Contrôle matériel des systèmes partiellement reconfigurables sur FPGA : de la modélisation à l'implémentation

Chiraz Trabelsi

Université des Sciences et Technologies de Lille

Thèse

présentée pour obtenir le titre de
DOCTEUR spécialité Informatique
par

CHIRAZ TRABELSI

Contrôle matériel des systèmes partiellement reconfigurables sur FPGA : de la modélisation à l'implémentation

Thèse soutenue le 03 juillet 2013, devant la commission d'examen formée de :

Jean-Philippe DIGUET ...	Directeur de recherche CNRS, Lab-STICC, UMR6285, Lorient	**Rapporteur**
Sébastien PILLEMENT ..	Professeur Polytech - Université de Nantes	**Rapporteur**
Dragomir MILOJEVIC ...	Professeur Université Libre de Bruxelles	**Examinateur**
Smail NIAR.....	Professeur Univérsité de Valenciennes	**Examinateur**
Samy MEFTALI	Maître de Conférences Université Lille 1 Sciences et Technologies	**Directeur**
Jean-Luc Dekeyser	Professeur Université Lille 1 Sciences et Technologies	**Co-Directeur**

UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE
LIFL - UMR 8022 - Cité Scientifique, Bât. M3 - 59655 Villeneuve d'Ascq Cedex

Table des matières

Table des figures

Liste des tableaux

Liste des algorithmes

Liste des listages

Chapitre 1

Introduction

1.1 Contexte de la thèse

Les systèmes embarqués envahissent jour après jour notre vie quotidienne et professionnelle. Ils couvrent plusieurs domaines tels que la télécommunication, l'avionique, l'automobile, la santé, etc. Grâce à un taux d'intégration de transistors sur la même puce qui n'a pas cessé d'augmenter, un type spécial de systèmes complexes apparaît dans les années 70. Ce sont les systèmes sur puce [1]. Un système sur puce (SoC) est un système hétérogène capable d'intégrer plusieurs composants sur une même puce. Parmi les composants qu'on peut trouver dans un système sur puce, on compte les processeurs (hardcore, softcore, DSP, ...), les mémoires (RAM, ROM, flash, ...), les interconnexions qui peuvent suivre plusieurs topologiques (bus, crossbar, réseau sur puce, ...), les interfaces externes (USB, Ethernet, PCI, FireWire, ...), les convertisseurs analogiques-numériques, les capteurs et les horloges. La figure 1.1 illustre un exemple de système sur puce [2].

Les systèmes sur puce peuvent être statiques ou reconfigurables. Parmi les implémentations les plus utilisées pour un système statique, on trouve les implémentations sur des ASICs (Application-Specific Integrated Circuits). Comme leur nom l'indique, les ASICs sont des circuits personnalisés pour une application spécifique. Ils ont comme avantages la réduction du nombre de composants sur le circuit imprimé par rapport aux circuits génériques, la confidentialité qu'offre la personnalisation du circuit et l'optimisation du circuit en termes de performances et de consommation d'énergie. Cependant, les ASICs ont plusieurs inconvénients à cause du fait qu'ils soient dédiés seulement à une seule application qui ne peut pas être modifiée après la fabrication, et du coût de conception élevé de ces circuits qui ont aussi un long temps de mise sur le marché au

[1] http ://www.computerhistory.org/semiconductor/timeline/
[2] http ://en.wikipedia.org/wiki/System_on_a_chip

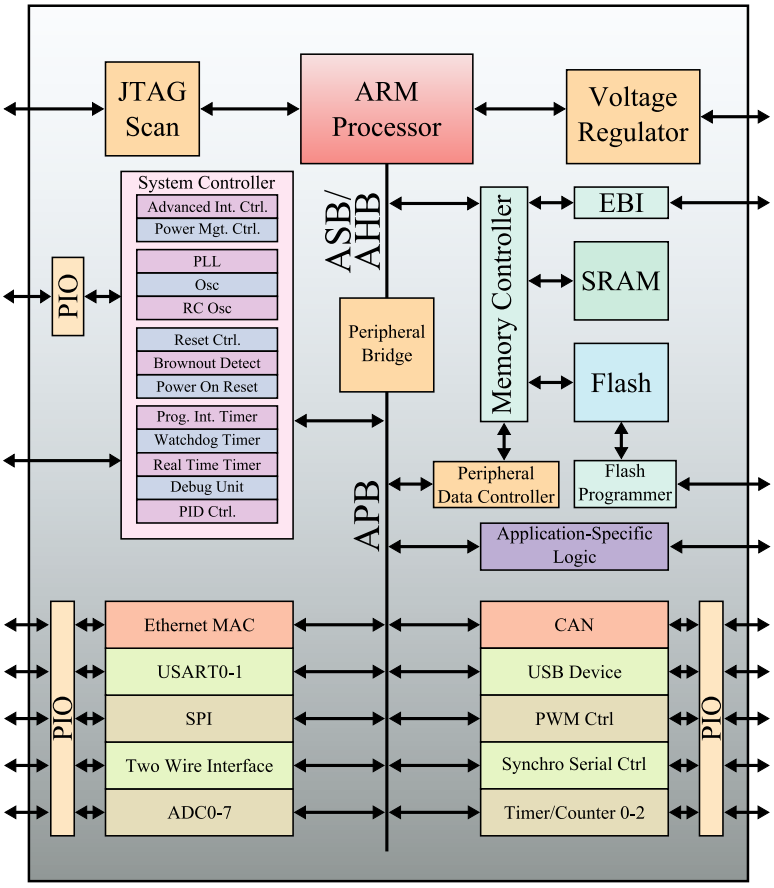


FIG. 1.1 – Exemple de système sur puce

cours duquel plusieurs erreurs de conception peuvent se produire. Comme alternative apportant souvent des solutions acceptables à ces problèmes, les circuits programmables et reconfigurables sont utilisés. Les systèmes reconfigurables sur puce peuvent intégrer les mêmes composants supportés par un SoC traditionnel à la différence qu'ils sont implémentés sur du matériel reconfigurable. De ce fait, ils offrent une grande flexibilité par rapport aux SoCs classiques. La reconfiguration permet à ces systèmes d'être reconfigurés un nombre illimité de fois, et offre aux concepteurs la possibilité d'ajouter de nouvelles fonctionnalités et d'effectuer des modifications sur un système après sa fabrication. La reconfiguration dynamique est un type spécial de la reconfiguration, qui permet la modification d'un système au cours de l'exécution en introduisant le concept du matériel virtuel. Ce type de reconfiguration permet aux systèmes de s'adapter à des changements au cours de l'exécution. Ces changements peuvent être liés aux préférences des utilisateurs, aux contraintes de qualité de service, de surface, de consommation d'énergie, etc.

Dans ce contexte, les FPGAs (Field Programmable Gate Arrays) représentent une solution idéale pour implémenter la reconfiguration dynamique du fait de leur grande flexibilité par rapport aux SoCs traditionnels. La reconfiguration dynamique partielle

Technologie	Année	Nombre maximal de cellules logiques
220nm	1998	27.648 (XCV1000 [140])
130nm	2002	99.216 (XC2VP100 [149])
90nm	2004	200.448 (XC4VLX200 [147])
65nm	2006	300.000 (XC5VLX330 [145])
40nm	2009	758.784 (XC6VLX760 [154])
28nm	2010	1.954.560 (XC7V2000T [150])

TAB. 1.1 – Evolution technologique de la famille Virtex de Xilinx

(RDP) est un type de reconfiguration supporté par certains FPGAs et permet de reconfigurer une partie du FPGA sans perturber le fonctionnement du reste. Ce type de reconfiguration permet par exemple de remplacer un module matériel, réalisant une tâche de l'application par un autre. Cela permet de cibler de larges applications en utilisant un nombre limité de ressources. Un tel mécanisme permet d'améliorer la performance globale en faisant se chevaucher l'exécution et la reconfiguration des différentes portions du système.

Grâce à une évolution technologique continue, le nombre de cellules logiques disponibles sur FPGA n'a pas cessé d'augmenter permettant ainsi de couvrir des applications d'une taille de plus en plus importante. La miniaturisation des FPGAs, et des circuits intégrés en général, suit la loi de Moore [85]. Selon cette loi, le nombre de transistors intégrés sur puce double chaque deux ans. Le tableau 1.1 illustre l'évolution technologique de la famille Virtex de Xilinx. Ceci montre bien que cette évolution est allée au-delà de la loi de Moore en 2010 avec le Virtex XC7V2000T. Ce FPGA n'est pas le résultat d'un simple passage d'un noeud technologique à un autre. En effet, pour assurer un nombre suffisant de ressources pour l'application ciblée, le passage d'un noeud technologique à un autre n'est plus suffisant vu le temps que prend ce processus. Pour résoudre ce problème, Xilinx a utilisé une nouvelle technologie pour fabriquer des FPGAs 2.5D [36] basés sur la connexion d'un nombre d'FPGA dans le même package. Avec cette technologie, le FPGA XC7V2000T offre presque 2 millions de cellules logiques permettant de cibler des applications de grandes tailles. Pour augmenter le nombre de cellules logiques disponibles pour un système basé sur FPGA, d'autres solutions ont également apparu telles que les systèmes multi-FPGA [113] [4] [92] [86] et les FPGAs 3D [68].

Devant l'évolution technologique continue des FPGAs, ils sont devenus capables de cibler des applications de plus en plus sophistiquées. Cependant, puisque les outils de conception n'évoluent pas au même rythme que la technologie matérielle, cela a engendré une grande complexité de conception résultant en un « gap » de productivité. La reconfigurabilité de tels systèmes augmente encore plus cette complexité en impliquant des tâches additionnelles de conception liées principalement au contrôle de l'adaptation dynamique. Dans ce contexte, une méthodologie efficace de conception de contrôle est requise afin de faciliter le travail des concepteurs et d'améliorer leur productivité. Les travaux décrits dans ce manuscrit s'inscrivent dans le contexte de la conception du contrôle des systèmes partiellement reconfigurables sur FPGA.

1.2 Motivations

Afin de faire face à la complexité croissante des systèmes ciblés par les technologies FPGA, un nombre de facteurs clés est requis pour la méthodologie de conception du contrôle qui sont :

- la flexibilité,
- la réutilisabilité,
- la scalabilité,
- et l'automatisation.

Une méthodologie basée sur ces facteurs permet de faciliter le travail du concepteur et d'améliorer sa productivité. Un cinquième facteur qui ne doit pas être négligé est l'efficacité de l'implémentation du contrôle en termes de temps d'exécution, de coût de communication, etc.

Le contrôle des systèmes reconfigurables implique souvent trois tâches principales : 1) la collecte d'informations qui pourraient déclencher la reconfiguration du système, 2) la prise de décision de reconfiguration et 3) le lancement de la reconfiguration. L'utilisation d'un contrôleur centralisé pour gérer ces trois tâches rend la conception d'un tel contrôleur très dépendante du système ciblé, ce qui représente une grande rigidité de conception et un obstacle devant la réutilisabilité et la scalabilité. De plus, du point de vue implémentation, avoir un contrôleur centralisé pour des systèmes de grande taille peut aussi engendrer des goulets d'étranglement entre le contrôleur et les composants qui font l'observation des données à prendre en compte pour la prise de décision. Ce problème peut influencer énormément la performance globale des systèmes de grande taille tels les systèmes multi-FPGAs où la communication entre les FPGAs peut avoir un coût non négligeable. En outre, pour les systèmes supportant la reconfiguration parallèle tels que les systèmes multi-FPGAs et les FPGAs 3D, le contrôle centralisé n'est pas le plus adapté pour le lancement des reconfigurations à cause de son comportement séquentiel, ce qui peut affecter la performance du système.

Une solution à ce problème, peut être de décomposer le contrôle entre un ensemble de contrôleurs gérant chacun des problèmes de contrôle locaux. Cette solution favorise la réutilisation de chaque contrôleur et facilite ainsi la scalabilité du contrôle pour des systèmes de grande taille. De plus, la distribution du contrôle permet de mieux s'adapter à l'évolution constante des tailles des systèmes ciblés en évitant les problèmes de communication centralisée. Cependant, appliquer ce modèle pour un système où les problèmes de contrôle gérés par chaque contrôleur sont dépendants et nécessitent une coordination entre contrôleurs diminue sa flexibilité. En effet, dans ce cas, chaque contrôleur doit communiquer avec les autres contrôleurs afin de s'assurer que les décisions de reconfiguration prises par chacun d'eux respectent bien les contraintes ou objectifs globaux du système. Chaque contrôleur nécessite donc d'avoir une vision globale du système, ce qui augmente sa dépendance du système ciblé et pourrait rendre difficile sa réutilisation pour d'autres systèmes. Cela représente aussi un obstacle devant la scalabilité. De plus, les échanges entre contrôleurs avant d'arriver à une configuration globale qui respecte les contraintes peuvent être d'un coût non négligeable. En effet, dans un modèle de contrôle distribué, chaque contrôleur doit dans certains cas envoyer ses décisions de reconfigurations à tous les autres contrôleurs, ce qui implique un grand

nombre de messages échangés. Ceci pourrait avoir un impact non négligeable sur la performance globale du système.

Dans ce contexte, une prise de décision hiérarchique ou semi-distribuée pourrait être bénéfique. Une structure à deux niveaux permet à chaque contrôleur du niveau inférieur d'avoir seulement une vision locale du problème de contrôle, favorisant ainsi sa réutilisation. Le contrôleur du niveau supérieur ne gère le problème du contrôle qu'à partir d'une vision globale sans entrer dans les détails des problèmes locaux des contrôleurs. Cela facilite une conception générique de ce contrôleur favorisant ainsi sa réutilisation, moyennant quelques modifications, pour d'autres systèmes. Dans ce cas, le contrôle hiérarchique facilite la scalabilité par rapport au modèle centralisé et au modèle distribué à un seul niveau. De plus, un tel modèle est facilement adaptable à l'évolution des systèmes FPGAs, que ce soit des systèmes mono-FPGAs, multi-FPGAs ou des FPGAs 3D et permet de minimiser le coût de la communication entre contrôleurs en la divisant en des communications locales et des communications globales selon les niveaux hiérarchiques utilisés.

L'utilisation d'un formalisme pour la conception du contrôle favorise aussi la flexibilité et la réutilisation. En effet, un formalisme de contrôle permet de traiter le comportement du système d'une manière abstraite avec des sémantiques précises, ce qui facilite la modification et la réutilisation du contrôle. Cela offre aussi la possibilité d'implémenter le contrôle ainsi conçu de différentes manières visant différentes plates-formes FPGA ou autres. Les formalismes souvent utilisés dans ce contexte sont basés sur les machines à états ou des réseaux de Pétri. Ces formalismes offrent une grande flexibilité avec des possibilités de compositions parallèles et hiérarchiques qui peuvent être facilement adapté au contrôle centralisé et décentralisé.

Un autre facteur facilitant le travail du concepteur et améliorant sa productivité est l'automatisation. Dans ce contexte, l'élévation du niveau de conception permet de cacher les détails techniques aux concepteurs du contrôle souvent source d'erreurs. Les systèmes de contrôle peuvent ensuite être générés automatiquement par les concepteurs sans qu'ils soient experts des FPGAs modernes. L'ingénierie dirigée par les modèles (IDM) est une approche très intéressante dans ce domaine. Avec l'IDM, les modèles deviennent un moyen de productivité. Sa nature graphique offerte par UML (Unified Modeling Language) rend la compréhensibilité d'un système plus facile et permet aux utilisateurs de modéliser leurs systèmes à haut niveau d'abstraction, de les réutiliser, les modifier et les étendre facilement, et de générer automatiquement le code correspondant.

Il y a plusieurs façons d'implémenter le contrôle des systèmes reconfigurables. Cette implémentation peut être logicielle ou matérielle, standalone ou par un système d'exploitation (OS). Dans ce cas, on peut dégager 4 premières catégories d'implémentations : logicielle standalone, matérielle standalone, logicielle par OS ou matérielle par OS. Des implémentations hybrides sont aussi possibles. Chacune de ces implémentations a ses avantages et ses inconvénients. L'avantage d'une implémentation logicielle est qu'elle ne nécessite pas beaucoup de ressources par rapport à une implémentation matérielle. Son inconvénient est son temps d'exécution qui peut être beaucoup moins important pour une implémentation matérielle grâce au parallélisme matériel. L'avantage d'une implémentation par OS est qu'elle offre une abstraction par rapport la plate-forme cible et permet d'offrir des services pour des systèmes complexes tels que les systèmes

multi-applications. Cependant, un OS a un coût d'exécution important par rapport à une solution standalone. Une implémentation matérielle des services de l'OS permet de diminuer le temps d'exécution, mais pourrait résulter en un coût important en termes de ressources matérielles et donc de consommation d'énergie. Tous ces points doivent être pris en compte lors de la conception du contrôle afin de bien respecter les contraintes des systèmes ciblés en termes de services, de performance et de ressources.

1.3 Contributions

La contribution de cette thèse concerne la proposition d'une méthodologie de conception de contrôle pour les systèmes reconfigurables basés sur FPGA visant à assurer la flexibilité, la réutilisabilité, la scalabilité et l'automatisation afin de faciliter le travail des concepteurs et d'améliorer leur productivité. Cette méthodologie doit en même temps permettre d'avoir une implémentation efficace du contrôle en termes de temps d'exécution, de coût de communication, etc. La méthodologie proposée dans ce travail [130] est basée sur un modèle de contrôle semi-distribué [129] où chaque contrôleur distribué contrôle l'auto-adaptation d'une région reconfigurable du système et les décisions de reconfiguration de ces contrôleurs sont coordonnées par un coordinateur afin de vérifier la configuration globale du système respecte bien ses contraintes/objectifs. Ce modèle assure la flexibilité, la réutilisabilité et la scalabilité grâce à :

- la décomposition du contrôle en des problèmes locaux gérés par les contrôleurs distribués et un problème global géré par le coordinateur global favorisant la réutilisation de chacun des contrôleurs et donc la scalabilité du modèle du contrôle, par rapport à un modèle centralisé ou complètement distribué,
- sa couverture des cas où le problème du contrôle nécessite une coordination entre contrôleurs. Ce sont les cas où les décisions prises en local pourraient avoir un impact sur le contrôle global du système. Cela permet ainsi de viser différents problèmes de contrôle,
- la modularité des contrôleurs favorisant la flexibilité et la réutilisation
- la distribution des services de reconfiguration pour les régions leur permettant d'être auto-adaptative et de bénéficier facilement de la reconfiguration parallèle dans les systèmes multi-FPGAs et futures technologies d'FPGA,
- la facilité de l'adaptation de ce modèle à différentes plates-formes (mono-FPGA, multi-FPGAs, FPGA 3D, etc) en adaptant les niveaux de hiérarchie dans le modèle de prise de décision,
- et l'utilisation d'un formalisme pour la conception de la prise de décision semi-distribuée favorisant aussi sa réutilisabilité et facilitant sa modification et son adaptation à différents systèmes.

Cette méthodologie est basée aussi sur une conception à haut-niveau [128] permettant de rendre les détails d'implémentation transparents aux concepteurs et d'automatiser la génération des systèmes de contrôle. Le modèle proposé assure aussi une efficacité en termes de coût de communication et d'exécution. En effet, la distribution spatiale lui permet de diminuer les coûts de communications locales liées à la collecte de données d'observation et la prise de décision locale, ce qui diminue l'impact du contrôle sur la

performance du système en favorisant aussi le parallélisme du traitement des données d'observation. Différentes implémentations sont possibles pour le modèle de contrôle proposé en logiciel, en matériel, en standalone ou en tant que services d'OS. Dans ce travail, nous proposons une implémentation matérielle standalone afin de réduire le coût du contrôle en temps d'exécution par rapport à une implémentation logicielle. Une implémentation matérielle en tant que services d'OS est aussi possible, mais n'est pas traitée dans ce travail. Le reste de cette section donne plus de détails sur la contribution de cette thèse.

Modèle de contrôle décentralisé et modulaire : Le modèle proposé divise le problème de contrôle en un ensemble de sous-problèmes locaux gérés par des contrôleurs autonomes. Chaque contrôleur gère l'auto-adaptation d'un composant reconfigurable du système à travers trois tâches allouées à trois modules différents : 1) l'observation des événements susceptibles de déclencher l'adaptation du composant contrôlé, 2) la prise de décision concernant les adaptations nécessaires et 3) la réalisation de l'adaptation/reconfiguration du composant contrôlé. L'allocation des tâches du contrôleur à des modules séparés facilite leur modification et réutilisation et par conséquent la scalabilité des systèmes de contrôle.

La coordination entre contrôleurs distribués : A cause de la vision locale de chaque contrôleur, le lancement de la reconfiguration du composant contrôlé pourrait avoir un effet non désirable sur le reste du système. En effet, avant de lancer une reconfiguration d'un composant, il faut vérifier si la configuration cible pour ce composant peut coexister avec les configurations courantes des autres composants du système à cause des contraintes de sécurité, de qualité de service, etc. Dans le cas contraire, il faut vérifier si les contrôleurs de ces régions sont d'accord pour une reconfiguration afin de respecter ces contraintes. Pour résoudre ce problème, nous proposons un mécanisme de coordination entre les contrôleurs. Cette coordination est réalisée par un coordinateur, ce qui rend le modèle de contrôle un modèle semi-distribué. L'avantage par rapport à une solution complètement distribuée est l'amélioration de la réutilisation de la conception. En effet, dans un modèle complètement distribué, les contrôleurs échangent directement leurs décisions, ce qui nécessite une vision globale de chaque contrôleur afin de réagir correctement aux décisions envoyées par les autres contrôleurs et respecter les contraintes globales du système. Cela rend le problème de contrôle géré par chaque contrôleur plus complexe et plus dépendent de l'implémentation du système et des contrôleurs avec qui il communique, ce qui représente un obstacle à la réutilisation de conception. Au lieu de cela, le modèle semi-distribué proposé se base sur des contrôleurs qui ont une vision locale du système, ce qui les rend plus facile à concevoir et à réutiliser pour des systèmes différents. De même, le rôle du coordinateur est de coordonner les décisions des contrôleurs sans entrer dans les détails des problèmes locaux des contrôleurs. Cela facilite une conception générique de ce coordinateur favorisant ainsi sa réutilisation pour d'autres systèmes.

Un formalisme pour la prise de décision : La prise de décision de reconfiguration est l'un des aspects les plus critiques dans la conception du contrôle. Pour diminuer la complexité de la conception de cet aspect, l'utilisation d'un formalisme de contrôle permet l'abstraction du problème de contrôle et diminue sa dépendance de l'implémentation du système. Elle facilite aussi la modification, la réutilisation et la scalabilité de la

conception. Dans le contexte des systèmes reconfigurables, le formalisme des automates de modes est un formalisme intéressant permettant de modéliser les différents modes d'un système ou un sous-système [73]. Ce formalisme offre aussi la possibilité de composition d'automates parallèles et hiérarchiques, ce qui permet une grande flexibilité de conception. Appliqué aux systèmes sur FPGA, le comportement d'un composant/région reconfigurable sur FPGA peut être modélisé par un automate de modes où chaque mode correspond à une de ses configurations possibles et où les conditions de transitions correspondent aux conditions déclenchant la prise de décision de reconfiguration. Dans ce travail, nous proposons un modèle de prise de décision semi-distribuée inspiré de ce formalisme.

Une conception à haut niveau dirigée par les modèles : Pour assurer une bonne productivité dans la conception du contrôle, notre méthodologie utilise une conception à haut-niveau d'abstraction basée sur l'ingénierie dirigée par les modèles (IDM). La modélisation de nos systèmes de contrôle a été faite dans le cadre de l'environnement Gaspard2 [42] dédié à la conception des systèmes embarqués. Cet environnement utilise le profil standard MARTE (Modeling and Analysis of Real-Time and Embedded systems) qui est spécifique au domaine des systèmes embarqués. Des extensions ont été proposées pour ce profil afin de faciliter, aux concepteurs des systèmes embarqués, la modélisation du contrôle semi-distribué. En utilisant l'automatisation offerte par l'IDM, notre approche permet de générer automatiquement le code d'un système de contrôle semi-distribué modélisé à haut-niveau en utilisant le profil MARTE.

Implémentation matérielle du contrôle semi-distribué sur FPGA : Pour l'implémentation du modèle de contrôle proposé, nous utilisons une solution matérielle afin d'éviter le surcoût en temps d'exécution de la solution logicielle. A l'aide de la conception dirigée par les modèles, différents systèmes intégrant le contrôle semi-distribué ont été modélisés. Le déploiement de ces modèles ciblent la technologie FPGA de Xilinx. La génération de code permet d'obtenir le code VHDL des systèmes de contrôle composés par les contrôleurs et les coordinateurs. Les codes générés ont permis de valider le modèle de contrôle semi-distribué proposé et ont montré que son surcoût en termes de ressources matérielles utilisées par rapport à une solution centralisée est acceptable devant les avantages qu'il offre par rapport à une solution centralisée en termes de flexibilité, de réutilisation et de scalabilité.

1.4 Plan de la thèse

Le chapitre 2 présente les concepts de base liés aux FPGA et à la reconfiguration puis explique avec détails le concept de la reconfiguration dynamique partielle, son application pour différents types de reconfigurations architecturales et le flot de conception des systèmes la supportant.

Dans le chapitre 3, différents travaux sur le contrôle des systèmes reconfigurables sont présentés et discutés. Quatre principaux aspects sont traités à travers ces travaux : l'architecture du contrôle, les mécanismes de coordination entre contrôleurs, l'utilisation du formalisme des automates de modes pour le contrôle et l'automatisation de la génération du contrôle à partir des modèles de haut niveau d'abstraction.

Dans le chapitre 4, notre modèle du contrôle semi-distribué est présenté. Après la description de la structure de ce modèle, la conception de la prise de décision basée sur le formalisme des automates de mode est détaillée.

Le chapitre 5 présente le flot de conception à haut-niveau du modèle de contrôle. L'intégration de ce flot dans l'environnement Gaspard2 est détaillée en présentant les extensions qui ont été faites pour passer des modèles de haut-niveau à la génération du code.

Une application est étudiée dans le chapitre 6 pour illustrer les étapes de conception présentées dans cette thèse.

Le chapitre 7 conclut ce manuscrit et donne quelques perspectives au travail présenté.

Chapitre 2

Les FPGAs et la reconfiguration dynamique

2.1 Introduction

Dans ce chapitre, nous présentons les concepts de base des systèmes reconfigurables sur FPGA. D'abord, la notion des circuits reconfigurables, les FPGAs et leur évolution technologique sont présentés. Plus de détails sont ensuite donnés sur le concept de la reconfiguration dynamique partielle, son application pour différents types de reconfigurations architecturales et le flot de conception des systèmes la supportant.

2.2 Les circuits reconfigurables

2.2.1 Avantages et inconvénients

Les FPGAs sont une catégorie de circuits reconfigurables (circuits qui peuvent être reconfigurés après leur fabrication). Les circuits reconfigurables sont apparus comme solution au manque de flexibilité dont souffrent les SoCs classiques tels que ceux basés sur les ASICs (Application Specific Integrated Circuits). Comme solution à ces problèmes, les circuits reconfigurables sont utilisés. Ces circuits ont pour avantages la conception rapide et la reconfiguration. Ce deuxième avantage permet aux circuits reconfigurables de changer de configuration selon l'application. De plus, il est possible d'effectuer la reconfiguration durant l'exécution de l'application comme c'est le cas pour les FPGAs. Cela résout beaucoup de problèmes qui peuvent se produire au cours de l'exécution et qui peuvent être liés à la tolérance aux fautes ou à la performance. En outre, la reconfiguration peut être effectuée un nombre arbitraire de fois. Un autre avantage des circuits reconfigurables tels que les FPGAs est que les SoCs construits à base de ces circuits ont une durée de conception réduite en comparaison à celles des ASICs. Ceci est dû au fait que les changements sur les FPGAs peuvent être immédiats avant la fabrication, ce qui fait des FPGAs une solution pour le prototypage des systèmes à base d'ASICs.

L'inconvénient des SoCs reconfigurables est leur coût en comparaison aux SoCs personnalisés comme les ASICs. Alors que les systèmes personnalisés ont un coût de conception important, ils ont un coût unitaire de puce moins important que celui des systèmes reconfigurables.

2.2.2 Structure de base

Un circuit intégré classique contient :
- des portes logiques,
- des connexions entre les portes logiques,
- des éléments de mémorisation (registres et/ou mémoires),
- des entrées/sorties,
- une (ou des) horloge(s)

Un circuit reconfigurable a les mêmes éléments de base qu'un circuit statique avec la notion de reconfigurabilité. Partant du principe qu'une fonction logique peut s'exprimer sous forme canonique (somme de produits), cette fonction logique peut être représentée par un réseau logique programmable constitué d'une matrice ET et d'une matrice OU. De cette façon, en agissant sur les liaisons de la matrice ET ou la matrice OU, on peut changer la fonction réalisée. C'est ce qu'on appelle programmabilité/reconfigurabilité. Parmi les circuits reconfigurables basés sur ce principe, on cite les PLA (Programmable Logic Array) où la matrice ET et la matrice OU sont reconfigurables, et les circuits PAL (Programmable Array Logic) où seule la matrice ET est reconfigurable. Une fonction logique reconfigurable peut aussi être implémentée par les tables de vérité (Look Up Tables : LUT) construites à base de mémoire SRAM par exemple. Dans ce cas, l'adresse d'une case mémoire constitue une combinaison des entrées d'une fonction. Le contenu de la case mémoire correspond à la valeur que prend la fonction pour cette combinaison.

En combinant des cellules/réseaux logiques reconfigurables tels que les LUTs et les matrices ET et OU, on obtient des circuits logiques reconfigurables PLD (Programmable Logic Device). Les circuits PLD peuvent se décomposer en trois catégories : les SPLDs (Simple PLDs), les CPLDs (Complex PLDs) et les FPGAs. Les SPLDs sont des circuits composés d'un bloc entrée, d'une matrice ET, d'une matrice OU et d'un bloc de sortie. Dans cette catégorie, on trouve les circuits PLA et PAL dont on a parlé précédemment. Les circuits CPLDs regroupent un ensemble de circuits reconfigurables. Les FPGAs sont composés d'un réseau de blocs logiques, de cellules d'entrée/sortie et de ressources d'interconnexion totalement flexibles. La différence entre les FPGAs et les CPLDs est au niveau de l'architecture. Alors qu'un CPLD est constitué d'un ensemble de PLDs interconnectés à la matrice globale d'interconnexion, un FPGA a ses propres blocs logiques et chaque bloc peut implémenter une fonction logique. Ces blocs sont connectés, par des commutateurs reconfigurables, pour implémenter une fonction logique complète, à la différence d'un CPLD où des fonctions logiques complètes sont implémentées individuellement par les PLDs.

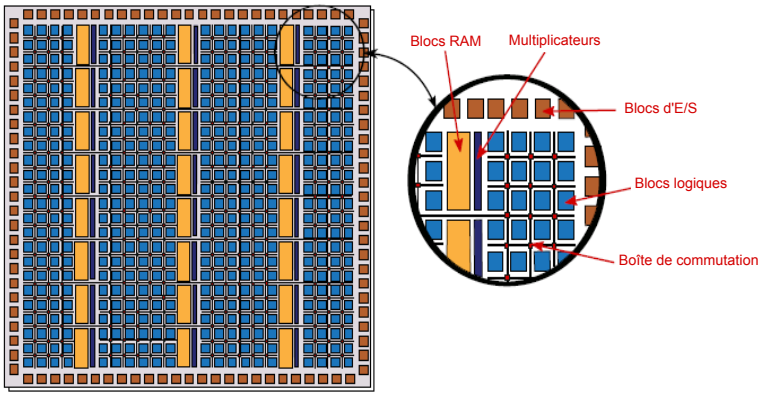


FIG. 2.1 – Les différents blocs d'un FPGA Xilinx

2.2.3 Les FPGAs

Les FPGAs peuvent être utilisés pour implémenter n'importe quelle fonction logique que les ASICs peuvent implémenter. Leur reconfiguration, qui peut être effectuée un nombre arbitraire de fois, représente l'un de leurs avantages majeurs par rapport aux ASICs. Un FPGA est généralement composé de deux couches. La première couche contient entre autres des blocs logiques reconfigurables (CLBs : Configurable Logic Blocks ou LEs : Logical Elements selon la terminologie du vendeur) et des blocs d'E/S, comme le montre la figure 2.1. Dans l'exemple des FPGAs de Xilinx, les blocs logiques sont interconnectés par une hiérarchie d'interconnexions reconfigurables. Ils sont généralement composés de LUTs (Look Up Tables) pour réaliser des fonctions élémentaires et des bascules flip-flop pour la mémorisation. Les fils de connexion sont des fils horizontaux et verticaux disposés entre les lignes et les colonnes des blocs logiques, comme le montre la figure 2.2. Chaque fil peut être vu comme un ensemble de segments. Chaque

segment couvre un bloc logique avant de se terminer dans une boîte de commutation. Chaque boîte de commutation relie donc quatre segments, ce qui donne à chaque segment trois possibilités de connexion aux autres segments. En activant un commutateur, on relie les segments d'interconnexion pour obtenir de longs chemins. Le fait d'avoir des lignes courtes est moins performant, vu le temps que prend le signal pour transiter d'une boîte de commutation à une autre. Pour cela, les FPGAs utilisent aussi des lignes de routage qui couvrent plusieurs blocs logiques.

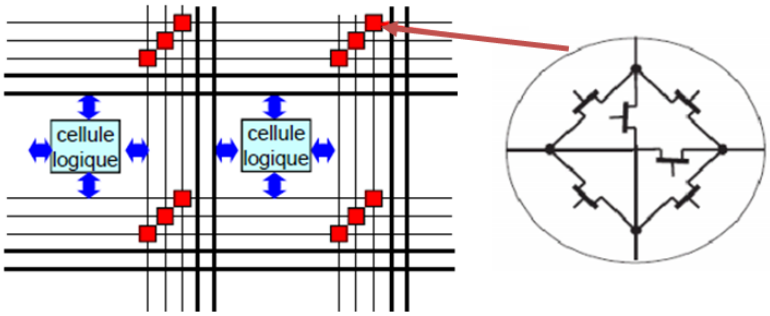


FIG. 2.2 – Connexions entre les blocs d'un FPGA

A côté des blocs logiques, la première couche du FPGA contient aussi des blocs de mémoire (BRAMs) et d'autres blocs qui ont des fonctionnalités spécifiques et sont utilisés fréquemment dans les systèmes implémentés sur FPGA. Le fait d'intégrer ces blocs sur FPGA permet de réduire la surface requise et d'augmenter la performance de ces blocs par rapport à leur implémentation en utilisant les blocs logiques. Parmi ces blocs, on trouve les multiplicateurs, les blocs DSP, les blocs mémoires et les processeurs embarqués.

La deuxième couche d'un FPGA constitue la mémoire de configuration. Pour écrire dans cette mémoire, on utilise des fichiers binaires appelés bitstreams. Ces fichiers contiennent les informations de contrôle pour la configuration ainsi que des données de configuration. Ces données permettent de décrire l'architecture du système se trouvant sur la première couche. La figure 2.3 montre la couche mémoire de configuration pour un Virtex-II Pro. Cette couche est organisée, elle aussi, en colonnes. Chaque colonne est constituée de sous-colonnes appelées trames. La taille de la colonne de la mémoire de configuration dépend du type et de la taille de la colonne de la première couche. Pour un Virtex-II Pro, la trame est la plus petite unité de configuration. Le nombre de trames sur un FPGA et le nombre de bits par trame dépendent de la taille du FPGA. Pour les FPGAs plus récentes tels que virtex-4, l'unité de configuration est plus petite [70]. Elle est sous forme d'une colonne correspondant à 16 CLBs . Cette unité de configuration est indépendante de la taille du FPGA. Pour accéder à une trame, il faut donc avoir l'adresse de la colonne sur laquelle cette trame se trouve et l'adresse de cette trame dans la colonne. Une trame contient une fraction des données de configuration pour une colonne de la première couche. La hauteur de la trame change d'une famille d'FPGAs à une autre. Elle est de 20 CLBs, 40 CLBs et 50 CLBs pour Virtex-5, Virtex-6 et Virtex-7 respectivement [43].

Parmi les technologies mémoire, les plus utilisées pour la mémoire de configura-

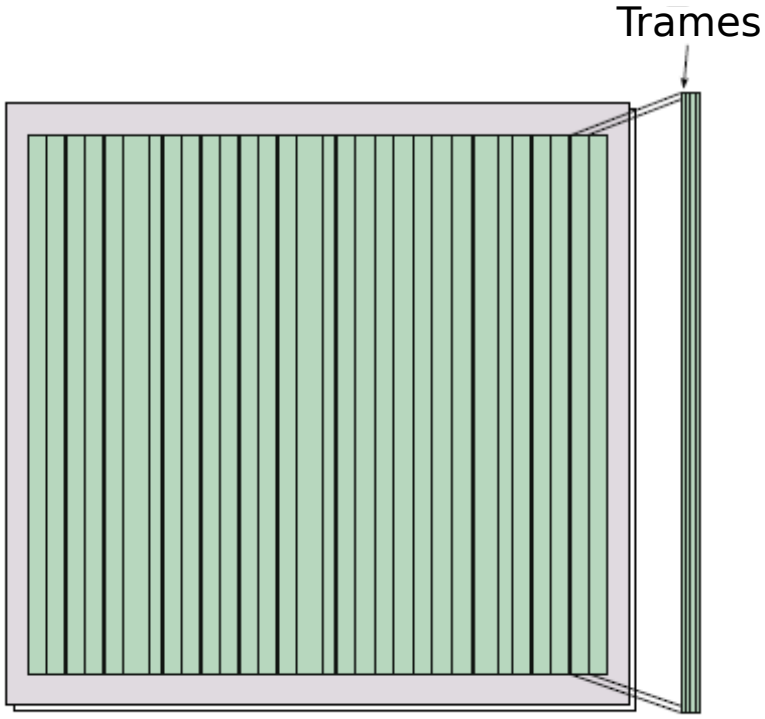


FIG. 2.3 – La mémoire de configuration d'un FPGA Virtex-II Pro de Xilinx

tion du FPGA, nous citons l'EEPROM (Electrically Erasable Programmable Read-Only Memory), Antifuse et SRAM (Static Random Access Memory). La technologie Antifuse est programmable une seule fois, donc elle ne convient pas pour les systèmes reconfigurables. Les FPGAs basés sur SRAM, qui est une mémoire volatile, nécessite une mémoire additionnelle non-volatile pour stocker les données de configuration et les charger sur SRAM à chaque mise sous tension. Malgré son non volatilité et sa taille réduite par rapport à SRAM, le temps de reconfiguration d'une mémoire EEPROM est plus important que celui de la mémoire SRAM [75]. De plus, le développement d'un FPGA basé sur SRAM prend moins de temps que celui d'un FPGA qui utilise EEPROM, ce qui fait que les FPGAs basés sur SRAM ont évolué plus rapidement pour avoir une part plus importante du marché des FPGAs.

2.3 Evolution technologique des systèmes embarqués et des FPGAs

Avant l'apparition des systèmes sur puce, on implémentait les différentes fonctions d'un système (la logique haute-performance, la logique basse-performance, les fonctions analogiques, etc) chacune dans son propre package [78]. L'un des avantages avec ce scénario est une grande flexibilité permettant à chaque package d'être implémenté avec la technologie la plus appropriée. Par contre, de tels systèmes sont relativement larges

et lourds et consomment beaucoup d'énergie. En outre, le temps que prend un signal pour se propager d'une puce à une autre a un impact négatif sur la performance globale du système. La taille et la consommation d'énergie de ces systèmes ne leur permettent pas de cibler de petits appareils alimentés par batterie tels que les smartphones et les tablettes.

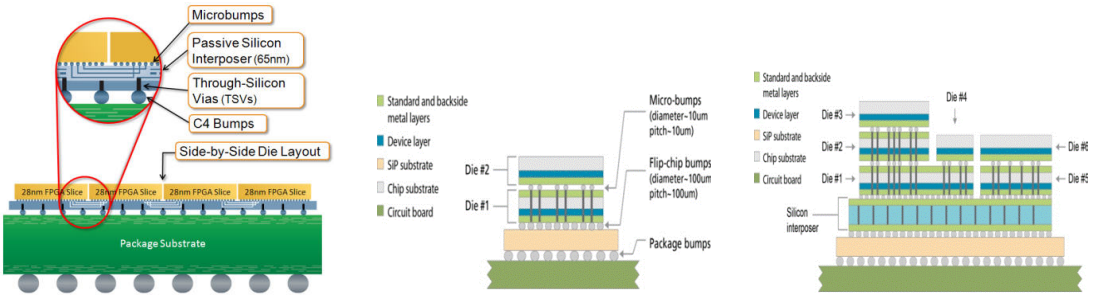
Les systèmes sur puce ont apparu dans le but d'intégrer toutes les fonctions sur une seule puce, afin d'avoir des produits de petites tailles à haute performance et à moindre consommation d'énergie. Cependant, la création de tels systèmes est compliquée et coûteuse en temps. De plus, pour certains systèmes, l'utilisation de fonctions analogiques et numériques sur une même puce est source de problèmes liés au bruit et à l'isolation [78]. Un autre désavantage des systèmes sur puce est que, pour les concepteurs, la modification d'une fonction dans de tels systèmes peut nécessiter la modification de tout le système devant les dépendances impliquées par l'implémentation de toutes les fonctions sur une même puce.

Les désavantages des systèmes sur puce ont entraîné l'apparition des Systems-in-Package (SiP). Les SiP sont constitués d'un ensemble de circuits intégrés qui sont déposés au même substrat qui les connectent tous. Le substrat et ses composants sont placés sur un même package. Le fait d'avoir des circuits intégrés séparés permet à chaque circuit d'être implémentée selon la technologie la plus appropriée. Les concepteurs peuvent aussi réutiliser ces circuits pour d'autres SiP. Cela facilite aussi la modification de ces systèmes qui nécessite généralement la modification d'un sous-ensemble des circuits qui les composent.

Il est possible d'assembler les SiP dans un seul package. Dans ce cas, on parle de Package-in-Package (PiP). Les SiP peuvent aussi être empilés pour construire un Package-on-Package (PoP) [1] permettant d'avoir un grand nombre de ressources dans de petits produits tels que les téléphones portables et les appareils photo numériques. Les PiP et les PoP peuvent être considérés comme des circuits 3D, puisqu'ils contiennent des dice empilés (un die (de l'anglais, pluriel :dice) est un petit morceau de semiconducteur sur lequel un circuit intégré électronique a été fabriqué [2]). Il y a aussi d'autres types de circuits 3D tels que les circuits basés sur une pile de dice connectés par des fils métalliques [76] [77]. Cependant, aucun des circuits 3D cités ci-dessus n'est plus efficace en termes de performance, de densité, de consommation d'énergie et de taille que les circuits 3D utilisant les TSV (Through-Silicon-Vias) [12]. Cette technologie est utilisée ces dernières années par des sociétés telles que Samsung, Micron et Elpida pour la production des mémoires 3D [64]. La technologie 3D est une évolution de la technologie 2.5D qui est basée sur l'utilisation d'un "silicon interposer" auquel les dice sont connectés. La technologie 2.5D permet une connexion plus fine entre les dice du système que celle basée sur la connexion directe sur le substrat, ce qui améliore la performance et réduit la consommation d'énergie. Le "silicon interposer" contient des connexions électriques verticales qui passent à travers un die en silicium permettant la connexion entre les dice du systèmes comme le montre la figure 2.4(a). Cette technologie est appelée 2.5D puisque les dice ne sont pas empilés. Un vrai circuit 3D est un circuit où les dice sont empilés et interconnectés par les TSV comme le montre la figure 2.4(b). A part les

[1] http ://en.wikipedia.org/wiki/Package_on_package
[2] http ://fr.wikipedia.org/wiki/Die_(circuit_intégré

piles de mémoires, il est rare de trouver de nos jours plus de deux dice empilés dans un système commercialisé [12]. Cependant, il est possible, dans le futur proche, de construire des scénarios complexes avec la technologie 3D comme le montre la figure 2.4(c). Vu l'épaisseur très petite de chacun des circuits empilés (de 0.2 à 0.7 mm [77]), les circuits 3D permettent d'avoir des produits de petite taille avec un grand nombre de ressources.



(a) Connexion entre interposer et dice dans la technologie 2.5D [74]

(b) Circuit 3D utilisant les TSV [12]

(c) Circuit 3D à plusieurs étages [12]

FIG. 2.4 – Les technologies FPGA 2.5D et 3D

Comme nous avons précisé dans le chapitre 1, parmi les motivations pour la distribution dans le modèle de contrôle proposé dans ce travail est d'offrir un modèle adaptable à l'évolution des systèmes FPGA. Cette évolution implique entre autres une taille grandissante et une possibilité de reconfiguration parallèle que le modèle distribué permet de gérer plus efficacement que la solution centralisée grâce à son comportement parallèle. Comme nous allons montré dans le reste de ce mémoire, notre modèle de contrôle a été appliqué pour des systèmes contenant un seul FPGA 2D. Cependant, la conception à haut-niveau d'abstraction que nous proposons pour le modèle du contrôle, qui est basée sur l'utilisation d'un formalisme de contrôle et de la modélisation UML-MARTE, le rend indépendant de la technologie ciblée. Cela permet de l'adapter aussi bien pour des systèmes mono-FPGA que pour des systèmes multi-FPFA ou 3D. L'application du modèle du contrôle proposé pour ces deux derniers types de systèmes sera étudiée dans le chapitre conclusion et perspectives.

2.4 Les types de reconfiguration pour les FPGAs

Les reconfigurations peuvent être classées selon où, quand et comment elles sont effectuées.

Où : la reconfiguration peut être externe (effectuée par un module en dehors du FPGA), ou interne (effectuée par le FPGA). Dans une reconfiguration interne, un contrôleur embarqué, qui peut être par exemple un processeur hard/soft, prend en charge la reconfiguration. La reconfiguration interne permet d'éviter les latences liées à la communication entre le contrôleur externe et le FPGA, permettant donc de diminuer l'impact du temps de reconfiguration sur la performance du système.

Quand : la reconfiguration peut être statique ou dynamique. Dans la reconfiguration statique, le FPGA doit être inactif pour être reconfiguré, alors que dans la reconfiguration dynamique le FPGA est reconfiguré durant l'exécution. Ce deuxième type de reconfiguration permet d'éviter le surcoût de la désactivation et de la réinitialisation en termes de temps d'exécution.

Comment : la reconfiguration peut être complète ou partielle (ne concerne que quelques régions du FPGA). Dans la configuration complète, un bitstream décrivant tout le FPGA est chargé même si la reconfiguration n'apporte que quelques modifications sur le système, ce qui constitue une perte de temps énorme qui a un impact négatif sur les performances du système. La reconfiguration partielle représente donc une solution à ce problème.

En se basant sur cette classification, on remarque que la reconfiguration interne dynamique et partielle permet des avantages par rapport aux autres types de reconfiguration. Plusieurs travaux de recherche ont étudié ce type de reconfiguration sous le nom de RDP (Reconfiguration Dynamique Partielle), et c'est celui qu'on traite dans notre travail.

2.5 La reconfiguration dynamique partielle

La reconfiguration dynamique partielle (RDP), illustrée dans la figure 2.5, permet la modification de certaines régions du FPGA durant l'exécution, ce qui permet le partage temporel des ressources matérielles entre les tâches mutuellement exclusives. Il y a plusieurs contextes où la reconfiguration dynamique partielle trouve son utilisation, tels que l'adaptation au changement de l'environnement, à la limitation de ressources, aux exigences de qualité de service, et aux contraintes de consommation d'énergie [104] [105], etc.

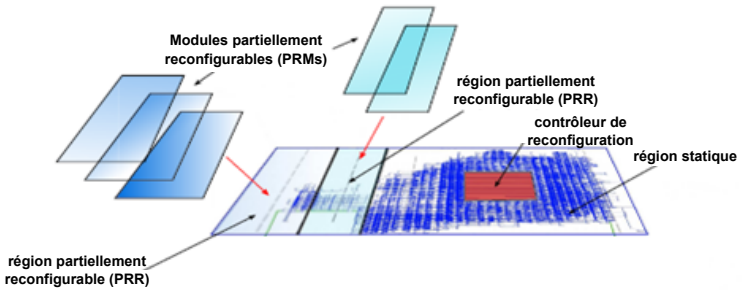


FIG. 2.5 – Vue globale d'un système partiellement reconfigurable

Le support commercial de la reconfiguration dynamique partielle a commencé avec l'apparition de la famille des FPGAs Virtex de Xilinx [79] à la fin des années 90. En 2010, Altera a commercialisé son premier FPGA supportant la reconfiguration dynamique partielle, le Startix-V [6]. Dans notre travail, nous avons utilisé les FPGAs de Xilinx sur lesquels beaucoup de travaux ont été basés. Xilinx a commencé par présenter deux méthodologies de reconfiguration partielle (basée-différence et basée-module) [141]

[143], suivies de la méthodologie EAPR (Early Access Partial Reconfiguration) en 2006 [146], et de Partition-based Partial Reconfiguration en 2010 [43].

2.5.1 Processus de la reconfiguration dynamique partielle

Pour reconfigurer partiellement un FPGA, il est nécessaire d'isoler la zone à reconfigurer et charger les bits de configuration (bitstream) correspondants à cette zone. Un outil appelé PARBIT [50] a été développé par Xilinx afin de générer les bitstreams partiels pour implémenter la reconfiguration partielle.

Pour charger un bitstream sur FPGA, on peut passer par un contrôleur externe ou un contrôleur interne. Pour la reconfiguration partielle externe, on utilise des interfaces telles que JTAG et SelectMap [28]. Pour la reconfiguration partielle interne, les FPGAs de Xilinx utilise l'ICAP (Internal Configuration Access Port) [16], qui est un sous-ensemble de l'interface parallèle à 8-bits SelectMAP [28]. L'ICAP permet donc d'accélérer le processus de reconfiguration dynamique partielle par rapport à l'interface SelectMAP et l'interface série JTAG [122]. L'ICAP est présent presque dans tous les FPGAs Xilinx [13]. Pour les Virtex-II et Virtex-II Pro, l'ICAP fournit des ports d'entrée/sortie de 8 bits, alors que pour Virtex-4, l'interface d'ICAP est de 32 bits avec une possibilité d'alternance entre 8 et 32 bits [29]. Pour les versions ultérieures de Virtex, l'interface d'ICAP offre la possibilité d'alternance entre 8, 16 et 32 bits. La fréquence maximale avec laquelle l'ICAP peut fonctionner est 100Mhz pour les Virtex-4,5,6 et 7 et 20Mhz pour Spartan-6 [152].

Le débit théorique de l'ICAP est la taille d'une donnée traitée divisée par la période du temps, ce qui fait un débit de 100 KO/ms pour une interface d'ICAP de 8 bit et une fréquence de 100Mhz. Cependant, cette valeur ne peut pas être atteinte en réalité. Par exemple, pour les Virtex-II Pro, quand la fréquence de l'horloge de l'ICAP est supérieure à 50 MHz (100 MHz pour Virtex-4), il est nécessaire de respecter le signal « handshaking » (busy) de l'ICAP, ce qui fait que la vitesse théorique maximale ne peut pas être atteinte [25]. Il y a plusieurs travaux qui ont ciblé l'augmentation du débit de l'ICAP. Parmi ces travaux, on cite ceux qui ont étudié l'optimisation du contrôleur de l'ICAP [27] [82] [24] et qui ont permis d'atteindre des débits de 400 MO/s (le débit maximal d'une interface à 32 bits) en utilisant des techniques telles que la connexion directe entre l'ICAP et la mémoire contenant le bitstream [24].

Le nombre d'ICAPs disponibles sur un FPGA est passé d'un seul pour les FPGAs Virtex-II Pro à deux pour les Virtex-4 et les FPGAs plus récents. Ces deux ICAPs sont placés généralement au centre du FPGA. Ils sont physiquement adjacents, l'un au-dessous de l'autre. Ces deux ICAPs ne peuvent pas être actifs simultanément. Le système doit commencer par l'ICAP situé au-dessus, qui est actif par défaut [153]. Au cours de l'exécution, on peut commuter vers l'autre ICAP en écrivant un bit de contrôle. L'utilité de cette redondance est que, dans le cas d'un dysfonctionnement ou blocage du premier ICAP, on peut toujours sélectionner le deuxième ICAP, ce qui laisse le temps pour réinitialiser le premier ICAP sans interrompre l'exécution de l'application.

ICAP ne peut pas être directement connecté au bus système puisqu'il nécessite un contrôleur pour gérer le flot de données venant du contrôleur de reconfiguration. L'outil EDK de Xilinx propose trois types de contrôleurs, le contrôleur OPB [142] qui permet l'interfaçage avec un bus OPB (On-chip Peripheral Bus) [138] (contrôleur supporté par

les familles Virtex-II et Virtex-II Pro), le contrôleur PLB [25] [148] qui permet l'interfaçage avec le bus PLB (Processor Local Bus) [138] (contrôleur supporté par les familles Spartan-6/XA, Virtex-4, Virtex-5, QVirtex-4, QrVirtex-4, et Virtex-6) et le contrôleur Axi [152] qui permet l'interfaçage avec un bus AXI [151] (contrôleur supporté par les familles Zynq-7000, Virtex-7, Kintex-7, Virtex-6 et Spartan-6).

Le mécanisme de reconfiguration utilisant ICAP s'appelle read-modify-write (RMW) [54]. Ce mécanisme permet au contrôleur de la reconfiguration de modifier le bitstream correspondant à un module dynamiquement reconfigurable à l'aide du module HwICAP. Un module HwICAP est composé d'un contrôleur de l'ICAP, d'une mémoire BRAM et du coeur de l'ICAP. Ce module permet au contrôleur de la reconfiguration d'accéder à l'ICAP pour lire et écrire des données de configuration dans la mémoire de configuration. Ce contrôleur peut être commandé par le processeur du système. Le processus de reconfiguration commence quand le processeur détermine le bitstream partiel à charger et son emplacement dans la mémoire externe. Le processeur envoie ensuite les trames une par une au contrôleur de l'ICAP. Celui-ci demande à l'ICAP de lire les trames correspondantes de la mémoire de configuration une par une. Les trames envoyées par l'ICAP sont stockées dans une mémoire BRAM du contrôleur. Cette mémoire a suffisamment de capacité pour stocker au moins une trame de configuration [55]. Le contenu de chaque trame est fusionné avec celui de la trame correspondante envoyée par le processeur. L'ICAP doit donc réécrire cette trame de la BRAM vers la mémoire de configuration. C'est le mécanisme read-modify-write. Quand une trame est réécrite dans la mémoire de configuration, les composants logiques correspondants aux données non modifiées de la trame continuent leurs opérations sont aucun changement [142].

2.5.2 Application de la RDP pour différents types de reconfigurations architecturales

La reconfiguration dynamique partielle peut affecter plusieurs composants du système (élément de traitement, mémoire, réseau d'interconnexion, etc.).

Dans [83], un outil pour la reconfiguration dynamique des BRAMs a été présenté. Cet outil permet de remplacer le contenu des BRAMs qui sont utilisées comme des mémoires locales pour les processeurs hardcore ou softcore sur FPGA, conduisant donc au changement du code exécuté ou des données traitées par le processeur, sans affecter le fonctionnement du reste du système. Cet outil permet de découpler les bitstreams en deux types : un pour configurer le contenu des BRAMs, et un autre pour la configuration du reste de la logique du FPGA. De cette façon, soit la logique change pour effectuer des calculs différents sur le contenu des BRAMs, soit le contenu des BRAMs change permettant à la logique d'effectuer les mêmes calculs sur des données différentes. La reconfiguration dynamique des mémoires peut résoudre le problème d'insuffisance de ressources matérielles. Par exemple, si le FPGA n'a pas d'espace suffisant pour implémenter les modules matériels nécessaires pour une fonctionnalité donnée du système, il vaut mieux charger un sous-ensemble des tâches sur le processeur au lieu d'attendre qu'un module matériel se libère pour le remplacer avec le module qui manque, ce qui permet une meilleure performance du système.

Dans [57], les processeurs dynamiquement reconfigurables ont été utilisés. L'archi-

tecture de ces processeurs RISP (Run-time Reconfigurable Instruction Set Processors) contient des modules matériels reconfigurables dans une ou plusieurs de leurs unités fonctionnelles. La reconfiguration de cette partie matérielle de ces processeurs permet d'adapter leurs jeux d'instructions selon les exigences de l'application durant l'exécution. En tirant profit du parallélisme offert par le matériel, la performance de ces processeurs est plus importante que celle des processeurs traditionnels. La difficulté de la conception de ce type de processeurs réside dans la conception de la partie reconfigurable du processeur ainsi que de la communication entre cette partie et le reste des modules du processeur.

Le type de reconfiguration le plus utilisé dans le domaine des FPGAs est celui basé sur des coprocesseurs ou accélérateurs matériels reconfigurables. Dans [131], un coprocesseur reconfigurable attaché au processeur softcore MicroBlaze à travers le bus FSL [139] a été utilisé. Ce bus permet de transférer les données entre les registres du MicroBlaze et le coprocesseur. Dans [20], un accélérateur matériel reconfigurable a été utilisé avec le MicroBlaze. Cet accélérateur est connecté à travers le bus standard OPB de CoreConnect [138], auquel sont connectés d'autres éléments tels que le contrôleur de la mémoire et d'autres périphériques. Puisque le transfert entre l'accélérateur et le processeur se fait à travers le bus OPB, ceci prend plus de temps que celui entre un processeur et un coprocesseur connectés directement par FSL. Cependant, la communication avec FSL nécessite des instructions spécifiques en comparaison avec la communication à travers un bus OPB qui se fait avec des instructions de lecture et écriture régulières. Dans [26] [133], les accélérateurs reconfigurables ont été utilisés avec le bus PLB [138] et AXI [151] respectivement. L'utilisation des coprocesseurs et des accélérateurs matériels reconfigurables permet d'économiser l'espace sur le FPGA en comparaison avec des solutions qui implémentent toute une application avec des modules statiques.

Dans [104], la reconfiguration dynamique du routage a été traitée. Puisque les lignes de routage n'ont pas la même performance, ni la même consommation d'énergie, il est très intéressant d'adapter le routage du système, au cours de l'exécution, aux contraintes de performance et de consommation. Le placement des IPs d'un système peut être raffiné selon les exigences de l'état interne du système et de la qualité de service requise pour l'application exécutée. Le placement dynamique permet de raccourcir les chemins de données (data-paths) pour améliorer la performance. Les limites des batteries peuvent forcer le système à modifier la topologie de ces IPs en prenant en compte une liste de priorités permettant de placer les IPs les plus prioritaires dans les positions les plus efficaces.

Dans [18], les auteurs proposent une reconfiguration dynamique de la communication au niveau paquet. Ce niveau de reconfiguration est utilisé pour les systèmes adoptant les réseaux sur puce (NoC) pour la communication. Dans ces systèmes, des modules de calcul peuvent être placés durant l'exécution, et les réseaux sur puce doivent s'adapter à ce changement pour assurer la communication entre les modules ajoutés et ceux qui sont initialement connectés. L'avantage de la communication basée sur les paquets est que l'altération du réseau ne va pas empêcher la communication, puisque les paquets peuvent toujours trouver leur chemin dans un réseau fortement connecté. Lorsqu'un module est placé dynamiquement sur le FPGA, il couvre un sous-ensemble des routeurs du NoC. A la différence d'un NoC statique, où chaque routeur a ses quatre

routeurs voisins toujours actifs, le NoC dynamique désactive les routeurs couverts pour le module placé durant l'exécution, ce qui nécessite une modification dans l'algorithme du routage. Pour ceci, un routeur doit vérifier si le routeur à qui il veut envoyer un paquet est activé. Si ce n'est pas le cas, le paquet sera envoyé dans la direction perpendiculaire à celle choisie précédemment. Dans [71], une solution pour la suppression des modules du réseau NoC a été proposée. C'est le cas d'un processeur et son noeud de routage qui peuvent être désactivés ou supprimés après que le processeur finisse son exécution, afin de réduire la consommation d'énergie.

Dans [105], les auteurs utilisent la RDP pour l'adaptation dynamique de la fréquence d'horloge aux contraintes de consommation d'énergie, en reconfigurant les DCMs (Digital Clock Manager) dans un FPGA Xilinx afin d'avoir deux fréquences différentes utilisées respectivement dans les états actif et inactif du processeur.

La méthodologie de conception de contrôle proposée dans ce travail convient pour ces différents types de reconfiguration puisqu'elle n'est pas dépendante des régions à contrôler. La validation de cette méthodologie a été basée sur son application à des accélérateurs matériels reconfigurables, qui est le type de reconfiguration le plus utilisé dans le domaine des systèmes reconfigurables sur FPGA et le plus simple à mettre en oeuvre.

2.6 Flot de conception des systèmes sur FPGA

2.6.1 Le flot de conception des systèmes statiques

Le flot de conception d'un système sur FPGA est présenté ici à travers en prenant comme exemple le flot proposé par les outils Xilinx, puisque la validation de notre méthodologie de conception de contrôle sera faite par une implémentation sur un FPGA de Xilinx. Ce flot est illustré dans la figure 2.6. Les vues logicielle, matérielle et simulation du flot de conception sont représentées de gauche à droite dans la figure. Un système implémenté sur FPGA est composé d'une partie logicielle et d'une partie matérielle.

2.6.1.1 La partie logicielle

Le fichier MSS (Microprocessor Software Specification) contient une "déclaration" des librairies, des pilotes et éventuellement des OS utilisés pour les composants matériels du système. A l'aide de l'outil LibGen les bibliothèques logicielles sont générées. L'application à exécuter par le microprocesseur est généralement écrite en langage assembleur ou C/C++. Le résultat de la compilation est un fichier ELF.

2.6.1.2 La partie matérielle

La partie matérielle est décrite en utilisant soit des langages de description matérielle (HDL) tels que VHDL et Verilog, soit une description schématique. Comme le montre la figure 2.6, pour générer la partie matérielle du système, les outils de Xilinx utilisent un fichier MHS (Microprocessor Hardware Specification) contenant entre autres une

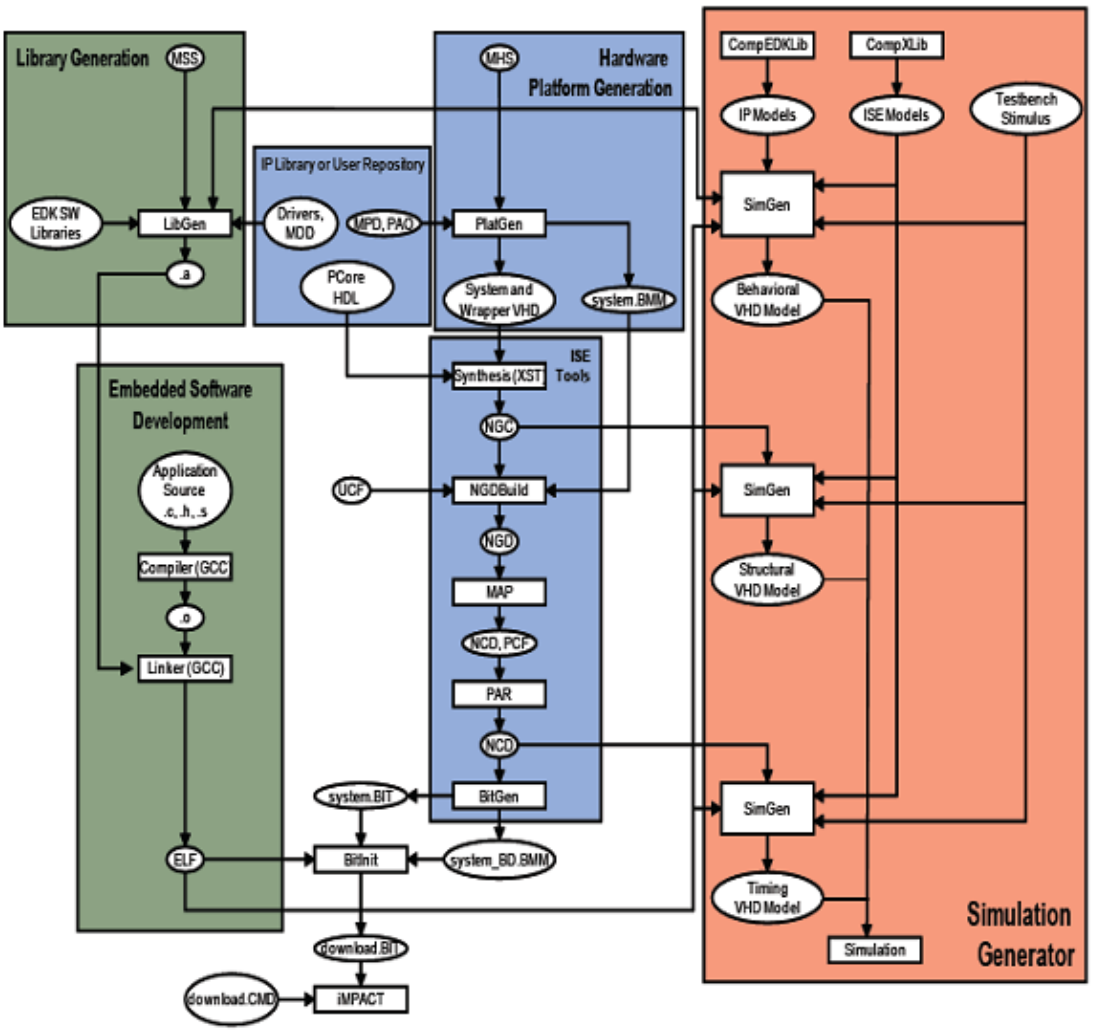


FIG. 2.6 – Flot de conception d'un système sur FPGA de Xilinx

"déclaration" des composants à instancier dans le système. En cherchant les IPs correspondantes dans la librairie d'IPs, la partie HDL du système est générée à l'aide de l'outil PlatGen. Cet outil génère aussi une description des BRAMs qui seront utilisées pour les microprocesseurs (system.BMM). L'étape de synthèse permet d'obtenir des netlists (fichiers NGC) qui sont une description au niveau portes logiques du design. L'étape de traduction (par l'outil NGDBuild) permet de faire la fusion entre les fichiers NGC, le fichier BMM (Block Memory Map), et les fichiers de contraintes (UCF) pour en construire un fichier design (NGD). L'étape de mapping permet de mapper la logique décrite dans les netlists sur les composants du FPGA (CLBs, I/Os, etc.). L'étape de placement et routage (PAR) permet de positionner les différents éléments du système implémenté sur FPGA et d'assurer leur interconnexion. Cette étape est réalisée en utilisant un outil propriétaire de placement et routage. Ensuite, le fichier de configuration (bitstream) est

généré.

2.6.1.3 L'association entre logiciel et matériel

A l'aide de l'outil BitInit, le bitstream correspondant à la partie matérielle du système est fusionné avec le fichier exécutable du logiciel, ce qui permettra de décrire le code et les données qui seront stockés dans les BRAMs pour exécuter l'application. Le résultat de cette fusion est le fichier bitstream download.bit qui peut être chargé par la suite sur le FPGA à partir du PC en utilisant l'outil iMPACT. Ce bitstream peut être également stocké dans une mémoire externe du FPGA ou un compact flash pour qu'il soit chargé automatiquement dès la mise sous tension du FPGA.

2.6.1.4 La simulation

Au cours de la conception du système, la simulation peut se faire à différents niveaux [144]. La simulation comportementale peut se faire avant la synthèse. Cette simulation permet de vérifier si le système fonctionne comme prévu. La simulation structurelle est appliquée après la synthèse. La différence entre le modèle de simulation structurelle et le modèle de simulation comportementale est que ce dernier utilise une abstraction des composants du système. Il contient des opérations à haut-niveau telles qu'un opérateur d'addition 4 bits, au lieu d'un additionneur Xilinx pour le modèle structurel. La simulation structurelle prend plus de temps que la simulation comportementale puisqu'elle intègre plus de détails. Pour cette raison, il est conseillé de commencer par une simulation comportementale afin de détecter le plus d'erreurs le plus tôt possible dans le flot de conception. La simulation temporelle est faite après le placement et routage. Elle permet de décrire le comportement du système sur le vrai circuit et de donner en plus des informations temporelles. Par exemple, pour un système synchrone, cette simulation permet de donner le temps t entre le front montant de l'horloge et le changement de la valeur d'un signal qui y est sensible. Ce temps semble être nul dans les simulations comportementales et structurelles qui ne prennent pas en compte les contraintes temporelles du système. Une telle information (le temps t par exemple), permet de déterminer si le système peut fonctionner avec une fréquence f donnée (vérifier que $t < 1/f$). Après la simulation temporelle, le bitstream du système peut être généré afin de le charger sur FPGA.

2.6.2 La conception des systèmes partiellement reconfigurables

Dans cette section, l'évolution technologique de la reconfiguration partielle pour les FPGAs de Xilinx est présentée avant de décrire brièvement le plus récent flot de conception lié à la reconfiguration dynamique. Plus de détails sur ce flot seront présentés dans le chapitre 6 dans la partie de l'implémentation physique de notre modèle de contrôle.

2.6.2.1 Reconfiguration partielle basée-différence

Ce type de reconfiguration convient pour de petits changements. Il est basé sur la comparaison de deux designs (le design avant et après la reconfiguration). Il détermine les trames différentes et crée un bitstream partiel qui modifie seulement les trames qui sont différentes, ce qui donne un temps de reconfiguration réduit. Les modifications peuvent affecter les standards I/O, le contenu de mémoires ou la configuration des LUTs. L'inconvénient de cette méthode est que le concepteur doit manipuler des éditeurs de bas niveau tels que FPGA Editor, pour effectuer ce genre de modification, ce qui nécessite une grande connaissance de l'architecture du FPGA. Cette méthode ne convient pas pour les designs complexes qui nécessitent de plus grandes régions reconfigurables.

2.6.2.2 Reconfiguration partielle basée-module

Cette méthode permet de diviser le design en modules. Ces modules peuvent être statiques (chargés seulement lors de la mise sous tension) ou dynamiquement reconfigurables. Cette division permet aux concepteurs de travailler indépendamment sur différents modules puis de fusionner ces modules pour obtenir un système complet. La séparation permet également de modifier un module indépendamment des autres lors de la reconfiguration. Pour ceci, un bitstream initial est nécessaire pour configurer tout le FPGA, et des bitstreams partiels sont utilisés pour configurer chaque région reconfigurable. Pour remplacer un module reconfigurable par un autre, il faut qu'ils aient la même interface. L'inconvénient de cette méthode est qu'un module reconfigurable doit occuper toute la hauteur du FPGA, ce qui constitue une perte de ressources surtout pour les FPGAs de grande taille.

2.6.2.3 Le flot Early Access Partial Reconfiguration

En 2006, Xilinx a introduit un flot de conception qui s'appelle Early Access Partial Reconfiguration (EAPR) et qui permet d'utiliser des modules reconfigurables à 2 dimensions. Un module reconfigurable à 2 dimensions est un module qui peut prendre une forme rectangulaire quelconque, à la différence des modules à une dimension qui occupaient toute la hauteur du FPGA avec la méthodologie modulaire de la reconfiguration partielle. L'EAPR permet aussi aux fils statiques de passer directement à travers les modules reconfigurables sans l'obligation d'utiliser les bus macros [3], ce qui permet d'améliorer la performance et simplifier la conception. Selon Xilinx, une région reconfigurable s'appelle PRR (Partial Reconfigurable Region). Une PRR peut avoir plusieurs implémentations possibles ou PRMs (Partial Reconfigurable Modules). Tous les PRMs d'une PRR ont la même interface externe pour faciliter la compatibilité. Tous les PRMs d'une PRR doivent être prédéterminés. Au début, un bitstream initial est chargé sur le FPGA. Ce bitstream contient la partie statique et les PRMs initiaux des PRRs. Ensuite,

[3] Pour la connexion entre un module statique et un module reconfigurable, on utilise des modules spéciaux nommés bus macros [143]. Les bus macros permettent aux broches des modules reconfigurables d'être compatibles avec la partie statique. Par conséquent, toutes les connexions entre les modules reconfigurables et la partie statique doivent passer à travers les bus macros, à l'exception des signaux globaux (signaux d'horloges, d'alimentation, etc.).

au cours d'une reconfiguration partielle, le contrôleur charge un bitstream partiel correspondant à un autre PRM de la même PRR. A l'aide du mécanisme Read-Modify-Write, les trames différentes entre les deux PRMs sont modifiées est réécrites dans la mémoire de configuration.

