les étapes à venir dans la conception du système. Dans la spécification de SysML, il est clairement mis en avant que le fait qu'une relation de type `Allocation` soit exprimée entre deux éléments de modèle donnera lieu à un type de relation plus concret (que ce soit au niveau des attributs de ces éléments, de leurs propriétés, ou encore de leurs opérations).

Concrètement, le concept de `Allocation` est mis en oeuvre via la définition d'un stéréotype «Allocation» (fig.3.8) s'appliquant aux dépendances UML de type `Usage` (définies dans le package `Dependencies`), et peut être exprimée entre deux "éléments nommés" (NamedElement), ce qui donne une idée du niveau de généricité auquel ce concept est défini. Les relations de type allocation sont orientées, un des éléments nommés jouant le rôle de "client" dans la relation, et l'autre jouant le rôle de "fournisseur". Dans UML 2, les dépendances de type `Usage` sont utilisées pour spécifier le fait que le client requiert la présence du fournisseur pour la mise en oeuvre de son comportement, ou encore qu'un élément requiert la présence d'un autre élément pour son implantation. Un certain nombre

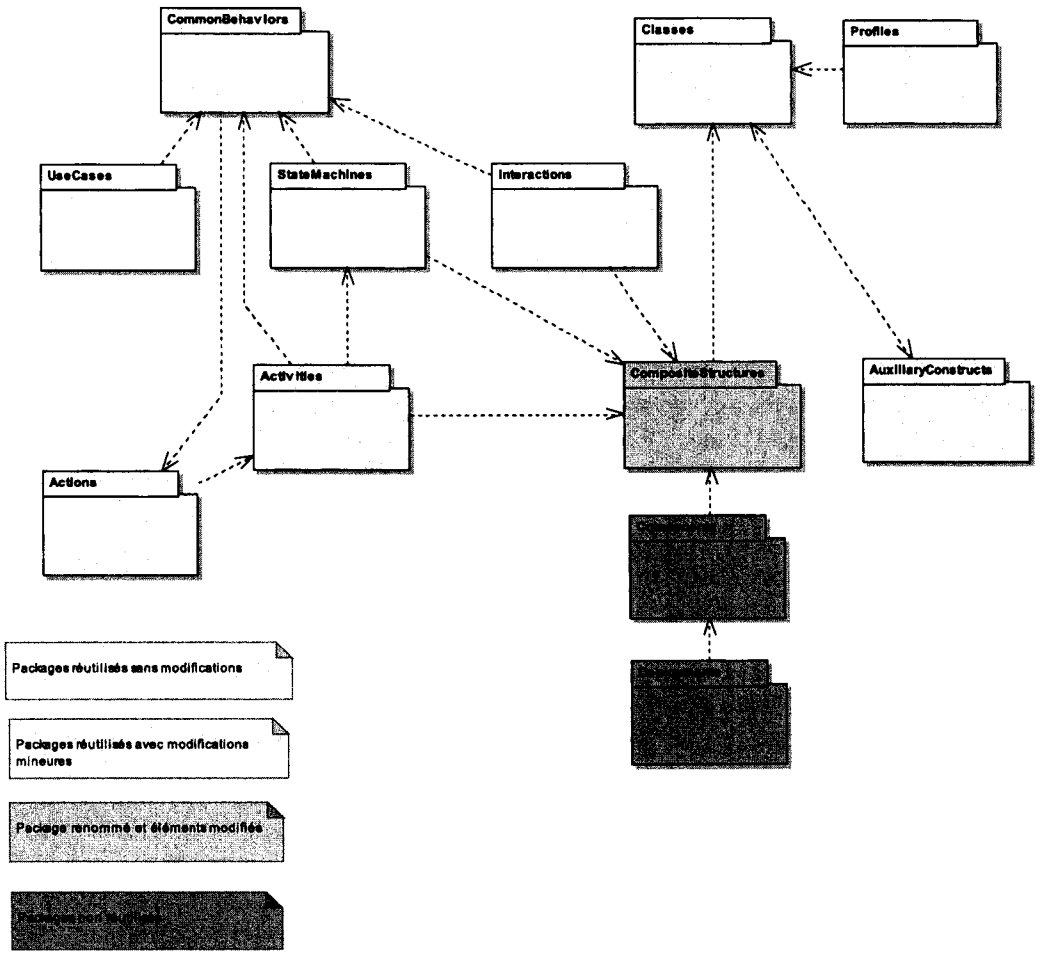


FIG. 3.7 – SysML vs UML 2 : Réutilisation des packages

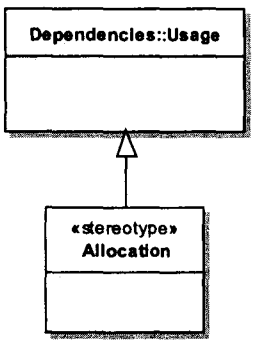


FIG. 3.8 – Mise en oeuvre du concept d'Allocation

de raffinements du concept de `Allocation` est proposé :

- **Requirement Allocation** : pour l'allocation de `Requirement` (ce concept est propre à SysML, et est utilisé pour définir une caractéristique attendue d'un élément) à différents éléments d'un système.
- **Functional Allocation** : pour l'allocation d'un élément fonctionnel (ou comportemental) à un élément structurel. Ce mécanisme permet une séparation des préoccupations dans la conception d'un système, avec d'un côté la définition d'un point de vue fonctionnel, et de l'autre côté un point de vue structurel.
- **Flow Allocation** : pour l'allocation de flots dans la vue fonctionnel à des flots dans la vue structurelle d'un système.
- **Structural Allocation** : pour l'allocation d'éléments structurels à d'autres éléments structurels. Typiquement, des éléments représentants des entités "logiques" sont alloués à des entités "physiques".
- **Property Allocation** : pour l'allocation d'informations de performance à différents éléments d'un système.
- **Deployment Allocation** : pour l'allocation d'éléments logiciels sur des éléments matériels (il s'agit d'un cas particulier d'allocation structurelle).

3.3 Bilan

SysML est globalement aligné sur UML 2. Il en réutilise une partie, et introduit un certain nombre de nouveaux concepts. Parmi ces concepts, le plus intéressant pour nous semble être celui de `Allocation`, bien que l'on puisse s'interroger sur l'intérêt d'utiliser le terme "allocation" pour couvrir un champ d'utilisation aussi large. Plus précisément, l'idée serait de réutiliser les allocations de type `Deployment` pour la modélisation dans l'environnement Gaspard du placement d'une application sur une architecture matérielle. Sur le fond, la notion d'allocation de déploiement est relativement proche du concept UML de `Deployment`. Néanmoins, UML 2 ne permet en résumé que d'allouer des artefacts logiciels à des noeuds (`Node`). Le concept de `Deployment Allocation` offre plus de flexibilité, et nous montrerons comment nous tirons partie de cette propriété dans la section décrivant le modèle

d'association utilisé dans le profil Gaspard.

4 Uml for SoC

Le profil UML for SoC [51], en cours de standardisation à l'OMG, est le résultat d'un processus différent de celui évoqué pour les précédents profils que nous avons présenté. Les profils SPT, QoS, et SysML sont des réponses à des RFP. La proposition d'une RFP pour la définition d'un profil est motivée par le fait que l'on estime que UML n'est pas adapté pour la modélisation de certains types de problèmes, dans certains domaines d'activités (par exemple pour les systèmes temps réel),... Une RFP définit donc un ensemble de requirements pour adapter UML à ces types de problèmes ou de domaines d'activité, éventuellement classée par ordre de priorité, auquel toute proposition devra répondre. Dans le cas de UML for SoC, aucune RFP n'a été rédigée. Le processus de standardisation de ce profil est ainsi identifié sous le terme de RFC (Request For Comment). Les organismes souhaitant soumettre (en l'occurrence Fujitsu, IBM et Nec) proposent une solution. Si cette solution est unanimement validée, la proposition devient un standard.

4.1 Extensions pour la modélisation structurelle des SoCs

Le profil Uml for SoC propose un certain nombre d'extensions sous la forme de stéréotypes et de représentations visuelles particulières pour la modélisation de systèmes sur puce en UML. Ces extensions visent particulièrement les éléments manipulés dans les diagrammes de structure, renommés pour l'occasion "SoCStructureDiagram". Le profil ne prend en compte que la modélisation structurelle, les aspects comportementaux étant négligés ou dans certains cas suggérés par les informations associés aux éléments structurels. Le vocabulaire utilisé pour la définition des différents stéréotypes est principalement inspiré des langages utilisés pour la programmation au niveau système, tels que SystemC [54] ou encore SpecC [22]. La spécification fournit d'ailleurs des exemples de projection de spécifications UML basées sur ce profil vers le langage SysML. Néanmoins, le spectre de langages cibles couvert par le profil est plus vaste que SystemC seul, une projection vers des langages tels que VHDL [27] étant également envisagée dans la spécification du profil. Quoi qu'il en soit, les utilisateurs familiers de SystemC retrouveront les concepts qu'ils ont l'habitude de manipuler, tels que `Module`, `Channel` ou encore `Clock` et `Reset`.

4.2 Bilan

La lecture de la spécification du profil UML for SoC provoque un sentiment mitigé quant à l'opportunité de sa standardisation. D'une part, le nom du profil peut sembler présomptueux dans la mesure où seuls les aspects structurels de la modélisation des systèmes sur puce sont pris en compte. D'autre part, le profil semble clairement orienté pour un type de technologie (les langages de programmation niveau système), et l'on a l'impression que le seul bénéfice lié à l'utilisation du profil est la possibilité de projeter facilement un modèle vers un langage qui offrira un véritable pouvoir d'expression. A priori, le rôle de l'OMG n'est pas de standardiser des profils pour des technologies particulières, mais plutôt de définir des profils permettant d'enrichir la sémantique d'UML, éventuellement pour pouvoir appliquer différentes techniques d'analyse aux modèles. C'est effectivement

le cas pour les profils SPT, Qos et SysML. Cependant, la spécification pouvant se résumer à une énumération des différents stéréotypes introduits, il n'est pas évident d'évaluer le potentiel du profil, d'autant plus que la relative discrétion qui l'entoure fait que peu d'exemples concrets d'utilisation sont disponibles. Quoi qu'il en soit, en l'état actuel de nos connaissances quant à ce profil, on ne retiendra aucun élément majeur pouvant être réutilisé pour la définition du profil Gaspard.

CHAPITRE 4

Le profil Gaspard

1 Architecture du profil 60
2 Package "component" 61
2.1 Vue domaine 61
2.2 Vue UML 65
3 Package "factorization" 76
3.1 Vue domaine 76
3.2 Vue UML 81
4 Package "hardwareArchitecture" 88
4.1 Vue domaine 88
4.2 Vue UML 99
4.3 Impact des mécanismes de factorisation 101
5 Package "application" 105
5.1 Vue domaine 105
5.2 Vue UML 107
5.3 Impact des mécanismes de factorisation 108
6 Package "association" 112
6.1 Vue domaine 112
6.2 Vue UML 116
6.3 Impact des mécanismes de factorisation 116

Dans le chapitre introductif, nous avons identifié quels étaient les objectifs, ou lignes directrices, des travaux présentés dans cette thèse :

- **la définition d'une méthodologie orientée composant,** afin d'offrir un support pour la définition et la réutilisation d'IPs logicielles ou matérielles.
- **la définition de mécanismes communs de factorisation,** pour l'expression sous une forme compacte du parallélisme potentiel des applications, du parallélisme disponible des architectures et de la topologie des connections entre composants, et du placement régulier des applications sur les architectures matérielles.
- **une séparation claire entre application et architecture matérielle,** afin de faciliter et d'encourager la réutilisation des modèles, et de garantir leur lisibilité.
- **la définition d'un modèle de description d'architectures matérielles,** permettant de dimensionner les ressources matérielles supportant l'exécution d'une application.
- **la définition d'un modèle d'association,** permettant de donner des directives de placement d'une application sur une architecture.
- **la définition de mécanismes de caractérisation,** visant à identifier des caractéristiques relatives à l'exécution d'une application (ou d'un composant applicatif) sur une architecture matérielle (ou un composant matériel), de façon à pouvoir orienter un placement manuel ou automatique de l'application sur l'architecture.

Sur la base de l'état de l'art réalisé dans les chapitres 2 et 3 , nous allons donner dans ce chapitre la définition du profil Gaspard, sous une forme fortement inspirée par la structure du profil SPT. Le profil est structuré sous forme de packages, chaque package capturant un domaine de préoccupations particulier, avec une architecture globalement alignée sur les objectifs précédemment évoqués. Pour chaque package, une "vue domaine" (simplification des métamodèles) définit les objectifs sous-jacents, les différents concepts requis pour la mise en oeuvre de ces objectifs, et les règles de construction qui régissent leur utilisation. Une "vue UML" est ensuite proposée, définissant une équivalence entre les différents concepts et la manière de les représenter en UML, soit par la définition de stéréotypes avec leurs tagged values associées, soit par la définition de contraintes et/ou de patterns de modélisation. Pour la définition des différents packages, nous basons notre proposition sur les éléments que nous avons jugé pertinents dans l'état de l'art, et définissons nos propres extensions sur cette base :

- **UML 2** pour la mise en oeuvre de l'**approche orientée composant,**
- **ArrayOL** pour la mise en oeuvre de **mécanismes communs de factorisation**
- **SPT** pour la mise en oeuvre de notre **modèle d'architecture matérielle,**
- **QoS** pour la mise en oeuvre des **caractérisations,**
- **SysML** pour la mise en oeuvre **modèle d'association.**

1 Architecture du profil

Le profil Gaspard s'articule autour de cinq packages (fig.4.1). Nous retrouvons dans la hiérarchie de packages les trois aspects essentiels du modèle Y de Gaspard : `application`, `hardwareArchitecture`, et `association`. La philosophie générale de cette hiérarchie est de définir un maximum de parties communes pour les différents aspects du Y. Ainsi, les packages `application` et `hardwareArchitecture` partagent la même définition du concept de composant, introduite dans le package `component`. Le package `association` introduit des concepts permettant de donner des directives de placement d'un modèle d'application sur un modèle d'architecture. La grande particularité de cette vue domaine se situe au niveau du package `factorization`, où sont introduits des mécanismes de factorisation structurelle inspirés par Array OL. Les packages `component` et `association` utilisent les mécanismes de factorisation définis dans le package `factorization`. Nous verrons que les extensions les plus lourdes sont introduites au niveau du package `factorization`, UML n'offrant quasiment pas de moyens pour décrire la topologie des liens entre éléments multiples.

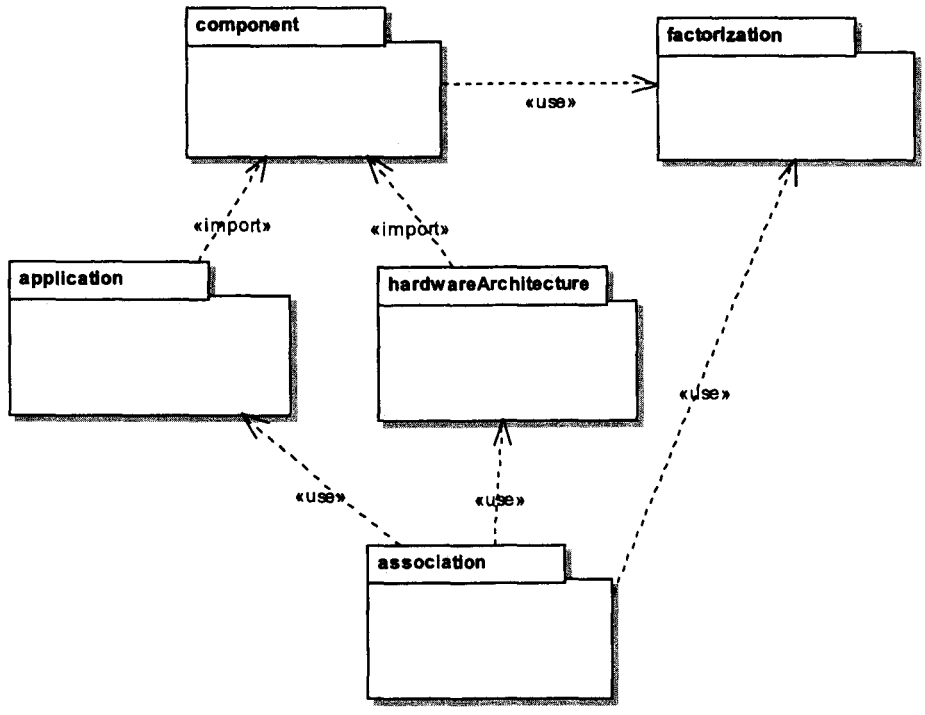


FIG. 4.1 – Architecture du profil Gaspard

2 Package "component"

2.1 Vue domaine

Le package `component` pose les bases de l'approche orientée composant sous-jacente au profil Gaspard, l'objectif étant de favoriser autant que possible la réutilisation. Le package `component` n'introduit pas de concepts originaux, mise à part quelques ajouts mineurs. Il s'agit simplement d'une simplification et d'une restructuration des mécanismes de définition de composants proposés dans UML 2, regroupés dans un seul package. L'objectif du package est de définir un support pour une méthodologie à base de réutilisation d'IPs logicielles ou matérielles, en permettant la mise en oeuvre de différents mécanismes :

- **L'encapsulation**, pour rendre la définition d'un composant indépendante de l'environnement dans lequel le composant est utilisé. Une frontière claire est tracée entre l'"implantation" du composant (basiquement sa structure), et la façon dont il perçoit son environnement, ou la façon dont il est perçu par son environnement.
- **La composition et l'assemblage**, pour permettre de structurer hiérarchiquement la définition d'un composant. Un composant peut être composé d'autres composants offrant des fonctionnalités plus simples. L'assemblage de ces composants peut permettre de mettre en oeuvre des comportements plus complexes.
- **Le paramétrage**, pour permettre de définir des IPs paramétrables, pratique courante chez les fournisseurs d'IPs.

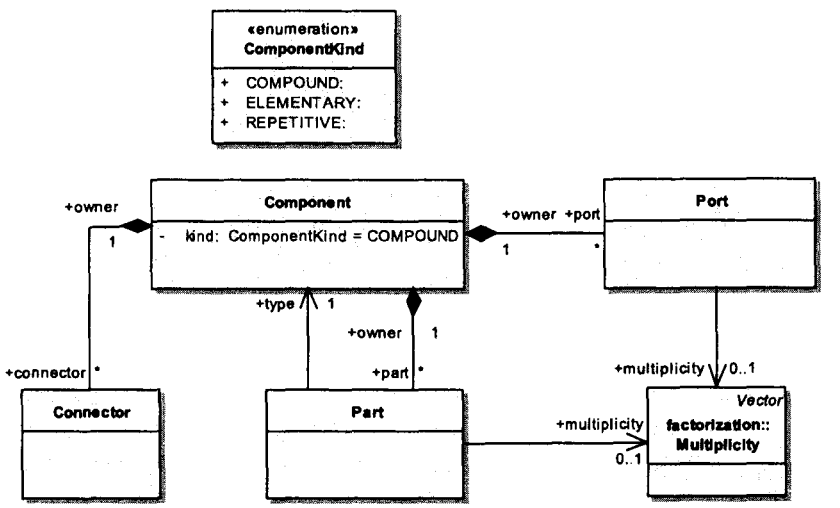


FIG. 4.2 – Component : Vue d'ensemble

Le concept de `Component` (fig.4.2) constitue l'élément de base réutilisable manipulé dans le contexte du profil Gaspard. Un composant est défini par un ensemble d'interfaces exposées via des `Port`, et par une structure constituée d'un assemblage de `Part` via des `Connector`. Les interfaces des composants (et par extension les ports auxquels elles sont rat-

tachées) constituent les seuls points d'interactions possibles des composants avec l'environnement dans lequel ils sont instanciés. Les parts assemblées dans la structure du composant sont susceptibles de collaborer pour mettre en oeuvre des comportements plus complexes. L'attribut `kind : ComponentKind` permet de donner des informations sur la structure du composant. Par défaut, un composant est dit `COMPOUND`. Ce cas correspond au cas général précédemment mentionné, où la structure du composant est définie par un assemblage d'un nombre quelconque de parts. Dans le cas dit `REPETITIVE`, la structure du composant ne contient qu'une seule part, directement connectée aux différents ports d'entrée ou de sortie du composant la contenant. Sont associées à la part et aux différents connecteurs des informations de "répétition". Ce point sera discuté plus en détail dans la section consacrée aux mécanismes de factorisation. Dans le cas dit `ELEMENTARY`, la structure du composant n'est pas décrite. Le composant ne contient ni part, ni connecteur, et sa description est supposée être disponible sous la forme d'une ou plusieurs implantations, dans différents langages cibles. La structuration des modèles sous forme de packages permet d'identifier l'endroit où doivent être définies les implantations des composants élémentaires. Le concept de `Package` permet de structurer les modèles, d'organiser la définition des composants, et ainsi de favoriser la réutilisation. Un package constitue un espace de nommage pour les éléments définis à l'intérieur de ce package.

Par la suite, le concept de `Component` sera raffiné en `ApplicationComponent`, ou composant applicatif, et `HardwareComponent`, ou composant matériel. Pour chacun de ces deux raffinements, une sémantique plus précise sera donnée.

2.1.1 Encapsulation

Encapsuler la définition d'un composant revient à rendre un composant indépendant de l'environnement dans lequel il est utilisé, afin de faciliter et d'encourager sa réutilisation.

▶ **Port** Le concept de `Port` permet d'identifier clairement les points d'encapsulation des composants. Lorsqu'un composant est instancié dans un environnement quelconque (c'est à dire en tant que part dans la structure d'un autre composant), les ports constituent le seul moyen pour lui d'interagir avec son environnement. L'interaction est mise en oeuvre via la réception et/ou l'émission de signaux et/ou d'invocations d'opérations. "L'orientation" du port est déterminée par l'orientation de l'interface qui lui est associée. Lorsque l'interface est `PROVIDED`, le composant fournit cette interface à son environnement, et est en mesure de recevoir les signaux ou invocations d'opérations relatifs à cette interface. Lorsque l'interface est `REQUIRED`, le composant est en mesure d'émettre les signaux ou invocations d'opérations relatifs à cette interface. Selon la nature des requêtes (signal ou invocations d'opérations) associées aux interfaces, l'interprétation de l'orientation du port peut être différente. Dans le cas d'invocations d'opérations, un port ayant une interface `REQUIRED` s'interprète comme un port "maître", et un port ayant une interface `PROVIDED` s'interprète comme un port "esclave" : Le maître peut initier la requête en demandant un service fourni par un esclave. Dans le cas de signaux, un port ayant une interface `PROVIDED` peut être vu simplement comme un port d'entrée et un port ayant une interface `REQUIRED` peut être vu comme un port de sortie : Une requête sans attente bloquante de retour est initiée par le composant ayant le port `REQUIRED`, via l'émission d'un signal à l'environnement, et est reçu par un port ayant une interface `PROVIDED` permettant la réception de ce type de signal.

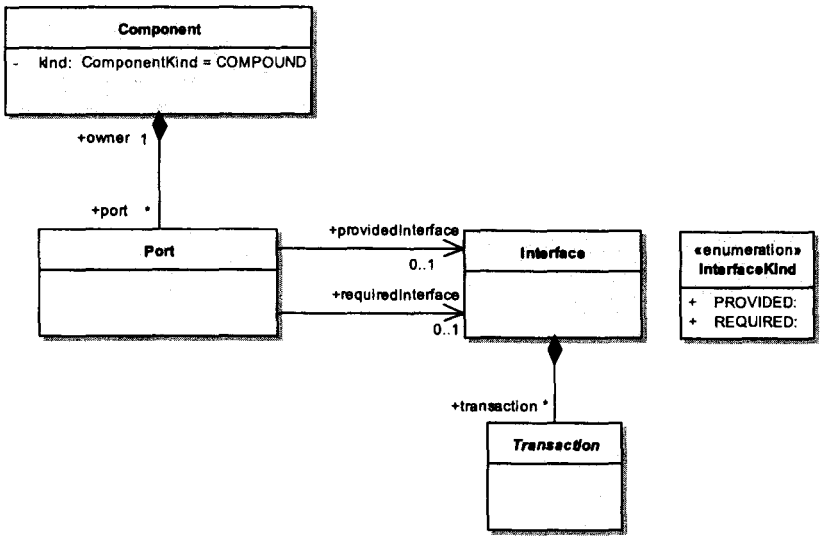


FIG. 4.3 – Component : Mécanismes d'encapsulation

▶ **Interface** Une interface représente une forme de contrat, et est décrite, comme nous l'avons déjà brièvement évoqué, par un ensemble de requêtes. Le concept abstrait de `Request` permet de définir le prototype des évènements supportés par une interface. Une interface permet d'identifier les requêtes pouvant être initiées ou subies par un composant via un port, en fonction du rôle qui lui a été donné (`PROVIDED` ou `REQUIRED`). Les interfaces permettent en particulier de garantir la validité d'une connexion entre deux composants.

2.1.2 Composition et Assemblage

Dans la section précédente, nous avons présenté les mécanismes permettant d'isoler la structure du composant de l'environnement dans lequel il est utilisé. Nous allons maintenant montrer comment la structure d'un composant peut être décrite, via un ensemble de parts, reliées entre elles et reliées aux ports du composant englobant par des connecteurs.

▶ **Part** Une `Part` représente une instance potentielle de composant, et peut être utilisée pour définir la structure d'un autre composant. Les parts définissant la structure d'un composant sont assemblées entre elles, via des connecteurs entre leurs ports, pour leur permettre de collaborer et éventuellement de mettre en oeuvre des comportements complexes.

▶ **Connector** Un `Connector` représente un lien de communication entre deux parts, via leurs ports, permettant à ces deux parts d'échanger des signaux et/ou des requêtes. Une connection entre deux ports est possible uniquement si les deux ports possède la même interface, et si d'un côté l'interface est `REQUIRED` (l'émetteur de la requête) et de l'autre côté `PROVIDED` (le récepteur de la requête).

On distingue basiquement deux cas d'utilisation des connecteurs. Dans le premier cas, le connecteur est exprimé entre deux parts, via leurs ports. Chacun des deux `ConnectorEnd` définissant le connecteur possède une référence sur la part concernée, et une référence sur le port connecté. Le port connecté est nécessairement inclus dans l'ensemble des ports appartenant au composant typant la part. Dans le second cas, le connecteur est exprimé entre une part, via l'un de ses ports, et un port en bordure du composant contenant cette part. Le `ConnectorEnd` correspondant au port en bordure du composant ne fait simplement référence à aucune part. Ce cas correspond à une délégation des requêtes émises/initiées depuis/vers l'environnement du composant, via ses ports.

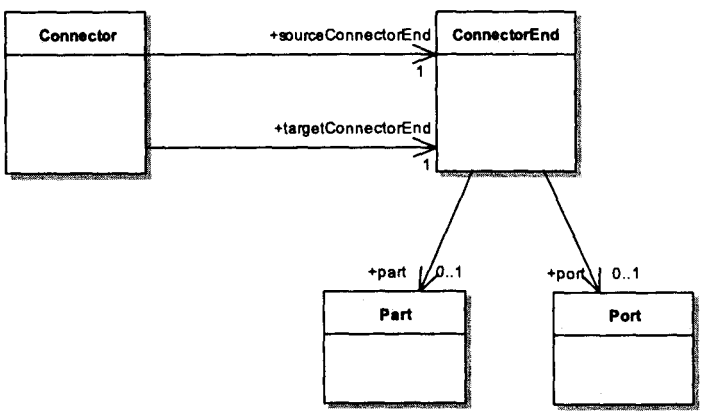


FIG. 4.4 – Connector : Définition

2.1.3 Paramétrages

Les composants, tels que nous les avons présentés jusqu'à maintenant, ne peuvent être modélisés que d'une façon complètement statique. Les concepts proposés dans cette section (qui n'existent pas dans UML 2) vont nous permettre de définir des composants paramétrables, pratique courante chez les fournisseurs d'IPs, via la possibilité d'associer des paramètres aux composants, et de fixer des valeurs d'instance de ces paramètres lors de la définition de parts dans la structure des composants.

▶ **Parameter** Le concept de `Parameter` permet d'associer à un composant un ensemble de paramètres. Un paramètre est la représentation formelle d'une valeur numérique, et peut être référencé à tous les niveaux de la spécification d'un composant, c'est à dire aussi bien au niveau des ports qu'au niveau des connecteurs ou des parts. Les paramètres sont intimement liés à l'utilisation des mécanismes de factorisation, et seront à ce titre décrits plus en détail dans la section factorisation.

▶ **ParameterInstanceValue** Le concept de `ParameterInstanceValue` permet d'associer aux parts des valeurs d'instance pour les paramètres qui ont été définis au niveau

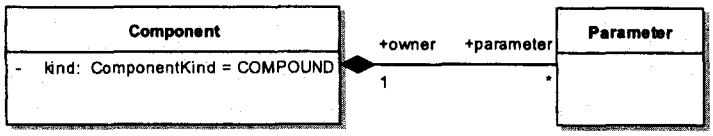


FIG. 4.5 – Parameter : Définition

des composants. L'attribut `value : Expression` représente concrètement la valeur d'instance du paramètre, et est définie sous la forme d'une expression représentant une valeur numérique entière.

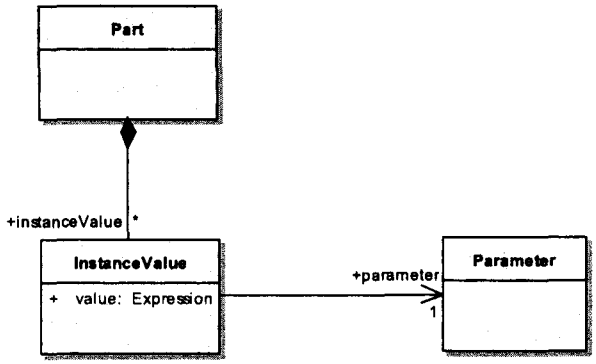


FIG. 4.6 – InstanceValue : Définition

2.2 Vue UML

Les sous-packages `InternalStructure` et `Ports` du pakage `Composite Structures` d'UML 2 (Superstructure, p.151) définissent pratiquement tous les éléments de base dont nous avons besoin pour modéliser des composants dans Gaspard. `InternalStructure` fournit les mécanismes de "composition et assemblage", via le concept abstrait de `StructuredClassifier` (fig.4.7), permettant la description de structures sous la forme d'éléments interconnectés, qui seront effectivement créés lorsque le classifier les contenant sera instancié. Une structure de ce type représente une décomposition de ce classifier, et est référencé sous le terme de "structure interne".

Le sous-package `Ports` fournit quant à lui les mécanismes "d'encapsulation", via le concept de `EncapsulatedClassifier` (qui étend `StructuredClassifier`, fig.4.8) permettant d'isoler un classifier de son environnement. Cette propriété est mise en oeuvre grâce à la possibilité de définir des points d'interactions entre la structure d'un classifier et l'environnement de ce classifier. Ces points d'interaction sont référencés sous le terme de port.

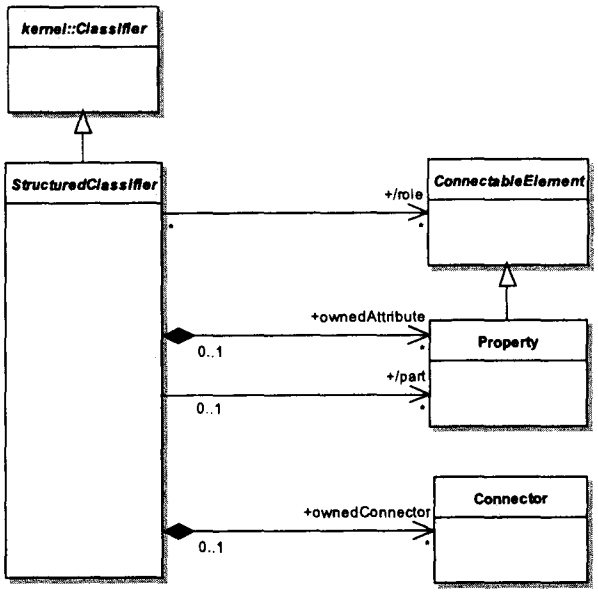


FIG. 4.7 – Structured classifier

Plusieurs ports peuvent être associés à un classifier, afin de pouvoir par la suite distinguer les interactions en fonction du port sur lequel elles ont lieu. En découplant la structure interne et l'environnement, les ports permettent la définition d'un classifier indépendemment de l'environnement dans lequel il sera instancié, facilitant sa réutilisation dans n'importe quel environnement conforme aux contraintes d'interactions imposées par ces ports.

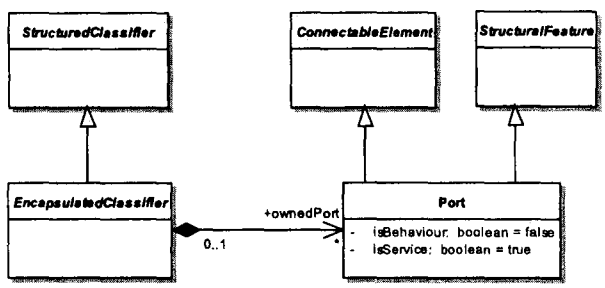


FIG. 4.8 – Encapsulated classifier

2.2.1 Composants

Dans le sous-package `StructuredClasses`, le concept de `Class` est spécialisé pour supporter concrètement ces mécanismes de composition, d'assemblage et d'encapsulation (fig.4.9).

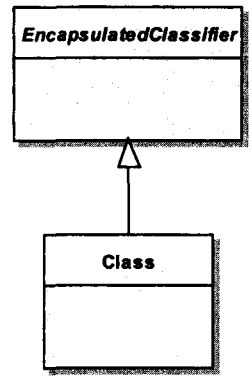


FIG. 4.9 – Class étendue avec support des structures internes

Le concept de `Class`, spécialisé dans le package `CompositeStructures`, suffirait amplement à supporter la définition du concept de `Component`, tel que nous l'avons donnée dans le chapitre précédent. La seule contrainte serait d'imposer l'utilisation de ports pour l'exposition des interfaces, et pour la mise en oeuvre des interconnexions entre parts. Le fait que le concept de `Part` n'apparaissent pas en tant que tel dans le métamodèle UML 2 (il apparaît en tant que rôle au niveau de l'association entre `StructuredClassifier` et `Property`) ne change rien fondamentalement, et tient au fait que le spectre de modélisation couvert par UML se veut par définition beaucoup plus large que celui de Gaspard. Dans le package `kernel` d'UML 2 où il est définit, `Property` est par exemple utilisé pour permettre de définir un attribut d'une classe, ou encore une extrémité non-navigable d'une association. Néanmoins, nous ne retiendrons pas le concept de `Class` comme projection du concept de `Component`. En effet, cela nous amènerait à introduire un stéréotype «`Component`» s'appliquant aux classes, alors que le concept de `Component` existe déjà dans UML (fig.4.10).

En tant que raffinement du concept de `Class` (tel qu'il est étendu dans le package `StructuredClasses`), le concept de `Component` permet de définir des composants encapsulés via des ports, et définis de façon hiérarchique par assemblage de parts via des connecteurs. La contrainte de modélisation que nous imposons est l'utilisation de ports pour l'exposition des interfaces et pour la mise en oeuvre des interconnexions entre parts.

Les aspects différenciant le concept de `Component` du concept de classe structurée ne sont pas pris en compte dans le profil. En particulier, les liens existant entre un composant UML et la façon dont il est implémenté (c'est à dire un ensemble d'artefacts) n'est pas considérée. Nous estimons en effet que cette information n'est pas pertinente à ce niveau, et la gérons implicitement au niveau de l'environnement Gaspard. La structuration des modèles sous forme de packages permet de définir implicitement l'endroit où

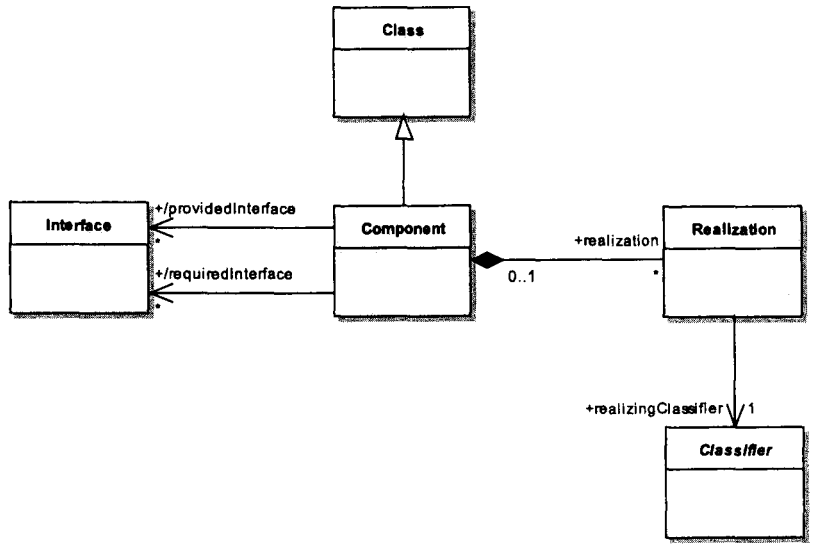


FIG. 4.10 – Composant UML 2

doivent être définies les implémentations des différents composants. S'il existe plusieurs implémentations pour un même composant (différents codages, langages cibles, ou niveaux d'abstraction), l'information est donnée explicitement au moment du déploiement. Pour caractériser les composants sans artefacts associés, la superstructure d'UML 2 introduit la notion de "composant logique" (logical component), par opposition à "composant physique" (physical component). Les composants, tels qu'ils sont définis dans le profil Gaspard, sont donc des composants logiques.