2.6.2.4 La reconfiguration partielle basée-partition (Partition-based Partial Reconfiguration)

L'apport du nouveau flot de conception de RDP de Xilinx se voit notamment au niveau de la phase de placement du système. Dans le nouveau flot, on ne parle plus de bus macros, mais de broches de partition (Partition Pins). Ces broches sont crées automatiquement pour les ports des partitions reconfigurables, alors que les bus macros se plaçaient manuellement sur le système en utilisant l'outil de placement PlanAhead de Xilinx. Dans le nouveau flot, les termes RP (Reconfigurable Partition) et RM (Reconfigurable Module) ont remplacé les termes PRR et PRM d'EAPR. Dans ce mémoire, nous avons utilisé ce flot de conception de systèmes reconfigurables pour la validation du modèle de contrôle proposé. Cependant, nous utilisons les anciens termes (PRR et PRM) qui sont les plus utilisés dans les travaux traitant la RDP.

L'implémentation d'un système partiellement reconfigurable sur FPGA est similaire à l'implémentation de plusieurs systèmes statiques qui partagent des parties logiques. Les partitions sont utilisées pour s'assurer que cette partie commune est identique pour tous ces systèmes. Ce concept est illustré dans la figure 2.7 [43]. Cette figure donne l'exemple d'un système ayant une seule région reconfigurable qui a N implémentations différentes (N modules reconfigurables). Le flot de conception de ce système suit les étapes suivantes :

1) Ecriture en HDL du module TOP décrivant tout le système avec la déclaration des instances des modules statiques et d'une instance des modules reconfigurables parce qu'ils seront implémentés dans la même région reconfigurable dans l'exemple de la figure 2.7.
2) Ecriture en HDL des modules statiques et des modules reconfigurables. Ces derniers doivent avoir le même nom puisqu'ils sont représentés par un seul composant dans le fichier TOP.
3) La synthèse des fichiers HDL pour le TOP, les modules statiques et les modules reconfigurables. Ces derniers ayant le même nom, doivent être synthétisés dans des projets Xilinx séparés.
4) L'implémentation du système est lancée N fois en prenant en compte à chaque fois des contraintes statiques, et les contraintes des modules reconfigurables. En d'autres termes, à chaque fois, on lance les phases NGDBuid (Translate), MAP et PAR pour un système constitué des netlists du TOP, des modules statiques et d'un module reconfigurable, avec les contraintes statiques et les contraintes du module reconfigurable comme le montre la figure 2.7. Pour éviter la réexécution de l'implémentation sur les parties qui ne sont pas modifiées entre les représentations du systèmes, ces parties sont copiées à partir de la première implémentation. Cette copie est utilisée pour le reste des implémentations comme le montre la figure 2.7. Pour générer les

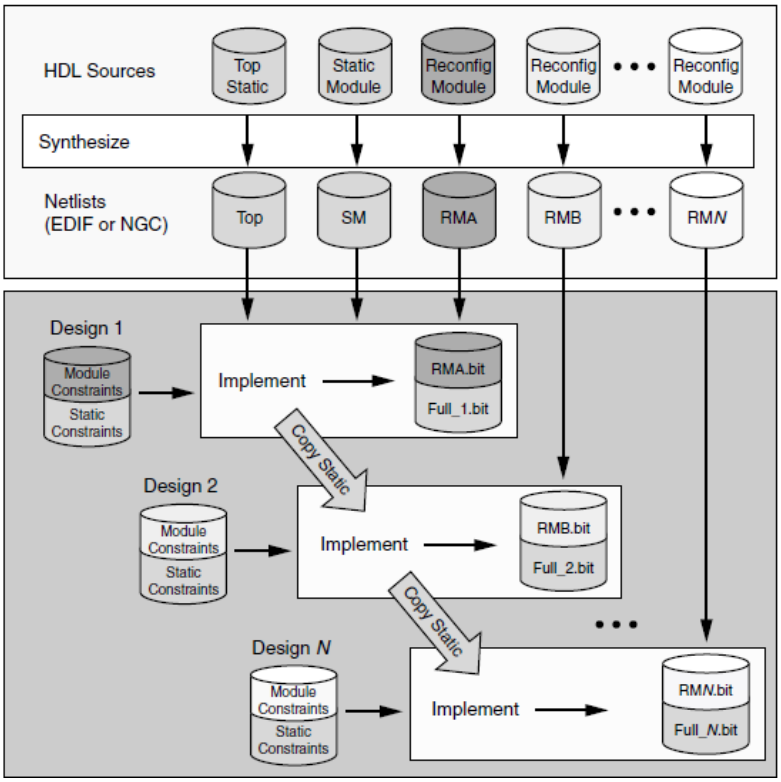


FIG. 2.7 – Flot de conception d'un système reconfigurable sur un FPGA de Xilinx

autres implémentations du système, il suffit de faire la fusion entre cette copie et les différentes implémentations des modules reconfigurables. Au cours de cette phase, la compatibilité entre celles-là et la copie de la partie statique est vérifiée afin de garantir le bon déroulement de la reconfiguration partielle au cours de l'exécution.

5) A partir de chaque implémentation, deux bitstreams sont générés : le bitstream total et le bitstream partiel. Au début de l'exécution du système, un bitstream total est chargé. Pour pouvoir changer la configuration de la région reconfigurable au cours de l'exécution, les bitstreams partiels seront chargés dans cette région l'un après l'autre. Pour cette raison, il faut choisir le bitstream total à charger initialement dans le FPGA, ce qui détermine la configuration initiale de la région reconfigurable. Par exemple, si on choisit de charger le bitstream Full_2.bit, la configuration de la région reconfigurable sera équivalente à un chargement du bitstream RMB.bit dans cette région.
6) Le bitstream total choisi est fusionné avec le fichier exécutable (le fichier elf) de la partie logicielle du système (si elle existe) comme on a montré dans la figure 2.6. Le fichier binaire qui en résulte peut être chargé dans le FPGA directement de l'ordinateur en utilisant le logiciel iMPACT ou en le sauvegardant dans une mémoire externe du FPGA ou un System ACE (utilisant une Compact Flash). La mémoire externe ou le compact Flash doit donc contenir en plus de la configuration totale du système, tous les bitstreams partiels à charger durant l'exécution. Dans ce cas, à la mise sous

tension, le bitstream total et le programme du processeur (s'il existe) sont chargés dans le FPGA et l'exécution de l'application commence.

2.7 Conclusion

La reconfiguration dynamique partielle est un mécanisme puissant qui permet de reconfigurer des portions du FPGA sans affecter le fonctionnement du reste du système. Ce mécanisme offre une grande flexibilité qui permet de n'adapter que la partie concernée du système suite à des changements durant l'exécution qui peuvent être liés à l'environnement, aux entrées des utilisateurs ou des contraintes internes du système (performance, consommation, contraintes liées aux données, etc). La prise en compte de ces différentes sources d'adaptation pourrait engendrer une grande complexité de conception pour des systèmes de grande taille. L'objectif de ce travail est de proposer une méthodologie de conception du contrôle de l'adaptation dynamique qui facilite le travail des concepteurs et améliore leur productivité. Dans le chapitre suivant, différentes approches de contrôle pour les systèmes reconfigurables sont discutées. La contribution de notre approche par rapport à celles-ci est ensuite présentée.

Chapitre 3

Etat de l'art du contrôle de l'adaptation dynamique des systèmes reconfigurables

3.1 Introduction

Devant l'évolution technologique continue des systèmes embarqués et la complexité croissante des applications qu'ils ciblent, la conception de tels systèmes est devenue une tâche complexe et coûteuse en temps à cause du fait que les outils de conception n'évoluent pas au même rythme que la technologie matérielle. La reconfigurabilité de certains systèmes augmente encore plus cette complexité avec des tâches additionnelles de conception liées principalement au contrôle de l'adaptation dynamique. Dans ce contexte, une méthodologie efficace de conception de contrôle est requise afin d'assurer les facteurs flexibilité, réutilisabilité, scalabilité et automatisation permettant de faciliter le travail des concepteurs et d'améliorer leur productivité. Les systèmes conçus en suivant une telle méthodologie doivent aussi être implémentés de façon efficace afin de minimiser l'impact du contrôle sur la performance globale des systèmes.

Dans ce contexte, différentes approches de contrôle ont été proposées pour réaliser ces objectifs, en traitant différents aspects tels que la distribution du contrôle, la coordination entre contrôleurs distribués, l'utilisation du formalisme pour le contrôle et l'automatisation de la génération du contrôle. Dans ce chapitre, nous commençons par présenter les différentes architectures de contrôle pour les systèmes reconfigurables. Différents mécanismes de coordination entre contrôleurs sont ensuite présentés. L'utilisation des formalismes pour la modélisation du contrôle des systèmes reconfigurables est ensuite étudiée à travers différents travaux dans ce domaine. Différentes approches d'automatisation dans la conception du contrôle sont ensuite présentées. Le chapitre se termine par une synthèse de tous ces travaux.

3.2 Les architectures de contrôle

Dans cette section, différentes architectures de contrôle de l'adaptation dynamique des systèmes reconfigurables sont présentées. D'un point de vue prise de décision de reconfiguration, ces architectures peuvent être classées en des architectures centralisées et décentralisées. C'est la classification utilisée dans cette section.

3.2.1 Les architectures centralisées

Le contrôle centralisé des systèmes dynamiquement reconfigurables sur FPGA a été étudié par plusieurs travaux. Ces travaux peuvent être classés selon le type de l'implémentation du contrôle. Cette implémentation peut être standalone ou par un système d'exploitation (OS). Dans cette section, nous présentons certains travaux dans ces deux catégories.

3.2.1.1 Implémentation standalone

La plupart des travaux sur les systèmes reconfigurables sur FPGA ont proposé une implémentation logicielle du contrôle. Cette implémentation convient bien pour des problèmes de contrôle simples où on n'a pas besoin de services avancés d'un système d'exploitation. L'implémentation logicielle du contrôle des systèmes reconfigurables sur

FPGA peut se faire par un processeur hardcore [61] ou softcore [17]. Dans le cas des systèmes supportant la reconfiguration dynamique partielle et interne (voir section 2.4), le processeur joue deux rôles : 1) la gestion de la commutation des configurations partielles des régions reconfigurables (prise de décision de reconfiguration) et 2) la communication avec le module qui charge les bitstreams partiels (l'ICAP pour les FPGAs Xilinx). La gestion de la commutation entre bitstreams partiels est généralement basée sur des sémantiques de contrôle telles que les machines d'états [58] et les réseaux de Petri [89]. Différentes situations peuvent déclencher la reconfiguration dynamique partielle. Par exemple, dans un système à ressources limitées, la reconfiguration peut être gérée par le graphe des tâches [131] [51] [132]. Dans ce cas, à chaque fois qu'on a besoin d'une tâche qui n'est pas chargée dans le système, le processeur lance une reconfiguration pour la charger à la place d'une autre tâche qui a terminé son exécution. La reconfiguration peut être aussi déclenchée par d'autres événements tels que le changement de l'environnement, qui peut nécessiter par exemple l'utilisation d'un filtre différent de détection/suivi d'objets pour un système d'aide à la conduite [48] [24].

Le type du processeur (hardcore ou softcore) utilisé a une influence sur la performance de la reconfiguration. L'avantage d'un processeur hardcore est que, puisqu'il est implémenté directement sur la puce au lieu d'être implémenté par des slices, il est fait d'une façon personnalisée et optimisée ce qui le rend plus performant qu'un processeur softcore implémenté sur du matériel générique (les slices). L'utilisation d'un processeur hardcore pour lancer la reconfiguration permet donc de diminuer le temps de reconfiguration. Dans [82], la performance du processeur hardcore PowerPC et le processeur softcore MicroBlaze en termes de temps de reconfiguration pour un Virtex-4 a étudiée. L'utilisation de ces deux processeurs pour 5 systèmes différents a montré que le processeur PowerPC peut augmenter la performance de reconfiguration jusqu'à 85% par rapport au MicroBlaze.

Dans [21], un autre type d'implémentation est proposée dans lequel le processeur gère la commutation des configurations partielles alors que la communication avec l'ICAP se fait par un contrôleur matériel. Le processus de reconfiguration se passe comme suit : le processeur envoie des commandes au contrôleur lui indiquant le module qu'il veut utiliser, et contrôleur matériel prend en charge la sélection du bitstream partiel correspondant au module demandé et la communication avec l'ICAP pour charger ce bitstream. L'utilisation du contrôleur matériel a permis d'accélérer la reconfiguration par rapport à une solution entièrement logicielle (un temps de reconfiguration 10 fois plus petit pour un Virtex-II). Cependant, le contrôleur matériel proposé ne présente pas d'interface standard mais une interface dédiée à la communication directe avec le processeur qui n'est pas non plus un processeur standard mais un processeur dont le jeu d'instructions a été étendu pour supporter l'utilisation de coprocesseurs reconfigurables. La gestion matérielle du lancement de la reconfiguration a été également proposée dans [63] permettant une grande réduction du temps de reconfiguration (18,2ms pour une région de 7840 LUTs et flip-flops LUTs, 112 DSP et 28 BRAMs pour un Virtex-5) cependant la gestion de la commutation des configurations partielles n'a pas été étudiée.

3.2.1.2 Implémentation par OS

Que ce soit centralisé ou distribué, l'utilisation d'un système d'exploitation pour le contrôle d'un système reconfigurable a beaucoup d'avantage tels que l'optimisation du partage de ressources entre différentes applications en rendant possible l'exploitation des ressources reconfigurables par différents processus à travers un système multitâche. La différence qu'ont les OSs pour les systèmes reconfigurables sur FPGA par rapport aux systèmes d'exploitation traditionnels est qu'ils doivent gérer l'hétérogénéité des tâches (tâches logicielles et matérielles) à ordonnancer et à faire communiquer. Ces OSs doivent offrir une abstraction de cette hétérogénéité et des services de reconfiguration, du côté de l'application [137]. Dans [117], la séparation entre les codes des applications de l'utilisateur et les processus système pour la reconfiguration a été étudiée. Cela permet de faciliter la réutilisation et la portabilité des codes des applications utilisateur puisque les appels de reconfiguration à haut-niveau qu'ils font ne contiennent aucune information sur l'implémentation à bas niveau du système laissant la gestion de ces détails au système d'exploitation. Dans [40], un OS temps-réel a était étendu pour permettre aux différentes implémentations logicielles et matérielles d'une tâche d'être exécutées d'une manière transparente en offrant aux tâches matérielles des services de communication et de synchronisation similaires à ceux offerts par les systèmes d'exploitation traditionnels aux tâches logicielles. Cela permet la simplification de l'accès à la technologie en faisant une abstraction sur le type des tâches (logicielles ou matérielles) et en virtualisant la communication entre elles.

La gestion de la reconfiguration par OS a été étudiée tant pour la reconfiguration logicielle (changements dans le code exécuté par un processeur) que pour la reconfiguration matérielle. Dans [53], le contrôle de la reconfiguration logicielle par un système d'exploitation a été utilisé pour un système multi-processeurs. Le contrôle est implémenté par un seul processeur. Celui-ci gère à travers le système d'exploitation qu'il implémente des services tels que l'ordonnancement, le transfert des messages entre les autres processeurs et la reconfiguration logicielle pour ces processeurs. Le processeur de contrôle observe les signaux nécessitant le chargement d'une tâche donnée de l'application et il cherche un processeur inactif parmi les processeurs du système pour charger et exécuter cette tâche. S'il ne trouve pas de processeur inactif, il émet une requête aux processeurs et attend à ce que l'un d'entre eux réponde indiquant que la fonction qu'il était en train d'exécuter n'est plus active. Dans ce cas, la tâche peut être chargée à travers une reconfiguration logicielle du code exécuté par l'un des processeur. L'utilisation d'un système d'exploitation pour la gestion de la reconfiguration matérielle nécessite des services plus complexes afin de gérer efficacement la dynamicité. La plupart des travaux sur les systèmes d'exploitation pour les systèmes reconfigurables sur FPGA ont traité principalement les services d'ordonnancement des tâches logicielles et matérielles et la communication entre ces tâches ainsi que l'allocation et le placement des tâches matérielles reconfigurables. Dans ce contexte, les algorithmes d'allocation implémentés visent à minimiser le nombre de refus de chargement des modules requis par les applications utilisateur à cause de l'absence d'espace disponible pour ces modules [137] [117]. Dans ce contexte, le système d'exploitation peut par exemple appliquer une politique d'allocation cherchant, pour chaque module matériel requis durant l'exécution, le nombre

minimal des slots consécutifs libres pour le charger afin de minimiser la fragmentation du FPGA [117]. Certains travaux ont aussi proposé des services d'OS pour la génération dynamique de bitstreams [137] [117] pour s'adapter à la dynamicité de la politique d'allocation.

Une approche différente a été proposée dans le projet FOSFOR [87] avec l'utilisation de deux types d'OS pour un même système. L'OS qui gère les tâches logicielles est implémenté en logiciel par les processeurs du système et celui qui gère les tâches matérielles est implémenté en matériel à travers des services distribués placés à côté des régions reconfigurables contrôlées. L'utilisation de l'OS matériel permet de réduire le coût de l'utilisation de l'OS sur la performance globale du système et d'assurer une meilleure synchronisation avec les tâches matérielles contrôlées. Pour la communication entre les tâches logicielles et matérielles, un intergiciel (middleware) a été développé. La communication entre les tâches matérielles est basé sur un réseau-sur-puce personnalisé pour ce type de communication [32]. Cette approche permet donc d'assurer une grande abstraction des tâches en réduisant le coût de l'utilisation d'un OS en termes de temps d'exécution. D'un point de vue prise de décision de reconfiguration des modules matériels, la décision est gérée par l'ordonnanceur du système d'exploitation matériel [88]. De ce fait, la distribution dans cette approche concerne plus les services de communication et d'exécution de la reconfiguration alors que la prise de décision de reconfiguration est centralisée par l'ordonnanceur.

Les décisions de reconfiguration gérées par les systèmes d'exploitation dans les travaux présentés ci-dessus sont pour la plupart orientées par l'ordonnancement. L'ordonnanceur dans ce cas, cherche une région vide pour le chargement de la nouvelle tâche à exécuter ou la charge à la place d'une tâche qui a fini de s'exécuter dans un module reconfigurable. L'adaptation des implémentations des tâches à des changements liés à l'environnement ou aux préférences des utilisateurs n'a pas été largement étudiée par les travaux sur le contrôle des systèmes FPGA en utilisant des OSs [40].

L'utilisation des systèmes d'exploitation pour le contrôle des systèmes reconfigurables sur FPGA permet d'offrir une grande flexibilité par rapport à une solution standalone à travers des services tels que l'allocation dynamique des tâches aux modules reconfigurables et la virtualisation de la communication entre tâches matérielles et logicielles. Cependant, leur coût en termes de temps d'exécution peut être non négligeable surtout s'ils sont implémentés en logiciel[117] [40]. Afin de réduire ce coût, l'implémentation matérielle des services de l'OS a été proposée par certains travaux [81] [84] [30] [9] [87]. Cependant, de telles implémentations pourraient aussi avoir un coût non négligeable en termes de ressources utilisées.

3.2.2 Les architectures décentralisées

Que ce soit implémenté en standalone ou par un OS, l'utilisation d'un contrôleur centralisé pour l'adaptation dynamique des systèmes reconfigurables pourrait diminuer la flexibilité du contrôle et la possibilité de le réutiliser et de l'adapter à des systèmes différents. En effet, du fait de sa vue globale, l'implémentation du contrôleur centralisé devient dépendante du système ciblé, ce qui diminue les possibilités de réutilisation et de scalabilité. Une solution aux problèmes de conception liés au contrôle centralisé

peut être de décomposer le contrôle entre un ensemble de contrôleurs gérant chacun des problèmes de contrôle locaux. Cette solution favorise la réutilisation de chaque contrôleur et facilite ainsi la scalabilité du contrôle pour des systèmes de grande taille. De plus, la distribution du contrôle permet de mieux s'adapter à l'évolution constante des tailles des systèmes ciblés en évitant les problèmes de communication centralisée.

Le contrôle décentralisé des systèmes reconfigurables n'a été traité que par quelques travaux sur FPGA. Cependant, il est très utilisé dans d'autres domaines tels que les systèmes mécatroniques, les systèmes robotiques, etc. Certains travaux dans ces domaines sont présentés dans cette section à côté de ceux liés au domaine des FPGAs.

Il y deux catégories de de contrôle décentralisé : celui avec des contrôleurs non-communicants et celui avec des contrôleurs communicants. La première catégorie est utilisée pour des systèmes où une décision faite par un contrôleur n'a pas d'influence sur les autres ou sur les contraintes/objectifs globaux des systèmes. Un tel système peut par exemple être composé de machines autonomes et intelligentes où les différentes configurations de chacune d'entre elles n'ont aucune impact sur les configurations des autres [134]. La seconde catégorie de contrôle est utilisée quand les contrôleurs doivent échanger leurs décisions afin de respecter les contraintes/objectifs globaux du système. Dans ce cas, une coordination entre contrôleurs est nécessaire. Le contrôle décentralisé peut être complètement distribué ou semi-distribué (hiérarchique).

3.2.2.1 Le contrôle complètement décentralisé

Dans [124] [125], une approche basée sur le contrôle distribué matériel est utilisée. Dans cette approche, chaque région du FPGA est contrôlée par un contrôleur matériel. Le rôle de ce dernier est de contrôler les tâches exécutées par la région associée. Une reconfiguration est déclenchée par le contrôleur à chaque fois qu'une tâche gérée par la région contrôlée est requise. L'avantage de cette implémentation matérielle est qu'elle permet de réduire l'impact du contrôle en termes de temps d'exécution en favorisant l'exécution parallèle du contrôle et des autres tâches de l'application. Cependant, cette approche est limitée à des problèmes de contrôle locaux et indépendants. De ce fait, les possibilités de communication ou de coordination entre contrôleurs n'est pas traitée.

Dans le projet AETHER [106] [107], un modèle général d'entités reconfigurables sur FPGA a été proposé. Chaque entité est auto-adaptative et gère un ensemble de tâches : le calcul, l'observation, le contrôle et la communication avec les autres entités. Cette approche favorise la réutilisation du contrôle puisque chaque entité a sa propre partie de contrôle. Cependant, dans l'implémentation proposée pour cette approche, la prise de décision de reconfiguration est faite en centralisé à travers un OS et le problème de coordination entre contrôleurs n'a pas été traité.

Dans DodOrg [119], une approche de contrôle distribué a été proposée pour des systèmes de calcul organique. Les systèmes ciblés sont composés de cellules de calcul matérielles implémentées sur FPGA. Chaque cellule est auto-adaptative et fournit des services de calcul, de mémoire et d'entrée/sortie et contient aussi une unité de contrôle implémentée en matériel. Pour résoudre le problème de la disponibilité d'un seul port ICAP sur FPGA, cette approche propose une technique d'accès distribué à ce port afin d'accélérer le processus de reconfiguration par rapport à un accès centralisé. Cependant,

cette approche se concentrait plus sur l'accès distribué à l'ICAP sans donner de détails sur les composants de contrôle utilisés par chaque entité (observation, décision, etc).

Le contrôle distribué des FPGA n'a été traité que par quelques travaux parmi lesquels ceux cités ci-dessus. Cependant, ces travaux ne traitent pas l'aspect coordination entre les décisions de reconfiguration. Ils présentent généralement deux modèles. Le premier modèle est le cas où les configurations partielles sont indépendantes [124] [125] [119] et n'ont pas besoin de coordination. Le deuxième modèle est le cas où la prise de décision est faite en centralisé [106] [107] bien que les autres parties (observation, reconfiguration, etc) soient distribuées.

L'aspect coordination a été traité dans des domaines autres que les FPGAs tels que les MPSoC [108] [109], les réseaux sans fils [37], les systèmes multi-équipements [52] [158], etc. Deux principaux modèles de coordination entre contrôleurs ont été proposés. Le premier modèle se base sur la communication de chaque contrôleur avec tous les autres [108] [109] [37]. Pour arriver à une configuration globale satisfaisante pour tous les contrôleurs, le nombre de messages échangés peut être considérable pour un système de grande taille, ce qui pourrait avoir un impact non négligeable sur la performance du système de contrôle. Les contrôleurs de ce type doivent prendre en compte les informations (décisions/états) envoyées par tous les autres contrôleurs, ce qui peut engendrer une grande complexité de chaque contrôleur et donc une grande consommation en ressources. De plus, la vision globale de chaque contrôleur le rend dépendant du système, ce qui pourrait représenter un obstacle devant la réutilisation. Le deuxième modèle de coordination est un modèle où chaque contrôleur ne communique qu'avec ses voisins [52] [158], ce qui permet de diminuer le nombre de messages échangés entre contrôleurs ainsi que la complexité de leurs communications. Cependant, cela demande encore de chaque contrôleur d'avoir une vision sur les comportements des autres contrôleurs, ce qui pourrait les rendre dépendants du système et difficiles à réutiliser.

3.2.2.2 Le contrôle hiérarchique

Afin de minimiser le nombre de messages échangés entre contrôleurs avant d'arriver à une configuration globale satisfaisante et d'améliorer la réutilisation des contrôleurs, des approches de contrôle hiérarchique ont été proposées. Ces approches permettent de séparer les problèmes gérés par un système de contrôle en des problèmes locaux gérés par les contrôleurs du niveau inférieur et des problèmes globaux gérés par les contrôleurs du niveau supérieur, ce qui facilite la réutilisation des deux types de contrôleurs pour d'autres systèmes.

Dans [60], une approche de contrôle basée sur la coordination a été proposée pour un système électrique multi-agents. Chaque agent est responsable du contrôle de la reconfiguration d'un appareil du système. Selon ses contraintes locales (nombres de pièces à traiter par une machine, composants en panne, etc), chaque agent peut décider de reconfigurer l'appareil qu'il contrôle. Cette décision est envoyée à un agent de coordination, qui grâce à sa vision globale du système vérifie si la requête peut être satisfaite directement ou si elle nécessite la reconfiguration d'autres machines. Dans ce cas, les agents des machines a reconfigurer sont informés en leur envoyant les reconfigurations

nécessaires auxquelles ils peuvent répondre par acceptation ou par refus. L'objectif de cette approche est de minimiser le nombre de messages échangés entre agents par rapport à un système complètement distribué où chaque décision de reconfiguration est envoyée à tous les agents, ce qui nécessite une connaissance de toutes les configurations possibles du système et leurs priorités par tous les agents.

Dans le projet RaaR [40] [35], une autre approche intéressante de contrôle hiérarchique a été proposée. Elle a été implémentée sous-forme de tâches logicielles dans un système FPGA intégrant un système d'exploitation. Cette approche est basée sur une architecture à deux étages hiérarchiques. Le premier étage est un étage où un contrôleur de configuration local (LCM) est lié à chaque application exécutée par le système d'exploitation et permet de déterminer la liste de reconfigurations possibles, suite à des changements liés aux métriques considérées par l'application. Le deuxième étage contient un contrôleur de configuration global (GCM) qui prend en considération des métriques liées au système (temps d'exécution, consommation d'énergie, etc) pour sélectionner à partir des possibilités données par le LCM celle qui correspond le plus aux objectifs du système. Grâce à la séparation du contrôle entre LCM et GCM, un LCM peut être réutilisé pour une même application indépendamment des contraintes du système vu qu'il ne prend pas en compte les métriques du système. Le GCM peut être aussi facilement réutilisé, moyennant quelques modifications selon les applications implémentées, puisqu'il considère chaque application du système comme une liste de configurations possibles sans entrer dans ses détails. Cependant, l'utilisation d'un seul LCM pour gérer toute une application pourrait limiter la flexibilité et la réutilisabilité d'un LCM pour différentes applications, ce qui pourrait réduire la productivité des concepteurs de LCMs surtout pour les applications de grande taille. Pour assurer plus de flexibilité et de scalabilité, l'approche proposée dans [40] [35] pourrait être étendue pour traiter les possibilités de conflits entre les configurations choisies par les LCMs dans un système multi-applications. Dans ce cas, le GCM doit en plus de la gestion des métriques du système, gérer la coordination entre les LCMs.

Dans [33] [34], le contrôle hiérarchique a été utilisé dans le cadre de la négociation pour la répartition des tâches dans un système multi-robots. Le contrôle est assuré par des robots intelligents qui sont répartis en clusters avec des leaders de clusters communicants. Chaque leader gère la négociation entre les robots de son cluster. Les stratégies de négociation des robots d'un cluster dérive de l'objectif du leader. Les leaders négocient aussi entre eux en utilisant des stratégies dérivées de l'objectif global du système, ce qui permet d'optimiser cet objectif global et de résoudre le problème de sous-optimalité de la négociation dans un système complètement distribué. Cette approche permet donc de diviser les stratégies de contrôle entre les leaders et les autres robots, ce qui facilite la réutilisation et la scalabilité de ce modèle et diminue en même temps le nombre d'échanges nécessaires pour la négociation.

3.2.3 Synthèse

Devant la complexité croissante des systèmes reconfigurables, l'utilisation d'un contrôleur centralisé pour l'adaptation dynamique de ces systèmes peut ne pas être la meilleure solution en vue d'avoir un modèle de contrôle flexible, réutilisable et scalable.

En effet, la vue globale du contrôleur centralisé le rend dépendant du système ciblé, ce qui représente une grande rigidité de conception et un obstacle devant la réutilisabilité et la scalabilité. De plus, d'un point de vue implémentation, avoir un contrôleur centralisé pour des systèmes de grande taille peut aussi engendrer des goulets d'étranglement. Ce problème peut influencer énormément la performance globale des systèmes de grande taille tels les systèmes multi-FPGAs où la communication entre les FPGAs peut avoir un coût non négligeable. En outre, pour les systèmes supportant la reconfiguration parallèle tels que les systèmes multi-FPGAs et les FPGAs 3D, le contrôle centralisé n'est pas le plus adapté pour le lancement des reconfigurations à cause de son comportement séquentiel, ce qui peut affecter la performance du système.

Le contrôle décentralisé des systèmes reconfigurables offre plus de flexibilité par rapport au contrôle centralisé et permet d'éviter les problèmes de communication avec le contrôleur central. La distribution favorise aussi la réutilisation de chaque contrôleur et facilite ainsi la scalabilité du contrôle pour des systèmes de grande taille. Dans ce contexte, le concept d'entité auto-adaptative [106] [107] [119] en fournissant pour chaque entité reconfigurable des services d'observation, de décision et de reconfiguration est une approche très intéressante qui offre une grande flexibilité et favorise la réutilisation de ces entités.

Le contrôle décentralisé peut être effectué par des contrôleurs non communicants dans le cas où une décision faite par un contrôleur n'a pas d'influence sur les autres ou sur les contraintes/objectifs globaux des systèmes, ou par des contrôleurs communicants dans le cas où ces derniers doivent échanger leurs décisions afin de respecter les contraintes/objectifs globaux du système. Dans ce travail, nous intéressons à ce deuxième type de contrôle et son application pour les systèmes FPGA, puisque généralement dans ces systèmes les configurations partielles ne sont pas indépendantes et ont souvent des impacts sur les contraintes/objectifs globaux. Les contrôleurs distribués ont donc besoin d'une coordination pour garantir que leurs décisions respectent bien les contraintes du système.

Dans un système avec des contrôleurs communicants, le contrôle complètement distribué pourrait avoir des impacts non négligeables sur la performance vu le grand nombre d'échanges nécessaires entre contrôleurs pour les systèmes de grandes tailles. De plus, dans le cas où les problèmes de contrôle gérés par chaque contrôleurs sont différents, l'utilisation de ce type de contrôle pourrait engendrer une grande complexité de chaque contrôleur puisqu'il doit avoir une vision sur les problèmes de contrôle des autres contrôleurs afin de déterminer ses interactions avec ceux-ci, ce qui rend les contrôleurs dépendants du système et des contrôleurs avec lesquels ils communiquent réduisant ainsi ses possibilités de réutilisation et de scalabilité. Pour résoudre ces problèmes, un contrôle hiérarchique a l'avantage de réduire le nombre d'échanges nécessaires entre contrôleurs avant d'arriver à une décision satisfaisante pour tous. La séparation entre les problèmes de contrôle locaux et le problème de contrôle global à travers un modèle hiérarchique améliore aussi la réutilisation des contrôleurs.

Que ce soit dans un modèle de contrôle complètement distribué ou hiérarchique, la gestion de la coordination est un aspect critique dans la conception du contrôle. Dans la section suivante, nous traitons différentes approches de coordination pour la prise de décision de reconfiguration.

3.3 Les approches de coordination entre contrôleurs

Le problème de la coordination entre contrôleurs a été traité dans différents domaines tels que les systèmes multi-processeurs sur puce (MPSoC) [108] [109], les systèmes multi-agents pour les domaines robotiques et mécatroniques [52] [60] [3] [158] [157], etc. Dans cette section, nous classons les approches de coordination selon le fait qu'elles soient basées ou non sur la négociation, puisque ces deux types de coordination impliquent deux formes différentes d'échanges entre contrôleurs.

3.3.1 Coordination avec négociation

La coordination avec négociation dans les systèmes reconfigurables a généralement la forme suivante. Lorsqu'un contrôleur a besoin d'une reconfiguration et que cette reconfiguration demande des reconfigurations gérées par d'autres contrôleurs, une négociation est déclenchée afin d'arriver à une configuration globale satisfaisante. Cette situation satisfaisante peut visait l'optimisation ou juste le respect de certaines contraintes. La coordination entre contrôleurs en utilisant la négociation a été traitée tant par des approches de contrôle complètement distribué et que par des approches hiérarchiques.

Dans [60], une forme simple de négociation basée sur une approche multi-agents [59] a été utilisée pour un système mécatronique. Cette négociation est gérée à travers un coordinateur. Le contrôleur qui initie la négociation exprime au coordinateur son besoin de reconfiguration de l'appareil qu'il contrôle. Le coordinateur traduit ce besoin en des propositions de reconfiguration envoyées aux autres contrôleurs si nécessaire. Ces derniers peuvent accepter ou refuser selon leurs contraintes/objectifs locaux.

Dans [91], une négociation à plusieurs niveaux a été utilisée pour l'allocation des tâches dans un système multi-équipements. Dans cette approche, une négociation est lancée par l'un des équipements qui a besoin d'un ensemble de tâches données qui peuvent être exécutées par des combinaisons différentes d'équipements dans le système. Dans ce cas, les équipements correspondants à ces différentes combinaisons forment des consortiums afin de donner leurs offres à l'équipement qui a initié la négociation. Selon les différentes possibilités d'allocation, des sous-consortiums sont construits. Chaque consortium/sous-consortium est géré par un équipement intelligent qui est capable de formuler l'offre de son consortium à partir des offres des différents équipements dans celui-ci. En remontant l'information de chaque contrôleur de sous-consortium au contrôleur du consortium dont il fait parti, les différentes offres arrivent à l'équipement qui a lancé la négociation. Si une des offres satisfait le niveau de service requis, la négociation est terminée, sinon la négociation est relancée en augmentant la priorité des tâches négociées.

Dans [37], la négociation a été utilisée pour le contrôle de puissance dans les réseaux sans fil. La communication entre négociateurs est directe (contrôle complètement distribué). Elle vise à maximiser le nombre de noeuds atteignant le niveau de qualité de service (le rapport signal sur bruit) ciblé en négociant les puissances de transmission des noeuds. L'algorithme proposé donne la chance à tous les noeuds insatisfaits de négocier la puissance de transmission durant chaque tour de transmission. Chaque noeud peut être un "vendeur" qui réduit sa puissance de transmission en faveur d'autres

noeuds (réduit l'interférence et donc favoriser la qualité de service des autres noeuds) ou un "acheteur" qui demande la réduction de puissance de transmission au vendeur. La décision d'un noeud d'être un vendeur ou un acheteur est faite aléatoirement. Les offres des acheteurs contiennent le pourcentage de réduction de puissance demandé au vendeur et la récompense qu'il lui offre s'il accepte la réduction. Cette récompense est calculé à partir de son budget courant de l'acheteur. Chaque vendeur accepte l'offre si la récompense offerte par l'acheteur est supérieure à celle qu'il aurait donnée s'il était acheteur. En collectant les récompenses des acheteurs, un vendeur augmente la chance que son offre soit acceptée s'il est acheteur dans les tours suivants, ce qui permet de maximiser le nombre de noeuds ayant atteint le niveau de qualité de service ciblé.

3.3.2 Coordination sans négociation

La coordination sans négociation est une coordination qui n'est pas basée sur des offres et des acceptations, mais juste sur le partage d'informations entre contrôleurs. Ces informations sont traitées par chacun des contrôleurs selon une stratégie donnée afin d'orienter ses décisions de reconfiguration. Dans ce cas, le contrôleur peut collecter ces informations de tous les contrôleurs où seulement de ses voisins.

Dans [108], une coordination basée sur la théorie des jeux a été utilisée. Cette coordination est faite entre processeurs dans un MPSoC afin d'optimiser la température des processeurs en maintenant la synchronisation entre tâches, à travers un ajustement dynamique de la fréquence et de la tension (Dynamic Voltage and Frequency Scaling : DVFS). Dans [109], la même approche a été utilisée pour l'optimisation de la consommation de puissance en respectant les contraintes de latence. Dans un contrôle basé sur la théorie des jeux, chaque contrôleur est un joueur qui prend des décisions en essayant de maximiser son profit selon ses préférences en effectuant un ensemble d'actions en la présence d'autres joueurs [101]. Dans [108] [109], la forme normale non coopérative de la théorie des jeux a été utilisée. Dans cette forme, les joueurs agissent simultanément et sans la coopération des autres joueurs (les actions des joueurs ne suivent pas un accord entre eux). Un jeu répété est donc une séquence de cycles du jeu. A chaque cycle, chaque joueur choisit son action pour maximiser son profit en tenant en compte les actions effectuées par les autres joueurs dans le cycle précédent. L'action est choisie en assumant que les actions des autres joueurs ne changeront pas dans le cycle suivant. Ce jeu se termine lorsque l'équilibre (Nash Equilibrium) [90] est atteint, c'est le cas lorsque chaque joueur ne peut plus améliorer son profit. En appliquant la théorie des jeux au problème d'optimisation par DVFS, les joueurs sont les processeurs et leurs actions sont l'ajustement de leurs fréquences. A chaque cycle du jeu, chaque processeur reçoit les valeurs de fréquence des autres processeurs et leur envoie la sienne. Chaque processeur détient une formule décrivant la fonction à optimiser en agissant sur la fréquence. Dans un problème d'optimisation multi-objectifs, cette fonction prend en compte plusieurs critères (consommation, température, performance, etc). Elle prend en compte aussi les fréquences des autres processeurs. Le problème de contrôle est donc géré par la même formule pour tous les processeurs. La fonction à optimiser s'écrit en fonction de la fréquence du processeur dans le cycle courant et des autres processeurs dans le cycle précédent. A chaque cycle, le processeur détermine la fréquence qui maximise cette

fonction et s'il trouve que cette fréquence est différente de sa fréquence du cycle précédent, il prend la nouvelle fréquence sinon, il garde l'ancienne valeur. Cette approche demande donc une communication directe entre tous les contrôleurs, ce qui pourrait ralentir la convergence vers une situation d'équilibre si le temps nécessaire pour chaque processeur pour avoir les informations nécessaires de tout le système est grand.

Pour des systèmes où la rapidité de la convergence vers une situation d'équilibre entre contrôleurs est importante, la coordination se fait généralement à travers une communication entre les contrôleurs voisins. Dans [72], l'optimisation de la consommation d'énergie par DFVS dans un MPSoC a été basée sur un échange local de valeurs de fréquences (entre chaque processeur et ses voisins). Dans cette approche, le concept de consensus est utilisé. Le consensus est un processus itératif utilisant un protocole d'échange de messages prédéfini permettant à un ensemble d'éléments communicants de se mettre d'accord sur une valeur ou un comportement [93]. Le consensus dans [72] est de converger vers le vecteur de fréquence correspondant à une configuration optimale du système. La stratégie de prise de décision de chaque processeur dans cette approche est basée sur la méthode d'optimisation par sous-gradients [49]. Dans les approches de contrôle dirigées par le concept de consensus, il peut y avoir également des contrôleurs qui jouent le rôle de leader. Les objectifs des leaders sont communiqués aux autres contrôleurs afin d'orienter leurs décisions et de converger vers un consensus [93]. Une variation de la notion de leader a été utilisée dans [158], dans laquelle les leaders sont implicites. Cela veut dire que les leaders n'ont pas besoin de communiquer directement leurs objectifs aux autres contrôleurs pour atteindre le consensus. Dans cette approche, la stratégie de chaque contrôleur contient deux parties : une partie qui tend vers le suivi des contrôleurs voisins et une partie qui tend vers l'atteinte d'un objectif global. Cette deuxième partie n'est prise en compte que pour certains contrôleurs qui se trouvent à un instant donné bien placés pour connaître l'objectif global à atteindre. Par exemple, dans un système multi-robots, les robots se trouvant près d'une cible donnée sont les plus capables d'identifier ou de détecter cette cible. Ces robots qui sont à cet instant les plus informés de la cible jouent le rôle de leaders qui orientent, en échangeant leurs directions avec les autres robots, de les orienter vers la cible.

3.3.3 Synthèse

La plupart des travaux sur la coordination entre contrôleurs complètement distribués visent des problèmes de contrôle homogènes entre contrôleurs qui suivent des formules bien définies qui sont les mêmes pour tous les contrôleurs à quelques différences près [37] [108] [109][72] [93] [158]. Ces formules considèrent le problème de contrôle comme une fonction des coûts de chaque configuration en utilisant les mêmes métriques pour tous les contrôleurs. Cela permet donc de faciliter la formulation du problème de contrôle géré par chaque contrôleur. Or, avec des problèmes de contrôle prenant en compte des données autres que les coût telles que le changement des préférences des utilisateur ou de l'environnement, la vision globale que doit avoir un contrôleur augmente le nombre de variables qu'il doit prendre en compte. Cela peut augmenter significativement la complexité d'un contrôleur et le rendre dépendant du système et difficile à réutiliser. Dans ce cas, le contrôle complètement distribué peut ne pas être efficace du point de vue

conception. De plus, le nombre de messages nécessaires pour arriver à une configuration satisfaisante pour tous peut avoir un impact non négligeable sur la performance du système du contrôle. Une solution a ce problème est d'utiliser un modèle de contrôle hiérarchique qui permet la séparation entre les problèmes de contrôle locaux et le problème de contrôle global facilitant ainsi la réutilisation des contrôleurs des différents niveaux et la scalabilité du modèle de contrôle.

3.4 Le formalisme des automates de modes pour le contrôle

L'utilisation d'un formalisme pour la conception du contrôle de l'adaptation dynamique favorise la réutilisation et la scalabilité. En effet, la formalisation du contrôle permet de traiter le comportement du système d'une manière abstraite avec des sémantiques précises, ce qui facilite la modification et la réutilisation du contrôle. Cela offre aussi la possibilité d'implémenter le contrôle ainsi conçu de différentes manières visant différentes plates-formes FPGA ou autres. Les formalismes souvent utilisés dans ce contexte sont des formalismes basés sur les machines à états [65] [67] ou des réseaux de Pétri [38]. Ces formalismes offrent une grande flexibilité avec des possibilités de compositions parallèles et hiérarchiques qui peuvent être facilement adaptées au contrôle centralisé et décentralisé. Le modèle de prise de décision proposé dans ce mémoire est inspiré des automates de modes [73]. Les automates de modes (inspirés des machines à états finis) sont à la base une extension des langages à flot de données synchrones (SDF) tels que Lustre [47]. Pour ce type de langages, la notion de mode opératoire correspond à des situations où les sorties du système sont gérées de la même façon (même équations/programmes) à des instants distincts de l'exécution. Pour ce type de situations, les langages synchrones utilisaient des structures conditionnelles. Par exemple, pour une sortie X, on écrivait *X=if (mode1) then ... else if (mode2) then ...*, ce qui conduisait à utiliser plusieurs fois la même structure conditionnelle pour toutes les variables qui dépendent des mêmes modes. Ceci rend le code difficile à lire et à modifier. Pour remédier à ce problème, le concept d'automate de modes a été introduit pour offrir des sémantiques claires pour le contrôle.

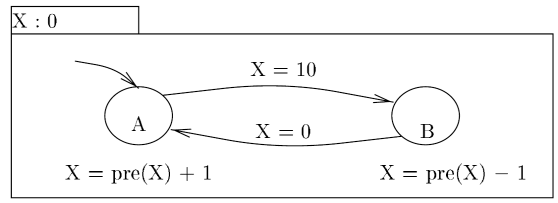
Le formalisme d'automates de modes convient donc pour la modélisation des systèmes à comportement modal en facilitant la gestion de la réaction entre le contrôle et les données. Un tel formalisme permet d'augmenter la lisibilité et facilite la compréhension du comportement du système puisque la structure de ces automates spécifie clairement les différents modes de fonctionnement et les conditions de commutation entre ces modes. Cette représentation en automates offre aussi la possibilité à des sous-systèmes d'avoir leurs propres modes. Dans ce cas, la vue globale d'un système donne un produit cartésien des ensembles de modes. Cette possibilité de composition permet ainsi d'augmenter la modularité du contrôle facilitant sa conception.

3.4.1 Définition formelle et sémantiques des automates de modes

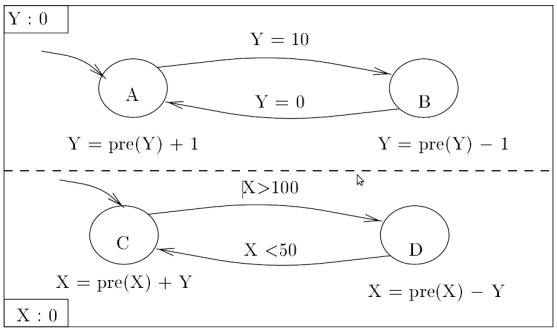
Un automate de modes est un tuple $(Q; q_0; V_i; V_o; I; f; T)$ [73] où :
– Q est l'ensemble d'états de l'automate et q_0 est l'état initial ;

- V_i et V_o sont les ensembles des variables d'entrée et de sortie, respectivement. Ces deux ensembles sont disjoints ($V_i \cap V_o = \phi$). On note $V = V_i \cup V_o$ l'ensemble de toutes les variables de l'automate de modes ;
- $I : V_o \rightarrow D$ est la fonction qui définit la valeur par défaut des variables de sortie ;
- $T \subseteq Q \times C(V) \times Q$ est l'ensemble de transitions libellées par des variables de V. Les conditions dans $C(V)$ sont des expressions booléennes mais sans des "pre operators" (des actions liées aux transitions) ;
- $f : V_o \rightarrow (Q \rightarrow EqR(V))$ est une fonction ; une variable dans V_o est associée à une fonction totale de Q vers l'ensemble $EqR(V)$ des expressions qui constituent les parties droites des équations.

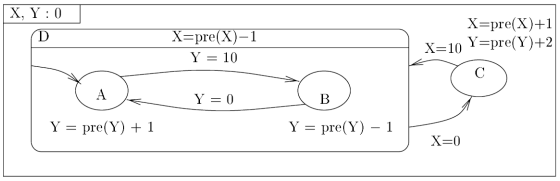
Les automates de modes sont donc composés de modes et de transitions comme le montre la figure 3.1(a). Puisque les variables d'entrée/sortie sont globales pour un même automate de modes, chaque mode à la même interface (mêmes variables d'entrée/sortie). Dans tout le manuscrit, nous utilisons la même notation pour indiquer les variables d'entrée et de sortie : l'entête de l'automate contient le nom de l'automate suivi des variables d'entrée entre parenthèses, suivis d'un signe égale et des variables de sortie entre parenthèses. Comme l'indique la figure 3.1(a), l'ensemble des variables d'entrée est vide et l'ensemble des variables de sortie contient seulement la variable X. Une variable d'entrée ne peut être utilisée que dans la partie droit des équations ou dans les conditions des transitions, alors qu'une variable de sortie peut être partout [73]. Les automates



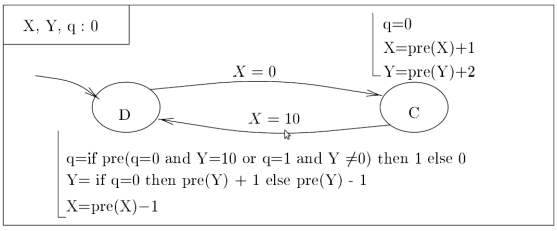
(a) Automate de modes simple



(b) Automates de modes parallèles



(c) Automate de modes hiérarchique



(d) La mise à plat de l'automate de la figure 3.1(c)

FIG. 3.1 – Exemples d'automates de modes

de modes peuvent être composés d'une façon parallèle ou hiérarchique. Dans une combinaison parallèle, les automates peuvent communiquer entre eux dans le sens où la

sortie d'un automate est l'entrée de l'autre, ce qui permet de synchroniser les automates parallèles. La figure 3.1(b) montre un exemple d'automates parallèles. L'ensemble des modes d'un automate équivalent est le produit cartésien des ensembles de modes des deux automates parallèles. Les conditions de transitions de l'automate équivalent sont l'union des conditions de transitions des automates parallèles. La composition hiérarchique est un raffinement de certains modes de l'automate comme le montre la figure 3.1(c) qui est un raffinement de l'automate 3.1(d). Cette composition permet de distinguer les différentes équations utilisées pour les mêmes variables de sortie. Ici, dans la figure 3.1(c), on distingue deux équations différentes pour la variables Y en utilisant un automate hiérarchique, ce qui permet de simplifier le code utilisé par rapport à son équivalent plat.

3.4.2 La vérification formelle

L'un des avantages de l'utilisation d'un formalisme dans la conception d'un système est qu'il existe des outils facilitant la vérification du bon fonctionnement du système à partir d'une description de celui-ci en utilisant le formalisme choisi. Cette vérification s'appelle la vérification formelle. Le Model-Checking [23] [110] est parmi les techniques les plus utilisées pour la vérification dans le contexte des formalismes inspirés des machines à états finis. Il s'agit ici, de faire une représentation du système à l'aide de machines à états finis et de spécifier une propriété temporelle à vérifier. L'utilisation d'un outil de Model-Cheking permet de vérifier si la propriété est vraie pour toute évolution possible du système. Pour le formalisme des automates de modes, l'outil SIGALI [14] est parmi les outils les plus utilisés pour le Model-Cheking. Cet outil se base sur une spécification du système dans un langage synchrone. Dans [159], SIGALI a été utilisé afin de permettre la vérification des propriétés non fonctionnelles (énergie, qualité de communication, etc) à partir d'un modèle UML dans l'environnement Gaspard2. A l'aide des transformations de modèles, le modèle UML a été transformé en code synchrone. Le code généré est utilisé dans l'outil SIGALI pour la vérification formelle des propriétés non fonctionnelles (vérifier par exemple que la somme des ressources utilisées par les différents modes actifs ne dépassent pas un certain seuil). Cette vérification permet donc de détecter les erreurs dans le design et le corriger si nécessaire. Une autre méthode de vérification formelle d'un design est la synthèse du contrôle. L'avantage de cette méthode par rapport au Model-Cheking est qu'elle permet de synthétiser automatiquement et formellement la fonction de contrôle à partir de descriptions haut niveau. Cette méthode part aussi d'une spécification du système en langage synchrone sous forme d'automates de modes et d'une propriété temporelle à vérifier et permet de générer automatiquement un contrôleur (centralisé) en langage C ou Java [31] qui assure que cette propriété soit vraie pour toute évolution possible du système.

Dans notre travail, puisque nous visons un contrôle matériel décentralisé, les outils de vérification formelle existants pour les systèmes spécifiés à l'aide d'automates de modes ne peuvent pas être utilisés directement. Pour l'application de la première méthode de vérification qui est le model checking, il faut suivre les étapes suivantes 1) mettre en place des transformations de modèles qui permettent d'obtenir du code synchrone à partir de la spécification UML MARTE d'un système de contrôle semi-distribué telles que celles

proposées dans [44] et 2) après la vérification, écrire une transformation permettant de passer du code synchrone en code VHDL. Pour appliquer la deuxième méthode qui est la synthèse de contrôle, il faut étendre les outils de synthèse existants afin de générer du code VHDL décentralisé au lieu d'un contrôleur centralisé en C ou Java. La vérification formelle n'est pas utilisée dans la version actuelle de ce travail, puisque notre objectif principal est de valider le modèle de contrôle décentralisé en matériel. La vérification du bon fonctionnement du modèle est donc laissée à la phase de simulation VHDL. Cependant, il est toujours possible d'intégrer la vérification formelle à notre approche en développant l'outillage nécessaire pour cela et en suivant les démarches indiquées ci-dessus.

3.4.3 Approches basées sur les automates de modes pour la conception du contrôle des systèmes reconfigurables

Les différentes configurations d'un système/sous-système correspondent à des modes exclusifs de celui-ci, ce qui correspond bien aux principes des automates de modes. Pour cette raison, certains travaux ont utilisé les automates de modes pour spécifier le contrôle des systèmes embarqués reconfigurables. Dans [62] [46], des approches IDM pour la conception du contrôle de la reconfiguration des tâches d'une application ont été proposées dans le cadre de l'environnement Gaspard2. Dans ces travaux, le concept des automates de modes a été introduit au profil et métamodèle de l'environnement Gaspard2 [42] afin de modéliser le contrôle et de générer du code synchrone correspondant. Dans [56], la modélisation du contrôle a été étudié au niveau déploiement permettant d'associer différentes implémentations VHDL à un accélérateur matériel du système. L'approche IDM utilisée permet de générer le code C du contrôleur correspondant. Dans [7] [31], la spécification des systèmes en automates de modes a été utilisée pour la synthèse du contrôleur. L'intégration de l'approche IDM à la synthèse du contrôleur afin de générer automatiquement le code synchrone permettant la synthèse du contrôleur en langage C a été proposée dans [44].

3.4.4 Synthèse

Le formalisme des automates de modes a été utilisé par différents travaux que ce soit pour les systèmes implémentés sur FPGA [56] [44] ou pour les autres systèmes embarqués [7] [62] [46] [31]. Ce formalisme a permis une grande facilité de la représentation du comportement des système pour ces différents travaux grâce à ses sémantiques claires. Il a été également utilisé pour la synthèse automatique du contrôle permettant ainsi d'accélérer les phases de conception. Cependant, l'implémentation ciblée par la plupart de ces travaux est une implémentation centralisée et logicielle, ce qui pourrait avoir un impact non négligeable sur la performance globale du système.

3.5 La modélisation à haut niveau et l'automatisation de la génération du code du contrôle

Devant la complexité et la taille croissantes des systèmes embarqués sur FPGA, la conception de tels systèmes est devenue une tâche coûteuse en temps et nécessite une grande maîtrise des détails techniques des plates-formes ciblées. Une solution pour améliorer la productivité de conception est d'élever le niveau d'abstraction afin de cacher les détails de bas-niveau et accélérer la phase de conception en utilisant un mécanisme automatisant le passage d'une modélisation à haut-niveau des systèmes au code permettant l'implémentation physique. Pour assurer ces objectifs, la méthodologie de conception du contrôle proposée dans ce travail utilise l'ingénierie dirigée par les modèles (IDM) afin d'assurer la génération automatique du code à partir de modèle de haut niveau et améliorer ainsi la productivité des concepteurs. Elle est intégrée dans l'environnement Gaspard2 [42] dédié à la conception des systèmes embarqués. Cet environnement utilise le profil UML standard MARTE (Modeling and Analysis of Real-Time and Embedded systems) [99] qui est un standard spécifique au domaine des systèmes embarqués.

Dans cette section, nous présentons brièvement les concepts de l'ingénierie dirigée par les modèles et son application au domaine des systèmes embarqués à travers des profils tels que le profil MARTE. Le flot de conception des systèmes embarqués basé sur IDM de l'environnement Gaspard2 est ensuite présenté. Les travaux sur la modélisation et la génération du contrôle dans Gaspard2 et dans d'autres projets sont ensuite étudiés.

3.5.1 L'ingénierie dirigée par les modèles (IDM)

Depuis longtemps, les modèles ont été utilisés dans les phases d'analyse et de conception pour servir de référence pour les phases d'implémentation. Ils ne constituaient que des documents qui décrivent notre perception du système. L'ingénierie dirigée par les modèles (IDM) fait sortir ces modèles d'une phase de passivité à une phase de productivité en faisant d'eux l'élément de base du processus de développement. L'intérêt pour l'IDM a été fortement amplifié à la fin du 20^{me} siècle lorsque l'OMG (Object Modeling Group) [94] a rendu publique son initiative MDA (Model Driven Architecture) [1]. Parmi les objectifs de l'MDA est d'automatiser la génération des applications suite à la modification des plates-formes cibles grâce à une séparation entre les parties métier et technique de l'application. Dans ce cas, pour migrer d'une plate-forme technique à une autre ou d'une version de cette plate-forme à une autre, il suffit de lancer automatiquement la plus grande partie du processus de conception en changeant simplement quelques détails dans les modèles. L'IDM représente ainsi une forme d'ingénierie générative où une grande partie de l'application est générée à partir des modèles.

La figure 3.2 illustre les 4 niveaux traités par l'IDM [15]. Le niveau M0 contient les entités à modéliser. Le niveau M1 contient les modèles utilisés pour les modéliser. Le niveau M2 contient les métamodèles qui définissent les modèles. Un métamodèle définit la structure que doit avoir tout modèle conforme à ce métamodèle. Le niveau

[1] http ://fr.wikipedia.org/wiki/Ingénierie_dirigée_par_les_modèles

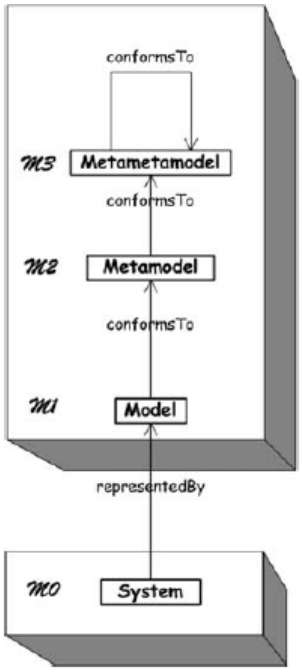


FIG. 3.2 – Architecture de la modélisation dans l'IDM

M3 contient le métamétamodèle qui s'auto-définit. C'est-à-dire qu'il n'y a pas de niveau supérieur au niveau M3. Le métamétamodèle définit par l'OMG est le MOF (Meta Object Facility).

Il existe trois types de modèles qui constituent le coeur de MDA : PIM, PSM et code. Le PIM (Platform Independent Model) est un modèle défini avec un niveau élevé d'abstraction indépendamment de toute technologie. Il décrit la partie métier du système indépendamment de l'implémentation. Le PSM (Platform Specific Model) est l'adaptation du PIM selon une technologie bien déterminée. Un PIM peut donner lieu à plusieurs PSM. Enfin, on trouve le code généré à partir d'un PSM. La figure 3.3 [8] représente les différents types de transformation permettant de passer d'un type de modèle à un autre ou de faire un raffinement sur un même type de modèles.

Le processus de développement basé sur l'IDM commence à un haut niveau d'abstraction et se termine par le modèle de bas niveau ciblé (un modèle exécutable ou un code), en suivant des niveaux d'abstraction intermédiaires à travers les transformations de modèles [120]. A chaque niveau intermédiaire, les détails d'implémentation sont ajoutés aux transformations. Une transformation est un processus qui transforme les modèles sources abstraits en des modèles cibles contenant plus de détails. Dans le cas d'une transformation exogène [80], les modèles source et cible sont conformes à des métamodèles différents comme le montre la figure 3.4. Dans une transformation endogène les modèles source et cible ont le même métamodèle. Une transformation est basée sur un ensemble de règles qui permettent de définir le passage des concepts du métamodèle source vers les concepts du métamodèle cible. Elle peut être étendue en ajoutant ou modifiant des règles afin d'obtenir une nouvelle transformation ciblant un

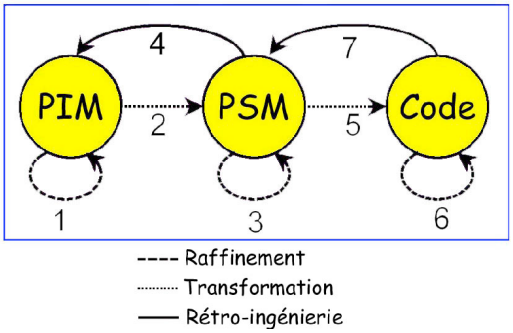


FIG. 3.3 – Les types de transformations de modèles

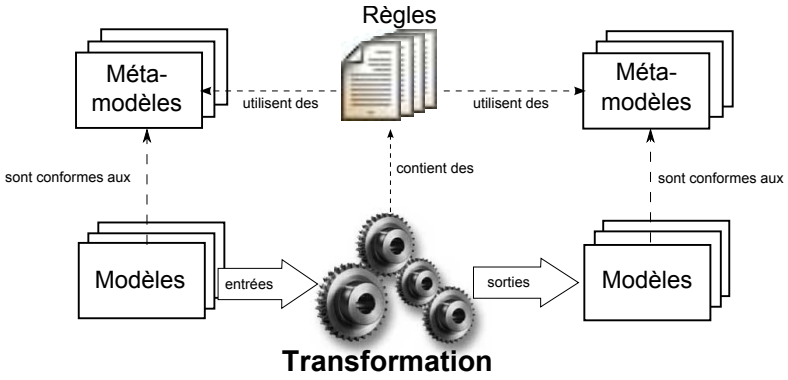


FIG. 3.4 – Une vue globale sur la transformation de modèles

modèle différent. L'avantage de cette approche est qu'elle permet de définir plusieurs transformations à partir du même niveau d'abstraction mais ciblant des niveaux plus bas différents, ce qui permet de cibler différentes plates-formes. Pour la spécification des transformations de modèles, l'OMG a proposé le standard QVT (Query/View/Transformation) [97] qui définit un ensemble de langages déclaratifs et impératifs pour la transformation. Parmi ces langages, on cite QVTO (QVT Operational) [111] qui est un langage impératif implémenté par Eclipse.

3.5.2 Les profils UML pour les systèmes embarqués

3.5.2.1 Les principaux profils UML proposés

UML (Unified Modeling Language) [98] est le langage visuel le plus utilisé dans l'IDM. Il fait parti des standards sur lesquels se base la MDA. Cependant, du fait qu'il était conçu à la base pour la standardisation des concepts de génie logiciel, il peut ne pas être suffisant pour d'autres domaines tel que celui des systèmes embarqués qui nécessite des concepts liés aux architectures matérielles. Pour résoudre ce problème, des profils UML ont été proposés pour différents domaines. Un profil UML est un ensemble de stéréotypes qui ajoutent à UML des informations spécifiques afin de décrire des systèmes d'un domaine spécifique.

Dans le domaine des systèmes embarqués, différents profils ont été proposés. Par exemple les profils UML for SoC [96] et UML for SystemC [115], sont des profils qui permettent de modéliser les systèmes embarqués en utilisant des concepts de bas niveau tels que les concepts Clock, Module, Channel qui sont principalement des concepts du SystemC. L'inconvénient de ces profils est qu'ils ne sont pas suffisamment abstraits et sont étroitement liés aux détails d'implémentation de bas niveau, ce qui ne leur permet pas de viser différents systèmes et plates-formes.

Le profil MARTE est l'un des profils qui offre une grande abstraction et généricité par rapport aux profils de bas niveaux tels que UML for SoC et UML for SystemC. Il est un profil standard de l'OMG pour la modélisation et l'analyse des systèmes temps-réel et embarqués. Ce profil vient remplacer et raffiner les concepts du profil UML SPT (Schedulability, Performance and Time) [95]. L'un des objectifs de MARTE est donc de surmonter les limites de SPT en traitant des aspects plus complexes des systèmes temps-réel tels, permettant ainsi de couvrir plus d'aspects liés à l'analyse de l'ordonnancement et de performance par rapport à SPT. Un autre objectif de MARTE est d'offrir un profil UML couvrant différents aspects logiciels et matériels des systèmes temps-réel et embarqués. Pour réaliser cet objectif tout en assurant une compatibilité avec des profils qui existent déjà dans ce domaine, MARTE a repris des concepts d'autres profils OMG tels que le profil UML QoS&FT (Quality of Service and Fault Tolerance) qui permet entre autres de traiter l'aspect qualité de service des systèmes et le profil SysML (System Modeling Language) [100] qui traite des aspects tels que la modélisation des plates-formes et leurs ressources matérielles et l'allocation du logiciel à ces plates-formes. MARTE est aussi compatible avec le standard AADL qui est un langage de conception et d'analyse de systèmes temps réel et embarqués très utilisé dans différents domaines tels que les domaines de l'avionique et de l'automobile. La compatibilité entre ces deux standards a été étudiée notamment dans [41]. L'utilisation de MARTE pour la modélisation des systèmes temps réel et embarqués offre beaucoup d'avantages par rapport à d'autres standards. En effet, il permet entre autres une meilleure analyse de l'ordonnancement et de la performance par rapport à SPT et des possibilités de traitement des propriétés non fonctionnelles telles que les contraintes de temps et de latence par rapport au profil SysML.

3.5.2.2 Le profil MARTE

Dans cette section, nous décrivons brièvement la structure du profil MARTE qui sera utilisé dans notre méthodologie de conception du contrôle semi-distribué. Les concepts MARTE pour la modélisation comportementale sont ensuite décrits vu qu'ils seront la base de la description du comportement du contrôle dans le cadre de cette méthodologie.

Structure du profil MARTE

Ce profil est composé de 4 packages comme le montre la figure 3.5 [99] : *foundations*, *design model*, *analysis model* et *annexes*. Ces packages sont construits selon leurs rôles. Le profil MARTE permet de traiter deux principaux aspects dans le domaine des systèmes temps-réel et embarqués qui sont la modélisation et l'analyse. Pour cela, MARTE utilise les packages *design model* et *analysis model* respectivement. Les éléments partagés par

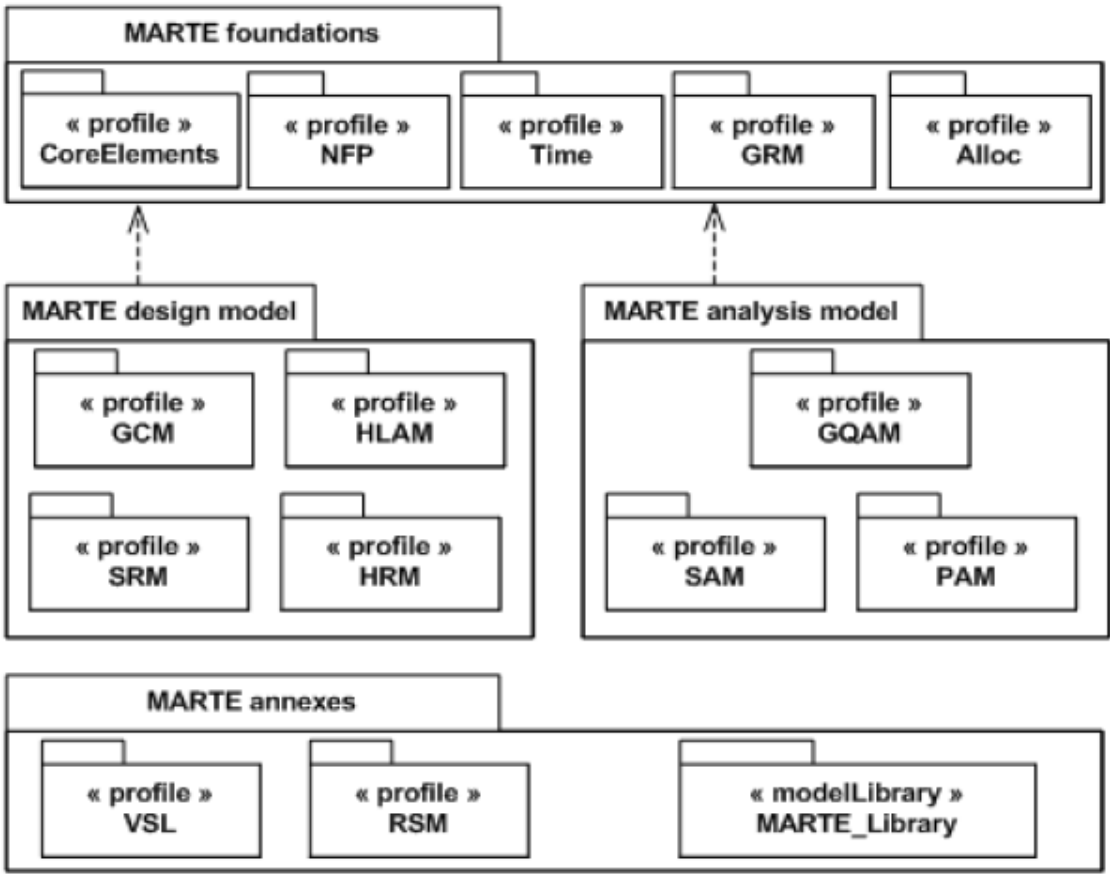


FIG. 3.5 – Une vue globale de l'architecture du profil MARTE

ces deux packages sont regroupés dans le package *foundations*. Le package *annexes* regroupe des notions qui pourraient être étendues pour des objectifs spécifiques de la conception. Ce package contient les types de données prédéfinis dans MARTE (sous-package MARTE_Library), des notions de parallélisme de données dans le sous-package RSM (Repetitive Structure Modeling) et le langage de spécification de valeur VSL dans le sous-package VSL (Value Specification Language).

Le package *foundations* contient les sous-packages *CoreElements*, *NFP*, *Time*, *GRM* et *Alloc*. Le sous-package *CoreElements* contient les éléments de base de tout modèle MARTE tels que ModelElement, Classifier, Instance, etc. Il contient aussi les éléments liés au comportement (Behavior, Action, Event, etc). Le sous-package *NFP* (Non Functional Properties) contient les notions permettant de modéliser les propriétés non fonctionnelles telles que la qualité de service, la consommation, etc. Le sous-package *Time* contient les notions liées au temps. Le sous-package *GRM* (Generic Resource Modeling) permet de modéliser des ressources d'une façon générique à travers des concepts tels que les ressources de calculs, de stockage, de synchronisation, etc. Le sous-package *Alloc* (Allocation) contient les notions permettant d'allouer l'application sur l'architecture.

Le package *design model* contient les sous-packages *GCM*, *HLAM*, *SRM* et *HRM*. Le

sous-package *GCM* (Generic Component Modeling) contient les composants génériques tels que les composants structurés, les ports de flots, les ports de messages, etc. Le sous-package *HLAM* (High-Level Application Modeling) contient principalement les notions de temps-réel telles que les services temps-réel, les politiques d'ordonnancement, etc. Le sous-package *SRM* (Software Resource Modeling) permet de modéliser les ressources logicielles telles que les ressources d'ordonnancement, d'interruption, etc. Le sous-package *HRM* (Hardware Resource Modeling) permet de modéliser les ressources matérielles telles que le processeur, la mémoire, etc.

Le package *analysis model* contient les sous-packages *GQAM*, *SAM* et *PAM*. Le sous-package *GQAM* (Generic Quantitative Analysis Modeling) contient les concepts génériques de l'analyse quantitative lié à des domaines tels que la performance, l'ordonnancement, la puissance, la mémoire, la sécurité, etc. Le sous-package *SAM* (Schedulability Analysis Modeling) hérite du sous-package *GQAM* et contient les notions nécessaires à l'analyse des scénarios d'ordonnancement. Le sous-package *PAM* (Performance Analysis Modeling) hérite lui aussi du sous-package *GQAM* et contient les notions nécessaires à l'analyse des propriétés temporelles.

Modélisation du comportement des systèmes embarqués dans MARTE

Pour la modélisation comportementale, MARTE fournit un package appelé Causality qui fait partie du package coreElements. Le package Causality contient les sous-packages *CommonBehavior*, *RunTimeContext*, *Invocation* et *Communication*, comme le montre la figure 3.6. Selon la spécification MARTE, le package *CommonBehavior* contient des éléments pour décrire deux types de comportements : le comportement basique et le comportement modal. La figure 3.7 décrit les éléments permettant de modéliser le comportement basique. La métaclasse Behavior représente le comportement d'un système ou d'une entité. Ce système/entité est représenté par la métaclasse BehavioredClassifier. Un BehavioredClassifier a un ensemble de Behaviors (comportements). Il peut aussi avoir un comportement principal parmi ses comportements. Un Behavior peut avoir des paramètres. Deux types de Bahavior peuvent être définis : l'action (comportement élémentaire) et le comportement composé (CompositeBahavior). Ce dernier est composé d'un ensemble d'actions. Un BehavioredClassifier contient un ensemble de déclencheurs (Trigger). Un trigger donne l'événement (Event) qui peut déclencher l'exécution du Behavior. Plus de détails sur la modélisation comportementale se trouve dans [99].

Le comportement modal distingue un type de comportement qui demande des considérations spécifiques pour les systèmes ayant des contraintes critiques de temps et de sécurité. Ce comportement est lié à la notion des modes opérationnels. Un mode opérationnel est un état d'un système ou d'un sous-système qui peut correspondre à une phase de l'opération (démarrage, arrêt,..). Les modes opérationnels d'un système peuvent aussi correspondre aux différents niveaux de qualité de services offerts selon les ressources disponibles, etc.

La figure 3.8 montre les éléments du comportement modal dans MARTE. Un BehavioredClassifier peut avoir un ensemble de modes modélisés par un ModeBehavior. Les modes d'un ModeBehavior sont mutuellement exclusifs. Un seul mode peut être actif à un instant donné. Les transitions entre les modes sont désignés par des ModeTransitions qui peuvent se produire en réponse à des Triggers. Comme a été décrit dans la section

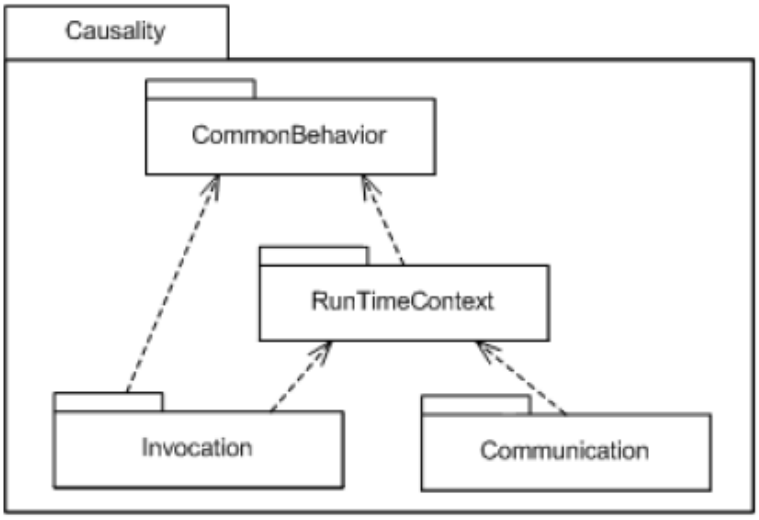


FIG. 3.6 – Structure du package Causality

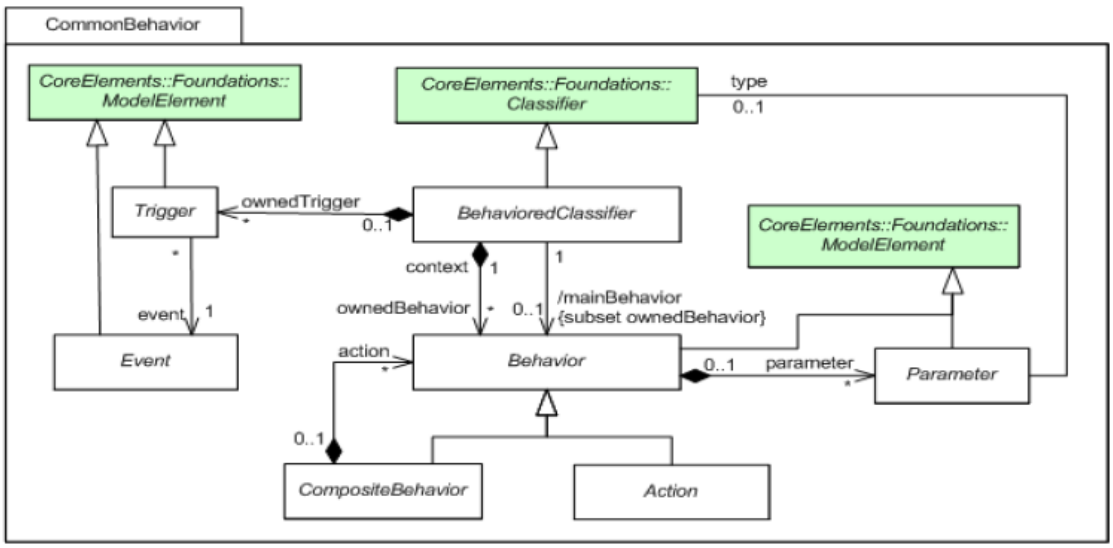


FIG. 3.7 – Partie du package CommonBehavior permettant de modéliser le comportement basique

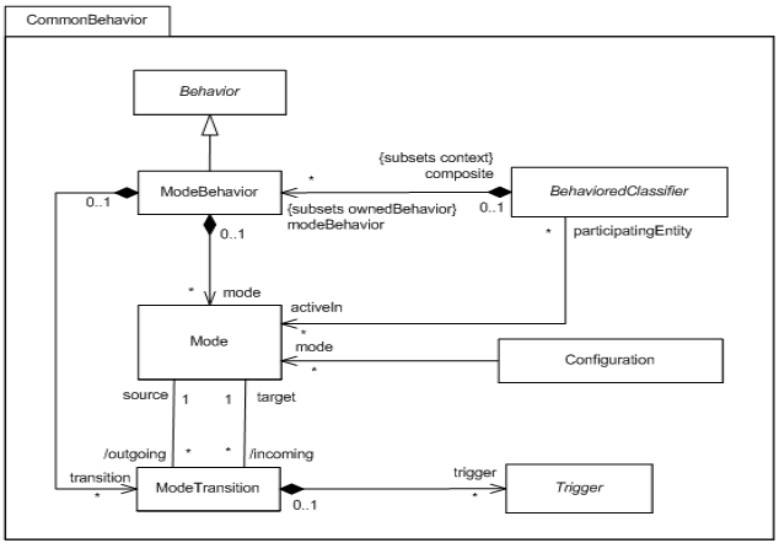


FIG. 3.8 – Partie du package CommonBehavior permettant de modéliser le comportement modal

du comportement basique, un trigger est lié à un event qui détermine les conditions causant ce comportement (la transition). Un BehavioredClassifier peut être actif dans zéro ou plusieurs modes. Ici, un mode correspond à un mode d'un système ou d'un sous-système auquel appartient le BehavioredClassifier. Une configuration dans MARTE représente zéro ou plusieurs modes.

La figure 3.9 montre la partie du profil MARTE qui permet de modéliser comportement modal. Les stéréotypes Mode, ModeTransition et ModeBehavior étendent les métaclasses State, Transition et StateMachine respectivement. Ceci permet de modéliser le ModeBehavior avec une représentation graphique de la machine d'états d'UML en utilisant des outils de modélisation tels que Papyrus [39]. Le stéréotype configuration étend les métaclasses StructuredClassifier et Package.

3.5.3 Le flot de conception des systèmes embarqués dans Gaspard2

Dans Gaspard2, l'application et l'architecture sont vues en tant qu'un assemblage de composants et sont modélisées par des composants UML. Le concept FlowPort de MARTE est utilisé pour les ports des composants dans l'application et l'architecture. Pour combler l'écart entre la modélisation à haut niveau utilisant MARTE et les plates-formes d'exécution, Gaspard2 utilise les concepts de déploiement et de transformation de modèles.

Le flot de conception des systèmes embarqués dans Gaspard2 suit les étapes suivantes : 1) la modélisation du système, 2) les transformations de modèles et 3) la génération de code, comme le montre la figure 3.10. Dans Gaspard2, il y a une séparation entre les modèles d'application et d'architecture. Le modèle d'application décrit les tâches logicielles ou matérielles d'une application. Le modèle de l'architecture décrit l'architecture sur laquelle l'application va être exécutée. Pour faire le lien entre les

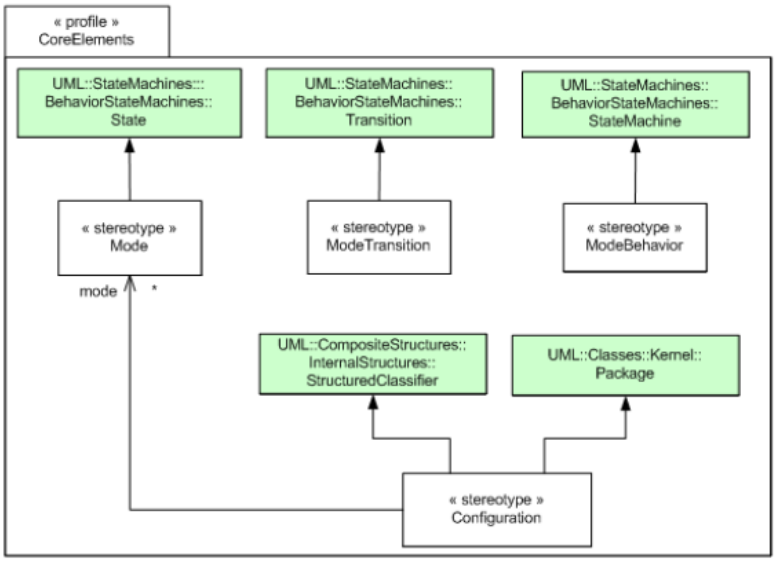


FIG. 3.9 – Partie du package CommonBehavior permettant de modéliser le comportement modal

modèles d'application et d'architecture, un modèle d'allocation est utilisé. Ces trois modèles utilisent le profil MARTE. Ils peuvent être considérés comme des modèles PIM (Platform Independent Model) puisqu'ils sont indépendants de la plate-forme cible. Au niveau déploiement, chaque composant élémentaire (un composant de l'application ou de l'architecture) du système est lié, à travers le profil de déploiement de Gaspard2, à un code qui existe déjà dans une bibliothèque d'IP facilitant ainsi la réutilisation des IPs. Le modèle de déploiement fournit des informations du les IPs qui seront utilisées par la suite par les transformations de modèles ciblant différents domaines (validation/ analyse [46], simulations [11] [127] [126], calcul haute-performance [116], synthèse [56], etc). Ce modèle est donc un PSM (Platform Specific Model). Après la phase de déploiement, les transformations de modèles permettent d'ajouter des détails au modèle d'entrée afin de se rapprocher de la technologie cible. Les transformations de modèles sont organisées en chaînes de transformations qui peuvent partager les transformations liées à des notions communes entre les cibles. A la fin d'une chaîne de transformations, on obtient un modèle PSM avec des détails techniques permettant la génération du code lié à la technologie cible. Gaspard2 permet la génération de code ciblant différents langages tels que Fortran, Lustre, SystemC, OpenCL, C et VHDL. Les outils utilisés dans Gaspard2 sont Papyrus [39] pour la modélisation, QVTO [111] pour la transformation de modèles et Acceleo [2] pour la génération de code.

Les applications ciblées par Gaspard2 sont les applications de traitement de signal intensif. Ces applications sont basées sur des calculs réguliers appliqués sur une grande quantité de données. Les systèmes implémentant ce type d'applications ont donc un comportement régulier. Il est possible aussi pour certains systèmes de passer d'un comportement régulier à un autre durant l'exécution. Ces systèmes sont donc à modes de fonctionnement. Par exemple, un système de traitement d'images peut passer du

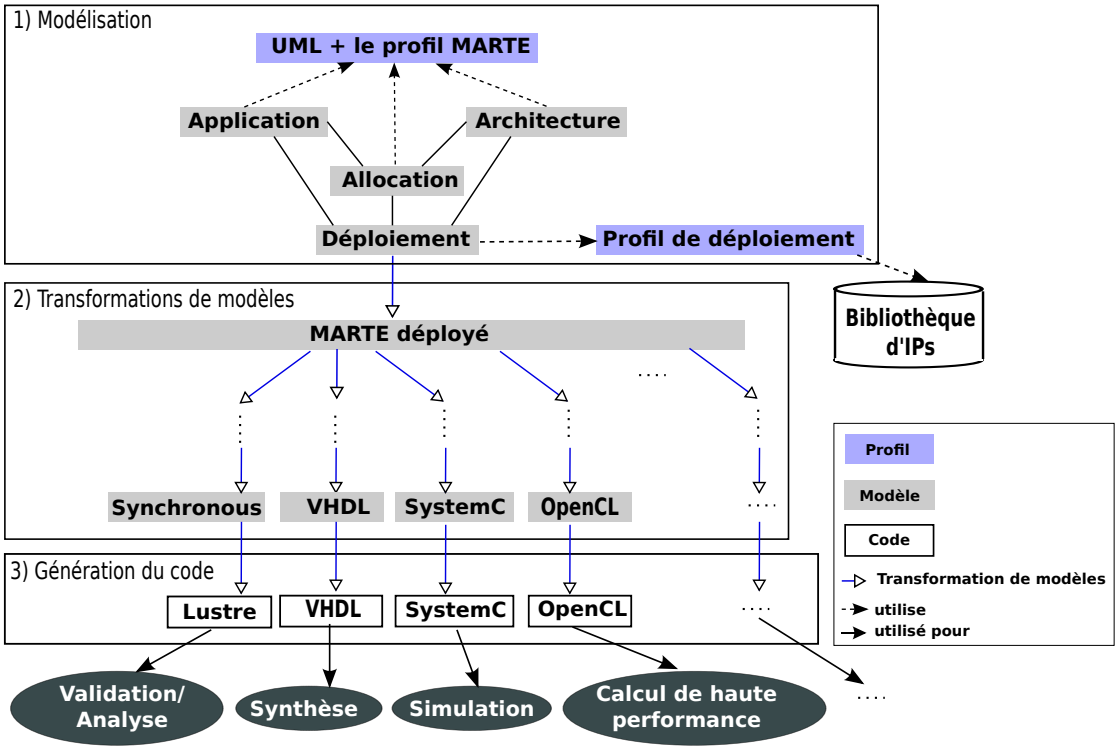


FIG. 3.10 – Le flot de conception des systèmes embarqués dans Gaspard2

mode monochrome au mode couleur. Le changement de mode peut être dû à des événements liés à l'environnement externe ou à l'état interne du système. Le formalisme d'automates modes convient bien pour la modélisation du contrôle de ces systèmes. Les approches de modélisation de contrôle dans Gaspard2 sont inspirées de ce formalisme.

3.5.4 Les approches de modélisation et génération du contrôle dans Gaspard2

3.5.4.1 Introduction des automates de modes dans Gaspard2

Les premiers travaux de modélisation du contrôle dans Gaspard2 se trouvent dans [62]. Ces travaux ont introduit la notion d'automate de contrôle inspiré des automates de modes. Comme précisé dans la section 3.4, les automates de modes ont été introduits pour les langages synchrones. Les travaux dans [62] sont inspirés de ces automates pour modéliser une application contrôlée en vue d'automatiser la génération de son code en langages synchrones qui est une des plates-formes ciblées par Gaspard2. Pour avoir un modèle clair et facile à maintenir et à vérifier, cette approche propose la séparation entre contrôle et données illustrée dans la figure 4.2. La partie contrôle est représentée par un automate de contrôle. La sortie de cet automate est une valeur de mode qui sera utilisée par la partie contrôlée de l'application pour choisir le mode de fonctionnement à activer parmi plusieurs modes exclusifs. Pour pouvoir intégrer cette modélisation du contrôle dans Gaspard2, un automate de contrôle est modélisé par une fonction de transition et

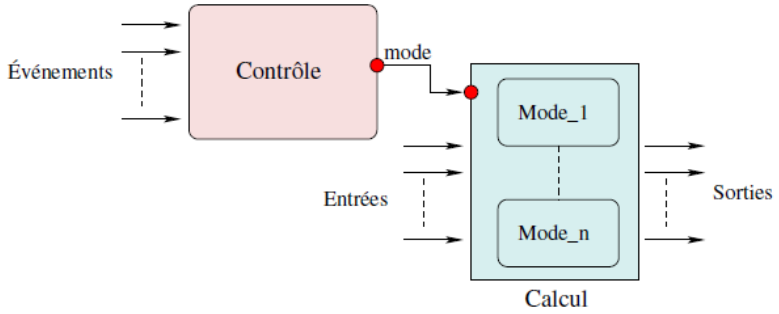


FIG. 3.11 – Vue globale de la séparation contrôle/données dans [62]

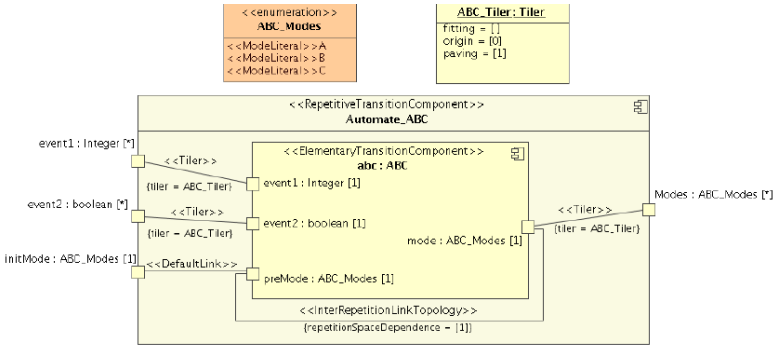


FIG. 3.12 – Modélisation d'un automate de contrôle dans [62]

une dépendance inter-répétition comme le montre la figure 3.12. La fonction de transition est modélisée par un composant ayant le stéréotype «ElementaryTransitionComponent» qui est une extension du profil Gaspard2. Ce composant a en entrée des événements et le mode courant de l'automate. Il donne en sortie le mode cible. Dans un automate, l'état/mode cible d'une transition est l'état/mode source de la transition suivante. Cette dépendance est modélisée par le stéréotype «InterRepetitionLinkTopology» de Gasaprd2. La fonction de transition est répétée une infinité de fois. L'automate est donc composé par une répétition du composant représentant la fonction de transition et par une relation de dépendance inter-répétition. L'entrée de cet automate est un flux d'événements. Le symbole [*] sur les ports représente l'infinité. Il donne en sortie un flux de modes qui vont être utilisés par l'application pour activer le mode de calcul indiqué par l'automate. Cette modélisation de l'automate de contrôle respecte bien le principe des modèles dans Gaspard2 qui sont basés sur la description des dépendances de données entre les composants du modèle. Ici, la relation de dépendance liée à la modélisation de l'automate de contrôle est relative à des données particulières (des données de contrôle) entre les différentes répétitions de la même instance de composant. Le mode initial est donné par le port stéréotypé «DefaultLink». Ce stéréotype est utilisé dans Gaspard2 pour modéliser la dépendance pour la première répétition d'un composant.

Dans la figure 3.12, le comportement de la fonction de transition est vue comme une boîte noire, ce qui limite les aspects modélisables du contrôle. Pour cette raison, à côté

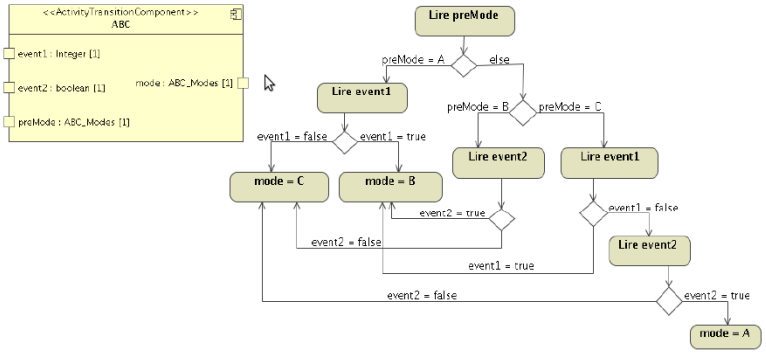


FIG. 3.13 – Modélisation d'une fonction de transition dans [62]

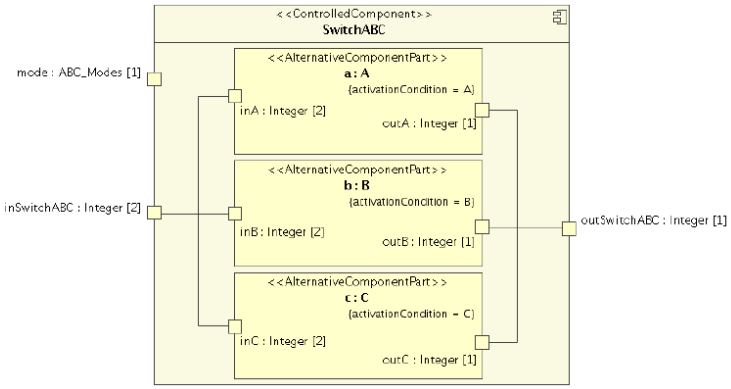


FIG. 3.14 – Modélisation d'un composant d'application contrôlé dans [62]

du stéréotype «ElementaryTransitionComponent» , un autre stéréotype peut être utilisé pour modéliser la fonction de transition. Ce stéréotype est le «ActivityTransitionComponent». Dans ce cas, le comportement de la fonction de transition est spécifié par un diagramme d'activités comme le montre la figure 3.13. Cependant, la correspondance entre les activités et les ports de la fonction de transition n'a pas été formalisée, ce qui ne permet pas à cette représentation d'être utilisée pour une génération de code de la fonction de transition.

La valeur de mode en sortie de l'automate de contrôle est utilisée pour déterminer le mode à activer pour la partie contrôlée de l'application. Pour cette raison, chaque tâche contrôlée est modélisée par un composant composé qui regroupe les différents modes d'exécution de cette tâche comme le montre la figure 3.14. La tâche contrôlée et ses différents modes d'exécution sont modélisés par les stéréotypes «ControlledComponent» et «AlternativeComponentPart» respectivement. Chaque «AlternativeComponentPart» a un attribut «activationCondition» qui indique le nom du mode qui doit être actif pour son exécution. Les modes d'un composant contrôlé doivent avoir la même interface (même nombre et types des ports d'entrée/sortie). Le «ControlledComponent» joue donc le rôle de «switch» qui selon la valeur du mode en entrée choisi le mode à activer.

Cette approche a été limitée à la modélisation, et la génération de code synchrone

correspondant n'a pas été traitée.

3.5.4.2 La notion de state graphs

Le modèle de contrôle proposé Les travaux dans [46] ont introduit la notion de State Graph dans Gaspard2. Cette notion est inspirée de la notion de fonction de transition utilisée dans [62]. Un State Graph est un ensemble d'états et de transitions. A chaque état est associé un ou plusieurs modes. L'avantage est de pouvoir associer un seul state graph à plusieurs tâches contrôlées en leur donnant en sortie les modes à activer pour chaque tâche, ce qui évite au concepteur de modéliser un automate de mode pour chaque tâche contrôlée.

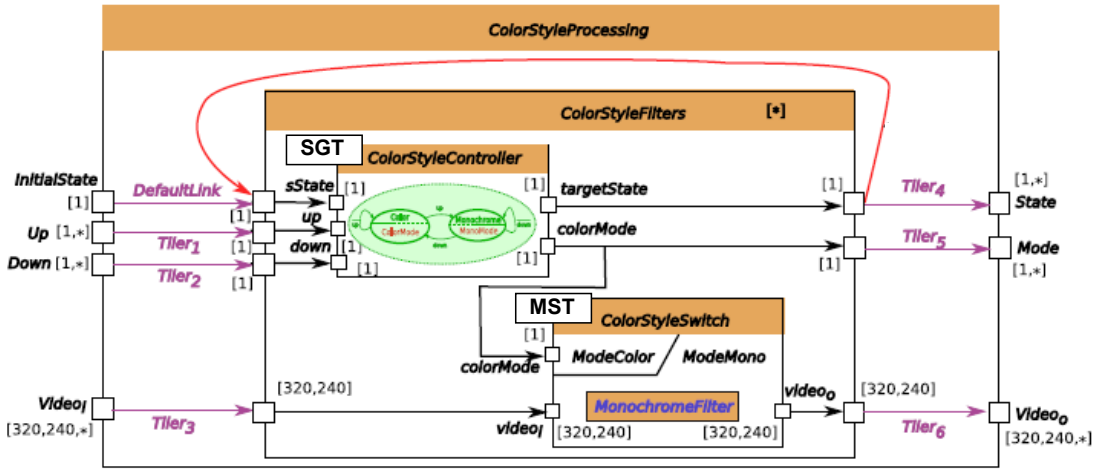


FIG. 3.15 – Modélisation d'un système de contrôle dans [46]

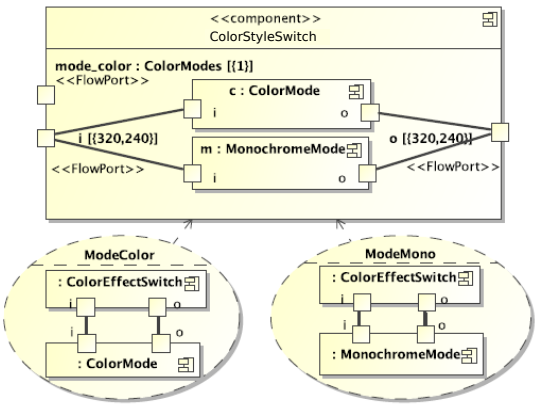


FIG. 3.16 – Modélisation d'un composant d'application contrôlé dans [46]

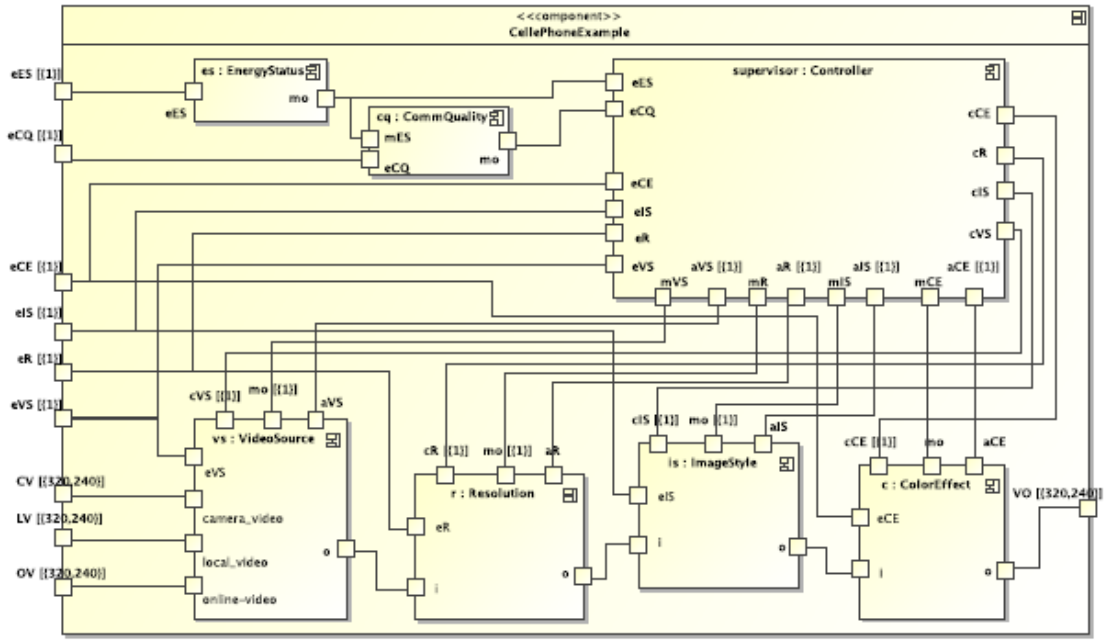


FIG. 3.17 – Exemple d'application utilisant plusieurs automates de modes dans [46]

La modélisation UML Pour intégrer la notion de State Graph dans Gaspard2, des State Graph tasks (SGT) ont été utilisées. Ce sont des tâches spéciales de l'application dans Gaspard2 qui permettent de déterminer les modes à activer pour une tâche contrôlée. L'entrée d'une SGT est un ensemble d'événements et un état source. La sortie est un état cible et un ou plusieurs modes associés à cet état. A une SGT est associée une machine à états UML qui représente le State Graph comme le montre la figure 3.15. Cette figure décrit un SGT nommé *ColorStyleController* pour un module de traitement d'images qui a deux modes Color et Monochrome. Le changement de ces modes est conditionné par les événements *up* et *down* qui permettent d'aller dans les modes Color et Monochrome respectivement. Le mode en sortie d'un SGT sera donné à une Mode Switch Task (MST). Cette MST est inspirée de la notion de switch utilisée dans [62]. Dans la figure 3.15, le MST contrôlé par le SGT s'appelle *ColorStyleSwitch*. Pour spécifier l'implémentation de chaque mode dans le MST, des collaborations UML sont associées à celui-ci comme le montre la figure 3.16. Chaque collaboration indique l'association entre la valeur du mode (nom de la collaboration) et le mode à exécuter. Ici, les tâches ColorMode ou MonochromeMode sont exécutées lorsque la valeur du mode actif est ModeColor ou ModeMono, respectivement. La modélisation d'un automate complet est basée sur la répétition d'un composant englobant la SGT et la MST en utilisant une dépendance inter-répétition comme le montre la figure 3.15.

A côté de la simplification de la modélisation à travers les machines à états UML au lieu des diagrammes d'activités et de la possibilité d'utiliser le même automate pour plusieurs composants contrôlés, cette approche a également introduit d'autres améliorations par rapport à celle utilisée dans [62] en offrant la possibilité de modéliser

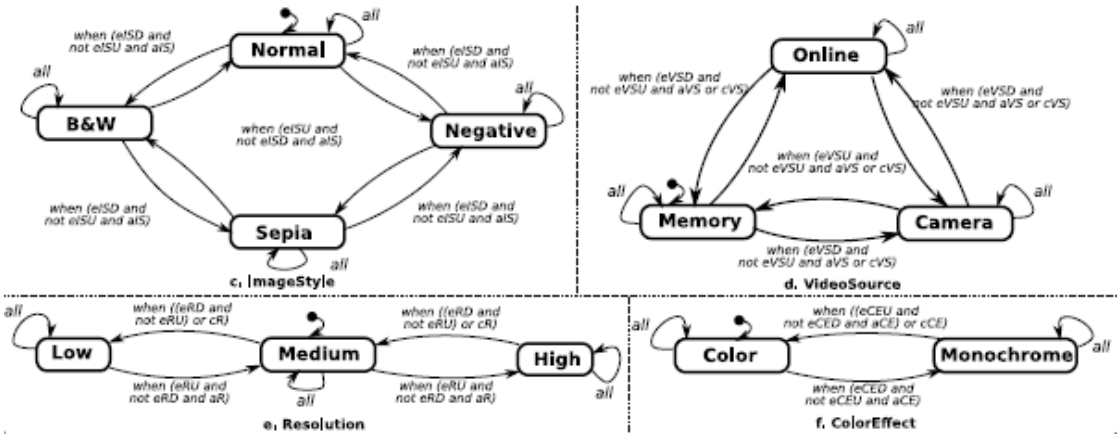


FIG. 3.18 – Les automates de modes des 4 tâches de l'application de la figure 3.17

des automates composés parallèlement et hiérarchiquement.

Synchronisation entre automates de modes La coordination entre les automates de modes des tâches de l'application a été aussi traitée dans ce travail. La figure 3.17 représente la modélisation d'une application de traitement d'images pour un téléphone portable, mettant en oeuvre la coordination entre automates de modes. Cette application est composées de 4 tâches. Les 4 composants en bas de la figure sont des composants qui englobent la SGT et la MST correspondantes à chacune de ces 4 tâches. Les contrôleurs *EnergyStatus* et *CommQuality* permettent de déterminer respectivement le niveau d'énergie et le niveau de communication. Ici, on suppose que ces deux données sont détectées par des capteurs de batterie et une antenne de communication. Le composant *Controller* permet la coordination entre les automates de modes des 4 tâches. Il collecte différents événements et décide les changements de modes à autoriser. Comme le montre la figure 3.17, ce contrôleur collecte entre autres les événements en entrées des automates de modes des 4 tâches pour prendre sa décision en se basant sur les ressources disponibles en termes d'énergie, de ressources matérielles, etc. Dans chacun des automates de modes des tâches de l'application, les conditions de transitions dépendent des événements en entrées et de l'autorisation du contrôleur comme le montre la figure 3.18. Dans cette approche, le contrôleur est écrit manuellement par le concepteur. Aucun formalisme pour la conception d'un tel contrôleur n'a été proposé. Ce contrôleur a donc une vision globale de tout le système et de ses événements, ce qui le rend très complexe pour des systèmes de grande taille. De plus, sa dépendance aux événements en entrée des automates de modes rend difficile sa réutilisation.

3.5.4.3 Le contrôle au niveau déploiement

Dans [56], les sémantiques de contrôle dans Gaspard2 ont été étendues pour pouvoir modéliser le contrôle des architectures reconfigurables ciblant la plate-forme FPGA. Cette approche utilise les mêmes concepts de state graph et de modes présentés dans [46]. A la différence des approches présentées précédemment, le contrôle est modélisé

au niveau déploiement en proposant une extension du profil déploiement de Gaspard2 en ajoutant la notion de configuration. Cette extension permet d'associer différentes implémentations (SoftwareIP) à la même tâche de l'application. Ces implémentations sont commutées en utilisant la RDP. Pour modéliser cet aspect, la notion de configuration a été introduite. La figure 3.19 donne un exemple de l'utilisation des configurations. Dans cet exemple, la tâche *MultiplicationAddition* a deux implémentations différentes : en « switch case » et en « if then else ». Cette tâche est déployée par un VirtualIP nommé *VirtualMultAdder*. L'utilisation d'un VirtualIP dans Gaspard2 permet à un même composant d'être déployé de différentes manières et de cibler différentes plates-formes. Plus de détails sur le déploiement dans Gaspard2 peuvent être trouvés dans [42]. Ici, au VirtualIP sont associés deux implémentations différentes ciblant la même plate-forme qui est VHDL. Chacune de ces implémentations est représentée par un SoftwareIP permettant d'indiquer des informations sur l'IP utilisée tel que le langage. La figure 3.20 indique les différents modes du systèmes. Dans cet exemple, ils correspondent aux modes de la tâche *MultiplicationAddition* puisqu'elle est la seule tâche reconfigurable du système. Les configurations dans la figure 3.19 correspondent aux différentes configurations du système. L'attribut *configurationID* d'une configuration donne la valeur de mode pour laquelle la configuration est active. L'attribut InitialState est un booléen qui indique s'il s'agit de la configuration initiale du FPGA. L'attribut ip d'une configuration permet de lier cette configuration aux IPs utilisées.

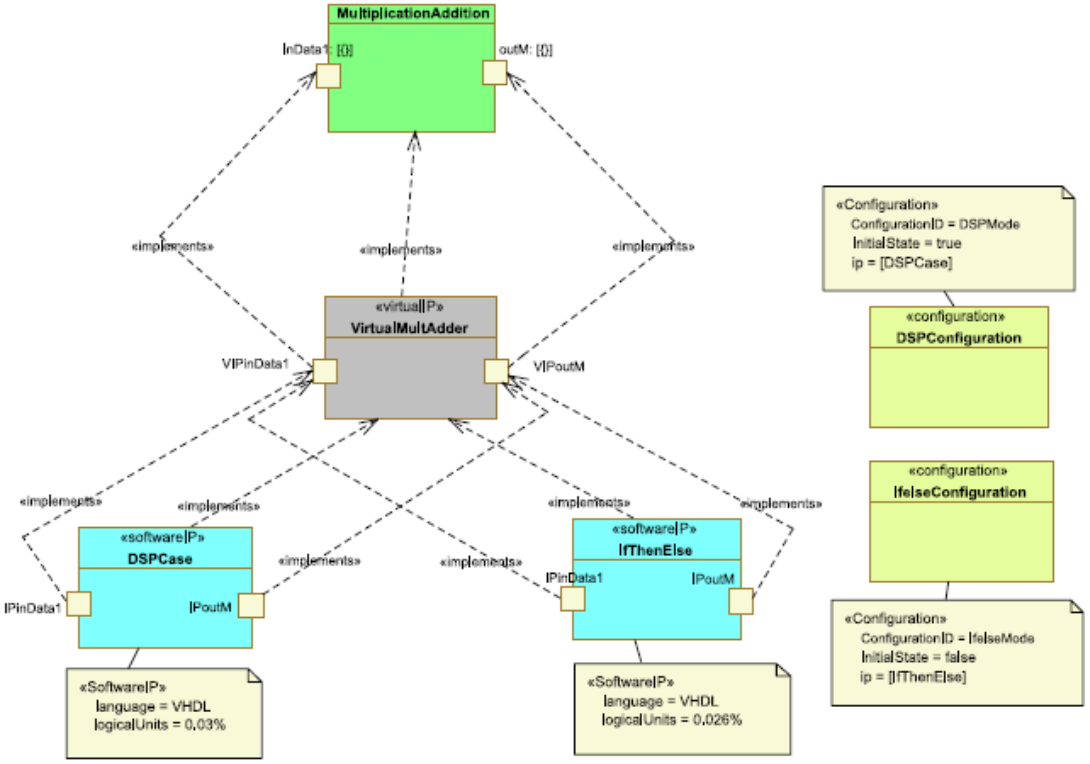


FIG. 3.19 – Déploiement d'une tâche reconfigurable dans [56]

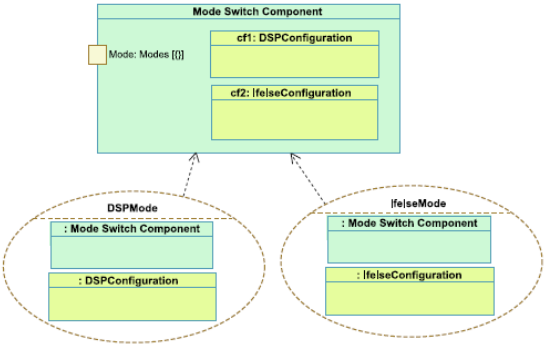


FIG. 3.20 – Association entre la notion de configuration et la notion de mode dans [56]

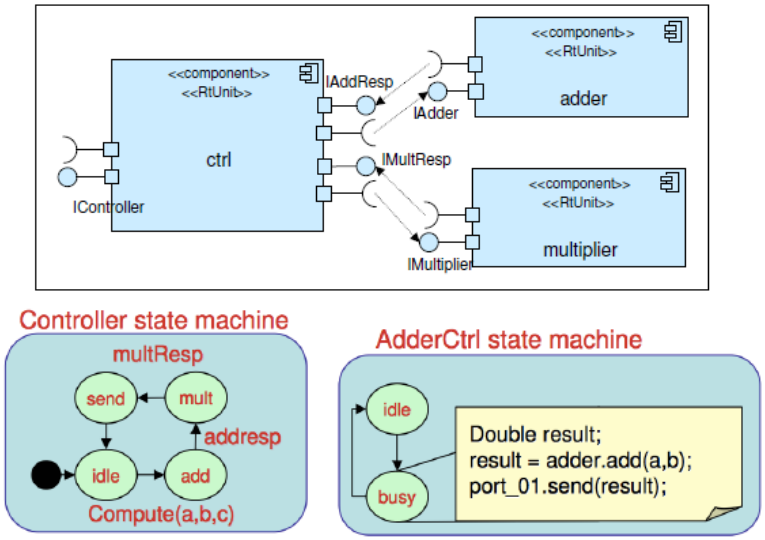


FIG. 3.21 – Modélisation du contrôle dans MOPCOM

3.5.5 Autres approches de modélisation et de génération du contrôle

Dans le projet MOPCOM [132], la modélisation du contrôle d'un système reconfigurable sur FPGA est basée sur les machines d'états UML. Dans cette approche, le système est contrôlé par un seul contrôleur qui est une tâche logicielle qui gère l'ordonnancement des tâches de l'application. Cette approche est illustrée dans la figure 3.21. La machine d'états du contrôleur représente le graphe de tâches de l'application. Une reconfiguration est lancée si la tâche à exécuter n'est pas chargée dans la région reconfigurable qui la gère. Dans le cas de la figure 3.21, le contrôleur communique avec deux modules : *adder* et *multiplier*. Chacun de ces modules contient un contrôleur modélisé par une machine à deux états (idle et busy) et un module qui une tâche de l'application. L'état *add* de la machine à états du contrôleur correspond à un contrôleur qui communique avec le composant *adder* pour exécuter la tâche d'addition et l'état *add* correspond à un contrôleur qui communique avec le composant *multiplier* pour exécuter la tâche de

multiplication. Cette approche ne considère donc la reconfiguration que d'un point de vue ordonnancement sans prendre en compte d'autres éléments tels que le changement dans l'environnement ou les préférences des utilisateurs, ce qui la rend peu flexible et limitée à certains problèmes de contrôle.

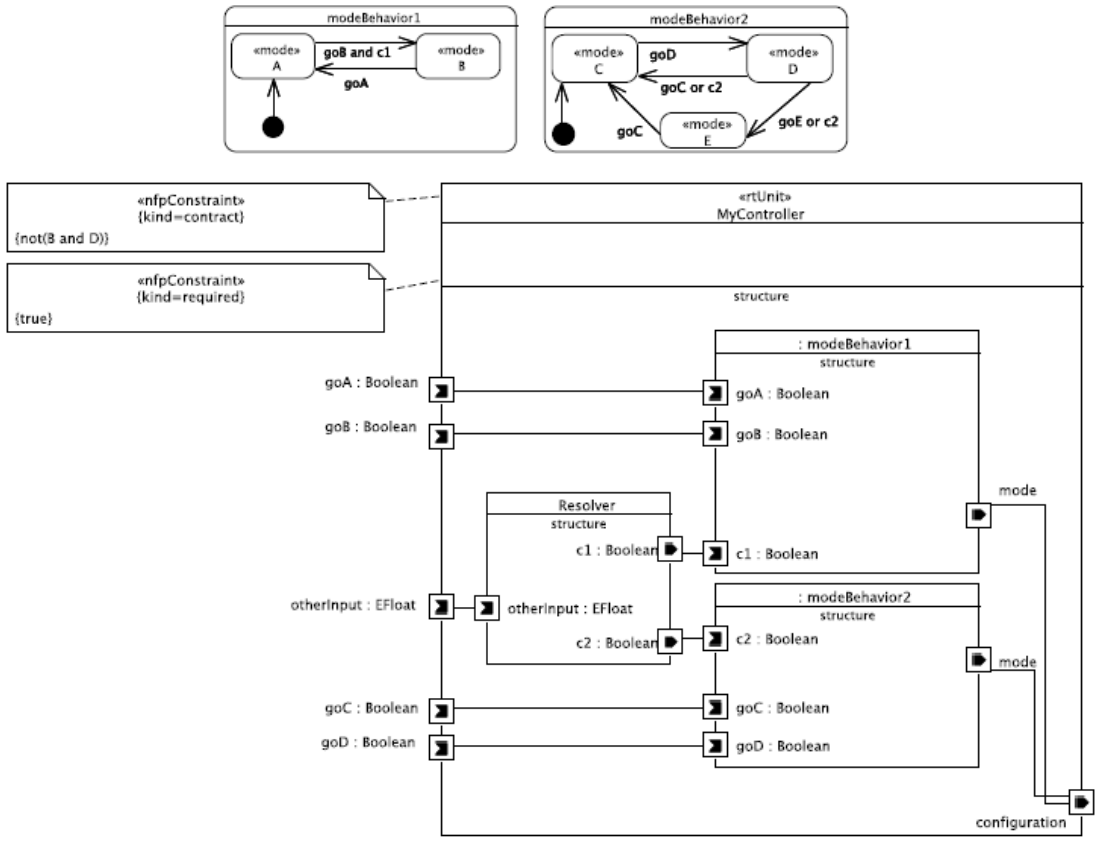


FIG. 3.22 – Modélisation du contrôle dans FAMOUS

Dans le projet FAMOUS [44] dont le flot de conception est inspiré de celui de Gaspard2, le contrôle est modélisé par un ensemble d'automates de modes utilisant les concepts de MARTE pour la modélisation du comportement modal par rapport aux approches de conception dans Gaspard2, qui ont été proposées avant la prise en compte de ce type de comportement par MARTE. Les systèmes ciblés dans le projet FAMOUS sont des systèmes reconfigurables sur FPGA. La figure 3.22 donne un exemple de modélisation du contrôle dans FAMOUS qui utilise les stéréotypes MARTE décrits dans la figure 3.9. Dans cet exemple, le comportement du système est modélisé par deux automates de modes. Chaque automate peut représenter les différents modes d'une tâche de l'application comme il peut être lié à des modes globaux tels que les modes d'énergie du système, etc. Les contraintes à respecter par l'ensemble des modes actifs des automates est spécifier sous la forme d'une contrainte «nfpConstraint» de MARTE. Dans l'exemple de la figure 3.22, la contrainte a respecter par le contrôleur est que les modes *B* et *D* ne doivent pas être actifs en même temps quelques soient les valeurs d'événements en en-

trée aux automates (*goA*, *goB*, *goC* et *goD*). Pour garantir que cette contrainte sera validée pour toute évolution du système, des variables contrôlables sont utilisées. Ces variables sont à ajouter aux conditions des transitions des automates de modes. Dans l'exemple de la figure 3.22, il s'agit des variables *c1* et *c2*. L'approche du projet FAMOUS pour la synchronisation entre automates de modes est basée sur la synthèse du contrôleur. Le rôle du concepteur ici est de spécifier la contrainte à vérifier, de choisir les variables contrôlables à utiliser et de les ajouter aux conditions d'origine des transitions en les combinant avec les événements de l'environnement (les variables non contrôlables). A partir de cette modélisation en MARTE, le code synchrone est généré et le contrôleur est synthétisé en langage C. Le rôle de ce contrôleur est de gérer automatiquement les valeurs des variables contrôlables à travers le resolver (voir figure 3.22) afin d'assurer le respect de la contrainte pour toute évolution du système. Cette approche est très intéressante du fait qu'elle automatise la génération d'un contrôleur en utilisant une implantation correcte par construction à l'aide de l'outil SIGALI. Cependant, elle permet la génération d'un seul contrôleur centralisé écrit en logiciel, ce qui peut avoir un impact non négligeable sur la performance globale du système.

3.5.6 Synthèse

L'application d'une approche IDM pour la modélisation et la génération du contrôle dans l'environnement Gaspard2 a été traitée par certains travaux qui sont tous inspirés du formalisme des automates de modes. L'approche utilisée dans [46] a intégré beaucoup d'améliorations par rapport à celle de [62] en utilisant des machines à états UML pour la modélisation du contrôle au lieu de diagrammes d'activités permettant ainsi de faire la correspondance entre les états UML et les modes. Elle a proposé également une extension du standard MARTE pour la modélisation du contrôle au lieu de l'utilisation du profil Gaspard et elle a traité aussi la génération du code synchrone à partir des modèles UML. Cependant, cette approche limite la modélisation du contrôle au niveau application, ce qui ne permet pas d'associer différentes implémentations à la même fonctionnalité offrant la possibilité à une tâche de l'application de s'adapter aux changements de l'environnement ou des préférences de l'utilisateur. Pour la gestion de la coordination entre les modes actifs des automates de modes de l'application, cette approche utilise un contrôleur qui prend en compte tous les événements en entrée à ces automates, ce qui le rend dépendant des événements spécifiques aux différentes tâches et limite ainsi sa flexibilité et sa réutilisabilité. [56] a traité le contrôle au niveau déploiement, ce qui permet d'associer plusieurs implémentations à la même fonctionnalité par rapport à [46]. Cette approche a été utilisée pour le contrôle des systèmes reconfigurables sur FPGA. Le code du contrôleur est généré en langage C afin d'être exécuté par le processeur des systèmes FPGA ciblés. Cependant, cette approche s'est limitée à l'utilisation d'un seul automate de modes pour tout le système et le problème de coordination entre automates de modes n'a pas été traité.

D'autres projets tels que MOPCOM et FAMOUS proposent aussi une approche IDM pour la modélisation et la génération du contrôle pour les systèmes reconfigurables sur FPGA. Ces deux projets ont traité le contrôle de deux façons différentes. Dans MOPCOM, le déclenchement des reconfigurations est gérée par un mécanisme d'ordonnancement

	FOSFOR [87]	AETHER [107]	DodOrg [119]	FAMOUS [44]	RaaR [40]	MOPCOM [132]	Gaspard2 [56]	Notre approche [129] [130]
Contrôleurs modulaires	oui	oui	oui	-	-	-	-	oui
Auto-adaptation distribuée	oui	oui	oui	-	oui	-	-	oui
Séparation entre décisions locales et globales	-	-	-	oui	oui	-	-	oui
Formalisme de contrôle	-	-	-	oui	-	-	oui	oui
Approche IDM	-	-	-	oui	-	oui	oui	oui
Implémentation décentralisée	oui	oui	oui	-	oui	-	-	oui
Implémentation matérielle	oui	oui	oui	-	-	-	-	oui

TAB. 3.1 – Synthèse des travaux sur le contrôle de l'adaptation dynamique des systèmes implémentés sur FPGA

des tâches. Dans FAMOUS, le contrôle est modélisé par un ensemble d'automate de modes représentant chacun les différents modes d'une tâche de l'application ou d'un aspect global tel que l'énergie, la qualité de la communication, etc. La coordination entre automates a été également traitée à travers la synthèse du contrôleur. Cependant, ces approches se sont limitées à des contrôleurs centralisés implémentés en logiciel, ce qui pourrait avoir un impact non négligeable sur la performance globale du système.

3.6 Synthèse

Différents travaux sur le contrôle des systèmes reconfigurables ont été présentés dans ce chapitre, en traitant les aspects architecture, algorithmes de coordination, formalisme et génération automatique, du domaine des FPGAs et d'autres domaines (mécatronique, robotique, etc). Etant donné que notre méthodologie de conception vise les systèmes partiellement reconfigurables sur FPGA, il faut préciser sa contribution par rapport aux travaux dans ce domaine. Le tableau 3.1 illustre l'évaluation des principaux travaux de contrôle de l'adaptation dynamique pour les systèmes implémentés sur FPGA. Cette évaluation se base sur un nombre de critères liés à la flexibilité, la réutilisabilité, la scalabilité, l'automatisation et l'efficacité de l'implémentation qui sont les objectifs visés par la méthodologie de conception de contrôle proposée dans ce travail. Comme le montre le tableau, la méthodologie de conception proposée dans ce travail, qui est illustrée par la dernière colonne, vise à couvrir différents points permettant d'assurer ses objectifs de flexibilité, de réutilisabilité, de scalabilité, d'automatisation et d'efficacité de l'implémentation. La contribution de notre méthodologie est d'offrir en même temps :

- des contrôleurs modulaires qui permettent l'auto-adaptation des composants reconfigurables facilitant leur réutilisation,
- une séparation entre les problèmes locaux et globaux de contrôle facilitant la réutilisation des contrôleurs et la scalabilité du contrôle,
- une gestion de la coordination entre les décisions des contrôleurs afin de respecter les contraintes/objectifs globaux du système,
- l'utilisation d'un formalisme de contrôle facilitant la modification et la réutilisation,
- l'automatisation de la génération du contrôle à partir de modèles de haut niveau

afin d'accélérer la phase de conception,
- une distribution spatiale permettant de s'adapter à différentes plates-formes FPGA telles que les systèmes multi-FPGAs en diminuant les coûts de communication par rapport aux solutions centralisées,
- une implémentation matérielle permettant d'assurer la performance du contrôle.

La méthodologie proposée dans ce travail permet de fournir à chaque composant reconfigurable, les services nécessaires en termes d'observation, de prise de décision de reconfiguration et de réalisation de reconfiguration afin d'assurer son auto-adaptation. De cette façon, le contrôleur offrant ces services peut facilement être entièrement réutilisé avec le composant contrôlé ou adapté à d'autres composants en entrant quelques changements sur certains de ses modules qui peuvent être testés séparément, ce qui facilite et accélère la phase de conception. Pour assurer la flexibilité et faciliter la réutilisation de ces contrôleurs, il faut réduire leur dépendance du système en les limitant à une vision locale du problème du contrôle. Cependant, il faut assurer en même temps que les décisions prises à partir de ces visions locales respectent bien les contraintes/objectifs globaux du système. Ici, la coordination entre contrôleurs doit être faite de façon que ces derniers ne perdent pas le caractère local des problèmes du contrôle qu'ils gèrent. Le mécanisme de coordination proposé dans ce travail se base sur la séparation entre les problèmes de contrôle locaux gérés par les contrôleurs et le problème de contrôle global du système. Ce dernier est géré par un coordinateur. Les décisions de reconfigurations prises par les contrôleurs doivent donc passer par ce coordinateur qui transforment ces décisions en des propositions de reconfiguration aux autres contrôleurs si nécessaire afin de satisfaire ces décisions tout en respectant les contraintes/objectifs globaux du système. Les échanges entre le coordinateur et un contrôleur donné sont donc limités à des requêtes et des propositions de reconfiguration qui concernent seulement le composant reconfigurable géré par ce dernier. Cela permet aux contrôleurs de garder leur vue locale et d'être facilement réutilisable. Cette séparation permet aussi au coordinateur d'être indépendant des problèmes locaux des contrôleurs et de n'avoir qu'une vue globale du système, ce qui facilite sa réutilisation et son adaptation à différents systèmes.

Pour faciliter la conception des systèmes de contrôle, la formalisation du comportement et des échanges entre les contrôleurs et le coordinateur est une approche très intéressante qui permet d'offrir des sémantiques claires facilement réutilisables. Le formalisme choisi pour ce travail est le formalisme des automates de modes qui convient bien la modélisation des différents modes des composants reconfigurables et des conditions de transitions entre modes. Grâce à la composition parallèle offerte par ce formalisme, les contrôleurs et le coordinateur peuvent être modélisés par des automates de modes parallèles communicants. Cela permet d'avoir un modèle homogène du contrôle facilitant ainsi le travail du concepteur pour le test et la modification. L'utilisation d'un tel formalisme facilite aussi l'abstraction du comportement du modèle de contrôle par rapport à l'implémentation physique offrant ainsi la possibilité d'adapter le modèle de contrôle ainsi construit à différentes technologies, protocoles de communication, plates-formes, etc. Afin de bien exploiter cette abstraction, l'automatisation de la génération du code à partir d'une description abstraite basée sur ce formalisme permet d'épargner au concepteur une manipulation directe des détails d'implémentation qui peut ralentir énormément la phase de conception surtout s'il n'est pas expert dans le domaine. Pour

cela, notre méthodologie propose aussi l'automatisation de la génération du contrôle à partir de modèles de haut-niveau d'abstraction utilisant le profil UML MARTE qui convient bien pour la modélisation de la structure des contrôleurs et du coordinateur et le comportement modal de ceux-ci.

3.7 Conclusion

Dans ce chapitre, nous avons présenté différents travaux sur le contrôle des systèmes reconfigurables en traitant les aspects architecture, algorithmes de coordination, formalisme et génération automatique. La contribution de notre travail a été enfin présentée par rapport à ce qui a été fait dans le domaine du contrôle de l'adaptation dynamique des systèmes FPGA. Cette contribution est basée sur un modèle de contrôle semi-distribué composé de contrôleurs modulaires assurant les tâches d'observation, de prise de décision et de reconfiguration des région contrôlées, et d'un coordinateur entre des décisions de ces contrôleurs afin de respecter les contraintes globales du système. Le modèle de prise de décision semi-distribué est basé sur le formalisme des automates de modes. A travers cette combinaison entre la modularité, la séparation des problèmes locaux et globaux du contrôle, et le formalisme, la méthodologie de conception proposée dans ce travail vise à assurer la flexibilité, la réutilisabilité et la scalabilité du contrôle. Cette combinaison est appuyée par une approche de conception basée sur l'IDM afin d'automatiser la génération du code à travers des modèles de haut-niveau d'abstraction, permettant ainsi de faciliter le travail des concepteurs et d'améliorer leur productivité.

Dans le chapitre suivant, le modèle de contrôle semi-distribué proposé dans ce travail est présenté en détaillant la structure et le comportement des contrôleurs et du coordinateurs et l'application du formalisme des automates de modes dans ce contexte. Le passage de la modélisation à haut-niveau d'abstraction à la génération du code et l'implémentation physique sur FPGA sera détaillé dans les chapitres qui suivent.

Chapitre 4

Le modèle de contrôle semi-distribué

4.1 Introduction

Dans ce chapitre, nous présentons le modèle semi-distribué proposé pour le contrôle de l'adaptation dynamique des systèmes FPGAs [129]. Ce modèle a pour but d'améliorer la productivité des concepteurs dans ce domaine en assurant la flexibilité, le réutilisation et la scalabilité dans la conception du contrôle. Cette distribution permet aussi de remédier aux problèmes de communication que pourrait engendrer une solution centralisée, tels que les goulets d'étranglement. Ce modèle est basé sur des contrôleurs modulaires gérant l'auto-adaptation des régions reconfigurables du système assurant ainsi leur flexibilité et réutilisabilité. Les décisions de reconfiguration prises par ces contrôleurs

sont coordonnées afin de respecter les contraintes globales du système. La modélisation de cette prise de décision décentralisée se base sur le formalisme d'automates de modes, ce qui facilite la réutilisation du contrôle, et offre une description à haut-niveau d'abstraction facilitant l'adaptation de ce modèle à différents systèmes et technologies d'FPGA (systèmes mono-FPGA, systèmes multi-FPFA, 3D, etc).

4.2 Modèle de contrôle semi-distribué

Le modèle proposé divise le problème de contrôle en un ensemble de sous-problèmes locaux gérés par des contrôleurs autonomes. Chaque contrôleur gère l'auto-adaptation d'un composant reconfigurable du système à travers trois tâches allouées à trois modules différents : 1) l'observation des événements susceptibles de déclencher l'adaptation du composant contrôlé, 2) la prise de décision concernant les adaptations nécessaires et 3) la réalisation de l'adaptation/reconfiguration du composant contrôlé. L'allocation des tâches du contrôleur à des modules séparés augmente leur flexibilité et facilite leur modification et réutilisation et par conséquent la scalabilité des systèmes de contrôle.

A cause de la vision locale de chaque contrôleur, le lancement de la reconfiguration du composant contrôlé par ce contrôleur pourrait avoir un effet non désirable sur le reste du système. En effet, avant de lancer une reconfiguration d'un composant, il faut vérifier si la configuration cible pour ce composant peut coexister avec les configurations courantes des autres composants du système à cause à des contraintes de sécurité, de qualité de service, etc. Pour résoudre ce problème, nous proposons un mécanisme de coordination entre les contrôleurs. Cette coordination est réalisée par un coordinateur, ce qui rend le modèle de contrôle un modèle semi-distribué.

La prise de décision est parmi les aspects les plus critiques dans la conception du contrôle. Pour diminuer la complexité de la conception de cet aspect, l'utilisation d'un formalisme de contrôle permet l'abstraction du problème de contrôle et diminue sa dépendance à l'implémentation du système. Elle facilite aussi la vérification, réutilisation et la scalabilité de la conception. Dans notre travail, nous proposons l'utilisation du formalisme des automates des modes [73] pour la prise des décisions de reconfigurations. Ce formalisme convient pour modéliser le contrôle des différents modes dans un système reconfigurable tels que les modes d'énergie ou d'affichage vidéo par exemple. Le formalisme des automates de modes permet de traiter le comportement d'un système/sous-système d'une manière abstraite (un ensemble de modes) avec des sémantiques claires et précises. L'utilisation de ce formalisme pour la modélisation de la prise de décision semi-distribuée permet ainsi d'obtenir un modèle de contrôle facilement déployable sur toute sorte d'FPGA ou d'autre matériel reconfigurable.

Dans le reste du chapitre, nous présentons la structure des contrôleurs et du coordinateur. Nous passons ensuite à détailler l'utilisation du formalisme pour la prise de décision semi-distribuée.

4.3 Contrôleurs autonomes et modulaires

La figure 4.1 donne une vue globale du modèle de contrôle proposé. Il contient n contrôleurs associés à n régions reconfigurables. Chaque contrôleur contient un module d'observation, un module de décision et un module de reconfiguration. Les modules de décision des contrôleurs communiquent avec le coordinateur pour la prise de décision coordonnée. Notre modèle est basé donc sur des boucles de contrôle. Ces boucles ont pour entrées des événements de l'environnement externe ou de l'état interne du système (l'observation). Elles prennent ensuite la décision de reconfigurer le système si nécessaire (la décision) et elles mettent à jour le système selon la décision (la reconfiguration).

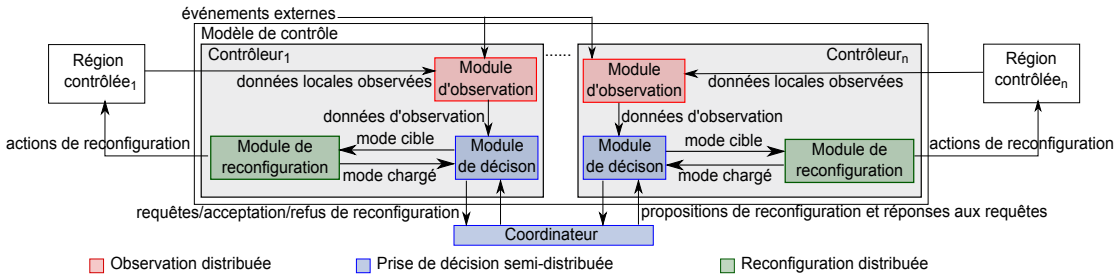


FIG. 4.1 – Vue globale du modèle de contrôle proposé

4.3.1 Observation

Dans notre modèle, les modules d'observation sont distribués entre les contrôleurs et sont en charge de l'observation des données qui pourraient déclencher une reconfiguration de la région contrôlée. Les informations collectées par le module d'observation peuvent provenir de l'environnement comme elles peuvent être liées à l'état interne ou le comportement de la région contrôlée. En effet, un contrôleur peut détecter les exigences de l'environnement du système en communicant par exemple avec un capteur qui détecte les changement dans la condition d'éclairage ou de chauffage, etc. En observant aussi les requêtes et les réponses échangées entre les régions contrôlées, plusieurs métriques peuvent être extraites telles que la consommation d'énergie [127] [126], la performance et d'autres paramètres dépendants de l'application.

Après la collecte d'information, le module d'observation les transforme en des données compréhensibles par le module de décision si nécessaire. Cette transformation consiste en l'application d'agrégats qui sont pris en compte lors de la décision. Ces agrégats peuvent être simples tels qu'une somme, des compteurs, etc. Ils peuvent être plus complexes tels que la consommation d'énergie de la région contrôlée ou la moyenne des intensités de pixels d'une image en entrée à cette région.

Les données d'observation sont ensuite envoyées au module de décision afin de les prendre en compte lors de la décision.

4.3.2 Prise de décision

Le processus de prise de décision est distribué entre les modules de décision des contrôleurs et le coordinateur comme le montre Figure 4.1. Les données d'observation sont prises en compte seulement par les contrôleurs, alors que le coordinateur gère la coordination des décisions prises par les modules de décision. Une telle décomposition du problème du contrôle permet d'améliorer la réutilisation et la scalabilité du modèle de contrôle. En se basant sur les données d'observation, chaque module de décision prend des décisions locales sur si la reconfiguration de sa région contrôlée est requise. Si c'est le cas, il envoie une requête de reconfiguration au coordinateur.

L'utilisation d'un coordinateur plutôt qu'une communication directe entre les contrôleurs a des avantages : 1) réduire le nombre de messages échangés. En effet, dans un modèle complètement distribué, un contrôleur pourrait être obligé à envoyer sa requête à tous les autres contrôleurs, ce qui implique un grand nombre de messages échangés avant d'arriver à une décision finale. 2) améliorer la réutilisation de la conception. En effet, dans un modèle complètement distribué, les contrôleurs échangent directement leurs décisions, ce qui nécessite une vision globale de chaque contrôleur afin de réagir correctement aux décisions envoyées par les autres contrôleurs et respecter les contraintes globales du système. Ceci rend le problème de contrôle géré par chaque contrôleur plus complexe et plus dépendent de l'implémentation du système et des contrôleurs avec qui il communique, ce qui représente un obstacle à la réutilisation de conception.

Le rôle du coordinateur est de vérifier si la requête de reconfiguration reçue (liée à une région donnée) ne nécessite pas la reconfiguration d'autres régions afin d'avoir une configuration globale qui respecte les contraintes/objectifs du systèmes. Si c'est le cas, le coordinateur autorise la reconfiguration directement. Sinon, il envoie des propositions de reconfiguration aux régions concernées. Selon les réponses (acceptations ou refus) des contrôleurs des ces dernières, le coordinateur prend sa décision à propos de la reconfiguration demandée. Cette décision peut une autorisation ou un refus de la reconfiguration. Dans le cas d'une autorisation, la reconfiguration peut être lancée par les modules de reconfiguration. Plus de détails sur la prise de décision semi-distribuée seront donnés dans le reste de ce chapitre.

4.3.3 Reconfiguration

La reconfiguration est gérée par les modules de reconfigurations des contrôleurs. Un module de reconfiguration a comme entrée une commande du module de décision indiquant la configuration à charger dans la région contrôlée. Le module de reconfiguration gère donc la reconfiguration de cette région à travers un port de configuration tel que l'ICAP pour les FPGAs Xilinx. Après avoir chargé le bitstream requis, le module de reconfiguration notifie le module de décision pour qu'il mette à jour sa vision de la configuration courante de la région, qu'il pend en compte dans la prise de décision.

L'utilisation de ce modèle distribué de reconfiguration permet de lancer les reconfigurations en parallèle, ce qui permet de réduire le temps de reconfiguration et d'améliorer la performance globale du système. Cela est intéressant pour des systèmes de grande

taille quand plusieurs reconfigurations partielles sont requises en même temps. La reconfiguration parallèle n'est actuellement possible que pour certains systèmes ayant plus d'un port de configuration tels que les systèmes multi-FPGAs. Cependant, notre modèle est toujours adaptable aux systèmes mono-FPGA. C'est ce que nous allons voir dans le chapitre 6 qui présente une étude de cas de l'application du modèle proposé à un système mono-FPFA.

4.4 Modèle de prise de décision semi-distribué basé sur les automates de modes

Comme précisé dans la section 3.4, le formalisme des automates de modes est à l'origine une extension aux langages synchrones. Il permet d'éviter l'écriture fastidieuse des équations lors de la spécification du contrôle des flots de données pour les différents modes d'exécution [73]. Les automates permettent d'améliorer la lisibilité et de faciliter la compréhension du comportement du système puisque la structure des automates spécifie clairement les modes d'exécution et les conditions de commutation entre eux. Les automates de modes sont principalement des automates dont les états (modes) correspondent aux différents modes d'exécution du système/sous-système qu'ils contrôlent. Un mode peut être vu comme une abstraction d'une collection d'états d'exécution. Ces états décrivent plus finement le comportement du système pour un mode d'exécution donné. En d'autres termes, un mode est un ensemble d'états dans lequel le système peut rester un certain temps sans aller dans des états qui ne sont pas dans cet ensemble [73]. Le comportement du système peut être vu comme une séquence de modes. Le formalisme des automates de modes permet donc de traiter le comportement d'un système d'une manière abstraite avec des sémantiques claires et précises afin de diminuer la complexité de la conception du contrôle.

Dans ce contexte, les différentes configurations d'une région du système peuvent être vues comme des modes exclusifs de cette région. Ce comportement modal des régions reconfigurables peut être donc modélisé en utilisant les automates de modes. Nous nous sommes inspirés de ce formalisme pour la modélisation des modules de décision des contrôleurs distribués qui gèrent chacun la commutation entre les modes de la région reconfigurable contrôlée. Ce choix permet de simplifier la conception des contrôleurs et faciliter leur modification et réutilisation, grâce aux sémantiques claires offertes par ce formalisme. Cela permet aussi de faciliter la génération automatique du code des automates pour différentes plates-formes en se basant sur la description à haut-niveau d'abstraction fournie par le formalisme des automates de modes. C'est ce que nous expliquerons dans le chapitre 5.

4.4.1 Adaptation des automates de modes pour le contrôle des régions reconfigurables

4.4.1.1 Séparation entre données et contrôle

Comme précisé dans la section 3.4, dans un automate de modes, les équations associées aux modes sont liées aux flux de données traités par les langages synchrones.

Donc, il n'y a pas de séparation entre données et contrôle. Pour adapter le formalisme des automates de modes à notre modèle de contrôle, la séparation entre contrôle et données est nécessaire puisque les deux sont traités par deux composants différents : le contrôle et le PRM (Partial Reconfigurable Module) [1], respectivement. Pour cette raison, nous proposons le modèle de séparation contrôle/données illustré dans la figure 4.2. L'automate de modes implémenté par le module de décision d'un contrôleur ne contient pas de traitement de données associées aux modes. Le traitement de données est fait par la la région reconfigurable (PRR) selon le mode actif. Pour lancer la reconfiguration correspondante au mode cible de son automate, le module de décision envoie ce mode au module de reconfiguration. Celui-ci fait la correspondance entre ce mode et le bitstream à charger dans la PRR contrôlée. La PRR n'exécute qu'un seul mode à un instant donné. Ce mode correspond au bitstream chargé dans cette région.