Comparativement aux composants gaspard, la seule différence se situe au niveau de l'attribut `kind : ComponentKind`. Pour la modélisation des composants, nous introduisons donc un stéréotype abstrait «**GaspardComponent**» (fig.4.11), s'appliquant au concept de `Component` d'UML. Dans les packages `hardwareArchitecture` et `application`, ce stéréotype sera respectivement raffiné en `HardwareComponent` et `ApplicationComponent`. Le stéréotype `GaspardComponent` ne sert qu'à structurer la définition des stéréotypes, et à factoriser la définition de l'attribut `kind : ComponentKind`.

2.2.2 Interfaces

Les interfaces, via les ports auxquels elles sont rattachées, constituent le seul moyen pour un composant de communiquer avec son environnement. Une interface permet d'identifier l'ensemble des interactions autorisées au niveau d'un port, pour la mise en oeuvre des communications entre composants. Avant de décrire la façon dont nous allons modéliser les interfaces des composants, intéressons nous aux mécanismes de communication définis dans UML.

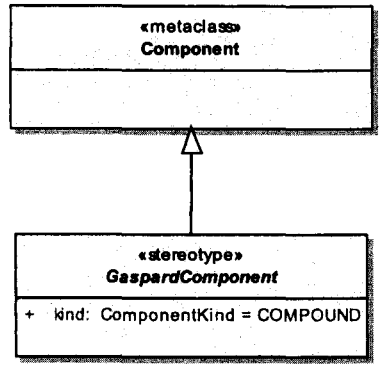


FIG. 4.11 – Stéréotype «GaspardComponent»

Le sous-package `Communications` (package `Common Behavior`) de la superstructure UML 2 définit les mécanismes nécessaires à la mise en oeuvre des invocations de comportement et des communications entre objets. Le pattern de communication de base est définit comme tel : Une action, représentant l'invocation d'une "capacité comportementale" (`behavioral feature`), est exécutée par un objet émetteur, et se traduit par la création d'un "évènement d'invocation" (`invocation event`). L'évènement d'invocation correspond soit à l'émission d'un signal, soit à l'appel d'une opération. Un évènement d'invocation résulte en la création d'un object "requête" (`request`), qui regroupe, entre autres, les données qui ont été passées en paramètre de l'action à l'origine de l'évènement d'invocation (qui correspondent aux paramètres de la capacité comportementale invoquée), des informations sur la nature de la requête, l'identité des objets émetteur et récepteur,... Ce mécanisme est illustré dans fig.4.12, une vue domaine extraite de la superstructure d'UML 2.

Dans UML, on distingue basiquement deux types de requêtes entre instances : l'émission de signaux, et l'invocation d'opérations. Le type de la requête est déterminé par l'évènement d'invocation qui l'a créé (fig.4.13).

▶ **Émission de signaux** L'émission d'un signal est utilisé pour déclencher une réaction asynchrone du récepteur, sans attente de réponse. Un évènement d'invocation de type "send" résulte en la création d'une requête de type "send", qui aboutit elle-même à un évènement de type "signal" chez le récepteur. Ce type d'évènement est concrètement défini dans le métamodèle UML par la métaclasse `SignalTrigger`. Le concept de `Signal` (fig.4.14) permet de définir un type pour les instances de requêtes de type "send". Les données supportées par la requête de type "send" sont représentées par les attributs de l'instance du signal. Comme illustré en fig.4.14, un signal est un `Classifier`, et possède à ce titre des attributs. Pour la même raison, un signal constitue une entité à part entière, définie de façon indépendante des classifiers qui le manipule.

▶ **Invocation d'opérations** L'invocation d'une opération revient à demander l'exécution d'une opération à une instance, cette exécution pouvant être déclenchée de

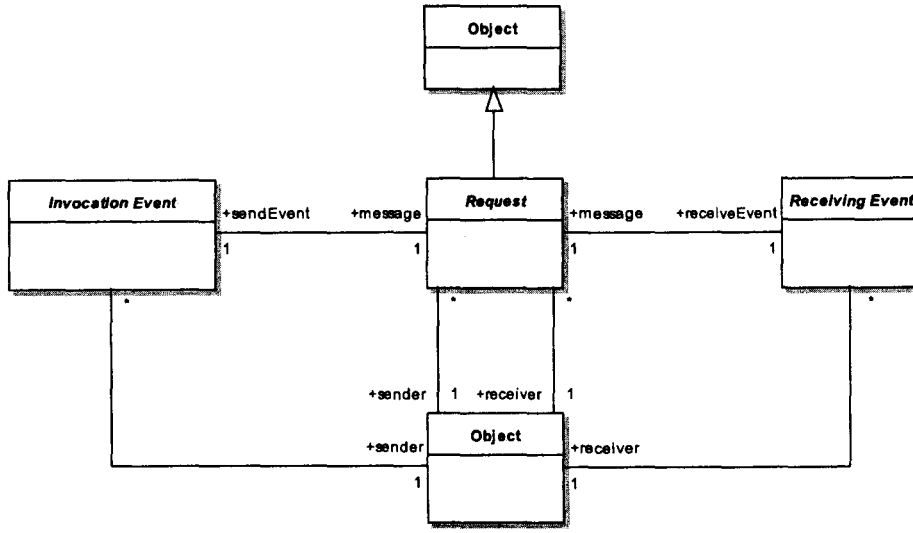


FIG. 4.12 – Vue domaine du "modèle" de communications dans UML

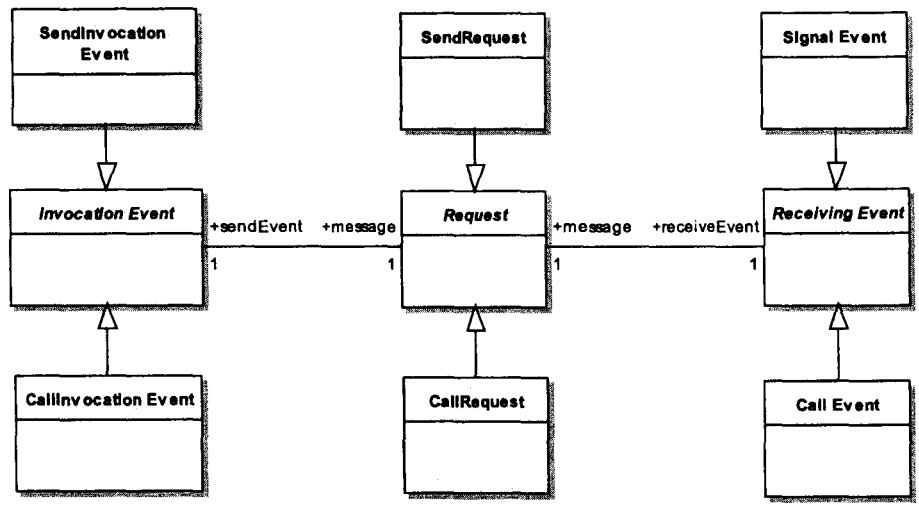


FIG. 4.13 – Vue domaine des différents types de requêtes dans UML

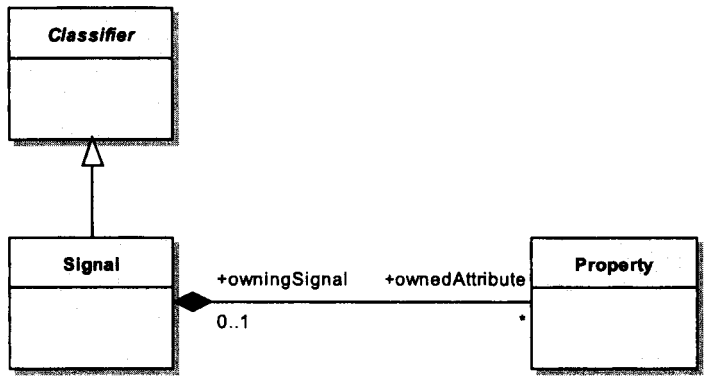


FIG. 4.14 – Définition du concept de `Signal` dans UML

façon synchrone ou asynchrone, éventuellement avec une réponse requise du récepteur à l'émetteur. Un évènement d'invocation de type "call" résulte en la création d'une requête de type "call", qui aboutit elle-même à un évènement de type "call" chez le récepteur. Concrètement, ce type d'évènement est défini dans le métamodèle UML par la métaclasse `CallTrigger`. L'évènement de type "call" résulte typiquement en l'exécution chez le récepteur du comportement associé à l'opération invoquée (i.e. exécution d'une méthode).

▶ **Interfaces pour un composant Gaspard** Le concept d'`Interface` (fig.4.15) est un type de `Classifier`, et possède à ce titre un ensemble de "capacités" (feature) structurelles et comportementales. Dans le profil Gaspard, on ne considérera que les behavioral features. Basiquement, une interface permet de spécifier une forme de contrat devant être satisfait par n'importe quelle instance de classifier réalisant cette interface. Un schéma classique dans UML consiste à définir un ensemble cohérent d'opérations dans une même interface. Tout classifier implémentant (ou réalisant) cette interface devra fournir une implantation pour chacune de ces opérations sous la forme de méthodes. L'approche orientée composant permet de raffiner un peu ce schéma. On considère qu'un composant expose des interfaces à son environnement, et que dans la mise en oeuvre de son comportement, un composant sera susceptible d'être stimulé par des requêtes correspondant aux opérations définies dans ces interfaces `PROVIDED`, et éventuellement d'émettre des requêtes à son environnement via ses interfaces `REQUIRED`. Comme nous l'avons déjà évoqué, nous imposons dans le profil Gaspard l'utilisation des ports pour l'exposition des interfaces.

Via les concepts de `CallTrigger` et de `SignalTrigger`, le package `Communications` définit la façon dont les éléments d'un système sont capables de communiquer. Les interfaces supportent naturellement l'utilisation des `CallTrigger` : La requête de type "call" correspond à une invocation d'une opération définie dans l'interface exposée. Pour le support des `SignalTrigger`, les classifiers (et par conséquent les interfaces) sont étendues dans le package `Communications` pour le support des `SignalTrigger` via le concept de `Reception` (fig.4.16). Le concept de `Reception` est en fait un type particulier de behavioral

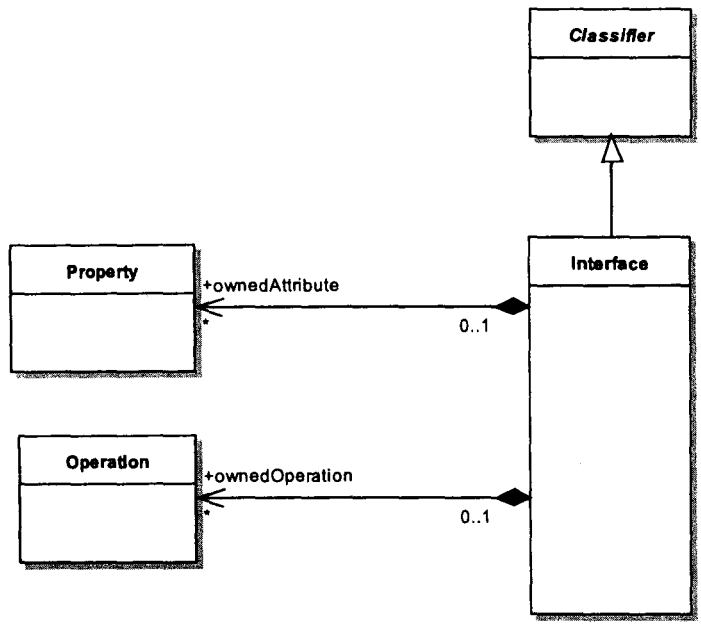


FIG. 4.15 – Définition du concept d'`Interface` dans UML

feature, au même titre que le concept d'`Operation`.

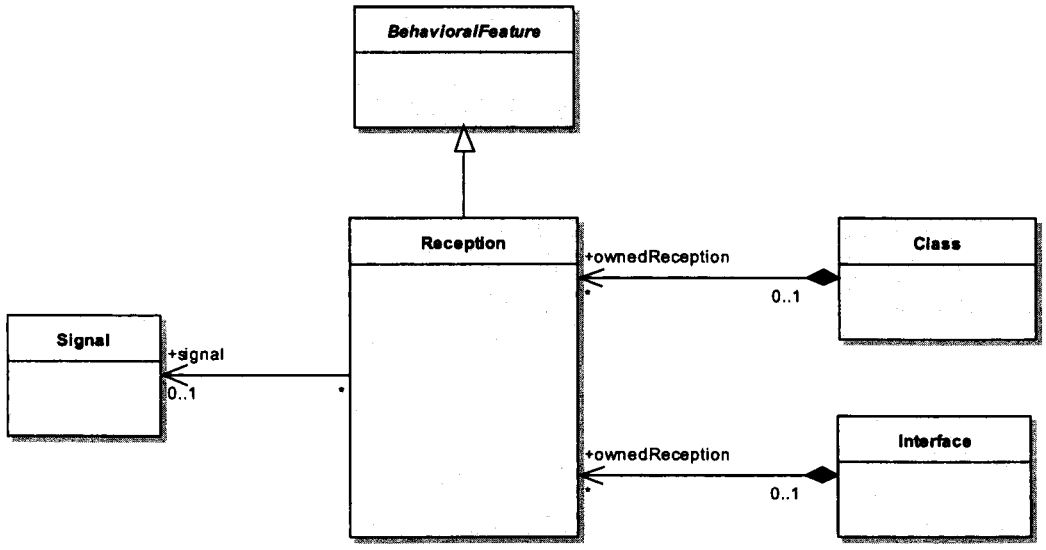


FIG. 4.16 – `Interface` étendue dans le package `Communications`

Définie dans une interface exposée de façon `PROVIDED`, une réception spécifie que le composant est en mesure de réagir à la réception d'un signal. Le type de signal auquel le classifier peut réagir est spécifié au niveau de la déclaration de la réception. Dans le profil Gaspard, les signaux, au même titre que les invocations d'opérations, sont délégués via les ports des composants aux parts décrivant la structure interne de chaque composant. Les requêtes sont finalement traitées de façon implicite par les composants élémentaires. Pour les interfaces exposées de façon `REQUIRED`, la spécification d'UML demeure par contre plus évasive quant à l'utilisation de réceptions. Dans le profil Gaspard, on considérera que lorsqu'un composant expose de façon `REQUIRED` une réception (via une interface), il est en mesure d'émettre des signaux du type décrit dans la réception, et il attend de son environnement qu'il soit en mesure de prendre en compte la réception de ce type de signaux. Nous verrons par la suite que pour la modélisation des applications, on ne considérera que les réceptions comme behavioral feature décrite au niveau des interfaces, et que pour la modélisation des architectures matérielles, on ne considérera que les opérations. Nous donnerons des exemples de définition d'interfaces dans les sections relatives à la description des packages `application` et `hardwareArchitecture`.

2.2.3 Paramètres

Nous avons montré dans la vue domaine que les composants étaient paramétrables, via le concept de `Parameter`. Les valeurs d'instance de ces paramètres peuvent être spécifiés au niveau de l'instanciation d'un composant, via une part dans la structure d'un autre composant. Pour la modélisation UML de ces paramètres, nous introduisons le

stéréotype «**Parameter**» (fig.4.17), s'appliquant aux attributs des composants stéréotypés «**GaspardComponent**» (ou plus précisément les stéréotypes qui en dériveront). Un paramètre peut être référencé (via son nom) à tous les niveaux de la spécification d'un composant, c'est à dire aussi bien au niveau des ports qu'au niveau des connecteurs ou des parts.

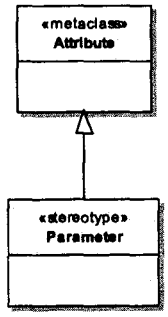


FIG. 4.17 – Stéréotype «Parameter»

Les valeurs d'instance des paramètres sont fixées lors de la définition d'une part, simplement en définissant une valeur pour les attributs stéréotypés «**Parameter**». Fig.4.18 illustre le mécanisme de définition de paramètres sur un composant `A`. Dans la structure du composant `B`, les valeurs d'instance des paramètres de `A` sont directement fixées au niveau de la part `a : A`. Dans la structure du composant `C`, les valeurs d'instance des paramètres sont fixées en faisant référence aux paramètres de `C`.

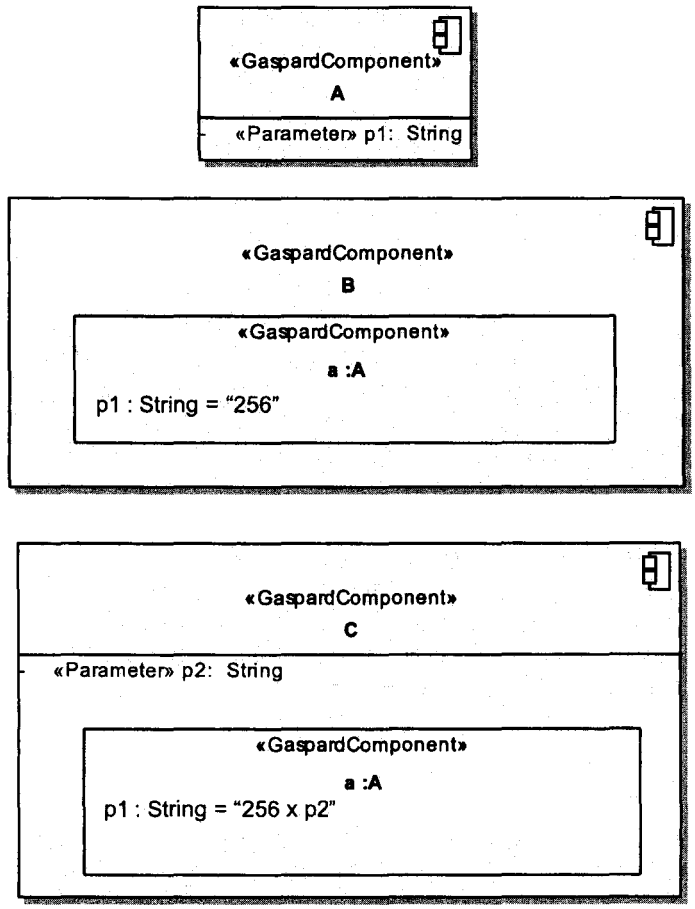


FIG. 4.18 – Exemple d'utilisation des paramètres

3 Package "factorization"

3.1 Vue domaine

Les mécanismes de factorisation décrits dans cette section sont fortement inspirés d'ArrayOL. Nous avons étudié dans la section consacrée à ArrayOL (chapitre positionnement) les mécanismes de pavage et d'ajustage des tableaux pour l'expression de tout le parallélisme potentiel des applications. D'une certaine façon, ce point de vue peut être généralisé en considérant que ces mécanismes permettent simplement d'exprimer des dépendances, ou encore des liens, entre les éléments des motifs des tableaux d'entrée et les éléments des motifs des tableaux de sortie d'une tâche. L'expression est juste réalisée de façon compacte, ou encore factorisée, via la définition d'un ensemble de vecteurs associés aux dépendances et la mise en oeuvre d'un processus itératif. En d'autres termes, on pourrait tout aussi bien tracer une à une les dépendances entre les motifs des tableaux d'entrée et les motifs des tableaux de sortie. Ces mécanismes permettent en quelques sortes de ne tracer qu'un seul lien symbolisant un ensemble de liens, les vecteurs de pavage et d'ajustage permettant de définir la topologie des liens. C'est sur la base de cette généralisation que nous introduisons les mécanismes de factorisation que nous présentons dans ce chapitre. En lieu et place des tableaux et des dépendances de données d'ArrayOL, nous allons respectivement considérer des "tableaux" d'éléments de modélisation et les relations entre ces éléments. Concrètement, les tableaux d'éléments vont être définis via le concept de `Multiplicity`, une multiplicité pouvant être associée à un port ou à une part. Des informations topologiques, via le concept de `LinkTopology`, vont permettent de définir la topologie des relations entre les éléments connectés ayant une multiplicité supérieure à un. Ces concepts vont nous permettre de spécifier de façon déterministe au moment de la conception tous les liens qui existeront potentiellement à l'exécution. L'idée de base consiste à identifier les relations entre les extrémités de liens concernées par chaque lien potentiel, et par conséquent déterminer de façon précise tous les liens potentiels. Nous verrons que ces mécanismes, bien qu'inspirés d'ArrayOL, permettent d'aller plus loin, et en particulier de définir des liens entre les instances potentielles correspondant à une même part[1].

3.1.1 Multiplicités multi-dimensionnelles

Le concept de `Multiplicity` permet de donner des directives concernant l'instanciation de l'élément auquel il est attaché (dans le profil Gaspard, seuls les ports et les parts supportent une multiplicité). Ces directives portent sur le nombre d'éléments potentiellement instanciables.

Le concept de multiplicité est familier des utilisateurs d'UML. Dans UML, il est défini comme étant un intervalle inclusif de valeurs entières non-négatives, débutant par une borne inférieure et se terminant par une borne supérieure. Il définit l'ensemble des cardinalités (ou nombres d'occurrences) valides pour l'élément auquel il est attaché. Dans le chapitre consacré à la vue UML du profil, nous étudierons plus en détail le concept de multiplicité dans UML. Nous pouvons juste considérer pour l'instant que dans UML, un élément possédant une multiplicité peut être vu comme un tableau d'éléments à une seule dimension. Dans le profil Gaspard, nous généralisons cette idée et considérons des multiplicités

[1] Les mécanismes d'ArrayOL pur ne concernent à priori que deux parts distinctes

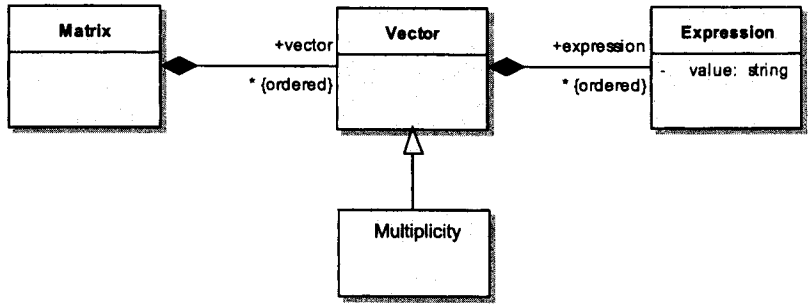


FIG. 4.19 – Multiplicity : Définition

potentiellement multi-dimensionnelles : Un élément ayant une multiplicité à plusieurs dimensions peut être vu comme un tableau d'éléments à plusieurs dimensions.

Comme illustré en fig.4.19, `Multiplicity` est définie comme étant un vecteur d'expressions. Une expression représente simplement l'abstraction d'une valeur numérique entière, potentiellement infinie, exprimée sous la forme d'une chaîne de caractères. Nous décrirons plus en détail à la fin de cette section le concept de `Expression`, en donnant en particulier une grammaire permettant de définir des valeurs valides pour l'attribut `value`. L'ensemble des expressions associées à une multiplicité étant ordonné, chaque expression détermine la taille de la dimension correspondante pour le tableaux multi-dimensionnel concerné. Dans Gaspard, contrairement à UML, on ne considère que les bornes supérieures. Nous verrons par la suite que l'interprétation de la sémantique de cette borne est différente pour la partie application et pour la partie architecture matérielle.

Dans Fig.4.20, nous illustrons l'utilisation des multiplicités multi-dimensionnelles sur des parts pour spécifier une topologie en cube de forme 2×2×2. Chaque instance potentielle de `a` possède implicitement un index qui identifie sa position dans le tableau multi-dimensionnel décrit par la multiplicité de `a`.

3.1.2 Modélisation de la topologie des liens

Afin de prendre en compte la modélisation de la topologie des liens, nous introduisons le concept abstrait de `LinkTopology`. `LinkTopology` est un ensemble optionnel d'informations pouvant être associé à une relation entre instances potentielles. Ce concept prend en compte les aspects multi-dimensionnels introduits dans la section précédente. Deux cas d'utilisation sont identifiés. Dans le premier cas, des liens sont exprimés entre des instances potentielles jouant le même rôle. Dans le second cas, ces instances potentielles ont un rôle différent. Ces deux cas nous amènent respectivement à deux raffinements de `LinkTopology` : `InterRepetitionLinkTopology` et `RepetitiveLinkTopology` (fig.4.21).

▶ **InterRepetitionLinkTopology** Les systèmes concernés sont composés d'une répétition d'un élément, tel que dans une topologie en grille ou en cube. Chaque ins-

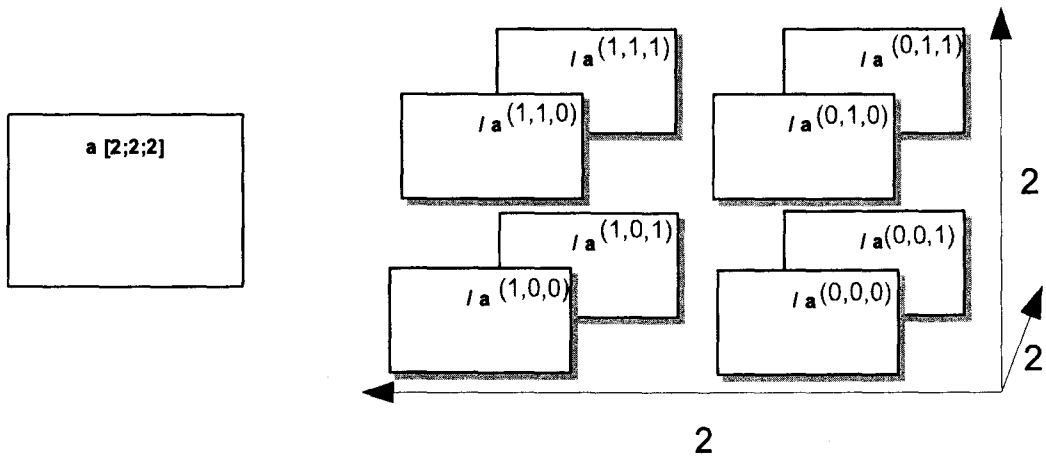


FIG. 4.20 – Topologie de type "cube" spécifiée via une multiplicité multi-dimensionnelle

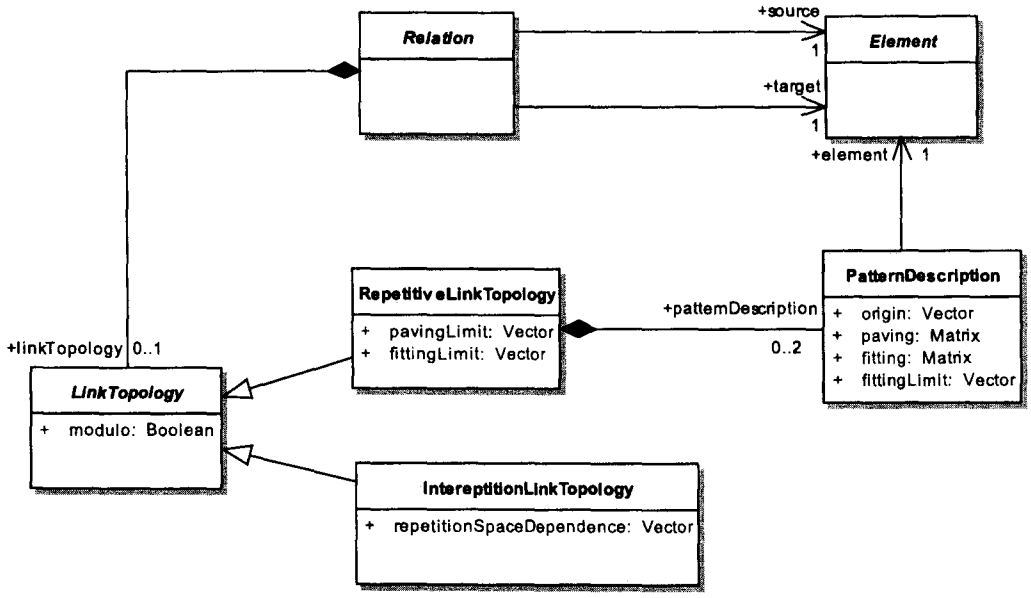


FIG. 4.21 – LinkTopology : Définition

tance potentielle de cet élément est connecté de façon régulière à d'autres instances potentielles de cet élément. Par exemple, dans le cas d'une grille torique, chaque instance est connectée à des voisins situés au nord, au sud, à l'est et à l'ouest. Le concept de `InterRepetitionLinkTopology` permet de spécifier la position des voisins de chaque instance potentielle d'un élément possédant une multiplicité multi-dimensionnelle. Chaque instance potentielle est implicitement associée à un point du tableau multi-dimensionnel décrit par la multiplicité de l'élément. L'attribut `repetitionSpaceDependance` est un vecteur de translation sur l'espace du tableau multi-dimensionnel, qui identifie la position d'un voisin donné. L'attribut `modulo` indique si la translation est appliquée modulo la taille du tableau multi-dimensionnel. Si `modulo` est égal à `false`, la translation n'est pas appliquée sur les bords du tableau, et le lien correspondant ne sera pas créé.

▶ **RepetitiveLinkTopology** Dans le second cas d'utilisation, des topologies complexes doivent être modélisées entre différentes instances potentielles jouant des rôles différents. Chaque élément répété typant une instance potentielle possède une multiplicité multi-dimensionnelle. Chaque point des tableaux multi-dimensionnels identifiés par les multiplicités multi-dimensionnelles correspond à une extrémité de lien potentielle. Dans le cas où l'élément répété possède des ports et qu'une connexion est exprimée sur l'un de ces ports (ce qui est le cas pour les connecteurs, mais pas nécessairement pour les dépendances d'allocation et de caractérisation que nous verrons dans le package association), les ports sont considérés comme les extrémités des liens, et la taille et le nombre de dimensions du tableau multi-dimensionnel est basé à la fois sur la multiplicité de l'élément répété et sur la multiplicité du port. Le mécanisme que nous introduisons via le concept `RepetitiveLinkTopology` permet de spécifier de façon compacte toutes les correspondances existant entre les extrémités contenues dans chacun des deux tableaux multi-dimensionnels, et par conséquent tous les liens qui existeront à l'exécution. Comme pour ArrayOL, l'idée consiste à identifier des motifs de points à l'intérieur de chacun des tableaux, et de mettre en relation les points contenus dans ces motifs (et par conséquent les extrémités des liens).

Dans le cas général, un `PatternDescription` est associé à chaque extrémité de la relation pour identifier les extrémités de liens appartenant à un motif (Il est ici important de noter qu'à la différence d'UML, toutes les relations qui sont considérées dans le profil Gaspard sont binaires. C'est pourquoi les deux "extrémités" d'une relation sont clairement identifiées dans fig.4.21 par deux rôles différents : `source` et `target`). Nous retrouvons ainsi l'attribut `paving` (pavage), défini par un ensemble de vecteurs (`Matrix`) permettant d'identifier l'origine de chaque motif à l'intérieur du tableau correspondant à une fin de relation. Le nombre de motifs contenus à l'intérieur d'un tableau est déterminé par l'attribut `pavingLimit`. Chaque tableau contient donc le même nombre de motifs, ces motifs pouvant éventuellement se recouvrir. Des règles identiques à celles d'ArrayOL s'appliquent : Identifier l'origine de chaque motif à l'intérieur d'un tableau peut être vu comme un processus itératif, où les limites d'itération sont données par le vecteur `pavingLimit`. Chaque origine de motif est calculée en multipliant chaque indice de l'espace d'itération par les vecteurs de pavage, en ajoutant le vecteur `origin` associé. A partir de chaque origine identifiée, les points appartenant au motif sont calculés grâce à l'attribut `fitting` (ajustage), lui aussi défini comme étant un ensemble de vecteurs. L'attribut `fittingLimit` détermine le nombre de points appartenant à chaque motif, ou en d'autres termes, la forme (nombre de

dimensions) et la taille (taille de chaque dimension) des motifs. Chaque point appartenant à un motif est calculé en multipliant les vecteurs d'ajustage par chaque indice de l'espace de répétition défini par l'attribut `fittingLimit`, en ajoutant l'origine du motif courant.

Pour un indice de répétition donné, $i, 0 \leq i <$ `pavingLimit`, le motif est composé des points indexés par l'ensemble suivant :

$$\{\texttt{origin} + \texttt{paving}.i + \texttt{fitting}.j \mid 0 \leq j < \texttt{fittingLimit}\} \quad (4.1)$$

si l'attribut `modulo` est égal à `false` et par

$$\{\texttt{origin} + \texttt{paving}.i + \texttt{fitting}.j \mod \texttt{shape} \mid 0 \leq j < \texttt{fittingLimit}\} \quad (4.2)$$

si l'attribut `modulo` est égal à `true`. Dans ce cas, `shape` représente la forme du tableau multi-dimensionnel auquel le motif appartient. Dans Fig.4.22, nous illustrons l'utilisation de ces mécanismes pour la définition d'un pattern de connection de type "perfect shuffle". Ce type de topologie se retrouve par exemple dans les réseaux multi-étages, tels que le réseau d'interconnexion Omega [13].

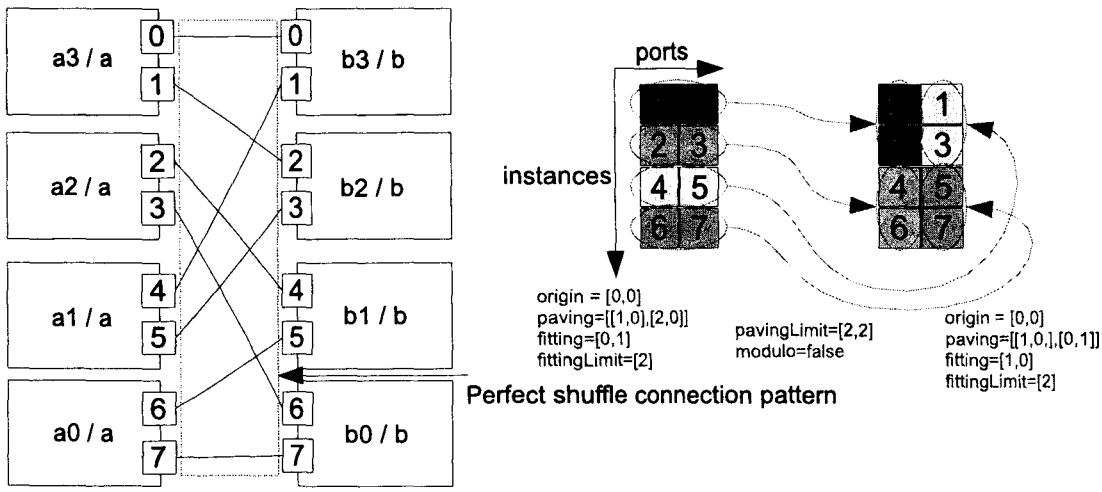


FIG. 4.22 – Modélisation d'un "perfect shuffle"

Dans le cas des composants ayant l'attribut `componentKind` égal à `REPETITIVE`, on autorise une simplification du mécanisme. Dans le chapitre sur ArrayOL, nous avons défini le concept d'espace de répétition (ou itération), via les attributs `pavingLimit` et `fittingLimit`. Un composant de type `REPETITIVE` représente un espace de répétition. La forme et la taille de l'espace de répétition sont déterminés par la multiplicité multi-dimensionnelle de l'élément qu'il contient. En d'autres termes, chaque instance potentielle de l'élément répété est implicitement associée à un point de l'espace de répétition. Les liens concernent les ports en bordure du composant et les ports de chaque instance potentielle de l'élément répété. Dans le contexte d'un composant de type `REPETITIVE`, les `RepetitiveLinkTopology` peuvent ne posséder qu'un seul `PatternDescription`, associé au port en bordure du composant. La valeur de l'attribut `pavingLimit` est déterminée par la multiplicité multi-dimensionnelle de l'élément répété, et la valeur de

l'attribut `fittingLimit` est déterminée par la multiplicité multi-dimensionnelle du port concerné sur l'élément répété. Plus généralement, pour le cas trivial où deux éléments ayant la même multiplicité sont connectés, on autorise à ne pas du tout spécifier de `PatternDescription`. Pour deux espaces de répétitions `E1` et `E2` de mêmes formes et de mêmes tailles, chaque élément de `E1` est relié uniquement à l'élément de `E2` ayant le même indice.

Les différents vecteurs manipulés, que ce soit au niveau des multiplicités, des pavages, des ajustages,... sont des vecteurs d'expressions. Une expression est une chaîne de caractères représentant une abstraction d'une valeur numérique. Plus précisément, il s'agit d'une expression affine (du type `y=ax+b`), pouvant faire référence à un paramètre défini au niveau du composant définissant l'espace de nommage de l'élément auquel est rattaché l'expression. Par exemple, pour un composant ayant un paramètre nommé `portSize`, la multiplicité d'un port de ce composant pourra être spécifiée de la manière suivante : `[portSize,portSize x 2]`. Si le composant est instancié en tant que `Part` dans la structure d'un autre composant avec pour valeur d'instance du paramètre `portSize` la valeur `256`, on pourra déduire que la multiplicité du port sera `[256,512]`. La valeur d'instance d'un paramètre peut être infinie, et est dans ce cas exprimée par le caractère `*`. Comme nous le verrons par la suite, `*` ne sera utilisé que pour la modélisation d'applications.

Les bases de nos mécanismes de factorisation étant maintenant posées, nous verrons dans les sections consacrées aux packages `application`, `hardwareArchitecture` et `association` ce qu'ils apportent pour les différents aspects du Y de Gaspard.

3.2 Vue UML

Dès ses premières versions (1.x [42]), UML proposait des mécanismes permettant d'exprimer de façon compacte la structure d'un système, ainsi que la description des relations entre les entités qui le composent. Les relations (associations, compositions, agrégations,...) peuvent être exprimées entre des entités ayant une existence au niveau conception (telles que des classes). Elles définissent les rôles qui seront jouées par les instances potentielles de ces entités (telles que des objets) à l'exécution, dans le contexte de ces relations. Via un mécanisme de multiplicité, elles permettent également de spécifier de façon compacte le nombre potentiel d'occurrences concernées par ces relations au moment de l'exécution.

UML 2 enrichit ces mécanismes, et permet de raffiner la description de ces relations dans le contexte des structures composites, que nous avons présenté dans la vue UML du package `component`. Les structures composites font référence à une composition d'éléments interconnectés. Ces éléments représentent des instances potentielles collaborant via des liens de communication pour atteindre des objectifs communs. Dans ce type de structure, les éléments sont effectivement instanciés lors de l'instanciation du classifier les contenant. Le nombre potentiel d'occurrences de ces éléments ainsi que le nombre de liens de communication peuvent également être spécifiés avec des multiplicités, comme nous l'illustrons en fig.4.23 avec un exemple tiré de la superstructure d'UML 2.

3.2.1 Limitations des mécanismes structurels d'UML

D'après la spécification de l'infrastructure UML 2, une multiplicité est la définition d'un intervalle inclusif de valeurs entières non négatives, débutant par une borne inférieure et se terminant par une borne supérieure potentiellement infinie. Elle spécifie l'ensemble des

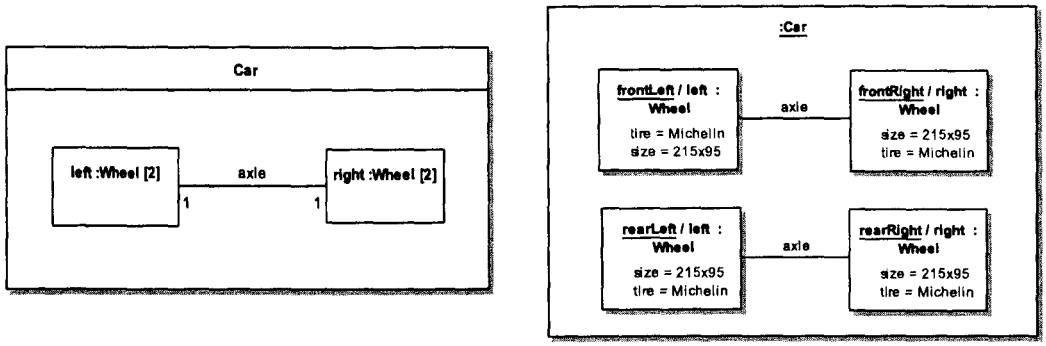


FIG. 4.23 – Exemple de structure composite et d'instanciation

cardinalités valides qu'un ensemble doit pouvoir supporter. Dans le package kernel du métamodèle UML 2 (Fig.4.24), le concept abstrait de `MultiplicityElement` est introduit pour supporter ces informations.

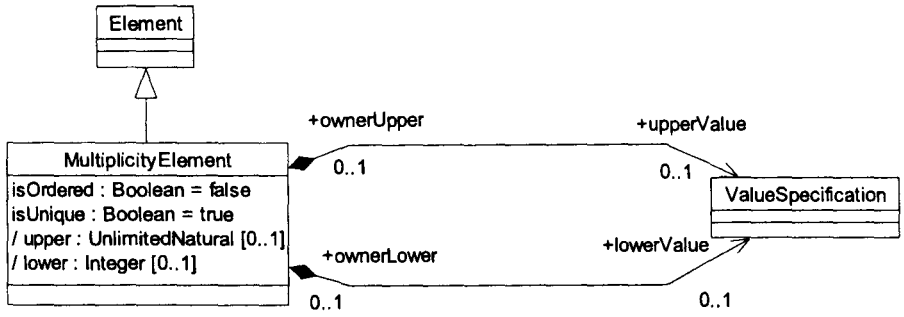


FIG. 4.24 – Définition des multiplicités (Infrastructure UML 2)

Ces mécanismes permettent de modéliser de façon compacte la structure d'un système ou d'un élément, mais ne supportent aucune information concernant la topologie des liens. Dans le cas des structures composites, ce fait peut introduire des problèmes d'ambiguïté. Basiquement, seuls deux patterns de connection peuvent être considérés comme étant déterministes : le pattern de connection "tableau", ou "un vers un" (Fig.4.23), et le pattern de connection "en étoile", ou "un vers tous" (Fig.4.25). En d'autres termes, un concepteur peut avoir une idée claire de la topologie qu'il modélise seulement dans les cas où (1)la multiplicité associée à chaque extrémité du connecteur est égale à 1, et les différents rôles possèdent la même multiplicité (pattern de connection "tableau"), ou (2)la multiplicité de chaque extrémité du connector est égale à la multiplicité du rôle correspondant (pattern de connection en "étoile").

De façon plus générale, la règle suivante (extraite de la super-structure d'UML 2) est appliquée pour déterminer la façon dont les liens seront effectivement créés au moment de

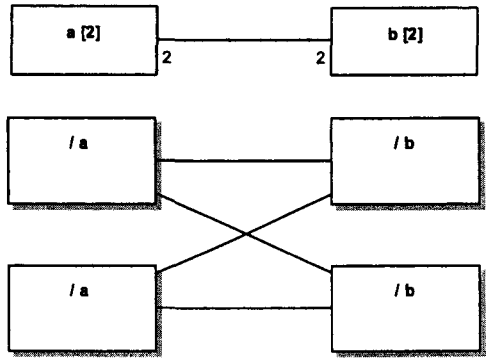


FIG. 4.25 – Pattern de connection en "étoile"

l'instanciation de l'englobant : "Les liens seront créés pour chaque instance (correspondant aux rôles connectés) en fonction de leur ordre, jusqu'à ce que la multiplicité minimum soit atteinte pour chacune des extrémités des connecteurs". Dans le cas spécifié en Fig.4.26, des connectors sont exprimés en trois instances potentielles de a et deux instances potentielles de b. Chaque a est connecté à au moins un b, et chaque b est connecté à au moins deux a. En appliquant la règle générale définissant la création des liens (en espérant avoir correctement compris la définition ambiguë qui nous est donnée dans la superstructure d'UML 2), nous obtenons deux topologies différentes dans le cas où l'on commence à tracer les liens en partant des instances de a, ou en partant des instances de b. Le numéro apparaissant sur chaque lien indique l'ordre dans lequel les liens ont été créés. A noter que les deux interprétations sont valides.

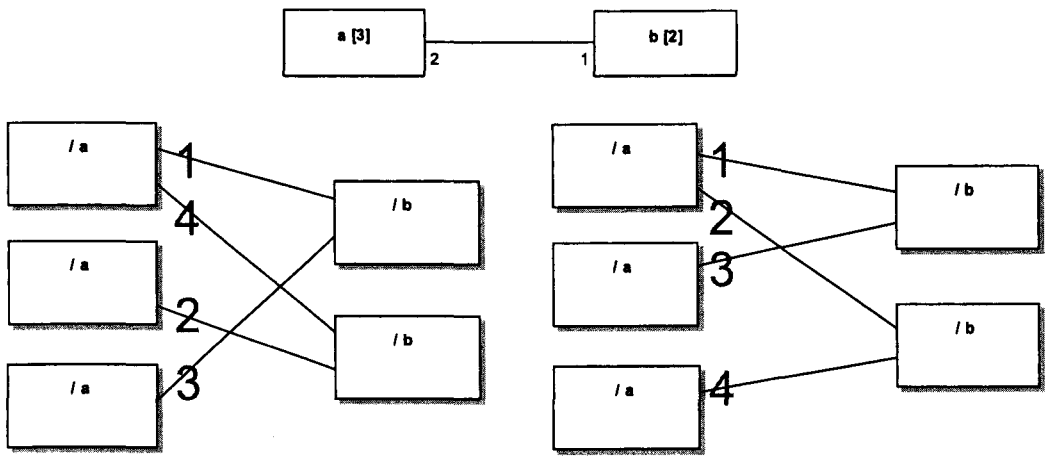


FIG. 4.26 – Exemple de structure composite ambiguë

Si l'on prend maintenant en compte les ports (un autre type essentiel d'élément structurel apparaissant dans les structures composites, et dont nous imposons l'utilisation comme nous l'avons vu dans le package `component`) et les multiplicités qui leur sont associées, le problème devient encore plus ambiguë. Nous avons par ailleurs montré que ces mécanismes ne supportent aucune information topologique. Les topologies modélisées sont la conséquence d'un processus de production, et non une propriété clairement identifiée des modèles. Si un concepteur veut spécifier une topologie particulière, l'information doit être facilement extractible du modèle. Les mécanismes que nous avons présenté dans la vue domaine offre cette propriété, et nous allons maintenant montrer comment nous les introduisons dans UML.

3.2.2 Multiplicités multi-dimensionnelles

Nous avons montré dans la section 3.2.1 que UML définissait le concept de multiplicité comme étant un intervalle inclusif de valeur entières non-négatives, débutant par une borne inférieure et se terminant par une borne supérieure. Dans Fig.4.24, nous pouvons constater que la métaclasse `MultiplicityElement` possède un attribut `isOrdered`, spécifiant si la "collection" d'éléments est ordonnée. Dans le cas où elle est ordonnée, cette "collection" peut être vue comme un tableau mono-dimensionnel, où les éléments sont implicitement indexés. Introduire des multiplicités multi-dimensionnelles revient à prendre en compte des tableaux multi-dimensionnels pour la description des "collections". Le sous-ensemble du métamodèle UML2 pour les multiplicités est étendu par l'introduction du concept de `MultiDimensionalMultiplicityElement` (Fig.4.27). Les attributs pour les bornes inférieures et supérieures sont définis par des vecteurs au lieu d'être définis par des entiers. Ici, chaque élément du `Vector` détermine la taille de la dimension correspondante pour le tableau multi-dimensionnel concerné. En d'autres termes, le concept de `MultiplicityElement` est vu comme un cas particulier du concept de `MultiDimensionalMultiplicityElement`. Les attributs de borne inférieure et supérieure de `MultiplicityElement` ne contiennent en effet qu'une seule valeur entière.

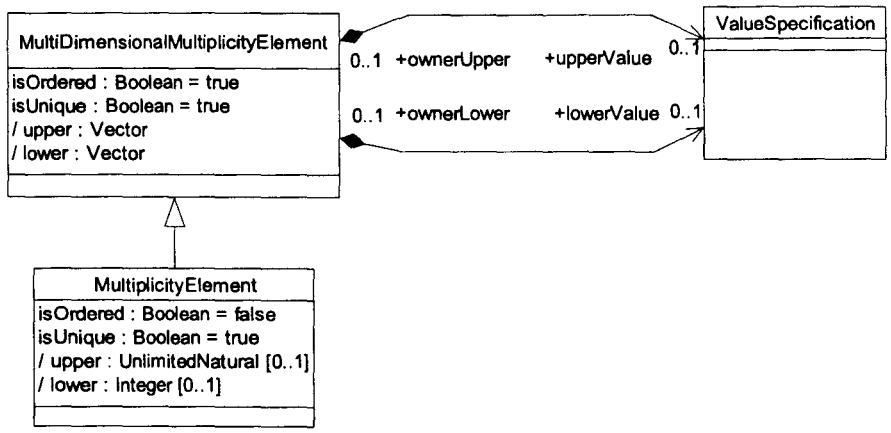


FIG. 4.27 – Définition des multiplicités multi-dimensionnelles

On constate par ailleurs que la valeur par défaut de l'attribut `isOrdered` de `MultiDimensionalMultiplicityElement` est égale à `true`. En effet, si les conditions multi-dimensionnelles ne sont pas ordonnées, nous perdons tout le déterminisme du mécanisme de définition de la topologie des liens. Nous imposons donc la contrainte que toute multiplicité multi-dimensionnelle définit est nécessairement ordonnée.

3.2.3 Modélisation de la topologie des liens

Afin de prendre en compte la modélisation de la topologie des liens, nous avons introduit dans la vue domaine le concept abstrait de `LinkTopology`. Pour l'implantation UML de ce concept, nous introduisons le stéréotype abstrait «**LinkTopology**», s'appliquant aux deux types de relations utilisés dans le profil Gaspard : les connecteurs et les dépendances (l'utilisation des dépendances sera évoquée dans le package `association`). Conformément aux deux cas d'utilisation identifiés dans la vue domaine, nous introduisons deux raffinements du stéréotype «**LinkTopology**» : «**InterRepetitionLinkTopology**» et «**RepetitiveLinkTopology**» (fig.4.28).

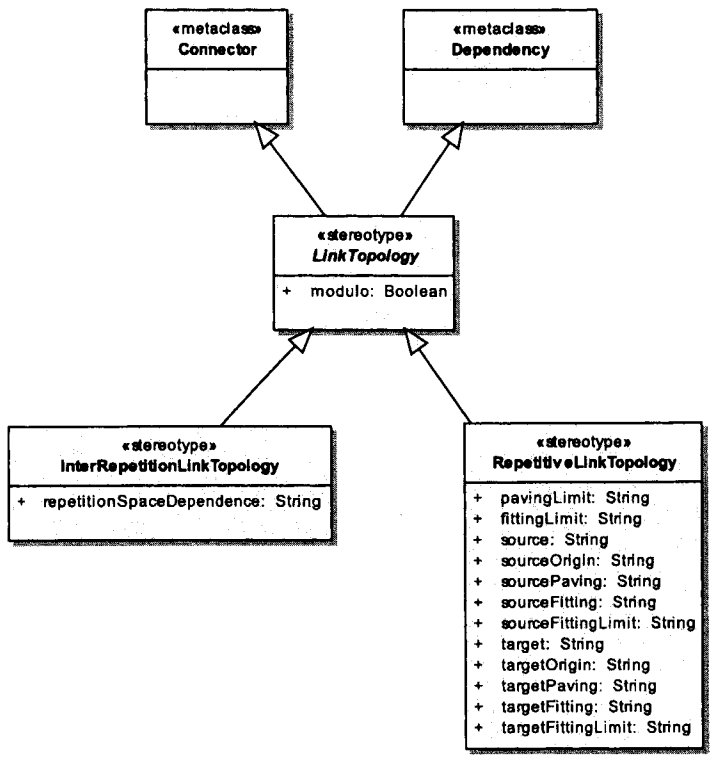


FIG. 4.28 – Définition du stéréotype «**LinkTopology**»

Les différents vecteurs que nous avons décrits dans la vue domaine se retrouvent sous la forme de tagged values de type `String` sur les stéréotypes «InterRepetitionLinkTopology»

et «RepetitiveLinkTopology», ces chaînes de caractères représentant des vecteurs d'expressions. Au niveau du stéréotype «RepetitiveLinkTopology», on constate par ailleurs que l'on retrouve les différents attributs de `PatternDescription`. Les tagged values `source` et `target` sont des chaînes de caractères référençant le nom des extrémités des relations. Chaque extrémité possède ainsi un ensemble de vecteurs permettant d'identifier les motifs concernés pour la définition de la topologie des liens.

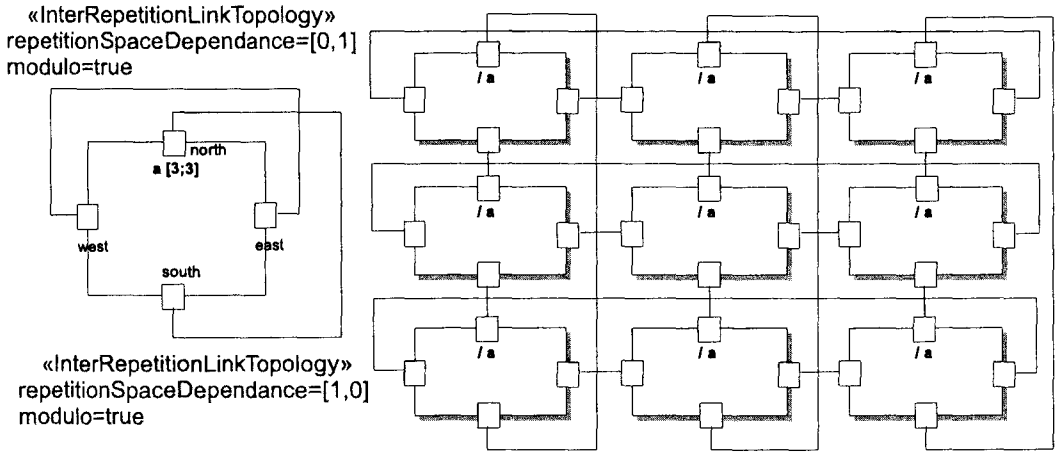


FIG. 4.29 – «InterRepetitionLinkTopology» : Modélisation d'une topologie en grille

Dans Fig.4.29, nous illustrons l'utilisation du stéréotype «InterRepetitionLinkTopology» pour la modélisation d'une topologie de type grille cyclique à deux dimensions.

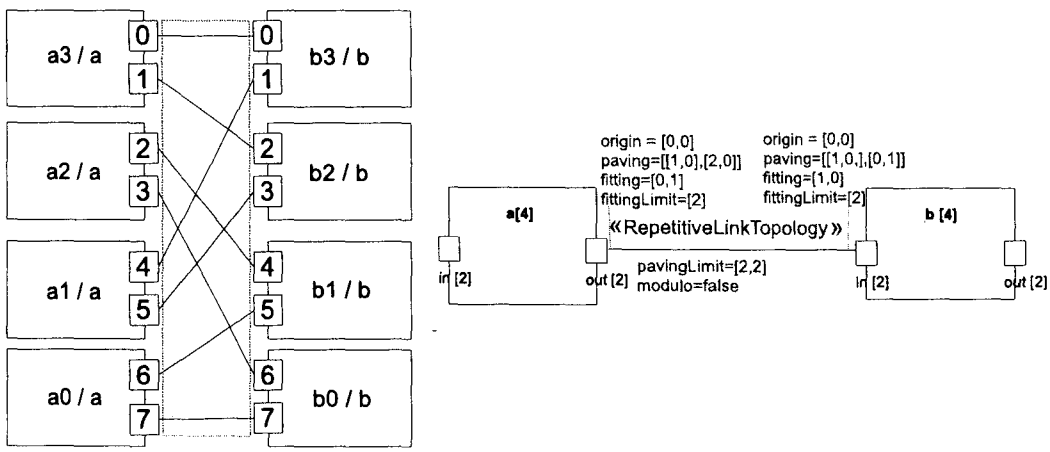


FIG. 4.30 – «RepetitiveLinkTopology» : Modélisation d'un "perfect shuffle"

Dans Fig.4.30, nous illustrons l'utilisation du stéréotype «**RepetitiveLinkTopology**»

pour la définition du pattern de connection de type "perfect shuffle", introduit dans la vue domaine. Ce type de topologie se retrouve par exemple dans les réseaux multi-étages, tels que le réseau d'interconnexion Omega, que nous étudierons dans le chapitre Étude de cas.

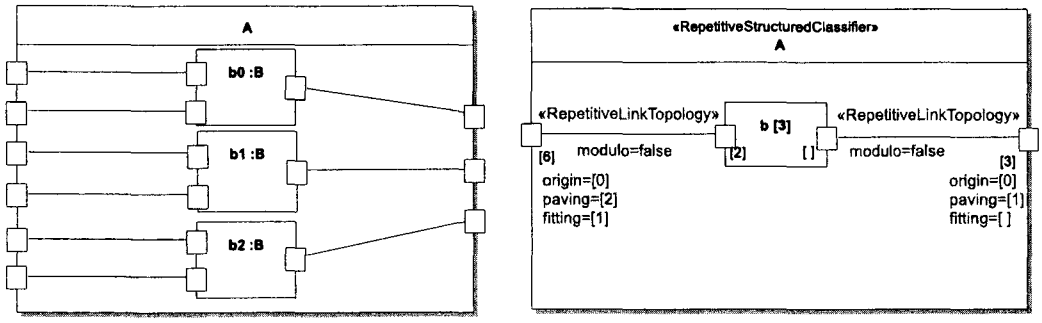


FIG. 4.31 – Utilisation simplifiée de «RepetitiveLinkTopology» dans un composant de type `REPETITIVE`

Dans Fig.4.31, nous illustrons l'utilisation simplifiée du stéréotype «RepetitiveLinkTopology» pour un composant de type `REPETITIVE`. On constate que, pour chacun deux connecteurs, les différents tagged values ne sont que partiellement spécifiés (ceux relatifs à la source pour l'un des connecteurs, et ceux relatifs à la cible pour l'autre connecteur). Les tagged values relatifs aux limites de répétition (pavage et ajustage) ne sont pas spécifiés. Conformément à ce que nous avons défini dans la vue domaine, les limites de répétition de pavage de chaque connecteur sont implicitement égales à la multiplicité multidimensionnelle de l'élément répété, et les limites d'ajustage sont égales à la multiplicité des ports de l'élément répété.

4 Package "hardwareArchitecture"

4.1 Vue domaine

La motivation du package `hardwareArchitecture` est de satisfaire le besoin grandissant des concepteurs de systèmes embarqués de spécifier l'architecture matérielle supportant le système à un haut niveau d'abstraction. L'objectif est d'être capable de dimensionner les ressources matérielles et de décrire la topologie de leurs interconnexions de façon suffisamment précise pour avoir un modèle pertinent, tout en s'abstrayant des détails, afin de pouvoir prendre rapidement des décisions efficaces. Comme nous l'avons évoqué dans l'introduction, la tendance actuelle pour les systems on chip est de concevoir des architectures multiprocesseurs. En anticipant un peu l'évolution de cette tendance, nous croyons que dans un futur plus ou moins proche, les systems on chip seront non seulement multiprocesseurs, mais également multi-processeurs réguliers. Cette anticipation est motivée par trois considérations :

1. Les contraintes de mise sur le marché deviennent si serrées que la réutilisation massive d'unités de calcul sera l'un des seuls moyens pour obtenir rapidement la puissance de calcul requise par les générations à venir d'applications embarquées.
2. Le parallélisme offre un moyen efficace pour réduire la consommation dynamique d'un system on chip. En effet, à puissance de calcul égale, une puce capable de réaliser plusieurs calculs simultanément pourra être cadencée à une fréquence inférieure à celle d'une puce pouvant réaliser moins de calculs dans un cycle donné (la fréquence est au carré dans l'équation de consommation d'énergie).
3. Une architecture parallèle régulière et homogène est plus facile à programmer qu'une architecture irrégulière.

La modélisation orientée composant et les mécanismes de factorisation vont nous permettre de modéliser facilement des system on chip multiprocesseurs de ce type, en exposant de façon compacte le parallélisme offert, et la topologie des connexions entre les différentes ressources.

Le concept de `HardwareComponent` étend le concept de `Component`, et hérite à ce titre de ses mécanismes d'encapsulation et de composition/assemblage. Un composant matériel est une abstraction d'une ressource matérielle physique, pouvant éventuellement être décrite de façon hiérarchique. Le type de structure du composant (`ELEMENTARY`, `COMPOUND` ou `REPETITIVE`) varie en fonction des cas d'utilisation. Un composant matériel ayant une structure de type `ELEMENTARY` est utilisé dans le cas où, par exemple, l'on dispose d'une description sous forme d'IP, où encore dans le cas où l'on ne souhaite pas décrire le composant à un niveau de granularité plus fin. Un composant ayant une structure de type `COMPOUND` peut représenter une "architecture exécutable" (c'est à dire complètement définie), ou encore une partie d'architecture pouvant être réutilisée dans d'autres contextes. Un composant de type `REPETITIVE` sera plus particulièrement utilisé pour la modélisation d'architectures parallèles, telles qu'une grille de processeurs, éventuellement réutilisé pour la modélisation d'autres architectures.

A haut niveau, une carte peut être vue comme une ressource offrant des capacités de calcul et de stockage de données. En raffinant la description hiérarchiquement, on peut par exemple identifier le fait que la carte est composée d'une grille de processeurs élémentaires disposant d'une mémoire locale, interconnectés sous forme de grille, ayant

accès à une mémoire globale partagée. La possibilité de définir des modèles d'architecture hiérarchiques est un avantage pour la modélisation du placement d'une application elle-même hiérarchique. Cela permet par exemple de mettre en oeuvre un processus de placement itératif, où l'on décrit dans un premier temps un placement à gros grain (l'application et les données qu'elle consomme sont mappées grossièrement sur la carte), et où l'on raffine ensuite le placement des différentes tâches composant l'application sur les différents processeurs contenus dans la carte, et le placement des données consommées sur les mémoires locales de chaque processeur.

4.1.1 Classification des ressources

Comme nous l'avons évoqué dans le chapitre Positionnement, le profil SPT propose une classification des ressources (matérielles et logicielles) selon trois critères : Purpose, Activeness, Protection. Nous proposons de réutiliser cette classification pour notre modèle d'architecture matérielle, en l'adaptant et en la spécialisant sensiblement. Plus précisément, nous orientons la structure de notre classification en se basant principalement sur la classification fonctionnelle (purpose) proposée par SPT : Processor, Communication et Device (fig.4.32). Pour chacun de ces types de fonction, nous définissons des spécialisations. Concernant la classification suivant les critères d'activité (Activeness) et de protection, nous verrons que la sémantique donnée aux différentes spécialisations donne implicitement une classification dans ces deux catégories.

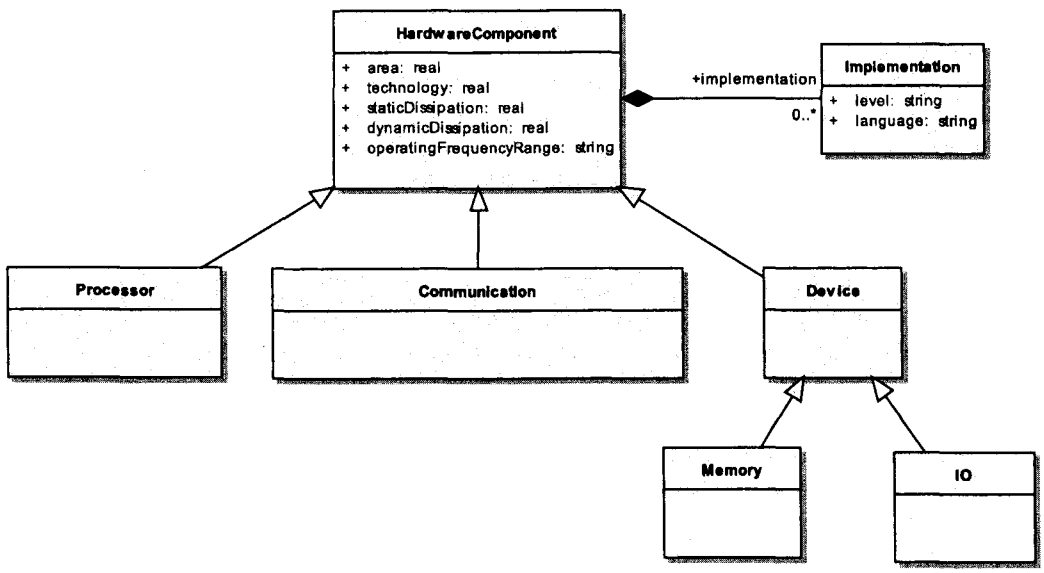


FIG. 4.32 – HardwareComponent : Classification des ressources

▶ **Structure de la classification** Dans le profil SPT, le concept de `Resource` définit l'ancêtre commun à tous les types de ressource manipulés. Par soucis d'homogénéité pour la définition du profil Gaspard, le concept de `Resource` est renommé en `HardwareComponent`. `HardwareComponent` devient donc l'ancêtre commun à tous les types de ressource manipulés dans le profil Gaspard. Les composants matériels disposent ainsi d'un certain nombre de caractéristiques communes, qui permettront d'orienter le choix des composants pour la conception d'une puce :

- **area : real,** la surface en mm^2 nécessaire à l'implantation physique du composant, compte tenu de la technologie de gravure utilisée.
- **technology : real,** la technologie de gravure en microns considérée pour l'implantation physique du composant (par exemple 0,35 microns).
- **staticDissipation : real,** la consommation d'énergie moyenne en watt lorsque le composant est dans un mode de fonctionnement statique (ou idle).
- **dynamicDissipation : real,** la consommation d'énergie moyenne en watt lorsque le composant est dans un mode de fonctionnement dynamique (ou actif). Pour l'exécution d'une application ou d'une tâche donnée, la consommation totale d'énergie est ainsi estimée en fonction du temps passé dans le mode statique et dans le mode dynamique.
- **operationFrequencyRange : string,** la gamme de fréquences d'horloge en mhz que le composant matériel est capable de supporter. Elle est exprimée sous la forme d'une énumération ou d'un intervalle de fréquences. Le choix de la fréquence de fonctionnement du composant est réalisé en fonction des caractéristiques attendues du système, telles que la limite de consommation d'énergie attendue ou encore le temps d'exécution requis pour l'exécution d'une application ou d'une tâche. Dans le cadre d'une architecture de type GALS, le réseau n'est typiquement pas cadencé par une horloge. Dans ce cas, la gamme de fréquence d'horloge du composant est définie par l'ensemble vide.

Outre les différents attributs que nous venons d'évoquer, la relation entre les concept de `HardwareComponent` et `Implementation` permet de définir les différentes implantations existantes d'un même composant. `Implementation` possède un attribut `language : string`, déterminant le langage dans lequel l'implantation du composant est définie, et un attribut `abstractionLevel : string`, déterminant le niveau d'abstraction auquel l'implantation du composant est définie. A la différence des artefacts des composants UML 2, le concept de `Implementation` ne définit pas un lien physique vers un fichier, mais spécifie simplement la disponibilité de l'implantation des composants dans différents langages, éventuellement à différents niveaux d'abstraction. Cette information a principalement un caractère de documentation, mais il est possible d'imaginer qu'elle puisse être utilisée par un outil paramétrable d'exploration d'architecture. Un exemple de critère d'exploration pourrait être de n'utiliser que des composants possédant une implantation au niveau `TLM`, afin de garantir une simulation rapide de l'architecture.

▶ **Ressources de type Processor** Les composants matériels de type `Processor` représentent des ressources capables de réaliser des calculs. En règle générale, ils sont également capables d'émettre des requêtes de lecture/écriture à leur environnement, bien que cela dépende évidemment des interfaces qui leurs sont associées, et de l'orientation de ces interfaces (`REQUIRED` ou `PROVIDED`). Dans le modèle d'association de Gaspard, les res-

sources de type `Processor` sont utilisées comme support d'exécution pour les composants applicatifs. En plus des caractéristiques héritées de `HardwareComponent`, le concept de `Processor` possède l'attribut `communicationFrequencyRange : string`, la gamme de fréquences supportées par le processeur pour les communications avec l'environnement. Les valeurs des fréquences de communication sont en général nettement plus faibles que les valeurs des fréquences de calcul.

Nous distinguons basiquement quatre types de processeurs : `Gpp`, `Dsp` (ces deux types processeurs étant regroupés dans la catégorie abstraite `ProgramableProcessor`), `Fpga` et `Asic` (fig.4.33) :

1. `ProgramableProcessor`, définit un type de processeur programmable, dans le sens ou il dispose d'un ou plusieurs jeux d'instructions. L'association entre les concepts de `ProgramableProcessor` et de `InstructionSet` permet de spécifier qu'un processeur supporte un ou plusieurs jeux d'instructions. Bien que l'on pourrait assez facilement décrire les instructions composant un jeu d'instructions, un jeu d'instructions n'est défini que par son nom dans le contexte du profil Gaspard. Nous verrons par la suite dans la section consacrée au modèle d'applications que les composants applicatifs possèdent également une association avec le concept de `InstructionSet`. Cette association permet de spécifier que l'on dispose de l'implantation d'un composant applicatif dans l'assembleur correspondant aux jeux d'instructions, ou que l'on dispose des compilateurs nécessaires à la production de ce code assembleur. Dans ce contexte, identifier les jeux d'instructions par leur nom suffit à déterminer si le placement d'un composant applicatif sur un processeur est pertinent, c'est à dire si le processeur supporte le ou les jeux d'instructions pour lesquels une implantation du composant applicatif est potentiellement disponible. `ProgramableProcessor` est un concept abstrait, et est raffiné en deux concepts concrets, ayant valeur de documentation (ils ne rajoutent pas d'attributs, et sont manipulés de la même manière dans les modèles) :
   - `Gpp`, pour General Purpose Processor, ou processeur généraliste.
   - `Dsp`, pour Digital Signal Processor, ou processeur de traitement de signal. Ces processeurs disposent d'architectures particulières leur permettant d'avoir de bonnes performances pour des calculs typiquement réalisés dans les applications de traitement de signal, comme par exemple la possibilité de réaliser en un seul cycle d'horloge une multiplication et une addition (mul/add)
2. `Fpga`, pour Field Programable Gate Array, correspondant à un circuit reconfigurable.
3. `Asic`, pour Application Specific Integrated Circuit, correspondant à un circuit intégré spécialisé pour une fonction précise.

▶ **Ressources de type Memory** Le concept abstrait de `Memory` représente une ressource offrant une capacité de stockage pour les données. Par rapport à la classification SPT, on peut donc le considérer comme un type particulier de `Device`, puisqu'il ne rentre ni dans la catégorie des processeurs, ni dans la catégorie des ressources de communication. Dans le modèle d'association, les mémoires sont utilisées pour le placement des données manipulées par les composants applicatifs. Les composants matériels de type `Memory` sont caractérisés par les attributs suivants :

- **wordSize : int**, la taille des mots mémoires en bit.
- **numberOfWords : int**, la taille de la mémoire exprimée en nombre de mots.

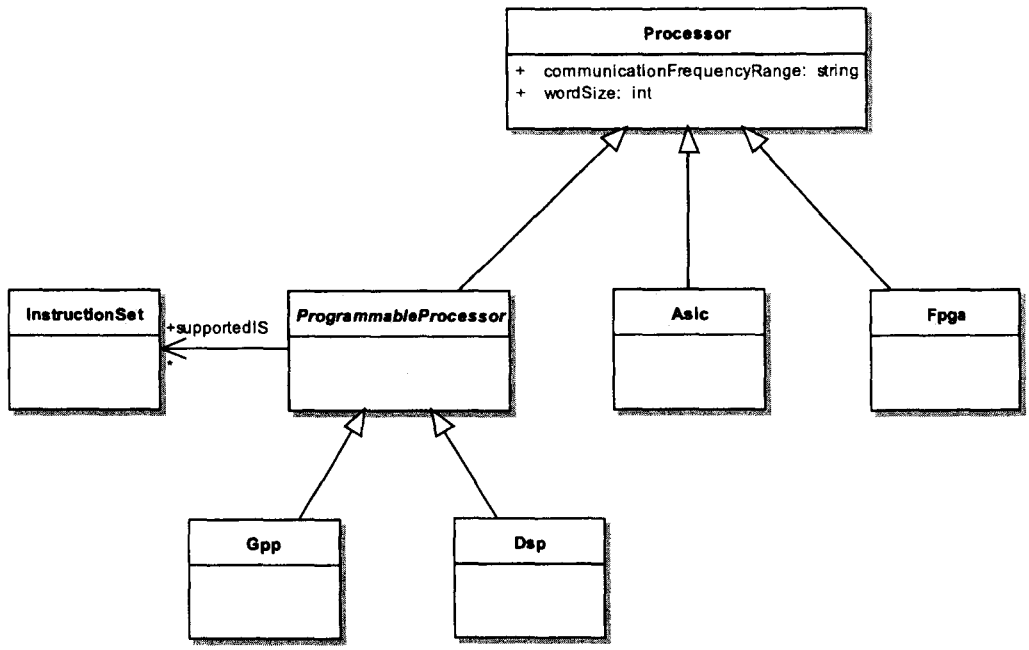


FIG. 4.33 – Raffinement des ressources de type Processor

Le concept abstrait de Memory est raffiné par les concepts de Cache et de Ram :

1. Cache : Ce concept définit l'abstraction d'une mémoire cache et de son contrôleur. Un composant matériel de type Cache possède ainsi typiquement un port avec interface PROVIDED fournissant des services de lecture/écriture à un processeur, et un port avec interface REQUIRED lui permettant de requérir des services de lecture à son environnement lorsqu'un défaut de cache se produit, ou encore de requérir des services d'écriture pour déléguer les requêtes émises par le processeur. Bien que les caches ne soient pas directement manipulables par un programmeur (dans le modèle d'association, on ne peut pas placer directement les données sur les mémoires cache), connaître la taille des caches et leur organisation permet de dimensionner les blocs de données à manipuler pour minimiser le nombre de transfert. En plus des attributs hérités de Memory, un composant de type Cache est caractérisé par les attributs suivants :
   - **cache : CacheType**, le type (UNIFIED pour un cache banalisé, DATA pour un cache de données, ou INSTRUCTION pour cache d'instruction) de mémoire cache considéré.
   - **associativity : int**, le degré d'associativité de la mémoire cache. Si l'attribut associativity est égal à 0, l'associativité est de type DIRECT MAPPED. S'il est égal à numberOfLines, l'associativité est de type FULL ASSOCIATIVE. Pour les valeurs intermédiaires, l'associativité est de type K-WAY ASSOCIATIVE, avec K = associativity.
   - **numberOfLines : int**, le nombre de lignes contenues dans la mémoire cache.