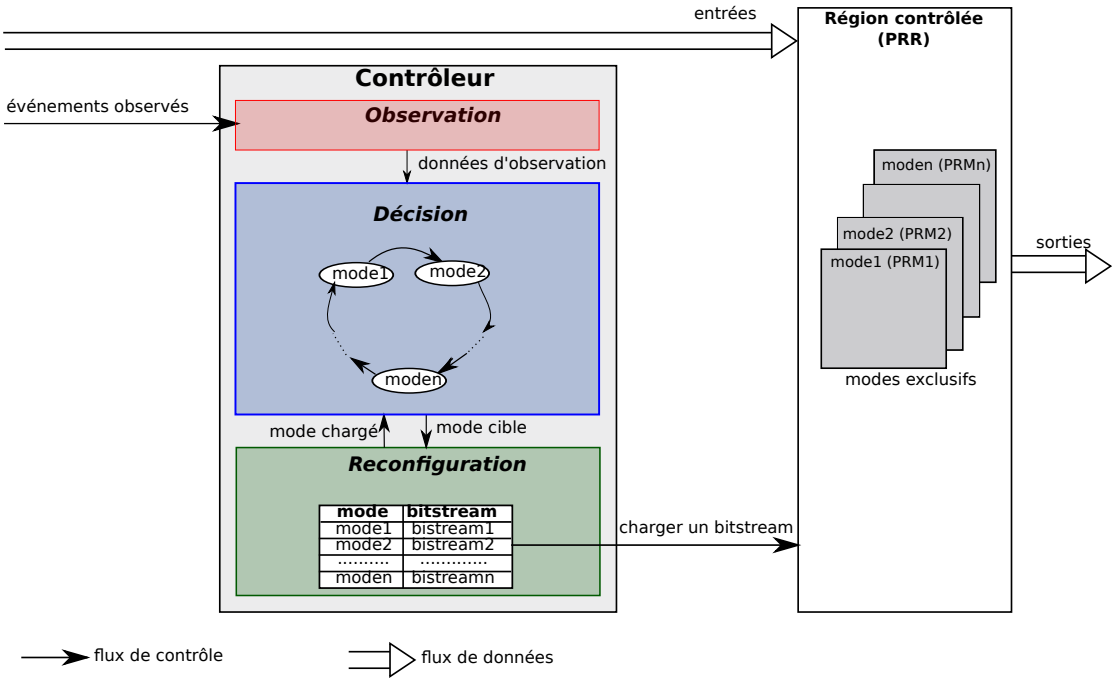


FIG. 4.2 – Modèle de séparation contrôle/données

4.4.1.2 Les actions liées aux transitions

Comme nous avons mentionné dans la section 3.4, dans la version originale des automates de modes, les sorties/actions sont seulement associées aux modes. L'association des sorties/actions aux transitions a été considérée comme une extension éventuelle en cas de besoin [73]. Dans notre modèle de prise de décision, les décisions des contrôleurs sont coordonnées pour respecter les contraintes globales du système. Pour cette raison,

[1] Comme nous avons précisions dans le chapitre 2, une PRR (Partial Reconfigurable Region) peut avoir plusieurs implémentations possibles ou PRMs

chaque contrôleur est amené à effectuer des actions et de produire des sorties liées à la coordination. Ces actions ne peuvent pas être liées aux modes puisqu'elles dépendent non seulement du mode courant mais aussi des événements en entrée. Pour ceci, nous avons ajouté à la version originale des automates de modes la possibilité d'associer des sorties/actions aux transitions.

Comme précisé dans la section 3.4, les automates de modes peuvent être composés d'une façon parallèle ou hiérarchique. Le modèle de prise de décision proposé dans ce travail est basé sur des automates de modes parallèles. Il est composé des automates des contrôleurs distribués et de l'automate du coordinateur. Ces automates communiquent en échangeant des informations de coordination (requêtes de reconfiguration, acceptation, refus, etc) à chaque fois qu'un contrôleur estime que la reconfiguration de la région qu'il contrôle est requise. Dans ce qui suit, nous détaillons la structure de ces automates et les interactions entre eux.

4.4.2 Les automates de modes des contrôleurs

Chaque module de décision d'un contrôleur est modélisé par un automate de modes. La figure 4.3 montre un exemple d'automate de modes nommé $decision\_controleur_i$. Chaque $Mode_{ij}$ correspond à une configuration/mode de la région contrôlée $rgion_i$ (avec $i \in [1..n]$; n est le nombre des régions reconfigurables du système ; $j \in [1..M_i]$; et M_i est le nombre des modes/configurations de $rgion_i$). Ici, on suppose que chaque région a un ensemble de possibilités de configuration qui est définit durant la phase de conception. Dans l'exemple de la figure 4.3, la région contrôlée a deux modes possibles. Comme le montre la figure, pour quitter un mode donné, il faut que le module de décision reçoive une notification ($loaded\_mode_i$) du module de contrôle lui indiquant que la configuration (bitstream) correspondante au mode cible est chargée dans la région contrôlée. Pour toutes les autres transitions, les modes source et cible sont identiques. Ces transitions servent à gérer la communication avec le coordinateur. Les transitions représentées à gauche des modes, dans la figure 4.3, sont liées aux requêtes et réponses envoyées au coordinateur. Les transitions à droite sont liées à la gestion des décisions envoyées par le coordinateur. La séparation entre les conditions et les actions de chaque transition est faite par un "/" dans cette figure et le reste de ce manuscrit.

4.4.2.1 Les entrées et sorties des automates

Les entrées et sorties de l'automate de modes sont présentées dans son entête.

Les entrées

Les entrées de l'automate de modes sont :

- les données d'observation ($monitoring\_data_i$) envoyées par le module d'observation,
- la notification après le chargement de la configuration cible envoyé par le module de reconfiguration ($loaded\_mode_i$),
- les informations de coordination envoyées par le coordinateur

Les informations de coordination incluent :

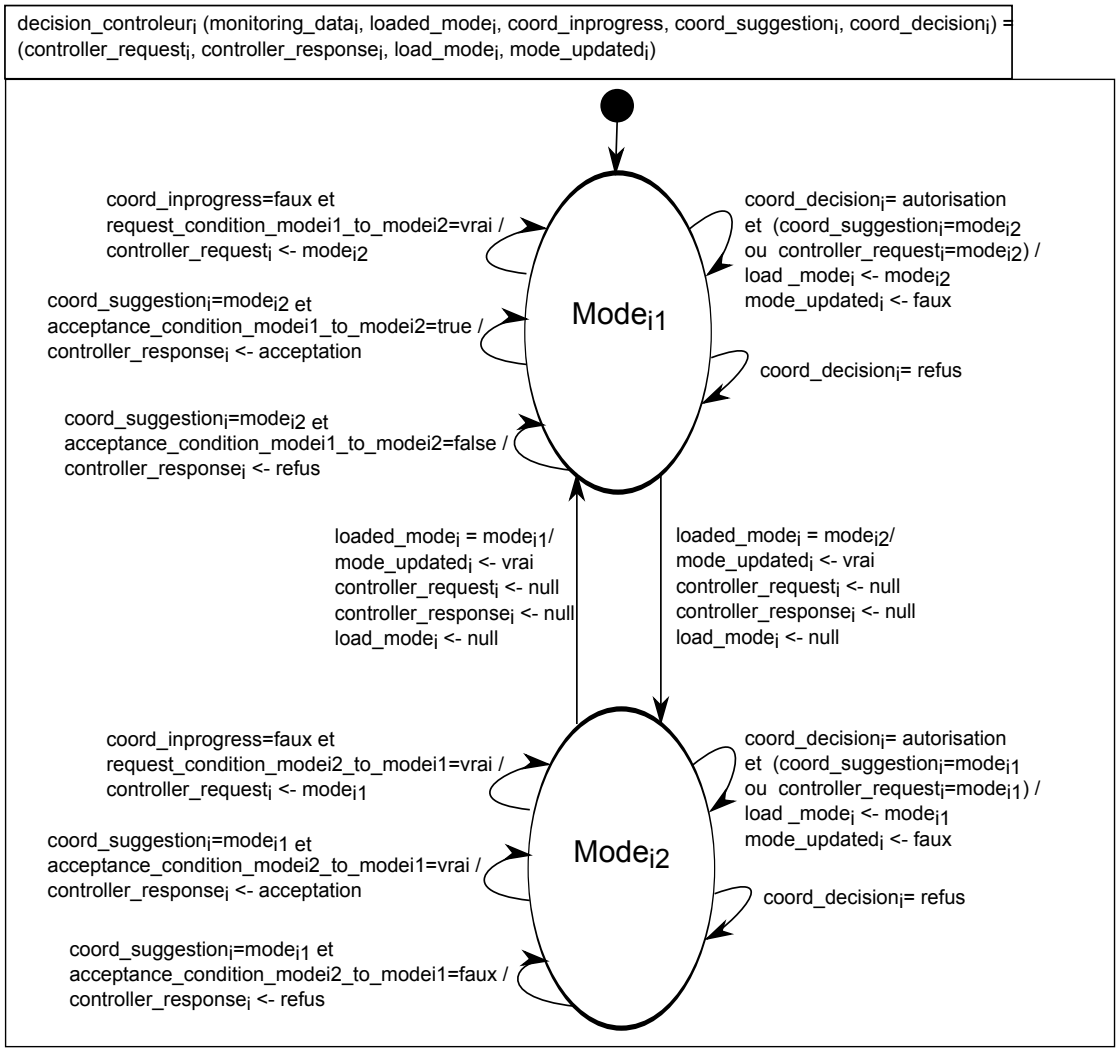


FIG. 4.3 – L'automate de modes d'un contrôleur

- la notification envoyée par le coordinateur au début et à la fin d'un processus de coordination (*coord_inprogress*) afin que les contrôleurs n'envoient pas de requêtes quand le coordinateur est en train de traiter une autre,
- les propositions de reconfiguration ($coord\_suggestion_i$) envoyées par le coordinateur,
- la décision du coordinateur ($coord\_decision_i$) à la fin d'un processus de coordination dans lequel le contrôleur est impliqué

Les sorties

Les sorties de l'automate de modes sont :

- la requête de reconfiguration ($controller\_request_i$) envoyée au coordinateur,
- la réponse à la proposition du coordinateur ($controller\_response_i$),
- la commande de reconfiguration au module de reconfiguration ($load\_mode_i = mode_{ij}$),
- la notification du coordinateur après l'exécution de la reconfiguration par le module de reconfiguration ($mode\_updated_i$)

4.4.2.2 Envoi des requêtes de reconfiguration

En se basant sur les données d'observation et le mode courant de la région contrôlée, si le contrôleur décide que sa région contrôlée a besoin d'être reconfigurée et qu'il n'y a pas de processus de coordination en cours ($coord\_inprogress = faux$), il envoie une requête de reconfiguration ($controller\_request_i$) au coordinateur. Par exemple, dans la figure 4.3, les condition qui doivent être valides pour décider qu'un passage d'un mode $mode_{ij}$ à un mode $mode_{ik}$ sont modélisées par $request\_condition\_mode_{ij}\_to\_mode_{ik}$. Ces conditions sont des expressions booléennes basées sur les données d'observation.

4.4.2.3 Réponses aux propositions de reconfiguration

Durant un processus de coordination, le contrôleur peut recevoir une proposition de reconfiguration envoyée par le coordinateur ($coord\_suggestion_i$), à laquelle il peut répondre avec une acceptation ou un refus ($controller\_response_i = acceptation/refus$). Pour déterminer cette réponse, le contrôleur se base sur ses données d'observation et sa stratégie de contrôle. Les conditions d'acceptation ou de refus du passage d'un mode $mode_{ij}$ à un mode $mode_{ik}$ sont représentées par la variable booléenne $acceptance\_condition\_mode_{ij}\_to\_mode_{ik}$.

4.4.2.4 Le traitement des décisions du coordinateur

Si la décision du coordinateur est l'autorisation de la reconfiguration de la région contrôlée par le contrôleur ($coord\_decision_i = autorisation$), ce dernier envoie une commande de reconfiguration au module de reconfiguration ($load\_mode_i = mode_{ij}$) indiquant que le $mode_{ij}$ doit être chargé. Si le coordinateur refuse la reconfiguration ($coord\_decision_i = refus$), elle ne peut pas être lancée. Après le chargement du bistream par le module de reconfiguration, celui-ci notifie le module de décision en indiquant le nouveau mode chargé ($loaded\_mode_i$). Dans ce cas, le module de décision notifie le

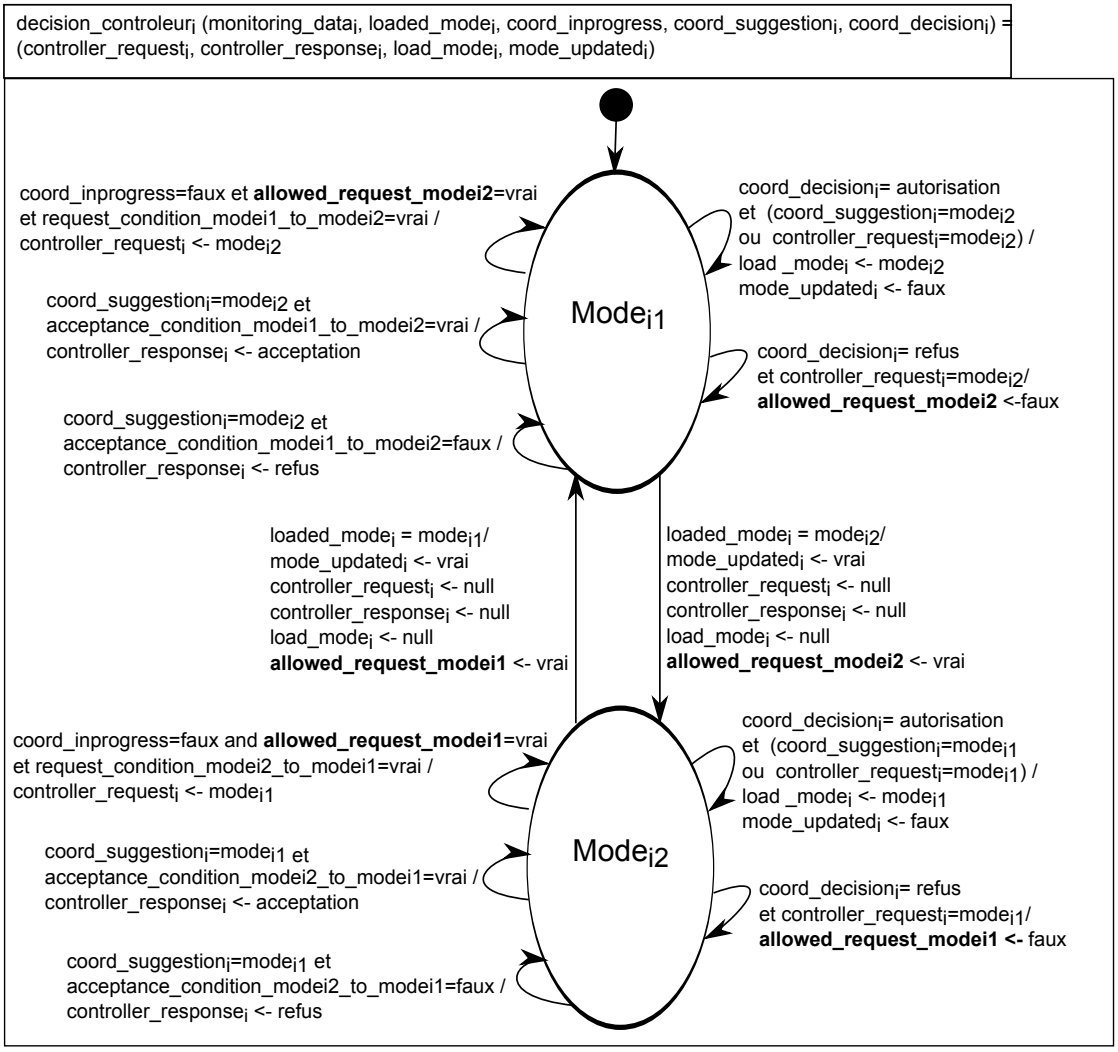


FIG. 4.4 – Gestion des refus de reconfiguration du coordinateur (version 1)

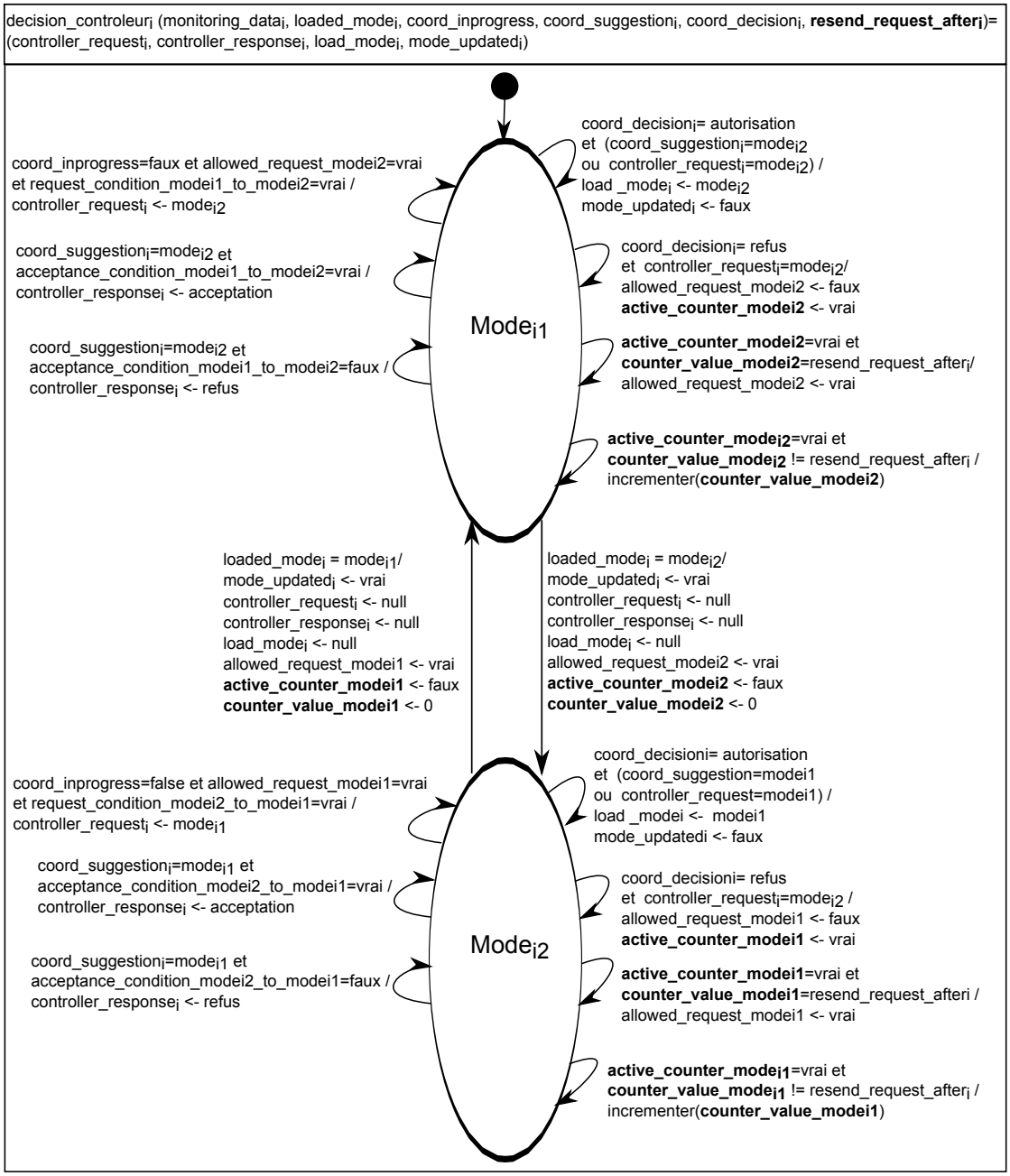


FIG. 4.5 – Gestion des refus de reconfiguration du coordinateur (version 2)

coordinateur (*mode_updated* = *vrai*), qui est en attente des notifications des contrôleurs concernées par un processus de coordination afin de mettre fin à celui-ci.

Comme on peut remarquer dans la figure 4.3, si une reconfiguration est refusée par le coordinateur et que les conditions d'envoi d'une requête de reconfigurations sont toujours valides, le contrôleur va envoyer encore une fois cette requête. Ceci peut entraîner un échange de messages inutiles sans arriver à une autorisation du coordinateur tant que les conditions d'acceptation des autres contrôleurs ne sont pas valides. Pour éviter ce problème, deux solutions peuvent être proposées. La première est que si une requête envoyée par un contrôleur a été refusée par le coordinateur, ce contrôleur n'envoie plus cette requête. Dans ce cas, il peut aller dans le mode désiré s'il reçoit une proposition par le coordinateur impliquant ce mode et que le processus de coordination se termine par l'autorisation de la reconfiguration. Cette solution est illustrée dans la figure 4.4. Les variables locales *allowed_request_mode*ij permettent de dire si une requête de reconfiguration vers un *mode*ij est permise. Ces variables prennent la valeur $faux$ si la requête a été refusée, ce qui fait que le contrôleur n'envoie plus cette requête tant qu'il n'a pas changé de mode courant. Ces variables sont réinitialisées à *vrai* lors du passage d'un mode à un autre comme le montre la figure 4.4.

Vu la dynamicité des systèmes reconfigurables, il se peut qu'une reconfiguration refusée à un moment donné pourrait être acceptée ultérieurement parce qu'il y a eu un changement de l'environnement ou de l'état interne du système favorable à cette reconfiguration. Pour gérer ce cas, notre deuxième solution est plus générique que la première. Elle offre la possibilité d'envoyer de nouveau la requête après un certain laps de temps. Cette solution est illustrée par la figure 4.5. Le laps de temps est modélisé par $resend\_request\_after_i$ qui est une entrée de l'automate et qui peut être un paramètre de celui-ci. Dans le cas où $resend\_request\_after_i$ est égal à 0, on est dans le cas de la première solution où un contrôleur n'envoie plus la même requête si elle a été refusée. Pour cette deuxième solution, on propose des variables locales pour la gestion des laps de temps. Les variables booléennes *active_counter_mode*ij permettent d'activer des compteurs qui permettent de vérifier si un laps de temps est écoulé. Ces compteurs sont représentés par les variables *counter_value_mode*ij comme le montre la figure 4.5. Une variable *allowed_request_mode*ij passe de la valeur $faux$ à la valeur *vrai* si le laps de temps $resend\_request\_after_i$ est écoulé après un refus d'une requête de reconfiguration envoyée par le contrôleur. Dans ce cas, il est possible d'envoyer de nouveau une requête qui a été refusée. Le choix du laps de temps est à la charge du concepteur et dépend de sa connaissance du système et de sa dynamicité.

4.4.3 L'automate de modes du coordinateur

L'automate du coordinateur, comme le montre la figure 4.6 contient 5 modes correspondant à différentes phases liées au processus de coordination. Nous avons choisi la représentation du coordinateur sous forme d'automate afin d'avoir un modèle de prise de décision homogène. La décomposition en modes permet de faciliter la conception et la modification du coordinateur.

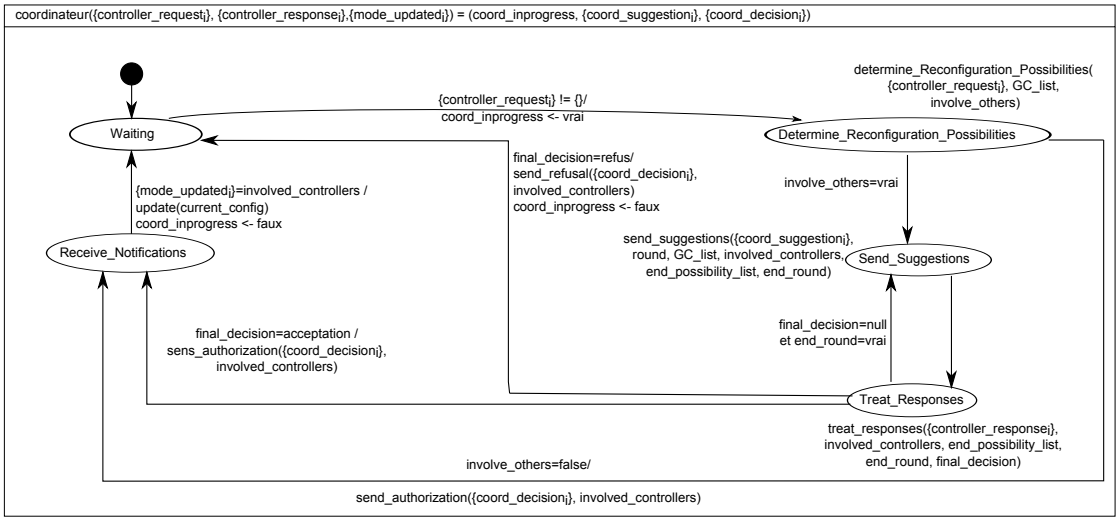


FIG. 4.6 – L'automate de modes du coordinateur

4.4.3.1 Les entrées et sorties de l'automate

Les entrées et sorties de l'automate du coordinateur sont présentées dans son entête.

Les entrées

Les entrées de l'automate sont :

- les requêtes des contrôleurs ($\{controller\_request_i\}$). Ici, la notation {} désigne un ensemble,
- les réponses des contrôleurs aux propositions ($\{controller\_response_i\}$),
- les notifications reçues des contrôleurs après l'exécution des reconfigurations ($\{mode\_updated_i\}$).

Les sorties

Les sorties de l'automate sont :

- la notification envoyée aux contrôleurs au début et à la fin d'un processus de coordination ($coord\_inprogress$),
- les propositions de reconfiguration ($coord\_suggestion_i$) envoyées aux contrôleurs,
- la décision du coordinateur ($\{coord\_decision_i\}$) envoyée aux contrôleurs à la fin d'un processus de coordination.

4.4.3.2 Le processus de coordination géré par l'automate de mode

Le mode *Waiting* correspond à un coordinateur en attente des requêtes des contrôleurs. Si le coordinateur reçoit une requête, il envoie une notification ($coord\_inprogress = vrai$) à tous les contrôleurs leur indiquant qu'il y a une coordination en cours. Il passe ensuite au mode $Determine\_Reconfiguration\_Possibilities$ qui permet de déterminer à travers l'action $determine\_Reconfiguration\_Possibilities$ ($\{controller\_request_i\}$, GC_list, $involve\_others$), les éventuelles reconfigurations requises pour les autres régions contrôlées. Cette action permet donc de garantir une configuration globale qui satisfait la requête et qui respecte en même temps les contraintes/objectifs du système. Dans le

cas où plus d'une configuration du système vérifient ces deux conditions, les différentes possibilités sont mises dans une liste ordonnée. Les entrées de cette action sont donc les requêtes des contrôleurs ($\{controller\_request_i\}$) et ses sorties sont la liste des possibilités de configurations satisfaisant la requête (*GC_list*) et la détermination du fait que la coordination implique ou pas la reconfiguration d'autres régions (*involve_others*).

Si la requête reçue n'implique pas la reconfiguration d'autres régions ($involve\_others = faux$), le coordinateur envoie son autorisation au contrôleur qui a envoyé la requête à travers l'action (*send_authorization* ($\{coord\_decision_i\}$, *involved_controllers*)). Sinon ($involve\_others = vrai$), le coordinateur passe au mode $Send\_Suggestions$ et envoie les propositions aux contrôleurs concernés à travers l'action $send\_suggestions(\{coord\_suggestion_i\}, round, GC\_list, involved\_controllers, end\_possibility\_list, end\_round)$. Ici, la variable locale *round* indique le numéro du tour de coordination considéré. En fait, si plus d'une configuration du système satisfont la requête et les contraintes/objectifs gérés par le coordinateur, le processus de coordination est divisé en un ensemble de tours ou chaque tour est lié à une des configuration considérée de la liste ordonnée *GC_list*. La variable *involved_controllers* est un tableau de booléens dont la taille est le nombre des contrôleurs. Elle permet d'indiquer les contrôleurs qui sont impliqués dans un tour de coordination. Le contenu de la variable *involved_controllers* est mis à jour en affectant la valeur *vrai* aux éléments dont les indices correspondent au contrôleur qui a envoyé la requête et les autres contrôleurs impliqués dans le tour de coordination courant. La variable *end_possibility_list* permet de déterminer si toute la liste *GC_list* a été parcourue. La variable *end_round* est une sortie qui détermine si un tour de coordination est fini. Cette variable est mise à $faux$ par l'action *send_suggestions* puisqu'il s'agit du début du tour. Elle prendra la valeur *vrai* après le traitement des réponses des contrôleurs comme nous allons voir dans la suite de cette section.

Après avoir envoyé les propositions, le coordinateur passe au mode $Treat\_Responses$ dans lequel il traite les réponses des contrôleurs aux propositions et détermine sa décision. Dans l'action $treat_responses$ ($\{controller\_response_i\}$, *involved_controllers*, *end_possibility_list*, *end_round*, $final\_decision$), le paramètre $final\_decision$ est une sortie qui détermine si le traitement des réponses a donné lieu à une prise de décision finale. Le paramètre *end_round* est une sortie qui détermine si toutes les réponses ont été traitées. Le paramètre *end_possibility_list* est une entrée qui indique si toutes les possibilités ont été parcourues, ce qui conduit à la prise d'une décision de refus si jamais les réponses du dernier tour de coordination n'ont pas conduit à une décision d'autorisation de reconfiguration. Si dans un tour donné, après le traitement de toutes les réponses, la décision finale du coordinateur n'est pas encore prise, il retourne au mode $Send_Suggestions$ pour traiter la possibilité suivante dans la liste *GC_list*. Si la décision du coordinateur est d'autoriser la reconfiguration, il envoie l'autorisation (*send_authorization* ($\{coord\_decision_i\}$, *involved_controllers*)) aux contrôleurs impliqués pour qu'ils lancent la reconfiguration. Si la décision est le refus, elle est envoyée au contrôleur qui a envoyé la requête à travers l'action $send\_refusal$ ($\{coord\_decision_i\}$, *involved_controllers*).

Après avoir envoyé la décision, s'il s'agit de refus le coordinateur passe au mode *Waiting*, sinon il passe au mode $Receive\_Notification$ dans lequel il traite les notifications ($mode\_updated_i$) des contrôleurs après le chargement des bitstreams. S'il re-

Configurations globales / Régions reconfigurable	1	2	.....	K
Région1	mode1 j1.1	mode1 j1.2	.....	mode1 j1.K
Région2	mode2 j2.1	mode2 j2.2	.....	mode2 j2.K
.....	.....	.....	.....	.....
Régionn	moden jn.1	moden jn.2	.....	mode2 jn.K

FIG. 4.7 – La table GC du coordinateur

çoit des notifications de tous les contrôleurs auxquels il a autorisé la reconfiguration ($mode\_updated_i = involved\_controllers$), il met à jour sa vision de la configuration courante du système à travers l'action $update(current\_config)$, notifie aux contrôleurs la fin du processus de coordination ($coord\_inprogress = faux$) et passe au mode *Waiting*. Plus de détails sur le mécanisme de coordination sont présentés dans le reste de cette section.

4.5 Le mécanisme de coordination

Comme expliqué précédemment, le rôle du coordinateur est de coordonner les décisions de reconfiguration des contrôleurs afin de garantir que la configuration du système respecte les contraintes/objectifs globaux. Pour cela, le coordinateur dispose d'une table contenant les configurations permises du système. Cette table permet d'éliminer un ensemble de configurations qui ne sont pas permises/ faisables à cause de contraintes de sécurité, de qualité de service, etc. Cette table est construite à la phase de conception. Elle peut être écrite manuellement ou générée à partir d'une description des contraintes globales du système en utilisant par exemple un langage basé sur les contrats [31]. Dans notre travail, nous supposons que cette table est remplie manuellement par le concepteur. On l'appelle la table *GC* (Global Configurations). Elle est illustrée dans la figure 4.7. Chaque ligne de cette table est une combinaison des configurations partielles des régions reconfigurables. Elle est définie comme suit : $GC[i,k] = mode_{ij_{i.k}}\ \forall i \in [1..n]$, $j_{i.k} \in [1..M_i]$ et $k \in [1..K]$. $mode_{ij_{i.k}}$ est le mode de la $rgion_i$ dans la configuration globale k. n est le nombre de régions contrôlées. K est le nombre de configurations globales permises et M_i est le nombre de modes de la $rgion_i$. La détermination des configurations globales permises par le concepteur peut dépendre de différents objectifs et contraintes. Elle peut, par exemple, être basée sur l'optimisation de certains critères (performance, consommation, qualité de service, etc). Dans ce cas, la table *GC* contiendra, par exemple, un résultat de Pareto. Cette solution est très intéressante en vue de réduire le nombre de configurations globales possibles et donc d'accélérer le processus de coordination grâce à la réduction du nombre de configurations globales satisfaisant une requête de reconfiguration donnée. La détermination de la table *GC* peut être également basée sur des contraintes plus simples éliminant, par exemple, les configurations qui ne respectent pas les dépendances entre les granularités des tâches exécutées par les modules reconfigurables. Dans notre travail, nous ne nous concentrons pas sur la détermination de la

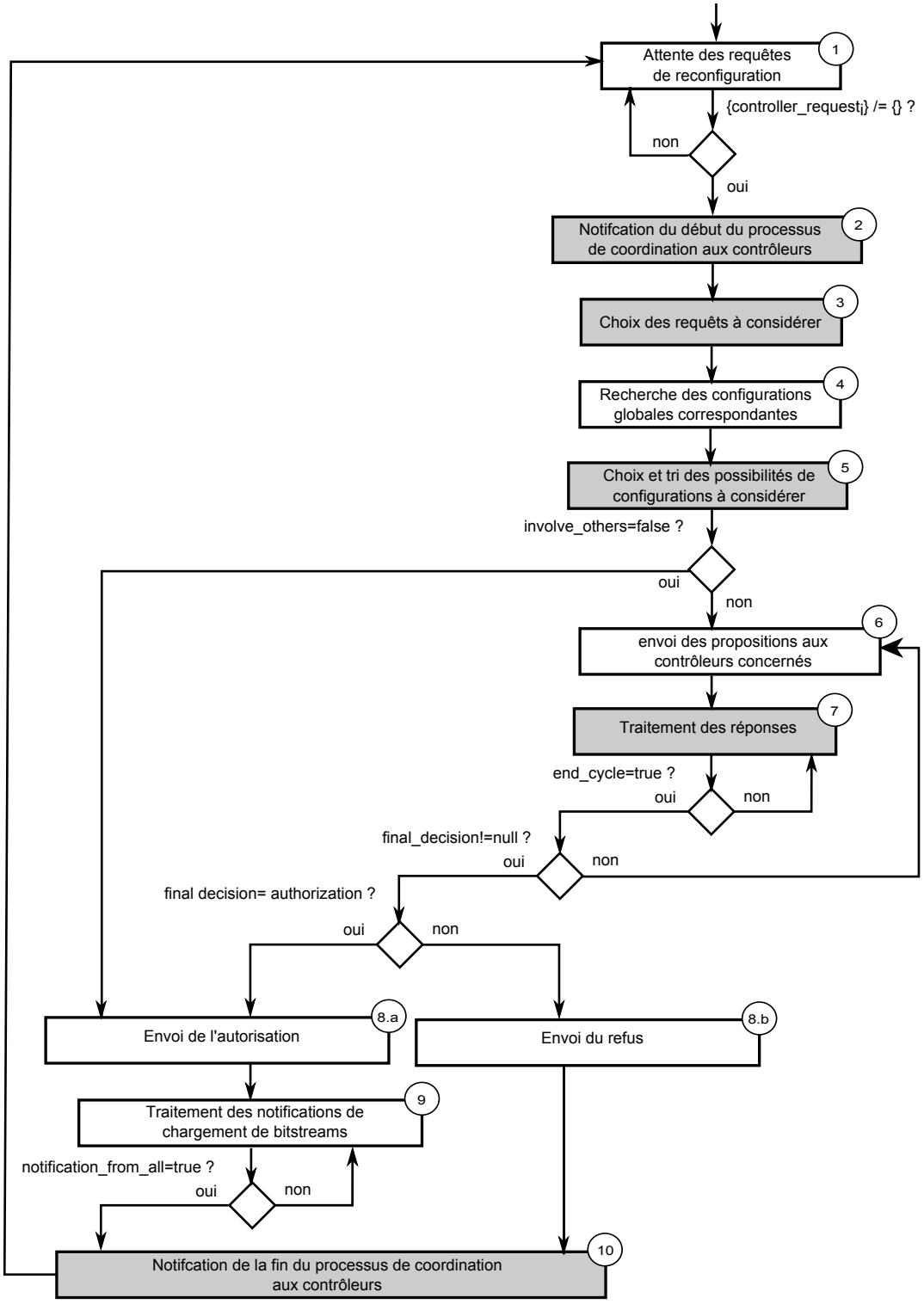


FIG. 4.8 – L'organigramme de l'algorithme de coordination

table *GC* qui peut suivre plusieurs méthodes, mais sur l'utilisation de cette table dans l'algorithme de coordination exécuté par le coordinateur.

4.5.1 La stratégie du coordinateur

L'automate de modes proposé pour le coordinateur, et illustré par la figure 4.6, permet de donner une vue abstraite de celui-ci et offre la possibilité aux actions de cet automate d'être implémentées de différentes manières selon la stratégie adoptée par le coordinateur. Dans cette section, on définit quelques stratégies possibles pour le coordinateur. La stratégie du coordinateur peut être déterminée principalement par 4 points stratégiques :

- l'autorisation/non-autorisation de l'envoi de requêtes au cours d'un processus de coordination,
- le traitement des requêtes reçues en même temps,
- le tri de(s) configuration(s) à considérer par le processus de coordination,
- le traitement des réponses aux propositions.

La différence du premier point par rapport au reste des points est qu'il a non seulement un impact sur l'automate de modes du coordinateur mais aussi sur ceux des contrôleurs, du fait que lorsque le coordinateur autorise aux contrôleurs d'envoyer des requêtes au cours d'un processus de coordination, les automates de modes ne prennent plus en compte la variable *coord_inprogress*. La définition de la stratégie du coordinateur sur ce point doit être donc prise en compte du côté des contrôleurs. Les trois points stratégiques qui restent n'implique pas de modification du côté des contrôleurs.

4.5.1.1 Autorisation/non-autorisation de l'envoi de requêtes au cours d'un processus de coordination

Pour une stratégie de coordinateur qui autorise aux contrôleurs d'envoyer des requêtes au cours d'un processus de coordination, ces requêtes peuvent être stockées dans une file pour être traitées une par une à la fin du processus de coordination courant. Cette implémentation a ses avantages et ses inconvénients. L'avantage est de permettre aux requêtes d'être considérées directement après la fin du processus de coordination courant, ce qui accélère la prise de décision. L'inconvénient est que cela pourrait impliquer la sauvegarde de requêtes inutiles. En effet, il y a une possibilité que ces requêtes soient liées à des régions dont les modes courants changeraient durant le processus de coordination courant, ce qui pourrait entraîner des situations où même les conditions locales des contrôleurs qui ont envoyé les requêtes ne sont plus favorables aux reconfigurations correspondantes. Cette autorisation pourrait aussi avoir un impact non négligeable sur le nombre de ressources utilisées pour 1) la gestion parallèle des propositions/réponses et des requêtes et 2) le stockage des requêtes. La non-autorisation de la réception de requêtes au cours d'un processus de coordination permet de réduire la complexité du coordinateur et sa consommation en ressources. Avec cette implémentation, les contrôleurs qui ont estimé, au cours du processus de coordination courant, que la reconfiguration de leurs régions était nécessaire et qui n'ont pas changé d'avis après la fin de ce processus enverront leurs requêtes qui seront traitées dans les processus de

coordination suivants.

4.5.1.2 Traitement des requêtes reçues en même temps

Deux requêtes reçues en même temps sont deux requêtes envoyées au coordinateur lorsqu'il est dans le mode inactif *Waiting*. Dans ce cas, la stratégie du coordinateur peut être de traiter toutes les requêtes ou de choisir une parmi elles. Le premier choix peut impliquer deux solutions possibles. La première est de ne considérer que les configurations globales qui satisfont en même temps toutes les requêtes. La deuxième est de considérer les configurations globales qui satisfont au moins une requête. Cette deuxième solution donne plus de flexibilité. En effet, selon la stratégie du coordinateur pour trier les possibilités de configurations à considérer dans un processus de coordination, une configuration qui ne satisfait pas toutes les requêtes peut être plus prioritaire que celles qui les satisfont toutes. Dans ce cas, les contrôleurs qui ont envoyé des requêtes qui ne sont pas satisfaites par cette configuration auront des propositions leur suggérant des modes différents de ceux requis. Les contrôleurs pourraient accepter ces propositions, ce qui permet d'aller dans une configuration qui convient plus pour le coordinateur que celles qui satisfont toutes les requêtes.

Dans le cas où la stratégie du coordinateur est de ne traiter qu'une seule requête lorsque plusieurs requêtes sont reçues en même temps. Cette requête peut être choisie aléatoirement ou selon une priorité des contrôleurs. Différentes façons d'attribution des priorités peuvent être utilisées. Par exemple, les priorités peuvent être attribuées selon les priorités des tâches exécutées par les régions contrôlées. Dans un algorithme équitable, la priorité d'un contrôleur peut aussi diminuer si sa requête a été traitée dans un processus de coordination précédent. On peut aussi attribuer les priorités selon l'indice du contrôleur. D'autres mécanismes de priorité peuvent aussi être implémentés.

4.5.1.3 Tri de(s) configuration(s) à considérer par le processus de coordination

Dans une requête de reconfiguration envoyée au coordinateur, le contrôleur indique la configuration/mode cible pour sa région. En considérant cette requête et selon sa stratégie, le coordinateur décide si cette reconfiguration peut être autorisée directement ou qu'elle nécessite la reconfiguration d'autres régions afin de respecter les contraintes/objectifs globaux du système. Dans ce cas, une ou plusieurs configurations globales de la table *GC* peuvent contenir le mode cible de la requête. Le coordinateur doit donc choisir une configuration globale ou bien trier les configurations globales possibles selon un ordre de priorité donné.

La sélection des configurations globales à considérer dans un processus de coordination dépend de la stratégie du coordinateur. Par exemple, si le but du coordinateur est de minimiser le nombre de reconfigurations, dans le cas où une des configurations possibles n'implique aucune reconfiguration des régions contrôlées à part celle concernée par la requête, le coordinateur ne considère que cette possibilité qui permet de minimiser le nombre de reconfigurations et autorise directement la reconfiguration. Dans le cas où la reconfiguration de certaines autres régions est nécessaire pour satisfaire la requête, les configurations globales possibles sont triées en donnant la plus haute priorité à celle

qui demande le petit nombre de reconfiguration. Le coordinateur peut aussi utiliser une double sélection. Par exemple, il peut commencer par sélectionner les configurations globales qui permettent une qualité de service supérieure à un seuil donné et les trier ensuite pour minimiser le nombre de reconfigurations.

D'autres algorithmes de sélection sont aussi possibles tels que ceux basés sur l'optimisation multi-critères. Par exemple, si le but du coordinateur est d'avoir une configuration globale qui optimise certains critères (qualité de service, performance, consommation, etc), une solution basée sur l'optimum de Pareto est convenable. Dans ce cas, on peut commencer par éliminer les combinaisons de modes qui ne conviennent pas pour une application donnée ou qui ne respectent pas une contrainte donnée pour construire la table *GC*, puis on applique sur la liste des configurations qui restent un Pareto. Pour cela, il faut indiquer pour chaque configuration de cette liste son poids selon les différents critères considérés dans l'optimisation. Le Pareto permet de déterminer les configurations optimales (celles qui n'ont aucune configuration qui est mieux qu'elles pour tous les critères considérés) et les configurations sous-optimales. Le résultat du Pareto peut être exploité de deux manière. La première solution est de ne retenir que les configurations optimales et de mettre à jour le contenu de la table *GC* en ne mettant que ces configurations. Dans ce cas, seules les configurations optimales seront implémentées à l'exécution. Une deuxième solution, qui est plus flexible, est de garder les configurations optimales et sous-optimales (ou bien garder seulement une partie des solutions sous-optimales à côté des solutions optimales) et de faire la sélection à l'exécution. Le résultat du Pareto peut être stocké dans un tableau ou autre structure de données qui indique pour chaque combinaison celles qui l'optimisent (qui sont mieux qu'elle pour tous les critères considérées). Ce résultat est utilisé par le coordinateur à l'exécution, à côté de la table *GC*. Un Pareto dynamique peut être appliqué à la réception d'une requête. Dans ce cas, le coordinateur détermine parmi les configurations globales satisfaisant cette requête, celles qui se trouvent sur la frontière de Pareto dynamique (celles qui ne sont pas optimisées par aucune configuration parmi celles qui satisfont la requête). Cette frontière est déterminée en utilisant le résultat du Pareto statique. Le résultat du Pareto dynamique peut être traité de deux manières : 1) ne considérer que les configurations de la frontière du Pareto dynamique ou 2) considérer toutes les configurations satisfaisant la requête et les trier selon les nombres des configurations qui les optimisent. Dans les deux solutions, la priorité est donnée aux configurations qui se trouvent sur la frontière du Pareto, mais la deuxième solution est moins restrictive en permettant aux configurations sous-optimales d'être chargées si les optimales n'ont pas eu d'avis favorables auprès des contrôleurs.

4.5.1.4 Traitement des réponses aux propositions

La détermination de la décision finale (autorisation ou refus de la reconfiguration) à partir des réponses des contrôleurs peut suivre différents algorithmes selon l'application implémentée. Par exemple, une reconfiguration peut être autorisée si un pourcentage donné des réponses est positif. Les contrôleurs peuvent aussi avoir leurs poids dans l'autorisation de la reconfiguration. Dans ce cas, on peut par exemple exiger que la reconfiguration soit acceptée par un ensemble de contrôleurs de poids forts.

D'autres implémentations sont aussi possibles pour les 4 points présentés. Nous n'avons cité que quelques unes. L'automate de modes du coordinateur peut être donc adapté selon les implémentations choisies pour ces 4 points. Dans ce travail, une seule implémentation a été retenue pour chaque point, afin de construire l'algorithme de coordination qui sera utilisé pour la validation du modèle de contrôle semi-distribué. Cet algorithme est présenté dans la section suivante.

Algorithm 1 Extrait de l'algorithme de coordination lié à la réception des requêtes

```
1: switch current_mode do
2:     case Waiting
3:         if controller_request ≠ {} then
4:             coord_inprogress ← vrai
5:             current_mode ← Determine_Reconfiguration_Possibilities
6:         end if
7:     .....
```

4.5.2 L'algorithme de coordination

Les échanges entre les contrôleurs et le coordinateur ne sont pas continus dans le temps. Ils se passent seulement si l'un des contrôleurs décide que la reconfiguration de sa région contrôlée est requise. Cela permet de diminuer l'impact de la communication liée au contrôle sur la performance globale. Pour mieux comprendre le mécanisme de coordination, nous utilisons l'organigramme de la figure 4.8 qui donne plus de détails sur ce mécanisme en traitant l'ensemble des activités correspondantes à chacune des actions de l'automate de la figure 4.6. Ces activités sont représentées par des boîtes. Les numéros sur ces boîtes indiquent les numéros des activités dans le processurs de coordination. Les boîtes blanches représentent les activités de base qui ne dépendent pas de la stratégie du coordinateur. Les boîtes grises représentent les activités dont l'implémentation dépend de la stratégie du coordinateur. Différentes implémentations sont possibles pour ces dernières dont certaines sont présentées dans la section 4.5.1. Dans cette section, nous présentons l'algorithme de coordination selon les implémentations choisies pour ces activités.

Réception des requêtes

Cette partie concerne les activités 1), 2) de l'organigramme de la figure 4.8 qui correspondent respectivement au mode *Waiting* et l'action *coord_inprogress = vrai* de la transition entre le mode *Waiting* et *Determine_Reconfiguration_Possibilities* de la figure 4.6. Au début de l'algorithme, le coordinateur attend les requêtes de reconfiguration (activité 1). Une fois qu'il reçoit une requête, selon la stratégie qu'il implémente, le coordinateur peut autoriser ou non l'envoi d'autres requêtes par les contrôleurs comme précisé dans la section 4.5.1. Dans ce travail, nous proposons la non-autorisation de l'envoi au cours du processus de coordination. Cette solution permet de réduire le nombre de ressources consommées pour sauvegarder les requêtes qui ne seraient, peut être, plus valides à la fin du processus comme précisé dans la

Algorithm 2 Extrait de l'algorithme de coordination lié au choix de la requête à considérer

```
1: case Determine_Reconfiguration_Possibilities
2:     //Choix des requêtes à considérer
3:     for i = 1 to n do
4:         if controller_request(i) ≠ null then
5:             request_controller_index ← i
6:             exit
7:         end if
8:     end for
9:     requested_mode ← controller_request(request_controller_index)
10:    .....
```

section 4.5.1. A la réception d'une requête, le coordinateur notifie donc aux contrôleurs qu'un processus de coordination est en cours (activité 2) pour qu'ils n'envoient pas de requêtes jusqu'à la fin de ce processus. L'algorithme 1 donne l'extrait lié à la réception des requêtes. Dans cet algorithme, la variable *current_mode* indique le mode courant du coordinateur. Le reste des variables sont celles utilisées dans la figure 4.6 de l'automate.

Algorithm 3 Extrait de l'algorithme de coordination lié à la détermination des possibilités de reconfiguration

```
1: //Recherche des configurations globales satisfaisant la requête
2: GC_list ← {}
3: nb_possibilities ← 0
4: for i = 1 to K do
5:     if GC_table(request_controller_index)(i) = requested_mode then
6:         GC_list(nb_possibilities) ← i
7:         nb_possibilities ← nb_possibilities + 1
8:     end if
9: end for
```

Détermination des reconfigurations considérées dans le processus de coordination

Cette partie concerne les activités 3), 4) et 5) de l'organigramme de la figure 4.8 qui sont une décomposition de l'action *determine_Reconfiguration_Possibilities* ($\{controller\_request_i\}$, *GC_list*, *involve_others*) du mode *Determine_Reconfiguration_Possibilities* de la figure 4.6. Après la notification, s'il s'agit de plusieurs requêtes reçues en même temps (ceci dépend du protocole de communication implémenté entre les contrôleurs et le coordinateur), la requête à traiter peut être choisie (activité 3) aléatoirement ou selon une priorité des contrôleurs. Différentes façon d'attribution des priorités peuvent être utilisées comme nous l'avons précisé au début de la section 4.5.1. L'implémentation proposée dans ce travail est de donner la priorité au contrôleur ayant l'indice le plus petit parmi ceux qui ont envoyé une requête. Cette implémentation est illustrée par l'algorithme 2. Dans cet algorithme,

la variable *n* correspond au nombre de contrôleurs et la variable *request_controller_index* permet de stocker l'indice du contrôleur dont la requête va être traitée.

Algorithm 4 Extrait de l'algorithme de coordination lié au choix des possibilités de reconfiguration à considérer

```
1: nb_reconfig ← {}
2: //Calcul du nombre de reconfigurations nécessaires pour chaque possibilité
3: for i = 1 to nb_possibilities do
4:     for j = 1 to n do
5:         if current_config(j) ≠ GC_table(j)(GC_list(i)) then
6:             nb_reconfig(i) ← nb_reconfig(i) + 1
7:         end if
8:     end for
9: end for
10:
11: //Tri des possibilités de configuration
12: for i = 1 to nb_possibilities do
13:     for j = 1 to nb_possibilities do
14:         if nb_reconfig(i) < nb_reconfig(j) et i < j then
15:             temp1 ← GC_list(i)
16:             temp2 ← nb_reconfig(i)
17:             GC_list(i) ← GC_list(j)
18:             nb_reconfig(i) ← nb_reconfig(j)
19:             GC_list(j) ← temp1
20:             nb_reconfig(j) ← temp2
21:         end if
22:     end for
23: end for
```

Le coordinateur cherche ensuite toutes les configurations globales qui correspondent à la requête considérée (activité 4). Si nous considérons que la configuration globale courante est la numéro 1 de la table de la figure 4.7 et que le coordinateur reçoit une requête du contrôleur $Controller_1$ qui cible le $mode_{1j_{1.k1}}$, les configuration globales qui satisfont cette requête correspondent aux colonnes contenant le $mode_{1j_{1.k1}}$. L'algorithme 3 donne l'extrait de l'algorithme de coordination lié à cette partie. La variable *GC_list* est utilisée pour stocker la liste des possibilités de reconfigurations sous forme d'un tableau d'indices, qui sont les indices des configurations globales satisfaisant la requête dans la table *GC* du coordinateur. La table *GC* est représentée par la variable *GC_table*. La variable *K* représente le nombre de configurations globales de la table *GC*.

Après avoir déterminé la liste des configurations globales correspondant à la requête, selon la stratégie du contrôle, le coordinateur détermine la liste des configurations *GC_list* qui vont être considérées durant le processus de coordination (activité 5). La stratégie du coordinateur dépend de l'application implémentée et de l'objectif du contrôle. De ce fait, plusieurs algorithmes peuvent être appliqués pour déterminer la liste *GC_list*. Par exemple, si le but du coordinateur est d'avoir une configuration

Algorithm 5 Extrait de l'algorithme de coordination lié au traitement des autorisations directes de reconfiguration

```
1: if nb_reconfig(0) = 1 then
2:     involve_others ← faux
3: else
4:     involve_others ← vrai
5: end if
6:
7: if involve_others = vrai then
8:     coord_decision(request_controller_index) ← autorisation
9:     current_mode ← Receive_Notifications
10: else
11:     current_mode ← Send_Suggestions
12: end if
```

globale qui optimise certains critères (qualité de service, performance, consommation, etc), une solution basée sur l'optimum de Pareto est convenable comme précisé dans la section 4.5.1. La stratégie du coordinateur implémentée dans ce travail n'est pas basée sur un problème d'optimisation complexe. Elle vise juste à minimiser le nombre de reconfigurations vu leur impact sur la performance et la consommation. Dans ce cas, la liste des configurations satisfaisant la requête est triée selon le nombre de reconfigurations impliquées en donnant la priorité à celles qui demandent moins de reconfigurations comme le montre l'algorithme 4. La variable *nb_reconfig* est un tableau d'entiers qui permet de stocker le nombre de reconfigurations requises pour chaque possibilité de la liste *GC_list*, puisque nous trions les possibilités selon le nombre de reconfigurations requises. La variable *current_config* est un tableau indiquant le mode courant de chaque contrôleur, ce qui donne la configuration globale courante du système.

Envoi des propositions de reconfiguration et traitement des réponses

Cette partie concerne les activités 6) et 7) de l'organigramme de la figure 4.8 qui correspondent respectivement aux actions *send_suggestions* ({$coord\_suggestion_i$}, *round*, *GC_list*, *involved_controllers*, *end_possibility_list*, *end_round*) et $treat_r esponses$ ({$controller\_response_i$}, *involved_controllers*, *end_possibility_list*, *end_round*, *final_decision*) des modes *Send_Suggestions* et *Treat_Responses* de la figure 4.6. A partir de la liste de possibilités retenues, deux cas de figures peuvent se présenter. Le premier cas est que la requête de reconfiguration peut coexister avec les configurations courantes des autres régions (cette nouvelle configuration du système satisfait bien les contraintes/objectifs gérés par le coordinateur). Dans ce cas, aucun autre contrôleur n'est impliqué dans le processus de coordination (*involve_others* = *faux*). Le coordinateur passe donc à l'activité de l'envoi de sa décision que nous décrivons plus loin dans ce chapitre. Dans le deuxième cas (*involve_others* = *vrai*), il passe à l'envoi des propositions de reconfiguration aux contrôleurs. L'algorithme 5 illustre les deux cas de figure. Ici, puisque les possibilités sont triées selon le nombre de reconfigurations, si la première possibilité requiert une seule reconfiguration, cela signifie que pour cette possibilité

Algorithm 6 Extrait de l'algorithme de coordination lié à l'envoi des propositions de reconfiguration

```
1: case Send_Suggestions
2:     //Réinitialisation des variables internes
3:     nb_involved_controllers ← 1
4:     involved_controllers ← {faux, faux, ....., faux}
5:     involved_controllers(request_controller_index) ← vrai
6:     end_round ← faux
7:     //Vérifier la fin de la liste des possibilités
8:     round ← round + 1
9:     if round = nb_possibilities then
10:        end_possibility_list ← vrai
11:    else
12:        end_possibility_list ← faux
13:    end if
14:    //Envoi des propositions aux contrôleurs concernés
15:    for i = 1 to n do
16:        if current_config(i) ≠ GC_table(i)(GC_list(round)) et i ≠
   request_controller_index then
17:            involved_controllers(i) ← vrai
18:            nb_involved_controllers ← nb_involved_controllers + 1
19:            coord_suggestion(i) ← GC_table(i)(GC_list(round))
20:        end if
21:    end for
22:    //Passer au mode Treat_Responses
23:    current_mode ← Treat_Responses
```

les configurations courantes des régions peuvent coexister avec la configuration ciblée par la requête. Dans ce cas, aucun autre contrôleur n'est impliqué dans le processus de coordination comme le montre l'algorithme 5.

L'activité 6) consiste à envoyer les propositions liées à un tour de coordination donné aux contrôleurs impliqués dans ce tour (selon la possibilité de la liste *GC_list* considérée dans ce tour). L'algorithme 6 illustre cette partie. La variable *nb_involved_controllers* permet de stocker le nombre de contrôleurs impliqués dans un tour de coordination. Cette variable est initialisée à 1 et correspond au contrôleur dont la requête est traitée dans le processus de coordination. La variable *involved_controllers* est réinitialisée à chaque tour en affectant la valeur *faux* à tous les éléments du tableau sauf l'élément ayant l'indice du contrôleur dont la requête est traitée dans le processus de coordination. Cette variable est ensuite mise à jour selon la possibilité considérée dans le tour courant comme le montre l'algorithme. Si le mode courant d'un contrôleur doit changer pour aller dans la configuration globale d'indice *GC_list*(*round*), comme le montre la ligne 16 de l'algorithme 6, l'élément du tableau *involved_controllers* correspondant à ce contrôleur prend la valeur *vrai*. Le coordinateur lui envoie une proposition de reconfiguration à travers la sortie *coord_suggestion*.

Algorithm 7 Extrait de l'algorithme de coordination lié aux traitement des réponses des contrôleurs

```
1: case Treat_Responses
2:     //Traitement des réponses
3:     for i = 1 to n do
4:         if controller_response(i) = acceptance et involved_controllers(i) = vrai et response_treated(i) = faux then
5:             nb_positive_responses ← nb_positive_responses + 1
6:             response_treated(i) ← vrai
7:         else
8:             if controller_response(i) = refus et involved_controllers(i) = vrai et response_treated(i) = faux then
9:                 nb_negative_responses ← nb_negative_responses + 1
10:                response_treated(i) ← vrai
11:            end if
12:        end if
13:    end for
14:
15:    //Décision
16:    if nb_positive_responses + nb_negative_responses = nb_involved_controllers − 1 then
17:        end_round ← vrai
18:        if nb_positive_responses = nb_involved_controllers − 1 then
19:            final_decision ← authorisation
20:        else
21:            if end_possibility_list then
22:                final_decision ← null
23:            elsefinal_decision ← refus
24:            end if
25:        end if
26:    end if
27:
28:    //Réinitialisation des variables internes à la fin d'un tour de coordination
29:    if end_round = vrai then
30:        nb_positive_responses ← 0
31:        nb_negative_responses ← 0
32:        response_treated ← {faux, faux, ....., faux}
33:    end if
```

Après avoir envoyé les propositions liées à un tour donné, le coordinateur attend les réponses des contrôleurs. La détermination de la décision finale à partir des réponses reçues peut suivre différents algorithmes selon l'application implémentée comme précisé dans la section 4.5.1. Dans l'implémentation que nous proposons dans ce travail, pour autoriser la reconfiguration elle doit être acceptée par tous les contrôleurs

concernés. Si le coordinateur reçoit des refus, et qu'il reste des configurations de la liste *GC_list* qui n'ont pas été considérées, le coordinateur revient à l'activité 6) pour envoyer des propositions de reconfiguration liées à la configuration globale suivante dans la liste *GC_list*. L'algorithme 7 illustre le traitement des réponses des contrôleurs. La variable *response_treated* est un tableau de booléens permettant d'indiquer si une réponse a déjà été traitée pour ne pas la traiter plusieurs fois. Les variables *nb_positive_responses* et *nb_negative_responses* permettent de stocker respectivement les réponses positives et négatives des contrôleurs. La décision du coordinateur est représentée par la variable *final_decision*.

Algorithm 8 Extrait de l'algorithme de coordination lié à l'envoi des décisions et la fin du processus de coordination

```
1: //Envoi de la décision finale
2: if final_decision ≠ null then
3:     current_mode ← Send_Suggestions
4: else
5:     if final_decision = authorisation then
6:         for i = 1 to n do
7:             if involved_controllers(i) = vrai then
8:                 coord_decision(i) ← autorisation
9:             end if
10:        end for
11:        current_mode ← Receive_Notifications
12:    else
13:        if final_decision = refus then
14:            coord_decision(request_controller_index) ← refus
15:            coord_inprogress ← faux
16:            current_mode ← Waiting
17:        end if
18:    end if
19: end if
20:
21: case Receive_Notifications
22:    if mode_updated = involved_controllers then
23:        //Mise à jour de la configuration globale courante
24:        for i = 1 to n do
25:            if involved_controllers(i) = vrai then
26:                current_config(i) ← GC_table(i)(GC_list(round))
27:            end if
28:        end for
29:        coord_inprogress ← faux
30:        current_mode ← Waiting
31:    end if
```

Envoi des décisions et fin du processus de coordination
Cette partie concerne les activités 8.a), 8.b), 9) et 10) de l'organigramme de la figure 4.8 qui correspondent respectivement aux actions (*send_authorization* ({$coord\_decision_i$}, *involved_controllers*)), *send_refusal* ({$coord\_decision_i$}, *involved_controllers*) et $coord\_inprogress = faux$ de la figure 4.6. L'algorithme 8 illustre cette partie. Si la décision finale est non nulle, elle est envoyée aux contrôleurs concernés. S'il s'agit d'une autorisation, elle est envoyée à tous les contrôleurs impliqués dans le dernier tour de coordination (ou seulement au contrôleur qui a envoyé la requête si on autorise directement sa reconfiguration) afin qu'ils puissent lancer les commandes de reconfiguration à travers les modules de reconfiguration. Le coordinateur passe ensuite au mode *Receive_Notifications*. S'il s'agit d'un refus, le coordinateur l'envoie au contrôleur qui a envoyé la requête, il finit le processus de coordination en notifiant tous les contrôleurs et passe au mode *Waiting*. Une fois dans le mode *Receive_Notifications*, si le coordinateur reçoit des notifications de tous les contrôleurs auxquels il a autorisé la reconfiguration, il met à jour le contenu de la variable *current_config* avec les modes courants des contrôleurs. Il notifie ensuite tous les contrôleurs pour indiquer la fin du processus de coordination et passe au mode *Waiting*.

4.6 Conclusion

Dans ce chapitre, le modèle de contrôle semi-distribué proposé dans ce travail a été détaillé. Ce modèle est composé d'un ensemble de contrôleurs distribués associés chacun à une région reconfigurable du système, et d'un coordinateur entre ces contrôleurs. Ces derniers ont une structure modulaire afin de garantir leur flexibilité et réutilisabilité pour d'autres systèmes. Les modules de chaque contrôleur offrent trois services : l'observation, la prise de décision et la reconfiguration, qui sont les services nécessaires pour l'auto-adaptation des régions contrôlées. Ceci facilite la réutilisation d'un contrôleur avec le composant reconfigurable qu'il contrôle et l'adaptation de certains modules à des composants reconfigurables différents.

Les décisions de reconfiguration prises par les contrôleurs sont coordonnées par un coordinateur afin de respecter les contraintes/objectifs globaux du système. L'utilisation d'un coordinateur au lieu d'une communication directe entre contrôleurs permet la séparation entre les problèmes de contrôle locaux gérés par les contrôleurs et le problème de contrôle global géré par le coordinateur en restreignant les échanges entre le coordinateur et un contrôleur donné à des requêtes et propositions de reconfiguration qui ne concernent que la région contrôlée par ce dernier. Cette séparation facilite la réutilisation des contrôleurs et du coordinateur. De plus, l'utilisation du coordinateur permet de diminuer le nombre de messages échangés avant d'arriver à une configuration globale satisfaisante, par rapport à une communication directe entre contrôleurs.

Le formalisme d'automates de modes parallèles utilisé pour la prise de décision semi-distribuée facilite la réutilisation et la scalabilité du modèle de contrôle. L'utilisation de ce formalisme permet aussi l'abstraction du mécanisme de prise de décision par rapport à l'implémentation physique, ce qui facilite son adaptation à différentes plates-formes.

Le mécanisme de coordination modélisé par l'automate de modes du coordinateur

offre la possibilité à différentes implémentations selon la stratégie du coordinateur qui dépend du problème de contrôle traité. Ce dernier peut être basé sur la satisfaction de certaines contraintes globales, l'optimisation de certains critères, etc. Cette flexibilité du mécanisme de coordination facilite son adaptation à différents problèmes de contrôle.

Le modèle de contrôle proposé peut bien profiter d'une approche IDM permettant d'automatiser la génération du code à partir d'une modélisation à haut-niveau d'abstraction en utilisant le profil UML MARTE. Ce dernier convient bien pour la modélisation de la structure des contrôleurs et du coordinateur et le comportement modal de ceux-ci. L'application de l'approche IDM au modèle de contrôle proposé dans ce travail est détaillée dans le chapitre suivant.

Chapitre 5

De la modélisation à la génération automatique du contrôle

5.1 Introduction

Afin de diminuer la complexité de la conception du contrôle de l'adaptation dynamique et d'améliorer la productivité des concepteurs, nous proposons une approche de conception de contrôle basée sur l'Ingénierie Dirigée par les Modèles (IDM), dans le cadre de l'environnement Gaspard2, afin de générer automatiquement le code du contrôle à partir de modèles à haut niveau d'abstraction utilisant le profil UML MARTE. Cette approche vise à cacher un grand nombre de détails d'implémentation qui seront gérés automatiquement par les transformations de modèles et la génération de code, ce qui permet de faciliter la conception, d'éviter les erreurs dues à l'implémentation directe du contrôle à un bas niveau tel que le niveau RTL (Register Transfer Level) et de réduire le temps de conception et de mise sur le marché des systèmes embarqués cibles.

Dans ce chapitre, le flot de conception de contrôle proposé dans le cadre de cette approche est détaillé en passant de la modélisation, aux transformations de modèles jusqu'à la génération du code. Ce flot permet d'obtenir des systèmes de contrôle en VHDL prêts à être intégrés dans des systèmes reconfigurables ciblant une plate-forme FPGA. Cette intégration est manuelle et elle sera implémentée dans le chapitre suivant à travers une étude de cas complète. A la fin du présent chapitre, nous proposons une approche de fusion du flot de conception du contrôle semi-distribué dans celui des

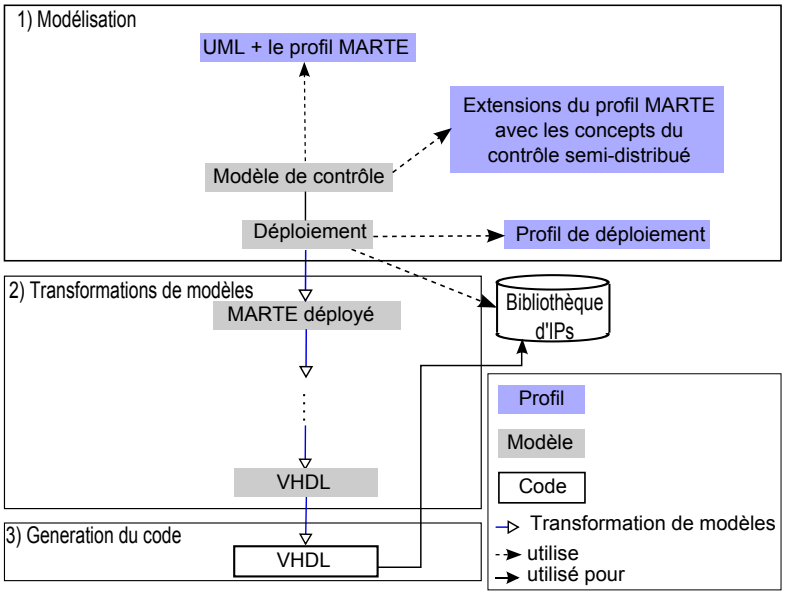


FIG. 5.1 – Vue globale du flot de conception du contrôle

systèmes reconfigurables conçus dans Gaspard2, afin d'éviter l'intégration manuelle du contrôle dans ces systèmes. Cette proposition se limite à la phase de modélisation, mais elle pourrait être facilement exploitée (en développant les règles de transformations de modèles et de génération du code nécessaires) pour générer le code de systèmes complets avec le contrôle intégré.

5.2 Le flot de conception du contrôle

Ce flot est inspiré de celui de la conception des systèmes embarqués dans Gaspard2 présenté dans la section 3.5.3. Comme le montre la figure 5.1, le flot de conception du contrôle est composé de trois étapes : 1) la modélisation, 2) les transformations de modèles et 3) la génération du code. Ce flot sera détaillé dans cette section.

5.2.1 Modélisation du contrôle

5.2.1.1 Modélisation de la structure des contrôleurs

Un contrôleur est modélisé avec un composant UML ayant des ports lui permettant de communiquer avec les autres composants du système. La figure 5.2 illustre la structure d'un contrôleur nommé *ControllerA*. Ce contrôleur est composé d'un ensemble de composants modélisant ses différents modules : observation, décision et reconfiguration. Les ports sont stéréotypés «*FlowPort*» en utilisant le profil MARTE permettant d'indiquer les directions des ports (entrée (in), sortie (out) ou entrée/sorties (inout)) comme le montre la figure 5.2.