   Par ailleurs, une mémoire cache est également caractérisée par une politique d'écriture, une politique de remplacement, et une politique de gestion de la cohérence (dans le cas d'une architecture multiprocesseurs). Cette caractérisation est mise en oeuvre via les associations entre les concepts de Cache et de CacheWritePolicy, CacheReplacementPolicy et CacheCoherencePolicy :
   - CacheWritePolicy : la politique d'écriture d'un cache. En fig.4.35, les types de politique d'écriture les plus courants (write back, write through, write once) sont identifiés. Comme pour les autres types de "politiques" associés aux caches, nous avons choisi de modéliser les politiques concrètes d'écritures de cache par extension de CacheWritePolicy, plutôt que par l'utilisation d'une liste énuméré. Ce choix est justifié par le fait qu'il n'est pas facile (voir impossible) d'être exhaustif, et que l'utilisation d'une énumération suggère que la liste identifiée est complète. L'ajout de nouvelles politiques d'écriture suppose que les outils gravitant autour de l'environnement Gaspard (optimisation, simulation, estimation de performance) sont étendus pour supporter ces politiques.
   - CacheReplacementPolicy : la politique de remplacement d'un cache. En fig.4.36, les types de politique de remplacement les plus courants (LFU, LRU, Random) sont identifiés.
   - CacheCoherencePolicy : la politique de gestion de la cohérence des caches dans le cadre d'une architecture multiprocesseur.
2. Ram : Contrairement aux mémoires de type Cache, les mémoires de type Ram ne peuvent disposer que de ports avec interface PROVIDED. Elles jouent ainsi le rôle de ressources matérielles ne pouvant que proposer des services de lecture/écriture aux autres ressources composant son environnement. Le concept de Ram est caractérisé par les attributs suivants :

- **bandwidth : real**, la bande passante de la mémoire.
- **readLatency : real**, la latence pour un accès en lecture.
- **writeLatency : real**, la latence pour un accès en écriture.

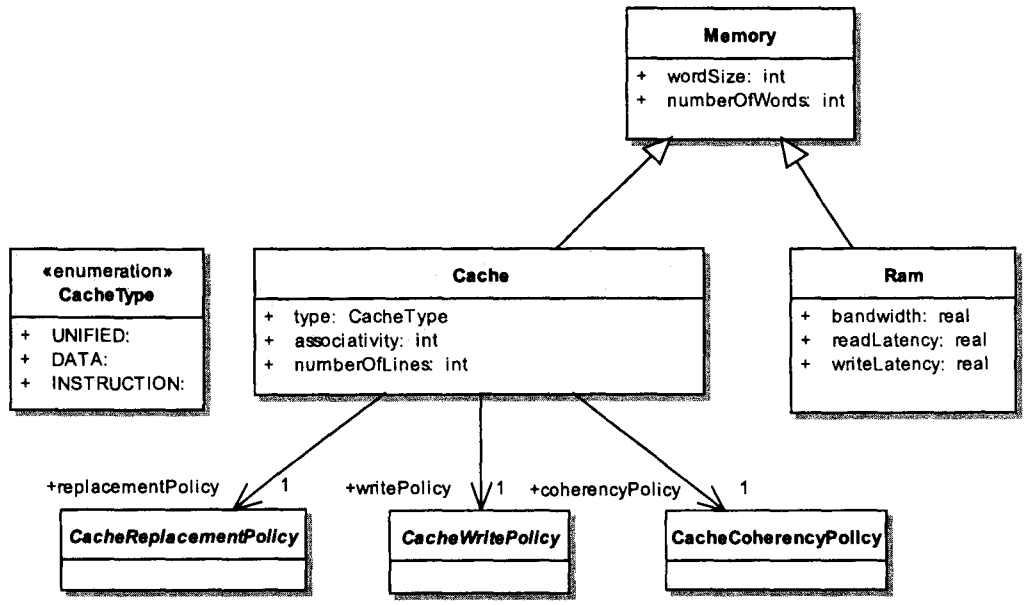


FIG. 4.34 – Raffinement des ressources de type Memory

▶ **Ressources de type Communication** Le concept de `Communication` représente un composant matériel permettant l'interconnexion entre les composants, et jouant un rôle de support pour les communications entre composants. Les composants matériels de type `Communication` sont caractérisés par les attributs suivants :

- **size : int**, la taille exprimée en bits des données véhiculées par le composant.
- **bandwidth**, la bande passante de l'élément de communication.

Nous distinguons quatre types de ressources de communications (fig.4.37) :

1. `Bus`
2. `Switch`
3. `Router`
4. `Bridge`, pour permettre l'interconnexion entre bus.

L'association entre le concept de `Communication` et le concept abstrait de `CommunicationArbitrationPolicy` permet d'associer une politique d'arbitrage des conflits à une ressource de communication. En fig.4.38, les types les plus courants de politiques d'arbitrage sont identifiés. Le mécanisme de définition de nouvelles politiques est le même que les mémoires caches.

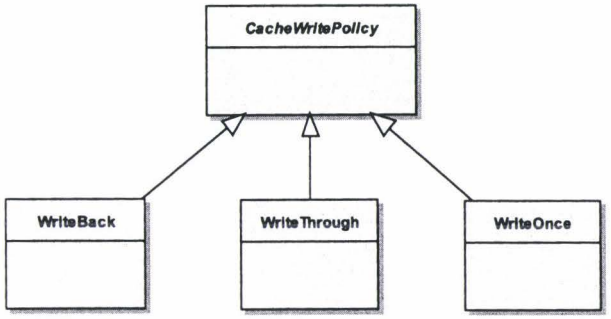


FIG. 4.35 – Politiques d'écriture des caches

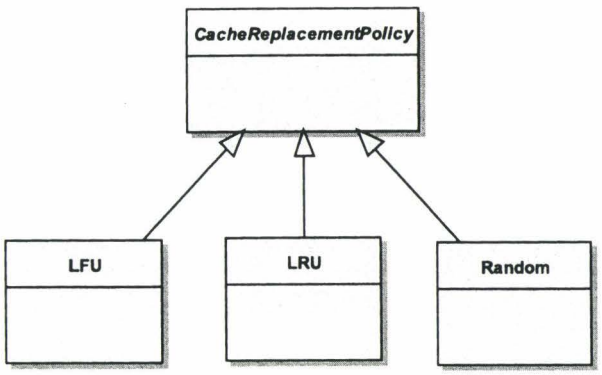


FIG. 4.36 – Politiques de remplacement des caches

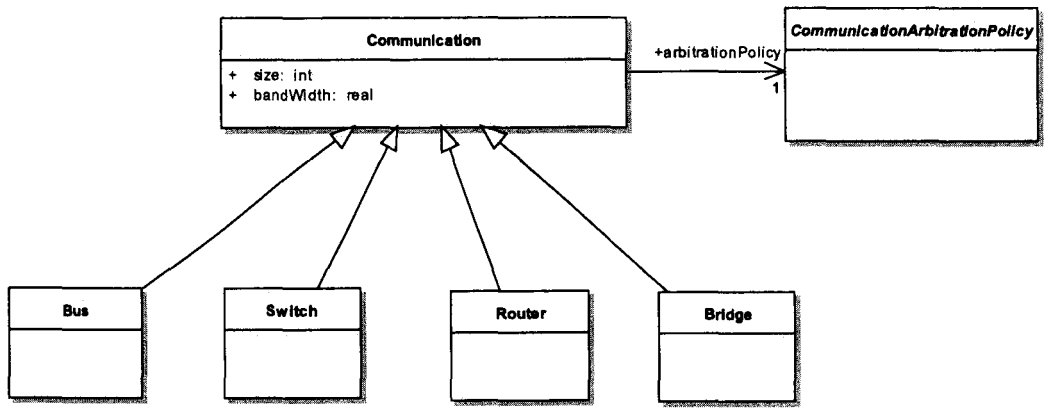


FIG. 4.37 – Raffinement des ressources de type Communication

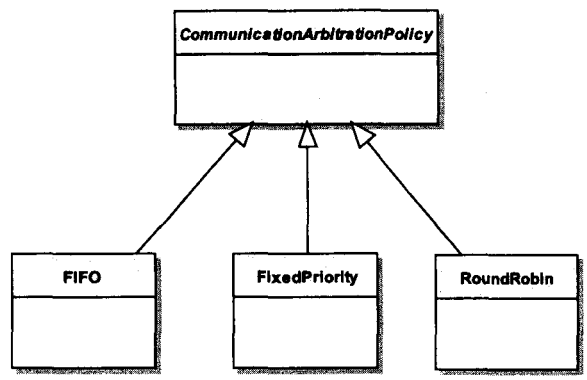


FIG. 4.38 – Politiques d'arbitrage

4.1.2 Communications

Sur la base des mécanismes décrits dans le package `component`, les composants matériels communiquent par émission/réception de requêtes à leur environnement. Compte tenu du niveau d'abstraction, on ne considère que deux types de requêtes : les requêtes de type `Send` et `Receive`. Par le biais d'une requête de type `Send`, un composant émet des données à son environnement, pour écriture si la requête est émise à une mémoire, ou éventuellement pour traitement si la requête est émise à un processeur. Par le biais d'une requête de type `Receive`, le composant requiert des données à son environnement. Si la requête est émise à une mémoire, la requête correspond à une lecture. De façon générale, une requête constitue une abstraction d'un protocole de communication. Au niveau RTL, l'émission d'une requête sera implantée sous la forme d'un échange d'un certain nombre de signaux (signaux d'adresse, de données, de contrôle, d'acquittement, etc...).

Par construction, un composant est capable d'émettre des requêtes via ses ports ayant une interface `REQUIRED`, et est susceptible de recevoir des requêtes via ses ports ayant une interface `PROVIDED`. On retrouve ici la sémantique de type "slave/master" évoquée dans la section relative au package `component`. Typiquement, une mémoire de type RAM ne disposera que de ports avec interface `PROVIDED` et jouera un rôle d'esclave : elle propose des services de lecture et d'écriture à son environnement, n'est pas en mesure d'émettre elle-même des requêtes à d'autres composants, et est donc considérée comme jouant un rôle passif dans les transactions (elle subit les transactions). A l'inverse, un processeur disposera de ports avec interface `REQUIRED` et jouera le rôle de maître : il est capable d'émettre des requêtes de lecture ou d'écriture à des mémoires, et est donc considéré comme actif dans les transactions (il initie les transactions).

Les concepts de `SendRequest` et de `ReceiveRequest` permettent respectivement de définir des requêtes de type `Send` et de type `Receive`, une interface matérielle (`HardwareInterface`) ne pouvant être composée que de requêtes de ce type (fig.4.39). Ces deux types de requêtes étendent le concept de `HwRequest`, caractérisé par l'attribut `coding : string`. Pour une requête donnée, cet attribut permet de définir le codage utilisé lorsque l'implantation de l'interface est réalisée pour un protocole particulier. Le concept de `Protocole` est l'abstraction d'un protocole physique de communication entre ressources. Ces protocoles sont généralement décrits au niveau RTL. Le concept de `Protocole` est caractérisé par les attributs suivants :

- **addressPorts : int**, le nombre de ports (on parle également de signaux ou de bits) utilisés pour le codage des adresses.
- **dataPorts : int**, le nombre de ports utilisés pour le codage des données
- **commandPorts : int**, le nombre de ports utilisés pour le codage des commandes. L'attribut `coding` de `HwRequest` est un codage binaire de longueur `commandPorts`. Dans le codage, la valeur `1` pour un port d'indice `i` signifie que ce port sera activé.
- **controlPorts : int**, le nombre de ports utilisés pour la gestion des signaux de contrôle.
- **testPorts : int**, le nombre de ports utilisés pour la gestion des tests, ou plus précisément les interactions entre le composant et l'environnement de simulation.

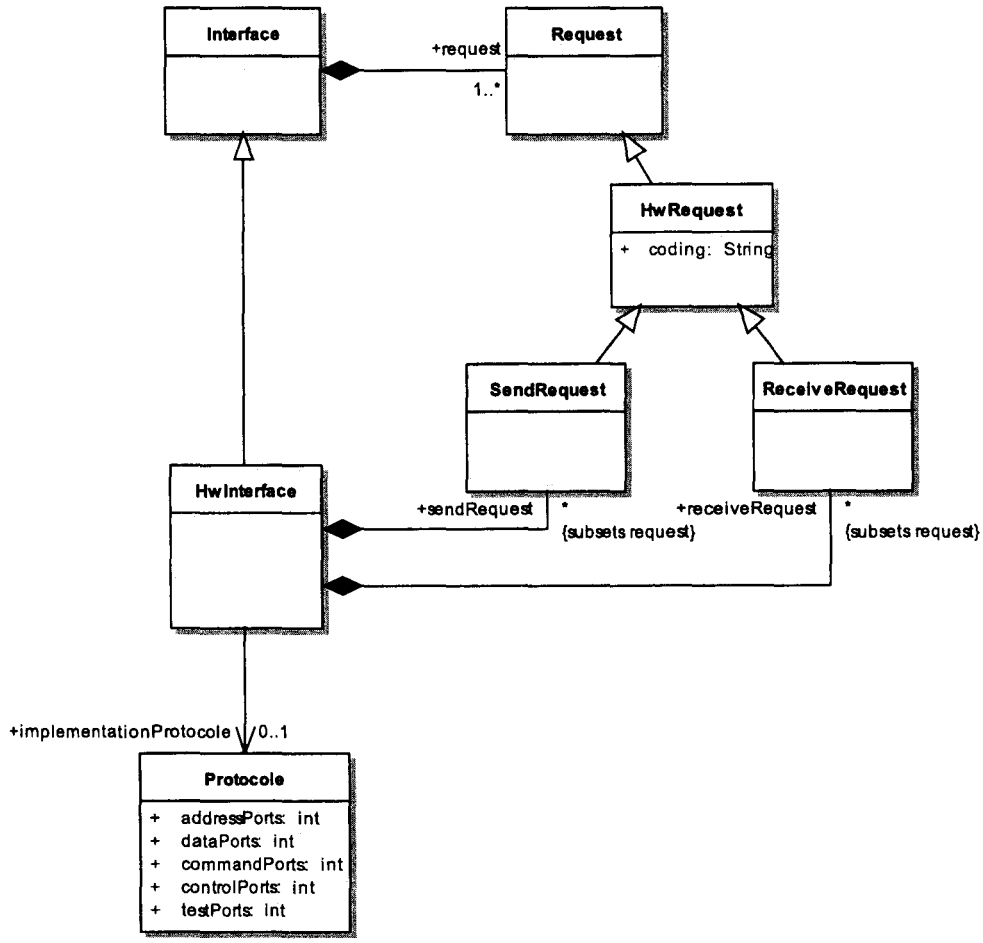


FIG. 4.39 – Interfaces pour les composants matériels

4.2 Vue UML

4.2.1 Modélisation des composants

Pour la modélisation des composants matériels, nous introduisons le stéréotype «HardwareComponent», qui étend le stéréotype «GaspardComponent», et s'applique à ce titre aux composants UML. Les différents raffinements du concept de `HardwareComponent` se retrouvent sous la forme de stéréotypes étendant «HardwareComponent» : «Processor», «Communication», et «Device» (fig.4.40). Pour chacun des différents raffinements exposés dans la vue domaine du profil, nous définissons également des stéréotypes[2].

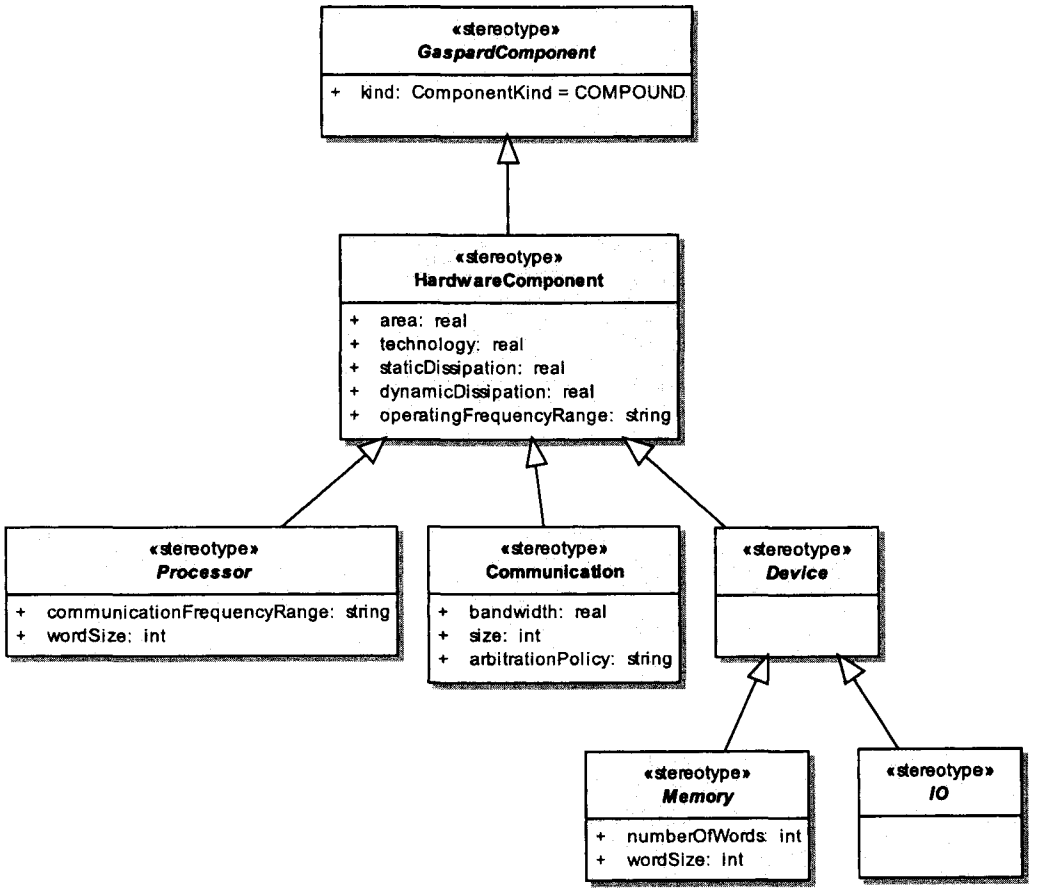


FIG. 4.40 – Définition du stéréotype «HardwareComponent»

A noter que les "politiques" associées aux caches et aux ressources de communication sont modélisées sous la forme de tagged values de type `string`. La valeur de ces tagged

[2]Nous ne détaillerons pas ces stéréotypes, puisqu'il existe pratiquement une équivalence un pour un entre les concepts définis dans la vue domaine, et les stéréotypes définis dans le profil.

BU LILLE 1

values fait référence aux noms donnés aux différentes politiques identifiées dans la vue domaine.

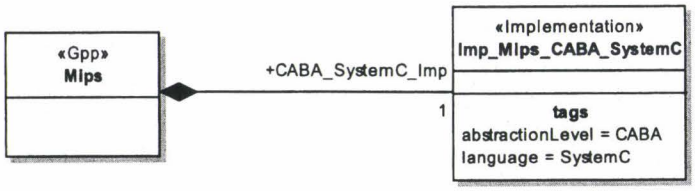


FIG. 4.41 – Exemple de définition d'implantation pour un composant matériel

Le concept de `Implementation` (associé aux composants matériels) est défini dans la vue UML par un stéréotype s'appliquant au concept de `Class`. En fig.4.41, nous illustrons la déclaration d'une implantation au niveau CABA dans le langage SystemC pour un composant de type Gpp nommé Mips.

FIG. 4.42 – Exemple de définition de jeu d'instructions pour un composant de type Gpp

De la même manière, le concept de `InstructionSet` est défini dans la vue UML par un stéréotype s'appliquant aux classes. En fig.4.42, nous illustrons la déclaration d'un jeu d'instructions nommé MipsIS. Le stéréotype «supportedIS» appliqué à la dépendance entre le composant et le jeu d'instructions permet de spécifier que le processeur généraliste Mips supporte le jeu d'instruction MipsIS.

4.2.2 Modélisation des interfaces

Dans la vue domaine du profil, nous avons défini le concept de `HwInterface`, pour la modélisation des interfaces des composants matériels. Nous ne définissons cependant pas de stéréotype «HwInterface», et imposons simplement la contrainte qu'une interface pour un composant matériel ne soit composée que d'opérations (pas de réceptions), stéréotypées «send» ou «receive». Chaque opération identifiée dans l'interface spécifie un service, pouvant être requis ou fourni par le composant en fonction de l'utilisation faite de l'interface. Si l'implantation de l'interface pour un protocole particulier est spécifiée (stéréotype «implementationProtocol» s'appliquant à une dépendance exprimée entre une interface et un protocole), le tagged value `coding` permet de spécifier un codage binaire de l'instruction

(ayant pour longueur la valeur du tagged value `commandPorts` du protocole) pour ce protocole. En fig.4.43, nous illustrons ce mécanisme via la définition d'un protocole nommé VCI, d'une interface générique MemoryServices, et d'une interface spécialisée VCIMemoryServices décrivant le codage des différents services dans le protocole VCI [62].

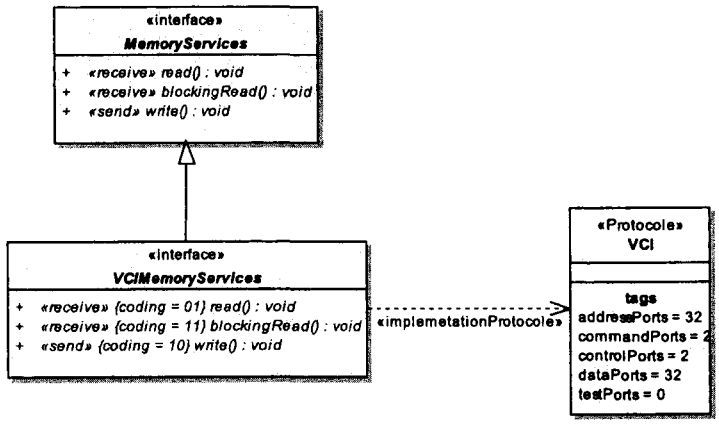


FIG. 4.43 – Exemple de définition d'interface pour un composant matériel

4.2.3 Exemple d'architecture : Quadri-processeurs Mips

L'exemple modélisé en fig.4.44 illustre la définition d'un composant de type `COMPOUND` nommé `QuadriProSoclib`. Il est composé de quatre processeurs généralistes Mips (pour lequel nous avons décrit dans les sections précédentes le jeu d'instruction et les implantations disponibles), disposant chacun d'un cache unifié nommé `XCache`. Les requêtes de services pouvant être émises par un processeur à son cache sont définies dans l'interface `VCICacheServices`, fournie par le cache et requise par le processeur. Le contrôleur du composant XCache peut quant à lui émettre des requêtes à son environnement via une interface `VCIMemoryServices` (un service de lecture, un service d'écriture), via son port nommé interconnect, connecté au composant de communication nommé `Crossbar`. Ce composant permet l'interconnection entre les différents caches, et les deux mémoires globales du système, `m0` et `m1` de type `SRam`. Les requêtes émises par les contrôleurs de cache sont simplement relayées par le composant d'interconnection aux deux mémoires.

4.3 Impact des mécanismes de factorisation

Dans le chapitre d'introduction de ce document, nous avons évoqué un certain nombre de tendances technologiques pour la conception des systèmes sur puce :

- Augmentation du nombre de ressources de calcul : Le parallélisme est un moyen naturel pour augmenter la puissance de calcul d'un système, mais permet également de diminuer la consommation d'énergie.

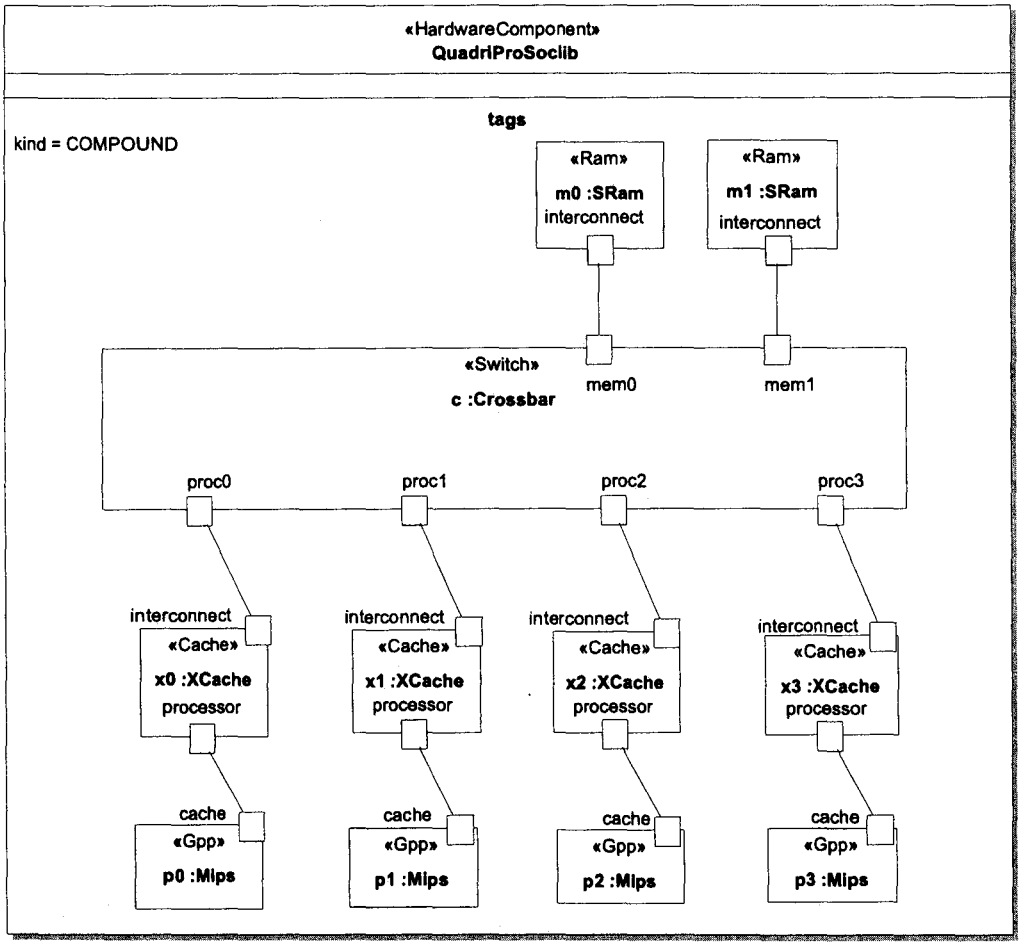


FIG. 4.44 – Architecture quadri mips basée sur la librairie SocLib

- Utilisation de réseaux (NoC) pour l'interconnexion des ressources : Les réseaux offrent potentiellement plus de scalabilité, et limitent les phénomènes physiques imprévisibles liés à l'utilisation des bus.

Par ailleurs, nous avons également évoqué la façon dont nous anticipons l'évolution des systèmes sur puce : les systèmes à venir seront non seulement des architectures multiprocesseurs, mais également multiprocesseurs homogènes et régulières. Par définition, les mécanismes de factorisation que nous proposons offrent un support puissant pour la modélisation de ce type d'architectures.

Pour illustrer l'utilisation des mécanismes de factorisation dans le package `HardwareArchitecture` du profil Gaspard, nous allons spécifier une architecture de type "tableau de processeurs". L'architecture est composée d'une répétition d'éléments de calculs (PE, pour Processing Element) sur deux dimensions. Chaque PE est constitué d'une unité de calcul, et d'une mémoire local. Les communications entre les différents PEs sont mises en oeuvre via l'utilisation d'un NoC avec une topologie de type "grille torique". La spécification sous une forme compacte et générique du NoC est illustrée en fig.4.45.

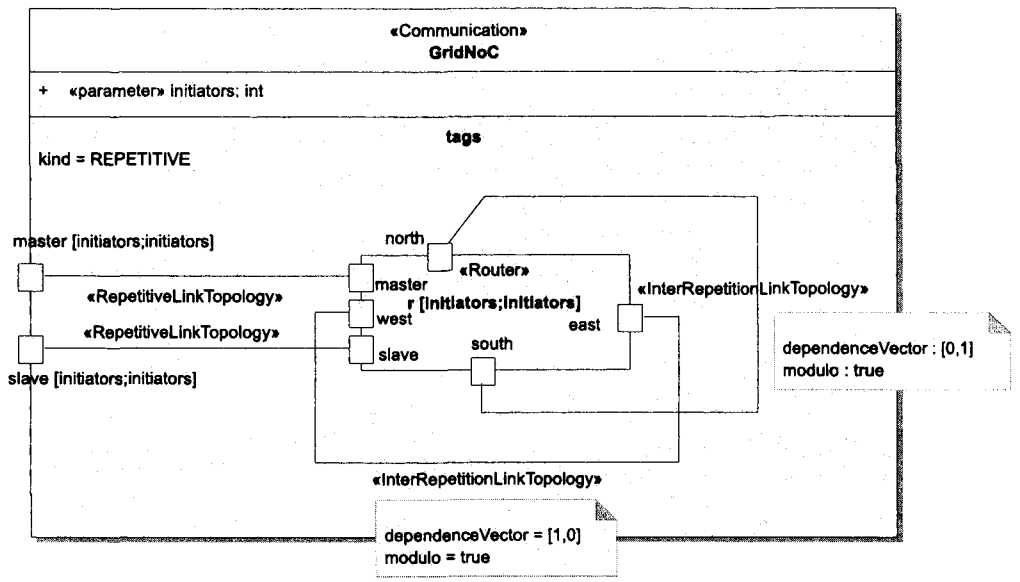


FIG. 4.45 – Spécification d'un réseau générique de type grille torique

La structure du NoC est spécifiée comme étant une répétition de routeurs sur deux dimensions (`GridNoc` est un composant de communication ayant une structure de type `REPETITIVE`), avec pour limite de répétition le paramètre `initiators`. Chaque routeur est connecté à ses voisins situés au nord, au sud, à l'est et à l'ouest, de façon torique. Cela signifie que pour les routeurs situés sur les "bords" du réseaux, les voisins sont situés sur le bord opposé. Cette topologie est spécifiée via l'utilisation de connecteurs stéréotypés «InterRepetitionLinkTopology», exprimé entre les bords de l'élément répété `r`. Chaque routeur

est par ailleurs connecté à un port `master` et à un port `slave` en bordure du composant `GridNoC`. Les ports `master` et `slave` du NoC possèdent également une multiplicité sur deux dimensions, avec pour limite de répétition la valeur du paramètre `initiators`. L'utilisation des connecteurs stéréotypés `RepetitiveLinkTopology` sans spécifications des différents paramètres (pavage, ajustage, origine,...) entre les ports de l'élément répété et les ports en bordure du composant signifie, comme nous l'avons évoqué dans la section consacrée au package `factorization`, que la topologie des connexions est triviale : chaque port d'indice `[i,j]` en bordure du composant est connecté au port de la répétition d'indice `[i,j]` correspondant.

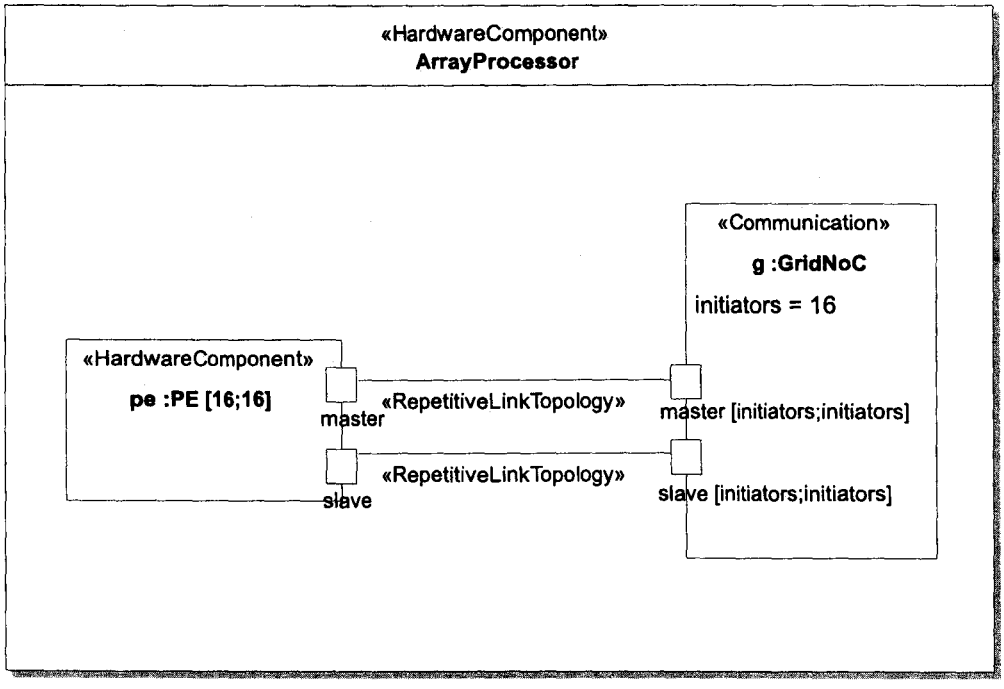


FIG. 4.46 – Spécification du tableau de processeurs

En fig.4.46, on constate que la structure du composant `ArrayProcessor` contient une répétition de PEs (16×16), tous connecté à un réseau `g : GridNoc`. On constate que le NoC est instancié avec pour valeur d'instance du paramètre `initiators` le nombre `16`. L'instance `g` est donc composée dans ce contexte d'une répétition de 16×16 routeurs. Par l'intermédiaire des ports `slave` et `master`, chaque PE est donc connecté à un routeur, et est en mesure d'émettre ou de recevoir une requête de ou vers chacun des autres PEs.

5 Package "application"

5.1 Vue domaine

Le paradigme sous-jacent à la partie application du profil Gaspard est la modélisation par unique expression des dépendances de données. Ce paradigme est directement dérivé d'ArrayOL, que nous avons présenté dans le chapitre 2. Comme nous l'avons évoqué dans la section décrivant l'architecture du profil Gaspard, le package `application` propose une approche orientée composant basée sur la définition du concept de `Component`, donnée dans le package `component`. Via l'utilisation des mécanismes de factorisation décrits dans le package `factorization`, le package `application` cible particulièrement la modélisation d'applications où les traitements sont intensifs et réguliers. Plus précisément, ces applications offrent potentiellement beaucoup de parallélisme de données. Concrètement, composants et dépendances de données (modélisées par des connecteurs, comme nous le verrons par la suite) permettent de spécifier complètement un algorithme, sans addition d'informations d'exécution parasites.

Un point important à souligner est que les applications modélisées en respectant ce profil ne donnent aucune information sur la façon dont seront exécutées les implantations dérivées des modèles. Une même modélisation pourrait donner lieu à une exécution séquentielle, pipelinée, ou encore data parallèle... Quel que soit le modèle d'exécution choisi dans les couches basses du Y de gaspard, la seule contrainte est le respect des dépendances de données qui ont été modélisées. Pour résumer, toutes les propriétés d'ArrayOL sont conservées.

5.1.1 Composants applicatifs

Une application est définie par une composition de composants applicatifs, ou `ApplicationComponent`. Une application est vue globalement comme un composant applicatif "racine". Le concept de `ApplicationComponent` étend le concept de composant (fig.4.47), en lui ajoutant simplement une connotation applicative (par opposition à matérielle). Ce concept n'introduit pas de nouveaux attributs, la différence majeure par rapport au concept parent se situant au niveau des contraintes de modélisation. Ces contraintes supplémentaires permettent d'interpréter les composants applicatifs comme des blocs fonctionnels. L'association entre les concepts de `ApplicationComponent` et de `InstructionSet` permet de spécifier que l'implantation d'un composant est disponible dans l'assembleur correspondant à un jeu d'instructions particulier, ou encore que l'on dispose des compilateurs nécessaires pour produire cet assembleur. Comme nous l'avons déjà évoqué dans la section relative à la description du package `hardwareArchitecture`, des mécanismes permettent de spécifier qu'un processeur supporte un ou plusieurs jeux d'instructions donnés. Pour la modélisation de l'association entre une application et une architecture, le fait de connaître du côté applicatif les jeux d'instructions disponibles pour une tâche et du côté matériel les jeux d'instructions supportés par un processeur permet de juger de l'opportunité du placement d'une tâche sur un processeur.

5.1.2 Communications

Les seuls types de transactions autorisés au niveau des interfaces des composants applicatifs sont des signaux. L'émission d'un signal s'interprète simplement comme l'émission

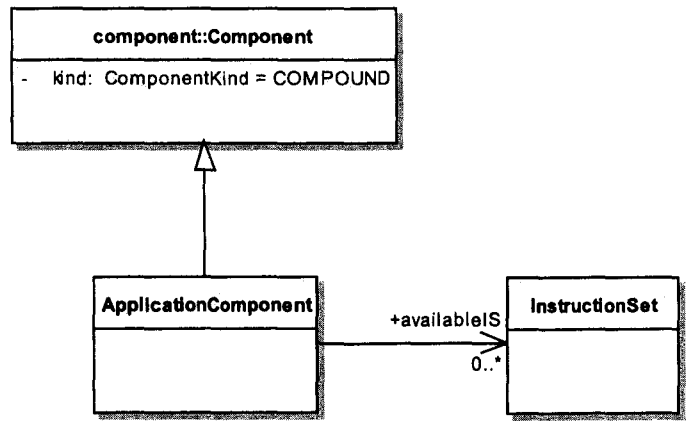


FIG. 4.47 – Définition du concept de ApplicationComponent

d'une donnée à l'environnement, sans attente bloquante de retour. Une structure décrite à base de composants applicatifs se construit comme un graphe orienté acyclique de tâches. L'échange de signaux entre composants applicatifs ne peut donc être qu'unidirectionnel. Ainsi, les ports ayant une interface `PROVIDED` peuvent être assimilés aux paramètres d'entrée d'une fonction (les ports sont en mesure de recevoir des signaux de leur environnement), et les ports ayant une interface `REQUIRED` peuvent être assimilés aux résultats produits par la fonction (les ports sont en mesure d'émettre des signaux à leur environnement).

Avec ou sans mécanismes de factorisation, la sémantique d'une application définie à l'aide d'`ApplicationComponents` reste fidèle à la philosophie d'Array OL. A savoir qu'une application modélisée à base de composants applicatifs n'impose à priori aucun modèle d'exécution. Compte tenu de la sémantique "fonctionnelle" induite par les contraintes de construction précédemment évoquées, les connecteurs peuvent être interprétés comme des dépendances de données. Pour une exécution donnée (c'est à dire indépendemment de toute répétition dans le temps), l'ordre dans lequel les signaux arrivent à un composant et sont ensuite éventuellement délégués aux parts qui le composent n'a pas d'incidence sur le résultat produit. La modélisation des dépendances (via la définition des connecteurs) n'impose qu'un ordre partiel d'exécution. La seule contrainte d'exécution concerne les composants applicatifs élémentaires (composant ayant l'attribut `kind` égal à `ELEMENTARY`, c'est à dire sans description structurelle par assemblage de parts), où l'on considère que tous les signaux d'entrée doivent être présents pour pouvoir exécuter le comportement associé au composant. Dans l'environnement Gaspard, l'implantation d'un composant applicatif élémentaire est effectivement définie sous la forme d'une fonction, produisant ses résultats sur la base des paramètres qui lui sont fournis en entrée.

La prise en compte du temps est quant à elle réalisée sous une forme particulière. Comme pour Array OL, le temps est banalisé dans la structure. Dans la section décrivant l'impact des mécanismes de factorisation pour la modélisation d'applications, nous décrirons ce point en détail.

5.2 Vue UML

5.2.1 Modélisation des composants

Le stéréotype «**ApplicationComponent**» est introduit pour la modélisation des composants applicatifs. Il étend le stéréotype abstrait «GaspardComponent» (fig.4.48), et s'applique à ce titre au concept UML de `Component`.

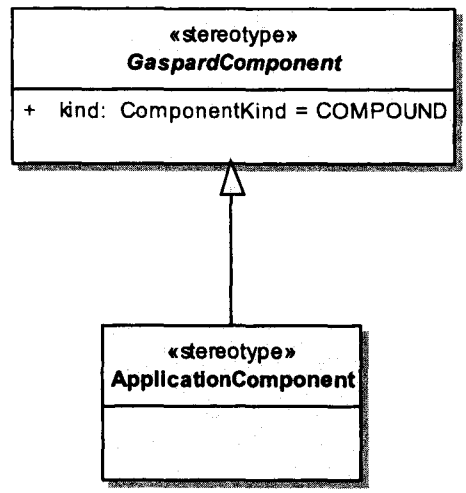


FIG. 4.48 – Stéréotype «ApplicationComponent»

De la même manière que pour les composants matériels, une dépendance stéréotypée permet de spécifier le lien entre un composant applicatif et un jeu d'instruction. L'utilisation du stéréotype «availableIS» est illustré en fig.4.49. On constate dans cet exemple que le composant applicatif `FFT` dispose potentiellement d'une implantation dans le jeu d'instructions `MipsIS`. Compte tenu de cette information, il pourrait être pertinent de placer un composant applicatif `FFT` sur un composant matériel `Mips` (que nous avons décrit dans la section consacrée au package `hardwareArchitecture`).

5.2.2 Modélisation des interfaces

La modélisation des interfaces des composants applicatifs suit le pattern de modélisation suivant. Chaque interface ne contient qu'une seule réception (nous avons décrit ce concept dans la section consacrée à la vue UML du package `component`). Le signal est quant à lui stéréotypé «TypedSignal», avec un tagged value `type : DataType`, représentant le type de données véhiculé par le signal, et manipulé par le composant. Le stéréotype «TypedSignal» est simplement une forme de raccourci, puisque l'on aurait pu tout aussi bien identifié le type de donnée par un attribut du signal. Si ce pattern peut sembler fastidieux à mettre en oeuvre, il s'avère en pratique relativement raisonnable, puisque les types de données manipulés par les composants sont souvent les mêmes : entiers, flottants, nombres complexes,...

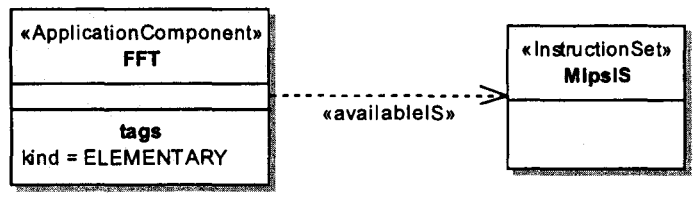


FIG. 4.49 – Spécification des jeux d'instructions disponibles pour un composant applicatif

5.3 Impact des mécanismes de factorisation

5.3.1 Banalisation des dimensions spatiales et temporelles

Comme nous l'avons évoqué dans la section consacrée à la description du package `factorization`, la répétition d'éléments structurels (port et part) est mise en oeuvre via l'utilisation de multiplicités multidimensionnelles. Une multiplicité est définie comme étant un vecteur d'expressions, une expression étant définie comme une abstraction d'une valeur numérique sous la forme d'une chaîne de caractères. Cette valeur numérique est potentiellement infinie. Cela signifie concrètement qu'il est possible de spécifier dans la structure du composant une infinité de répétition d'une part, ou encore une infinité de répétition d'un port. Dans le modèle d'architecture matérielle, cette propriété n'est pas utilisable, tout simplement parce que le nombre de ressources composant une architecture matérielle est fini. Pour le modèle d'application, cette propriété est par contre tout à fait raisonnable. Pour bien appréhender ce point, il convient de se rappeler l'objectif d'Array OL, et a fortiori l'objectif de notre modèle d'application : Exprimer tout le parallélisme potentiel d'une application. Pour une application où les calculs réalisés à l'instant `t` ne dépendent pas des calculs réalisés à l'instant `t-1`, l'ordre d'exécution n'a pas d'importance. Si tous les échantillons normalement obtenus au cours du temps étaient présents à un instant donné, tous les calculs pourraient être réalisés de façon parallèle. En d'autres termes, exprimer une boucle sur le temps est une façon de parasiter l'expression du parallélisme potentiel de l'application. Concrètement, l'utilisation d'une dimension infinie sur les ports d'un composant applicatif est une façon de modéliser l'ensemble des échantillons traités au cours du temps par un composant applicatif. L'utilisation des dimensions infinies pour la multiplicité des parts (qui va de paire avec l'utilisation des dimensions infinies sur les ports) revient à considérer le fait que si l'on disposait d'une machine proposant une infinité de ressources de calcul et de stockage, chaque part pour réaliser ses calculs parallèlement aux autres parts. La subtilité sémantique (que nous avons introduit dans la section consacrée à la description du package `component`) concernant la définition du concept de `Part` prend ici toute son importance : une part est définie comme étant une instance potentielle d'un composant, et non une instance concrète. Potentiellement, toutes les parts pourraient réaliser parallèlement leurs calculs. Physiquement, seul un nombre fini d'instances pourra exister simultanément. En fig.4.50, nous illustrons l'utilisation d'une dimension infinie sur les ports d'un composant, pour la modélisation non détaillée d'une application de traitement d'image. La part `s : Sensor` produit un flot infini de pixels, consommé par la part `i : ImageProcessing`,

qui produit elle-même un flot infini de pixels. Ces pixels sont ensuite consommé par la part `a : Actuator`. La structure du composant `ImageProcessing` est de type `REPETITIVE`, et contient une part `p` avec une multiplicité égale à `[*]`. Chaque instance potentielle est reliée à un port en entrée du composant (`pixelIn`) et à un port en sortie (`pixelOut`). Concrètement, un signal arrivant sur le port d'indice `i` est délégué à l'instance potentielle d'indice `i`, qui émettra elle même un signal correspondant au résultat calculé au port `pixelOut` d'indice `i`. Compte tenu du fait que le temps est matérialisé dans la structure, il est important de noter que chaque répétition d'une part ne réalisera qu'un et un seul calcul.

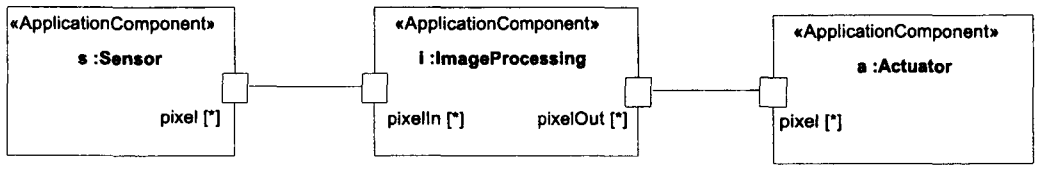


FIG. 4.50 – Exemple d'utilisation de dimension infini pour les ports d'un composant applicatif

5.3.2 Expression du parallélisme de données

L'expression du parallélisme de données est mise en oeuvre via l'utilisation du concept de `RepetitiveLinkTopology`, dans le contexte des composants de type `REPETITIVE`. Pour illustrer l'utilisation de ce mécanisme, nous reprenons l'exemple du produit de matrices (fig.4.51) illustré dans le chapitre positionnement, dans la section consacrée à la description du langage Array-OL.

Le composant `MatrixProduct` (structure de type `REPETITIVE`), possèdent deux ports `in1` et `in2` de multiplicité `[6;6]`, exposant l'interface `IInteger` de façon `PROVIDED`, et un port `out` également de multiplicité `[6;6]`, exposant la même interface mais de façon `REQUIRED`. La structure du composant `MatrixProduct` est constituée d'une part `d : DotProduct` de multiplicité `[6;6]`. Les tagged values associés au stéréotype `RepetitiveLinkTopology` du connecteur entre les ports `in1` de `MatrixProduct` et de `d` permettent de spécifier que les ports `in1` de chaque instance potentielle de `d` sont connectés à une ligne de ports `in1` du composant `MatrixProduct`. De la même manière, pour les ports `in2`, il existe un lien au niveau des colonnes, et pour les ports `out`, il existe un lien point à point. En d'autres termes, nous avons exprimé via cette modélisation tout le parallélisme de données potentiel du produit de matrice : Chaque instance potentielle de `d` est connectée à un ensemble de ports (motifs) à l'intérieur de chacun des ports de `MatrixProduct`.

5.3.3 Dépendances inter-répétitions

Les applications que nous avons considérés jusqu'à maintenant sont constitués de répétitions de tâches indépendantes les unes des autres. Il existe cependant beaucoup d'applications où les différentes répétitions ne sont pas nécessairement indépendantes. En d'autres termes, le résultat du calcul associé à une répétition peut dépendre du calcul réalisé

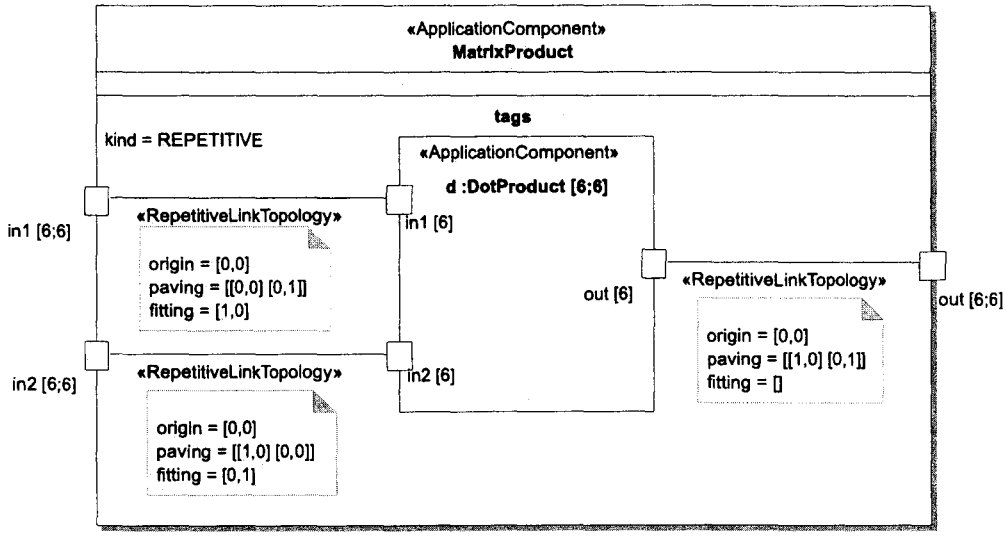


FIG. 4.51 – Modélisation du produit de matrices

par une autre répétition. Le stéréotype `InterRepetitionLinkTopology`, s'appliquant aux connecteurs, nous permet d'adresser également ce type d'applications, tout en gardant un mode d'expression similaire à celui que nous avons précédemment évoqué : répétitions d'éléments structurels (ports et parts) spécifiés par des multiplicités multidimensionnelles (potentiellement avec une dimension infinie pour la prise en compte du temps), et description des liens entre les parts via des connecteurs. En fig.4.52, nous illustrons l'utilisation de ce mécanisme par la modélisation d'un exemple académique : une réduction sur le temps.

Le composant applicatif `Reduction` (structure de type `REPETITIVE`) est composée d'une part `add : Add` de multiplicité `[*]`. Chaque part d'indice `i` est connectée au port `in` d'indice `i` du composant `Reduction` (pour les entrées), et au port `out` d'indice `i` pour les sorties. L'utilisation du connecteur stéréotypé `InterRepetitionLinkTopology` entre les ports `out` et `in2` de la part `add` spécifie que le résultat produit par une répétition d'indice `i` est utilisé comme seconde opérande par la répétition d'indice `i+1`. La valeur par défaut de la seconde opérande pour la première répétition est fournie en entrée du composant `Reduction`. Le connecteur stéréotypé `RepetitiveLinkTopology` entre le port `initialValue` du composant englobant et le port `in2` de la part `add` spécifie que seul le port `in2` de la part d'indice `0` est connecté au port `initialValue`. Cette part est donc la seule à prendre cette valeur pour seconde opérande. Dans cette exemple, il n'existe donc plus de parallélisme potentiel, l'exécution ne pouvant être que séquentielle. On pourrait cependant imaginer d'autres cas avec un espace de répétition multidimensionnel, une dépendance inter-répétitions sur une dimension, et du parallélisme potentiel sur les autres dimensions.

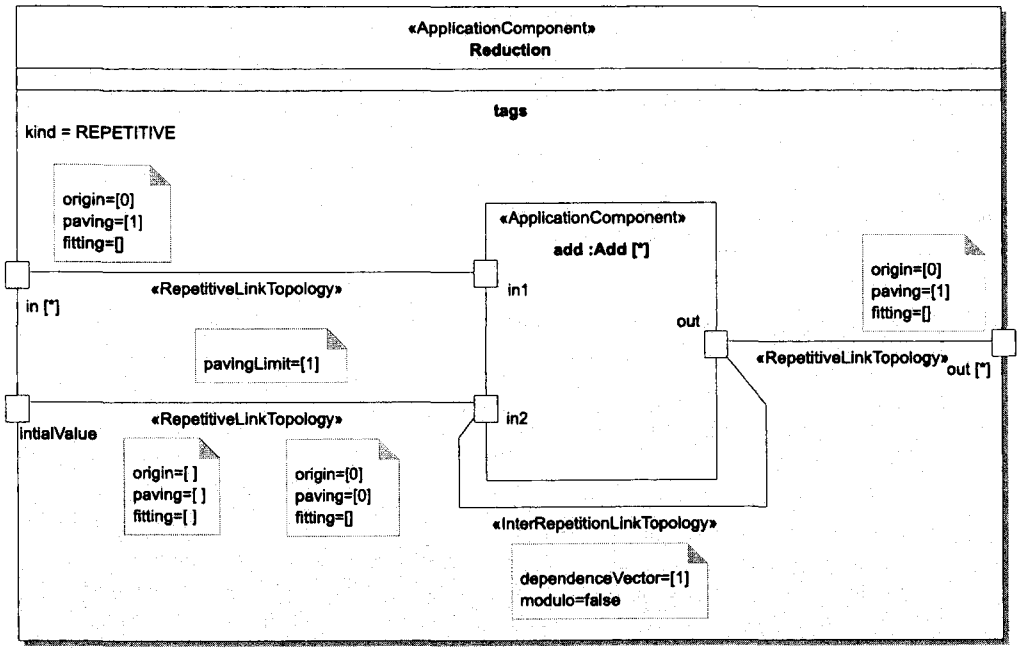


FIG. 4.52 – Modélisation de la réduction d'un vecteur

6 Package "association"

Dans les sections précédentes, nous avons décrit la manière de modéliser des applications et des architectures matérielles à l'aide du profil Gaspard. Il nous reste maintenant à prendre en compte le dernier aspect des couches hautes du Y de Gaspard : l'association. L'objectif du package `association` est de fournir les outils nécessaires pour établir un lien entre une application et une architecture matérielle utilisée comme support d'exécution. Concrètement, l'association englobe deux aspects : caractérisation et allocation. L'objectif de la caractérisation est de fournir des informations permettant d'orienter le placement d'une application sur une architecture, à la fois du côté de l'application (telles que le temps d'exécution maximal requis pour l'exécution d'une tâche, ou encore la quantité de mémoire nécessaire à l'exécution de la tâche) ou du côté de l'architecture (telles que le temps d'exécution qu'est capable de fournir un processeur pour l'exécution d'une tâche). Sur la base de ces informations, l'allocation consiste à définir un placement des calculs sur les ressources de calculs, et un placement des données sur les mémoires, tout en prenant en compte la hiérarchie et les aspects répétitifs du système (à la fois du côté de l'application et du côté de l'architecture)[3].

6.1 Vue domaine

6.1.1 Caractérisation

D'un point vue utilisateur, l'architecture matérielle d'un système peut être vue comme un ensemble de ressources offrant des services pour l'exécution de l'application (exécution des tâches, support des données). Ce constat, que nous avons établi dans le chapitre positionnement (section consacrée à la description du profil QoS), va nous permettre d'utiliser les mécanismes de définition de qualité de service et d'annotation des modèles définis dans le profil QoS pour la mise en oeuvre de notre processus de caractérisation. Plus les informations de caractérisation seront représentatives de la réalité (qu'elles soient le fruit d'une estimation ou d'une mesure), plus le placement sera orienté de façon pertinente. Du côté applicatif, on parlera de qualité de service requise : Les informations représentent des caractéristiques requises pour l'implantation de la fonctionnalité décrite dans l'application. Du côté matériel, on parlera de qualité de service fournie : Les informations représentent les caractéristiques que l'architecture matérielle est capable de fournir en supportant l'exécution de l'application. Dans le profil Gaspard, nous identifions trois catégories de qualités de service : `EnergyConsumptionQoS`, `DataStorageQoS` et `DelayQoS` (fig.4.53) [4].

▶ **EnergyConsumptionQoS** La catégorie `EnergyConsumptionQoS` vise à caractériser des éléments applicatifs ou matériels avec des informations relatives à la consommation d'énergie. Par exemple, le fait que l'implantation de la fonctionnalité décrite par une application doivent consommer au maximum une quantité donnée d'énergie peut faire partie du cahier des charges. Du côté matériel, on pourra spécifier que l'architecture est capable de supporter l'exécution de l'application en utilisant une certaine quantité d'énergie.

[3] Le placement des communications est pour l'instant négligé.

[4] Cette liste n'est pas exhaustive, et est donnée à titre d'illustration. On anticipe qu'elle sera complétée ou modifiée en fonction de la façon dont évolueront les outils gravitant autour de l'environnement Gaspard.

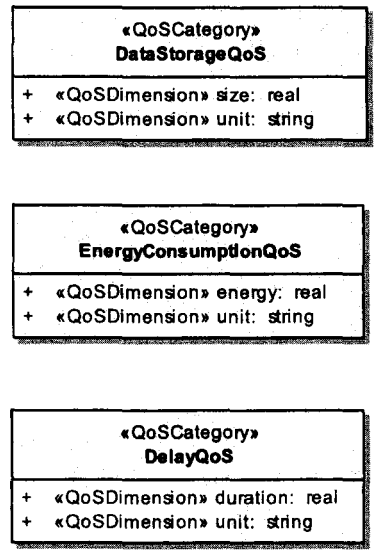


FIG. 4.53 – Exemple de catégories de QoS pour Gaspard

En fig.4.54, nous illustrons l'utilisation de ce mécanisme pour la caractérisation d'un produit de matrices 12×12 et de l'architecture quadri-mips présenté dans la section relative à la description du package `hardwareArchitecture`. La valeur de qualité de service requise par l'application est totalement arbitraire (l'application est ici spécifiée indépendemment de tout contexte réel), et est donnée en guise d'illustration. La valeur de qualité de service pour l'architecture matérielle a par contre été obtenue par une simulation au niveau CABA.

▶ **DataStorageQoS** La catégorie `DataStorageQoS` vise à caractériser des éléments applicatifs ou matériels avec des informations relatives au stockage de données. Du côté applicatif, la valeur prise en compte représente la taille mémoire nécessaire pour le stockage des variables locales et des résultats intermédiaires. Les données consommées (fournies par l'environnement) ou produites (fournies à l'environnement) ne sont pas prises en compte. On estime en effet que l'information est déjà présente dans le modèle d'application, puisque l'on connaît les dimensions des ports et par conséquent la quantité de données manipulée. Du côté matériel, la valeur prise en compte représente la taille de l'espace mémoire accessible par l'élément caractérisé, que les mémoires soient définies dans sa propre structure ou qu'elles fassent partie de son environnement.

▶ **DelayQoS** La catégorie `DelayQoS` vise à caractériser des éléments applicatifs ou matériels avec des informations relatives au délai pour l'exécution d'une tâche. Du côté de l'application, la valeur prise en compte représente en général la durée maximale d'exécution tolérée. Typiquement, dans une application de traitement vidéo tel qu'un décodage MPEG, le traitement d'une image ne doit pas excéder 1/30 seconde. Du côté de l'architecture

matérielle, l'information représente le temps d'exécution qu'est capable de fournir l'élément considéré, pour une application ou un composant applicatif donné.

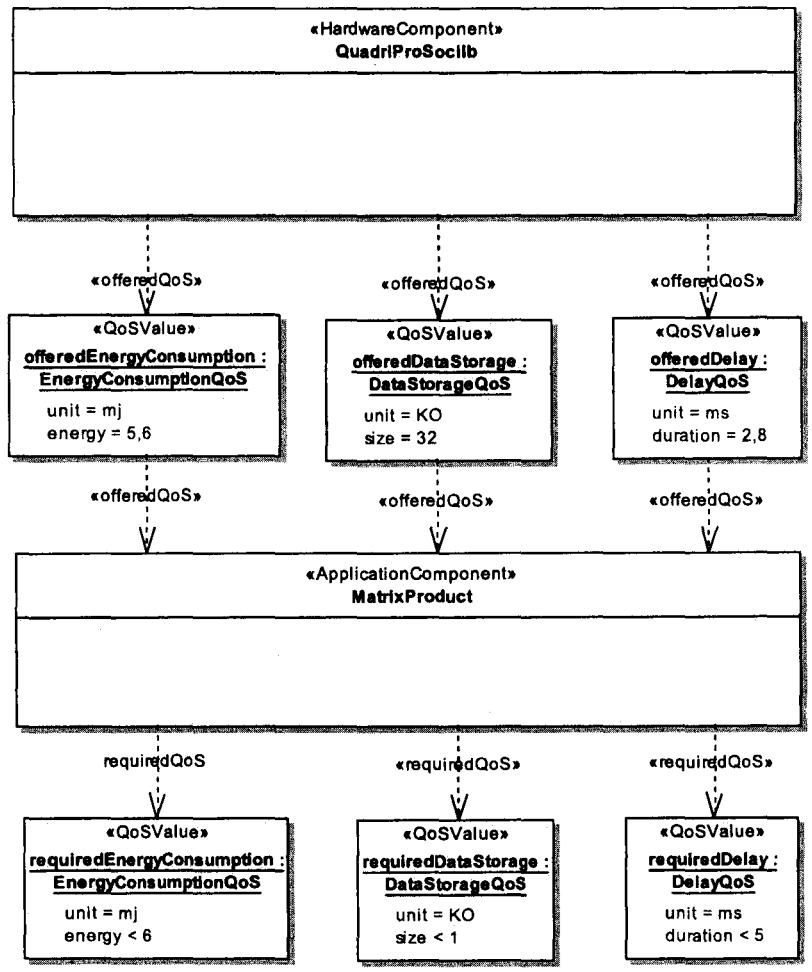


FIG. 4.54 – Exemple de caractérisation

6.1.2 Allocation

Dans le chapitre positionnement, nous avons présenté SysML et ses mécanismes d'allocation. Plus précisément, nous avons identifié le concept de `DeploymentAllocation`, utilisé dans SysML pour l'allocation d'éléments logiciels à des éléments matériels. Comme nous l'avons déjà évoqué, le concept de `DeploymentAllocation` est similaire au concept de `Deployment` dans UML, mais couvre un spectre d'utilisation plus large et offre plus de flexibilité. Pour la mise en oeuvre de notre processus d'allocation, nous allons donc considérer comme élément de base le concept de `DeploymentAllocation`. Dans le profil Gaspard,

l'allocation de l'application sur l'architecture revêt deux aspects : le placement des données et les placements des calculs. Ces deux aspects donnent respectivement lieu à deux raffinements du concept de `DeploymentAllocation : DataAllocation` et `TaskAllocation` (fig.4.55).

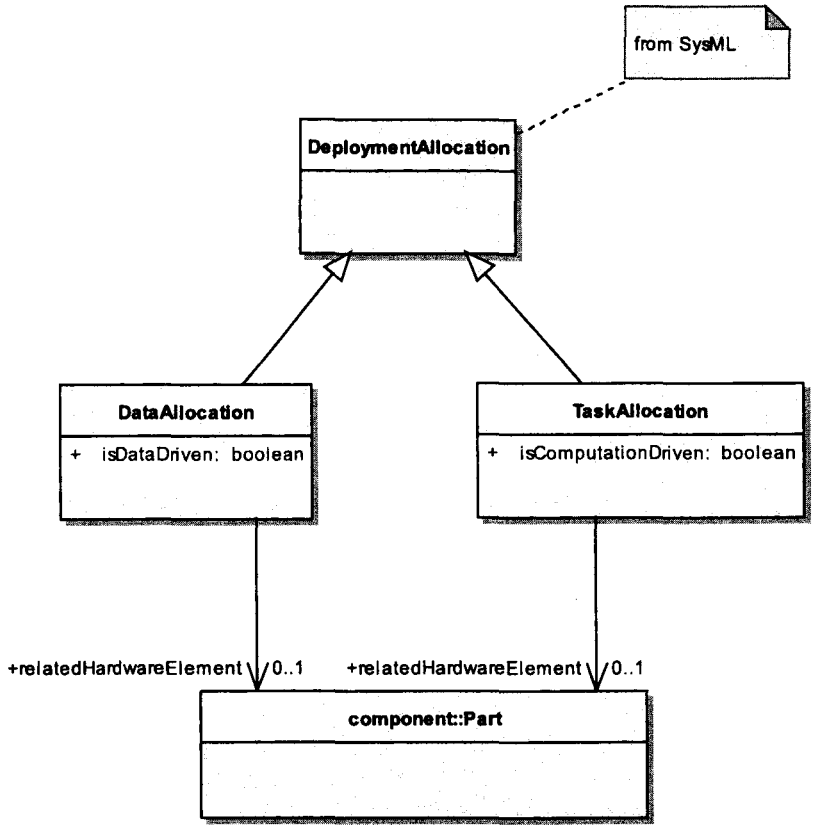


FIG. 4.55 – Définition des concepts de `DataAllocation` et `TaskAllocation`

▶ **DataAllocation** Comme nous l'avons déjà évoqué dans la section décrivant le package `application`, les données manipulées dans une application ne sont pas directement modélisées. Leur présence est plutôt suggérée par la présence des ports des composants applicatifs, points par lesquels sont véhiculés les signaux "transportant" les données entre les différents composants. Exprimer un placement des données va donc revenir à exprimer un lien d'allocation de type `DataAllocation` entre les ports des composants applicatifs et les composants matériels de type `Memory`. L'attribut `isDataDriven : boolean` spécifie si le placement exprimé est dirigé par les données, c'est à dire si les calculs vont "suivre" les données. Dans le cas où `isDataDriven = true`, le `DataAllocation` fait référence à

une part (rôle `relatedHardwareElement dans fig.4.55`) représentant une ressource de calcul. Cela signifie que les calculs consommant ou produisant les données placées (en d'autres termes la part qui possède le port qui a été placé) seront placés sur cette ressource de calcul. Ce mécanisme est évidemment bien adapté aux architectures à mémoire distribuée, où chaque élément de calcul dispose d'une mémoire local. Il permet d'une part de faciliter la modélisation du placement, et d'autre part de limiter les risques d'erreur dans la modélisation (données placées sur une mémoire inaccessible par la ressource sur laquelle sont placés les calculs).

▶ **TaskAllocation** Dans un modèle décrivant une application, les calculs sont représentés par les instances potentielles des composants. Exprimer le placement des calculs revient donc à exprimer l'allocation des parts d'une application sur les parts disposant d'une capacité de calcul (intrinsèquement ou dans leur structure) d'une architecture matérielle, via l'utilisation d'une dépendance d'allocation de type `TaskAllocation`. L'attribut `isComputationDriven : boolean` spécifie si le placement est dirigé par les calculs. Si `isComputationDriven = true`, le `DataAllocation` fait référence à une ressource de stockage de données. Cela signifie que les données consommées et produites par le composant (identifiées par ses ports) seront placées sur cette ressource de stockage.

6.2 Vue UML

Pour la modélisation des caractérisations dans la vue UML du profil, nous utilisons les mécanismes évoqués dans la section décrivant le profil QoS (chapitre positionnement), et illustré en fig.4.55 (définition des catégories de QoS dans Gaspard) et fig.4.54 (annotation des modèles avec des valeurs de QoS).

Pour la modélisation des allocations, nous définissons deux stéréotypes, «DataAllocation» et «TaskAllocation» s'appliquant au concept de `Dependency` (4.56). Pour chacun de ces deux stéréotypes, dans le cas où le placement est dirigé par les données ou par les calculs (tagged values `isDataDriven` et `isComputationDriven`), le tagged value `relatedHardwareElement : string` fait référence au nom de la part sur laquelle un placement des données ou des calculs est implicitement effectué.

6.3 Impact des mécanismes de factorisation

Dans les sections précédentes, nous avons montré comment les mécanismes de factorisation permettait de décrire de façon compacte les topologies des liens entre des éléments structurels ayant des multiplicités multidimensionnelles. Les concepts de `DataAllocation` et `TaskAllocation` sont mis en oeuvre, comme nous venons de l'évoquer, par l'utilisation de liens (`Dependency`) stéréotypés. Pour exprimer le placement régulier d'une application sur une architecture, nous allons utiliser les mêmes mécanismes de factorisation, en ajoutant des informations topologiques sur les dépendances stéréotypées «DataAllocation» et «TaskAllocation». Nous allons montrer comment ces mécanismes peuvent être utilisés pour mettre en œuvre trois patterns classiques de distribution : bloc, cyclique, et cyclique(k). Pour illustrer l'utilisation de ces trois patterns, nous allons nous baser sur une application composée de 32×32 répétitions d'une tâche élémentaire, et une architecture composée de 4 processeurs. Pour chaque pattern de distribution, on considérera une distribution par colonnes des tâches. L'idée commune est que du côté matériel, chaque

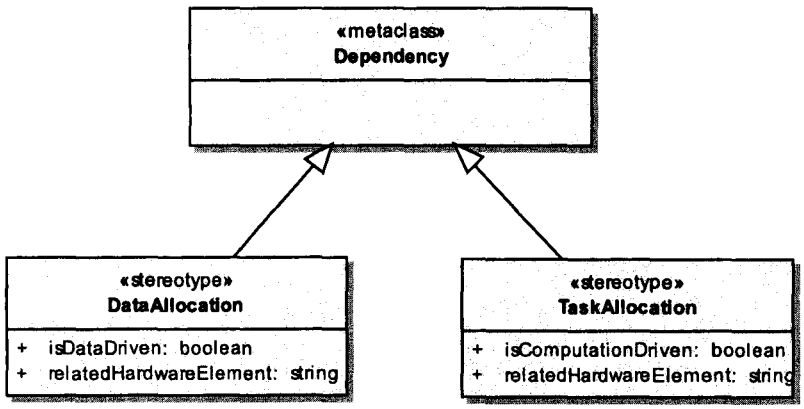


FIG. 4.56 – Définition des stéréotypes de `DataAllocation` et `TaskAllocation`

motif considéré est composé d'un seul processeur, soit 4 motifs au total. Pour la modélisation des trois patterns, la différence résidera donc dans la façon de construire les quatre motifs du côté applicatif. Les différents exemples illustrent l'utilisation du stéréotype «TaskAllocation». A noter que les mécanismes de factorisation s'appliquent de la même manière aux dépendances stéréotypées «DataAllocation»

► **Allocation par blocs** Définir une allocation de tâches par blocs revient à identifier `n` blocs (`n` = nombre de processeurs) parmi un ensemble de tâches (que l'on représentera par un tableau). Tous les blocs possèdent les mêmes dimensions (pour notre exemple 8×32), et sont compacts. Dans la section consacrée à la définition des mécanismes de factorisation, nous avons montré que la forme d'un motif est déterminée par l'ensemble des vecteurs d'ajustage. Définir un motif compact revient à considérer un ensemble de vecteurs d'ajustage correspondant à une matrice identité : Pour chaque dimension, le point suivant est obtenu par une translation de 1 à l'intérieur du tableau. Ainsi, dans fig.4.57, on constate que les vecteurs d'ajustage du côté de l'application ont pour valeur [[1,0][0,1]]. La limite du processus d'ajustage est simplement égale à [8,32], et correspond aux dimensions du motif considéré.

Pour identifier l'origine des différents motifs dans une distribution par blocs des colonnes (en considérant une répétition sur deux dimensions des tâches et sur une dimension des processeurs), le vecteur de pavage aura typiquement la forme suivante : `[c/p,0]`, avec `c` représentant le nombre de colonnes du tableau, `p` représentant le nombre de processeurs, en considérant `[p]` comme vecteur de limite d'itération de pavage. Pour notre exemple, on obtient ainsi le vecteur de pavage suivant : `[8,0]`, avec une limite de pavage égale à [4]. Cette forme d'expression peut être facilement adaptée à différents types de distribution par blocs (par exemple une distribution par ligne), et à différents types de multiplicités (plusieurs dimensions du côté de l'application comme du côté de l'architecture).