Le composant *MonitoringModuleA* représente le module d'observation. Il a comme

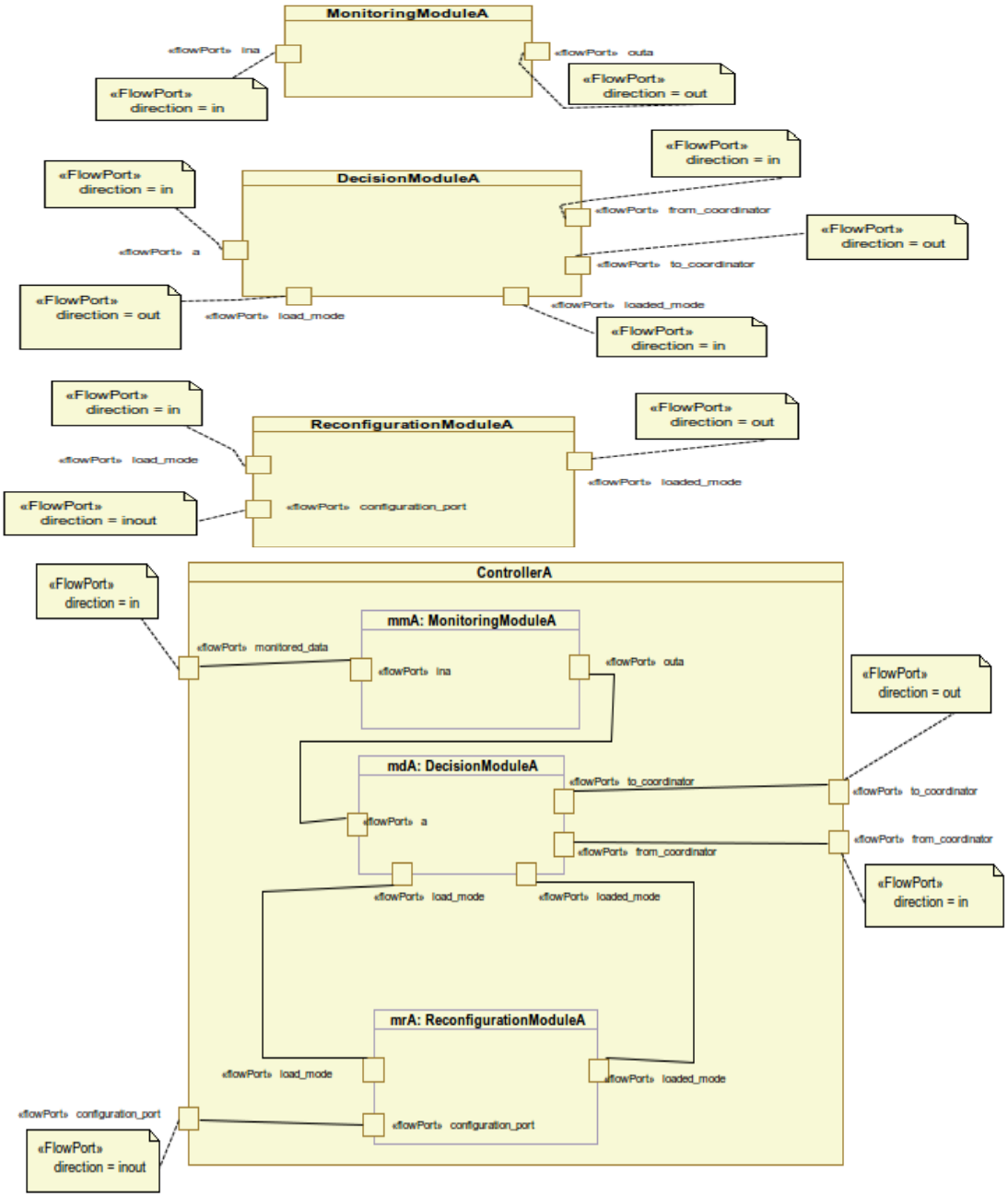


FIG. 5.2 – Modélisation de la structure d'un contrôleur

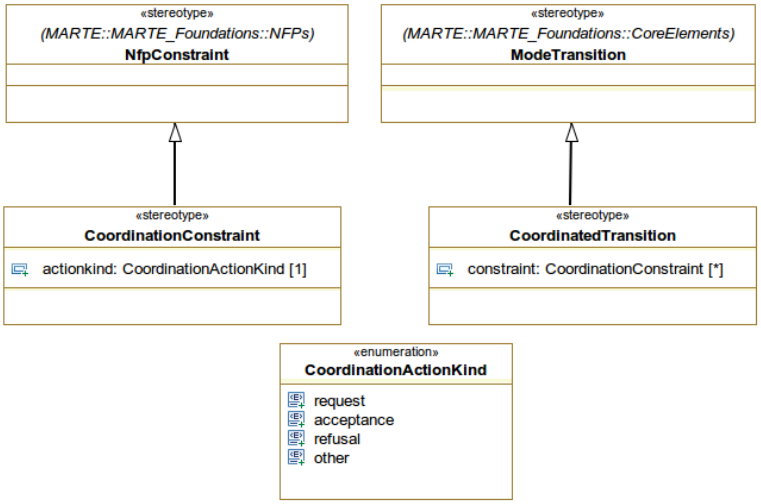


FIG. 5.3 – Partie de l'extension du profil MARTE pour la modélisation des contrôleurs distribués

entrée le port *ina*, qui représente les données observées par ce module. Le module d'observation analyse et transforme ces données, si nécessaire, sous une forme qui sera directement prise en compte par le module de décision. Les données de sortie du composant *MonitoringModuleA* sont représentées par le port *outa* qui est connecté au port *a* du composant *DecisionModuleA* représentant le module de décision. Pour communiquer avec le coordinateur, *DecisionModuleA* utilise deux ports : *from_coordinator* et *to_coordinator*. Le module de décision indique au module de reconfiguration le mode à charger à travers le port *load_mode*. Pour charger le bitstream, le module de reconfiguration communique avec le(s) port(s) de configuration du FPGA à travers le port *configuration_port*. Après le chargement du bitstream, le module de reconfiguration notifie le module de décision à travers le port *loaded_mode* indiquant le mode chargé dans la région contrôlée.

5.2.1.2 Modélisation de la prise de décision semi-distribuée

Le mécanisme de prise de décision proposé dans ce travail se base sur le formalisme des automates de modes parallèles. Le profil MARTE fournit des outils de modélisation pour le comportement modal qui sont eux aussi inspirés du formalisme des automates de modes. Ces outils peuvent être donc exploités pour la modélisation de la prise de décision. Cependant, le profil MARTE ne supporte pas certaines notions telles que la notion de coordination et la notion de la liste de configurations globales d'un système. Pour cette raison, des extensions au profil MARTE sont proposées pour pouvoir modéliser tous les aspects de la prise de décision semi-distribuée.

La modélisation de l'automate de modes d'un contrôleur est faite en associant une machine à états UML, appliquant les concepts de comportement modal de MARTE, au composant représentant le module de décision. La figure 5.4 illustre la modélisation de l'automate de modes du contrôleur dont la structure a été décrite dans la figure 5.2. Ce

modèle applique aussi des extensions proposées dans ce travail pour le profil marte, qui sont illustrées dans la figure 5.3. L'objectif de ces extensions est de faciliter la modélisation aux concepteurs en leur donnant des outils permettant de minimiser les données à modéliser et de maximiser l'automatisation. Dans le cas de l'automate des modes du module de décision, le concepteur a besoin, en plus des concepts de MARTE permettant la modélisation les modes et les transitions, d'indiquer les conditions d'envoi des requêtes de reconfiguration et les conditions d'acceptation/refus des propositions de reconfiguration. Tout le reste du mécanisme géré par l'automate du contrôleur (la communication avec le coordinateur, le module de reconfiguration et le module d'observation) est caché au concepteur et sera géré par les transformations de modèles et l'outil de génération du code. Pour faciliter la modélisation des conditions de requêtes et d'acceptation/refus aux propositions, celles-ci sont attachées aux transitions de la machine à états, comme le montre la figure 5.4, ce qui permet de déduire le mode source et le mode cible de la requête/proposition. Pour faire la différence entre ces deux types de conditions, des extensions au profil MARTE ont été proposées. Pour faire la différence entre une transition «*ModeTransition*» de MARTE, qui indique le passage effectif d'un mode source à un mode cible, et une transition qui est sujet d'une coordination entre contrôleurs (elle peut être autorisée comme elle peut être refusée), le stéréotype «*CoordinatedTransition*» a été proposé. Il hérite du stéréotype «*ModeTransition*» comme le montre la figure 5.3 et il est appliqué aux transitions à la place du stéréotype de base «*ModeTransition*». Cette extension rend possible l'association des conditions de requête, acceptation ou refus à une transition entre deux modes. Pour faire la différence entre ces différents types de conditions, le stéréotype «*CoordinationConstraint*» est proposé. Celui-ci hérite du stéréotype «*NfpConstraint*» de MARTE, qui permet de modéliser des contraintes liées à des propriétés non fonctionnelles et qui convient donc pour définir des contraintes sur les données d'observation (collectées par le module d'observation d'un contrôleur) qui sont des propriétés non fonctionnelles. Pour indiquer le type de la condition/contrainte (requête, acceptation ou refus), le stéréotype «*CoordinationConstraint*» a un attribut *actionKind*, qui peut prendre une valeur parmi celles définies par l'énumération *CoordinationActionKind* comme le montre la figure 5.3. L'attribut *constraint* du stéréotype «*CoordinatedTransition*» permet d'associer les «*CoordinationConstraint*»s aux transitions.

Comme le montre la figure 5.4, les décisions prises par le contrôleur sont basées sur la comparaison d'une donnée *a* envoyée par le module d'observation avec deux constantes *ta1* et *ta2* (*ta1* < *ta2*). Dans cet exemple, le contrôleur estime qu'une transition du *Mode1* au *Mode2* est nécessaire si la condition $a > ta2$ est valide. Dans ce cas, il envoie une requête de reconfiguration au coordinateur. Cette décision est modélisée par une contrainte «*CoordinatedTransition*» ayant *request* comme valeur pour l'attribut *actionKind*, indiquant qu'il s'agit d'une requête de transition. Il en est de même pour les acceptations et refus des propositions de reconfiguration envoyées au coordinateur qui sont modélisées par des contraintes «*CoordinatedTransition*» ayant respectivement *acceptance* et *refusal* comme valeurs pour l'attribut *actionKind*. Ici, la modélisation des contraintes de refus n'est pas obligatoire si on a modélisé la contrainte d'acceptation parce que de toute façon la condition de l'une est l'inverse de l'autre. La figure 5.4 contient les contraintes d'acceptation et de refus pour mieux présenter les possibilités de

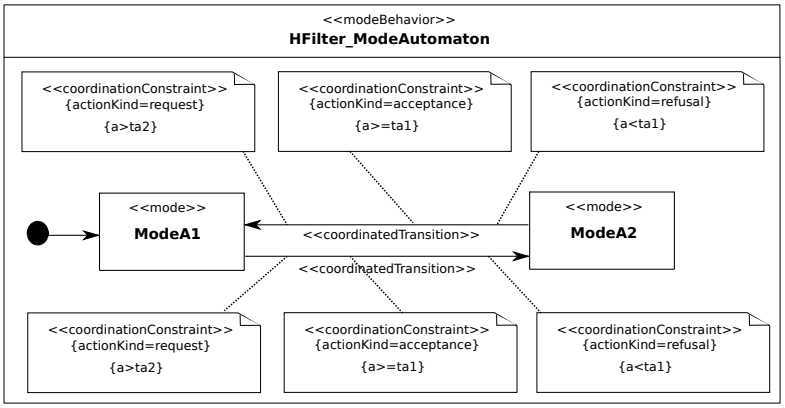


FIG. 5.4 – Exemple d'un modèle d'automate de modes d'un contrôleur

modélisation, mais il suffit de mettre soit l'une soit l'autre.

Cette modélisation du contrôleur permet de cacher plusieurs détails qui seront gérés par l'outil de génération du code tels que la traduction des différentes conditions de transitions selon un protocole de communication entre le contrôleur et le coordinateur, la notification envoyée par le coordinateur au début/fin d'un processus de coordination, l'envoi de la commande de reconfiguration au module de reconfiguration et la notification du coordinateur après le chargement du bitstream. Le code à générer à partir de la modélisation de la figure 5.4 donnera un automate de modes contenant plus de détails sur le mécanisme de coordination tels qu'ils ont été présenté dans le chapitre 4. Cet automate est illustré dans la figure 5.5. Cette figure montre que les transitions de modes effectives (par rapport à celles modélisées dans la figure 5.4) ne sont faites qu'après l'autorisation du coordinateur et le chargement du bitstream par le module de reconfiguration.

Pour la modélisation du coordinateur, le concepteur a besoin principalement de la notion de la liste de configurations globales d'un système (la table *GC* présentée dans le chapitre précédent). Cette liste est une matrice de modes, indiquant les combinaisons possibles entre les modes des régions reconfigurables du système. Pour bien faire la correspondance entre les cases de cette matrice et les contrôleurs gérant les modes, le coordinateur a besoin de la liste ordonnée des contrôleurs coordonnés. Quant à l'automate de modes du coordinateur, le concepteur n'a pas besoin de le modéliser puisqu'il s'agit du même automate de modes moyennant le changement de la table de configurations globales gérée par le coordinateur et les contrôleurs coordonnés. Le concepteur n'a donc besoin que de modéliser ces deux points. Pour cela, des extensions ont été proposées au profil MARTE puisqu'il ne contient pas à la base des notions telles que la matrice de modes. Ces extensions sont illustrées dans la figure 5.6(a). Pour pouvoir modéliser un coordinateur le stéréotype «*Coordinator*» qui hérite du stéréotype MARTE «*RtUnit*» du package *HLAM* est proposé. Ce dernier permet de modéliser des unités temps réel qui ont un ensemble de ressources à ordonnancer [99]. Ces unités peuvent être vues comme des ressources d'exécution autonomes capables de gérer différents messages en même temps. Dans ce contexte, le coordinateur peut être vu comme une

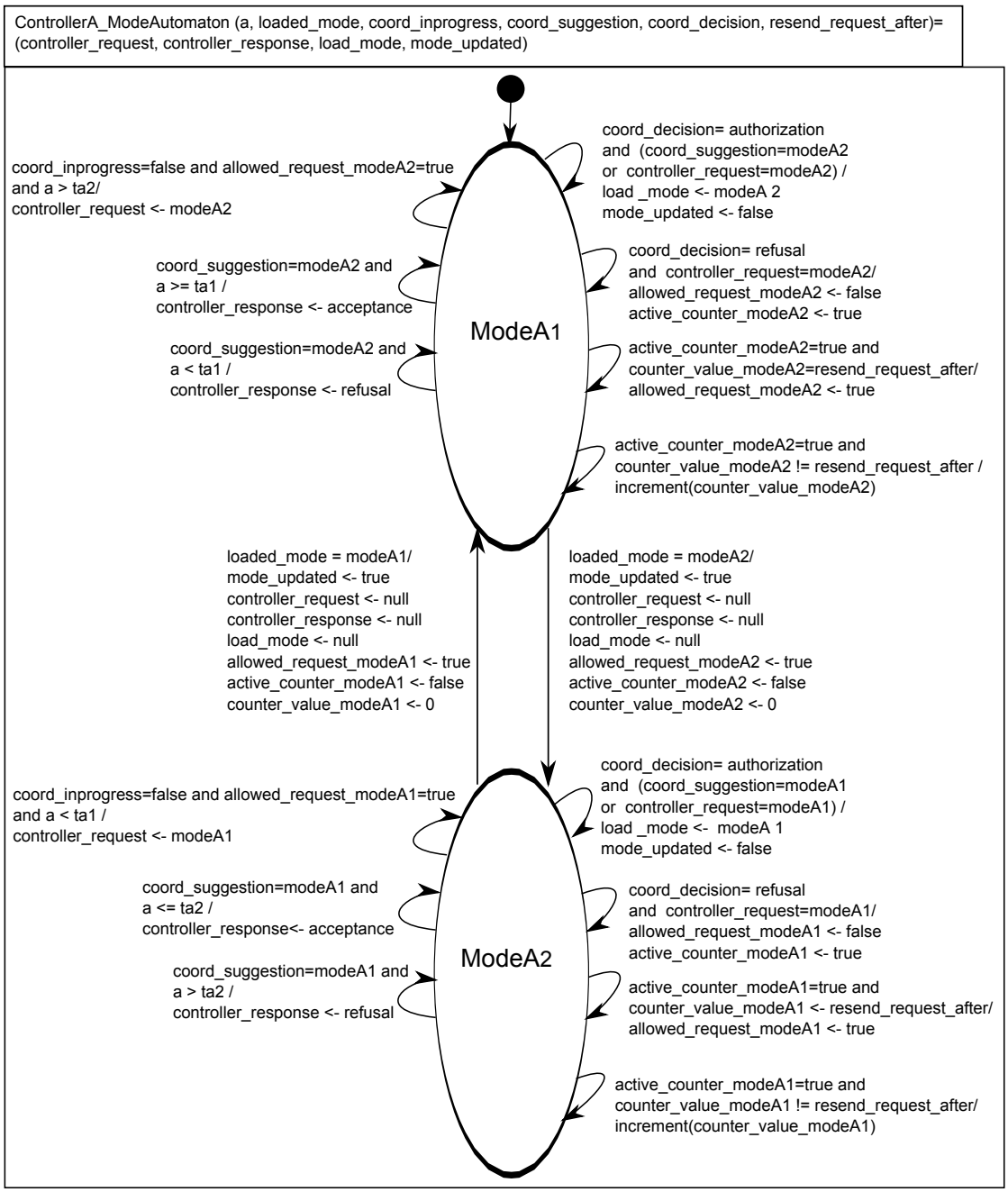
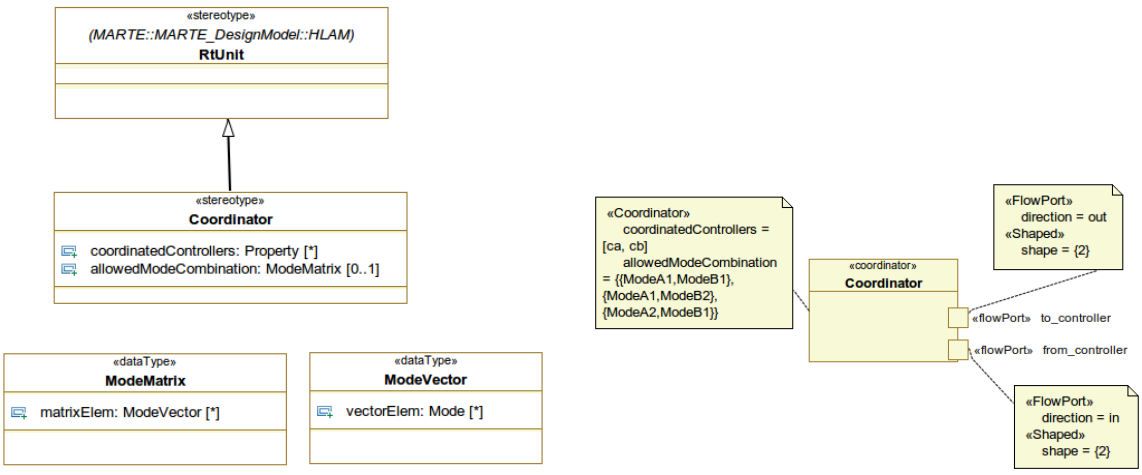


FIG. 5.5 – Automate de modes à générer à partir de celui de la figure 5.3

ressource d'exécution qui gère "l'ordonnancement" d'un ensemble de ressources (les configurations des régions reconfigurables) et qui traite différents messages (requêtes et réponses) venant des contrôleurs des systèmes. Le stéréotype «*Coordinator*» a un attribut nommé *coordinatedControllers* qui permet de spécifier les contrôleurs coordonnés. Cet attribut est de type tableau de *Property*, ce qui permet d'associer un ensemble d'instances de contrôleurs au coordinateur. L'utilisation du type *Property* au lieu de *Class* offre la possibilité d'associer au coordinateur plusieurs instance du même contrôleurs si on a des régions semblables. La table *GC* du coordinateur décrite dans le chapitre précédent est représentée par l'attribut *allowedModeCombination* qui a comme type *ModeMatrix* permettant de modéliser une matrice de modes. Pour définir ce nouveau type, nous avons utilisé des «*dataType*» stéréotypés «*CollectionType*» de MARTE. Ce stéréotype permet de modéliser une liste d'éléments ayant le même type. Ici, une *ModeMatrix* est une liste de *ModeVector*s et un *ModeVector* est une liste de *Mode*.

La figure 5.6(b) montre un exemple de modélisation d'un coordinateur. Ici, on considère que le système de contrôle contient deux contrôleurs *ControllerA* et *ControllerB* qui ont comme modes *ModeA1* et *ModeA2*, et *ModeB1* et *ModeB2* respectivement. Le coordinateur contrôle une instance de chaque contrôleur (*ca* et *ca*). Le rôle du coordinateur est de garantir que la configuration globale du système respecte un ensemble de contraintes globales. Dans cette exemple, on suppose que ces contraintes indiquent que les modes *ModeA2* et *ModeB2* ne doivent pas être actifs en même temps. Dans ce cas, la table *GC* ne contient que trois configurations globales possibles. Ces configurations sont illustrées par l'attribut *allowedModeCombination* dans la figure 5.6(b). Les ports du coordinateur ont le stéréotype «*Shaped*» du package *RSM* de MARTE, qui permet de représenter la répétition de ces ports. Ici, on donne la valeur 2 à l'attribut *shape* de ce stéréotype pour les ports *to_controller* et *from_controller* puisqu'ils permettent la connexion du coordinateur à deux contrôleurs.



(a) Partie de l'extension du profil MARTE pour la modélisation du coordinateur

(b) Exemple de modèle de coordinateur

FIG. 5.6 – Modélisation du coordinateur

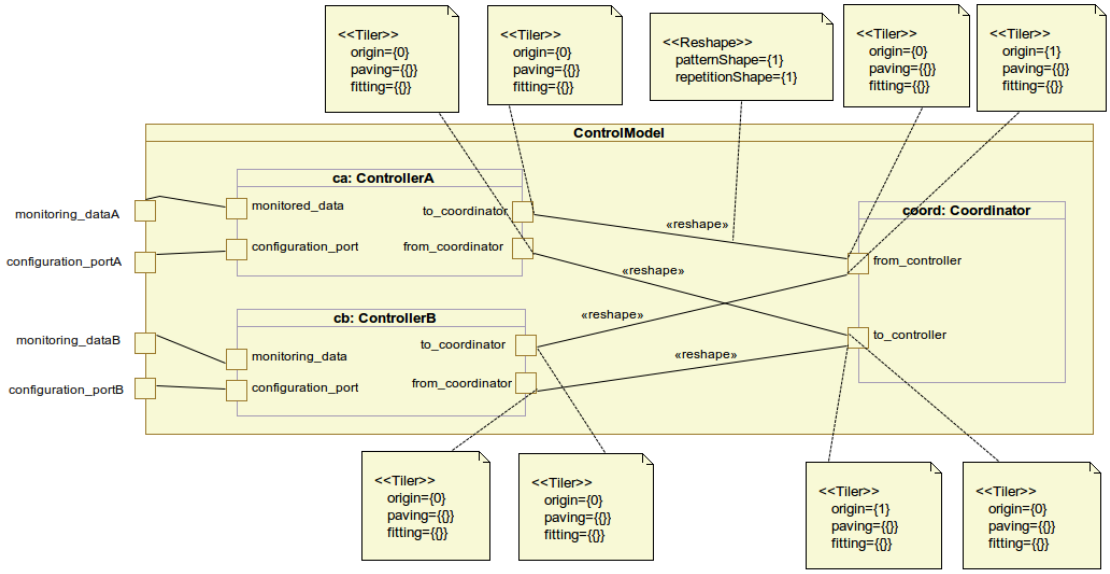


FIG. 5.7 – Modélisation d'un système de contrôle

5.2.1.3 Modélisation de l'architecture des systèmes de contrôle

Un système de contrôle contient les contrôleurs distribués et le coordinateur. La figure 5.7 montre le système de contrôle utilisant le contrôleur de la figure 5.4 et le coordinateur de la figure 5.6(b). Pour la connexion entre contrôleurs et coordinateur, nous appliquons le stéréotype «*Reshape*» du package RSM sur les connecteurs. Ce stéréotype permet d'exprimer le réarrangement des éléments entre deux tableaux multidimensionnels source et destination. Il a deux attributs : *patternShape* et *repetitionShape*. Le premier permet de déterminer le *shape* (le motif) des éléments du tableau source avec lesquels le tableau destination va être rempli. L'attribut *repetitionShape* indique l'espace de répétitions que prend le réarrangement des éléments entre deux tableaux multidimensionnels source et destination. Dans l'exemple de la figure 5.7, pour le connecteur entre le port *to_coordinator* du contrôleur *ca* et le port *from_controller* du coordinateur *coord*, les valeurs des attributs *patternShape* et *repetitionShape* sont {1} {1}. La première valeur indique qu'on va connecter un seul élément (ici on n'en a qu'un) de l'interface *to_coordinator* au coordinateur et qu'on va faire cela une seule fois.

Pour mieux comprendre l'utilisation du package *RSM* pour la modélisation des systèmes de contrôle, un exemple plus générique est présenté dans l'annexe A. L'exemple de la figure 5.7 donne un cas particulier de la connexion avec des «*Reshape*» ayant des tableaux sources à un élément et des tableaux destinations à une dimension. Pour indiquer comment les éléments du tableau source seront parcourus et comment ils seront rangés dans le tableau destination, on utilise les stéréotypes «*Tiler*» qu'on applique sur les extrémités des connecteurs. Un *Tiler* a trois attributs : *origin*, *paving* et *fitting*. L'*origin* permet d'indiquer l'origine à partir de laquelle on commence à extraire (pour un *Tiler* source) ou remplir (pour un *Tiler* destination) les éléments du tableau. Le pavage (*paving*) indique comment se déplace l'origine du *shape* on passant d'une répétition à la suivante.

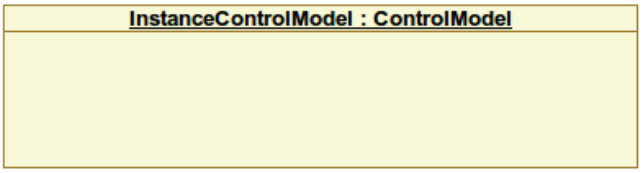


FIG. 5.8 – Instance du système de contrôle

Le pavage est une matrice ayant comme nombre de lignes la dimension du tableau et comme nombre de colonnes la dimension de l'espace de répétitions. L'ajustage (*fitting*) indique comment le *shape* est construit à partir des éléments du tableau. L'ajustage est une matrice ayant comme nombre de lignes la dimension du tableau et comme nombre de colonnes la dimension du *shape*. Dans ce cas particulier, les matrices de pavage et d'ajustage sont vides. Pour les origines des *Tiler*, ils ont toutes la valeur {0} sauf pour les *Tiler* des connecteurs entre le contrôleur *cb* et le coordinateur du côté du coordinateur qui ont la valeur {1}, ce qui indique que le port *to_coordinator* du contrôleur *cb* est connecté au deuxième élément du tableau *from_controller* du coordinateur et que le port *from_coordinator* du contrôleur *cb* est connecté au deuxième élément du tableau *to_controller* du coordinateur.

Avant de passer au déploiement des composants élémentaires, il faut créer une instance UML du système qui permet d'indiquer aux transformations le composant englobant tout le système. Dans Gaspard2, pour la modélisation des systèmes embarqués, on a généralement besoin de deux instances UML : une pour l'application et une pour l'architecture. Celle de l'application peut être par exemple traduite, à la phase de génération de code, en la fonction main du processeur en langage C. L'instance de l'architecture peut être traduite par exemple en un module *top* en *VHDL*. Ici, nous visons la génération du système de contrôle seulement, nous avons donc besoin de créer une instance du composant *ControlModel* comme le montre la figure 5.8. Cette instance sera traitée comme si c'était une instance d'architecture dans Gaspard2 et sera traduite en des modules VHDL à la phase de génération du code.

5.2.1.4 Déploiement du système de contrôle

Comme nous avons précisé dans la section 3.5.3, le déploiement dans Gaspard2 consiste à attacher chaque composant élémentaire du système à un code existant dans la bibliothèque d'IP facilitant ainsi la réutilisation des IPs. Pour le déploiement des systèmes de contrôle, nous utilisons le profil déploiement de Gaspard2. Les composants élémentaires dans notre exemple sont les modules d'observation, de décision et de reconfiguration pour les deux contrôleurs ainsi que le coordinateur. Le déploiement dans Gaspard2 se fait en deux étapes. La première étape consiste à modéliser pour chaque composant élémentaire, le(s) composant(s) qui l'implémente(nt). En suivant la méthodologie de Gaspard2, chaque composant élémentaire doit être implémenté par une «*VirtualIP*» qui permet d'ajouter un niveau d'abstraction et d'associer différentes implémentations pour le même composant. Par exemple, un composant peut être implémenté par deux «*HardwareIP*», une écrite en langage VHDL et une écrite en

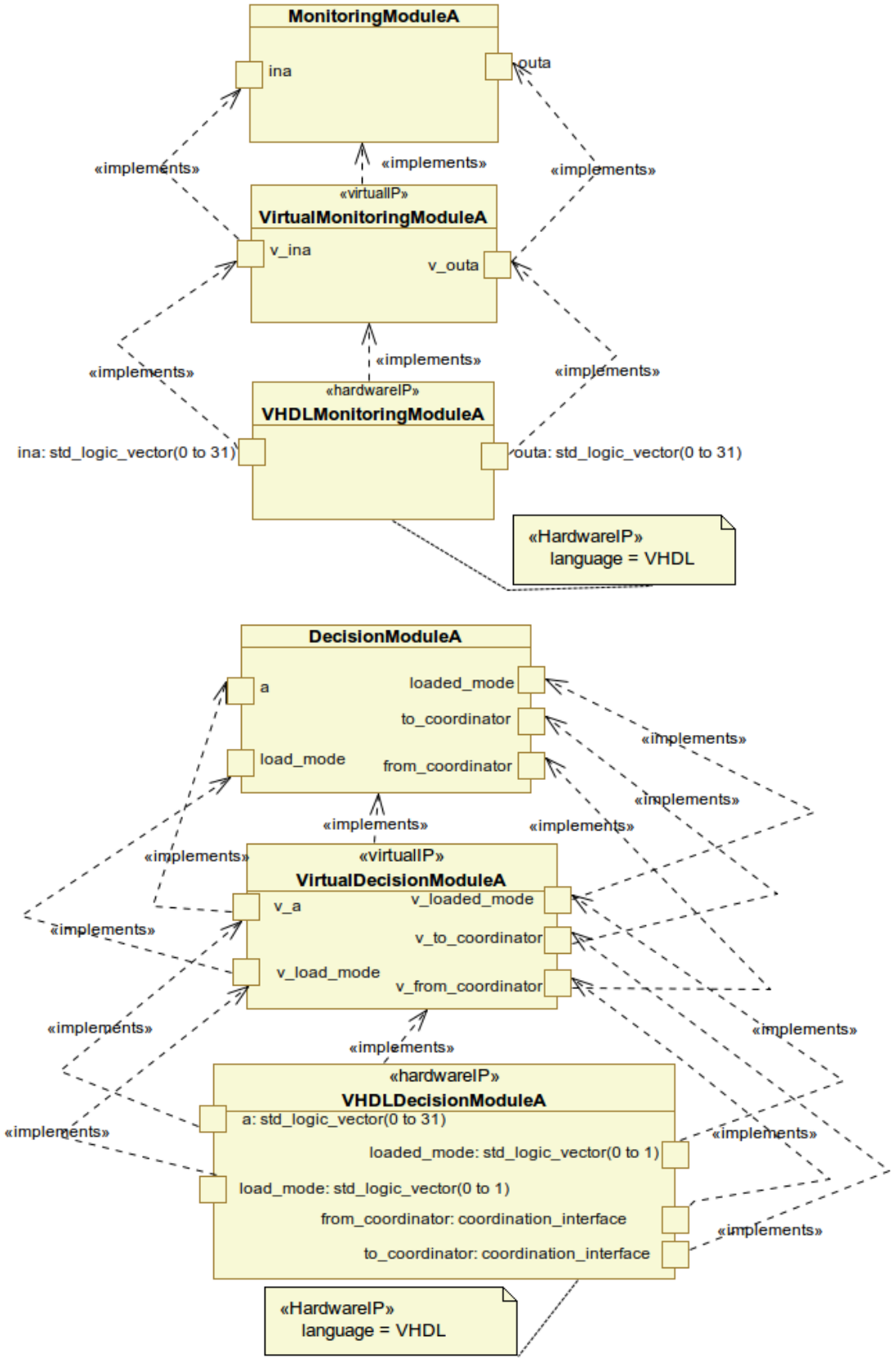


FIG. 5.9 – Déploiement des composants élémentaires du système

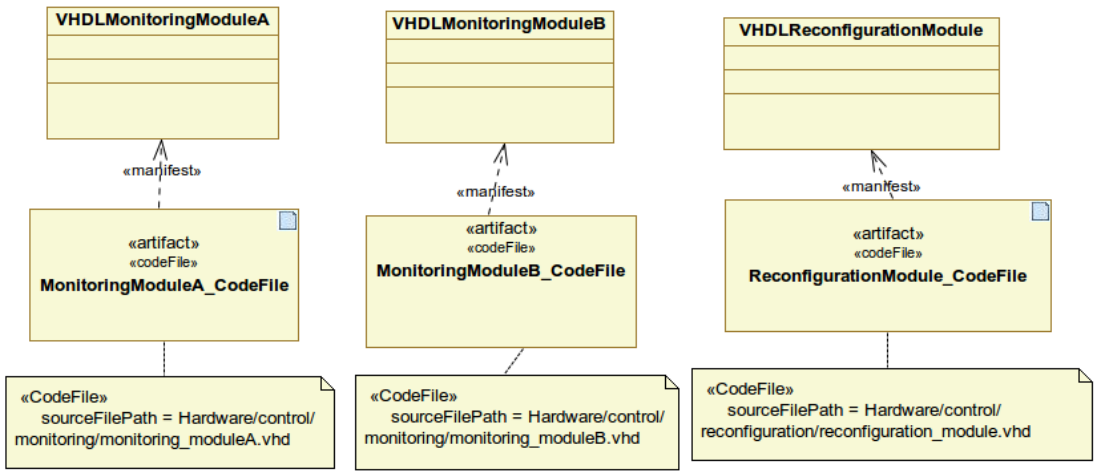


FIG. 5.10 – Association du code aux composants déployés

System-C visant ainsi différentes plates-formes à partir d'un même modèle. La figure 5.9 illustre la première étape de déploiement pour les modules d'observation et de décision du contrôleur *ControllerA* traité dans ce chapitre. La figure montre le déploiement des composants élémentaires d'un seul contrôleur. Le déploiement de l'autre contrôleur suit le même principe. Dans cette étape, on indique les types des ports pour les «*HardwareIP*». Ces types dépendent de la plate-forme ciblée. Ici, nous utilisons des types du langage VHDL puisque nous visons ce langage lors de la génération du système de contrôle. Comme le montre la figure 5.9, il y a des ports de types primitifs (*std_logic_vector(0 to 1)* et *std_logic_vector(0 to 31)*) et il y a de nouveaux types définis spécialement pour le système de contrôle. Ces derniers regroupent plusieurs signaux VHDL ayant différents types. Par exemple, le type *coordination_interface*, utilisé pour la communication entre les contrôleurs et le coordinateur, regroupe différents signaux gérant l'envoi des requêtes, des propositions, des réponses, etc. Dans l'étape de génération du code, ce type sera transformé en un enregistrement (record) VHDL regroupant les signaux nécessaires.

La deuxième étape de déploiement consiste à associer aux «*HardwareIP*» (et «*SoftwareIP*») des codes existants dans la bibliothèque d'IP. Puisque les modules d'observation et de reconfiguration communiquent avec des éléments à l'extérieur du système du contrôle, comme le montre la figure 5.7, leurs implémentations dépendent du reste du système reconfigurable. Par exemple, les données collectées par le module d'observation dépendent de l'application implémentée, du problème de contrôle traité, etc. Il peut par exemple effectuer un traitement spécial sur les données observées qui soit difficile à modéliser. Pour cette raison, il faut écrire manuellement le code des modules d'observation et de reconfiguration et le mettre dans une bibliothèque d'IP pour pouvoir les attacher aux «*HardwareIP*» correspondants dans le déploiement et les réutiliser pour d'autres systèmes. En ce qui concerne les modules de décision et de coordination, ils sont générés automatiquement à partir des modèles comme nous allons expliquer dans la suite du chapitre. Donc, la deuxième étape de déploiement ne concernent ici que les modules d'observation et de reconfiguration. La figure 5.10 illustre le déploiement de ces

modules. Pour associer les codes aux composants déployés, on utilise un artifact UML auquel on applique le stéréotype «*CodeFile*». Ce stéréotype a un ensemble d'attributs qui décrit les caractéristiques de ce code (commande de compilation, d'édition de liens, les fichiers d'entête, etc). Pour spécifier l'emplacement des codes des modules d'observation et de reconfiguration dans la bibliothèque d'IP, on utilise l'attribut *sourceFilePath* comme le montre la figure 5.10. Ici, on utilise la même IP pour le module de reconfiguration pour les deux contrôleurs mais deux IPs différentes pour les modules d'observation.

5.2.2 Les transformations de modèles

5.2.2.1 Le mécanisme de fusion dans Gaspard2

Le mécanisme de merge dans UML permet au contenu d'un package d'être étendu par le contenu d'un autre package [98]. Cela permet à un élément source d'ajouter des caractéristiques de l'élément cible à ses caractéristiques résultant en un élément qui combine les caractéristiques des deux. Ce mécanisme peut être utilisé quand des éléments définis dans des packages différents ont le même nom et représentent le même concept. Souvent ce mécanisme est utilisé pour fournir différentes définitions pour un concept donné pour différents objectifs, commençant par une définition de base commune. Le concept de base est étendu d'une façon incrémentale où l'incrément est défini dans un package séparé [98].

L'utilisation de ce mécanisme de merge dans la transformation de modèles permet de réduire le nombre de règles de transformation puisque le métamodèle cible n'est qu'une version étendue du métamodèle source. Ces règles considèrent dans ce cas la partie du modèle source qui sera traduite conformément aux concepts introduits par le métamodèle cible et qui étendent le métamodèle source.

La transformation de modèles dans Gaspard2 est basée sur ce mécanisme. A partir du métamodèle MARTE différents métamodèles sont construits. Une chaîne de transformations dans Gaspard2 est composée dans un ensemble de transformations qui commencent par une transformation d'UML vers MARTE. Ensuite, les transformations qui suivent ont des métamodèles cibles qui sont des versions étendues du métamodèles MARTE. Ce mécanisme permet aussi à des chaînes ciblant différentes plates-formes (OpenCl, SystemC, VHDL, etc) des transformations avant d'aller dans des transformations qui ajoutent des concepts de plus en plus spécifiques à la plate-forme cible. Les transformations dans Gaspard2 sont écrites en QVTO [111]

5.2.2.2 La chaîne de transformations des modèles de contrôle

La figure 5.11 montre la chaîne de transformations que nous utilisons pour la génération des systèmes de contrôle en VHDL. Les deux premières transformations existent déjà dans Gaspard2. Pour pouvoir générer les systèmes de contrôle modélisés en utilisant le profil MARTE avec les extensions proposées dans la section 5.2.1.2, nous avons modifié la transformation *UML2MARTE*. A la base, cette transformation traduit le modèle en entrée en un modèle conforme au métamodèle MARTE tel qu'il est défini par l'OMG. Nous avons modifié cette transformation afin que les modèles en sortie supportent les concepts du contrôle semi-distribué qu'on a introduit dans les extensions

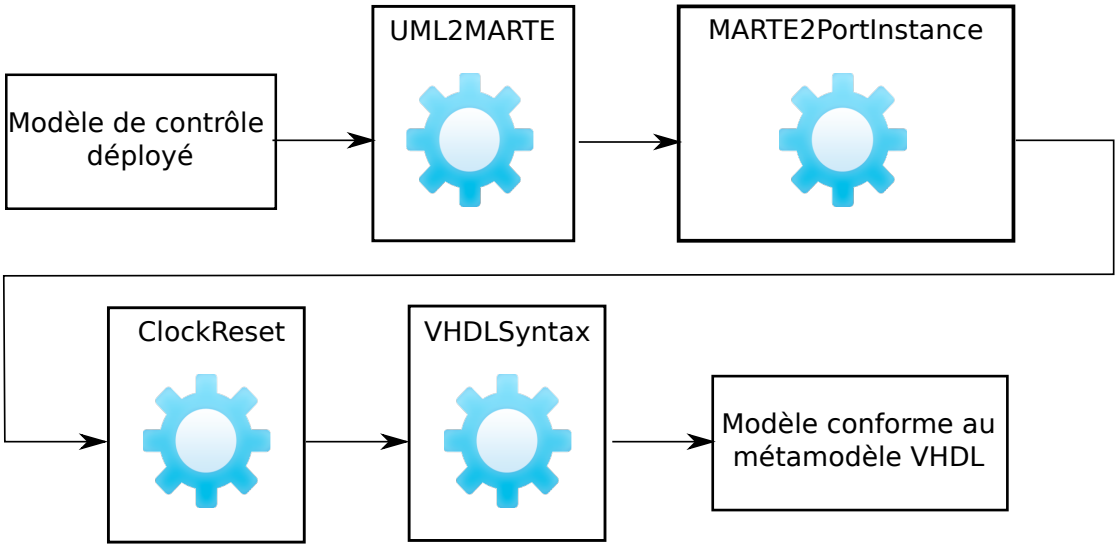


FIG. 5.11 – Chaîne de transformations pour les modèles de contrôle

du profil MARTE proposées. Le métamodèle cible de cette transformation n'est plus le métamodèle MARTE mais une version étendue de ce modèle et qui contient les concepts du contrôle semi-distribué. Les trois transformations qui restent font parti de notre contribution dans le environnement Gaspard2. A la base, notre proposition de ces trois transformations était dans le cadre de la génération de systèmes ciblant la plate-forme VHDL dans Gaspard2. Elle visait à la génération d'accélérateurs matériels à partir d'un modèle d'application en entrée mettant en oeuvre le parallélisme de données en utilisant le package RSM de MARTE. La figure 5.12 illustre la chaîne de transformations utilisée pour cela. Cette chaîne contient des transformations génériques (qui peuvent être utilisées par différentes chaînes indépendamment de la plate-forme visée). Ces transformations sont les quatre premières transformations qui existaient déjà dans Gaspard2 avant notre contribution. Les trois dernières transformations font parti de notre contribution dans l'environnement Gaspard2 pour la génération du code VHDL.

La chaîne de transformations que nous utilisons pour la génération des systèmes de contrôle réutilise un sous-ensemble des transformations de la chaîne de la figure 5.12 moyennant une modification dans la transformation *UML2MARTE* pour y intégrer les concepts de contrôle semi-distribué. Les transformations Tiler2Task et Shape2Loop de la chaîne de la figure 5.12 ne sont pas utilisées dans la chaîne de contrôle puisque les modèles UML utilisés ne contiennent pas de *Tiler*s et il n'y a pas de répétitions d'instances de composants. Ces deux transformations peuvent toujours être utilisées dans la chaîne de contrôle, mais il n'y aura aucune différence entre leurs entrées et sorties. Plus de détails sur ces deux transformations peuvent être trouvées dans [10].

Dans le reste de cette section, nous décrivons brièvement la chaîne de transformations du modèle de contrôle. Plus de détails sur les transformations et les métamodèles proposés pour cette chaîne se trouvent dans l'annexe B. La transformation *UML2MARTE* a comme entrée le modèle UML et comme sortie un modèle conforme au métamodèle MARTE. Pour que le modèle généré par la transformation supporte les notions du

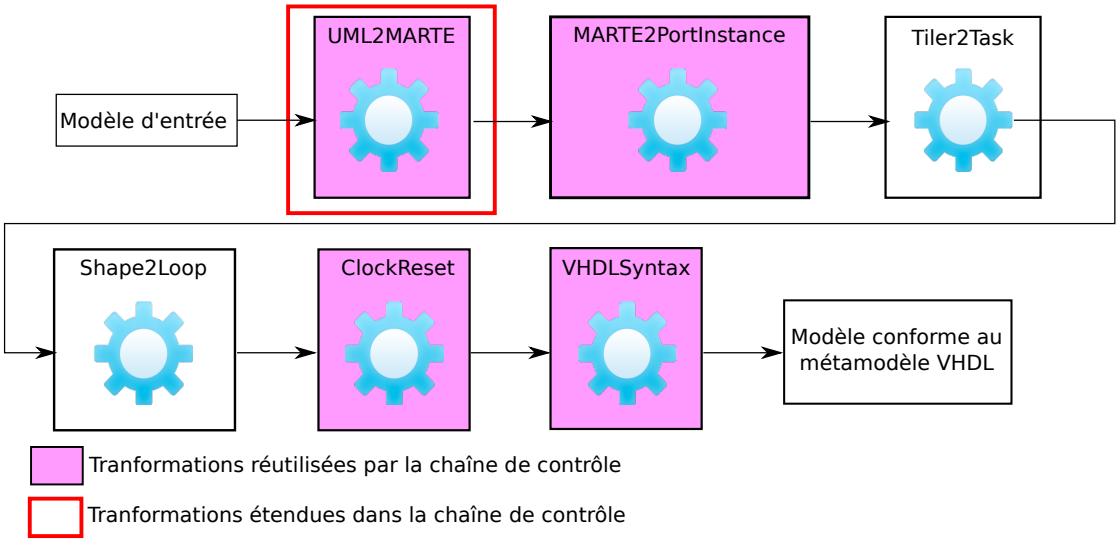


FIG. 5.12 – Chaîne de transformations de la plate-forme VHDL

contrôle semi-distribué, le métamodèle MARTE a été étendu. Cette extension est une traduction de l'extension que nous avons présentée dans la section 5.2.1.2 pour le profil MARTE selon la syntaxe du métamodèle. Plus de détails sur cette extension se trouvent dans l'annexe B. La transformation *UML2MARTE* a été également étendue en ajoutant des règles de transformations permettant de traduire les composants UML appliquant les stéréotypes proposés pour le contrôle en des classes selon l'extension du métamodèle MARTE. La transformation *MARTE2PortInstance* déjà existante dans Gaspard2 permet d'ajouter au modèle intermédiaire la notion d'instances de ports qui n'existe pas dans MARTE. La transformation *ClockReset* permet d'ajouter la notion de ports d'horloge et réinitialisation aux composants puisque la plate-forme cible ici est la plate-forme VHDL. Une extension du métamodèle MARTE a été proposée pour réaliser cette transformation. La transformation *VHDLSyntax* permet de traduire les éléments du modèle en entrée selon les concepts VHDL. Ces concepts sont introduits dans par un métamodèle VHDL qui étend le métamodèle MARTE. La transformation *VHDLSyntax* permet donc d'avoir un modèle contenant toutes les notions nécessaires (entités, composants, architectures, ports, signaux, etc) permettant de faciliter la génération du code en VHDL. Pour la transformation modèle vers texte permettant la génération du code, nous avons choisi de la décrire dans ce chapitre puisqu'elle permet d'obtenir le résultat final des transformations qui sera utilisé dans le chapitre suivant pour l'implémentation physique du contrôle semi-distribué.

5.2.3 La génération automatique du code

La génération de code est faite par une transformation modèle vers texte nommée *CodeVHDL* et illustrée dans la figure 5.13. Cette transformation permet de générer un système de contrôle composé de contrôleurs et d'un coordinateur. Le code généré contient tous les éléments liés à la structure des composants composés. Pour les composants

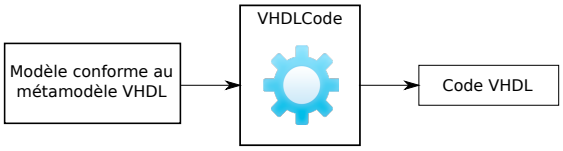


FIG. 5.13 – La transformation modèle vers texte pour la génération du code VHDL

élémentaires, leurs contenus sont soit générés soit récupérés à partir d'une bibliothèque d'IPs. Dans le cas des systèmes de contrôle utilisés dans notre travail, les contenus des modules de décision des contrôleurs sont générés ainsi que celui du coordinateur. Pour le code du reste des composants élémentaires, nous supposons qu'il existe déjà dans une bibliothèque d'IPs.

La transformation *CodeVHDL* génère un package *userlibrary* contenant la déclaration des types ainsi que de tous les composants utilisés dans le système. Ce package sera importé par tous les fichiers VHDL générés pour ne pas avoir à redéclarer à chaque fois les types et composants dont on a besoin. La transformation *CodeVHDL* est basée sur des fichiers de transformation Acceleo [2]. Le listage 5.1 donne un extrait de la transformation liée à la génération de la déclaration des types et composants. La transformation part de noeud correspond à l'instance du système de contrôle pour chercher les noeud de type *DeclaredType* et les parcourt dans une boucle *for* tel que le montre la ligne 1 du listage. Pour la déclaration des types dans VHDL, il s'agit d'écrire le mot clé **Type** puis le nom du type et sa description comme le montre la ligne 2 du listage. Pour déclarer les composants, il s'agit d'écrire le mot clé **Component**, le nom du composant et les ports (avec noms, directions et types) et finir par **end component ;** comme le reste du listage. Le listage 5.2 donne le code généré pour la déclaration du composant lié au module d'observation du contrôleur *ControllerA*. A côté du package contenant la déclaration des types et composants, la transformation *CodeVHDL* génère un fichier pour chaque *Entity* du modèle en entrée. Chaque fichier contient l'entité et son architecture. La transformation génère aussi les fichiers d'implémentation des contrôleurs et du coordinateur.

Listage 5.1 – Extrait de la transformation CodeVHDL lié à la génération de la déclaration des types et de composants

```
1  <%for (self.top.ownedElements.filter("DeclaredType")){%>
2  Type <%name%>  <%self.description%>;<%}%>
3  <%for (self.top.ownedElements.filter("Component")){%>
4  component <%name%>  is port(
5  <%for (self.clk){%>
6  <%name%> : <%if (direction == "_in"){%> IN <%}else{%> OUT<%}%> <%type.name%>;
7  <%}%>
8  <%for (self.rst){%>
9  <%name%> : <%if (direction == "_in"){%> IN <%}else{%> OUT<%}%> <%type.name%>
10 <%if ( self.parent.ports.nSize()!=0){%>;<%}%><%}%>
11 <%for (self.ports){%>
12 <%name%>:  <%if (direction == "_in"){%> IN <%}else{%> OUT<%}%> <%self.type.name%>
13 <%if (self!=self.parent.ports.nLast){%>;<%}%><%}%>
14  );
15 end component;
```

L'implémentation du coordinateur et des contrôleurs consiste à implémenter leurs automates de modes ainsi que le protocole de communication entre ces automates afin d'assurer la synchronisation entre eux et la bonne gestion des échanges de requêtes et de réponses. Pour le protocole de communication, nous avons choisi le standard OCP (Open Core Protocol) [102]. L'avantage de l'utilisation d'un protocole standard tel que OCP est qu'elle facilite l'intégration de différents composant dans un même système indépendamment de leurs sources et leurs types [103]. La généricité offert le standard OCP facilite l'adaptation des interfaces de ces composants à une interface commune, ce qui favorise la réutilisation des composants pour différents systèmes.

Le protocole OCP offre une grande flexibilité par rapport à d'autres standards, qui se concentrent sur l'aspect flot de données, en offrant des signaux hors bande (de contrôle) et de test [103], ce qui permet de couvrir différents types de communication que ce soit peer-to-peer, par un bus, etc. Ce protocole offre donc les outils nécessaires pour implémenter la communication entre contrôleurs et coordinateur de différentes façons selon les contraintes de performance, de ressources disponibles, etc. OCP offre une grande gamme de signaux qui peuvent être activés selon le besoin pour implémenter différents types de communication. La plupart des tailles de ces signaux est configurables, ce qui offre une grande flexibilité.

Listage 5.2 – Exemple de déclaration de composants VHDL généré par la transformation *VhdlCode*

```
1 component VHDLMonitoringModuleA  is port(
2 clk :  IN  STD_LOGIC;
3 rst :  IN  STD_LOGIC;
4 outa:   OUT std_logic_vector(0 to 31);
5 ina:   IN  std_logic_vector(0 to 31)
6 );
7 end component;
```

Le mécanisme de coordination entre le coordinateur et les contrôleurs implique, entre autres, l'envoi des requêtes de reconfiguration par les contrôleurs et l'envoi des propositions de reconfiguration par le coordinateur. Cela nécessite que le contrôleur et le coordinateur implémentent des interfaces masters. Chaque interface master nécessite une interface slave de l'autre côté de la communication, ce qui fait que chaque contrôleur et coordinateur doivent implémenter une interface master et une interface slave. Pour la communication entre les contrôleurs et le coordinateur nous avons choisi d'implémenter une communication peer-to-peer, ce qui implique que chaque contrôleur a une interface master et une interface slave et chaque coordinateur a autant d'interfaces master et slave qu'il y a de contrôleurs. Nous avons choisi ce type de communication parce qu'il est facile à implémenter et à tester et offre un plus grand parallélisme dans le traitement des requêtes et des réponses que pour une communication à travers un bus par exemple. Comme nous avons vu dans la modélisation UML des contrôleurs, un contrôleur a un port en entrée et un port de sortie (*from_coordinator* et *to_coordinator*) permettant sa communication avec le coordinateur, ce qui permet de faciliter au concepteur la modélisation de cette communication. Pour simplifier la traduction de ces ports en des

interfaces OCP, nous avons eu recours aux enregistrements (record) VHDL permettant de regrouper des signaux pour les utiliser comme type d'un port VHDL. Cette fonctionnalité permet de regrouper donc les signaux qui auront la même direction (entrée ou sortie). Pour cela, nous avons fait la fusion de l'interface master et l'interface slave de chaque contrôleur pour regrouper les signaux en entrée et les signaux en sortie, ce qui donne deux ports VHDL ayant le même type et deux directions différentes puisque les signaux en entrée d'une interface slave sont des signaux en sortie d'une interface master et vice-versa.

Pour implémenter la communication nous avons utilisé un sous-ensemble des champs OCP et nous avons paramétré leurs tailles (pour les champs paramétrables) selon le besoin de la communication. Les signaux utilisés sont les suivants [103] :

- MCmd qui est le champs de commande de l'interface master indiquant le type de la commande. Ce champ a une taille de 3 bits (non-paramétrable) pour traiter les différents types de commandes. Pour notre cas, nous utilisons deux valeurs possibles de ce champs : "000" pour indiquer qu'il n'y a pas de commande et "001" pour tous les autres types de commande (envoi de requête et de réponse par le contrôleur et envoi de proposition et de décision par le coordinateur).
- MData qui est un champ de l'interface master et qui permet d'indiquer la donnée de commande. Nous avons utilisé ce champs pour indiquer le mode concerné par la commande. Pour simplifier la génération de la table *GC* du coordinateur, nous utilisons un codage binaire des modes. Nous utilisons donc le même nombre de bits pour tous les modes. La taille du champ MData dépend donc du nombre maximal de modes traités par les contrôleurs puisque ce champ contient un codage binaire du mode. Par exemple, pour un contrôleur ayant quatre modes, ces modes sont représentés par les codes "001", "010", "011" et "100" dans l'automate de modes du contrôleur et dans la table *GC* du coordinateur, ce qui donnent un champ MData de taille 3 bits si les autres contrôleurs ont un nombre de modes inférieur ou égal à 7 mode (code "111").
- SResp ce champ de l'interface slave indique si le slave a bien reçu la commande du master. Ce champ et de taille 2 bits (non-paramétrable) indiquant les différentes possibilités de réponse (pas de réponse, validation, échec, erreur). Dans l'implémentation des contrôleurs et du coordinateur nous avons utilisé les deux premières possibilités de réponses.
- MReqInfo qui est un champ de l'interface master permettant de donner des informations supplémentaires sur la commande envoyée. Nous avons utilisé ce champ pour indiquer si la commande en cours concerne une requête, une acceptation ou un refus. Pour cela, nous avons configuré ce champs pour avoir une taille de 2 bits permettant de traiter ces possibilités du côté du contrôleur et du côté du coordinateur donnant la valeur "01" pour la requête d'un contrôleur et la proposition du coordinateur, la valeur "10" pour l'acceptation du contrôleur et l'autorisation du coordinateur et la valeur "11" pour le refus du contrôleur et du coordinateur.
- MFlag qui est un champ de l'interface master. Ce champ permet d'envoyer un signal de contrôle non lié à la commande. Son implémentation est spécifique au module. Nous avons utilisé ce champ pour la notification envoyée par le coordinateur aux contrôleurs au début et à la fin d'un processus de coordination

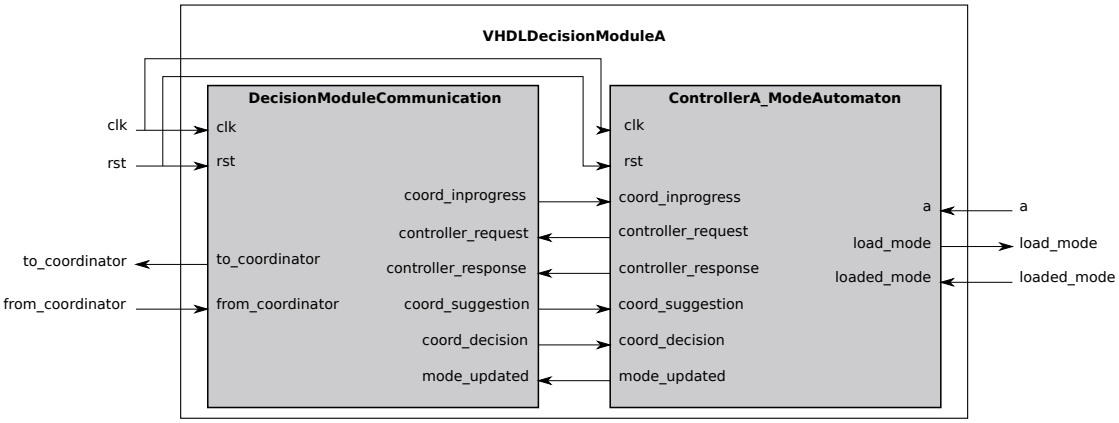


FIG. 5.14 – la structure du module de décision du contrôleur *ControllerA* généré par la transformation *CodeVHDL*

et la notification envoyée par un contrôleur au coordination après le rechargement du bitstream. Nous avons configuré ce champ pour avoir la taille d'1 bit ce qui est suffisant pour notre cas.

Parmi les déclarations de types générées dans le package *userlibrary* par la transformation *CodeVHDL*, cette transformation génère aussi la déclaration du type *coordination_interface* qui intègre les signaux OCP décrits précédemment comme le montre le listage 5.3. Ce type est un enregistrement VHDL permettant de regrouper les signaux de différents types. Le listage 5.4 montre la déclaration de l'entité VHDL correspondante au module de décision du contrôleur *ControllerA* avec les types et les directions des ports. Cette entité contient deux ports de type *coordination_interface* l'un de direction *in* regroupant les signaux en entrée des interfaces slave et master et l'autre de direction *out* regroupant les signaux en sortie des interfaces slave et master.

Listage 5.3 – Déclaration du type des interfaces entre les contrôleurs et le coordinateur

```
1 type coordination_interface is
2 record
3  OCP_MData                    :  std_logic_vector(0 to 1);
4  OCP_MCmd                     :  std_logic(0 to 2);
5  OCP_MReqInfo                 :  std_logic_vector(0 to 1);
6  OCP_MFlag                    :  std_logic;
7  OCP_SResp                    :  std_logic_vector(0 to 1);
8 end record;
```

Pour rendre possible la réutilisation des contrôleurs et coordinateurs avec d'autres types de communication (par bus, par réseau-sur-puce, etc) et/ou autres protocoles de communication, nous séparons la partie communication de la partie traitement pour ces modules. La figure 5.14 illustre la structure du module de décision du contrôleur *ControllerA* généré par la transformation *CodeVHDL*. Comme le montre la figure, le module *ControllerA_ModeAutomaton* est la traduction en VHDL de l'automate de modes de la figure 5.5 selon les concepts introduits dans le chapitre 4. Les ports du module *ControllerA_ModeAutomaton* correspondent aux entrées et sorties représentées dans

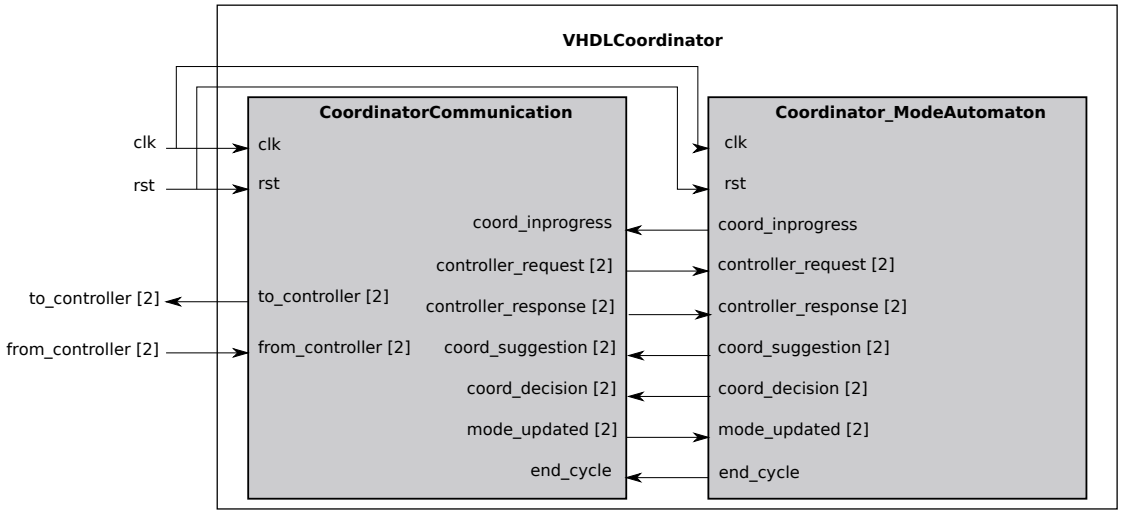


FIG. 5.15 – la structure du coordinateur généré par la transformation *CodeVHDL*

l'entête de l'automate de modes de la figure 5.5. Le module *DecisionModuleCommunication* joue le rôle d'une enveloppe OCP qui permet de traduire les signaux OCP qu'il reçoit du coordinateur à travers le port *from_coordinator* en des entrées compréhensibles par le module *ControllerA_ModeAutomaton* et les sorties qu'il reçoit de ce module en des signaux OCP à envoyer au coordinateur à travers le port *to_coordinator*. Puisque le module *DecisionModuleCommunication* est indépendant du contrôleur (ses données observées, ses modules, etc), il est réutilisé par tous modules de décision des contrôleurs. Le listage 5.5 donne un extrait du module *ControllerA_ModeAutomaton* traduisant l'automate de modes en une instruction *case* comme le montre la ligne 1 du listage. Cette instruction teste la valeur du mode courant stockée dans le signal *current_mode*. La valeur affectée à la sortie *controller_request* correspond au code du mode ciblé par le contrôleur. Une valeur "10" de la sortie *controller_response* correspond à une acceptation et une valeur "11" correspond à un refus comme le montre le listage. Pour l'implémentation du traitement des décisions du coordinateur (la partie gauche de la figure 5.5, nous avons choisi de la faire sortir de l'instruction *case* puisqu'il s'agit du même traitement pour les différents modes afin d'économiser des ressources matérielles en évitant les répétitions.

Listage 5.4 – Déclaration de l'entité VHDL correspondante au module de décision du contrôleur *ControllerA*

```
1 entity VHDLDecisionModuleA is port(
2 clk :  IN  STD_LOGIC;
3 rst :  IN  STD_LOGIC;
4 a:   IN  std_logic_vector(0 to 31) ;
5 from_coordinator:   IN  coordination_interface;
6 load_mode:   OUT std_logic_vector(0 to 1) ;
7 to_coordinator:   OUT coordination_interface;
8 loaded_mode:   IN  std_logic_vector(0 to 1)
9 );
10 end DecisionModuleA;
```

Pour la génération du coordinateur, nous avons suivi la même démarche de séparation entre communication et traitement comme le montre la figure 5.15. Les ports *to_controller* et *from_controller* sont des tableaux de 2 éléments pour gérer les deux contrôleurs du système. Le contenu du module *Coordinator_ModeAutomaton* suit la description de l'automate de modes de la figure 4.6. Ses entrées et sorties sont des tableaux dont la taille est le nombre des contrôleurs sauf pour la sortie *coord_inprogress* qui est un signal à diffuser à tous les contrôleurs et le signal *end_cycle*. Ce dernier permet la synchronisation entre le module *CoordinatorCommunication* et le module *Coordinator_ModeAutomaton* pour déclencher l'envoi des propositions du coordinateur au début d'un cycle de coordination. Le listage 5.6 donne un extrait de l'architecture du *Coordinator_ModeAutomaton*. Cet extrait donne la déclaration en VHDL des variables de l'automate dans un processus VHDL *Coordination*. Ces variables sont celles utilisées dans la présentation de l'algorithme de coordination dans la section 4.5.2. Par exemple, la variable $current_config$ permettant de stocker les modes courants des contrôleurs et initialisée à ("01", "01") qui sont les codes des modes *ModeA1* et *ModeB1*, modes initiaux des automates respectifs des contrôleurs *ControllerA* et *ControllerB*. La constante GC_table contenant toutes les configurations globales possibles a comme valeur (("01", "01", "10"), ("01", "10", "01")). Cette matrice contient donc deux lignes correspondant aux deux contrôleurs. Le nombre de colonnes de cette matrice est 3 puisqu'on autorise 3 possibilités de configuration globale dans la modélisation du coordinateur dans la figure 5.6(b).

Listage 5.5 – Extrait du module *ControllerA*

```
case current_mode is

  when "01" =>--ModeA1--

          ---------------------Transition to ModeA2--------------------
    --Transition request--
    if (a > ta2) and coord_inprogress='0' and allowed_request_ModeA2='1' then
       controller_request<="10";
    end if;
    --Transition acceptance--
    if coord_suggestion="10" and coord_decision="00" and (a >= ta1) then
       controller_response<="10";
    end if;
    --Transition refusal--
    if coord_suggestion="10" and coord_decision="00" and (a < ta1) then
       controller_response<="11";
    end if;

  when "10" =>--ModeA2--
     ...........

  when others => null;
end case;
```

Il faut noter ici que le code généré pour le coordinateur est l'implémentation en VHDL de l'algorithme de coordination présenté dans la section 4.5.2. Le concepteur peut toujours partir du code généré par défaut et faire des modifications selon ses besoins (pour changer, par exemple, le mécanisme de sélection de configurations globales à traiter lors

de la réception d'une requête).

Listage 5.6 – Extrait de l'architecture du *Coordinator_ModeAutomaton*

```
architecture Behavioral of Coordinator_ModeAutomaton is

type Coordinator_modes is (Waiting, Determine_Reconfiguration_Possibilities,
Send_Suggestions, Treat_Responses, Receive_Notifications);
type possibility_array is array (0 to 2) of integer;
type controllers_modes is array (0 to 1) of std_logic_vector (0 to 1);
type gc_table_row_type is array (0 to 2) of std_logic_vector (0 to 1);
type gc_table_type is array (0 to 1) of gc_table_type;
begin

Coordination: process(clk) is
variable current_config: controllers_modes:=("01","01");
variable requested_mode: std_logic_vector(0 to 1);
variable current_mode: Coordinator_modes:= Waiting;
variable nb_positive_responses:integer:=0;
variable nb_negative_responses:integer:=0;
variable involved_controllers: std_logic_vector( 0 to 1):=(others =>'0');
variable response_treated: std_logic_vector( 0 to 1):=(others =>'0');
variable nb_involved_controllers:integer:=0;
variable request_controller_index:integer:=0;
constant GC_table: gc_table_type:=(("01", "01", "10"),("01", "10", "01"));
variable GC_list:possibility_array:=(others=>0);
variable nb_possibilities:integer:=0;
variable nb_reconfig:possibility_array:=(others=>0);
variable cycle: integer:=0;
.....
```

5.3 Intégration du contrôle semi-distribué dans le flot de conception des systèmes embarqués

Dans cette section, nous présentons notre proposition pour l'intégration du contrôle semi-distribué dans la modélisation des systèmes embarqués reconfigurables dans Gaspard2. Cette proposition, illustrée dans la publication [128], pourrait être exploitée pour générer le code de systèmes complets avec le contrôle intégré.

Dans cette section, l'intégration du système de contrôle étudié dans ce chapitre, aux différents niveaux de modélisation de systèmes embarqués dans Gaspard2 (la modélisation de l'application, de l'architecture, de l'allocation et du déploiement) est étudiée.

La figure 5.16 illustre la modélisation de l'application qui sera contrôlée par les contrôleurs et le coordinateur étudiés dans ce chapitre. Comme le montre la figure le graphe de tâches de l'application est modélisé par des composants UML. L'application contient 4 tâches. La tâche *RetrieveInputs* sert à récupérer des entrées de l'utilisateur à partir d'un hyperterminal par exemple. La tâche *TaskA* applique des calculs sur les données en entrée. Elle sera contrôlée par le contrôleur *ControllerA*. La tâche *TaskB* applique des calculs sur les sortie de la tâche *TaskA*. La tâche *TaskB* sera contrôlée par le contrôleur *ControllerB*.

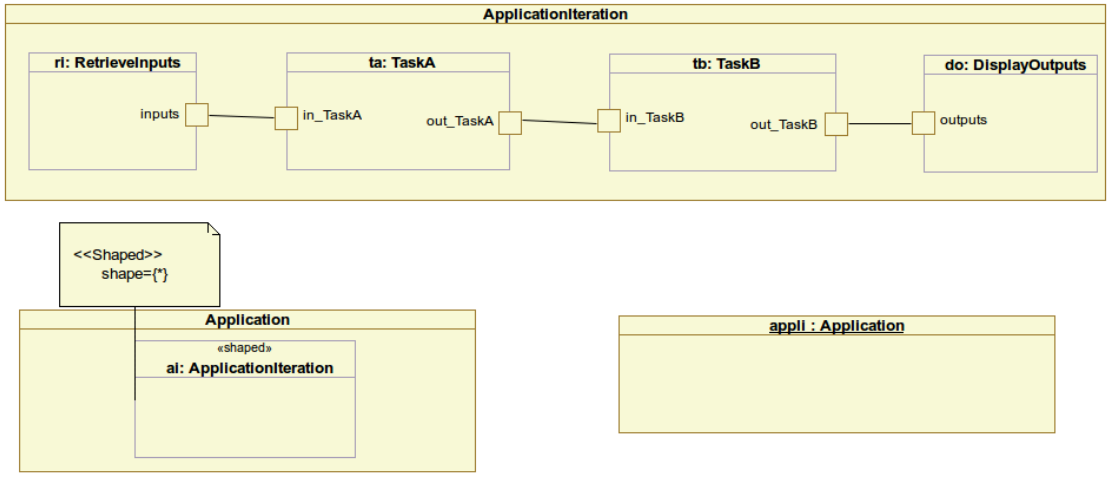


FIG. 5.16 – Modélisation de l'application

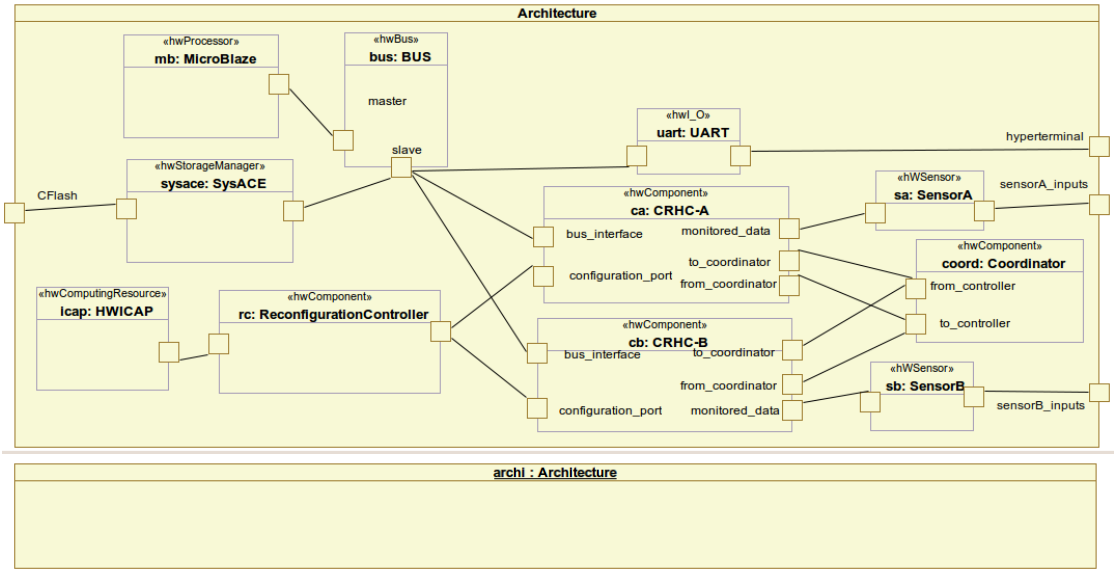


FIG. 5.17 – Modélisation de l'architecture

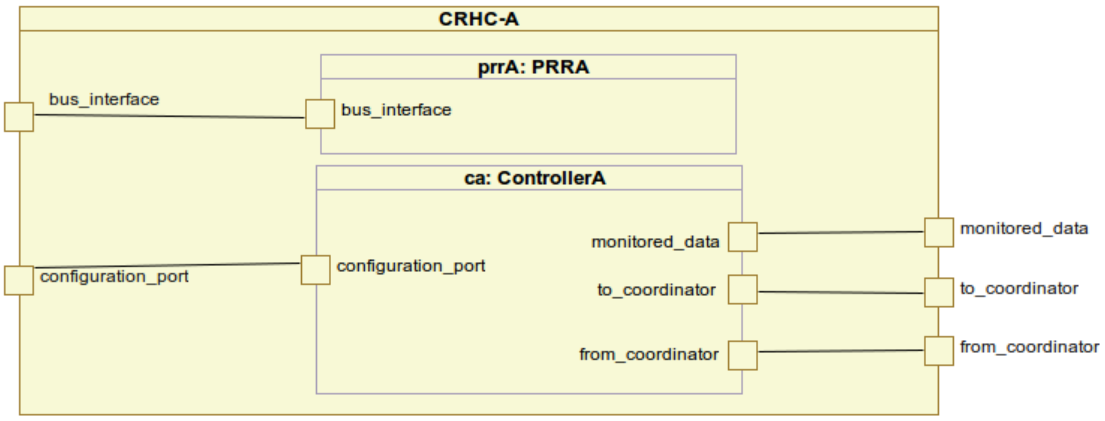


FIG. 5.18 – Modélisation d'un composant matériel reconfigurable contrôlé (CRHC)

La figure 5.17 illustre la modélisation de l'architecture sur laquelle sera mappée l'application de la figure 5.16. Cette architecture contient un microprocesseur de type *MicroBlaze*. Ce microprocesseur va exécuter les tâches de l'application modélisée dans la figure 5.16. Il communique avec le reste des composants de l'architecture à travers un bus comme le montre la figure. Les périphériques de l'architecture sont, entre autres, un module *UART* pour récupérer les entrées des utilisateurs et les afficher après le traitement, un module *SysACE* pour contrôler la mémoire Flash contenant les bitstreams et un module *HWICAP* permettant de charger les bitstreams dans la mémoire de configuration du FPGA. L'architecture contient deux composants matériels reconfigurables qui sont *CRHC-A* et *CRHC-B*. Ces composants englobent les contrôleurs et les régions reconfigurables contrôlées. C'est pour cela qu'on les a appelés *CRHCs* (Controlled Reconfigurable Hardware Components). Chaque *CRHC* a un ensemble de ports. Le port *bus_interface* permet la communication entre la région reconfigurable et le bus. Les ports *to_coordinator* et *from_coordinator* permettent la communication du contrôleur avec le coordinateur tel expliqué dans la modélisation des contrôleurs. Le port *monitored_data* permet au contrôleur de collecter les données à observer à travers un capteur (*SensorA* ou *SensorB* dans notre example). Le port *configuration_port* permet au contrôleur de lancer les reconfigurations à travers le module de reconfiguration. Dans l'exemple modélisé, la reconfiguration est contrôlée par un module matériel qui est en communication avec le port ICAP permettant de gérer l'accès distribué à ce dernier. La figure 5.18 illustre la modélisation du *CRHC-A*. Il contient le contrôleur *ControllerA* et la région reconfigurable *PRRA*.