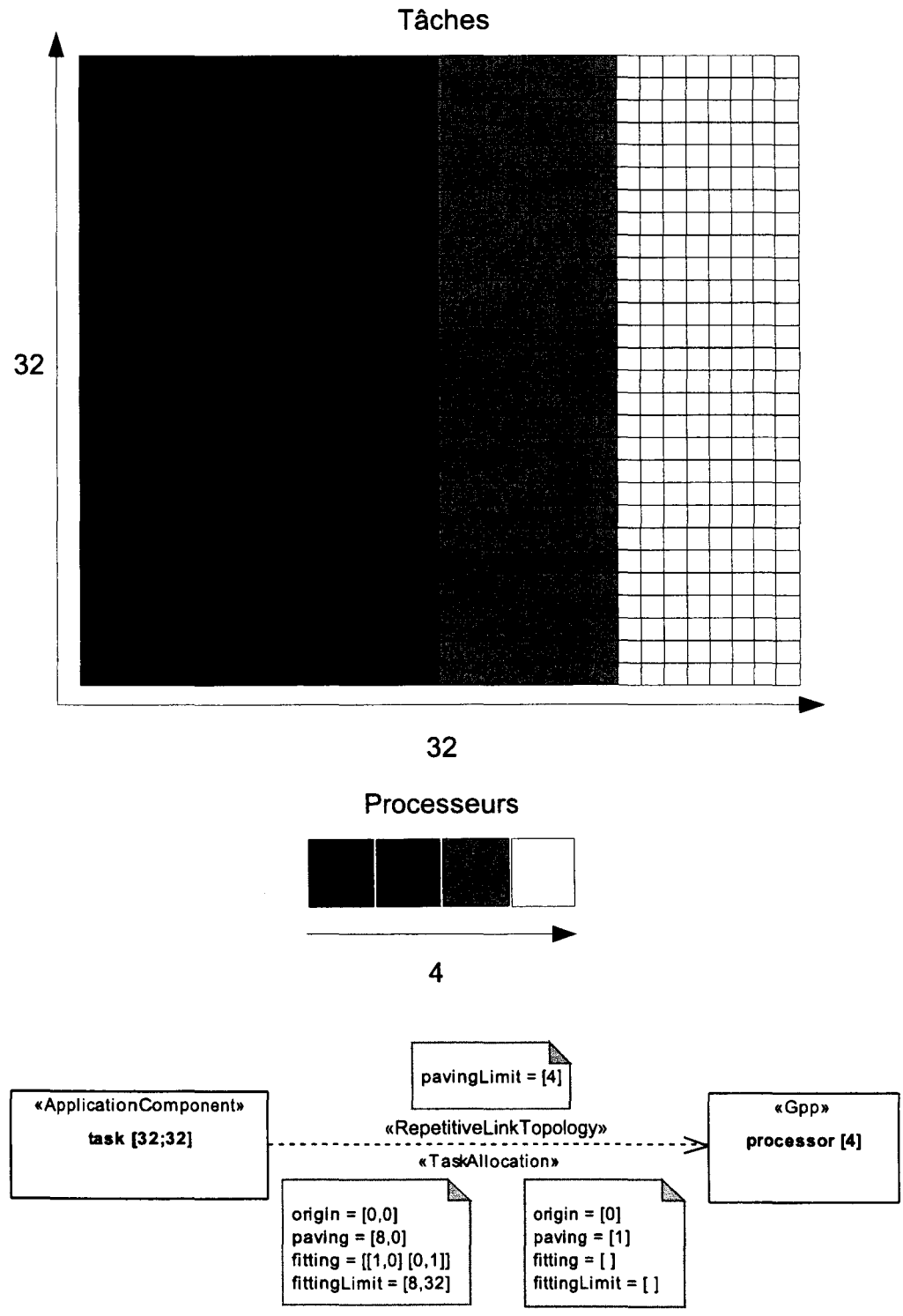


FIG. 4.57 – Allocation des colonnes par blocs

▶ **Allocation cyclique** Comme pour l'allocation par blocs, définir une allocation cyclique revient à identifier n motifs (n = nombre de processeurs) parmi un ensemble de tâches. Tous les motifs sont constitués du même nombre d'éléments (ici 8×32). Cependant, à la différence d'une distribution par blocs, les motifs considérés ne sont pas compacts (fig.4.58). Toute la complexité de la spécification d'une distribution cyclique réside donc dans la manière de spécifier les vecteurs d'ajustage. Pour la distribution cyclique par colonnes (en considérant une répétition d'une tâche sur deux dimensions et une répétition d'un processeur sur une seule dimension), les vecteurs d'ajustage auront typiquement la forme suivante : [[p,0] [0,1]], avec p représentant le nombre de processeurs, en considérant comme limites d'ajustage les dimensions du motif. Le premier vecteur permet d'identifier l'origine des différentes colonnes composant un motif (à partir d'une origine donnée par le processus de pavage), alors que le deuxième vecteur détermine les points appartenant à une colonne. Pour notre exemple, on obtient les vecteurs d'ajustage [[4,0] [0,1]], avec comme limite d'itérations le vecteur [8 ;32].

Comme on peut le constater en fig.4.58, le pavage est quant à lui trivial. L'origine des différents motifs (4 motifs dans notre exemple) est simplement obtenue par un décalage de 1 sur la dimension distribuée, soit le vecteur de pavage [1,0]. La combinaison des vecteurs de pavage et d'ajustage permet ainsi de spécifier un entrelacement des motifs.

▶ **Allocation cyclique(k)** L'allocation cyclique(k) est une généralisation de l'allocation cyclique présentée dans le paragraphe précédent (où k=1). Dans une distribution par colonnes, chaque "cycle" est constitué de k colonnes. Fig.4.59 illustre une distribution par colonnes des 32×32 tâches sur 4 processeurs en cyclique(4). La complexité de l'expression réside là aussi dans la spécification des vecteurs d'ajustage, qui auront typiquement la forme suivante[5] : [[pk,0] [1,0] [0,1]], avec comme limite d'itération d'ajustage le vecteur [c/(pk),k,l], où p représente le nombre de processeurs, c représente le nombre de colonnes, et l le nombre de lignes. Pour la définition d'un motif à deux dimensions, on constate l'utilisation de trois vecteurs d'ajustage. Pour une origine déterminée par le processus de pavage, le premier vecteur identifie l'origine d'un "cycle", le second vecteur identifie les colonnes appartenant à un cycle, et le troisième vecteur identifie les différents points appartenant à une colonne. Pour notre exemple, on obtient ainsi les vecteurs de pavage [[16,0] [1,0] [0,1]], avec comme limite d'itération d'ajustage [2,4,32]. "Répartir" l'espace de répétition sur 3 dimensions (au lieu de 2) permet simplement de faciliter l'expression de l'ajustage.

Concernant l'expression du pavage, on obtiendra typiquement la forme suivante : [k,0], avec p comme limite d'itération de pavage. Pour notre exemple, on obtient donc les valeurs suivantes : pavage égal à [4,0], et limite d'itération de pavage égale à [4]. En appliquant ce pattern à k=1, on obtient des valeurs équivalentes à celles définies dans le paragraphe précédent :

- limite de pavage : [4]
- pavage : [1,0]
- limites d'ajustage : [8,1,32]
- ajustage : [[4,0] [1,0] [0,1]] (Comme la limite d'ajustage pour le deuxième vecteur est égale à 1, ce vecteur n'est pas pris en compte dans le processus d'ajustage.)

[5] L'expression est simplifiée en considérant que le nombre de colonnes est un multiple de k.

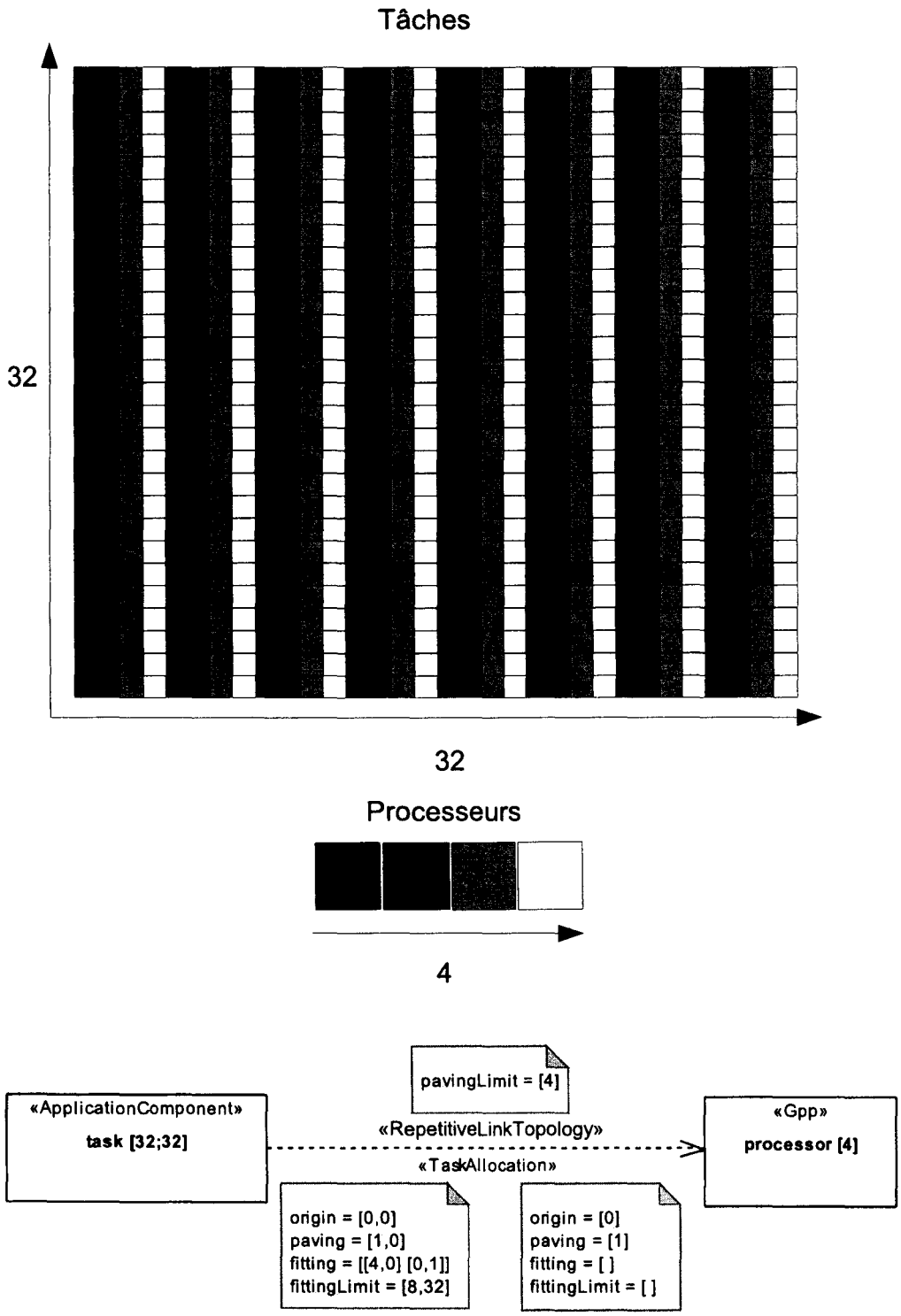


FIG. 4.58 – Allocation des colonnes cyclique

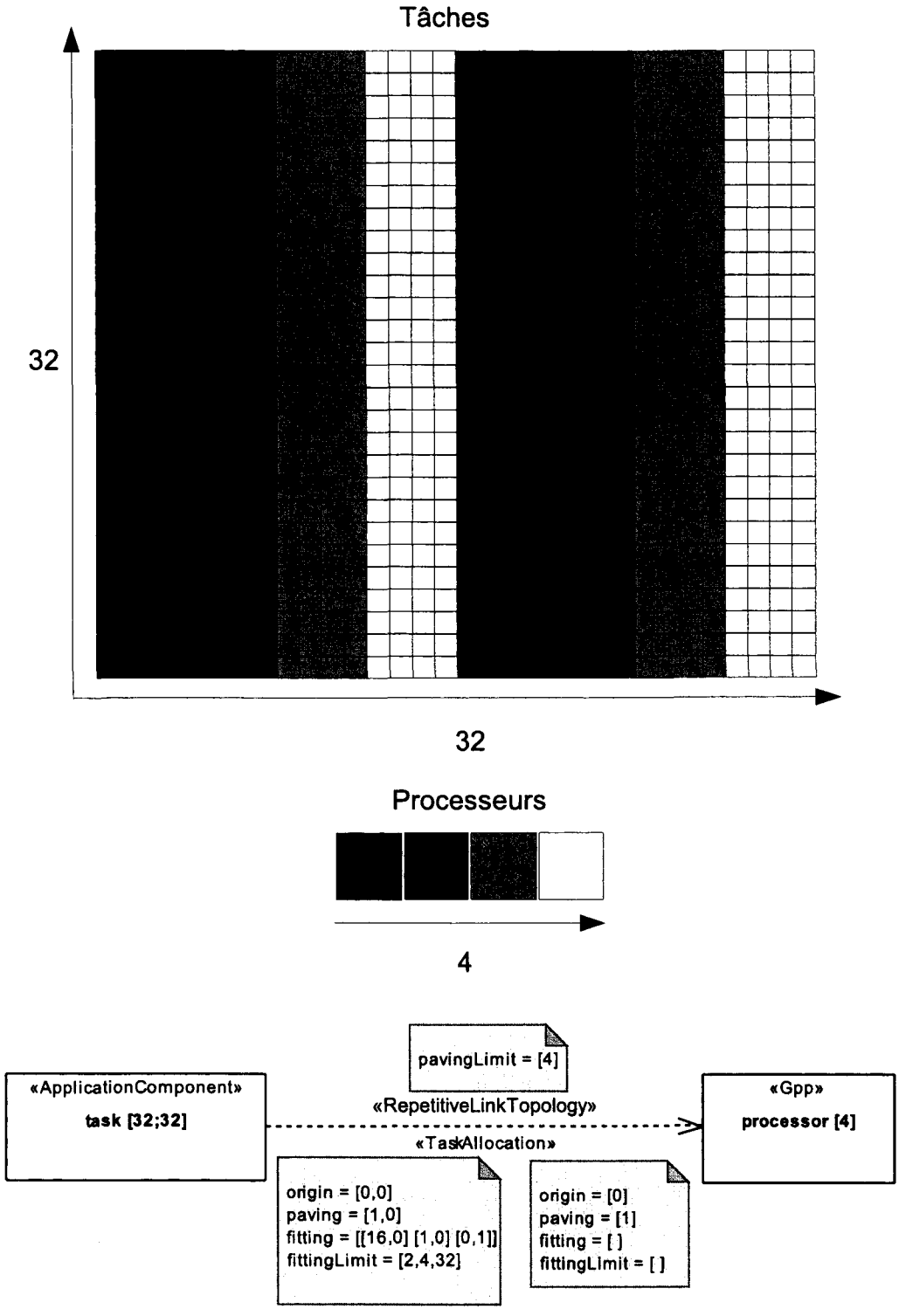


FIG. 4.59 – Allocation des colonnes en cyclique k (k=4)

CHAPITRE 5

Étude de cas

1 Application : DownScaler 125
1.1 Filtre horizontal 125
1.2 Filtre vertical 128
1.3 Structure de l'application 129
1.4 Optimisation 129
2 Architecture matérielle : Cluster de multi-mips 132
2.1 Nœud multi-mips 132
2.2 Réseau d'interconnexion Ω N×N 135
2.3 Cluster de multi-mips 137
3 Association 139
3.1 Dimensionnement de l'architecture et transformation de l'application 139
3.2 Modélisation du placement 141

Dans le chapitre précédent, nous avons proposé une spécification du profil Gaspard. L'objectif de cet avant dernier chapitre est d'illustrer l'utilisation des différents mécanismes que nous avons introduits, dans un exemple mettant en œuvre les trois aspects des couches hautes du Y de Gaspard : Application, Architecture matérielle, et Association.

1 Application : DownScaler

Pour illustrer l'utilisation des mécanismes de définition d'applications offerts par le profil Gaspard, nous allons modéliser une application de mise à l'échelle d'un flux vidéo décompressé en haute définition vers une définition standard : le DownScaler. Le principe de fonctionnement de l'application est illustré en fig.5.1.

Les images considérées en entrée du flux sont constituées de 1920×1080 pixels. Après application d'un filtre horizontal et d'un filtre vertical, les images produites dans le flux de sortie ne sont plus constituées que de 720×480 pixels. On a donc un facteur d'échelle horizontal de 8/3, et un facteur d'échelle vertical de 9/4. Pour le flux horizontal, 13 pixels sont en fait nécessaires pour le calcul de 3 pixels de l'image de sortie. La première étape du filtre consiste à calculer, pour 8 pixels, la moyenne des 6 pixels adjacents à chacun de ces pixels. Sur les 8 moyennes calculées, 3 pixels sont sélectionnés sur la base des coefficients du filtre. Ces coefficients sont déterminés au préalable, typiquement par des experts de traitement vidéo. Le principe du filtre vertical est similaire, sauf que l'on va considérer 14 pixels en entrée du filtre, calculer 9 moyennes, et retenir 4 pixels pour l'image de sortie. Cette application dispose potentiellement de beaucoup de parallélisme, puisque pour chaque filtre, les calculs des groupes de pixels pour l'image de sortie sont tous indépendants. Il s'agit donc d'une application privilégiée pour le profil Gaspard. Dans les sections qui suivent, nous allons dans un premier temps modéliser le parallélisme potentiel du filtre horizontal, dans un second temps le parallélisme potentiel du filtre vertical, et enfin la structure globale de l'application. Nous verrons qu'une première optimisation, indépendante de l'architecture matérielle visée, peut déjà être réalisée.

1.1 Filtre horizontal

Le composant `DataParallelHF` (fig.5.2) (structure de type `REPETITIVE`) définit l'ensemble du parallélisme potentiel pour le calcul des filtres horizontaux, pour tous les pixels traités dans le cycle de vie de l'application. Il expose de façon `PROVIDED` l'interface `IPixel` via son port `inputPixel [1924;1084;*]`. Il est ainsi en mesure de recevoir des signaux de type `Pixel` de son environnement. `DataParallelHF` expose la même interface de façon `REQUIRED` via son port `outputPixel [720;1084;*]`, et est ainsi en mesure d'émettre des signaux de type `Pixel` à son environnement.

Pour chacun des ports, les deux premières dimensions sont spatiales, et représentent le nombre de pixels considérés dans une image. La dernière dimension est quant à elle temporelle, et représente le flux potentiellement infini d'images. On constate que les dimensions spatiales sur la multiplicité du port `inputPixel` sont surdimensionnées (+4 sur chaque dimension) par rapport à la taille originale de l'image. Cette différence s'explique par l'ajout de valeurs neutres sur les bords de l'image, pour le calcul des moyennes sur les bords de l'image.

`DataParallelHF` ne contient dans sa structure (de type `REPETITIVE`) qu'une seule

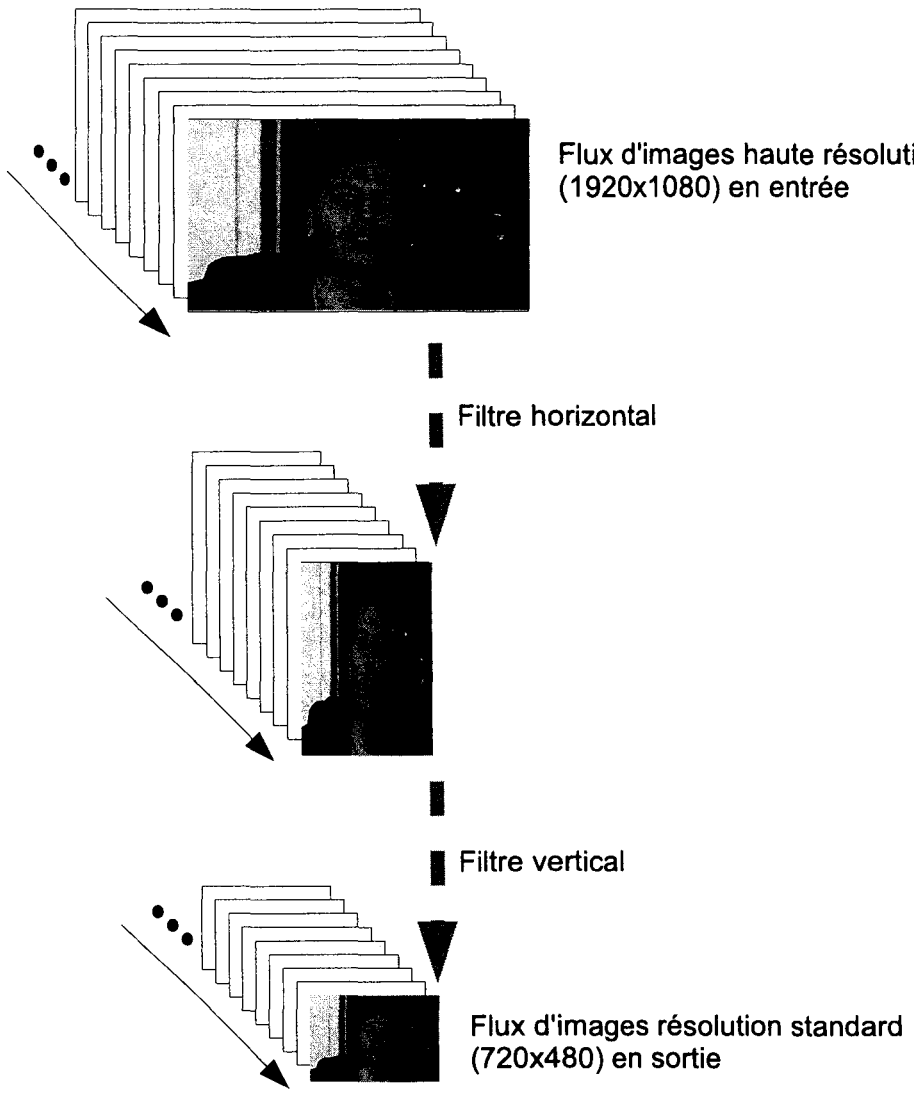


FIG. 5.1 – Principe du DownScaler

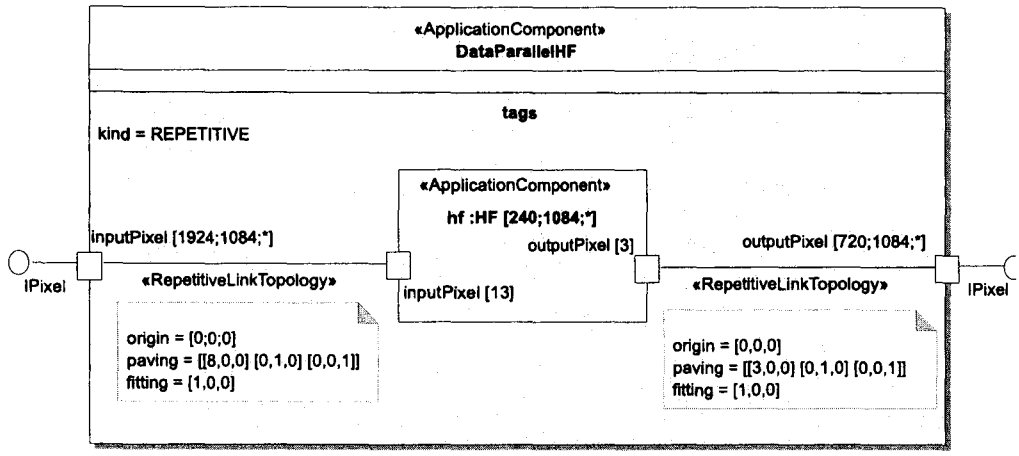


FIG. 5.2 – Filtre horizontal

part `hf : HF`, de multiplicité `[240;1084;*]`. Ce cas ne présentant pas de dépendances inter-répétitions, cela signifie que potentiellement, $240 \times 1084 \times \infty$ instances de `hf` pourraient réaliser leurs calculs en parallèle, dans un cas idéal où tous les pixels de toutes les images seraient disponibles simultanément. Les informations topologiques sur le connecteur entre les ports `inputPixel` de `DataParallelHF` et de `hf` identifient les motifs de ports connectés. On constate que les motifs se chevauchent sur la dimension horizontale, avec un vecteur de pavage égal à `[8,0,0]` et une limite d'ajustage égale à [13]. La limite d'ajustage correspond à la multiplicité du port `inputPixel` du composant `HF` (fig.5.3). Ce chevauchement s'explique par le fait que, comme nous l'avons évoqué dans l'introduction de la description de l'application, 13 pixels de l'image d'entrée sont nécessaires pour calculer la moyenne de 8 pixels, ces 8 moyennes étant elle-même utilisées pour produire 3 pixels de l'image de sortie. Les informations topologiques sur le connecteur entre les ports `outputPixel` de `hf` et `DataParallel` spécifient par contre un pavage compact par blocs de 3 ports.

Le composant `HF` (fig.5.3), avec une structure de type élémentaire, expose de façon `PROVIDED` l'interface `IPixel` via son port `inputPixel [13]`, et expose la même interface de façon `REQUIRED` via son port `outputPixel [3]`. Sur la base de 13 signaux de type `Pixel` reçus sur le port `inputPixel` (ou plus précisément un signal par instance de port), le composant `HF` applique le processus que nous avons déjà décrit : calcul de 8 moyennes, et sélection de 3 pixels d'après un filtre prédéfini. Les 3 pixels calculés sont transmis à l'environnement via l'émission de 3 signaux (un par instance du port `outputPixel`). A noter que le composant élémentaire `HF` dispose, comme illustré en fig.5.3, d'une implantation dans le jeu d'instructions `MipsIS`, supporté par le processeur `Mips`. Cette information nous servira pour le choix de l'architecture matérielle supportant l'exécution de l'application.

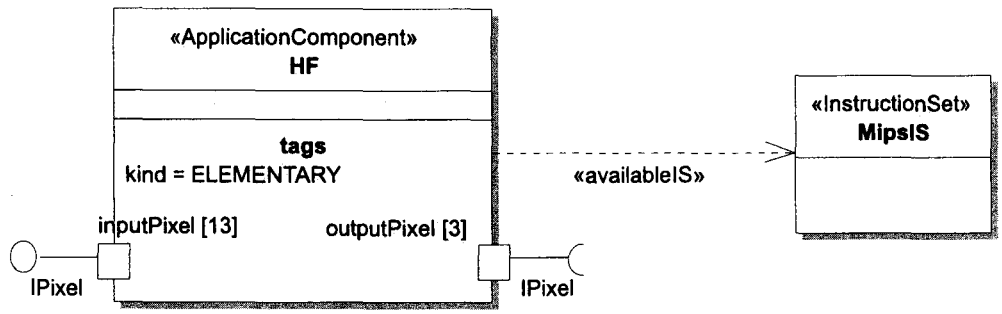


FIG. 5.3 – Composant élémentaire pour le filtre horizontal

1.2 Filtre vertical

Le composant `DataParallelVF` (fig.5.4) (structure de type `REPETITIVE`) définit l'ensemble du parallélisme potentiel pour le calcul des filtres verticaux, pour tous les pixels traités dans le cycle de vie de l'application. Comme pour `DataParallelHF`, il expose l'interface `IPixel` de façon `PROVIDED` via son port `inputPixel [720;1084;*]`, et de façon `REQUIRED` via son port `outputPixel [720;480;*]`. Les dimensions du port `inputPixel` correspondent aux dimensions d'une image après un filtre horizontal. Les dimensions du port `outputPixel` correspondent aux dimensions d'une image en résolution standard.

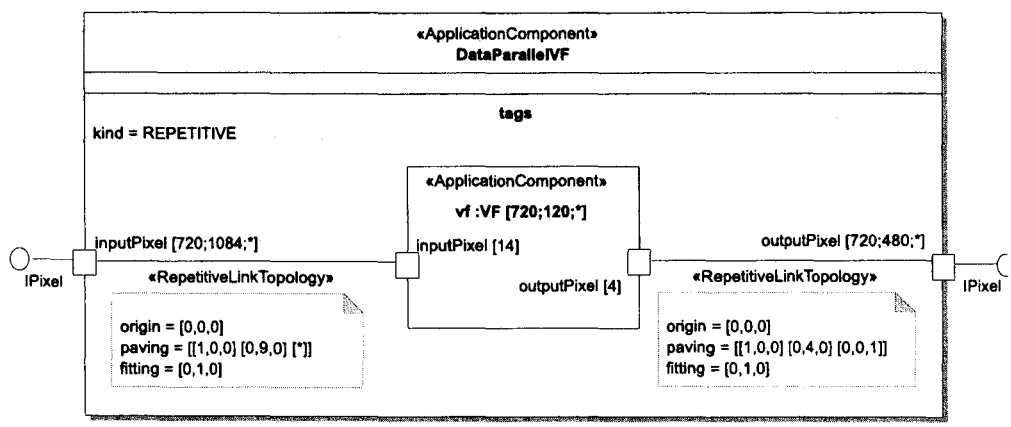


FIG. 5.4 – Filtre vertical

La structure de `DataParallelVF` suit un schéma similaire à celui de `DataParallelVF`. Elle ne contient qu'une seule part `vf : VF`, de multiplicité

[720;120;*]. Les informations topologiques sur le connecteur entre les ports inputPixel de DataParallelVF et de vf identifient les motifs de ports connectés. Les motifs se chevauchent sur la dimension verticale, avec un vecteur de pavage égal à [9,0,0] et une limite d'ajustage égale à [14]. La limite d'ajustage correspond à la multiplicité du port inputPixel du composant VF (fig.5.5). Les informations topologiques sur le connecteur entre les ports outputPixel de vf et DataParallel spécifient là aussi un pavage compact, par blocs de 4 ports.

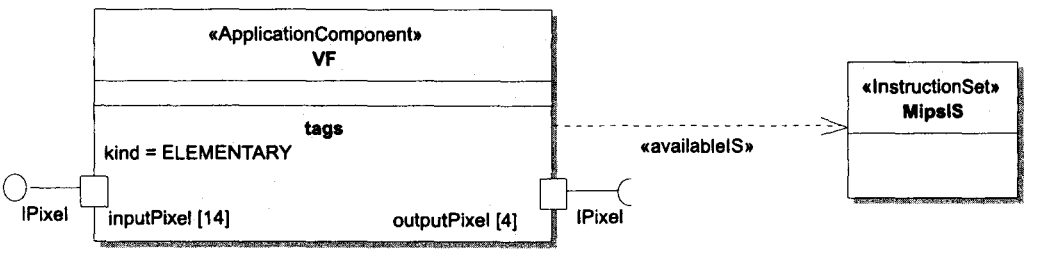


FIG. 5.5 – Composant élémentaire pour le filtre vertical

Le composant VF (fig.5.5), avec une structure de type élémentaire, expose de façon PROVIDED l'interface IPixel via son port inputPixel [14], et expose la même interface de façon REQUIRED via son port outputPixel [4]. Sur la base de 14 signaux de type Pixel reçus sur le port inputPixel, le composant VF applique le processus suivant : calcul de 9 moyennes, et sélection de 4 pixels d'après un filtre prédéfini. Les 4 pixels calculés sont transmis à l'environnement via l'émission de 4 signaux (un par instance du port outputPixel). A noter que comme pour HF, le composant élémentaire VF dispose, comme illustré en fig.5.5, d'une implantation dans le jeu d'instructions MipsIS, supporté par le processeur Mips.

1.3 Structure de l'application

Le composant DownScaler (fig.5.6), avec une structure de type COMPOUND, est basiquement composé d'un assemblage de deux parts, hf : DataParallelHF et vf : DataParallelVF, dont nous avons décrit la structure dans les sections précédentes. Les signaux en entrée du port inputPixel de DownScaler sont délégués à la part hf, qui leur applique un filtre horizontal. Les résultats obtenus par hf sont ensuite transmis sous forme de signaux à vf, qui leur applique un filtre vertical. Les résultats obtenus par vf sont enfin transmis sous forme de signaux aux ports outputPixel de DownScaler, et par conséquent transmis à l'environnement dans lequel sera instancié DownScaler.

1.4 Optimisation

Dans le modèle que nous venons de proposer, deux parts, elles-mêmes potentiellement constituées d'une infinité de filtres élémentaires, sont assemblés pour réaliser la mise à l'échelle d'un flux d'images haute résolution vers des images en résolution standard. Dans

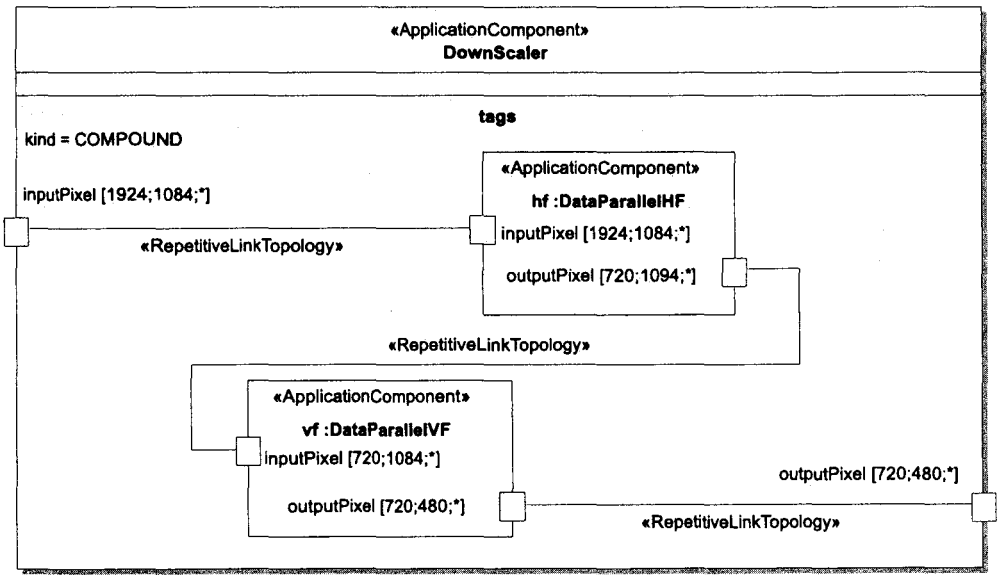


FIG. 5.6 – Structure de l'application Downscaler

un schéma de compilation simpliste, on pourrait imaginer projeter ce modèle vers une spécification où les parts `hf` et `vf` du composant `DownScaler` deviennent des fonctions, consommant des tableaux à trois dimensions, avec une dimension de taille infinie. Prises indépendemment, on pourrait imaginer que ces fonctions soient implémentées sous la forme d'une boucle imbriquée, avec deux indices de boucles bornés par les dimensions de l'image pour le balayage spatial, et un indice de boucle non borné sur le temps. Si l'on considère maintenant ces deux fonctions appliquées en séquence, on se rend compte que la deuxième fonction doit prendre en paramètre un tableau dont le calcul ne sera par définition jamais terminé par la première fonction (la boucle sur le temps n'est pas bornée). Pour résoudre ce problème, il est indispensable pour ces deux fonctions de partager la même boucle temporelle.

Les techniques d'optimisation brièvement évoquées dans la section consacrée à l'environnement Gaspard (chapitre introductif) proposent une solution à la résolution de ce problème, par des transformations appliquées à la structure d'un composant. Dans notre étude de cas, une transformation nommée "fusion" va nous permettre de fusionner les parts `hf` et `vf` du composant `DownScaler`, tel qu'illustré en fig.5.7, pour qu'elles partagent le même espace de répétition sur la dimension infinie. Un composant nommé `HFVF` (fig.5.8) a été généré par le processus de fusion. La structure du composant `DownScaler` est modifiée : elle est désormais de type `REPETITIVE`, et ne contient plus qu'une seule part de type `HFVF`, de multiplicité `[240;120;*]`.

`HFVF` contient un nombre fini de répétitions de filtres horizontaux (parts de type `HF`), et de filtres verticaux (parts de type `VF`). L'application de la fusion a donc permis de résoudre le problème de dimension infinie "intermédiaire". On constate par ailleurs au niveau de la

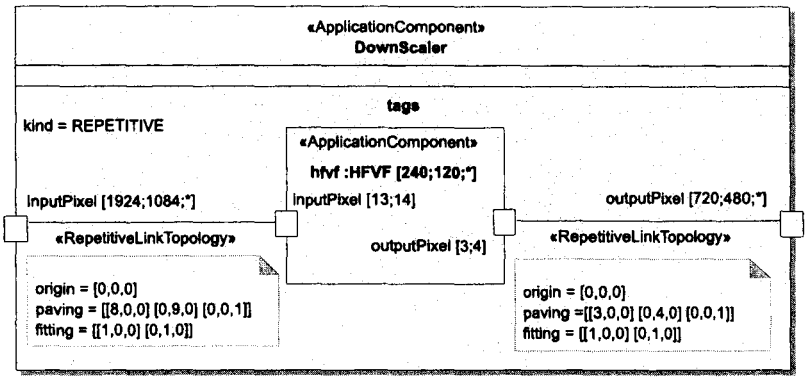


FIG. 5.7 – Structure de l'application Downscaler après fusion

multiplicité du port `inputPixel` de `HFVF` que la fusion a déterminé que la taille optimale des motifs est de 13×14. Sans rentrer dans les détails techniques de la transformation (disponibles dans [16]), ce résultat s'interprète intuitivement en considérant qu'un filtre horizontal a besoin de 13 pixels sur l'axe horizontal d'une image, et qu'un filtre vertical a besoin de 14 pixels sur l'axe vertical.

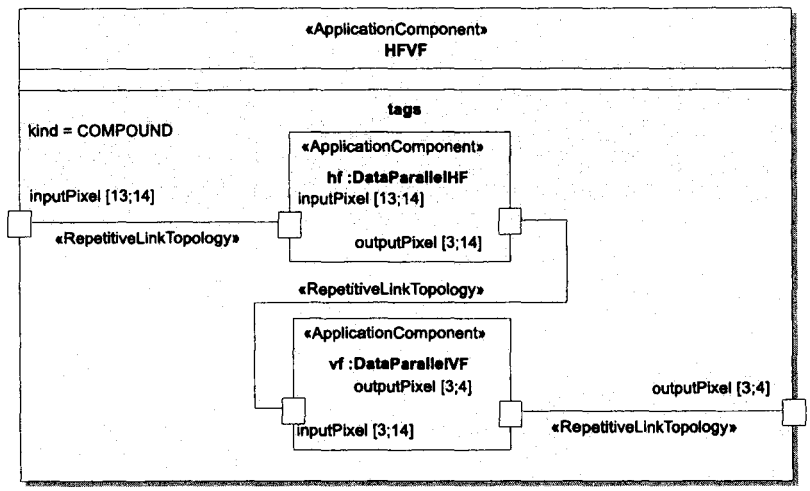


FIG. 5.8 – Structure du composant HFVF généré par la fusion

2 Architecture matérielle : Cluster de multi-mips

Dans cet exemple, nous proposons de nous baser sur une architecture générique et paramétrable pour supporter l'exécution du Downscaler. Ce type de méthodologie, où l'on dispose d'une solution architecturale prédéfinie que l'on va pouvoir dimensionner afin de l'adapter à une application particulière, est dite de type "platform-based". Dans ce type de méthodologie, la partie d'exploration d'architecture se limite à dimensionner correctement la plate-forme en fournissant des valeurs d'instance des paramètres adaptées à l'application visée. D'autres méthodologies existent, telles que des méthodologies dites "IP based", où l'architecture est construite sur mesure sur la base d'une exploration plus complexe à mettre en œuvre. Les IPs sont sélectionnées parmi une ou plusieurs bibliothèques et assemblées pour définir une solution adaptée au problème. A noter que les mécanismes offerts par le profil Gaspard ne sont liés à aucune méthodologie. Ils offrent simplement un support de modélisation. Les aspects méthodologiques sont plutôt liés à l'environnement et aux outils exploitant le profil.

Dans cette étude de cas, l'objectif n'est pas de définir une solution nécessairement réalisable d'un point de vue technologique, mais d'illustrer l'utilisation des mécanismes de :

- **hiérarchie**, pour la définition de la plate-forme par assemblage d'autres composants,
- **paramétrage**, pour définir la plate-forme de façon générique et adaptable,
- **factorisation**, pour modéliser de façon compacte la topologie de l'architecture.

Basiquement, nous allons nous inspirer de l'architecture quadri-mips décrite dans la section consacrée à la modélisation d'architectures matérielles, la généraliser (en la rendant paramétrable), et la réutiliser pour définir un cluster. Le choix d'une architecture parallèle à base de processeurs Mips comme support d'exécution du Downscaler est justifié par le fait que l'application offre potentiellement beaucoup de parallélisme, et que l'on dispose pour les différents composants élémentaires manipulés dans l'application d'une implantation pour le jeu d'instruction du Mips. Pour la mise en œuvre de l'interconnexion entre les différents nœuds du cluster, nous allons définir un réseau d'interconnexion multi-étages de type Ω (Omega). Ce réseau repose sur un pattern d'ISC (Inter Stage Connection, ou interconnexion entre étages) de type "perfect shuffle". Ce pattern a déjà été introduit dans la section consacrée à la description des mécanismes de factorisation du profil Gaspard. Le dimensionnement de l'architecture sera réalisé dans la dernière section de ce chapitre, lors de la définition de l'association entre le Downscaler et le cluster de multi-mips.

2.1 Nœud multi-mips

Les nœuds que nous allons considérer dans notre cluster sont inspirés du composant quadri-mips décrit en fig.4.44. L'idée est de considérer explicitement le facteur de répétition du Mips (et du cache auquel chaque Mips est connecté) via l'utilisation des mécanismes de factorisation. Concrètement, le composant `MultiMips` (fig.5.9)est composé d'une répétition de Mips (part `mips : Mips`) et de caches unifiés (part `xcache : XCache`), d'une mémoire locale (part `localRam : SRam`), et d'un composant d'interconnexion permettant notamment aux différents Mips d'accéder à la mémoire locale. Le composant `MultiMips` est rendu générique par l'utilisation de différents paramètres :

- numberOfMips : représente le facteur de répétition du Mips. Il est également utilisé comme facteur de répétition du cache. Le connecteur stéréotypé «RepetitiveLinkTopology» entre les parts `mips` et `xcache`, spécifié sans informations particulières, décrit

un pattern de connexion trivial de type un pour un entre chaque part `mips` et `cache` d'indice `i`.
- memorySize : la taille de la mémoire `sram` en nombre de mots.
- cacheSize : la taille des mémoires `xcache` en nombre de mots.

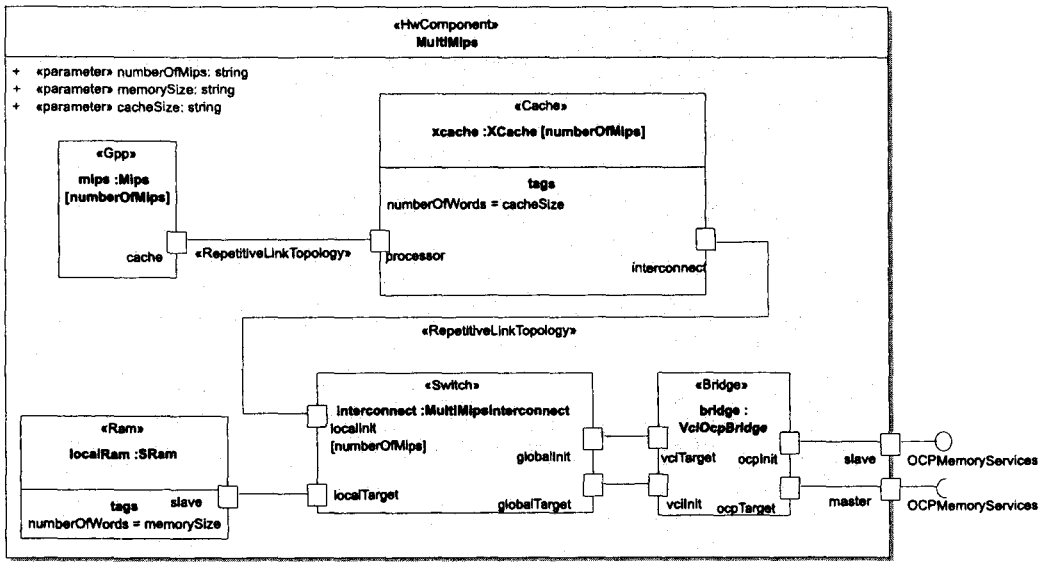


FIG. 5.9 – Définition du composant MultiMips

Les différents composants instanciés dans la structure de `MultiMips` communiquent sur la base d'une interface `MemoryServices`, exposant basiquement deux services : `read()` et `write()`. Comme on peut le constater en fig.5.10, cette interface est raffinée pour le support de deux protocoles de communications : `VCIProtocole` et `OCPProtocole`. Pour les communications locales au composant `MultiMips`, l'interface `MemoryServices` est utilisée. Pour les communications avec l'environnement du composant, l'interface `OCPMemoryServices` est utilisée, tout simplement parce que le composant d'interconnexion (que nous définissons dans la section suivante) utilisé pour la mise en œuvre du cluster expose cette interface.

Dans la structure du composant MultiMips, la part `bridge : VCIOCPBridge` (dont la définition est illustrée en fig.5.11) joue un double rôle. D'une part, elle permet de relier le composant d'interconnexion local au composant `MultiMips` (`interconnect : MultiMipsInterconnect`) au composant d'interconnexion globale du cluster. D'autre part, elle joue le rôle d'adaptateur entre les interfaces `VCIMemoryServices` et `OCPMemoryServices`. Concrètement, à ce stade de développement de l'étude, on ne dispose pas encore de l'IP `VCIOCPBridge`. Les informations données dans la description des protocoles et dans les interfaces (codage des requêtes) permettront ultérieurement de générer automatiquement l'adaptateur correspondant au comportement du composant `VCIOCPBridge`.

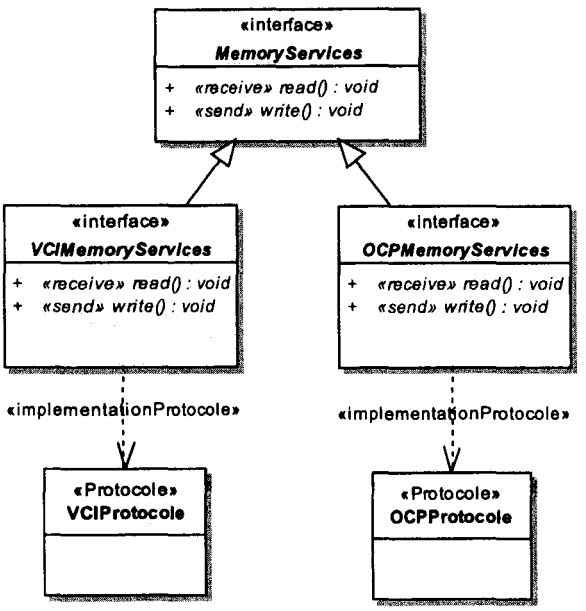


FIG. 5.10 – Raffinement de l'interface MemoryServices pour le support des protocoles OCP et VCI

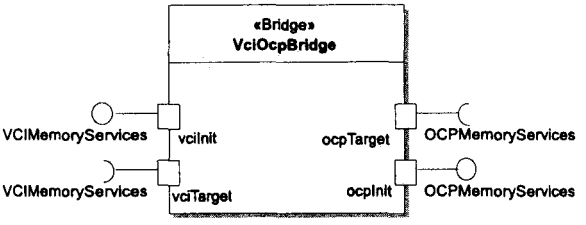


FIG. 5.11 – Définition du composant VCIOCPBridge

2.2 Réseau d'interconnexion Ω N×N

Dans le chapitre introductif de ce document, nous avons évoqué un certain nombre de tendances technologiques pour la conception des systèmes sur puce. En particulier, nous avons montré que l'utilisation des bus comme support d'interconnexion était rendue obsolète par la complexification des systèmes à considérer, impliquant une multiplication du nombre de ressources. Pour les systèmes complexes, les bus sont progressivement remplacés par des réseaux d'interconnexion. L'utilisation de réseaux comme support d'interconnexion pour les architectures parallèles n'est pas une solution originale. Cette solution trouve son origine dans les systèmes dédiés au calcul hautes performances, constitués d'un grand nombre de ressources de calcul, et où la performance des communications joue un rôle considérable dans la performance globale du système. Il est ainsi intéressant de constater que des solutions vielles de plusieurs décennies (les réseaux de type Ω ont par exemple été proposées en 1975 [13]), typiquement appliqués à des systèmes massivement parallèles pour le calcul haute performance, sont aujourd'hui appliqués à l'échelle d'une puce grâce à l'évolution des technologies d'intégration.

Pour la définition de notre cluster de multi-mips, l'utilisation d'un réseau est clairement justifiée par le fait qu'un grand nombre de ressources de calculs va être utilisé. Pour la mise en œuvre du cluster, nous allons utiliser un réseau particulier : le réseau Ω. Avant de décrire plus en détail les caractéristiques de ce réseau, il est important de noter que ce choix n'est pas basée sur une étude concrète, justifiant particulièrement l'utilisation d'un réseau de type Ω plutôt qu'un autre réseau. L'intérêt pour cet exemple est avant tout la modélisation de sa topologie, et d'autres réseaux auraient tout aussi bien pu être utilisés pour la mise en œuvre du cluster (gamma, butterfly,...).

Le réseau Ω est un réseau d'interconnexion multi-étages. D'une manière générale, les réseaux multi-étages constituent un mécanisme modulaire pour construire des switchs complexes à partir de switchs élémentaires, généralement de degré 2 (pour l'interconnexion entre deux initiateurs et deux cibles). Concrètement, chaque étage du réseau est constitué du même nombre de switchs élémentaires. Entre chaque étage, des liens connectent les différents switchs, en suivant généralement une topologie régulière. La topologie des connexions combinée à l'utilisation de plusieurs étages permet à tous les initiateurs de communiquer avec toutes les cibles. Dans le cas du réseau Ω, la topologie d'interconnexion est la même entre chaque étage du réseau. En fig.5.12, nous illustrons la structure d'un réseau oméga 8×8 (8 initiateurs et 8 cibles). La topologie d'interconnexion entre les différents étages est un perfect shuffle [25].

De façon plus générale, un réseau Ω à N entrées requiert $\log_2 N$ étages, chaque étage étant composé de N/2 switches 2×2, avec N étant une puissance de 2 (pour avoir un nombre entier d'étages). Pour le réseau Ω 8×8, on obtient ainsi $\log_2 8 = 3$ étages, et $8/2 = 4$ switches par étage. L'idée directrice pour la modélisation compacte d'un réseau Ω à l'aide du profil Gaspard est de considérer un composant de type répétitif paramétré par le nombre d'entrée N (fig.5.13), et composé d'une répétition de switches 2×2 (`xbar : Crossbar2x2`) sur deux dimensions, avec une multiplicité (référençant le paramètre N et étant définie par une expression) égale à $[\log_2 N; N/2]$. La première dimension représente les différents étages, et la deuxième dimension représente les switches composant les différents étages. Le fait que la multiplicité des ports du composant soit sur deux dimensions s'explique de façon pragmatique par le fait que la description de la topologie s'avère plus facile en considérant un tableau à deux dimensions plutôt qu'un vecteur.

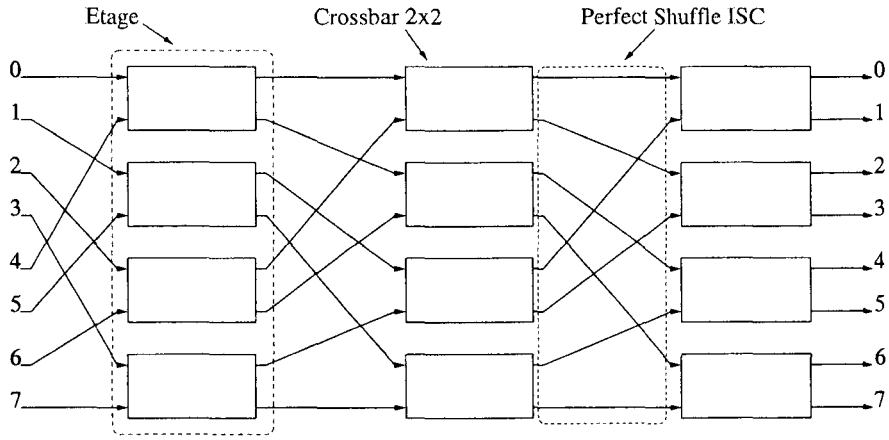


FIG. 5.12 – Réseau Ω 8×8

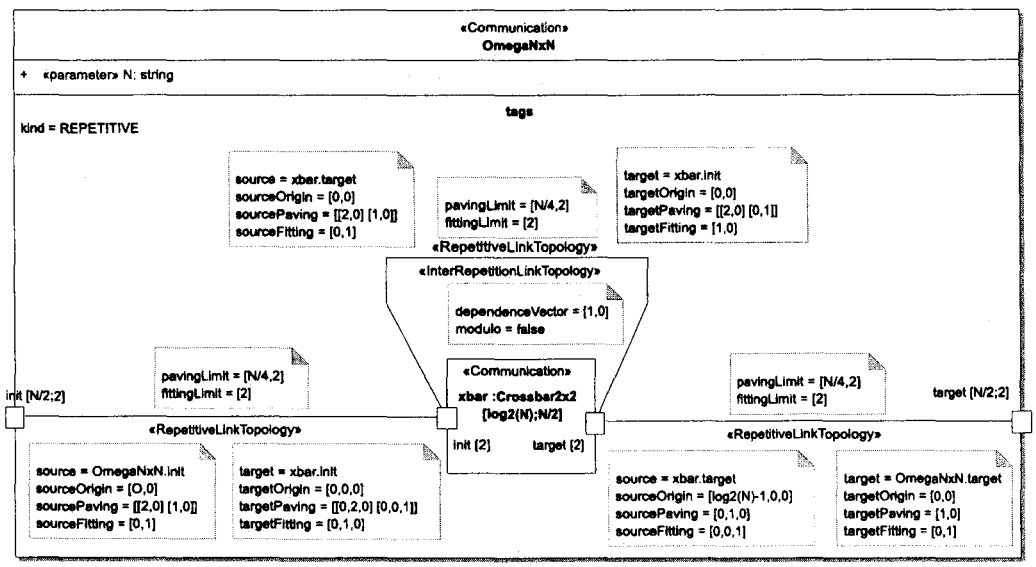


FIG. 5.13 – Modélisation du réseau Ω N×N

Dans un réseau Ω à N entrées, chaque étage d'indice `i` est connecté à l'étage d'indice `i+1` par le biais d'un perfect shuffle. Pour la modélisation de la topologie des liens, trois aspects sont à considérer :

- L'interconnexion entre les différents étages,
- L'interconnexion entre les ports "initiateurs" du réseau et le premier étage,
- L'interconnexion entre le dernier étage du réseau et les ports "cibles".

Le premier aspect est mis en œuvre via l'utilisation des stéréotypes «RepetitiveLinkTopology» et «InterRepetitionLinkTopology» sur le connecteur exprimé entre les ports `target[2]` et `init[2]` de l'élément répété `xbar`. L'utilisation du stéréotype «InterRepetitionLinkTopology» permet de spécifier de façon grossière qu'il existe une dépendance uniforme sur l'espace de répétition de `xbar`, borné par la multiplicité [$\log_2$N ; N/2], sur la première dimension (le tagged value `dependenceVector` est égal à [1,0]). Cette dimension correspond en effet, comme nous l'avons déjà évoqué, aux différents étages du réseau. Concrètement, ces informations spécifient que les crossbars d'un étage d'indice `i` sont connectés aux crossbars de l'étage d'indice `i+1`, d'une façon non encore déterminée. A noter que le tagged value `modulo` est égale à false, ce qui signifie que les ports `target` du dernier étage ne sont connectés à rien. Pour prendre en compte complètement le premier aspect, il nous reste à spécifier la topologie des connexions entre deux tableaux de connector ends de dimension [N/2 ;2], N/2 représentant les différents crossbars, et 2 représentant les ports de chaque crossbar (Comme illustré en fig.5.13, la multiplicité des ports `init` et `target` du composant `Crossbar2x2` est égale à [2]). La description de cette topologie est mise en œuvre via l'utilisation du stéréotype «RepetitiveLinkTopology» sur le même connecteur. Les différentes valeurs de pavage et d'ajustage sont les mêmes que celles utilisées dans fig.4.30, et correspondent à la définition d'un pattern de connection de type perfect shuffle.

Le deuxième aspect est mis en œuvre via l'utilisation du stéréotype «RepetitiveLinkTopology» entre le port `init[N/2 ; 2]` sur le connecteur en bordure du composant `OmegaNxN` et le port `init[2]` de la part `xbar`. Les informations associées au stéréotype correspondent à la définition d'une topologie de type perfect shuffle entre le port d'entrée du composant et le premier étage du réseau.

Le dernier aspect est également mis en œuvre via l'utilisation du stéréotype «RepetitiveLinkTopology» sur le connecteur entre le port `target[2]` de la part `xbar` et le port `target[N/2 ; 2]` en bordure du composant `OmegaNxN`. Les informations associées au stéréotype correspondent à la définition d'une topologie de type perfect shuffle entre le dernier étage du réseau et le port de sortie du composant `OmegaNxN`. On constate ainsi que le tagged value `sourceOrigin` est égal à [$\log_2$N - 1 ;0 ;0], $\log_2$N-1 représentant l'indice du dernier étage du réseau.

2.3 Cluster de multi-mips

La dernière étape dans la conception de notre plate-forme générique consiste maintenant à définir un composant matériel composé d'une répétition de composants `MultiMips` et d'un composant `OmegaNxN` : le composant `MultiMipsCluster`, illustré en fig.5.14. On constate que les différents paramètres du composant `MultiMips` sont remontés au niveau du composant `MultiMipsCluster`, afin de faciliter le dimensionnement de l'architecture. Tous les paramètres à faire varier se retrouvent au niveau du composant de plus haut niveau. Le composant `MultiMipsCluster` introduit par ailleurs le paramètres `numberOfNodes`,

nombre de nœuds dont est constitué le cluster. Dans la section suivante, nous allons voir comment cette architecture générique peut être dimensionnée de façon raisonnable pour supporter l'exécution de l'application Downscaler, en fixant des valeurs d'instance pour les différents paramètres : nombre de nœuds, nombre de processeurs par nœud, taille de la mémoire locale de chaque nœud, et la taille des mémoires cache.

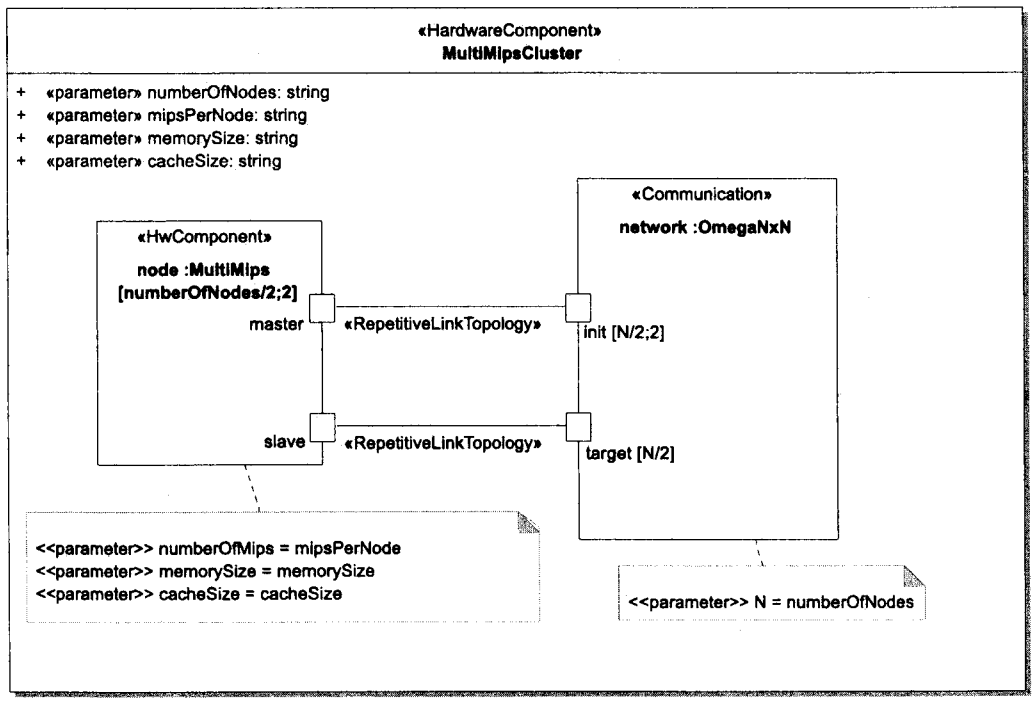


FIG. 5.14 – Modélisation du cluster de MultiMips

3 Association

Dans un flot de conception complet, c'est à dire allant d'une spécification haut niveau à une implantation physique, l'association que nous allons décrire dans cette section pourrait être vue comme une première itération dans le processus de conception du système. Par rapport au degré de précision des informations dont nous disposons ou que nous souhaitons considérer à ce niveau de l'étude, nous allons proposer un premier placement, relativement grossier. Une deuxième étape, que nous ne décrirons pas dans ce document, serait typiquement d'évaluer la pertinence de l'association proposée en utilisant des outils d'analyse ou de simulation, de la modifier, ou encore de la raffiner. Dans un premier temps, nous proposons un dimensionnement de l'architecture `MultiMipsCluster`, en fixant des valeurs d'instance des paramètres du composant, afin d'obtenir un dimensionnement réaliste, bien qu'approximatif, pour le support de l'exécution du `Downscaler`. Dans un second temps, nous proposons une deuxième transformation de la structure de l'application, en vue de faciliter l'expression du placement décrit dans la dernière étape. Il convient de rappeler, une nouvelle fois, que l'objectif n'est pas de proposer une solution nécessairement viable technologiquement, mais d'illustrer l'utilisation des différents mécanismes proposés par le profil Gaspard.

3.1 Dimensionnement de l'architecture et transformation de l'application

L'information de base dont nous disposons pour dimensionner le nombre de ressources de calcul est que le temps de calcul nécessaire pour la mise à l'échelle d'une image sur un mips est d'approximativement 1 seconde. Compte tenu de cette information, on estime ainsi grossièrement qu'une quarantaine de processeurs devrait suffire à obtenir l'accélération nécessaire pour parvenir à obtenir un flot de 30 images mises à l'échelle par seconde. Compte tenu des caractéristiques du réseau Ω (le nombre d'entrée doit être une puissance de 2), on choisit de répartir ces quarante processeurs en 8 nœuds (ce qui nous ramène à un réseau Ω 8×8), à raison de 5 Mips par nœud.

Pour le dimensionnement des mémoires, l'idée directrice est de stocker dans la mémoire locale de chaque nœud un huitième de l'image en entrée, un huitième de l'image en sortie, et répartir un huitième des calculs sur chaque nœud (avec évidemment un alignement des données par rapport aux calculs). Cela va nous amener à introduire une nouvelle transformation dans l'application, en rajoutant un niveau de hiérarchie pour faciliter l'expression du placement. En fig.5.7, nous avons montré que le calcul de la mise à l'échelle d'une image haute définition pouvait être réalisée par une répétition de 240×120 filtres `HFVF`, chaque filtre consommant 13×14 pixels parmi les 1924×1084 de l'image d'entrée, et en produisant 3×4 parmi les 720×480 de l'image de sortie. Pour répartir les calculs (ainsi que les données) en bloc de 8, on applique sur l'application Downscaler une transformation simple, nommée Nesting. Fig.5.15 illustre la définition du composant Downscaler après application du nesting.

On constate qu'un composant nommé `HFVFNesting` (fig.5.16) a été généré par la transformation. La structure du composant `HFVFNesting` décrit le traitement d'un bloc de 244×1084 pixels, soit un huitième d'une image haute définition. Dans la structure du composant Downscaler, il est instancié sous la forme d'une part `hfvf` de multiplicité [8;*]. Les connecteurs stéréotypés «RepetitiveLinkTopology» spécifient une répartition par bloc des différents pixels entre les 8 instances potentiels de `hfvf`. Dans la section suivante, nous ver-

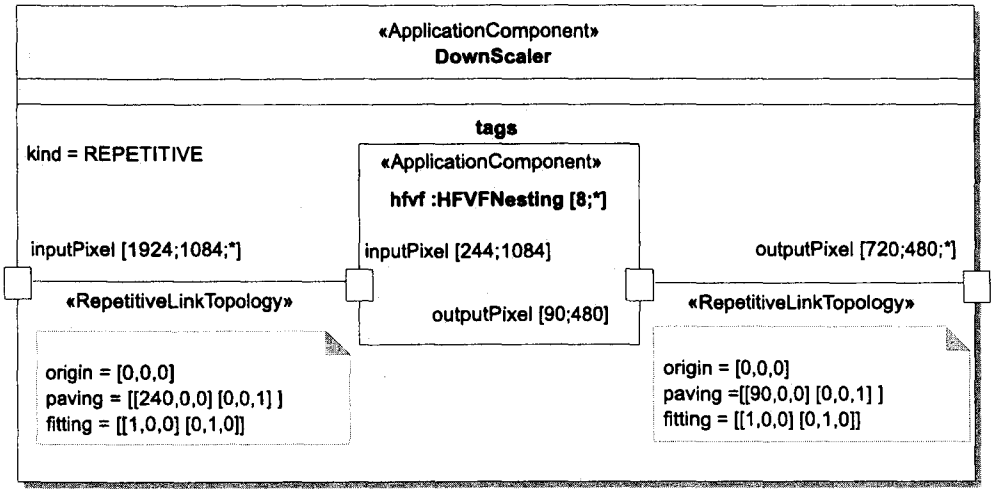


FIG. 5.15 – Structure du composant Downscaler après application du nesting

rons que chacune de ces instances potentielles sera placée sur un nœud, ainsi que les données produites et consommées par chacune de ces instances.

Compte tenu de ces informations, les mémoires peuvent être dimensionnées de la façon suivante :

- Mémoire locale de chaque nœud (part `local : SRam`) : L'idée est de disposer d'une mémoire suffisamment grande pour pouvoir stocker dans la mémoire locale de chaque nœud un huitième de l'image en entrée, et un huitième de l'image en sortie. Cela correspond aux données consommées et produites par un huitième des filtres. La taille minimale de la mémoire (en mots[1]) est donc égale à : 244×1084 (image d'entrée) + 90×480 = 307696 mots (≈ 1 Mo).
- Mémoire cache de chaque Mips à l'intérieur d'un nœud (part `xcache : XCache`) : En fig.5.16, on peut constater qu'un composant `HFVFNesting` est composé d'une part de type `HFVF` de multiplicité [30 ;120]. En considérant que ces 30×120 instances potentielles soient réparties par blocs sur les 5 Mips composant un nœud (par exemple des blocs de 30×24), il est nécessaire, si l'on souhaite que toutes les données utilisées pour le calcul d'un bloc soient stockées dans un cache, que la taille du cache soit égale à 244×220 = 53680 mots, soit ≈ 210 Ko.

Pour résumer, nous avons donc défini comme valeur d'instance des paramètres pour le composant `MultiMipsCluster` :

- numberOfNodes = 8
- mipsPerNode = 5
- memorySize = 307696
- cacheSize = 53680

[1] 1 pixel est codé sur 32 bits, et la taille des mots mémoires est également de 32 bits

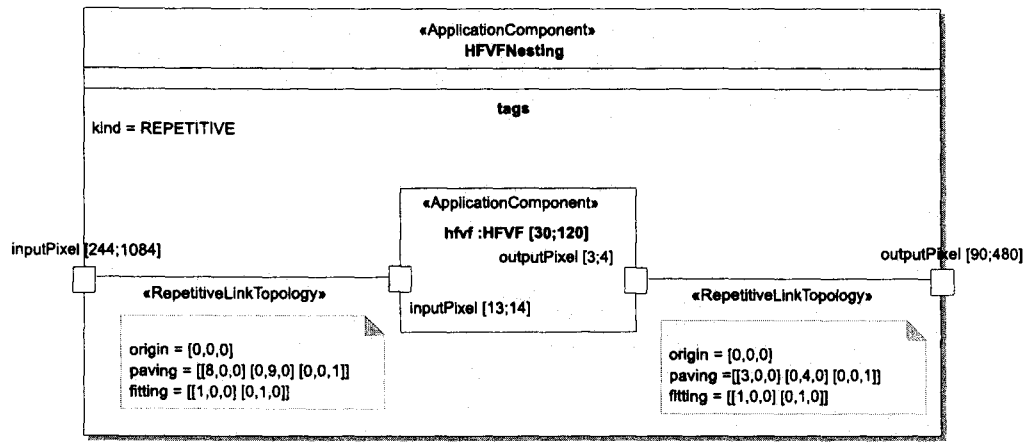


FIG. 5.16 – Composant HFVFNesting généré par le nesting

Fig.5.17 illustre la définition de ces paramètres pour le composant `MultiMipsCluster`, sous la forme d'une note associée.