Les tâches *TaskA* et *TaskB* modélisées dans la figure 5.16 sont des tâches exécutées respectivement par les régions reconfigurables *PRRA* et *PRRB*. Du côté du processeur, ces deux tâches consistent en l'envoi des données aux régions reconfigurables et la récupération des résultats ensuite. Dans notre exemple, chaque région reconfigurable exécute différents modes de la même tâche de l'application. Donc, pour chaque configuration d'une région reconfigurable, il faut indiquer la version de la tâche à allouer à la région. La figure 5.19 illustre les deux configurations MARTE permettant de modéliser les différentes configurations de la région reconfigurable *PRRA*. Chaque configuration

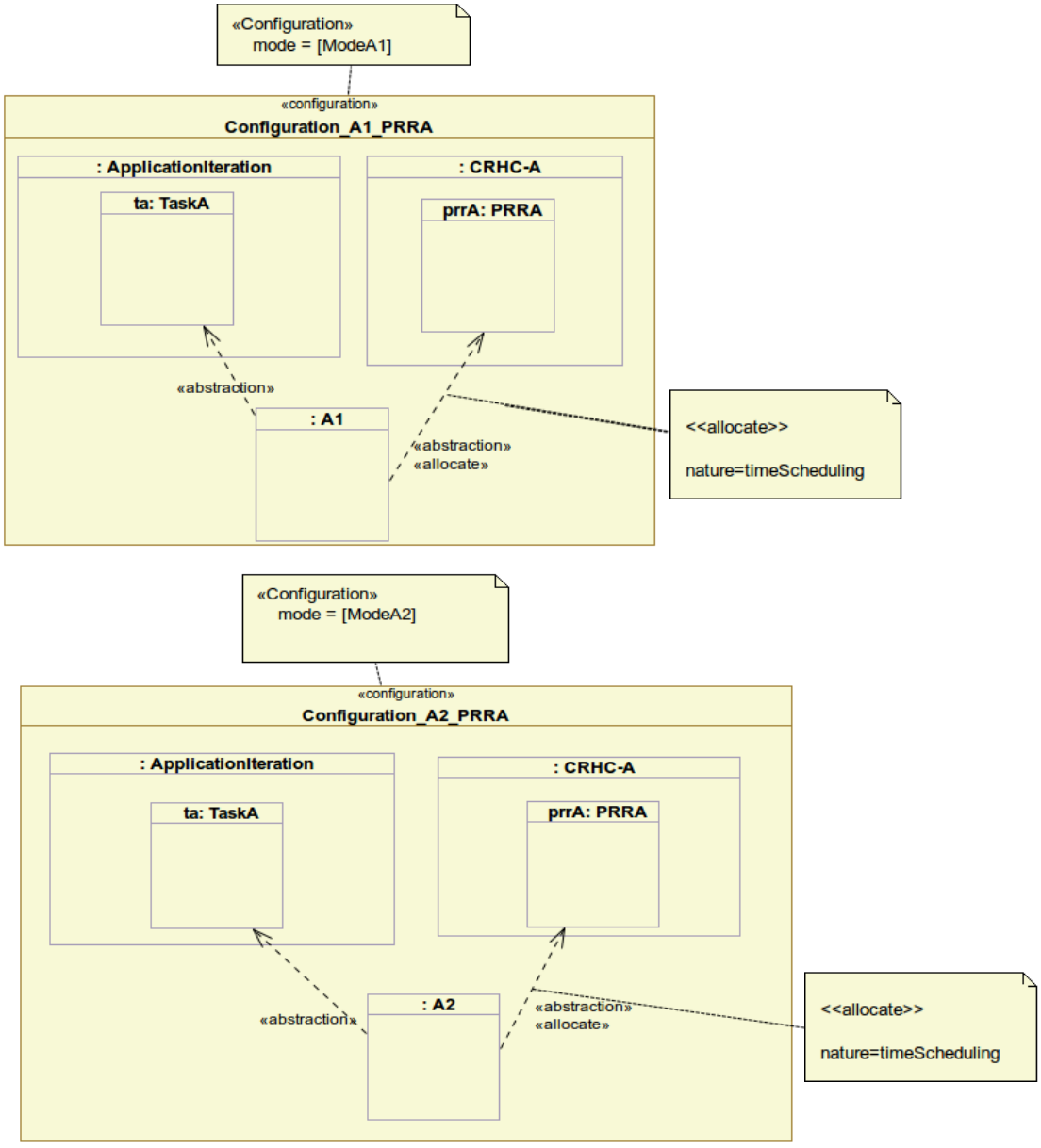


FIG. 5.19 – Modélisation des configurations des régions reconfigurables

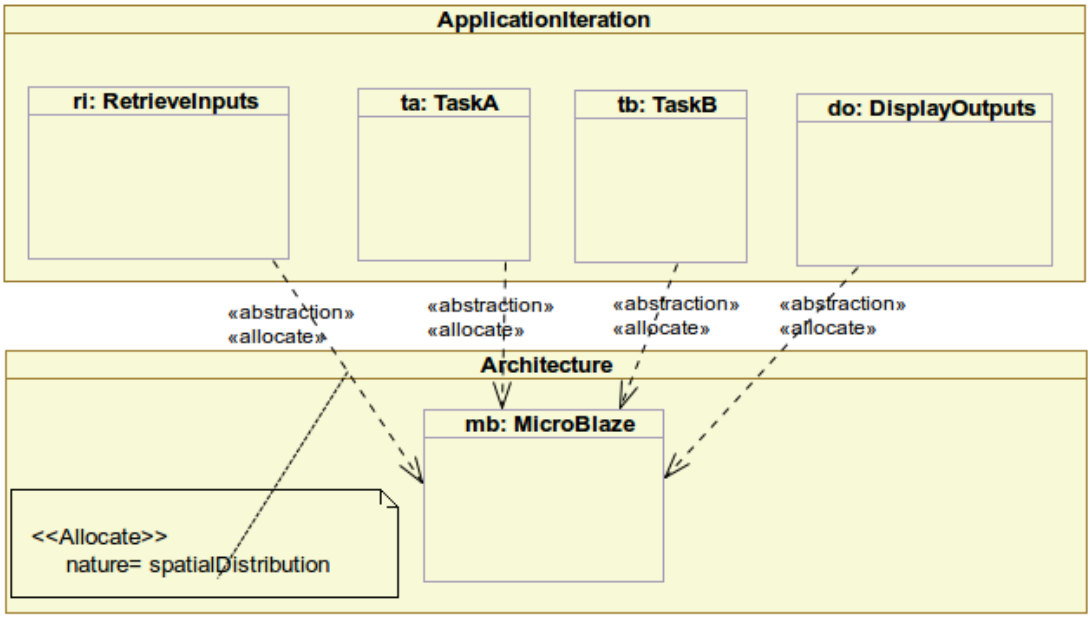


FIG. 5.20 – Allocation des tâches statiques au processeur

correspond à un des modes modélisés dans le *ModeBehavior* du contrôleur de la région (ici, c'est celui de la figure 5.4). Comme le montre la figure, dans la configuration correspondante au mode *ModeA1*, c'est la version *A1* de la tâche *TaskA* qui est utilisée et elle est allouée à la région reconfigurable *PRRA*. Le stéréotype *«allocate»* est utilisé pour indiquer l'allocation de la tâche sur la région. La valeur *timeScheduling* de l'attribut *nature* indique que l'allocation n'est pas statique dans le temps.

Il faut noter que dans notre exemple, du côté du processeur la tâche *TaskA* ne change pas puisqu'il s'agit toujours des mêmes commandes envoyées à la région reconfigurable *PRRA*. Donc, cette tâche peut être vue comme statique pour le processeur et reconfigurable pour la région *PRRA*. La figure 5.20 illustre l'allocation des tâches au processeur. Puisque cette allocation est statique, l'attribut *nature* du stéréotype *«allocate»* a la valeur *spatialDistribution*. Dans le cas où les tâches exécutées par le processeur sont reconfigurables (plusieurs versions de la même tâche), on a besoin d'un *ModeBehavior* pour chaque tâche. La configuration correspondante à chaque mode détermine la version de la tâche à exécuter par le processeur. Plusieurs autres possibilités peuvent être modélisées pour l'aspect dynamique du système. Une région reconfigurable peut implémenter des tâches complètement différentes à condition qu'elles aient la même interface. Les tâches exécutées par le processeur peuvent être aussi changer en changeant par exemple la répartition des tâches entre le processeur et les accélérateurs matériels statiques ou reconfigurables. Dans ce cas, on a besoin d'un *ModeBehavior* pour tout le système. La configuration correspondante à chaque mode de ce *ModeBehavior* décrit les tâches allouées à chacune des ressources de calcul selon le mode actif.

Le déploiement dans Gaspard2 permet d'attacher à chaque composant élémentaire du système (faisant parti de l'application ou de l'architecture) un code existant dans

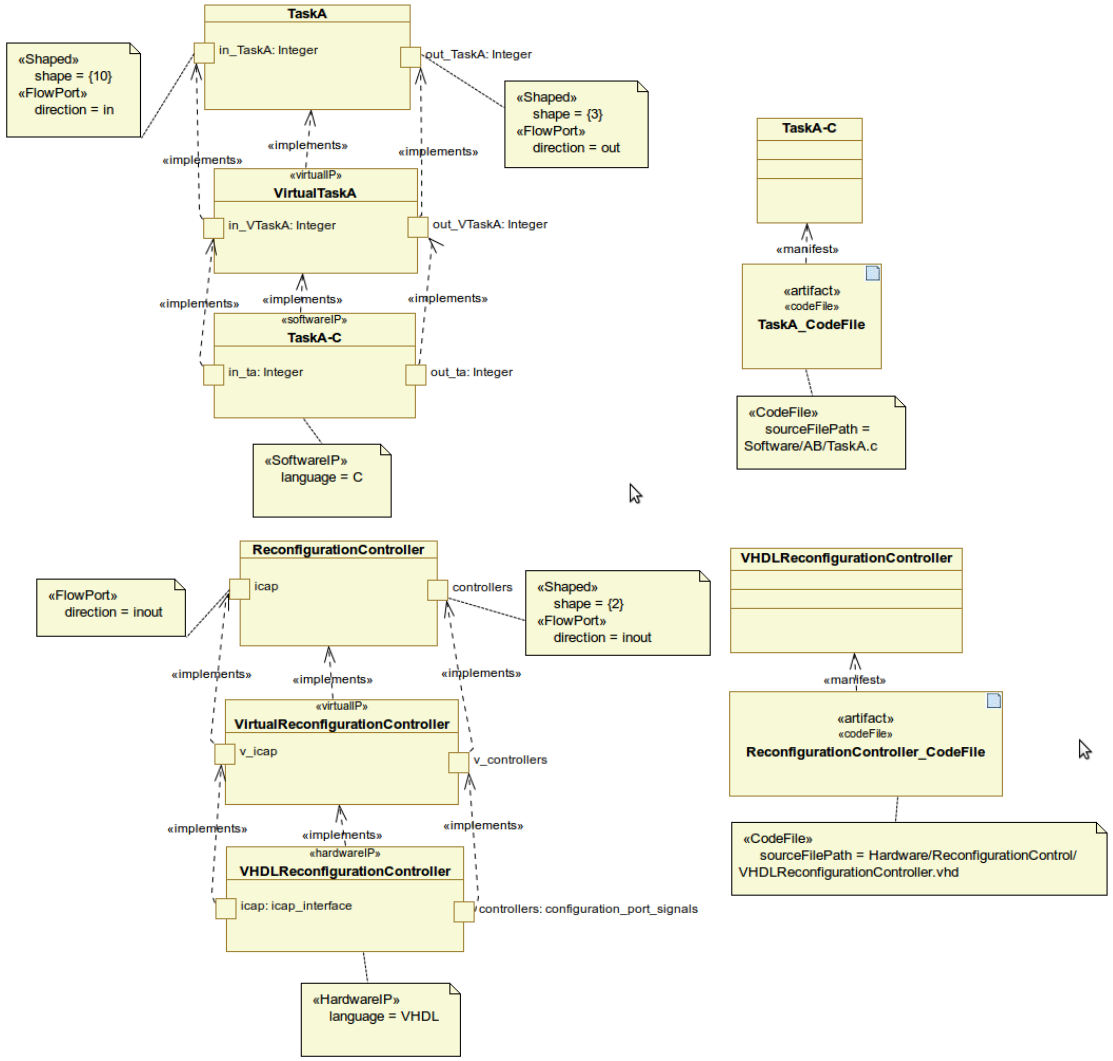


FIG. 5.21 – Exemple de déploiement des composants logiciels et matériels

la bibliothèque d'IP. Ce déploiement se fait en deux étapes. La première étape permet d'associer les *SoftwareIP* et *HardwareIP* respectivement aux composants logiciels et matériels à déployer. La deuxième étape consiste à attacher les *CodeFile* à ces *SoftwareIP* et *HardwareIP*. Les tâches du processeur peuvent être déployées avec du code C par exemple. Les composants matériels peuvent être déployés par des modules en VHDL. La figure 5.21 illustre le déploiement d'un composant logiciel et d'un composant matériel du système.

Le déploiement des régions reconfigurables est différent des déploiements précédents puisqu'il implique deux points. Le premier est le module matériel implémentant la région reconfigurable. Ici, on a besoin du déploiement de l'interface de ce module (qui reste statique) pour assurer la communication entre la région reconfigurable et le reste des composants. Le deuxième est le contenu de la région reconfigurable qui change

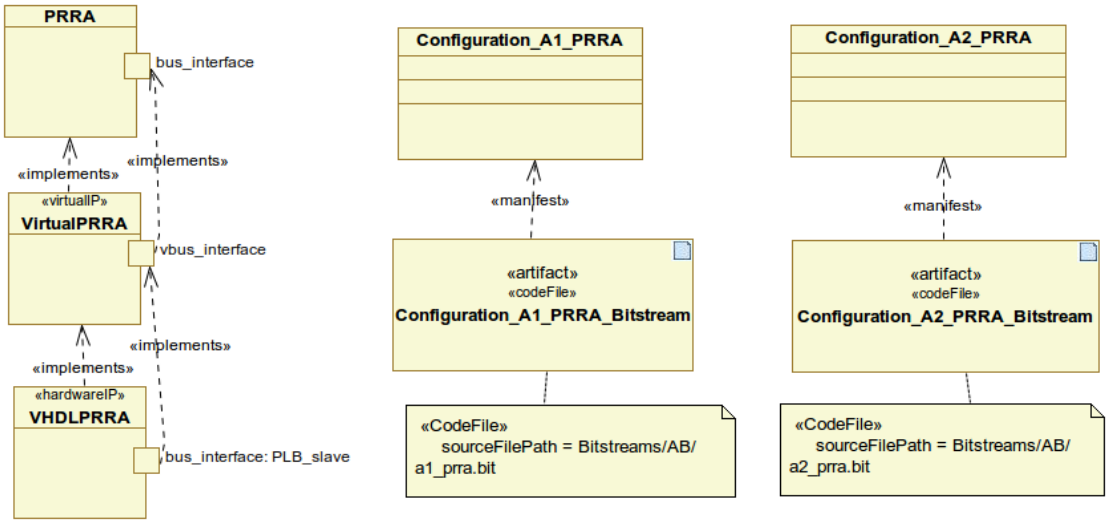


FIG. 5.22 – Déploiement des régions reconfigurables

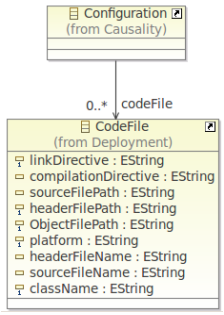


FIG. 5.23 – Extension du métamodèle de déploiement pour supporter le déploiement des configurations

selon la configuration active. Ce contenu est un bitstream qui dépend de l'association entre la tâche de l'application et la région reconfigurable. Ici, la région reconfigurable ne peut être déployée par un *CodeFile* puisque cette région peut être implémentée par plusieurs codes selon la configuration active. Pour cela, nous proposons de déployer le module matériel représentant la région reconfigurable en utilisant un *HardwareIP* et associer un *CodeFile* aux différentes configurations de la région. Ici, le *CodeFile* représente un bitstream. La figure 5.22 illustre le déploiement de la région *PRRA*. Par conséquent, le métamodèle de déploiement de Gaspard2 doit être étendu pour supporter le déploiement des configurations MARTE. La figure 5.23 illustre l'extension proposée.

5.4 Conclusion

Dans ce chapitre, une approche de conception de contrôle basée sur l'IDM, dans le cadre de l'environnement Gaspard2, a été proposée. Elle vise à rendre transparents aux

concepteurs les détails d'implémentation de bas niveau et d'automatiser la génération du code, permettant ainsi d'accélérer la phase de conception et d'éviter les erreurs dues à la manipulation directe des ces détails. Dans cette approche, la modélisation de la structure des systèmes de contrôle se base sur les concepts de composants d'UML et de MARTE. Le comportement des modules de décision a été modélisé en utilisant des machines à états UML appliquant les concepts MARTE pour le comportement modal qui sont inspirés du formalisme des automates de modes. Des extensions au profil MARTE ont été proposées dans ce chapitre afin de pouvoir intégrer les concepts de prise de décision semi-distribuée. Ces concepts concernent principalement les notions de requête, d'acceptation et de refus de reconfiguration, ainsi que la notion de table de configurations globales gérée par le coordinateur. A travers ces concepts, le concepteur ne modélise que le minimum nécessaire pour la génération des systèmes de contrôle. Tout le mécanisme de communication et de coordination derrière sera géré par les transformations de modèles et l'outil de génération de code, ce qui facilite et accélère la conception.

Le langage cible dans ce travail pour la génération des systèmes de contrôle est le VHDL, afin de pouvoir les implémenter d'une façon matérielle assurant ainsi la performance du contrôle. Pour arriver jusqu'à la génération du code, une chaîne de transformation a été utilisée. Cette chaîne réutilise certaines transformations qui existent déjà dans Gaspard2 et propose de nouvelles transformations adaptées à la cible VHDL. Le modèle résultant de cette chaîne de transformations est donné à un outil de génération de code basé sur des règles de génération adaptées aux concepts du contrôle semi-distribué. Cet outil permet, entre autres, de traduire les échanges entre les contrôleurs et le coordinateur à travers des variables selon les concepts présentés dans le chapitre précédent, en des communications matérielles basées sur le protocole standard OCP. Les systèmes de contrôle générés peuvent être intégrés manuellement à des systèmes reconfigurables pour une implémentation physique sur FPGA.

Ce chapitre propose aussi une approche pour l'intégration du modèle de contrôle dans le flot de conception des systèmes reconfigurables sur FPGA dans l'environnement Gaspard2 afin de pouvoir générer tout un système reconfigurable intégrant le contrôle semi-distribué. Cette proposition se limite au niveau modélisation, mais elle pourrait être facilement exploitée (en développant les règles de transformations de modèles et de génération du code nécessaires) pour générer le code de systèmes complets avec le contrôle intégré.

Pour vérifier le bon fonctionnement des systèmes de contrôle générés et pour estimer leurs coûts de communication en termes de ressources matérielles utilisées, des outils de simulation, de synthèse et d'implémentation doivent être utilisés. Dans notre cas, il s'agit des outils Xilinx puisque la plate-forme cible pour l'implémentation physique dans ce travail est une Virtex-6 de Xilinx. Le processus de validation et d'implémentation est détaillé dans le chapitre suivant en prenant une application de traitement d'images comme étude de cas.

Chapitre 6

Etude de cas

6.1 Introduction

Dans ce chapitre, nous validons notre approche basée sur le contrôle semi-distribué par une application de traitement d'images. Cette validation comprend : 1) l'évaluation de l'efficacité de notre approche en termes de flexibilité, réutilisation et scalabilité, 2) la validation du fonctionnement du contrôle semi-distribué par simulation, 3) l'évaluation

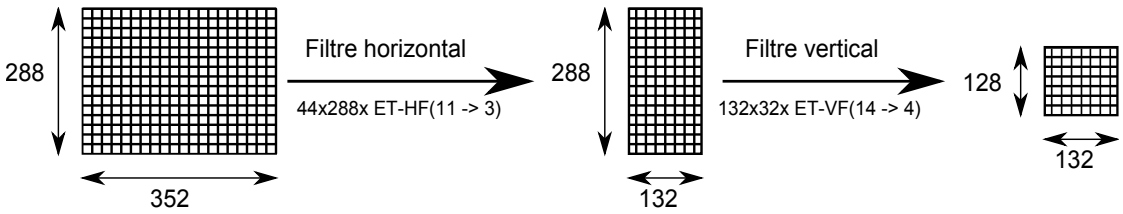


FIG. 6.1 – Vue globale de l'application Downscaler

du coût de cette approche en termes de ressources matérielles et 4) l'intégration du contrôle semi-distribué dans un système implémenté sur FPGA.

Nous commençons par présenter l'étude de cas et l'objectif du système de contrôle pour l'application étudiée. Nous passons ensuite à la modélisation du contrôle en utilisant les extensions proposées dans le chapitre précédent. Le code VHDL généré par la chaîne de transformations de modèles est ensuite validé par simulation avant d'être intégré dans un système reconfigurable implémenté sur FPGA.

6.2 Présentation de l'étude de cas

6.2.1 L'application

L'application étudiée *Downscaler* est le redimensionnement d'images, qui peut être utilisé par exemple dans le streaming pour des appareils de petite taille tels que les téléphones portables, en redimensionnant les images des vidéos. Les images d'entrée sont au format CIF (Common Intermediate Format : 352x288 pixels). Ceux de sortie sont au format 132x128 pixels. L'application est composée de deux filtres : un filtre horizontal et un filtre vertical. En appliquant le filtre horizontal sur l'image en entrée, sa taille est réduite à 132x288 pixels. Ensuite, en appliquant le filtre vertical, on obtient la taille cible (132x128 pixels), comme le montre la figure 6.1. Chaque filtre est composé d'une répétition de tâches élémentaires sur les blocs d'une image. Le filtre horizontal est composé d'une répétition de 44x288 fois d'une tâche élémentaire *ET-HF* qui s'applique sur un bloc de taille 11 pixels pour donner un bloc de 3 pixels. Le filtre vertical est une répétition de 132x32 fois d'une tâche élémentaire *ET-VF* qui s'applique sur un bloc de taille 14 pixels pour donner un bloc de 4 pixels), comme le montre la figure 6.1.

Pour assurer la performance de l'application, les deux filtres sont implémentés en utilisant des accélérateurs matériels. Chaque filtre peut être implémenté par un ou plusieurs accélérateurs matériels. En utilisant différentes configurations d'un accélérateur, il peut implémenter simultanément une ou plusieurs tâches élémentaires. Cela permet de réduire le temps d'exécution au coût d'une augmentation de ressources matérielles et consommation d'énergie. En variant les configurations durant l'exécution, le système peut s'adapter aux différentes contraintes de performance et de consommation.

La variété des possibilités de parallélisme liée à cette application permet de valider la scalabilité de notre modèle de contrôle semi-distribué en variant le nombre des régions contrôlées. Cela permet aussi de réutiliser les contrôleurs pour les régions implémentant le même type de filtre accélérant ainsi la phase de conception.

Modes des régions	Taille du bloc d'entrée (pixels)	Taille du bloc de sortie (pixels)
$HFilter\_mode_1$	35	12
$HFilter\_mode_2$	19	6
$HFilter\_mode_3$	11	3
$VFilter\_mode_1$	41	16
$VFilter\_mode_2$	23	8
$VFilter\_mode_3$	14	4

TAB. 6.1 – Les tailles des blocs d'entrée et de sortie pour les différents modes des régions reconfigurables

6.2.2 L'objectif du contrôle

L'objectif du contrôle dans cette étude de cas est d'adapter le système aux changements liés aux exigences de performance et de consommation durant l'exécution. Il faut noter que l'objectif dans cette étude de cas n'est pas d'assurer l'optimisation de la performance et la consommation d'énergie, mais de prouver la faisabilité de notre approche de contrôle avec une implémentation physique.

Chaque accélérateur matériel associé à l'un des deux filtres est implémenté dans une région reconfigurable en utilisant trois versions différentes en termes de performance et de consommation. Ces versions appliquent des degrés de parallélisme différents en utilisent des tailles différentes de blocs d'entrée, comme le montre le tableau 6.1. Le premier mode de chacun des types d'accélérateurs ($HFilter\_mode_1$ et $VFilter\_mode_1$) est un mode qui correspond à des accélérateurs effectuant simultanément 4 tâches élémentaires. Le deuxième et troisième modes correspondent à des accélérateurs effectuant respectivement 2 et 1 tâches élémentaires. Le premier mode correspond donc à la plus haute performance, pour les deux types de filtres, mais en même temps à la plus haute consommation.

Le système de contrôle semi-distribué utilisé pour cette étude de cas alloue un contrôleur à chaque région reconfigurable afin de contrôler son comportement et de permettre la permutation entre différents modes selon les exigences en termes de performance et de consommation. Les décisions de reconfiguration des contrôleurs distribués sont coordonnées par le coordinateur qui permet de vérifier que la configuration globale du système respecte les contraintes indiquées dans la table GC.

Le problème de contrôle traité dans cette étude de cas n'est pas très complexe. L'objectif ici est de valider le fonctionnement du modèle de contrôle semi-distribué en mettant en oeuvre ses différents aspects (observation, décision semi-distribuée, etc). Les entrées des modules d'observation pour tous les contrôleurs sont le niveau de performance requis par l'utilisateur et le niveau courant de la batterie, qui sont des données globales. Cependant, chaque contrôleur les traite selon ses contraintes locales, comme sera expliqué plus loin dans ce chapitre. Dans la section suivante, la modélisation UML MARTE du système de contrôle pour l'étude de cas est présentée.

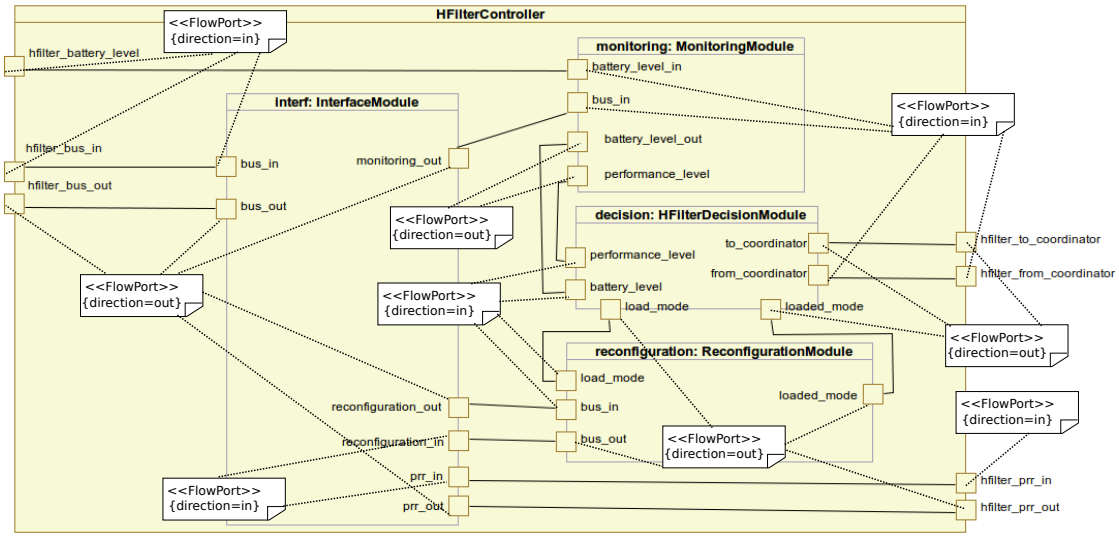


FIG. 6.2 – Modélisation du contrôleur *HFilterController*

6.3 Modélisation

6.3.1 Modélisation de la structure des contrôleurs

La figure 6.2 illustre la structure d'un contrôleur d'une région implémentant le filtre horizontal. Celle d'un filtre vertical suit le même concept. Chaque contrôleur contient quatre composants : modules d'observation, de décision, de reconfiguration et de communication avec le bus. Ce dernier module permet la communication entre le processeur et le contrôleur. Les contrôleurs à générer partageront la même interface processeur avec la région contrôlée. De ce fait, le module de communication avec le bus permet d'extraire les données destinées au contrôleur, de celles destinées à la région contrôlée. Comme le montre la figure 6.2, les ports *hfilterbus_in* et *hfilterbus_out* du composant *BusCommunicationModule* permettent de recevoir les requêtes du processeur et lui envoyer les réponses. Les ports *prr_in* et *prr_out* permettent de recevoir les réponses de la région reconfigurable et de lui envoyer les requêtes du processeur. Le port *monitoring_out* de ce composant permet d'envoyer au module d'observation le niveau de performance requis par l'utilisateur et transmis par le processeur. Ce niveau de performance peut prendre trois valeurs 1, 2 et 3. La valeur 1 correspond au plus haut niveau de performance requis et la valeur 3 au plus bas niveau.

Les ports *reconfiguration_in* et *reconfiguration_out* du composant *BusCommunicationModule* permettent la communication entre le processeur et le module de reconfiguration. En fait, dans notre implémentation du mécanisme de reconfiguration, nous utilisons le processeur qui lit, après le traitement de chaque image de la séquence vidéo, un registre du module de reconfiguration qui est dédié à stocker la configuration requise par le module de décision. Si la valeur de ce registre est non nulle, le processeur lance la reconfiguration correspondante à travers l'ICAP. Cette implémentation peut aussi être remplacée par un module matériel qui communique avec l'ICAP [63]. Ce type de module a été proposé

par certains travaux [63]. La reconfiguration parallèle n'est pas encore possible pour les systèmes contenant un seul FPGA puisqu'un seul port de configuration peut être actif à la fois. Le système de contrôle pourrait profiter de la reconfiguration parallèle s'il ciblait un système contenant plus qu'un port de configuration tel qu'un système multi-FPGA. Or, notre étude de cas cible un système mono-FPGA. Les ports *reconfiguration_in* et *reconfiguration_out* du composant *BusCommunicationModule* permettent donc d'envoyer la requête de lecture du processeur au module de reconfiguration et de récupérer le résultat de la lecture pour l'envoyer après au processeur à travers le port *hfilterbus_out*.

Les entrées du module d'observation *MonitoringModule* sont donc le niveau courant de la batterie et le niveau de performance requis par l'utilisateur. Le niveau de batterie est donné par une connexion directe à un capteur de batterie à travers le port *battery_level_in* du module d'observation. Le changement du niveau de performance requis par l'utilisateur à travers un bouton poussoir est traité comme une interruption par le processeur et il est envoyé par ce dernier à tous les contrôleurs à travers le bus. Les sorties du composant *MonitoringModule* sont le niveau de batterie et le niveau de performance envoyés au module de décision. La sortie *battery_level_out* et une transmission directe de l'entrée *battery_level_in*. La sortie *performance_level* est déterminée à partir de la requête envoyée par le processeur à travers le port *bus_in*.

A l'exception des entrées envoyées par le module d'observation, le reste des ports du module de décision sont les mêmes que ceux décrits dans le chapitre précédent, et qui permettent de communiquer avec le coordinateur et le module de reconfiguration. Ce dernier communique aussi avec le module BusCommunicationModule afin de recevoir la requête de lecture du registre de reconfiguration envoyée par le processeur et de lui envoyer la réponse à cette requête.

Les interfaces du bus sont représentées par deux types de ports : des ports qui ont la direction *in* regroupant les signaux en entrée à une interface et les ports qui ont la direction *out* regroupant les signaux de sortie. Ces deux types de ports seront traduits durant l'étape de génération de code en deux enregistrements VHDL. De même, la communication entre le contrôleur et le coordinateur est modélisée par deux ports regroupant les signaux d'entrée et ceux de sortie, comme nous avons expliqué dans le chapitre précédent. Cela permet de simplifier la modélisation. Le concepteur n'a que modéliser deux ports pour chaque type de communication, au lieu de modéliser tous les ports correspondant aux différents signaux utilisés pour cette communication.

6.3.2 Modélisation du système du contrôle et du coordinateur

La figure 6.3 illustre le système de contrôle. Il contient deux instances du contrôleur *HFilterController* et deux instances du contrôleur *VFilterController* permettant de contrôler, respectivement, deux régions implémentant le filtre horizontal et deux autres implémentant le filtre vertical. Le système contient aussi le coordinateur. Comme le montre la figure 6.4, la table *GC* du coordinateur n'autorise que les configurations globales où toutes les régions implémentent le même numéro de mode, qui est un exemple de tables *GC* possibles. Le tableau 6.2 traduit le contenu de la propriété *allowedModeCombination* en une table *GC* sous la forme utilisée dans le chapitre 4.

	Numéro de la configuration globale		
	1	2	3
Région 1	$HFilter\_mode1$	$HFilter\_mode2$	$HFilter\_mode3$
Région 2	$HFilter\_mode1$	$HFilter\_mode2$	$HFilter\_mode3$
Région 3	$VFilter\_mode1$	$VFilter\_mode2$	$VFilter\_mode3$
Région 4	$VFilter\_mode1$	$VFilter\_mode2$	$VFilter\_mode3$

TAB. 6.2 – La table GC pour le système à 4 régions reconfigurables

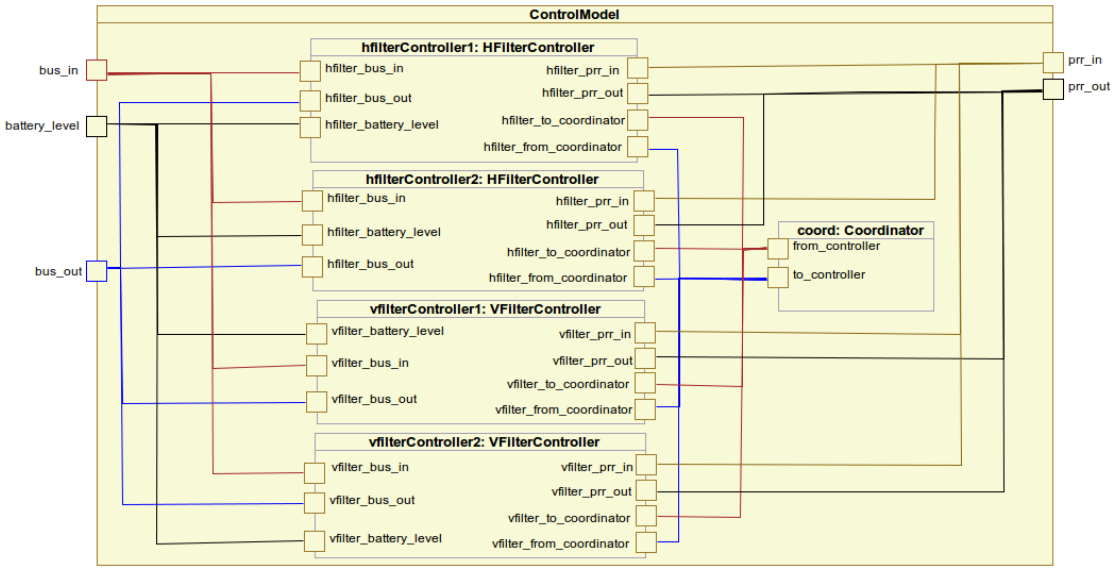


FIG. 6.3 – Modélisation du système de contrôle

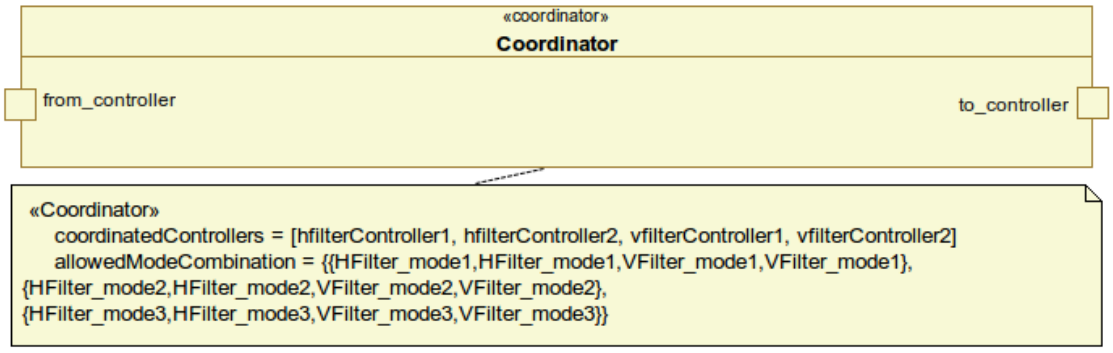


FIG. 6.4 – Modélisation du coordinateur

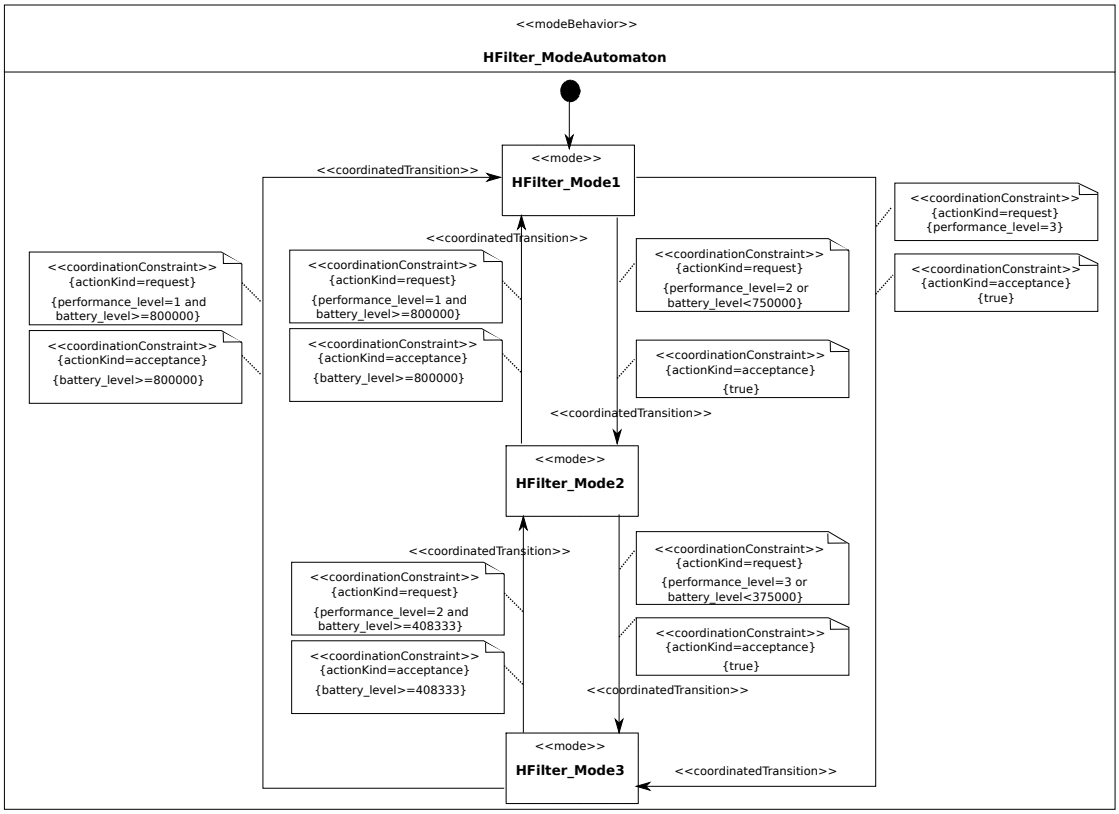


FIG. 6.5 – Modélisation de l'automate de modes du contrôleur *HFilterController*

6.3.3 Modélisation des automates de modes des contrôleurs

La figure 6.5 illustre l'automate de modes du contrôleur de la figure 6.2. La stratégie de chaque contrôleur dans cette étude de cas est de favoriser l'économie d'énergie dans la mesure du possible. De ce fait, les décisions des contrôleurs se basent sur les règles suivantes :

Les conditions des requêtes de reconfiguration
Etant dans un mode $H/VFilter\_mode_{j1}$, le contrôleur décide d'aller dans un mode $H/VFilter\_mode_{j2}$ qui consomme moins si l'utilisateur demande un niveau de performance plus bas que le niveau courant, ou si le niveau de la batterie se trouve au-dessous d'un certain seuil qui ne permet plus de rester dans le mode $H/VFilter\_mode_{j1}$. Par exemple, comme le montre la figure 6.5, si le mode courant est $HFilter\_mode1$ et que le niveau de performance requis par l'utilisateur est 2, ou si le niveau de la batterie est inférieur à 750000, le contrôleur envoie une requête au coordinateur pour passer au mode $HFilter\_mode2$. Les conditions des requêtes pour passer à un mode qui consomme moins utilisent la formule :

$$battery\_level/H_{j1} < a_{j1,j2} x FB/H_1 \tag{6.1}$$

avec H_j (V_j pour le filtre vertical) est l'énergie consommée par cycle de la région contrôlée

en mode $H/VFilter\_mode_j$, *battery_level* est le niveau de la batterie envoyé par le capteur de batterie et FB est l'énergie d'une batterie pleine. Le terme $a_{j1,j2}$ est un pourcentage qui permet de déterminer le seuil de la batterie au dessus duquel une reconfiguration du mode $H/VFilter\_mode_{j1}$ au mode $H/VFilter\_mode_{j2}$ est requise. Dans notre étude de cas, ce terme prend les valeurs suivantes : $a_{1,2} = 75\%$ et $a_{2,3} = 75\%.75\%$ pour les deux types de contrôleurs. Ces deux pourcentages sont utilisés pour envoyer des requêtes de reconfiguration pour passer respectivement du mode $H/VFilter\_mode1$ au mode $H/VFilter\_mode2$, et du mode $H/VFilter\_mode2$ au mode $H/VFilter\_mode3$.

Le terme $battery\_level/H_{j1}$ donne l'autonomie restante en cycles si la région reste dans le mode $H/VFilter\_mode_{j1}$ jusqu'à ce que la batterie se trouve vide. Ici, on néglige la consommation des autres régions puisque chaque contrôleur à une vision locale du problème de contrôle. Cependant, appliquer cette contrainte à tous les contrôleurs permet de l'appliquer à tout le système. Il faut noter ici que cette consommation ne fait pas la différence entre consommation dynamique et consommation statique. On considère que les valeurs utilisées représentent la consommation totale par cycle de chaque mode d'une région.

Le terme FB/H_1 donne l'autonomie en cycles de la région si elle reste dans le mode qui consomme le plus et qui est le plus performant (autonomie minimale). En faisant un compromis entre la performance et la consommation, le contrôle des régions reconfigurables vise à améliorer l'autonomie par rapport à une configuration statique implémentant le "mode" le plus performant, qui est en même temps celui qui consomme le plus. Pour cette raison, l'autonomie minimale est prise comme référence dans la partie droite de l'inéquation 6.1.

Dans notre étude de cas, les régions implémentant le filtre vertical consomment plus que les autres puisque les blocs en entrée et en sortie de ce filtre sont de plus grandes tailles par rapport au filtre horizontal. Les valeurs prises pour la consommation par cycle de ces régions selon les modes actifs sont les suivantes : $H_1 = 6$, $H_2 = 4$, $H_3 = 2$, $V_1 = 7$, $V_2 = 5$, et $V_3 = 3$. Ces valeurs sont exprimées en unité de consommation qui peut être de l'ordre de $10^{-10}J$. Elles sont prises en tant qu'exemples pour la validation du modèle de contrôle proposé. Elles peuvent être remplacées par des valeurs effectives si on dispose du matériel nécessaire pour les mesures de consommation d'énergie durant l'exécution de l'application. La valeur du niveau maximal de la batterie prise ici est $FB = 1000000$ désignant une énergie de l'ordre de $100\mu J$ disponible dans la batterie. Cette petite valeur est utilisée afin d'accélérer le processus de la simulation (arriver tôt aux seuils qui déclenchent les reconfigurations) que nous présentons dans la suite de ce chapitre. En reformulant l'inéquation 6.1, une requête de reconfiguration est envoyée pour passer du mode $H/VFilter\_mode_{j1}$ au mode $H/VFilter\_mode_{j2}$ lorsque la condition suivante est valide :

$$battery\_level < a_{j1,j2}.FB.H_{j1}/H_1 \tag{6.2}$$

En remplaçant les termes par leurs valeurs dans la condition du passage du mode $HFilter\_mode1$ au mode $HFilter\_mode2$, le niveau de batterie qui déclenche la requête de reconfiguration est $0.75x1000000x6/6 = 7500000$ comme le montre la figure 6.5.

Etant dans un mode $H/VFilter\_mode_{j2}$, le contrôleur décide d'aller dans un mode $H/VFilter\_mode_{j1}$ qui consomme plus si l'utilisateur demande un niveau de performance

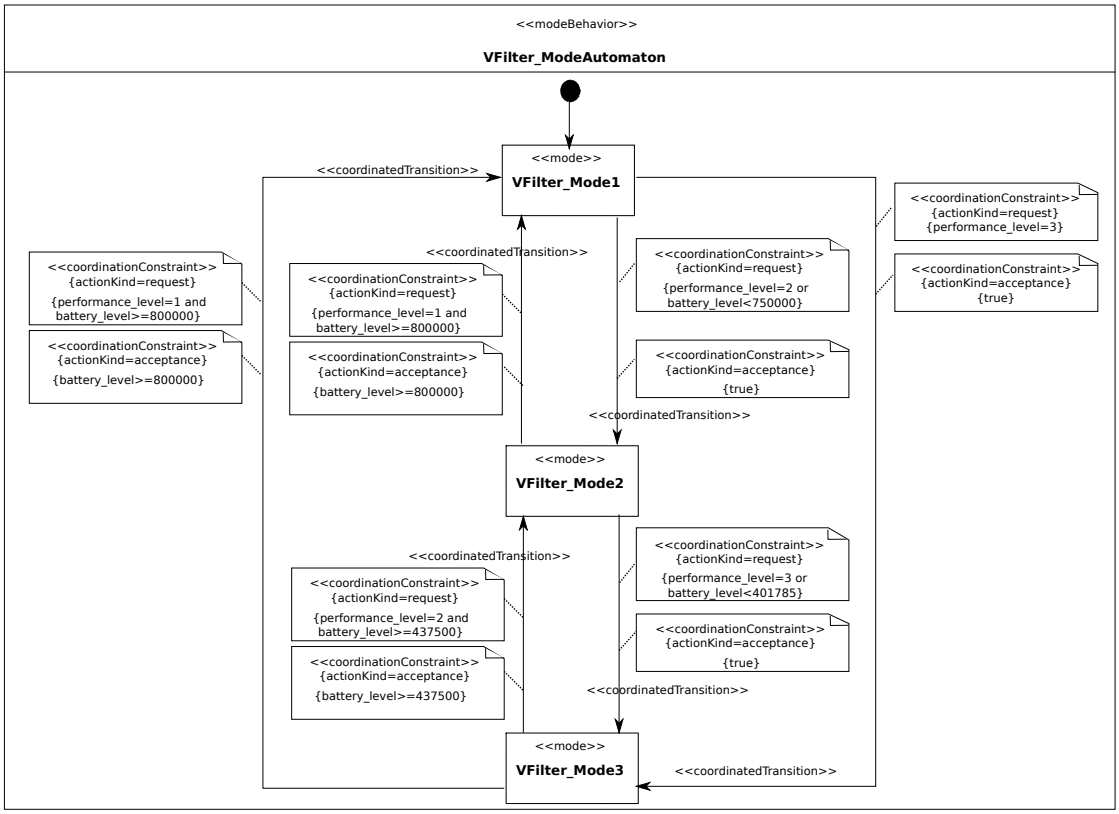


FIG. 6.6 – Modélisation de l'automate de modes du contrôleur *VFilterController*

plus haut que le niveau courant et que le niveau de la batterie se trouve au-dessus d'un certain seuil qui permet de passer au mode $H/VFilter\_mode_{j1}$. Par exemple, comme le montre la figure 6.5, si le mode courant est $HFilter\_mode2$ et que le niveau de performance requis par l'utilisateur est 1 et le niveau de la batterie est supérieur à 800000, le contrôleur envoie une requête au coordinateur pour passer au mode $HFilter\_mode1$. Les conditions des requêtes pour passer à un mode qui consomme plus suivent la formule :

$$battery\_level/H_{j1} >= b_{j2,j1}.FB/H_1 \tag{6.3}$$

Il faut noter que la valeur de $b_{j2,j1}$ est supérieure à celle de $a_{j1,j2}$ de l'équation 6.1 pour éviter qu'une fois dans le mode $H/VFilter\_mode_{j1}$, les contraintes de l'équation 6.1 entraînent l'envoi une requête pour revenir au mode $H/VFilter\_mode_{j2}$ très tôt, ce qui peut entraîner une boucle infinie. Dans notre étude de cas, nous prenons les valeurs suivantes des termes $b_{j2,j1}$: $b_{2,1} = b_{3,1} = 75\% + 5\%$ et $b_{3,2} = 75\%.75\% + 5\%$.

Les conditions de l'acceptation/refus des propositions de reconfiguration
Comme nous avons précisé dans le chapitre 4, le contrôleur accepte ou refuse les propositions de reconfiguration envoyées par le coordinateur en se basant sur les données d'observation et sur sa stratégie de contrôle, qui est ici de favoriser l'économie d'énergie. Le contrôleur traite donc les propositions de reconfiguration de la manière suivante. Si la

proposition concerne le passage à un mode qui consomme moins, le contrôleur l'accepte directement. Sinon, le contrôleur vérifie le niveau de performance requis et la contrainte de l'équation 6.1 afin d'accepter ou refuser la proposition comme le montre la figure 6.5. Ici, les contraintes de l'acceptation du passage du mode $HFilter\_mode1$ au mode $HFilter\_mode2$ ou $HFilter\_mode3$ et du mode $HFilter\_mode2$ au mode $HFilter\_mode3$ ont la valeur *true* alors que pour le reste des contraintes d'acceptation, le niveau de batterie est vérifié. Par exemple, pour la contrainte de l'acceptation du passage du mode $HFilter\_mode2$ au mode $HFilter\_mode1$, la condition $battery\_level >= 800000$ doit être valide. La contrainte du refus n'est pas modélisée ici puisqu'on a modélisé la contrainte d'acceptation. L'automate du contrôleur lié au filtre vertical est illustré dans la figure 6.6. Il suit les mêmes principes que celui lié au filtre horizontal en utilisant des seuils de consommation différents puisque les deux filtres n'ont pas la même consommation.

6.3.4 Le déploiement

Le déploiement dans cette étude de cas suit les mêmes principes présentés dans le chapitre précédent. Les *CodeFile*s sont attachés aux composants déployant les modules d'observation, de reconfiguration et de communication avec le bus puisqu'ils seront réutilisés à partir d'une bibliothèque d'IPs. Le reste du système de contrôle (les modules de décision, le coordinateur, les contrôleurs et le système de contrôle) sera généré automatiquement. Les ports liés aux signaux du bus sont déployés par des ports ayant les types *IPIC_slave_in_signal* et *IPIC_slave_out_signal*. Ces types représentent une interface slave de type *IPIC* (IP Interconnect). C'est l'interface utilisée par Xilinx pour tous les IPs indépendamment du type de bus utilisé. Pour pouvoir connecter l'IP au bus, il faut ajouter un adaptateur entre cette interface et celle du bus. Les ports de communication entre les contrôleurs et le coordinateur sont déployés par des ports ayant le type *coordination_interface* qui est le même type utilisé dans le chapitre précédent.

6.4 Transformations de modèles et génération du code

Après la chaîne de transformations présentée dans le chapitre précédent, le système de contrôle contenant 4 contrôleurs et un coordinateur est généré en VHDL. Parmi les fichiers VHDL générés pour ce système de contrôle, un fichier *userlibrary* qui représente un package VHDL contenant les déclarations de tous les types et composants utilisés dans le système (élémentaires et composés). Pour les modules d'observation, de reconfiguration et de communication avec le bus, des entités sont créés avec des architectures qui instancient les modules VHDL existants dans la bibliothèque d'IPs. Pour les modules de décision, des entités sont créées avec des architectures qui instancient un module implémentant l'automate de mode et un module assurant la communication avec le coordinateur tel que nous l'avons expliqué dans la section 5.2.3. Le coordinateur généré contient aussi un module implémentant l'automate de modes et un module de communication avec les contrôleurs. Cette séparation entre traitement et communication facilite l'adaptation de la communication entre contrôleurs et coordinateur à d'autres protocoles et topologies de communication.

La structure du système de contrôle généré est illustrée dans la figure 6.7 qui donne une vue de ce système dans l'outil ISE 12.4 utilisé pour la simulation de ce système, comme nous expliquerons dans la suite de ce chapitre.

Listage 6.1 – Déclaration du composant représentant le système de contrôle et des types utilisés

```
1
2 type IPIC_slave_in_signals is
3 record
4     Bus2IP_Data                    :   std_logic_vector(0 to 31);
5     Bus2IP_BE                      :   std_logic_vector(0 to 3);
6     Bus2IP_RdCE                    :   std_logic_vector(0 to 3);
7     Bus2IP_WrCE                    :   std_logic_vector(0 to 3);
8     Bus2IP_RdReq                   :   std_logic;
9     Bus2IP_WrReq                   :   std_logic;
10 end record;
11
12 type IPIC_slave_out_signals is
13 record
14     IP2Bus_AddrAck                 :  std_logic;
15     IP2Bus_Data                    :  std_logic_vector(0 to 31);
16     IP2Bus_RdAck                   :  std_logic;
17     IP2Bus_WrAck                   :  std_logic;
18     IP2Bus_Error                   :  std_logic;
19 end record;
20
21 Type array_4_Of_IPIC_slave_in_signals   is array(0 to 3)
22 of IPIC_slave_in_signals;
23
24 Type array_4_Of_IPIC_slave_out_signals   is array(0 to 3)
25  of IPIC_slave_out_signals;
26
27  COMPONENT ControlModel
28     PORT(
29          clk : IN  std_logic;
30          rst : IN  std_logic;
31          bus_out : OUT  array_4_Of_IPIC_slave_out_signals ;
32          prr_out : OUT array_4_Of_IPIC_slave_in_signals ;
33          bus_in : IN  array_4_Of_IPIC_slave_in_signals ;
34          prr_in : IN  array_4_Of_IPIC_slave_out_signals ;
35          battery_level : IN  std_logic_vector(0 to 31)
36         );
37     END COMPONENT;
```

Le composant VHDL implémentant le système de contrôle est illustré dans le listage 6.1. Ce listage montre que les ports *bus_out*, *prr_out*, *bus_in* et *prr_in* de ce composant sont des tableaux de taille 4 correspondant au nombre de régions contrôlées. Les types *IPIC_slave_in_signals* et *IPIC_slave_out_signals* utilisés sont des enregistrements (record) VHDL comme le montre le listage. La combinaison des signaux de ces deux types constitue une interface slave de type IPIC.

Comme le montre la figure 6.7, 8 fichiers sont générés pour chaque contrôleur :

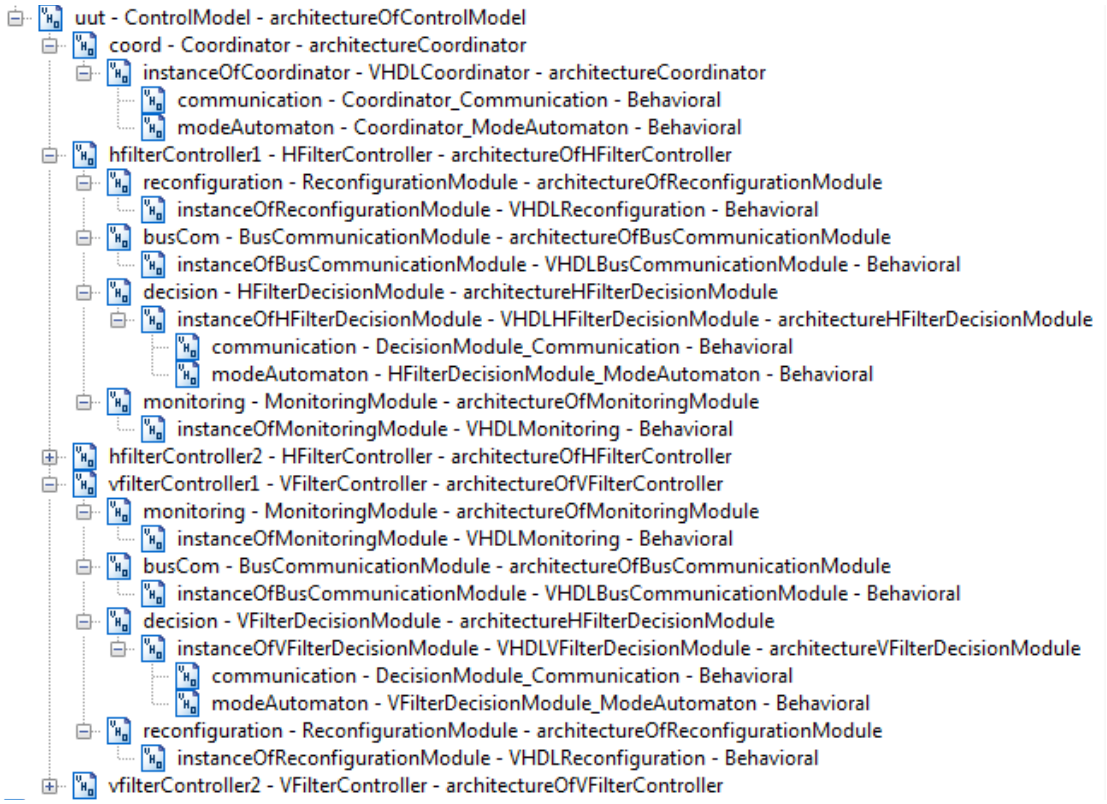


FIG. 6.7 – Structure du système de contrôle généré

- un fichier pour l'entité et l'architecture du contrôleur (composant *HFilterController* ou *VFilterController*) qui instancie les composants d'observation, de décision, de reconfiguration et de communication avec le bus
- un fichier pour le module d'observation qui contient une entité et une architecture qui instancie un IP *VHDLMonitoring* qui existe déjà dans la bibliothèque d'IPs
- un fichier pour le module de reconfiguration qui contient une entité et une architecture qui instancie un IP *VHDLReconfiguration* qui existe déjà dans la bibliothèque d'IPs
- un fichier pour le module de communication avec le bus qui contient une entité et une architecture qui instancie un IP *VHDLBusCommunicationModule* qui existe déjà dans la bibliothèque d'IPs
- un fichier pour l'entité et l'architecture du module de décision (composant *HFilterDecisionModule* ou *VFilterDecisionModule*) qui instancie un composant déployant ce module (composant *VHDLHFilterDecisionModule* ou *VHDLVFilterDecisionModule*)
- un fichier pour le composant déployant le module de décision qui instancie un composant pour la communication avec le coordinateur et un composant pour implémenter l'automate de modes
- un fichier pour le module de communication entre le module de décision et le coordinateur appelé *DecisionModule_Communication* (le même composant pour les

deux types de contrôleurs)
- un fichier pour le module implémentant l'automate de modes (composant *HFilterDecisionModule_ModeAutomaton* ou *VFilterDecisionModule_ModeAutomaton*)

Pour le coordinateur, 4 fichiers sont générés :
- un fichier pour l'entité et l'architecture du coordinateur (composant *Coordinator*) qui instancie le composant qui le déploie (composant *VHDLCoordinator*)
- un fichier pour le composant *VHDLCoordinator* qui déploie le coordinateur
- un fichier pour le module de communication avec les contrôleurs (composant *Coordinator_Communication*)
- un fichier pour le module implémentant l'automate de modes du coordinateur (composant *Coordinator_ModeAutomaton*)

6.5 Réutilisabilité et scalabilité du modèle de contrôle semi-distribué

6.5.1 Evaluation du modèle semi-distribué

Pour évaluer l'efficacité du modèle semi-distribué en termes de réutilisation et de scalabilité de conception, nous avons modélisé des systèmes de contrôle ayant différents nombres de contrôleurs (2, 4, 8 et 16) dont la moitié est liée au filtre horizontal. Pour cela, il fallait juste réutiliser le modèle de la figure 6.3 en entrant quelques modifications. Ces modifications consistent en : 1) la modification du nombre d'instances de contrôleurs et 2) la modification des valeurs des propriétés du stéréotype «Coordinator» du coordinateur pour mettre à jour les instances de contrôleurs coordonnées et la table *GC*. Grâce à la flexibilité offerte par le modèle semi-distribué, l'ajout de contrôleurs au système de contrôle de l'étude de cas n'implique que des modifications minimes. La division du contrôle entre les contrôleurs coordonnés facilite la réutilisation des contrôleurs par simple instanciation pour passer à un système de contrôle contenant plus de contrôleurs. L'algorithme de coordination implémenté par le coordinateur est réutilisé aussi puisqu'il n'a pas une vision de ce qui se passe dans les régions contrôlées. L'échange entre le coordinateur et les contrôleurs est juste un échange de modes qui sont soit un sujet de requête, de proposition, d'acceptation, de refus ou d'autorisation. Il suffit donc de changer la table *GC* pour l'adapter aux nouvelles contraintes du système. Cela montre bien que le modèle de contrôle semi-distribué est facilement scalable grâce à sa grande flexibilité et réutilisabilité. Dans le reste de cette section, nous étudions les aspects de réutilisation et de scalabilité du modèle centralisé en le comparant au modèle semi-distribué.

6.5.2 Comparaison avec le modèle centralisé

La génération automatique de contrôleurs centralisés n'a pas été étudiée dans notre méthodologie dirigée par les modèles. Cependant, on peut imaginer à quoi ressemblerait la modélisation du contrôle centralisé. La prise de décision peut être modélisée de deux façons différentes : 1) un seul automate ou 2) des automates parallèles.

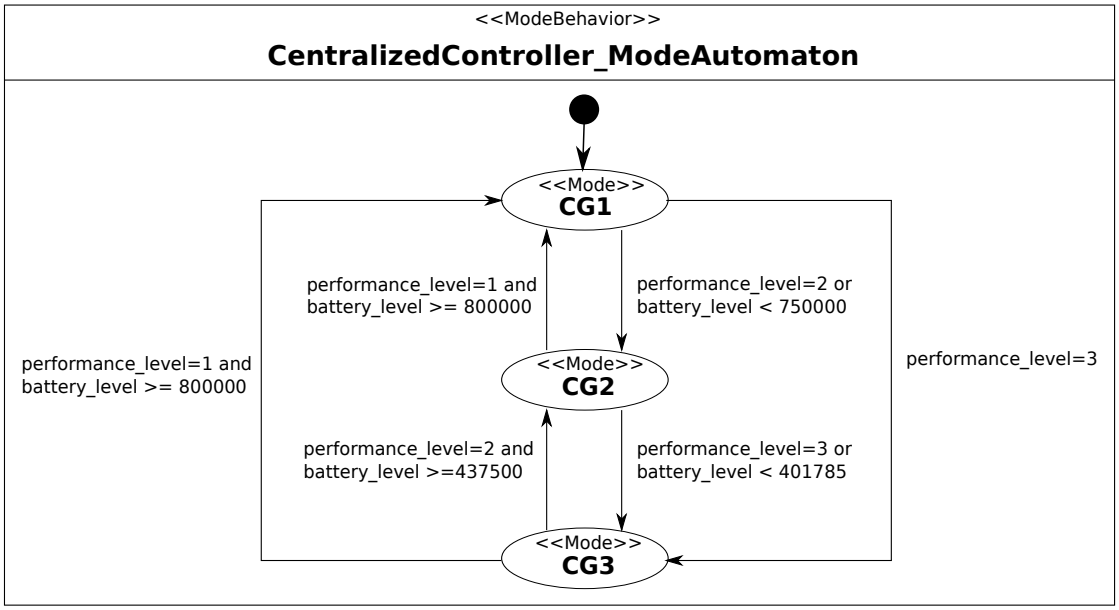


FIG. 6.8 – Modélisation du contrôleur centralisé par un seul automate

6.5.2.1 Le contrôle centralisé modélisé par un seul automate

Dans la première solution, chaque mode de l'automate représente une configuration globale du système. Les transitions entre les modes dépendent des données d'observation et des contraintes globales qui sont les mêmes que celles gérées par le coordinateur dans le modèle semi-distribué. Les transitions dépendent donc d'une vision globale du système, ce qui complique la modélisation de cet automate pour des systèmes de grande taille.

La figure 6.8 illustre la modélisation de l'automate centralisé de l'étude de cas. Chaque mode représente une combinaison de modes correspondant à une ligne de la table *GC* du coordinateur dans le modèle semi-distribué. Par exemple, le mode $GC1$ correspond à des régions implémentant toutes le mode numéro 1 de l'un des filtres. Ici, la modélisation est assez simple puisque les données d'observation liées au contrôle des différentes régions sont les mêmes (niveau de performance et niveau de batterie), ce qui simplifie énormément les conditions des transitions. Si l'étude de cas traitait des données d'observation différentes pour chaque région, les expressions des conditions seraient beaucoup plus longues et plus compliquées. De plus, le nombre de combinaisons de modes est réduit dans cette étude de cas, ce qui simplifie aussi la modélisation de l'automate.

L'implémentation du contrôle centralisé par un seul automate peut donc ne pas convenir à des systèmes de grande taille impliquant un grand nombre de possibilités de configuration en adaptation aux changements d'un grand nombre de données d'observation. La vision globale de cet automate complique la modélisation de ses conditions de transitions. Cette vision globale implique aussi une dépendance au système qui représente un obstacle devant la réutilisation et la scalabilité de la conception du contrôle

centralisé.

6.5.2.2 Le contrôle centralisé modélisé par un ensemble d'automates

Le contrôleur centralisé peut être modélisé par un ensemble d'automates où chaque automate contrôle le comportement d'une région reconfigurable du système. Ces automates doivent être synchronisés afin de respecter les contraintes globales du système et éviter les combinaisons non permises de modes. Le contrôleur centralisé peut être donc modélisé par 4 automates (2 instances de l'automate contrôlant une région implémentant le filtre horizontal et 2 instances de l'automate contrôlant une région implémentant le filtre vertical). La figure 6.9 illustre les automates implémentant la solution centralisée. Elle ne contient qu'une seule instance pour chaque automate. Les conditions de transitions sont les mêmes conditions de requêtes dans le modèle semi-distribué. Ces conditions intègrent, en plus, des variables pour la synchronisation. Tous les automates commencent en modes $H/VFilter\_mode1$. Si le niveau de performance passe à 2 ou le niveau de la batterie est inférieur à 750000, tous les automates passent en même temps aux modes $H/VFilter\_mode2$. Ici, on n'a pas de problème de synchronisation parce que la nouvelle configuration (tous les modes des automates représentent le mode numéro 2) est permise et la transition a les mêmes conditions dans tous les contrôleurs. Or, une fois dans la nouvelle configuration, si le niveau de la batterie descend au-dessous de 401785, il faut éviter le passage des automates du filtre vertical en modes $VFilter\_mode3$ alors que les autres automates restent en modes $HFilter\_mode2$. Pour cela, une variable booléenne a est ajoutée à la transition entre le mode $HFilter\_mode2$ et $HFilter\_mode3$. Cette variable a la valeur *true* lorsque le niveau de la batterie est inférieur à 401785, ce qui permet de passer à la configuration globale 3 dès que le niveau de batterie descend au-dessous de 401785. Cette contrainte peut être modélisée par une contrainte «NfpConstraint» comme le montre la figure 6.9. De la même façon, pour éviter que les automates du filtre horizontal passent du mode $HFilter\_mode3$ au mode $HFilter\_mode2$ si le niveau de performance a la valeur 2 et le niveau de batterie est supérieur à 408333 alors que les autres automates restent en modes $VFilter\_mode3$, la variable b est utilisée. Cette variable prend la valeur *true* si le niveau de batterie est supérieur ou égal à 475000.

Le choix des variables de synchronisation et l'affectation de leurs valeurs selon les cas peut augmenter la complexité du contrôle s'il s'agit d'un système de grande taille. En effet, si on ajoute des régions aux systèmes, le nombre des variables de synchronisation augmentent et la détermination de leurs valeurs devient plus compliquée. Une telle évolution du système pourrait impliquer aussi l'ajout de variables de synchronisation aux automates déjà existants, ce qui représente un obstacle devant la réutilisation et la scalabilité de la conception du contrôleur.

Pour éviter la gestion manuelle des valeurs des variables de synchronisation, certains travaux proposent la synthèse du contrôleur [45]. Cette approche permet de gérer automatiquement les valeurs de ces variables (appelées variables contrôlables) afin d'assurer une propriété du système (ici il s'agit d'assurer qu'aucune combinaison non permise de modes ne soit active), ce qui aide à diminuer la complexité de la conception. Avec cette approche, il est possible de générer le contrôleur en langage C d'une façon automatique en utilisant le compilateur synchrone HEPTAGON [31]. Cependant,

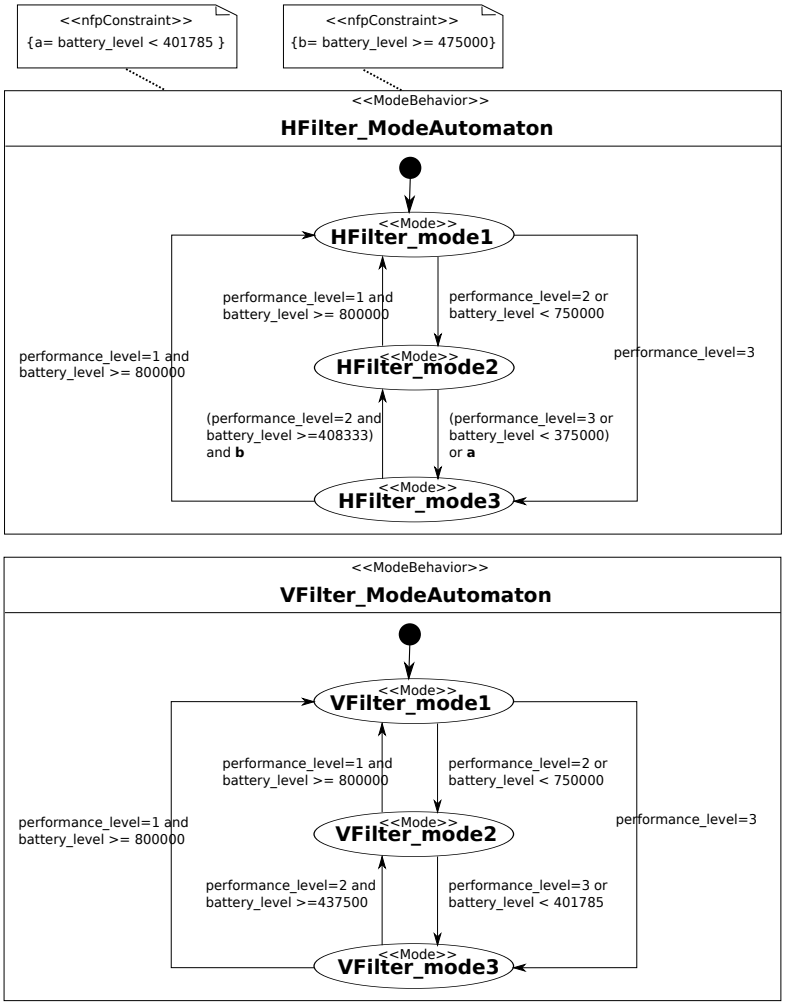


FIG. 6.9 – Modélisation du contrôleur centralisé par des automates parallèles

la génération d'un contrôleur matériel n'est pas supportée. Avec cette approche, le concepteur doit toujours gérer manuellement l'ajout des variables contrôlables aux conditions des transitions dans les automates, ce qui pourrait être compliqué pour des systèmes de grande taille.

En comparaison à la solution centralisée, le modèle de contrôle semi-distribué proposé permet de garder une vue locale pour chaque contrôleur et simplifie la "synchronisation" entre les contrôleurs à travers un mécanisme de coordination réutilisable et paramétrable, ce qui facilite la scalabilité.

6.5. RÉUTILISABILITÉ ET SCALABILITÉ DU MODÈLE DE CONTRÔLE SEMI-DISTRIBUÉ

Configuration globale	1	2	3
Pas de décrémentation par cycle de la batterie (mode déchargement)	46	38	30
Pas d'incrémentation par cycle de la batterie (mode chargement)	4	12	20
Durée de traitement d'une trame (ns)	100	130	200

TAB. 6.3 – Valeurs numériques prises pour la simulation

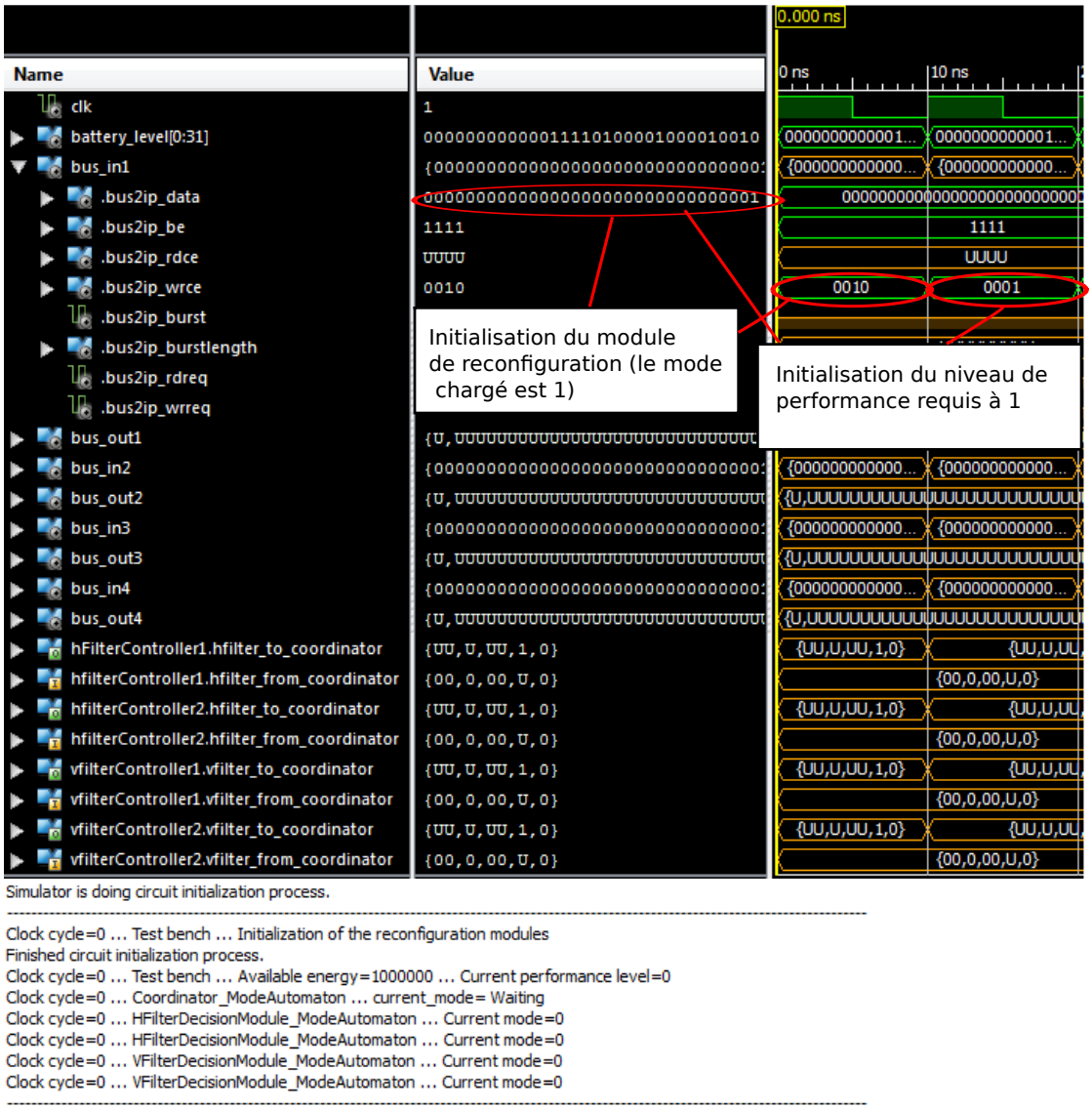


FIG. 6.10 – Début de la simulation

6.6 Simulation des systèmes de contrôle semi-distribué générés

Les systèmes de contrôle générés pour différents nombres de contrôleurs peuvent être intégrés dans des architectures utilisant différents degrés de parallélisme de l'application *Downscaler* en variant le nombre de régions reconfigurables implémentant les filtres. Dans cette section, la simulation du système de contrôle composé de 4 contrôleurs et d'un coordinateur modélisé dans la figure 6.3 est étudiée. Cette simulation utilise l'outil ISE 12.4 de Xilinx en ciblant le FPGA Virtex6-xc6vlx240t sur lequel le système reconfigurable sera implémenté plus tard.

Pour la simulation du système de contrôle généré, nous avons créé un fichier de testbench qui l'instancie et qui simule ses entrées par des processus VHDL. Les entrées du système de contrôle sont le niveau de batterie envoyé par le capteur de batterie et les commandes du processeur. Ces commandes permettent d'envoyer aux contrôleurs le niveau de performance requis par l'utilisateur, de lire les modes cibles des modules de reconfiguration et de notifier ces derniers après le chargement des bitstreams. Ici, on suppose que le processeur lit les registres de reconfiguration après chaque traitement d'une image de la séquence vidéo en entrée. Pour cela, on associe une durée de temps approximative à ce traitement selon la configuration courante. Le processus simulant le processeur doit donc attendre (insertion des instructions wait dans le testbench) pendant une durée donnée avant d'envoyer les requêtes de lecture aux contrôleurs pour lire les registres de reconfiguration.

Pour le processus simulant la batterie, on considère deux modes : le chargement et le déchargement. Les pas d'incrémentation et de décrémentation du niveau de batterie selon son mode actif sont illustrés dans le tableau 6.3. Pour le mode déchargement, on suppose que la consommation de la partie statique vaut 20 unités par cycle, le reste de la consommation est donnée par la somme des consommations des régions reconfigurables.

Le tableau 6.3 indique aussi les valeurs utilisées pour les durées de traitement d'une image selon la configuration active du système. Ces valeurs indiquent à quels moments les requêtes de lecture sont envoyées aux contrôleurs pour lire les registres de reconfiguration. Un compteur de cycles d'horloge est utilisé dans le simulateur. Les instants de changement du niveau de performance se basent donc sur la valeur de ce compteur. Les instructions du changement du niveau de performance requis par l'utilisateur sont insérées, dans le processus simulant le processeur, pour des valeurs du compteur de cycles d'horloge permettant de tester les acceptations et les refus des contrôleurs. Dans la simulation, les requêtes envoyées aux régions reconfigurables par le port *bus_in* ne sont pas considérées puisqu'elles n'interviennent pas ici dans la prise de décision. De même les valeurs des signaux des ports *prr_in* et *prr_out* ne sont pas considérées.

Pour bien suivre l'évolution de la communication entre les contrôleurs et le coordinateur, des instructions d'affichage dans la console sont ajoutées dans les codes de ceux-ci. Nous avons pris un cycle d'horloge de 10ns pour la simulation, ce qui correspond à la fréquence (100MHz) qui sera utilisée pour le système à implémenter sur FPGA. La figure 6.10 illustre les valeurs des signaux du testbench au début de la simulation. Comme le montre la figure, les signaux des régions reconfigurables ont été enlevés de

la waveform pour ne garder que ceux qui changent de valeurs durant la simulation. Les entrées et sorties du bus sont représentées par des signaux individuels au lieu de tableaux de taille 4 pour faciliter le suivi de l'évolution des signaux durant la simulation. L'affichage de certains signaux internes au système de contrôle, liés à la communication entre les contrôleurs et le coordinateur, a été également ajouté.

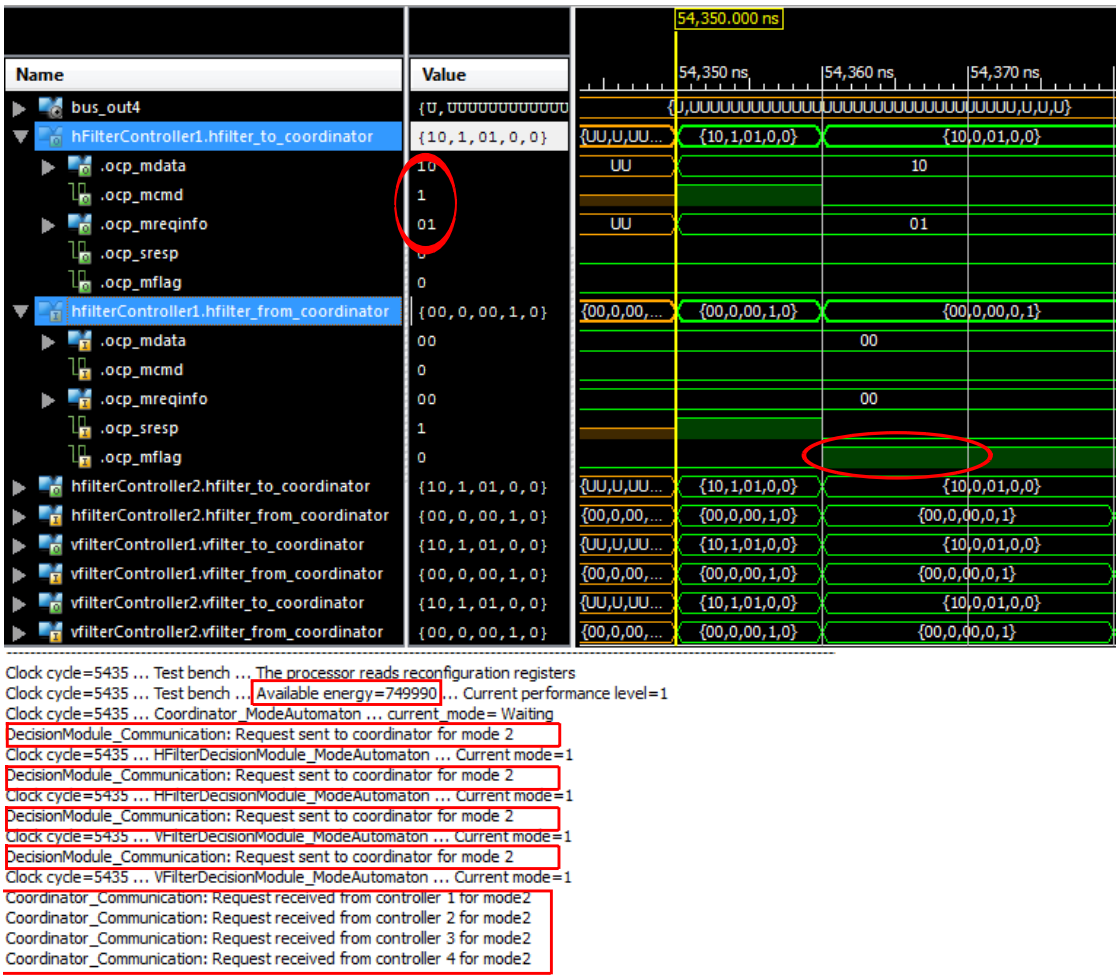


FIG. 6.11 – Requêtes de reconfiguration de *H/VFilter_mode*1 à *H/VFilter_mode*2

La simulation commence par une initialisation des modules de reconfiguration, dans laquelle le processeur indique les modes chargés dans les régions reconfigurables. Comme le montre les lignes 6 et 7 du listage 6.1, les champs *Bus2IP_Rd CE* et *Bus2IP_WrCE* utilisés par le processeur pour lire ou écrire dans les registres du contrôleur ou de la région contrôlée sont de taille 4 bits. Ici, le contrôleur partage la même interface bus que sa région contrôlée, les deux premiers bits de ces champs permettent de lire et d'écrire les registres de la région contrôlée. Les deux autres sont liés aux registres du contrôleur : le troisième bit est consacré à la communication avec le module de reconfiguration et le quatrième avec le module d'observation. Comme le montre la figure 6.10, le troisième bit des champs *Bus2IP_WrCE* des ports *bus_in* est utilisé pour

l'initialisation des modules de reconfiguration. Ici, l'initialisation de tous les modules est faite simultanément, ce qui n'est pas le cas en réalité puisqu'elle sera exécutée par le processeur, dans l'implémentation finale du système. Le niveau de performance est initialisé à 1. A partir de t=20ns la valeur affichée du mode courant des automates des contrôleurs est 1, ce qui correspond au mode numéro 1 ($H/VFilter\_mode1$).

A t=54350ns, le niveau de batterie est au-dessous du seuil 750000. Tous les contrôleurs envoient une requête au coordinateur pour passer au mode 2 ($H/VFilter\_mode2$) comme le montre la figure6.11. Le signal *ocp_mcmd* des ports *to_coordinator* est donc activé. Le champ *ocp_mreqinfo* prend la valeur "01" pour indiquer qu'il s'agit d'une requête et le champ *ocp_mdata* prend la valeur "10" qui est le code du deuxième mode de la région contrôlée. Dans le cycle d'horloge suivant, les contrôleurs sont notifiés qu'il y a un processus de coordination en cours à travers le signal *ocp_mflag* des ports *from_coordinator* comme le montre la figure6.11.

La figure 6.12 illustre l'évolution du processus de coordination. Seule la requête du contrôleur 1 est considérée, puisqu'on donne la priorité au contrôleur ayant le plus petit indice si on reçoit plus d'une requête simultanément. Le coordinateur passe du mode *Waiting* au mode *Determine_Reconfiguration_Possibilities*. Ici, il s'agit d'une seule possibilité qui est la configuration globale numéro 2 du tableau 6.2 comme le montre la figure 6.12. Le coordinateur passe ensuite au mode *Send_Suggestions*, la variable *involved_controllers* indiquant les contrôleurs impliqués dans un tour de coordination est de type tableau de 4 bits. La valeur affichée pour cette variable est une traduction décimale comme le montre la figure 6.12. Ici, tous les contrôleurs sont impliqués : la requête du premier est retenue et le coordinateur envoient aux trois autres les propositions de reconfiguration. Le signal *ocp_mcmd* des ports *from_coordinator* est activé. Le champ *ocp_mreqinfo* prend la valeur "01" pour indiquer qu'il s'agit d'une requête (une proposition de reconfiguration) et le champ *ocp_mdata* prend la valeur "10" qui est le code du deuxième mode de chacune des régions contrôlées par les trois contrôleurs en question.

Après la réception des propositions, les contrôleurs envoient leurs réponses au coordinateur comme le montre la figure6.13. Le signal *ocp_mcmd* du port *to_coordinator* est donc activé. Le champ *ocp_mreqinfo* prend la valeur "10" pour indiquer qu'il s'agit d'une acceptation. Toutes les réponses sont donc des acceptations puisque la stratégie de décision des contrôleurs dans cette étude de cas est d'accepter toute transition vers un mode qui consomme moins d'énergie que le mode courant.

Ayant reçu des réponses positives à toutes les propositions, le coordinateur envoie une autorisation de reconfiguration à tous les contrôleurs impliqués (tous les quatre) comme le montre la figure 6.14. Le signal *ocp_mcmd* des ports *from_coordinator* est activé. Le champ *ocp_mreqinfo* prend la valeur "10" pour indiquer qu'il s'agit d'une autorisation. Le coordinateur passe ensuite au mode *Receive_Notifications* et attend les notifications des modules de décision après le chargement des bitstreams. Ici, le chargement des bitstreams n'est pas considéré dans la simulation. Le processeur lit les registres des modules de reconfiguration et leur envoie la notification, tout de suite après, comme le montre la figure 6.15. Les modules de décision notifient donc le coordinateur en activant le signal *ocp_mflag* des ports *to_coordinator*. Dans le cycle d'horloge suivant, les contrôleurs sont notifiés de la fin du processus de coordination en désactivant le signal

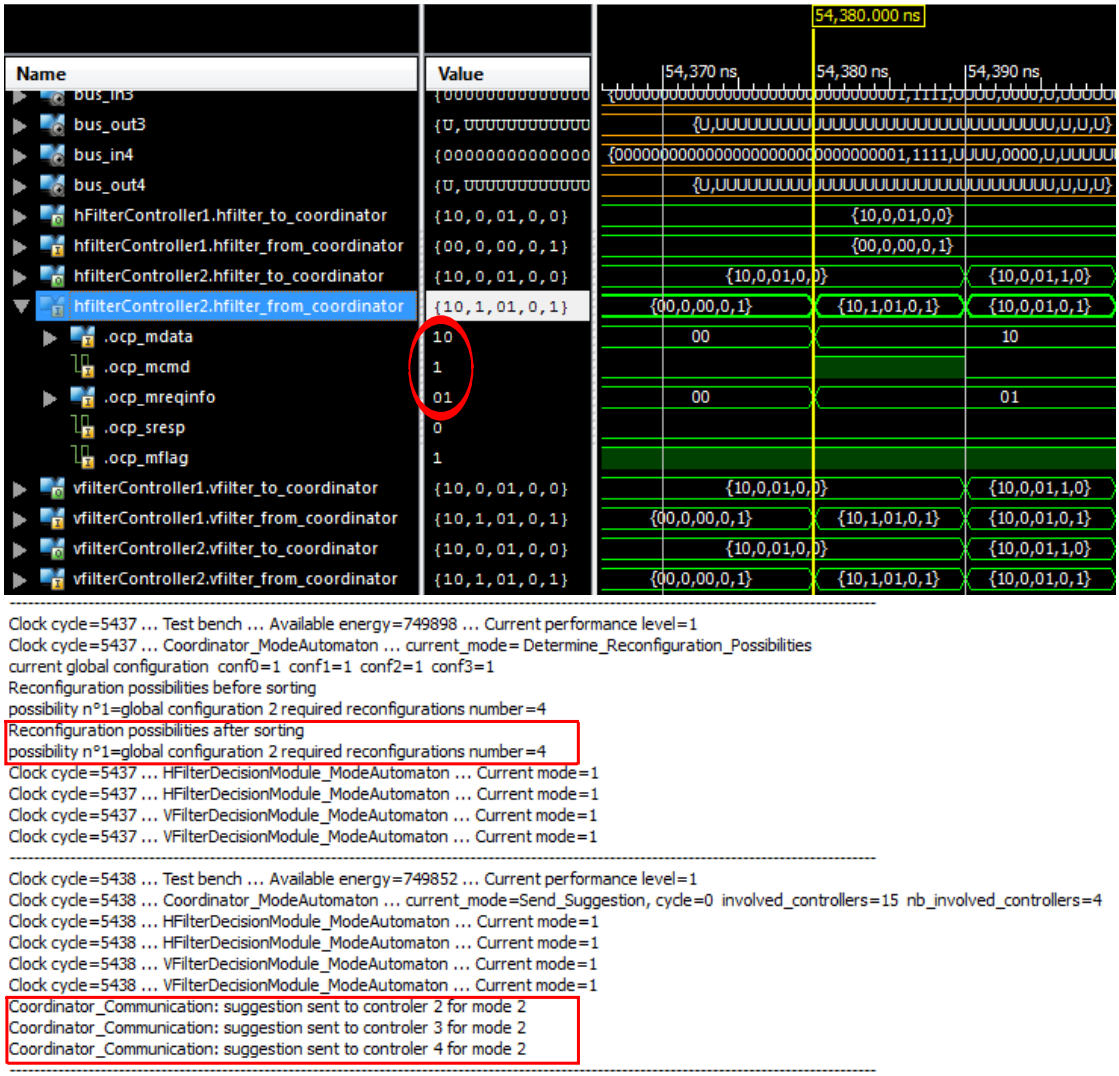


FIG. 6.12 – Envoi des propositions de reconfiguration par le coordinateur

ocp_mflag des ports *from_coordinator*.

Sans considérer le temps d'attente de la notification après le chargement des bitstreams, la durée du processus de coordination suit la formule

$$coordination\_process\_length = 4 + nb\_coordination\_rounds * 4$$

avec *nb_coordination_rounds* est le nombre de tours du processus de coordination avant d'arriver à une décision finale. Dans l'implémentation étudiée, il s'agit du nombre de tours avant d'arriver à des réponses positives de tous les contrôleurs concernés ou finir le processus si aucune des possibilités de reconfiguration n'a été acceptée par tous les contrôleurs. Les 3 premiers cycles d'horloge sont nécessaires pour : 1) envoyer une requête au coordinateur, 2) prise en compte de la requête par le coordinateur et 3) détermination des possibilités de reconfiguration. Pour chaque tour de coordination,

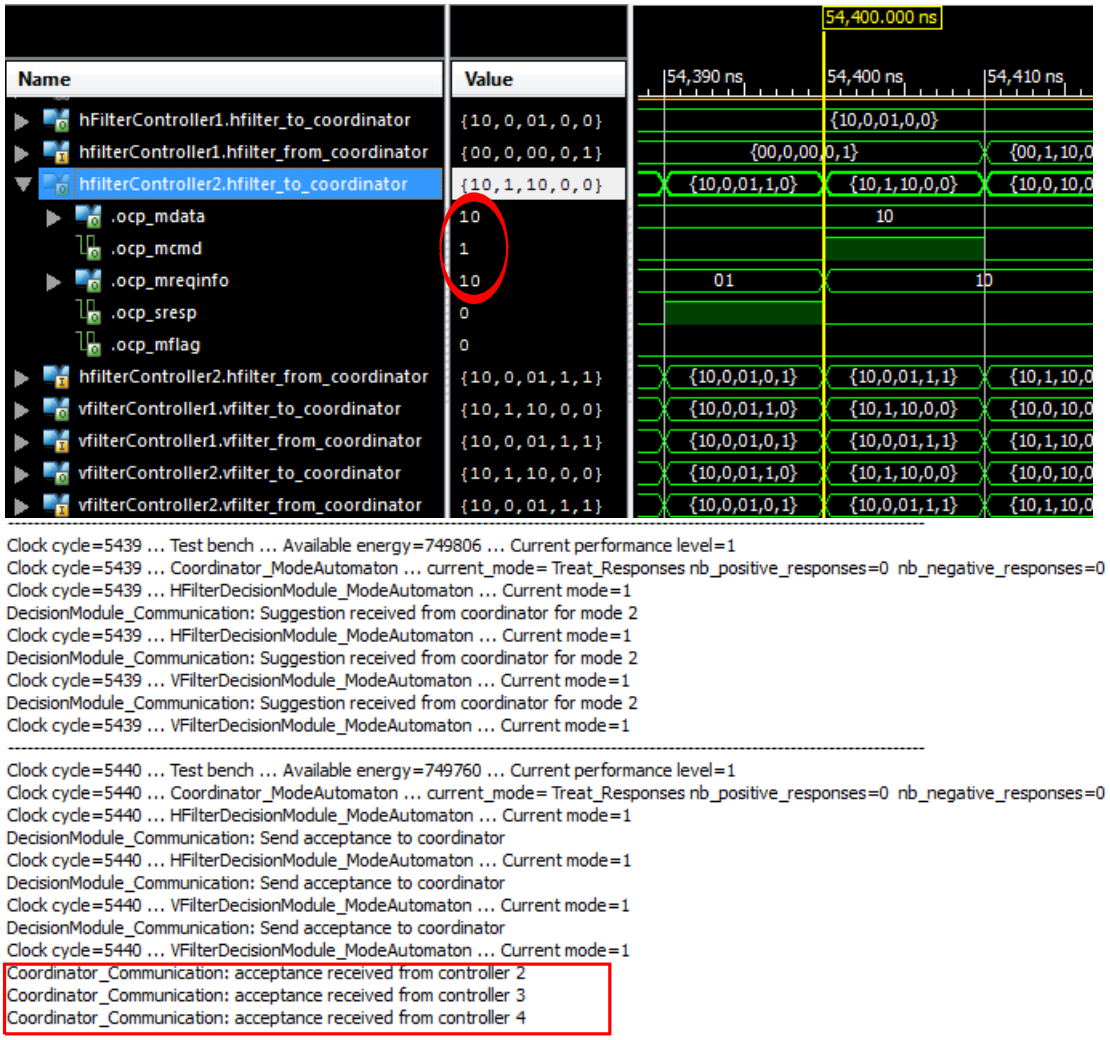


FIG. 6.13 – Acceptation des propositions par les contrôleurs

il faut 4 cycles d'horloge pour : 1) envoi des propositions aux contrôleurs, 2) prise en compte des propositions par les contrôleurs 3) envoi des réponses aux propositions par les contrôleurs, et 4) traitement des réponses et éventuellement envoi de la décision. Au dernier cycle d'horloge d'un processus de coordination, les contrôleurs reçoivent la décision du coordinateur.

Dans l'exemple précédent le *nb_coordination_rounds* est égal à 1 puisqu'il n'y avait qu'une seule possibilité de reconfiguration. Le processus a duré donc 8 cycles d'horloges, ce qui peut être considéré comme une petite durée pour un processus qui fait communiquer plusieurs modules du système. Cette rapidité dans le processus de coordination est obtenue grâce à l'implémentation matérielle du système de contrôle et la communication peer-to-peer entre les contrôleurs et le coordinateur, ce qui a facilité le traitement des réponses des contrôleurs en un seul cycle.

A t=145960ns, le seuil 401785 de la batterie est dépassé, ce qui fait que les contrôleurs

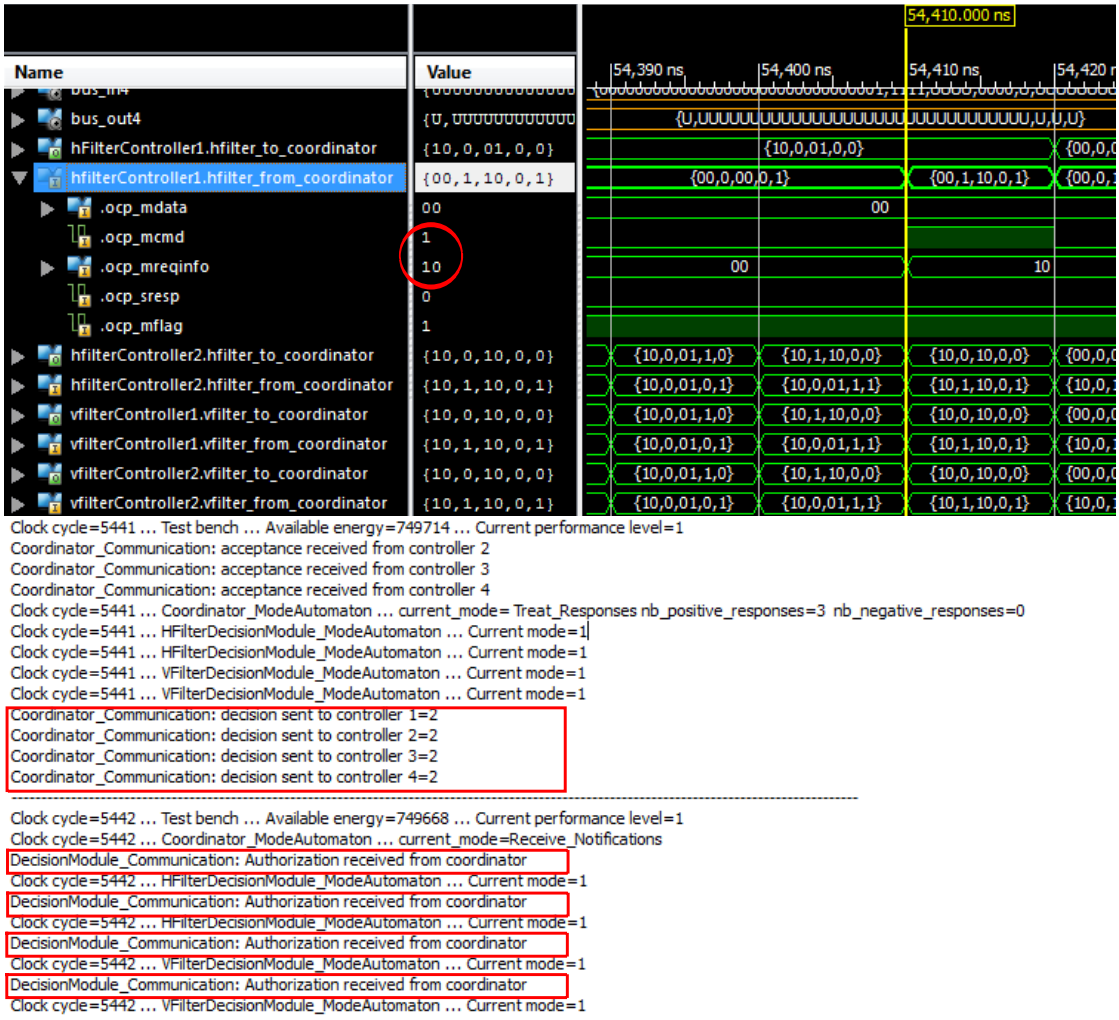
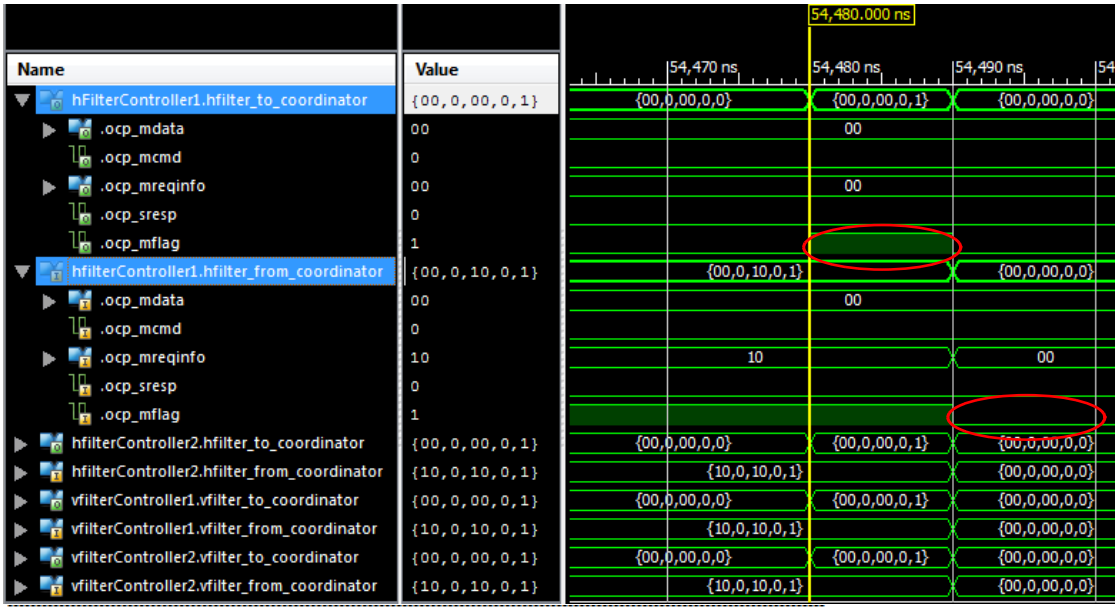


FIG. 6.14 – Envoi de la décision du coordinateur aux contrôleurs

3 et 4 envoient une requête de reconfiguration au coordinateur pour passer au mode 3 (*VFilter_mode*3) comme le montre la figure 6.16. La requête du contrôleur 3 est retenue puisqu'il a le plus petit indice. Une seule configuration globale satisfait cette requête qui est la configuration numéro 3 du tableau 6.2. Le coordinateur envoient des propositions aux contrôleurs 1,2 et 4 pour passer au mode 3 et ils acceptent tous. Le coordinateur est ensuite notifié après le chargement des bitstreams à t=14670ns, comme le montre la figure 6.17, et le système passe à la configuration globale 3.

A t=279880ns, la batterie est vide, le processus de la batterie passe donc au mode chargement comme le montre la figure6.18. A t=484050ns, le seuil 408333 de la batterie est dépassé. Le niveau de performance requis est 2. Les contrôleurs 1 et 2 envoient donc une requête au coordinateur pour passer au mode 2 (*HFilter_mode*2) comme le montre la figure 6.19. Le coordinateur envoient des propositions aux contrôleurs 2,3 et 4 pour passer au mode 2. Le contrôleur 2 accepte la proposition, alors que les autres la refusent puisque le niveau de la batterie n'a pas encore atteint le seuil (437500) permettant aux




```
Clock cycle=5446 ... Test bench ... The processor reads reconfiguration registers
Clock cycle=5446 ... Test bench ... Available energy=749484 ... Current performance level=1
Clock cycle=5446 ... Coordinator_ModeAutomaton ... current_mode=Receive_Notifications
Clock cycle=5446 ... HFilterDecisionModule_ModeAutomaton ... Current mode=1
Clock cycle=5446 ... HFilterDecisionModule_ModeAutomaton ... Current mode=1
Clock cycle=5446 ... VFilterDecisionModule_ModeAutomaton ... Current mode=1
Clock cycle=5446 ... VFilterDecisionModule_ModeAutomaton ... Current mode=1

Clock cycle=5447 ... Test bench ... Available energy=749438 ... Current performance level=1
Clock cycle=5447 ... Coordinator_ModeAutomaton ... current_mode=Receive_Notifications
Clock cycle=5447 ... HFilterDecisionModule_ModeAutomaton ... Current mode=1
Clock cycle=5447 ... HFilterDecisionModule_ModeAutomaton ... Current mode=1
Clock cycle=5447 ... VFilterDecisionModule_ModeAutomaton ... Current mode=1
Clock cycle=5447 ... VFilterDecisionModule_ModeAutomaton ... Current mode=1

Clock cycle=5448 ... Test bench ... Reconfiguration to global configuration 2
Clock cycle=5448 ... Test bench ... Available energy=749400 ... Current performance level=1
Clock cycle=5448 ... Coordinator_ModeAutomaton ... current_mode=Receive_Notifications
Clock cycle=5448 ... HFilterDecisionModule_ModeAutomaton ... Current mode=1
Clock cycle=5448 ... HFilterDecisionModule_ModeAutomaton ... Current mode=1
Clock cycle=5448 ... VFilterDecisionModule_ModeAutomaton ... Current mode=1
Clock cycle=5448 ... VFilterDecisionModule_ModeAutomaton ... Current mode=1

Clock cycle=5449 ... Test bench ... Available energy=749362 ... Current performance level=1
Clock cycle=5449 ... Coordinator_ModeAutomaton ... current_mode=Receive_Notifications
Clock cycle=5449 ... HFilterDecisionModule_ModeAutomaton ... Current mode=2
Clock cycle=5449 ... HFilterDecisionModule_ModeAutomaton ... Current mode=2
Clock cycle=5449 ... VFilterDecisionModule_ModeAutomaton ... Current mode=2
Clock cycle=5449 ... VFilterDecisionModule_ModeAutomaton ... Current mode=2

Clock cycle=5450 ... Test bench ... Available energy=749324 ... Current performance level=1
Clock cycle=5450 ... Coordinator_ModeAutomaton ... current_mode= Waiting
Clock cycle=5450 ... HFilterDecisionModule_ModeAutomaton ... Current mode=2
Clock cycle=5450 ... HFilterDecisionModule_ModeAutomaton ... Current mode=2
Clock cycle=5450 ... VFilterDecisionModule_ModeAutomaton ... Current mode=2
Clock cycle=5450 ... VFilterDecisionModule_ModeAutomaton ... Current mode=2
```

FIG. 6.15 – Mise à jour de la configuration courante

régions implémentant le filtre vertical de passer au mode 2. Le processus de coordination se termine donc par un refus envoyé au contrôleur 1 comme le montre la figure 6.20. Le contrôleur 1 n'envoie pas de nouveau la requête au coordinateur bien que les données d'observation lui permettent de le faire puisque, comme précisé dans le chapitre 4,

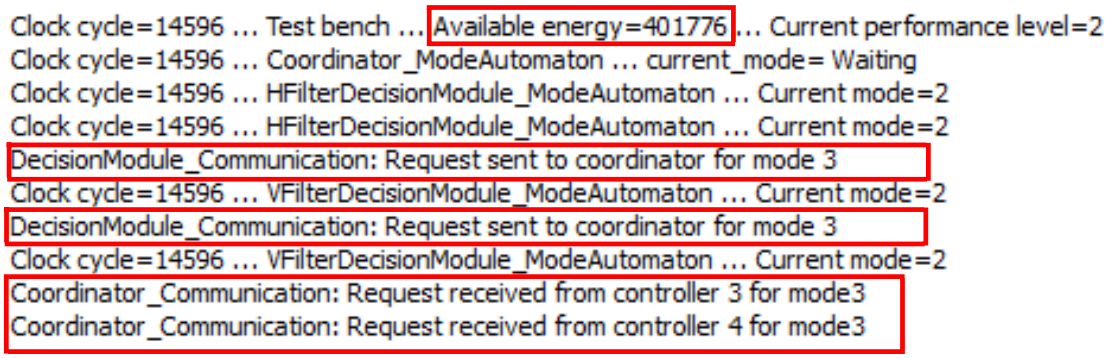


FIG. 6.16 – Envoi des requêtes de reconfiguration par les contrôleurs liés au filtre vertical pour passer au mode $VFilter\_mode3$

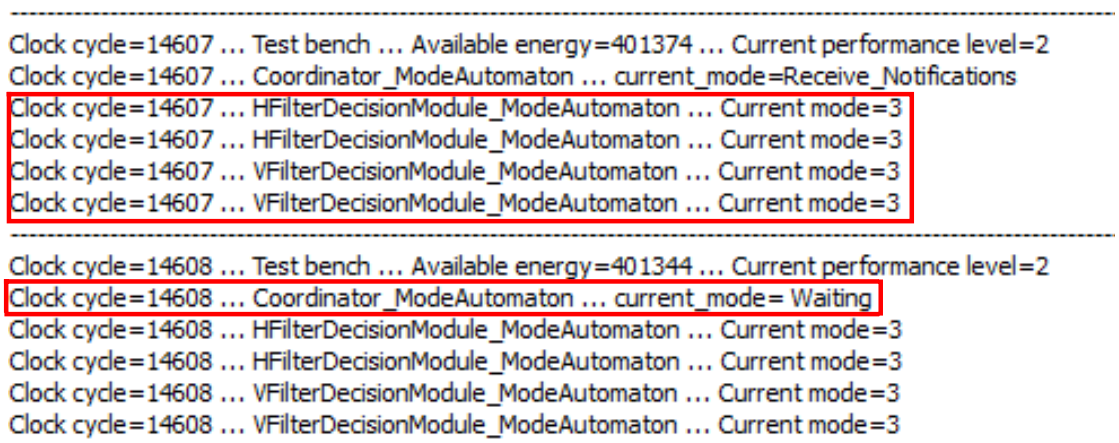


FIG. 6.17 – Passage à la configuration globale 3

l'envoi de requête n'est pas permis après son refus jusqu'à l'écoulement d'une durée de temps donnée. Donc, pour envoyer une requête pour passer au mode $mode_i$, il faut que la condition $counter\_value\_mode_i = resend\_request\_after_i$ soit valide, pour l'envoi de la requête après son refus ne soit permis. Dans cette étude de cas, nous donnons des valeurs nulles à tous $resend\_request\_after_i$, ce qui n'autorise pas l'envoi d'une requête refusée. Ces valeurs conviennent pour l'étude de cas puisque la seule situation où une requête refusée ne soit acceptée dans un processus de coordination ultérieur est lorsque le niveau de batterie atteint un certain seuil, ce qui déclenche des requêtes envoyées par les contrôleurs qui avaient refusé la reconfiguration. Dans ce cas, les contrôleurs dont les requêtes ont été refusé recevront des propositions de reconfigurations concernant les mêmes modes qu'il accepteront. Après la fin du processus de coordination, le contrôleur 2 envoie une requête pour passer au mode 2. Les propositions envoyées par le coordinateur sont acceptées par le contrôleur 1 et refusées par les contrôleurs 3 et 4. Le processus de coordination se termine par l'envoi de refus au contrôleur 2.

A t=498630ns, le seuil 437500 de la batterie est dépassé. Les contrôleurs 3 et 4 envoient des requêtes au coordinateur pour passer au mode 2 ($VFilter\_mode2$) comme le montre

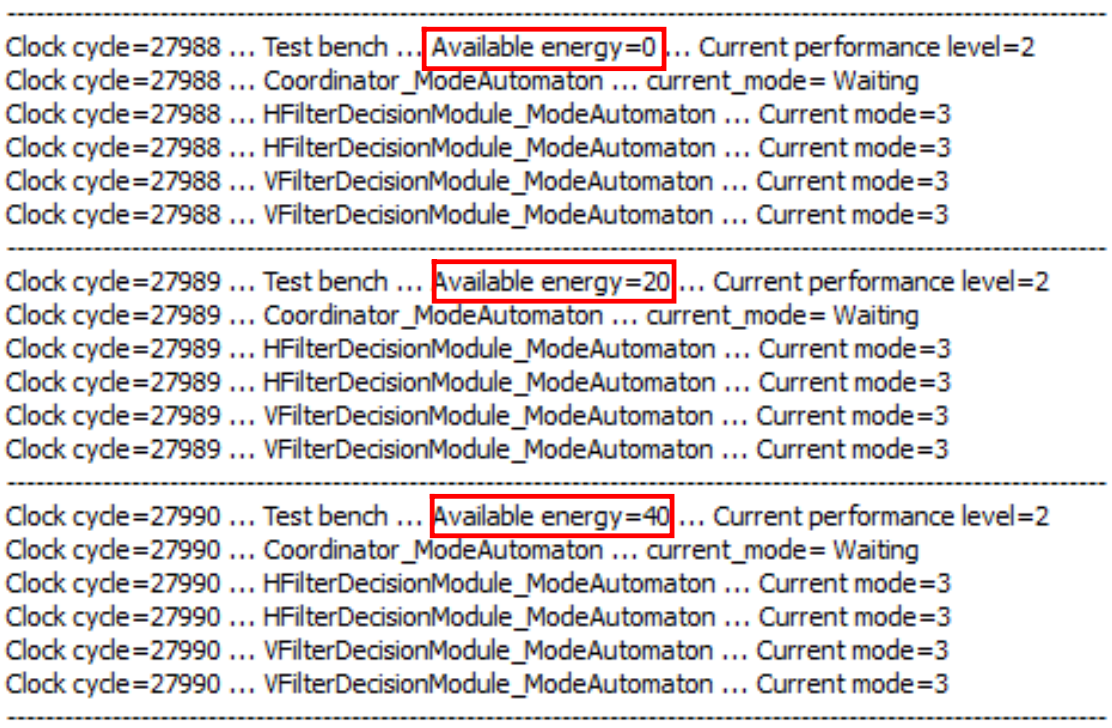


FIG. 6.18 – Passage de la batterie au mode chargement

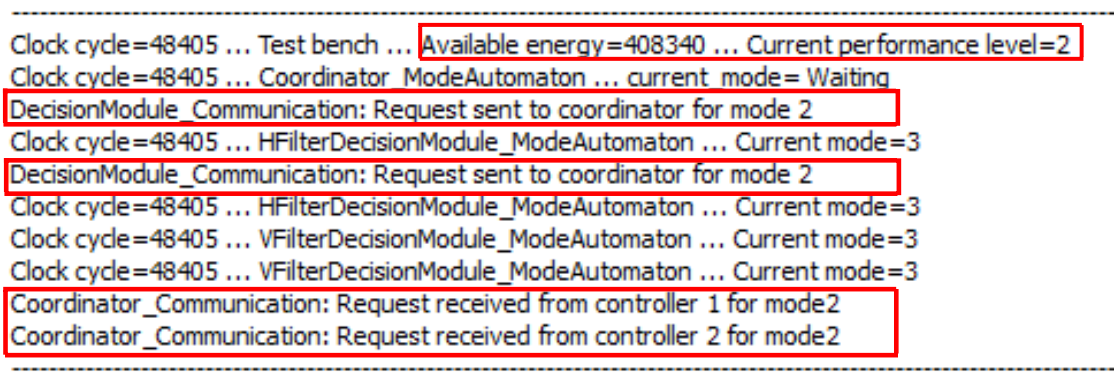


FIG. 6.19 – Envoi des requêtes de reconfiguration par les contrôleurs liés au filtre horizontal pour passer au mode $HFilter\_mode2$

la figure 6.21. Les propositions du coordinateur sont acceptées par les contrôleurs 1,2 et 4. Le système passe à la configuration globale 2 comme le montre la figure 6.22.

6.7 Résultats de synthèse

Après la validation par simulation comportementale, le système de contrôle généré est évalué en termes d'occupation de ressources matérielles pour différents nombres de

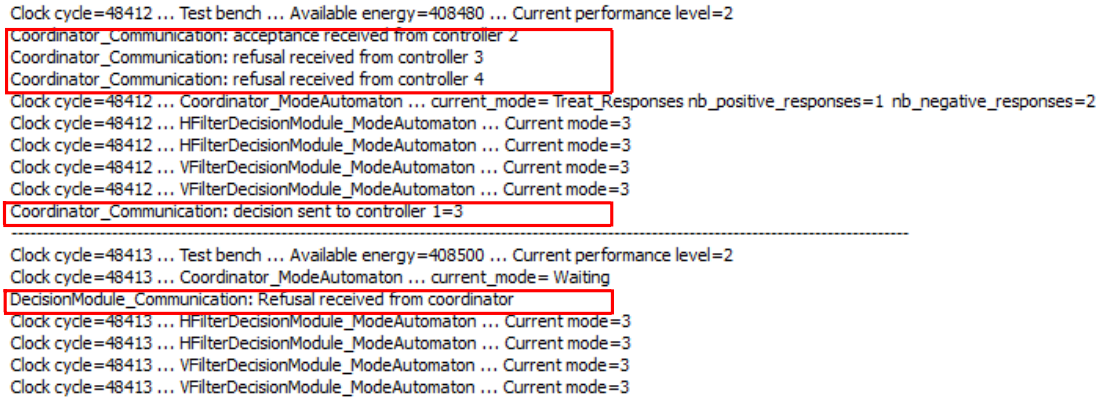


FIG. 6.20 – Refus de la reconfiguration par le coordinateur

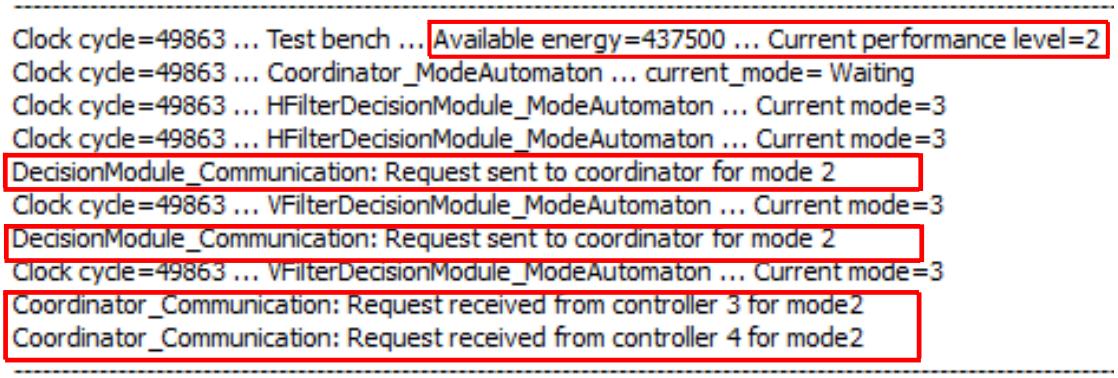


FIG. 6.21 – Envoi des requêtes de reconfiguration par les contrôleurs liés au filtre vertical pour passer au mode $VFilter\_mode2$

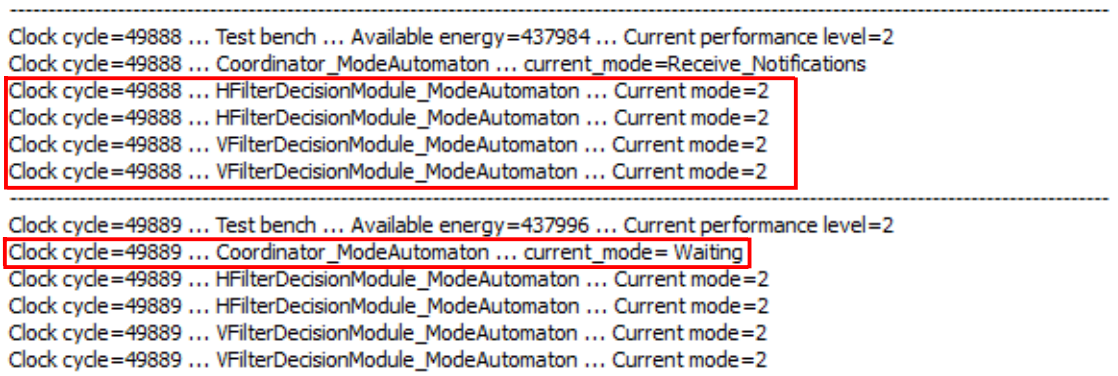


FIG. 6.22 – Passage à la configuration globale 2

régions contrôlées. Les résultats de synthèse sont aussi comparés à ceux du modèle centralisé. Les coûts de la modularité et de la séparation entre communication et traitement

Ressources occupées	Contrôleur Contrôleur	Module d'observation	Module de reconfiguration	Module de communication avec le bus	Module décision
Nombre de Slice Registers	58	2	2	0	54
Nombre de Slice LUTs	240	1	4	34	207

TAB. 6.4 – Détails des ressources occupées par un contrôleur

Ressources occupées	Nombre de régions contrôlées			
	2	4	8	16
Nombre de Slice Registers	54	104	191	363
Nombre de Slice LUTs	112	224	443	911

TAB. 6.5 – Les ressources occupées par le coordinateur selon le nombre de régions contrôlées

dans le modèle semi-distribué sont aussi évalués.

Le tableau 6.4 donne les détails des ressources occupées par un contrôleur. Un contrôleur occupe donc $0,019\%$ des registres et $0,159\%$ des LUTs d'un Virtex6-xc6vlx240t. Le plus grand module du contrôleur est le module de décision qui effectue plus de traitement que les autres modules. Ce module occupe $0,017\%$ des registres et $0,137\%$ des LUTs d'un Virtex6-xc6vlx240t.

L'occupation du coordinateur dépend du nombre de contrôleurs qu'il coordonne puisqu'il implémente une communication peer-to-peer permettant de traiter d'une façon parallèle l'envoi de plusieurs propositions et la réception de plusieurs réponses. L'utilisation d'un autre type de communication (un bus, un NoC, etc) pourrait réduire le nombre de ressources occupées par le coordinateur au coût d'une performance moins importante. Le tableau 6.5 donne les détails de l'occupation du coordinateur selon le nombre de régions contrôlées. Le coordinateur occupe donc de $0,017\%$ des registres et $0,074\%$ des LUTs d'un Virtex6-xc6vlx240t pour 2 régions contrôlées, jusqu'à $0,12\%$ des registres et $0,604\%$ des LUTs d'un Virtex6-xc6vlx240t pour 16 régions contrôlées. Comme le montre le tableau, cette occupation est une fonction linéaire du nombre de régions contrôlées. Cela est dû principalement aux ressources nécessaires pour l'implémentation de la communication peer-to-peer.

L'occupation du modèle semi-distribué (somme de l'occupation des contrôleurs et du coordinateur) est illustrée dans la figure 6.23. Cette occupation est une fonction linéaire du nombre de régions contrôlées puisque les contrôleurs sont réutilisés pour passer d'un degré de parallélisme à un autre et que l'occupation du coordinateur est linéaire elle aussi. Comme le montre la figure, l'occupation du modèle semi-distribué atteint $0,428\%$ des registres et $3,152\%$ des LUTs d'un Virtex6-xc6vlx240t pour 16 régions contrôlées. Ces valeurs peuvent être considérées comme acceptables pour ce grand nombre de régions contrôlées. Cette occupation est encore moins importante pour de plus grands FPGAs tel que le XC7V2000T [150] de la famille Virtex-7.

Pour comparer l'occupation du modèle semi-distribué à celle du modèle centralisé,

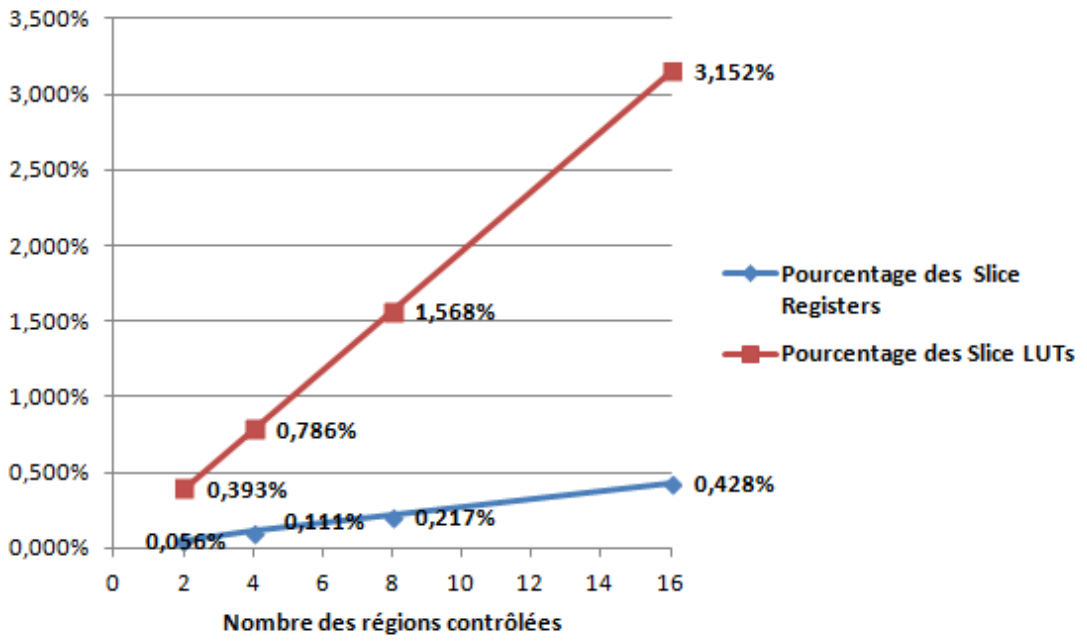


FIG. 6.23 – Le pourcentage de ressources occupées par un système de contrôle semi-distribué

nous avons implémenté des contrôleurs centralisés pour différents nombres de régions contrôlées. Cette implémentation est basée sur un ensemble d'automates de modes synchronisés comme a été expliqué dans la section 6.5.2.2. Le tableau 6.6 illustre l'occupation du modèle centralisé en comparison au modèle semi-distribué. Il montre que l'occupation du modèle semi-distribué est presque trois fois celle du modèle centralisé pour les différents nombres de régions contrôlées. Cette différence d'occupation est due à trois facteurs : 1) la multiplication de ressources occupées en multipliant le nombre de contrôleurs dans le modèle semi-distribué, 2) l'absence de modularité dans l'implémentation des contrôleurs centralisés, et 3) le coût de la séparation entre communication et traitement de la prise de décision dans le modèle semi-distribué.

Pour évaluer le coût de la flexibilité de conception et la réutilisation offerte par le

Ressources occupées		Nombre de régions contrôlées			
		2	4	8	16
Modèle centralisé	Nombre des Slice registers	250	388	662	1209
	Nombre des Slice LUTs	326	499	868	1529
Modèle semi-distribué	Nombre des Slice registers	170	336	655	1291
	Nombre des Slice LUTs	592	1184	2363	4751

TAB. 6.6 – Comparaison entre l'occupation du modèle centralisé et celle du modèle semi-distribué

Ressources occupées	Contrôleur non modulaire	Contrôleur modulaire sans séparation	Contrôleur avec séparation
Nombre des Slice registers	28	28	58
Nombre des Slice LUTs	187	203	240

TAB. 6.7 – Comparaison des occupations des différentes implémentations d'un contrôleur

Ressources occupées	Nombre de régions contrôlées			
	2	4	8	16
Coût en Slice Registers	20	42	86	174
Coût en Slice LUTs	8	-18	75	86

TAB. 6.8 – Coût de la séparation entre communication et traitement dans le coordinateur

modèle semi-distribué proposé, en termes de ressources matérielles, les occupations de différentes implémentations du contrôleur sont comparées : 1) le contrôleur non modulaire, 2) le contrôleur modulaire sans séparation entre la communication et traitement dans le module de décision et 3) le contrôleur avec séparation. Le tableau 6.7 illustre l'occupation de ces différentes implémentations. Le coût de la modularité est 16 LUTs soit $0,011\%$ des LUTs d'un Virtex6-xc6vlx240t. Le coût de la séparation est 30 registres $(0,01\%)$ et 37 LUTs $(0,025\%)$ pour un contrôleur. Le tableau 6.8 illustre le coût de la séparation entre communication et traitement en termes de ressources occupées d'un coordinateur. Ce coût peut atteindre 174 registres soit $0,058\%$ des registres d'un Virtex6-xc6vlx240t et 86 LUTs $(0,057\%)$ pour 16 régions contrôlées. Ces coûts sont donc acceptables devant la flexibilité et la réutilisation offertes par la solution semi-distribuée.

6.8 Implémentation physique en utilisant les outils de Xilinx

Pour la conception du système reconfigurable, le flot Partition-based Partial Reconfiguration [43] de Xilinx, présenté dans le chapitre 2 et illustré par la figure 2.7, est utilisé. Dans cette section, nous décrivons les principales étapes suivies pour notre étude de cas qui sont :

1) la création du système matériel dans EDK, dans laquelle on crée l'architecture du système et on y intègre le contrôle,

2) la création des différentes versions des modules à placer dans les régions reconfigurables,

3) l'implémentation des tâches logicielles de l'application,

4) l'implémentation du système dans PlanAhead : placement les régions reconfigurables, implémentation des différentes configurations du système et création des bitstreams,

5) l'exécution du système.

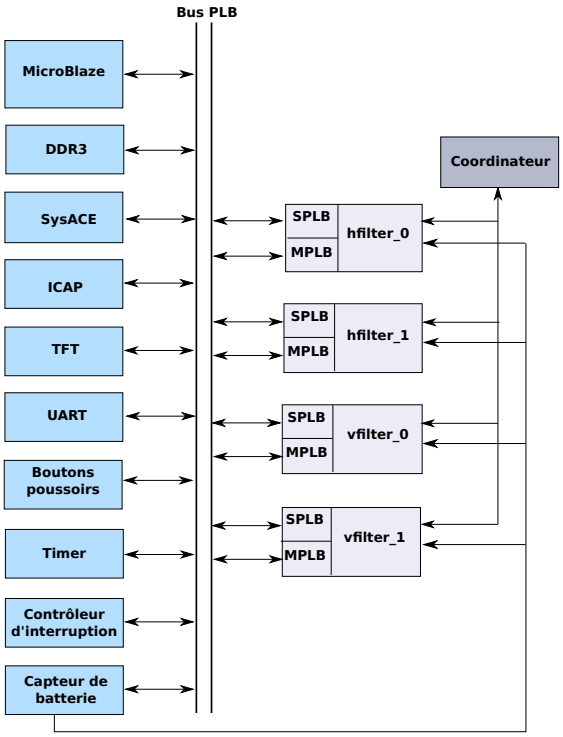


FIG. 6.24 – Architecture du système reconfigurable

6.8.1 Création du système dans EDK

6.8.1.1 L'architecture

Comme expliqué au début du chapitre, dans l'application Downscaler, il s'agit du redimensionnement des images d'une vidéo pour passer du format CIF (Common Intermediate Format : 352x288 pixels) en des images de taille 132x128 pixels. Les images redimensionnées seront affichées successivement sur l'écran (un moniteur connecté au FPGA). L'architecture sur laquelle l'application Downscaler sera exécutée est illustrée dans la figure 6.24. Elle contient un processeur MicroBlaze pour exécuter les tâches logicielles de l'application et pour communiquer avec les accélérateurs matériels et les périphériques. La mémoire locale du processeur est implémentée par les BRAMs du FPGA. L'architecture contient aussi 4 accélérateurs dont deux ($hfilter\_0$ et $hfilter\_1$) sont dédiés au filtre horizontal et les 2 autres ($vfilter\_0$ et $vfilter\_1$) au filtre vertical. Ces accélérateurs ont une interface slave (SPLB) et une interface master (MPLB) avec le bus PLB. L'interface slave est partagée entre le contrôleur et la région contrôlée pour communiquer avec le processeur. L'interface master est utilisée seulement par la région contrôlée pour communiquer avec le bus afin de permettre le transfert des données entre la région contrôlée et la mémoire externe sans passer par le processeur. La mémoire externe utilisée ici est une DDR3. Cette mémoire est accessible à travers un contrôleur de mémoire. Un contrôleur TFT (Thin Film Transistor) est utilisé afin d'afficher successivement les images redimensionnées de la vidéo sur l'écran. L'architecture contient aussi

un UART (Universal Asynchronous Receiver Transmitter) afin d'afficher des messages sur un hyperterminal permettant de suivre l'évolution de l'application et du contrôle. Les boutons poussoirs sont utilisés ici pour donner différentes valeurs au niveau de performance requis durant l'exécution. Ce changement de niveau de performance est traité comme une interruption par le processeur. Pour cela, un contrôleur d'interruption est utilisé. Le capteur de niveau de batterie est émulé par une IP développée en suivant le même principe utilisé dans la simulation de la section 6.6 (un mode de chargement et un mode de déchargement). Un Timer est utilisé afin de déterminer, entre autres, les temps requis pour l'exécution des filtres selon la configuration active. Un contrôleur d'ICAP est utilisé pour communiquer avec l'ICAP afin de réaliser les reconfigurations partielles. Les bitstreams partiels sont stockées dans un CompactFlash. Pour y accéder, un contrôleur System ACE est utilisé. Ce CompactFlash contient aussi le système à charger initialement dans le FPGA (une combinaison entre le bitstream total initial et du logiciel à exécuter par le processeur). La vidéo à traiter est aussi stockée dans le CompactFlash. On pourrait remplacer cela par une entrée caméra. Enfin, l'architecture contient aussi le coordinateur qui est connecté aux contrôleurs des régions reconfigurables.

La figure 6.25 illustre l'implémentation de l'architecture avec l'outil EDK 12.4 de Xilinx en indiquant les interfaces des différents composants. Les interfaces représentées par des carrés sont des interfaces master et celles représentées par des cercles sont des interfaces slave du bus PLB (en jaune), ou du bus local (en bleu) permettant la connexion entre le processeur et les contrôleurs de sa mémoire locale (implémentée par des BRAMs). Les fréquences utilisées pour cette architecture sont 100MHz pour le processeur, 200MHz pour la DDR3, 100MHz pour l'ICAP et 25MHz pour le TFT. Les autres périphériques fonctionnent à 100MHz.

6.8.1.2 Intégration du contrôle semi-distribué

La figure 6.26 illustre l'intégration du contrôleur dans l'accélérateur matériel implémentant le filtre horizontal. Lors de la création d'une IP dans l'outil EDK, lorsqu'on active les options de la logique master (pour implémenter l'interface master) et le burst (dont nous expliquerons l'utilité plus tard), l'IP générée contient trois composants : 1) le *plbv*46*_slave_burst*, 2) le *plbv*46*_master_burst*_01_*v*1 et 3) le *user*$_l$*ogic*. Les deux premiers composants sont des enveloppes d'interface permettant l'adaptation entre les interfaces PLB slave et master et les interfaces IPIC utilisées par tous les IP-Core Xilinx indépendamment du bus utilisé (OPB, PLB ou AXI). Pour intégrer le contrôleur dans cette IP, il faut l'intercaler entre l'enveloppe de l'interface slave et le composant *user_logic* puisqu'ici l'interface slave du bus est partagée par le contrôleur et la région contrôlée. Il faut aussi ajouter le port lié au capteur de niveau de batterie et ceux liés au coordinateur en tant que ports de l'IP. Le composant *user_logic* instancie un composant *hfilterPRR*. C'est ce composant qui va être placé dans une région reconfigurable. Le code VHDL de ce module n'est pas intégré dans le système puisque plusieurs implémentations sont possibles pour ce module. Donc, seule l'interface externe du module est déclaré dans le module *user_logic*.

Pour le coordinateur, il suffit juste de l'adapter au format Xilinx, en lui créant un fichier PAO (Peripheral Analyze Order) pour la déclaration des fichiers vhdl qu'il

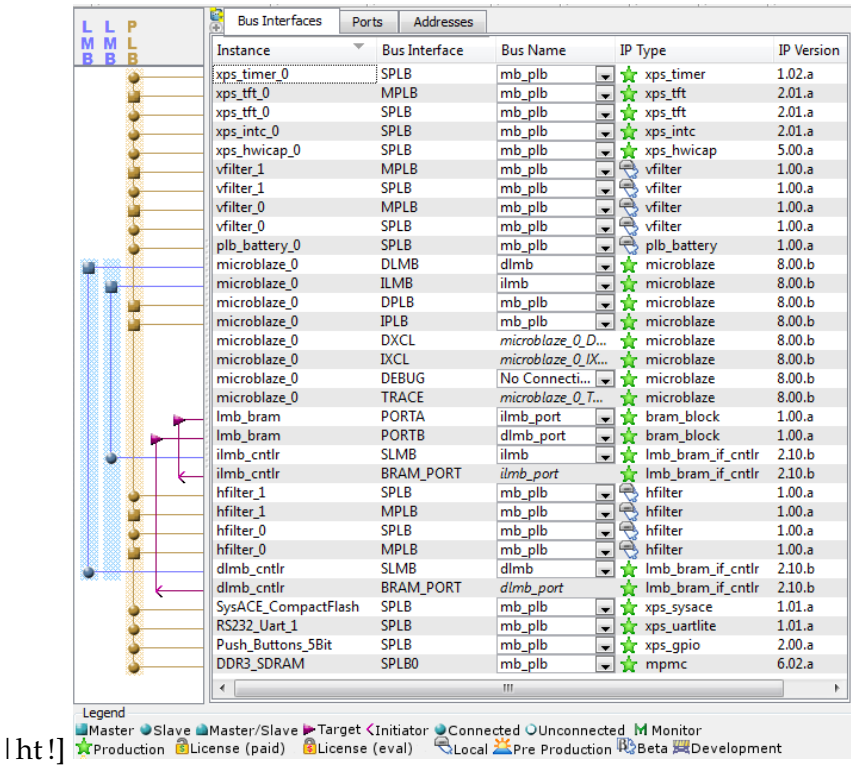


FIG. 6.25 – Arborescence des composants du système dans l'outil EDK

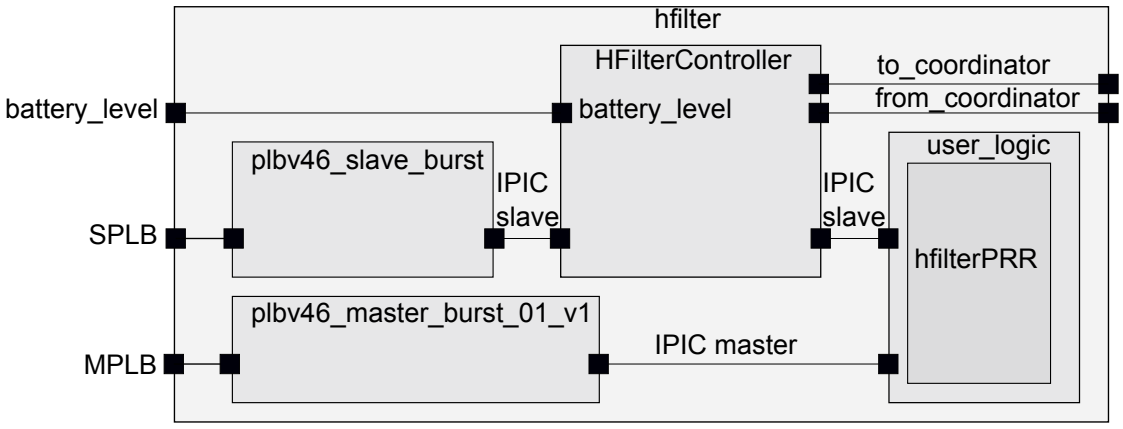


FIG. 6.26 – Intégration d'un contrôleur dans le système reconfigurable

Ressources occupées	Mode		
	$HFilter\_mode1$	$HFilter\_mode2$	$HFilter\_mode3$
Number of Slice Registers	1846	1097	733
Number of Slice LUTs	3597	1997	1154
Number of Block RAM/FIFO	1	1	1
	$VFilter\_mode1$	$VFilter\_mode2$	$VFilter\_mode3$
Number of Slice Registers	2221	1317	865
Number of Slice LUTs	4369	2408	1441
Number of Block RAM/FIFO	1	1	1

TAB. 6.9 – Ressources occupées par les différentes implémentations des filtres

contient et un fichier MPD (Microprocessor Peripheral Description) pour déclarer, entre autres, ses ports. Pour l'intégration des contrôleurs, il s'agit de modifier les fichiers PAO et MPD pour prendre en compte les nouveaux fichiers VHDL et les nouveaux ports par rapport à ceux de l'IP générée par Xilinx. Ensuite, il faut établir la communication entre les IPs $hfilter\_0$, $hfilter\_1$, $vfilter\_0$, $vfilter\_1$ et le coordinateur. Les ports liés à la communication avec le coordinateur ne sont pas déclarés en tant que records VHDL dans le fichier MPD de l'IP $hfilter$, puisque ce fichier ne supporte pas ces types de données. Pour cela, les ports liés à la communication du module $hfilter$ doivent être déclarés séparément (décomposer les records) et liés aux champs correspondants des ports $to\_coordinator$ et $from\_coordinator$ du module $HFilterController$.