3.2 Modélisation du placement

Comme nous venons de l'expliquer, l'architecture matérielle est basiquement constituée de deux niveaux de hiérarchie. Le premier niveau est constitué de 8 nœuds, interconnectés via un réseau Ω. Le deuxième niveau concerne la structure de chaque nœud, constitué de 5 Mips, disposant chacun d'un cache banalisé, et d'un accès à une mémoire locale. Par ailleurs, nous avons appliqué une transformation à l'application (nesting) afin de rajouter un niveau de hiérarchie. Cette transformation a été orientée afin d'adapter la "forme" de l'application à celle de l'architecture. Ainsi, le niveau de hiérarchie crée dans l'application est constitué une répétition de $8\times*$ HFVFNesting (composant généré par la transformation), chaque instance potentielle consommant $1/8$ d'une image en entrée pour produire un $1/8$ d'une image en sortie. Chacune de ces instances potentielles est constituée d'une répétition de 30×120 répétitions du filtre HFVF (filtre horizontal suivi d'un filtre vertical). L'idée de base pour la définition de l'allocation est de considérer une première répartition des $8\times*$ répétitions de HFVFNesting sur les 8 nœuds. A chaque répétition de `node` est alloué un motif de tâche de dimension [*]

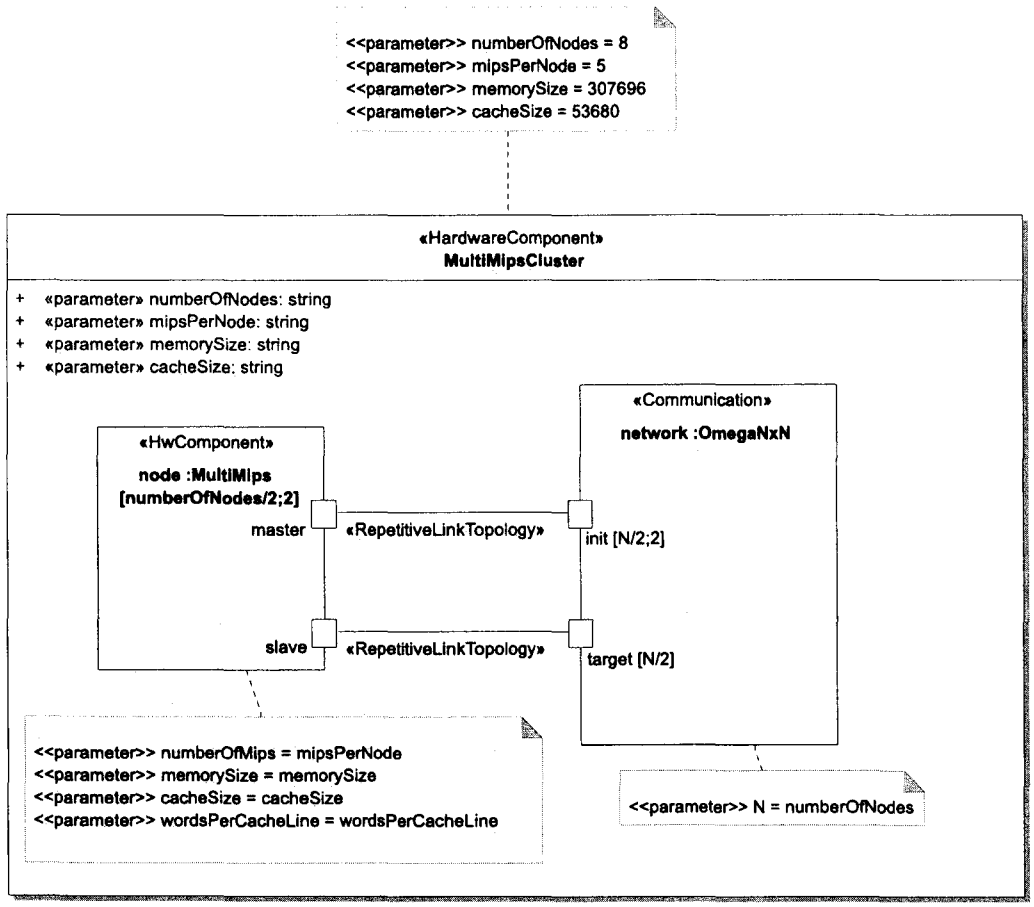


FIG. 5.17 – Paramétrage du composant MultiMipsCluster

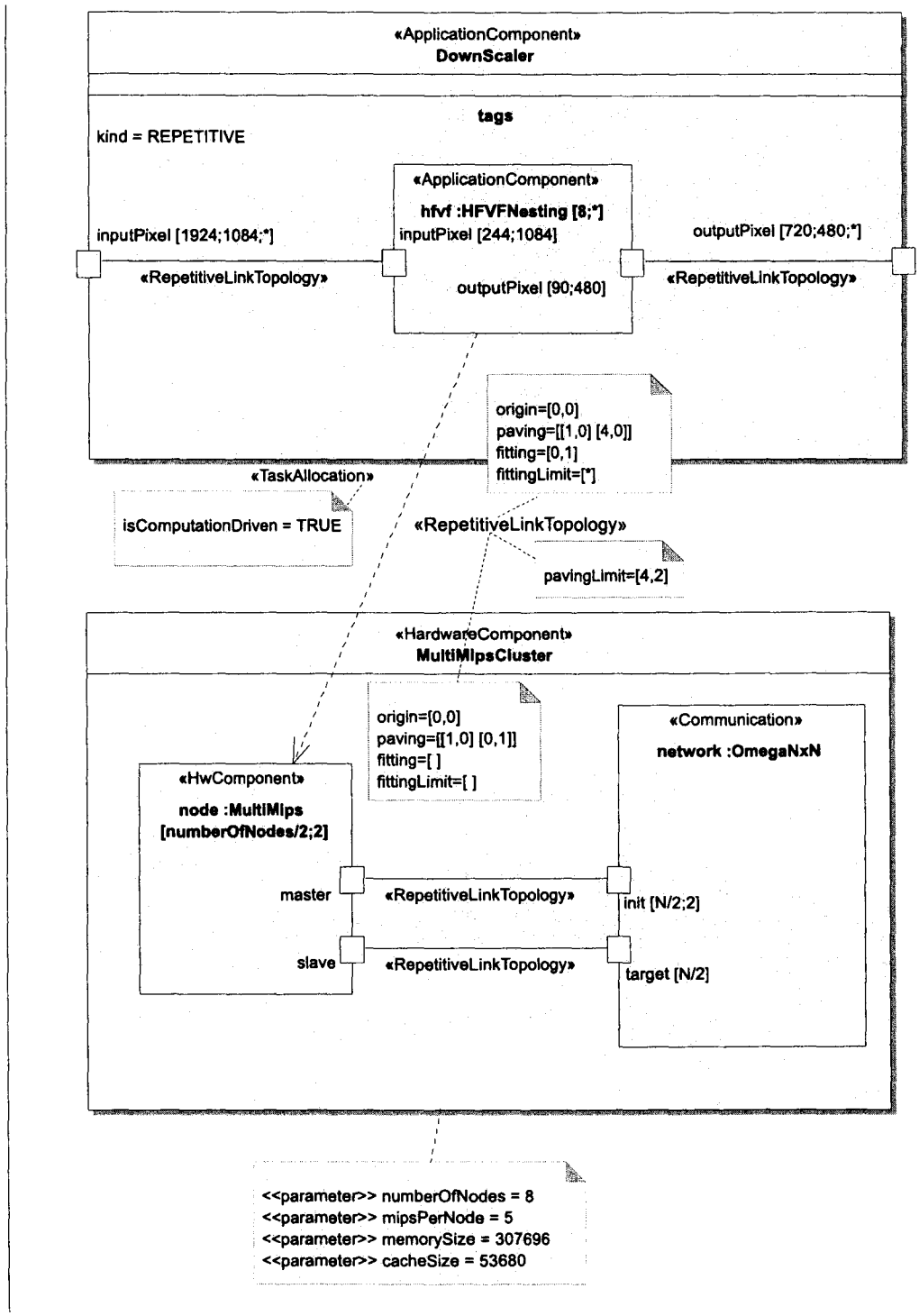


FIG. 5.18 – Allocation par blocs (colonnes) des instances potentielles de HFVFNesting sur les nœuds du cluster

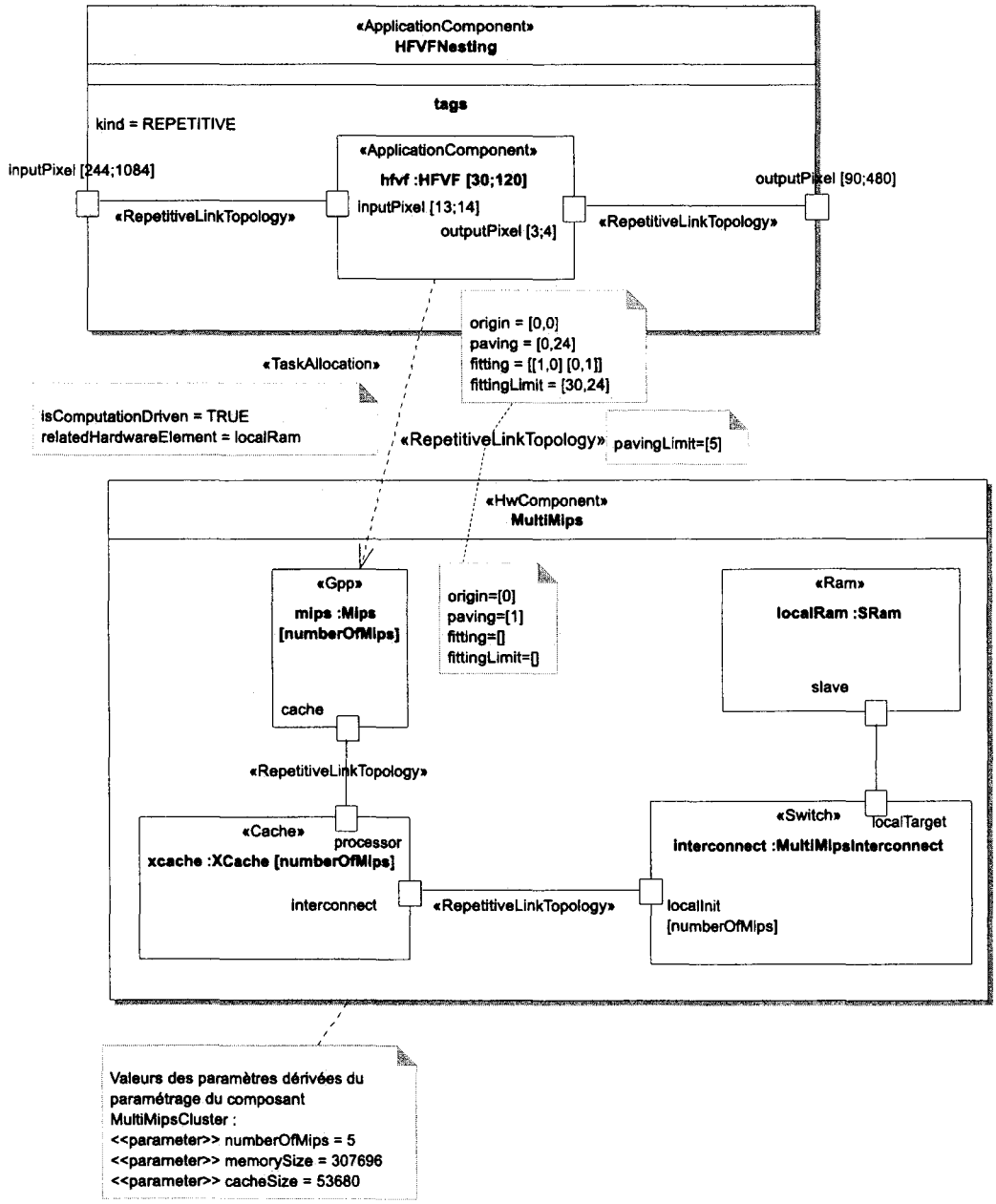


FIG. 5.19 – Allocation par blocs (lignes) des instances potentielles de HFVF sur les mips d'un noeud

CHAPITRE 6

Conclusion et perspectives

1 Bilan 147

2 Perspectives 148

2.1 Alignement avec MARTE 148

2.2 Introduction du contrôle dans les modèles 149

2.3 Extension du modèle d'association 150

2.4 Prise en compte de l'environnement 150

2.5 Extension des mécanismes de factorisation 151

1 Bilan

L'objectif de cette thèse était de fournir un support pour la modélisation à haut niveau d'abstraction (fonctionnel) des systèmes sur puce à hautes performances, avec une contribution à deux niveaux :

- la définition d'une syntaxe abstraite par des métamodèles exprimés en MOF (servant d'infrastructure pour la mise en œuvre du flot de conception),
- la définition d'une syntaxe concrète par un profil pour UML 2 (que l'on nommera par la suite le profil Gaspard).

Le profil Gaspard (et les métamodèles associés), que nous avons décrit dans le chapitre 4, apporte une solution pour les points suivants :

- la définition d'une méthodologie orientée composant,
- la définition de mécanismes communs de factorisation,
- une séparation claire entre application et architecture matérielle,
- la définition d'un modèle de description d'architectures matérielles,
- la définition d'un modèle d'association,
- la définition de mécanismes de caractérisation.

Le profil intègre des solutions pour la modélisation de trois aspects d'un système : la modélisation des applications, des architectures matérielles, et de l'association entre application et architecture. Outre une méthodologie orientée composant offrant un support pour la réutilisation d'IP logicielles ou matérielles, nous avons proposé des mécanismes unifiés permettant de décrire les aspects répétitifs et réguliers dans la conception de tels systèmes [11]. Ces mécanismes englobent deux aspects : considérer des multiplicités multi-dimensionnelles pour la répétition d'éléments structurels, et ajouter des informations sur les différents types de relations (connecteurs et dépendances) pouvant être modélisés entre éléments structurels, afin de décrire leur topologie. Comme nous l'avons illustré dans les deux précédents chapitres de ce document, ces mécanismes permettent d'exprimer de façon compacte :

- dans la modélisation des applications, tout le parallélisme de données potentiel, de façon non biaisée par un ordre d'exécution artificiel,
- dans la modélisation des architectures matérielles, tout le parallélisme disponible, ainsi que la topologie des connexions entre les éléments architecturaux,
- dans la modélisation d'une association entre application et architecture, le placement régulier des parties régulières de l'application sur les parties régulières de l'architecture.

Cette version du profil n'a pour l'instant pas fait l'objet d'une intégration dans la chaîne de transformations de Gaspard, bien qu'une implantation du profil pour l'outil Enterprise Architect [59] aient été définies. La version actuelle de l'environnement Gaspard [34] [3] (définie sous la forme d'un plug-in Eclipse [18]) est basée sur des versions précédentes du profil ([17] pour la partie application, et [12] [10] pour l'architecture). La version précédente de la partie du profil concernant la modélisation d'architectures matérielles a notamment été définie dans le cadre du projet ITEA P2I [9], pouvant être considérée comme une étude préliminaire ayant conduit à la définition de la RFP MARTE (que nous présentons dans la prochaine section). A court terme, la version actuelle du profil doit faire l'objet d'une intégration dans la chaîne de transformations de Gaspard.

2 Perspectives

Les perspectives d'évolution pour les travaux présentés dans cette thèse sont multiples, et concernent principalement :

- Un alignement avec la RFP MARTE,
- L'introduction de l'expression du contrôle dans les modèles : les applications considérées pour l'instant sont purement data parallèles,
- Une extension du modèle d'association : pour une prise en compte du placement temporel,
- La prise en compte de l'environnement : les systèmes modélisés avec la version actuelle du profil ne prennent absolument pas en compte l'environnement,
- L'extension des mécanismes de factorisation : pour pouvoir prendre en compte d'autres topologies de connexion.

2.1 Alignement avec MARTE

La stratégie d'alignement vis à vis des standards OMG pour la définition du profil Gaspard, que nous avons évoqué dans le chapitre 3, n'est pas anodine. Il s'agit en effet globalement de la stratégie qui va être utilisée pour la définition de la réponse à la Request For Proposal MARTE (Modeling and Analysis of Real Time and Embedded systems) [52] [55], nommée proMARTE. L'équipe DaRT a activement participé à la définition de la RFP, et continue à participer activement à la rédaction de la réponse proMARTE, dans le cadre du consortium PROTES [61]. La RFP MARTE a récemment été votée par l'OMG (février 2005). Cette RFP sollicite des soumissions pour la définition d'un profil UML permettant la modélisation des systèmes temps réel embarqués, et l'analyse des performances et de l'ordonnançabilité de spécifications UML. La motivation initiale de la RFP était de proposer une mise à jour du profil SPT, rendu obsolète par l'apparition de nouveaux standards, tels que UML 2 et le profil QoS. SPT ayant par ailleurs fait l'objet d'un certain nombre de suggestions pour l'amélioration et la consolidation du profil, la communauté a considéré qu'une révision mineure du profil n'était pas envisageable. Cet état de fait a encouragé la définition de la RFP MARTE.

Dans le cadre de la réponse proMARTE, DaRT est responsable de la rédaction de deux requirements :

- 19 : concernant la définition d'un catalogue de qualités de service pour la prise en compte des contraintes d'embarquabilité. Typiquement, ces contraintes concernent la consommation d'énergie ou la prise en compte de l'espace mémoire requis pour l'exécution d'une application ou d'une tâche. Les qualités de services définies dans la section association du chapitre décrivant le profil Gaspard constituent des points de départ pour la définition de ce catalogue.
- 23 : concernant la définition de mécanismes communs pour la factorisation de la modélisation des structures répétitives, à la fois pour l'application, l'architecture, et l'allocation. Les mécanismes de factorisation que nous avons proposé dans ce document constituent à priori une réponse satisfaisante à ce requirement. Néanmoins, ces mécanismes sont fortement orientés vers l'utilisation de composants. Un travail de généralisation est peut être à envisager.

Par ailleurs, DaRT participera (au moins) à la rédaction des requirements suivants :

- 3 : consistant à garantir que la réponse proposée sera correctement alignée avec les mécanismes de définition de profil proposés dans UML 2.
- 17-24 : concernant la définition d'un modèle d'allocation d'une application sur une architecture matérielle. Les réponses proposées dans la section Association (chapitre décrivant le profil Gaspard) vont dans cette direction.
- 25 : consistant à garantir que la proposition supporte une modélisation orientée composant des systèmes. Là aussi, il s'agissait, comme nous l'avons évoqué, d'un des objectifs forts ayant orienté la définition du profil Gaspard. DaRT est donc en mesure de proposer des idées allant dans cette direction.

2.2 Introduction du contrôle dans les modèles

Dans la version actuelle du profil Gaspard, seules des applications décrivant un traitement systématique peuvent être modélisées. Des travaux actuellement en cours ([33] décrit les premiers résultats obtenus) visent à introduire la possibilité d'exprimer du contrôle dans les applications, en s'inspirant du formalisme des automates de mode [37], afin d'étendre le domaine des applications adressé par Gaspard. A l'origine, ce formalisme a été proposé pour la modélisation des différents modes de fonctionnement d'un système, chaque mode de fonctionnement étant décrit par un graphe flot de données. Chaque mode propose typiquement un comportement régulier et systématique, et pourrait être décrit avec les mécanismes proposés actuellement dans le profil. Les extensions du profil reviendraient à rajouter une couche de contrôle pour la gestion des changements de mode, avec une séparation claire entre la spécification de la partie contrôle et de la partie calcul. Ce type de description propose de nombreux avantages, d'un point de vue modélisation comme d'un point de vue vérification, principalement dus à la séparation entre les parties contrôle et calcul :

1. D'un point de vue modélisation :
   - Garantir la lisibilité des modèles (le contrôle n'est pas "noyé" dans les calculs),
   - Faciliter la maintenance ou la modification des modèles (les différents aspects du modèle sont clairement identifiés)
2. D'un point de vue vérification :
   - Gain en temps et en espace mémoire (les parties contrôle et calcul pouvant être vérifiées séparément),
   - Localisation des erreurs.

L'objectif à moyen terme est de pouvoir transformer les modèles produits via l'utilisation du profil Gaspard (étendu avec des mécanismes d'expression du contrôle sous forme d'automates de modes) vers des langages synchrones tels que Lustre ou Signal, eux-mêmes orientés vers le dataflow, afin de pouvoir réutiliser les outils de vérification développés autour de ces outils.

A plus long terme, nous envisageons d'étudier la possibilité d'étendre ces mécanismes d'expression du contrôle à la modélisation de l'association et à la modélisation des architectures matérielles, le tout étant évidemment lié aux modes de fonctionnement de l'application. Le premier cas reviendrait à considérer l'expression d'une association comme une entité à part entière (ce qui n'est pas le cas actuellement), et à modifier dynamiquement le placement en fonction du mode de fonctionnement de l'application, un "mode" d'association étant rattaché à un ou plusieurs modes de fonctionnement. Ce type de mécanisme permettrait par exemple de mettre en oeuvre des directives de type `REDISTRIBUTE` (proposée

par le langage HPF), pour une redistribution dynamique des données. Du côté de l'architecture matérielle, ce mécanisme serait typiquement utile pour permettre la reconfiguration dynamique d'un système composé d'un ou plusieurs FPGA.

2.3 Extension du modèle d'association

Les mécanismes de factorisation que nous avons proposé permettent, lorsqu'ils sont appliqués aux dépendances d'allocation des données ou des tâches, d'exprimer de façon compacte une distribution spatiale des parties régulières d'une application sur les parties régulières d'une architecture matérielle. Une première extension viserait à rajouter une dimension temporelle dans l'expression du placement, afin de décrire un ordonnancement. Concrètement, pour une architecture composée par exemple d'une répétition de processeurs sur `m` dimensions, la description de la topologie des dépendances d'allocation entre les tâches d'une application et les processeurs de l'architecture serait réalisée non pas en considérant `m` dimensions du côté de l'architecture, mais `m+1` dimensions : La dimension rajoutée correspondrait au temps. L'avantage de cette solution est qu'elle ne nécessite pas de remise en cause fondamentale des différents modèles que nous avons proposé. La dimension temporelle est une dimension d'un tableau comme les autres, et peut donc être adressée par les mécanismes de factorisation. Cependant, associer une instance de tâche d'indice `i` à l'indice `i` de la dimension temporelle de l'architecture, et associer une instance d'une tâche d'indice `i+1` à l'indice `i+1` de la dimension temporelle de l'architecture, revient à spécifier que la tâche d'indice `i` doit être exécutée avant la tâche d'indice `i+1`. Si le mécanisme semble relativement simple, un certain nombre de points restent à étudier. En particulier, dans une architecture composée de plusieurs niveaux de hiérarchies, et de plusieurs répétitions d'éléments architecturaux (processeurs et mémoires), la principale difficulté consiste à déterminer quel est le référentiel de temps commun entre les différents éléments.

Un autre aspect non traité dans la version actuelle du profil concerne la façon de placer plus finement les données dans les mémoires, et en particulier la façon de linéariser en mémoire les tableaux traités dans les applications. Là encore, les mécanismes de factorisation pourraient être utilisés. L'idée serait de considérer une mémoire comme une répétition d'éléments de stockage élémentaires, sur une seule dimension. Linéariser les tableaux reviendrait à rajouter des informations topologiques sur une dépendance d'allocation de données exprimée entre le port d'un composant applicatif et une mémoire décomposée en éléments de stockages, la topologie de la connexion décrivant la linéarisation du tableau. Combiné aux mécanismes de prise en compte du temps dans le placement, cela permettrait de définir comment l'espace mémoire est réutilisé au cours du temps.

2.4 Prise en compte de l'environnement

Dans la classification des ressources matérielles que nous avons proposé, nous avons identifié le concept d'`IO` (Input/Ouput, ou composant d'entrée/sortie). Typiquement, ce type de composant serait utilisé pour modéliser des capteurs, ou des actionneurs, ou plus généralement l'environnement de l'architecture matérielle. Néanmoins, dans les différentes études que nous avons mené, nous n'avons pas réellement pris en compte ces aspects. Pour un composant réalisant l'acquisition des données, il conviendrait de faire un lien entre les données consommées dans l'application, et la façon dont les données sont produites par le capteur (fréquence d'échantillonage, ordre dans lequel sont produites les différentes

données composant un tableau). La prise en compte de ces informations devrait par ailleurs avoir un impact non négligeable sur l'expression du placement temporel d'une application sur une architecture.

2.5 Extension des mécanismes de factorisation

Les mécanismes que nous avons défini permettent d'exprimer, comme nous l'avons illustré à plusieurs reprises, des topologies relativement complexes, telles qu'un perfect shuffle entre deux étages d'un réseau Ω. Cependant, ils ne permettent pas de modéliser des topologies arborescentes, par exemple utilisées pour la mise en œuvre de réseaux d'interconnexion de type Fat Tree [26]. Côté applicatif, ce type de topologie est particulièrement adapté pour la mise en œuvre d'algorithmes de réduction. Proposer une solution permettant la modélisation de ce type de topologies fait donc partie des perspectives d'évolution du profil à court terme.

Table des figures

1.1 Portes disponibles vs Portes utilisées 11

2.1 Flot traditionnel de conception de SoC 18
2.2 Structuration de la conception sous forme de niveaux d'abstraction 19
2.3 Exemple de NoC : Topologie en grille 20
2.4 Métamodèles pour le flot de conception Gaspard 22
2.5 Flot en Y dans le MDA 23
2.6 Dépendances tableau dans un modèle global 31
2.7 Dépendances motif dans un modèle local 31
2.8 Exemples d'ajustage 32
2.9 Exemples de pavages 33
2.10 Parallélisme potentiel du produit de matrices 35
2.11 Expression du parallélisme de données 35

3.1 Structure du profil SPT 45
3.2 Modèle client/serveur et qualité de service dans le GRM 46
3.3 Framework de définition de qualités de service (diagramme simplifié) 48
3.4 Définitions de caractéristiques QoS 49
3.5 Définitions d'un "modèle de qualité" spécifique 50
3.6 Architecture de SysML 51
3.7 SysML vs UML 2 : Réutilisation des packages 53
3.8 Mise en oeuvre du concept d'Allocation 54

4.1 Architecture du profil Gaspard 60
4.2 Component : Vue d'ensemble 61
4.3 Component : Mécanismes d'encapsulation 63
4.4 Connector : Définition 64
4.5 Parameter : Définition 65
4.6 InstanceValue : Définition 65
4.7 Structured classifier 66
4.8 Encapsulated classifier 66
4.9 Class étendue avec support des structures internes 67
4.10 Composant UML 2 68
4.11 Stéréotype «GaspardComponent» 69
4.12 Vue domaine du "modèle" de communications dans UML 70
4.13 Vue domaine des différents types de requêtes dans UML 70
4.14 Définition du concept de `Signal` dans UML 71

4.15 Définition du concept d'`Interface` dans UML 72
4.16 `Interface` étendue dans le package `Communications` 73
4.17 Stéréotype «Parameter» 74
4.18 Exemple d'utilisation des paramètres 75
4.19 Multiplicity : Définition 77
4.20 Topologie de type "cube" spécifiée via une multiplicité multi-dimensionnelle 78
4.21 LinkTopology : Définition 78
4.22 Modélisation d'un "perfect shuffle" 80
4.23 Exemple de structure composite et d'instanciation 82
4.24 Définition des multiplicités (Infrastructure UML 2) 82
4.25 Pattern de connection en "étoile" 83
4.26 Exemple de structure composite ambiguë 83
4.27 Définition des multiplicités multi-dimensionnelles 84
4.28 Définition du stéréotype «**LinkTopology**» 85
4.29 «InterRepetitionLinkTopology» : Modélisation d'une topologie en grille . . . 86
4.30 «RepetitiveLinkTopology» : Modélisation d'un "perfect shuffle" 86
4.31 Utilisation simplifiée de «RepetitiveLinkTopology» dans un composant de type `REPETITIVE` 87
4.32 HardwareComponent : Classification des ressources 89
4.33 Raffinement des ressources de type Processor 92
4.34 Raffinement des ressources de type Memory 94
4.35 Politiques d'écriture des caches 95
4.36 Politiques de remplacement des caches 95
4.37 Raffinement des ressources de type Communication 96
4.38 Politiques d'arbitrage 96
4.39 Interfaces pour les composants matériels 98
4.40 Définition du stéréotype «HardwareComponent» 99
4.41 Exemple de définition d'implantation pour un composant matériel 100
4.42 Exemple de définition de jeu d'instructions pour un composant de type Gpp 100
4.43 Exemple de définition d'interface pour un composant matériel 101
4.44 Architecture quadri mips basée sur la librairie SocLib 102
4.45 Spécification d'un réseau générique de type grille torique 103
4.46 Spécification du tableau de processeurs 104
4.47 Définition du concept de ApplicationComponent 106
4.48 Stéréotype «ApplicationComponent» 107
4.49 Spécification des jeux d'instructions disponibles pour un composant applicatif 108
4.50 Exemple d'utilisation de dimension infini pour les ports d'un composant applicatif 109
4.51 Modélisation du produit de matrices 110
4.52 Modélisation de la réduction d'un vecteur 111
4.53 Exemple de catégories de QoS pour Gaspard 113
4.54 Exemple de caractérisation 114
4.55 Définition des concepts de `DataAllocation` et `TaskAllocation` 115
4.56 Définition des stéréotypes de `DataAllocation` et `TaskAllocation` 117
4.57 Allocation des colonnes par blocs 118
4.58 Allocation des colonnes cyclique 120
4.59 Allocation des colonnes en cyclique k (k=4) 121

5.1 Principe du DownScaler 126
5.2 Filtre horizontal 127
5.3 Composant élémentaire pour le filtre horizontal 128
5.4 Filtre vertical 128
5.5 Composant élémentaire pour le filtre vertical 129
5.6 Structure de l'application Downscaler 130
5.7 Structure de l'application Downscaler après fusion 131
5.8 Structure du composant HFVF généré par la fusion 131
5.9 Définition du composant MultiMips 133
5.10 Raffinement de l'interface MemoryServices pour le support des protocoles OCP et VCI 134
5.11 Définition du composant VCIOCPBridge 134
5.12 Réseau Ω 8×8 136
5.13 Modélisation du réseau Ω N×N 136
5.14 Modélisation du cluster de MultiMips 138
5.15 Structure du composant Downscaler après application du nesting 140
5.16 Composant HFVFNesting généré par le nesting 141
5.17 Paramétrage du composant MultiMipsCluster 142
5.18 Allocation par blocs (colonnes) des instances potentielles de HFVFNesting sur les nœuds du cluster 143
5.19 Allocation par blocs (lignes) des instances potentielles de HFVF sur les mips d'un noeud 144

Bibliographie

[1] Acotris (analyse et conception à objets temps réel pour implantation asynchrone/synchrone). `http://www.acotris.c-s.fr/`.

[2] L. Benini and GD Micheli. Networks on chips : A new SoC paradigm. *IEEE Computer*, 35(1) :70–80, January 2002.

[3] Lossan Bondé, Pierre Boulet, Arnaud Cuccuru, Jean-Luc Dekeyser, Cédric Dumoulin, Philippe Marquet, Samy Meftaly, and Mickael Samyn. *Model Driven Engineering for Distributed Embedded Real-Time Systems*, chapter Model Driven Architecture for Intensive Embedded Systems. Hermes Science, 2005.

[4] Luc Bougé. The data parallel programming model : A semantic perspective. In *The Data Parallel Programming Model*. Springer, 1996.

[5] Thomas bräunl. Parallaxis-iii : A language for structured data-parallel programming. Technical report, Computer Science Department, Univ. Stuttgart, 1995.

[6] Peter Conradi. *Reuse in Electronic Design*. Wiley, 1999.

[7] Thinking Machine Corporation. In *CM-2 Technical summary*, 1991.

[8] Thinking Machine Corporation. In *CM-5 Technical summary*, 1991.

[9] A. Cuccuru, R. De Simone, T. Saunier, G. Siegel, and Y. Sorel. P2I : An innovative MDA methodology for embedded real-time systems. In *DSD'2005, 8th Euromicro conference on digital system design*, Porto, Portugal, September 2005.

[10] Arnaud Cuccuru, Pierre Boulet, and Jean-Luc Dekeyser. Regular hardware architecture modeling with UML2. In *FDL'04*, Lille, France, September 2004.

[11] Arnaud Cuccuru, Jean-Luc Dekeyser, Philippe Marquet, and Pierre Boulet. Towards UML 2 extensions for compact modeling of regular complex topologies. In *MODELS/UML 2005, ACM/IEEE 8th international conference on model driven engineering languages and systems*, Montego Bay, Jamaica, October 2005.

[12] Arnaud Cuccuru, Philippe Marquet, and Jean-Luc Dekeyser. Uml 2 as an adl : Hierarchichal hardware modeling. In *WADL'04*, Toulouse, France, August 2004.

[13] Lawrie D.A. Access and alignment of data in an array processor. *IEEE Trans. Comput.*, C-24(12) :1145–1155, December 1975.

[14] Alain Demeure, Anne Lafage, Emmanuel Boutillon, Didier Rozzonelli, Jean-Claude Dufourd, and Jean-Louis Marro. Array-OL : Proposition d'un formalisme tableau pour le traitement de signal multi-dimensionnel. In *Gretsi*, Juan-Les-Pins, France, September 1995.

[15] Bruce Powell Douglass. *Real Time UML : Advances in the UML for Real-Time Systems, 3rd Edition*, chapter UML Profile for Schedulability, Performance, and Time. Addison Wesley Professional, February 2004.

[16] Philippe Dumont and Pierre Boulet. Transformations de code Array-OL : implémentation de la fusion de deux tâches. Technical report, Laboratoire d'Informatique fondamentale de Lille et Thales Communications, October 2003.

[17] Cédric Dumoulin, Pierre Boulet, Jean-Luc Dekeyser, and Philippe Marquet. UML 2.0 structure diagram for intensive signal processing application specification. Research Report RR-4766, INRIA, March 2003.

[18] eclipse.org. Eclipse. `http://www.eclipse.org`, 2005.

[19] Peter H. Feiler, Bruce Lewis, and Steve Vestal. The SAE avionics architecture description language (AADL) standard : A basis for model-based architecture-driven embedded systems engineering. In *RTAS 2003 Workshop on Model-Driven Embedded Systems*, May 2003.

[20] M.J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. on Computers*, 21(9) :948–960, 1972.

[21] D. D. Gajski and R. Kuhn. Guest editor introduction : New VLSI-tools. *IEEE Computer*, 16(12) :11–14, December 1983.

[22] Gerstlauer, Dömer, Peng, and Gajski. *System Design : A Practical Guide with SpecC*. Kluwer Academic Publishers, June 2001.

[23] Peter Green and Martyn Edwards. The modeling of embedded systems using hasoc. In *Design, Automation and Test in Europe Conference and Exhibition (DATE'02)*, Paris, France, March 2002.

[24] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21 :666–677, 1978.

[25] Kai Hwang and Zhiwei Xu. *Scalable Parallel Computing*, chapter Buses, Crossbar, and Multistage Switches, pages 301–302. WCB, 1998.

[26] Kai Hwang and Zhiwei Xu. *Scalable Parallel Computing*, chapter Network Topologies and Properties, page 290. WCB, 1998.

[27] IEEE, editor. *Std 1076-1993 IEEE Standard VHDL Language, Reference Manual - Description*.

[28] INRIA. DaRT home page. `http://www.inria.fr/recherche/equipes_ur/dart.fr.html`, 2005.

[29] American National Standards Institute. American national standard programming language, fortran - extended : Ansi x3.198-1992 : Iso/iec 1539 : 1991 (e). Technical report, 1992.

[30] ITRS. Design, 2003 edition. `http://public.itrs.net/`, 2003.

[31] Axel Jantsch and Hannu Tenhunen. *Networks on chip*. Kluwer Academic Publishers, 2003.

[32] Axel Jantsh and Hannu Tenhunen. *Networks on chip*, chapter Will networks on chip close the productivity gap. Kluwer Academic Publishers, 2003.

[33] Ouassila Labbani, Jean-Luc Dekeyser, and Pierre Boulet. Mode-automata based methodology for scade. In *Hybrid Systems : Computation and control, 8th international workshop*, LNCS series, pages 386–401, Zurich, Switzerland, March 2005. Springer.

[34] Laboratoire d'informatique fondamentale de Lille. Gaspard home page. `http://www.lifl.fr/west/gaspard/gaspardMaven`, 2005.

[35] P. Lanusse, S.Gérard, and F.Terrier. Real-time modeling with UML : The ACCORD approach. In *UML 98 : Beyond the notation*, Mulhouse, France, 1998.

[36] Christian Lefebvre. *HPF-Builder : Un environnement visuel de placement et distribution dédié à HPF*. PhD thesis, Université des Sciences et Technologies de Lille, 1998.

[37] F. Maraninchi and Y. Rémond. Mode-automata : About modes and states for reactive systems. In *European Symposium On Programming*, Lisbon, Portugal, March 1998. Springer verlag.

[38] Philippe Marquet and Jean-Luc Dekeyser. Supporting irregular and dynamic computations in data parallel languages. In Springer, editor, *The Data Parallel Programming Model*, 1996.

[39] N. Medvidovic. Domains of concern in software architectures and architecture description languages. In *USENIX Conference on Domain-Specific Languages*, Santa Barbara, CA, october 1997.

[40] Meincke, Hemani, Kumar, Ellervee, Öberg, Olsson, Nilsson, Lindqvist, and Tenhumen. Globally asynchronous locally synchronous architecture for large high performance ASICs. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 2, pages 512–515, Orlando, USA, May 1999.

[41] G. Nicolescu, K. Svarstad, W. Cesario, L. Gauthier, D. Lyonnard, S. Yoo, P. Coste, and A.A. Jerraya. Desiderata pour la spécification et la conception des systèmes électroniques. *Techniques et Sciences de l'Informatique*, march 2002.

[42] Object Management Group, Inc. Unified modeling language (UML) 1.3. `http://www.omg.org/cgi-bin/doc?formal/00-03-01`, March 2000.

[43] Object Management Group, Inc. The model driven architecture, September 2003. `http://www.omg.org/mda/`.

[44] Object Management Group, Inc., editor. *UML 2 Diagram Interchange (final adopted specification)*. `http://www.omg.org/cgi-bin/doc?ptc/2003-09-01`, September 2003.

[45] Object Management Group, Inc., editor. *UML 2 Infrastructure (Final Adopted Specifcation)*. `http://www.omg.org/cgi-bin/doc?ptc/2003-09-15`, September 2003.

[46] Object Management Group, Inc., editor. *UML 2 OCL (Final Adopted specification)*. `http://www.omg.org/cgi-bin/doc?ptc/2003-10-14`, October 2003.

[47] Object Management Group, Inc., editor. *UML for Systems Engineering RFP*. `http://syseng.omg.org/UML_for_SE_RFP.htm`, March 2003.

[48] Object Management Group, Inc., editor. *UML 2 Superstructure (Available Specification)*. `http://www.omg.org/cgi-bin/doc?ptc/2004-10-02`, October 2004.

[49] Object Management Group, Inc., editor. *UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms*. `http://www.omg.org/cgi-bin/doc?ptc/2004-09-01`, September 2004.

PPN: 115058257

[50] Object Management Group, Inc., editor. *SysML v0.9*. http://www.omg.org/cgi-bin/doc?ad/05-01-03, January 2005.

[51] Object Management Group, Inc., editor. *UML Extension Profile for SoC RFC*. http://www.omg.org/cgi-bin/doc?realtime/2005-03-01, March 2005.

[52] Object Management Group, Inc., editor. *UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) RFP*. http://www.omg.org/cgi-bin/doc?realtime/2005-02-06, February 2005.

[53] Object Management Group, Inc., editor. *(UML) Profile for Schedulability, Performance, and Time Version 1.1*. http://www.omg.org/technology/documents/formal/schedulability.htm, January 2005.

[54] Open SystemC Initiative. SystemC. http://www.systemc.org/, 2005.

[55] L. Rioux, T. Saunier, S. Gerard, A. Radermacher, R. de Simone, T. Gautier, Y. Sorel, J. Forget, J.-L. Dekeyser, A. Cuccuru, C. Dumoulin, and C. Andre. MARTE : A new profile RFP for the modeling and analysis of real-time embedded systems. In *UML-SoC'05, DAC 2005 Workshop UML for SoC Design*, Anaheim, CA, June 2005.

[56] M. Shaw and al. Abstractions for software architecture and tools that support them. *IEEE Transactions on Software Engineering*, 21(4), 1995.

[57] Bernardi Simona and Dorina Petriu. Comparing two UML profiles for non-functional requirement annotations : the SPT and QoS profiles. In *Seventh International Conference on the Unified Modeling Language (UML 2004)*, Lisbon, Portugal, October 2004.

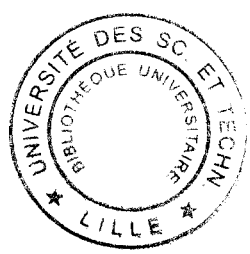
[58] Yves Sorel and Christophe Lavarenne. *SynDEx Documentation Index*. INRIA, 2000. http://www-rocq.inria.fr/syndex/doc/.

[59] Sparx Systems. Enterprise architect. http://www.sparxsystems.com.au/, 2005.

[60] Bassam Tabbara and Kamal Hashmi. Transaction-level modeling and debug of SoCs. In *IP Based SoC Design*, France, December 2004.

[61] The Carroll Research Programme. , 2005. http://www.carroll-research.org/.

[62] VSI Alliance, editor. *Virtual Component Interface Standard Version 2*. http://www.vsi.org/, April 2001.

[63] M. Williams. *A History of Computing Technology*. IEEE Comp. Soc., 1997.

[64] Qiang Zhu, Akio Matsuda, Shinya Kuwamura, Tsuneo Nakata, and Minoru Shoji. An object-oriented design process for system-on-chip using UML. In *Proceedings of the 15th international symposium on System Synthesis*, pages 249–259, Kyoto, Japan, 2002.