Après l'intégration du contrôle, la partie statique du système est synthétisée. Les différentes versions des modules $hfilterPRR$ et $vfilterPRR$ sont synthétisées séparément comme nous expliquerons dans la sous-section suivante. Les résultats de synthèse (partie statique et les différentes versions des modules de reconfigurables) seront ensuite intégrés dans l'outil PlanAhead pour implémenter les différentes configurations du systèmes et en générer les bitstreams nécessaires.

6.8.2 Création des différentes versions des modules reconfigurables

Comme expliqué au début de ce chapitre, les filtres sont exécutés par les régions reconfigurables. Le fonctionnement de la région reconfigurable est donc comme suit : 1) stockage du bloc d'entrée à partir de la mémoire externe dans une FIFO, 2) effectuer le redimensionnement sur ce bloc, 3) stocker le bloc résultat dans une deuxième FIFO et écrire son contenu dans la mémoire lorsque le processeur envoie une requête pour cela. En changeant le mode courant, la taille des FIFOs change et le traitement du filtre doit être adapté pour pouvoir s'appliquer sur un bloc plus au moins grand. La figure 6.9 illustre les résultats de synthèse des différentes versions des filtres qui seront implémentées dans les régions reconfigurables. Comme le montre le tableau, pour les deux filtres, l'occupation en ressources matérielles augmente en allant du mode 1 au mode 2 et du mode 2 au mode 3 en raison de l'augmentation des tailles des blocs d'entrée et de sortie.

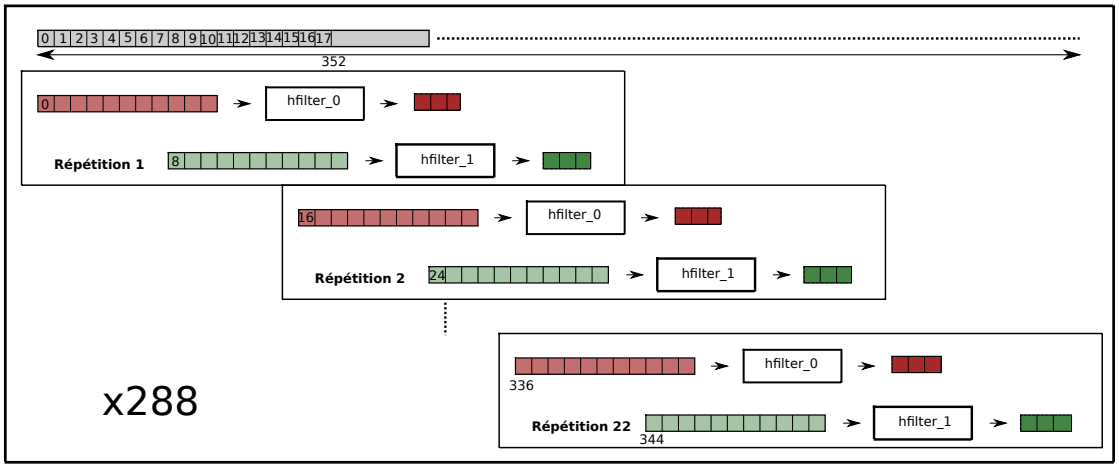


FIG. 6.27 – Traitement des blocs de l'image par les deux accélérateurs implémentant le mode *HFilter_mode*3 du filtre horizontal

6.8.3 Implémentation des tâches logicielles de l'application

Le rôle du processeur est de lire successivement les images de la vidéo contenu dans le CompactFlash et de leur appliquer le redimensionnement à travers les filtres implémentés en matériel. L'application peut être vue comme une boucle dont le nombre d'itérations est celui des images de la vidéo. Ici, nous utilisons une boucle infinie qui retraite la vidéo plusieurs fois. Ce comportement permet de prolonger l'exécution de l'application le plus longtemps qu'on veut. Grâce à l'activation de l'option burst pour les IPs des filtres, il est possible de faire des transferts de plusieurs données concaténées avec une seule commande du processeur, ce qui permet de les accélérer. Ces transferts se déroulent comme suit : 1) le processeur envoie des commandes au bus en indiquant l'adresse du bloc dans la mémoire DDR3 et l'adresse de l'IP auquel il faut envoyer le bloc ainsi que la taille du bloc, 2) l'IP récupère le bloc à partir de la DDR3 à travers son interface master, 3) l'IP exécute le redimensionnement, 4) le processeur envoie une requête permettant à l'IP d'envoyer son bloc résultat à la mémoire DDR3. Les images traitées sont en format RGB, les filtres sont donc appliqués respectivement aux trois composantes R, G et B de l'image d'entrée qui sont stockée dans trois tableaux r, g et b dans la mémoire externe. Pour chaque redimensionnement, le filtre horizontal est appliqué d'abord sur tout le tableau r, puis sur g puis sur b. Le résultat est stocké dans trois tableaux $r2$, $g2$ et $b2$. Ensuite le filtre vertical est appliqué respectivement sur ces trois tableaux. Le résultat final est stocké à nouveau dans les trois premiers tableaux r, g et b. Le contenu de ces tableaux est ensuite organisé en mots de 24 bits représentant les pixels et stocké dans la mémoire externe. L'adresse de ce résultat est donnée au contrôleur TFT pour afficher le résultat sur écran.

Le filtre horizontal est exécuté en parallèle par les deux accélérateurs $hfilter\_0$ et $hfilter\_1$. De même, le filtre vertical est exécuté en parallèle par les deux accélérateurs $vfilter\_0$ et $vfilter\_1$. Les tailles des blocs traités par ces 4 accélérateurs dépendent du mode actif de la région reconfigurable implémentant le filtre. Pour faciliter l'adaptation

Variables	Mode		
	$HFilter\_mode1$	$HFilter\_mode2$	$HFilter\_mode3$
$hfilter\_in\_BlocSize$	35	19	11
$hfilter\_out\_BlocSize$	12	6	3
$hfilter\_Paving$	64	32	16
$hfilter\_HRep$	5	11	22
$hfilter\_1\_Origin$	32	16	8
	$VFilter\_mode1$	$VFilter\_mode2$	$VFilter\_mode3$
$vfilter\_in\_BlocSize$	41	19	14
$vfilter\_out\_BlocSize$	16	8	4
$vfilter\_Paving$	72	36	18
$vfilter\_HRep$	4	8	16
$vfilter\_1\_Origin$	36	18	9

TAB. 6.10 – Variables du code du processeur dont les valeurs dépendent des modes actifs des régions reconfigurables

du code du processeur aux configurations actives des régions reconfigurables, les variables suivantes sont utilisées : $hfilter\_in\_BlocSize$, $hfilter\_out\_BlocSize$, $hfilter\_Paving$, $hfilter\_HRep$ et $hfilter\_1\_Origin$. Ces variables désignent respectivement la taille du bloc en entrée à un accélérateur implémentant le filtre horizontal, la taille du bloc en sortie, le déplacement de l'origine du bloc en entrée à $hFilter\_0$ dans l'image entre deux itérations successives (ce déplacement a la même valeur pour $hfilter\_1$), le nombre de répétitions (dans le sens horizontal) de l'exécution du filtre par chacun des accélérateurs, et l'origine du premier bloc en entrée à $hfilter\_1$. Par exemple, si le mode $HFilter\_mode3$ est actif dans les deux accélérateurs, la taille du bloc en entrée à chacun d'eux est 8 (8 entiers liées à une composante donnée :R, G ou B). La taille du bloc de sortie est 3. La valeur du $hfilter\_Paving$ est 16, ce qui signifie qu'entre deux itérations successives le déplacement de l'origine du bloc en entrée à $hfilter\_0$ dans l'image est 16. La valeur de $hfilter\_HRep$ est 22, ce qui veut dire qu'il y aura $22x288$ répétitions de l'exécution du filtre horizontal par chacun des accélérateurs. L'origine du bloc en entrée de $hfilter\_1$ est $hfilter\_1\_Origin = 8$. La figure 6.27 illustre l'exécution du filtre horizontal pour l'une des composantes R, G ou B lorsque le mode $HFilter\_mode3$ est actif. Pour chaque composante $hfilter\_0$ exécute $22x288$ fois une tâche permettant d'obtenir à partir d'un bloc de taille 11 un bloc de taille 3. Le tableau 6.10 illustre les valeurs des variables pour le filtre vertical et horizontal selon le mode actif.

L'algorithme 9 montre le principe de l'exécution de l'application du côté du processeur. Le processeur envoie les blocs aux accélérateurs à travers une commande permettant à ces derniers d'accéder directement à la mémoire externe où les blocs d'entrée sont stockés. L'accélérateur stocke son bloc d'entrée dans une FIFO, effectue le traitement et enregistre le résultat dans une autre FIFO. Le contenu de cette dernière est ensuite écrit dans la mémoire externe à travers l'interface master lorsque le processeur envoie une commande activant ce transfert de données.

Après le traitement de tous les blocs d'une image, elle est affichée sur l'écran. Le processeur passe ensuite à la communication avec le système de contrôle. Il lit les contenus

Algorithm 9 Exécution des filtres

```
1: read_image(filename,r,g,b) //Lecture de l'image
2: //Stockage des composantes R, G et B de l'image dans les tableaux r, g et b
3: //Filtre horizontal sur la composante R
4: for j = 1 to 288 do
5:     for i = 1 to hfilter_HRep do
6:         //remplissage du bloc d'entrée à hfilter_0 (hfilter_0_in_Block) à partir du
7:         //tableau r
8:         //remplissage du bloc d'entrée à hfilter_1 (hfilter_1_in_Block) à partir du
9:         //tableau r
10:        //Envoie du bloc hfilter_0_in_Block à hfilter_0
11:        //Envoie du bloc hfilter_1_in_Block à hfilter_1
12:        //Récupération du bloc hfilter_0_out_Block de hfilter_0
13:        //Récupération du bloc hfilter_1_out_Block de hfilter_1
14:        //Stockage des contenus des hfilter_0_out_Block et hfilter_1_out_Block dans
15:        //tableau r2
16:    end for
17: end for
18: //Filtre horizontal sur la composante G
19: .....
20: //Filtre horizontal sur la composante B
21: .....
22: //Filtre vertical sur la composante R
23: for i = 1 to 132 do
24:     for j = 1 to vfilter_VRep do
25:         //remplissage du bloc d'entrée à vfilter_0 (vfilter_0_in_Block) à partir du
26:         //tableau r2
27:         //remplissage du bloc d'entrée à vfilter_1 (vfilter_1_in_Block) à partir du
28:         //tableau r2
29:         //Envoie du bloc vfilter_0_in_Block à vfilter_0
30:         //Envoie du bloc vfilter_1_in_Block à vfilter_1
31:         //Récupération du bloc vfilter_0_out_Block de vfilter_0
32:         //Récupération du bloc vfilter_1_out_Block de vfilter_1
33:         //Stockage des contenu des vfilter_0_out_Block et vfilter_1_out_Block dans
34:         //tableau r
35:     end for
36: end for
37: //Filtre vertical sur la composante G
38: .....
39: //Filtre vertical sur la composante B
40: .....
```

des registres de reconfiguration de chaque contrôleur et exécute les reconfigurations requises (si elles existent) à travers l'ICAP. Il notifie ensuite les modules de reconfiguration

et il passe au traitement de l'image suivante. L'algorithme 10 illustre le contenu de la fonction *main* du processeur. Le processeur commence par l'initialisation des modules de l'architecture. Il alloue les zones mémoires afin de permettre le transfert direct entre la mémoire externe et les accélérateurs. Il instancie aussi la table des bitstreams indiquant la correspondance entre le numéro du mode d'une région et le nom du bitstream partiel dans le CompactFlash. Le processeur initialise ensuite les contrôleurs en leur indiquant les modes initiaux (le processeur indique le mode au module de reconfiguration et ce dernier notifie le module de décision) et il passe ensuite à l'exécution du Downscaler.

Algorithm 10 La fonction main du processeur

```
1: //Initialisation des modules SysACE, ICAP, TFT, boutons, etc
2: //Allocation mémoire pour les tableaux r, g, b, r2, g2, b2 et les blocs d'entrées et de
3: //sortie aux accélérateurs
4: //Instanciation de la table des bitstreams
5: //Initialisation des modes actifs des contrôleurs à travers les modules de
6: //reconfiguration
7: while(1){
8: //Si fin de la vidéo, relecture de la première image ,sinon lecture de l'image suivante
9: //Exécution des filtres
10: //Affichage de l'image redimensionnée à l'écran
11: //Lecture des registres de reconfiguration des contrôleurs
12: //Lancer éventuellement les reconfigurations requises
13: //Notifier éventuellement les modules de reconfiguration des contrôleurs
14: //Effectuer éventuellement les adaptations nécessaires des variables du processeur
15: //selon la configuration globale courante
16: }
```

L'appui sur l'un des boutons poussoirs de l'architecture est traité comme une interruption par le processeur. Selon le bouton pressé, le processeur envoie la valeur correspondante du niveau de performance aux 4 contrôleurs.

6.8.4 Implémentation du système dans PlanAhead

Au cours de cette phase, le résultat de synthèse de la partie statique est donné comme entrée à l'outil PlanAhead. Dans cet outil, les modules $hfilterPRR$ et $vfilterPRR$ sont considérés comme des partitions auxquelles on associe le résultat de synthèse des différentes versions implémentées pour ces modules. On obtient donc un système avec le résultat de synthèse de la partie statique et trois versions différentes de synthèse (implémentant trois modes différents) pour chacune des 4 partitions du système.

6.8.4.1 Placement des régions reconfigurables

Les régions reconfigurables sont ensuite placées sur le FPGA dans l'outil PlanAhead. La figure 6.28 montre que les tailles des régions implémentant le même type de filtre sont identiques. Celles implémentant le filtre vertical sont plus grandes puisqu'elles nécessitent plus de ressources. Les tableaux 6.11 et 6.12 donnent les ressources disponibles

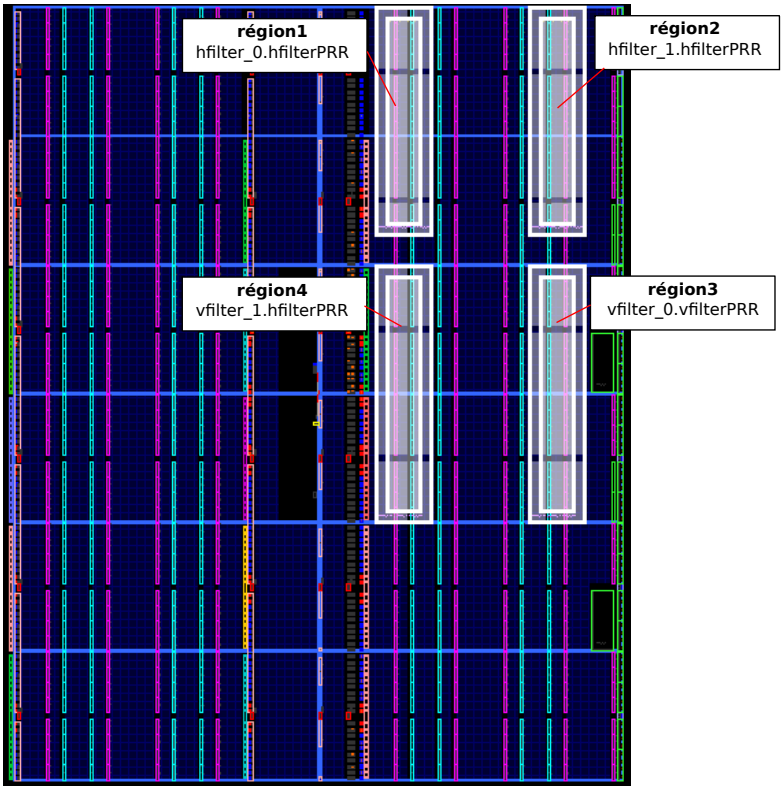


FIG. 6.28 – Placement des régions reconfigurables dans l'outil PlanAhead

dans les régions reconfigurables et celles requises pour les différentes implémentations des modules reconfigurables pour les régions implémentant le filtre horizontal et le filtre vertical, respectivement. Les valeurs contenues dans ces tableaux sont liées aux blocs physiques de Virtex6-xc6vlx240t, ce qui permet de donner plus de détails par rapport aux résultats de synthèse qui sont plus abstraits. Les tailles et les emplacements des régions reconfigurables doivent être choisis de manière à couvrir toutes les ressources requises tout en évitant les risques de congestion à cause de régions pas assez grandes. Xilinx recommande que les ressources requises représentent au maximum 80% des ressources disponibles dans une région reconfigurable [43]. Comme le montre les tableaux 6.11 et 6.12, les pourcentages des ressources occupées par le mode le plus consommant en ressources (mode 1) pour les deux filtres ne dépassent pas les 79%.

6.8.4.2 Implémentation des différentes configurations du système

Après le placement des régions reconfigurables, l'implémentation des différentes configurations du système peut être lancée. Elle suit les phases Translate, MAP et PAR (expliquées dans le chapitre 2). Dans cette étude de cas, on a besoin à la fin du flot d'un bitstream complet implémentant la configuration globale 1 et de trois bitstreams partiels par région reconfigurable. Pour assurer la compatibilité du contenu des régions reconfigurables avec le la partie statique, indépendamment de leurs modes actifs, il faut

Type de ressource	Ressources disponibles dans la région	Ressources requises par le mode 1	Ressources requises par le mode 2	Ressources requises par le mode 3
LUT	4480	3315 (74%)	1853 (42%)	1097 (25%)
FD_LD	8960	1846 (21%)	1097 (13%)	733 (9%)
SLICEL	700	518 (74%)	290 (42%)	169 (25%)
SLICEM	420	311 (75%)	174 (42%)	102 (25%)
RAMBFIFO36E1	14	1 (8%)	1 (8%)	1 (8%)

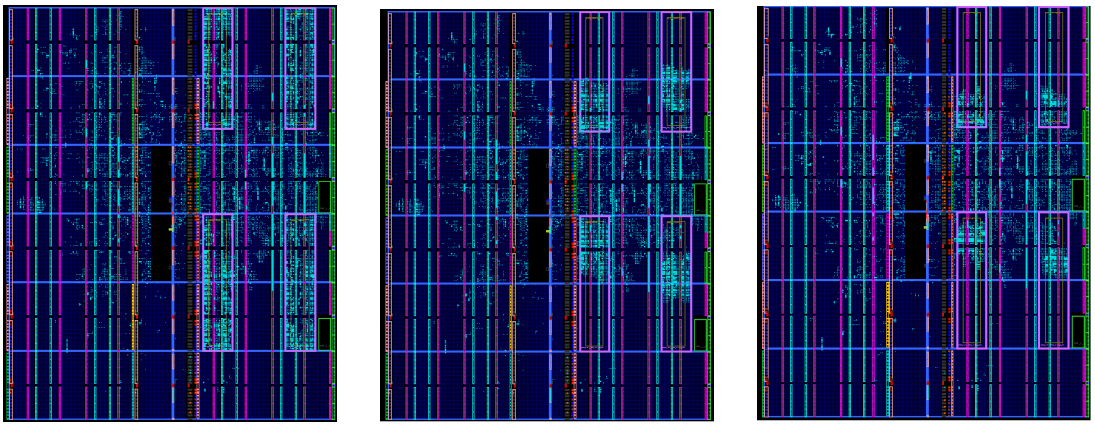
TAB. 6.11 – Ressources disponibles et requises pour les régions reconfigurables implémentant le filtre horizontal

Type de ressource	Ressources disponibles dans la région	Ressources requises par le mode 1	Ressources requises par le mode 2	Ressources requises par le mode 3
LUT	5120	3995 (79%)	2218 (44%)	1343 (27%)
FD_LD	10240	2221 (22%)	1317 (13%)	865 (9%)
SLICEL	800	625 (79%)	347 (44%)	210 (27%)
SLICEM	480	375 (79%)	208 (44%)	126 (27%)
RAMBFIFO36E1	16	1 (7%)	1 (7%)	1 (7%)

TAB. 6.12 – Ressources disponibles et requises pour les régions reconfigurables implémentant le filtre vertical

lancer l'implémentation pour une configuration donnée du système et faire la copie du résultat de l'implémentation de la partie statique, pour l'intégrer directement lors du lancement des autres implémentations du système. Pour cela, l'implémentation du système en activant le mode 1 pour toutes les régions a été lancée en premier. Cela servira après pour la génération d'un bitstream complet pour la configuration initiale du système et 4 bitstreams partiels pour le mode 1 des régions. Il reste ici de générer deux bitstreams partiels par région implémentant le mode 2 et le mode 3. Pour ceci, il suffit de lancer deux implémentations en activant pour la premières le mode 2 pour toutes les régions et pour la deuxième le mode 3 pour toutes les régions. Les bitstreams complets de ces deux implémentations ne seront pas utilisés ici car on a juste besoin de celui de la configuration initiale. D'autres scénarios d'implémentations peuvent être envisagés en activant des numéros de modes différents entre les régions. L'essentiel est qu'en total, pour chaque mode d'une région donnée, il y a au moins une implémentation qui l'active.

Les figures 6.29(a), 6.29(b) et 6.29(c) illustrent le résultat de l'implémentation de la première, la deuxième et la troisième configuration du système respectivement. Les blocs remplis en bleu clair sont ceux utilisés dans l'implémentation. Comme le montre ces figures, la partie statique reste la même pour les trois configurations, alors que le contenu des régions reconfigurables change selon la configuration active. Le tableau 6.13 donne l'occupation en ressources des différentes implémentations du système. Ce tableau montre bien que la différence en nombre de ressources occupées dépend bien



(a) Résultat de l'implémentation de la première configuration du système

(b) Résultat de l'implémentation de la deuxième configuration du système

(c) Résultat de l'implémentation de la troisième configuration du système

FIG. 6.29 – Résultats de l'implémentation du système reconfigurable

Type de ressource	Implémentation 1	Implémentation 2	Implémentation 3
Register	21924 (7%)	17834 (5%)	15768 (5%)
LUT	27241 (18%)	21503 (14%)	18558 (12%)
Slice	9867 (26%)	8287 (21%)	7458 (19%)
IO	118 (19%)	118 (19%)	118 (19%)
RAMBFIFO36E1	42(5%)	42(5%)	42(5%)
RAMBFIFO18E1	10 (1%)	10 (1%)	10 (1%)
DSP48E1	3 (1%)	3 (1%)	3 (1%)
MMCM_ADV	2 (16%)	2 (16%)	2 (16%)

TAB. 6.13 – Ressources occupées par les différentes implémentations du système

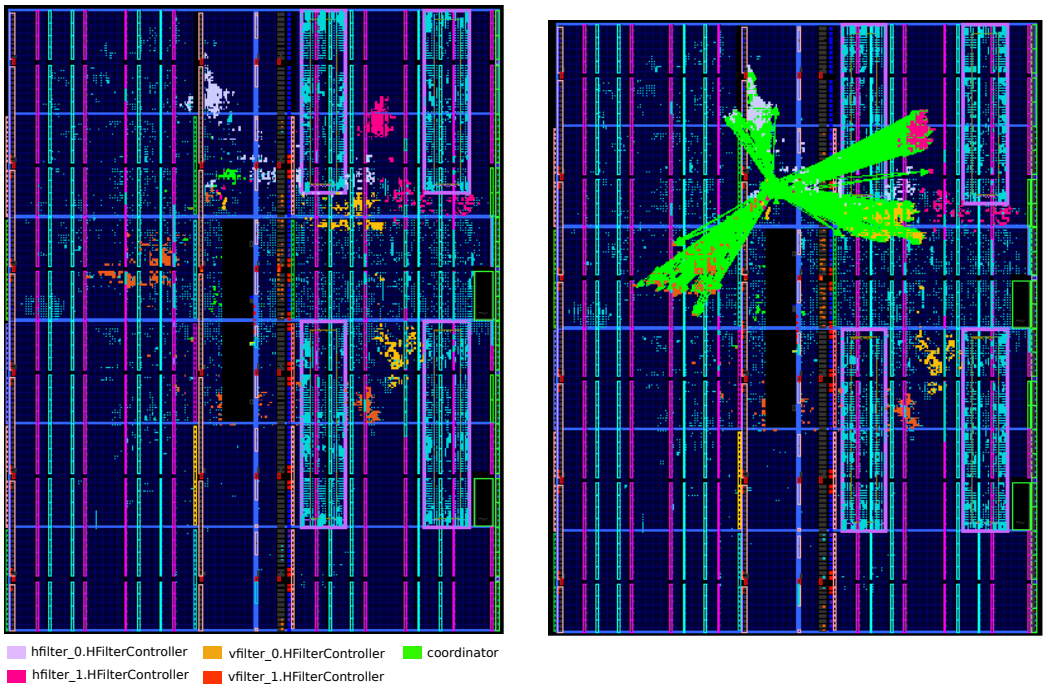
des modes actifs des régions reconfigurables. Le tableau 6.14 illustre la différence entre les implémentations du système en termes de consommation de puissance dynamique. Cette différence est due aux différences de consommation des modes des régions reconfigurables. Ces résultats sont donnés par l'outil Xpower Analyzer de Xilinx.

La fréquence FMax (fréquence du chemin critique) dans les trois configurations est 76,278MHz (après PAR). Pour évaluer le coût de la dynamicité sur la fréquence d'horloge, nous avons implémenté le même système en version statique (les modules $hfilterPRR$ et $vfilterPRR$ implémentent le mode 1). La fréquence FMax après PAR était de 85,092MHz. Le coût de la dynamicité dans ce cas est une dégradation de $10,36\%$ de la fréquence d'horloge, ce qui est prévu par Xilinx [43]. L'impact du système du contrôle sur la fréquence FMax est aussi évalué en implémentant le même système reconfigurable sans les contrôleurs et le coordinateur. La fréquence FMax de ce système est 80MHz. Le système de contrôle a donc dégradé légèrement la fréquence FMax $(4,65\%)$.

La figure 6.30(a) illustre l'emplacement du système de contrôle dans le FPGA. L'outil

Implémentation	Consommation de puissance (mW)
Implémentation 1	589
Implémentation 2	583
Implémentation 3	580

TAB. 6.14 – consommation de puissance des différentes implémentations du système



(a) Emplacement du système de contrôle dans le FPGA

(b) La communication entre contrôleurs et coordinateur

FIG. 6.30 – Résultats de l'implémentation du système de contrôle

planAhead a placé chaque contrôleur de façon à mettre une partie de chacun d'eux proche du coordinateur, afin de minimiser la longueur des connexions entre les contrôleurs et le coordinateur. Pour les ressources des contrôleurs qui ne communiquent pas avec le coordinateur, elles sont placées plus loin selon l'espace libre et la stratégie du placement. Les connexions entre les contrôleurs et le coordinateur sont colorés en vert clair dans la figure 6.30(b).

6.8.4.3 Génération des bitstreams

Après l'implémentation des différentes configurations du système, la génération des bitstreams peut être lancée. Cette étape permet de générer pour chaque implémentation du système, un bitstream total et des bitstreams partiels selon les versions activées des régions reconfigurables. Le tableau 6.15 donne les tailles des bitstreams générés. Il

Bitstream	Taille
bitstream total	8,80 Mo
bitstream partiel	300 Ko

TAB. 6.15 – Taille des bitstreams

	Région1	Région2	Région3	Région4
$H/VFilter\_mode1$	h_1_1.bit	h_2_1.bit	v_1_1.bit	v_2_1.bit
$H/VFilter\_mode2$	h_1_2.bit	h_2_2.bit	v_1_2.bit	v_2_2.bit
$H/VFilter\_mode3$	h_1_3.bit	h_2_3.bit	v_1_3.bit	v_2_3.bit

TAB. 6.16 – Noms des bitstreams partiels

montre que tous les bitstreams ont la même taille. C'est parce que toutes les régions ont la même largeur, ce qui correspond au même nombre de trames de configurations. La hauteur d'une trame est marqué par des traits verticaux en bleu dans la figure 6.28 par exemple. Comme le montre cette figure, toutes les régions occupent 2 trames en hauteur. Ayant la même largeur, les régions occupent donc le même nombre de trames au total, ce qui fait qu'ils ont la même taille de bitstreams. Le tableau 6.16 illustre les noms donnés pour chaque bitstreams partiels. Ces noms seront utilisé par le processeur pour charger ces bitstreams à travers l'ICAP.

Le bitstream total est ensuite fusionné avec l'exécutable de la partie logicielle de l'application (le programme qui sera exécuté par le processeur MicroBlaze). Le bitsream résultant est converti dans un format ACE pour qu'il puisse être lu du compact flash et chargé dans le FPGA. Pour l'exécution, le compactFlash doit donc contenir le fichier ACE, les bitstreams partiels et la vidéo à traiter.

6.8.5 Exécution du système

L'évolution du système est suivi par l'hyperterminal. Le niveau de batterie est initialisé à 1.000.000 d'unités de consommation. La figure 6.31 illustre le contenu de l'hyperterminal au début de l'exécution. Elle montre qu'on affiche pour chaque image traitée, les nombres de cycles d'horloge requis pour l'exécution du filtre horizontal, du filtre vertical et de leur somme (le nombre de cycles requis pour l'exécution du redimensionnement d'une image). Le niveau de la batterie est aussi affiché ainsi que les états courants des régions reconfigurables et leurs éventuelles reconfigurations requises. Les nombres de cycles d'horloge sont donnés par un timer. Les instructions d'activation et de désactivation de ce timer sont gérées par le code exécuté par le processeur. La figure 6.31 montre que quand le mode actif dans toutes les régions est le mode 1, le nombre de cycles nécessaire pour le redimensionnement d'une image est aux alentours de 80.100.000 cycles.

Quand le niveau de la batterie se trouve au-dessus du seuil 750000 (voir les automates des figures 6.5 et 6.6), la reconfiguration requise par les modules de reconfiguration des contrôleurs est le mode 2 comme le montre la figure 6.32. Le processeur lance les reconfigurations à travers l'ICAP en utilisant les noms des bitreams correspondants du

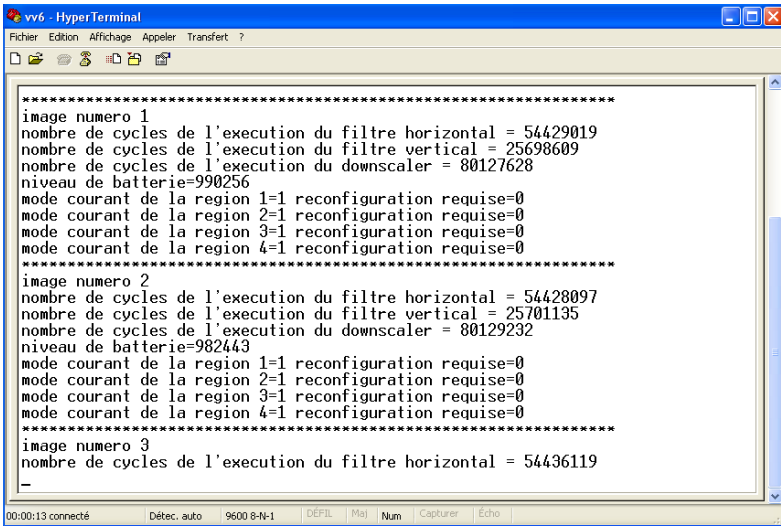


FIG. 6.31 – Début de l'exécution de l'application

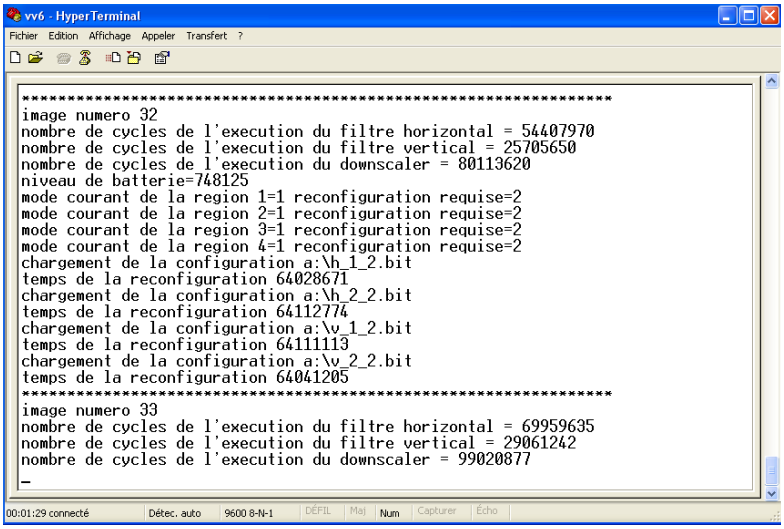


FIG. 6.32 – Reconfiguration des régions en modes $H/VFilter\_mode2$

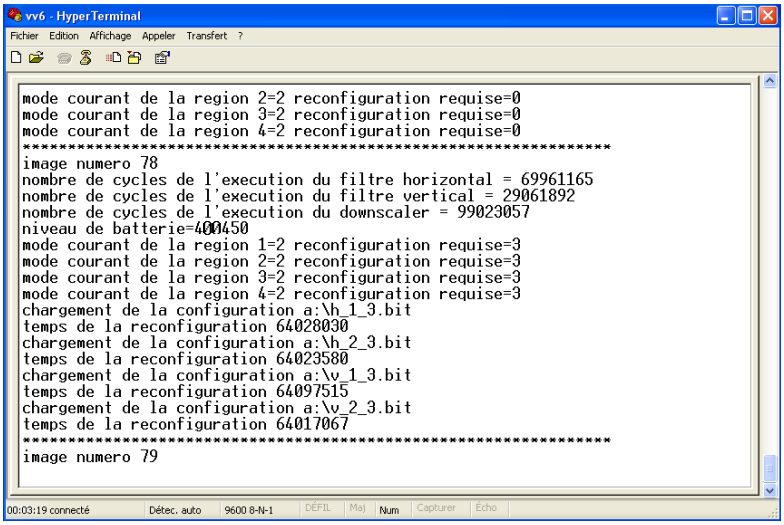


FIG. 6.33 – Reconfiguration des régions en modes $H/VFilter\_mode3$

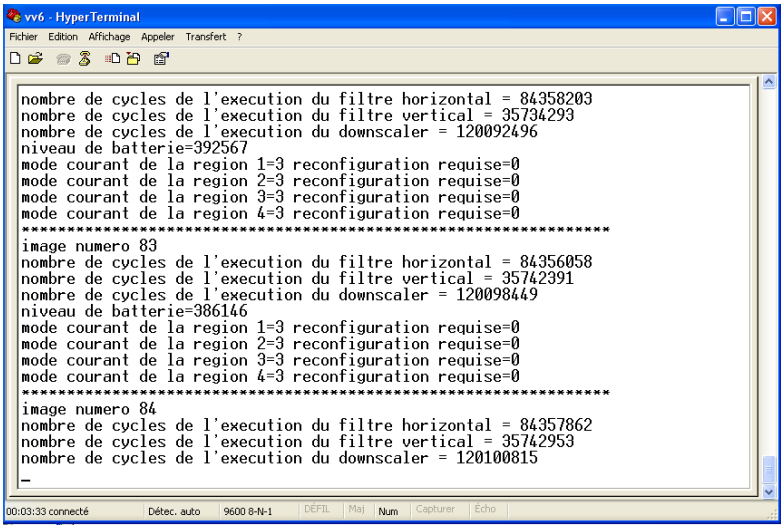


FIG. 6.34 – Exécution en modes $H/VFilter\_mode3$

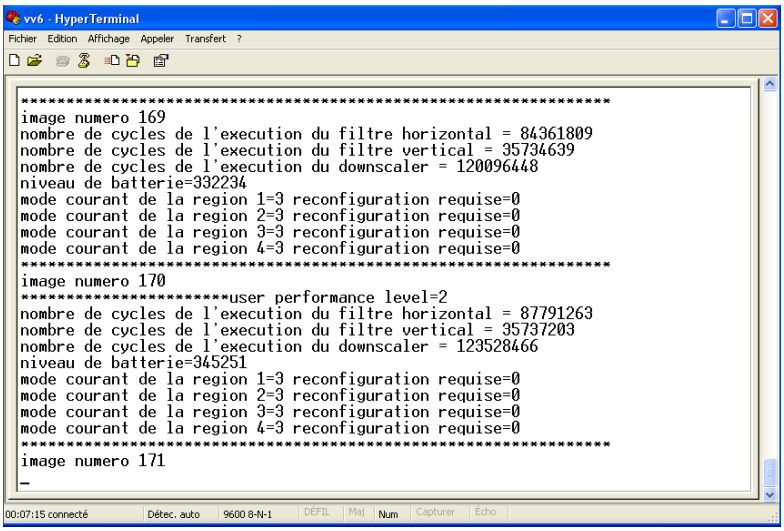


FIG. 6.35 – Changement du niveau de performance requis à 2

tableau 6.16. Le nombre de cycles nécessaire pour le chargement de chaque bitstream partiel est presque 64.000.000. Ces cycles incluent la communication avec le compact-Flash pour la lecture du bitsream et son chargement à travers l'ICAP. La reconfiguration des 4 régions demande presque 2s. Ce temps de reconfiguration peut être réduit significativement en utilisant un module matériel qui gère la communication avec l'ICAP. L'implémentation d'un tel module est étudiée récemment par certains travaux. Dans [63], les auteurs ont montré qu'on peut reconfigurer une région de 7840 LUTs et flip-flops LUTs, 112 DSP et 28 BRAMs en 18,2ms.

En chargeant le mode 2 dans les régions reconfigurables, la performance du downscaler diminue et son temps d'exécution augmente à 99.000.000 cycles par image, comme le montre la figure 6.32. Quand le niveau de la batterie descend au dessous de 401785 (seuil des régions implémentant le filtre vertical), la reconfiguration requise par les contrôleurs est le mode 3 comme le montre la figure 6.33. Le processeur lance les reconfigurations requises. La performance diminue et le temps d'exécution augmente à 120.000.000 cycles par image, comme le montre la figure 6.34. Quand le niveau de la batterie atteint 332234, l'utilisateur appuie dur le bouton correspondant au niveau de performance 2, comme le montre la figure 6.35. Aucune reconfiguration n'est requise par les contrôleurs puisque le niveau de batterie n'a pas atteint 437500. Quand ce niveau est dépassé, les reconfigurations requises par les contrôleurs ciblent les modes $H/VFilter\_mode2$, comme le montre la figure 6.36. Les figures 6.37 et 6.38 illustre le changement du niveau de performance à 1 et la reconfiguration du système vers la configuration 1 quand le niveau 800000 est dépassé en réponse à la requête de l'utilisateur.

Pour compter le nombre exact d'images traitées pour une batterie initialisée à 1.000.000 d'unités de consommation, les affichages sont supprimés du programme du processeur ainsi que les instructions du timer. Avec un système statique implémentant la configuration 1, l'énergie disponible initialement dans la batterie permet de redimensionner 230 images. Grâce au contrôle de l'adaptation dynamique, le système devient

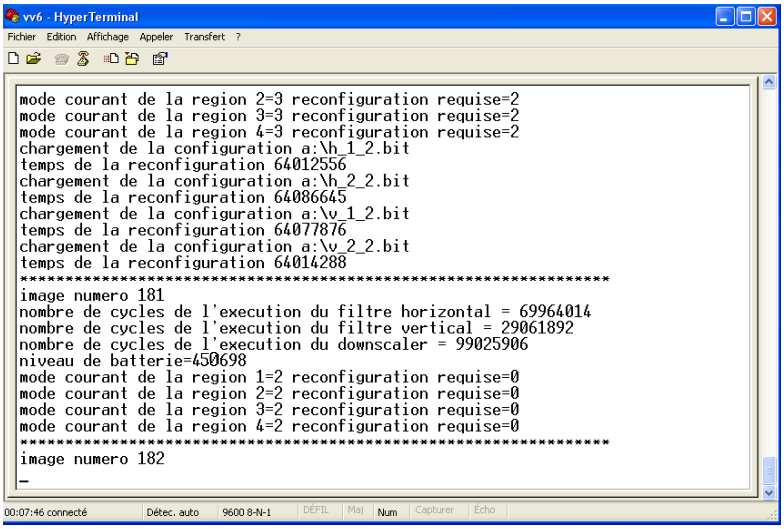


FIG. 6.36 – Reconfiguration des régions en modes $H/VFilter\_mode2$

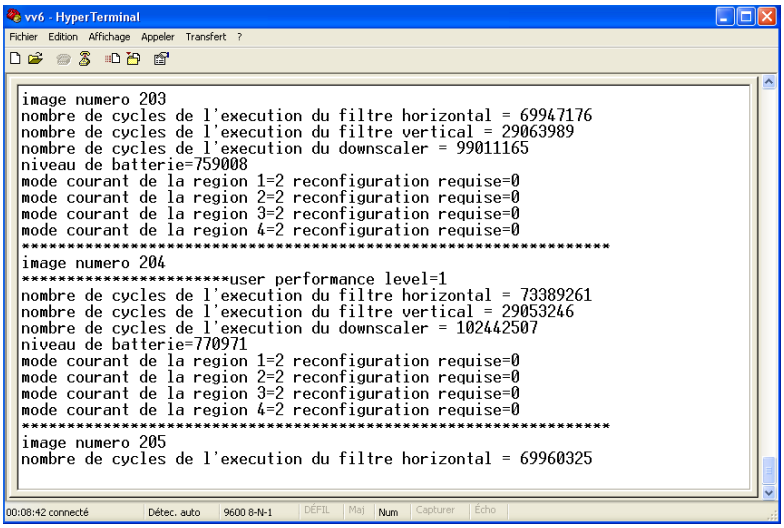


FIG. 6.37 – Changement du niveau de performance requis à 1

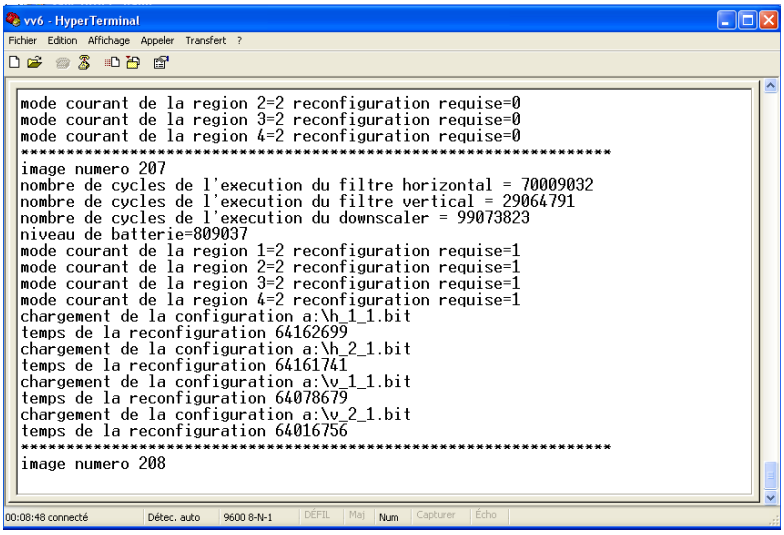


FIG. 6.38 – Reconfiguration des régions en modes $H/VFilter\_mode1$

capable de gérer les exigences de l'utilisateur en termes de performance et d'économiser en même temps de l'énergie. Faisant un compromis entre performance et énergie, le reconfiguration dynamique du système a permis de redimensionner 251 images pour le même niveau de batterie, soit une augmentation de 9,13%. Ce compromis peut être affiné encore plus en explorant d'autres implémentations des filtres de l'application.

6.9 Conclusion

Dans ce chapitre, nous avons validé le modèle de contrôle semi-distribué proposé dans ce travail à travers une application de traitement d'images. La réutilisabilité et la scalabilité de ce modèle ont été évaluées en variant le nombre de contrôleurs distribués considérés dans les systèmes de contrôle modélisés. Les codes générés ont été simulés en utilisant différents scénarios. Les simulations ont montré l'avantage de l'implémentation matérielle des systèmes de contrôle en termes de temps d'exécution. L'évaluation du coût du modèle proposé en termes de ressources matérielles utilisées a été faite en se basant sur les résultats de synthèse pour des systèmes de contrôle ayant des nombres différents de contrôleurs. Cette évaluation a montré que ce coût est acceptable pour un Virtex6-xc6vlx240t (jusqu'à 3% des slice LUTs pour un système composé de 16 contrôleurs et d'un coordinateur). Il peut être beaucoup moins contraignant pour des FPGAs de plus grande taille tels que le XC7V2000T [150] de la famille Virtex-7.

Un système de contrôle composé de 4 contrôleurs et d'un coordinateur a été intégré dans un système reconfigurable pour une implémentation physique sur FPGA. Le flot de conception de ce système a été détaillé dans ce chapitre allant de la création du système dans l'outil EDK de Xilinx, l'association des contrôleurs aux composants reconfigurables et l'intégration du coordinateur, jusqu'au placement physique des régions reconfigurables et la génération des bitstreams. L'exécution du système reconfigurable

implémenté a permis d'exploiter la RDP (Reconfiguration Dynamique Partielle) pour assurer l'objectif du modèle de contrôle semi-distribué pour cette étude de cas, qui est l'adaptation dynamique aux changements liés aux contraintes de performance et de consommation d'énergie du système.

Chapitre 7

Conclusion et perspectives

7.1 Conclusion générale

L'objectif de ce travail était de proposer une méthodologie de conception du contrôle des systèmes reconfigurables sur FPGA qui permet d'améliorer la productivité des concepteurs et assurer une implémentation efficace du contrôle. Pour réaliser ces objectifs, cette méthodologie se base sur deux points. Le premier point est un modèle de contrôle semi-distribué assurant la flexibilité, la réutilisabilité et la scalabilité du contrôle. Le deuxième point est un flot de conception automatisant la génération du code à partir d'une modélisation à haut niveau d'abstraction visant une implémentation matérielle du contrôle. Cette méthodologie a été validée en la mettant en oeuvre sur une application de traitement d'images. Dans ce qui suit, ces trois aspects de la méthodologie proposée sont résumés.

7.1.1 Un modèle de contrôle semi-distribué

Afin d'assurer la flexibilité, la réutilisabilité et la scalabilité du contrôle, le modèle proposé se base sur les points suivants :

- la distribution des services nécessaires pour l'adaptation dynamique (l'observation, la décision et la reconfiguration) des régions reconfigurables à travers des contrôleurs modulaires favorisant la réutilisabilité de ces contrôleurs et leur adaptation à d'autres systèmes,
- la prise en compte des situations où la coordination entre les décisions des contrôleurs distribués est nécessaire afin de respecter les contraintes/objectifs globaux du système, ce qui permet de viser différents problèmes de contrôle,
- un modèle semi-distribué (hiérarchique) de prise de décision basé sur la décomposition du contrôle en des problèmes locaux gérés par les contrôleurs distribués et un problème global géré par le coordinateur. Ceci permet aux contrôleurs de garder leurs visions locales et au coordinateur de ne pas entrer dans les détails de prise de décision de ces derniers. Cela facilite ainsi la réutilisation des contrôleurs et du coordinateur et assure la scalabilité du modèle de contrôle. Ce modèle est aussi facilement adaptable à différentes plates-formes (mono-FPGA, multi-FPGAs, FPGA 3D, etc) en adaptant les niveaux de hiérarchie dans le modèle de prise de décision,
- et l'utilisation du formalisme d'automates de modes pour la conception de la prise de décision semi-distribuée, ce qui facilite la conception grâce aux sémantiques claires de ce formalisme et permet d'abstraire le contrôle par rapport à l'implémentation physique. Cela facilite l'adaptation du modèle de contrôle proposé à différentes implémentations et différentes plates-formes.

Le modèle de contrôle proposé est donc composé d'un ensemble de contrôleurs distribués et d'un coordinateur. Chaque contrôleur est composé de trois modules effectuant chacun une des trois tâches suivantes : 1) l'observation des des événements susceptibles de déclencher l'adaptation du composant contrôlé, 2) la prise de décision concernant les adaptations nécessaires et 3) la réalisation de l'adaptation/reconfiguration du composant contrôlé. Chaque contrôleur est associé à une région reconfigurable du système afin d'assurer son auto-adaptation durant l'exécution. Les modules d'observation des contrôleurs collectent des informations qui peuvent être liées au comportement des régions contrôlées ou provenir d'un autre composant tel qu'un capteur, par exemple. Ces informations sont ensuite prises en compte par les modules de décision, qui décident si une reconfiguration des régions contrôlées est requise. Les décisions des contrôleurs sont coordonnées par le coordinateur afin de garantir une configuration globale qui respecte les contraintes/objectifs globaux du système. Après l'autorisation des reconfigurations par le coordinateur, elles peuvent être lancées par les modules de reconfiguration des contrôleurs. La distribution des services de reconfiguration entre contrôleurs leur permet de bénéficier facilement de la reconfiguration parallèle dans les systèmes multi-FPGAs et ou les futures technologies d'FPGA.

Le modèle de prise de décision semi-distribué proposé dans ce travail s'inspire du formalisme d'automates de modes. Ce modèle est composé des modules de décision des contrôleurs et du coordinateur. Chacun d'entre eux est modélisé par un automate

de modes, ce qui donne des automates de modes parallèles et communicants. Pour les automates de modes des contrôleurs, chaque mode représente un mode/configuration de la région contrôlée. Chaque automate peut faire les actions suivantes : envoyer une requête de reconfiguration au coordinateur, répondre par acceptation ou refus à une proposition de reconfiguration du coordinateur, envoyer une commande de reconfiguration au module de reconfiguration en cas de l'autorisation de celle-ci, et la notification du coordinateur après le chargement du bitstream requis dans la région contrôlée. Pour l'automate du coordinateur, chaque mode représente une activité dans le processus de coordination : l'attente de requêtes de reconfiguration, l'envoi de propositions, etc. Pour assurer la coordination, le coordinateur détient une table contenant les configurations globales possibles du système. Cette table est déterminée à la phase de conception. Dans ce travail, nous ne traitons pas les mécanismes de la détermination de cette table, mais nous supposons qu'elle est déjà construite et prête à être utilisée par le coordinateur. A la réception d'une requête, le coordinateur consulte cette table pour déterminer les configurations globales satisfaisant la requête. Puis, détermine si cette requête peut être autorisée directement ou qu'elle nécessite l'adaptation d'autres régions du système, auquel cas il propose des reconfigurations aux contrôleurs des régions concernées. Si plus d'une configuration globale satisfait la requête, les configurations possibles sont triées selon la stratégie du coordinateur et sont traitées une par une dans le processus de coordination. Ce dernier se termine si les réponses des contrôleurs pour une possibilité donnée sont toutes favorables (auquel cas la reconfiguration est autorisée) ou si aucune des possibilités n'a été acceptée par tous les contrôleurs concernés (auquel cas la reconfiguration est refusée). Selon la stratégie du coordinateur, les actions associées aux modes et aux transitions de son automate peuvent être implémentées de différentes manières. Par exemple, si l'objectif du coordinateur était d'optimiser certains critères, il déterminerait les possibilités à traiter dans le processus de coordination et leur ordre de traitement en utilisant un mécanisme différent que si son objectif était de respecter un ensemble de contraintes. Cette flexibilité du mécanisme de coordination modélisé par l'automate de modes du coordinateur facilite donc son adaptation à différents problèmes de contrôle. Dans ce travail, nous avons considéré une des implémentations possibles de l'algorithme de coordination afin de valider le modèle de contrôle proposé.

7.1.2 Un flot de conception de la modélisation à la génération automatique de code

L'objectif du flot de conception proposé est d'automatiser la génération du code du contrôle. Il est basé sur une approche IDM (Ingénierie dirigée par les modèles) qui permet, à partir de modèles de haut-niveau d'abstraction utilisant le standard UML MARTE, de générer du code VHDL pour l'implémentation matérielle du contrôle. Cette approche permet donc de rendre les détails d'implémentation transparents aux concepteurs et accélère la phase de conception en évitant les erreurs dues à la manipulation directe de ces détails. Le flot de conception suit trois étapes : 1) la modélisation et le déploiement du système, 2) les transformations de modèles, et 3) la génération de code. Cette approche a été implémentée dans le cadre de l'environnement Gaspard2 dédié à la conception des systèmes embarqués basée sur l'IDM.

La modélisation des systèmes de contrôle semi-distribué traitent deux points qui sont la structure/architecture et le comportement. Pour le premier point, les composants UML sont utilisés. Ces composants peuvent être élémentaires ou composés permettant de modéliser ainsi un module d'un contrôleur, un contrôleur en entier, un coordinateur, etc. La communication entre ces composants est assurés à travers leurs ports et les connexions entre eux. Pour modéliser l'aspect comportement, le standard MARTE offre des concepts de modélisation du comportement modal qui sont inspirés du formalisme des automates de modes. Ceci convient bien pour modéliser le comportement des modules de décision des contrôleurs qui sont inspirés du même formalisme. Cependant, les automates de modes des modules de décision intègrent des détails de communication entre le module de décision et le coordinateur, et entre le module de décision et des autres modules du contrôleur. Pour rendre ces détails transparents aux concepteurs, il faut minimiser les données qu'ils doivent modéliser et rendre le reste générable automatiquement. Dans le cas de l'automate des modes du module de décision, le concepteur doit juste indiquer les modes de l'automate, les conditions d'envoi des requêtes de reconfiguration et les conditions d'acceptation/refus des propositions de reconfiguration. L'automate est donc modélisé par une machine à états UML appliquant les stéréotypes MARTE pour le comportement modal. Chaque état de cette machine correspond à un mode de la région contrôlée. Pour faciliter la modélisation des conditions de requêtes et d'acceptation/refus aux propositions, celles-ci sont attachées aux transitions de la machine à états, ce qui permet de déduire le mode source et le mode cible de la requête/proposition. Pour faire la différence entre ces deux types de conditions, des extensions au profil MARTE ont été proposées. Quant à l'automate de modes du coordinateur, il n'est modélisé en utilisant UML mais il est généré automatiquement puisqu'il s'agit du même automate de modes moyennant le changement de la table de configurations globales gérée par le coordinateur et les contrôleurs coordonnés. Pour cela, le concepteur ne modélise que ces deux points. Là aussi, des extensions du profil MARTE ont été proposées pour permettre la modélisation de ces aspects.

Pour arriver jusqu'à la génération du code, une chaîne de transformation a été utilisée. Cette chaîne réutilise certaines transformations qui existent déjà dans Gaspard2 et propose de nouvelles transformations adaptées à la cible VHDL. Le modèle résultant de cette chaîne de transformations est donné à un outil de génération de code basé sur des règles de génération adaptées aux concepts du contrôle semi-distribué. Cet outil permet principalement de traduire la structure modélisées des systèmes de contrôle selon les concepts de VHDL, d'implémenter les automates des contrôleurs et du coordinateur et de mettre en oeuvre un protocole de communication entre les contrôleurs et le coordinateur.

7.1.3 La mise en oeuvre du contrôle semi-distribué sur une application de traitement d'images

Le flot de conception proposé a été appliqué pour modéliser et générer le code d'un système de contrôle ciblant une application de traitement d'images. Les codes générés ont été validés par simulation. Le coût en termes de ressources utilisées a été évalué par la synthèse pour des systèmes de contrôle ayant différents nombres de contrôleurs.

Parmi les systèmes générés, un système de contrôle composé de 4 contrôleurs et

d'un coordinateur a été intégré dans un système reconfigurable développé pour la validation physique du contrôle. L'exécution du système reconfigurable implémenté a permis d'exploiter la RDP (Reconfiguration Dynamique Partielle)pour assurer l'objectif du modèle de contrôle semi-distribué dans le cadre de l'application de traitement d'images étudiée, qui est l'adaptation dynamique aux changements liés aux contraintes de performance et de consommation d'énergie du système.

7.2 Perspectives

7.2.1 Optimisation du mécanisme de coordination

Comme expliqué dans le chapitre 6, l'implémentation du mécanisme de coordination proposé dans ce travail fait qu'un processus de coordination dure $4 + nb\_coordination\_rounds * 4$ cycles d'horloge (sans compter le temps nécessaire pour le chargement de bitstreams). Cette durée dépend donc du nombre de tours de coordination (nombre de tours) jusqu'à avoir des réponses favorisant l'autorisation de la reconfiguration. Une solution pour réduire cette durée est de remplacer le mécanisme basé sur propositions/réponses par un mécanisme de vote permettant d'avoir un seul tour des contrôleurs concernés. Dans ce cas, après avoir déterminé la liste des possibilités à considérer dans le processus de coordination, le coordinateur détermine les contrôleurs concernés par au moins une de ces possibilités (union des ensembles de contrôleurs concernés par chaque possibilité). Le coordinateur envoie ensuite à ces contrôleurs une requête leur demandant de voter tous les changements possibles de leurs modes courants. Chaque contrôleur envoie ses votes dans un seul transfert de données. Dans ce cas, la réponse du contrôleur peut avoir la forme d'une séquence binaire où chaque bit (1 pour acceptation et 0 pour refus) correspond au vote pour le passage entre le mode courant et le mode dont le numéro est indiqué par l'indice du bit dans la séquence. A la réception des réponses des contrôleurs, le coordinateur fait la somme des votes pour chaque possibilité et autorise celle qui a eu des votes favorables par tous les contrôleurs qu'elle concerne (équivalent à des acceptations envoyées par tous les contrôleurs concernés dans l'implémentation proposée auparavant). Si plus d'une possibilité a eu des réponses favorables pour tous les contrôleurs concernés, le coordinateur en autorise une selon l'ordre qu'il a utilisé pour trier la liste des possibilités.

7.2.2 Implémentation d'autres stratégies du coordinateur

Dans la section 4.5.1, différentes stratégies du coordinateur, qui pourraient être implémentées pour le modèle du contrôle semi-distribué proposé, ont été présentées. Parmi les points les plus intéressants à explorer dans ce contexte est l'implémentation d'autres stratégies de sélection et de tri des configurations globales à considérer dans un processus de coordination. Comme précisé dans la section 4.5.1.3, une implémentation possible de la stratégie du coordinateur peut être basée sur le Pareto si le but de celui-ci est d'optimiser un certain nombre de critères (qualité de service, performance, consommation, etc).

L'utilisation du Pareto pour la gestion de l'adaptation dynamique des systèmes-sur-puce a été étudié par certains travaux. Dans [155] [156], une approche pour l'adaptation dynamique d'un système MPSoC multi-applications est proposée. Cette approche se base sur le calcul de la frontière de Pareto de chaque application en prenant comme coût à optimiser la consommation d'énergie et comme contrainte le délai à ne pas dépasser. Chaque configuration d'une application dépend du nombre de processeurs et de la fréquence d'horloge à utiliser. Chaque configuration est caractérisée par un code spécifiant la parallélisation des tâches et les transferts de données selon le nombre de processeurs à utiliser. Les configurations se trouvant sur la frontière de Pareto pour chaque application sont données en entrée à un gestionnaire d'adaptation dynamique implémenté comme un service d'OS afin d'adapter le système selon les changements tels que le déclenchement ou l'arrêt d'une application donnée. Le fonctionnement de ce gestionnaire dynamique est de d'aller d'un point de Pareto à un autre pour chaque application selon le nombre de processeurs disponibles dans le système. Le déclenchement d'une nouvelle application peut être géré en diminuant le nombre de processeurs alloués aux applications déjà existantes. Il s'agit ici de prendre d'autres points de Pareto pour ces applications. Ces points pourraient demander d'augmenter les fréquences d'horloge pour ces applications afin de respecter leurs délais, ce qui augmenterait leurs consommations d'énergie. L'arrêt d'une application libère des ressources qui pourraient être utilisées par d'autres applications, ce qui leur permettrait de diminuer leurs fréquences tout en respectant leurs délais réduisant ainsi leurs consommations d'énergie. Dans [40] [35], le problème d'optimisation a été traité par un contrôle hiérarchique sur FPGA. Le Pareto est appliqué aux différentes configurations possibles d'une application. Une configuration est une combinaison d'implémentations logicielles et matérielles des tâches de l'application. Seules les configurations qui se trouvent sur la frontière du Pareto sont retenues pour être implémentées à l'exécution. Le contrôle de l'adaptation dynamique est divisé entre un contrôleur de l'application (LCM) et un contrôleur global (GCM). Selon les changements dynamiques des métriques spécifiques à l'application, le LCM propose une liste de configurations possibles au GCM. Le GCM n'a pas une vision des métriques prises en compte par le LCM. Il gère des métriques du système (QoS, consommation d'énergie, etc). Selon ces métriques, le GCM choisit parmi la liste fournie par le LCM, la meilleure configuration. Pour cette sélection, le GCM utilise différentes techniques telles que la régulation, l'amortissement de la reconfiguration, la distance de Hamming entre configurations et le vote Borda.

La mise en oeuvre d'une approche de coordination orientée par l'optimisation multi-objectifs est très intéressante étant donné que plusieurs problèmes de contrôle des systèmes reconfigurables de nos jours visent l'optimisation. Dans ce contexte, une approche basée sur le Pareto peut être utilisée. Comme précisé dans la section 4.5.1.3, le coordinateur peut utiliser une table *GC* qui ne contient que les configurations de la frontière de Pareto, comme c'est le cas dans les travaux cités ci-dessus. Il peut garder aussi des configurations sous-optimales à côté des optimales. Dans cette deuxième solution, les configurations sous-optimales peuvent être considérées dans un processus de coordination si les optimales n'ont pas eu d'acceptation par tous les contrôleurs concernés.

7.2.3 Application du modèle de contrôle proposé aux systèmes multi-FPGAs

Dans cette section, nous commençons par présenter les systèmes multi-FPGAs et l'application de la reconfiguration dynamique pour ces systèmes à travers certains travaux. L'application du modèle du contrôle proposé dans ce travail à ce type de systèmes est ensuite étudiée.

7.2.3.1 La reconfiguration dynamique des multi-FPGAs

Devant le nombre limité de ressources disponibles dans les FPGAs, les systèmes multi-FPGA sont devenus de plus en plus utilisés ces dernières années. Ces systèmes sont flexibles et ont un coût raisonnable [114]. L'avantage le plus important de ces systèmes est lié à la reconfiguration dynamique partielle. En effet, il est possible de reconfigurer un système entier, chaque FPGA séparément ou seulement une portion d'un FPGA, ce qui offre une grande flexibilité dans l'adaptation dynamique. De plus, la reconfiguration parallèle est possible avec de tels systèmes puisque plusieurs ports de configuration sont disponibles, ce qui permet de réduire le temps de reconfiguration par rapport à un système mono-FPGA.

Dans [113], un système multi-FPGA est proposé où des FPGAs esclaves sont attachés à un FPGA maître contenant un processeur. Ces FPGAs esclaves sont considérés comme ressources partiellement reconfigurables par le FPGA maître. La gestion de la reconfiguration est faite par un système d'exploitation embarqué dans le processeur du FPGA maître afin de cacher l'infrastructure physique du système aux utilisateurs. Ce système d'exploitation est basé sur Linux et a été étendu pour gérer la reconfiguration dynamique par de simples appels de fonctions. La reconfiguration de ce système est donc faite de manière centralisée. Dans [19], une architecture de contrôle distribué a été proposée pour la détection et correction des erreurs d'un système multi-FPGA. Dans cette architecture, la correction d'erreur est faite par la reconfiguration partielle des composants défaillants afin de retrouver leurs configurations nominales. La reconfiguration des régions d'un FPGA est faite par un contrôleur de reconfiguration d'un FPGA voisin. L'utilisation de cette solution au lieu d'accorder à chaque FPGA la tâche de gérer la correction de ces propres erreurs permet d'éviter d'avoir un point de défaillance du système. Les FPGAs sont connectés en mailles. Le rôle de chaque contrôleur de reconfiguration d'un FPGA est d'observer les signaux d'erreur provenant des régions du FPGA voisin, et de déclencher si nécessaire la reconfiguration de ces régions afin de corriger les erreurs. Le contrôleur lit aussi périodiquement la partie de mémoire de configuration lié au contrôleur du FPGA voisin afin de détecter et de corriger les éventuelles erreurs de ce contrôleur. L'architecture proposée a donc des avantages dans le domaine de détection des erreurs, mais ne permet pas de couvrir d'autres déclencheurs de reconfiguration tels que les entrées des utilisateurs ou les changements de l'environnement du système. Le contrôle hiérarchique pour les systèmes multi-FPGAs reconfigurables a été traité dans [4]. Cette structure du contrôle a été utilisé pour la correction des erreurs des FPGAs durant l'exécution. Ce problème de contrôle a été divisé en des problèmes locaux, où chaque contrôleur d'FPGA gère son auto-réparation. Dans le cas où un contrôleur ne parvient pas à résoudre le problème qu'il gère, on a recours à un FPGA maître qui fait la migration des tâches du FPGA défectueux vers un autre. La diffusion des données

d'observation à tous les FPGAs a été proposée dans [92] afin d'éviter d'avoir un noeud centralisé ou une communication "multi-hop" entre les FPGAs. Cette technique a été utilisée pour diffuser les données de puissance et de température collectées par les capteurs de chaque FPGAs à tous les autres FPGAs. Ces données sont utilisées par chaque FPGA pour contrôler le degré de parallélisme qu'il implémente en réduisant l'effort de calcul de certains éléments si les contraintes de puissance et de température ne sont pas respectées. Cette structure de contrôle convient donc lorsque les contrôleurs ont besoin des données d'observation collectées dans tout le système pour faire l'adaptation de son FPGA. Ici, le problème de contrôle traité par chaque contrôleur d'FPGA n'est pas très complexe puisqu'il dépend de deux données seulement (la puissance et la température). L'utilisation d'un seul contrôleur par FPGA peut engendrer la complexité de ce contrôleur pour des problèmes de contrôle plus complexes où plusieurs variables doivent être prises en compte pour l'adaptation de chaque FPGA, ce qui rend difficile la réutilisation d'un tel contrôleur et son adaptation à différents systèmes. Dans ce cas, la décentralisation du contrôle de chaque FPGA offre plus de flexibilité de conception et de possibilités de réutilisation et de scalabilité.

La plupart des travaux présentés ci-dessus utilise un seul contrôleur par FPGA, ce qui pourrait rendre chaque contrôleur dépendant du système implémenté dans son FPGA et représenter un obstacle devant la réutilisabilité. De plus, cette solution centralisée par FPGA pourrait engendrer de grands coûts de communication et des goulets d'étranglement. La distribution du contrôle d'un FPGA entre un ensemble de contrôleurs améliore la réutilisabilité et la scalabilité. Le modèle de contrôle semi-distribué proposé dans ce travail peut être donc une solution pour améliorer la flexibilité du contrôle et améliorer la productivité des concepteurs par rapport aux travaux présentés ci-dessus.

7.2.3.2 Le contrôle semi-distribué pour les systèmes multi-FPGAs

Le modèle de contrôle semi-distribué proposé dans ce travail peut être adapté aux systèmes multi-FPGAs. Le contrôle peut être implémenté de différentes manières. Une première implémentation est d'allouer un ensemble de contrôleurs distribués à chaque FPGA et d'utiliser un seul coordinateur pour tout le système implémenté dans un FPGA maître (2 niveaux de hiérarchie). Une autre implémentation est d'utiliser un coordinateur par FPGA et un coordinateur global si nécessaire (2 ou 3 niveaux de hiérarchie). Une troisième implémentation est d'utiliser des clusters de contrôleurs coordonnés par FPGA permettant ainsi d'avoir plusieurs niveaux de hiérarchie. Pour ces différentes implémentations, le principe reste toujours le même. Un coordinateur est consulté seulement si un contrôleur estime que la reconfiguration de l'élément qu'il contrôle est nécessaire. Si le coordinateur reçoit des acceptations des contrôleurs qu'ils coordonnent, il envoie une requête de reconfiguration à un autre coordinateur si jamais les contrôleurs de ce dernier sont concernés.

7.2.4 Application du modèle de contrôle proposé aux FPGAs 3D

Dans cette section, nous commençons par présenter les FPGAs 3D et l'application de la reconfiguration dynamique pour ces FPGAs à travers certains travaux. L'application

du modèle du contrôle proposé dans ce travail à ce type d'FPGA est ensuite étudiée.

7.2.4.1 Les FPGAs 3D

La miniaturisation des transistors en passant d'un noeud technologique à un autre en suivant la loi de Moore a permis une commutation plus rapide des transistors et donc une plus grande performance en offrant en même temps une plus grande densité des circuits. Cependant, avec des transistors de plus en plus petits, les fils qui les connectent sont devenus de plus en plus étroits, longs et proches les uns des autres [136]. Cela fait que le temps RC [1] (le temps que prend la tension pour effectuer 63% de la variation nécessaire pour passer de sa valeur initiale à sa valeur finale) de ces fils n'est plus négligeable devant le temps de propagation d'un signal, ce qui freine la croissance de la performance des circuits. La miniaturisation a entraîné aussi une dissipation plus importante de puissance qui dépend en grande partie de la longueur des fils [136]. Pour un FPGA, la surface occupée par les interconnexions est entre 70% et 80% de la surface totale [68]. Etant donné que la surface est un facteur très important dans la détermination du coût de fabrication, la réduction de la surface occupée par les interconnexions est nécessaire.

Une solution à tous ces problèmes est le passage aux circuits 3D qui permet d'utiliser des interconnexions courtes grâce à la communication verticale. Cela permet donc d'améliorer la performance et de réduire la dissipation de puissance. Les circuits 3D ont aussi l'avantage d'une grande densité pour une même surface qu'un circuit 2D. Un FPGA 3D est composé d'un ensemble de couches. Ces couches peuvent avoir la même structure (dans le cas des FPGAs homogènes), comme elles peuvent être spécialisées chacune pour un type de ressources (logique, mémoire, E/S, etc). Les FPGAs homogènes ont été étudiés par plusieurs travaux, parmi lesquels on cite [5] [66] [1]. Ces FPGAs sont un empilement d'FPGAs 2Ds contenant chacun des composants logiques, mémoire, etc. Les FPGAs hétérogènes ont été proposés dans quelques travaux [22] [68] [121]. Dans [22], les auteurs propose l'allocation d'une grande partie des connexions à une couche à part du FPGA. Ils montrent que cela permet d'augmenter la performance grâce à des connexions plus courtes. [68] montre l'amélioration de performance due à la séparation entre la logique et la mémoire en deux couches différentes. [121] étudie l'allocation des E/S à une couche séparée du FPGA 3D afin de diminuer la longueur des communications avec les E/S et améliorer donc la performance et diminuer la dissipation de puissance.

Il y a quelques FPGAs 3D disponibles sur le marché tels que les FPGAs fournis par Tezzaron Corp [123] et les FPGAs 3D de la famille Virtex-7 de Xilinx [36]. En fait, les FPGAs 3D de Virtex-7 ne sont pas de vrais 3D mais plutôt des FPGAs 2.5D parce qu'ils connectent les FPGA 2D horizontalement dans un même package. Cette technologie est appelée Stacked Silicon Interconnect (SSI) [36]. Le premier FPGA Xilinx utilisant cette technologie est Virtex-7 2000T [36] [74] qui intègre 4 puces FPGA dans le même pacakge. La communication entre FPGAs se fait à travers le "silicon interposer". Virtex-7 2000T offre 2 millions de cellules logiques, ce qui fait deux fois plus de portes logiques qu'un FPGA du même noeud technologique (28nm) [118]. L'avantage de SSI est l'efficacité de la communication entre les FPGAs qui composent le package. Le SSI permet d'atteindre

[1] http ://fr.wikipedia.org/wiki/Circuit_RC

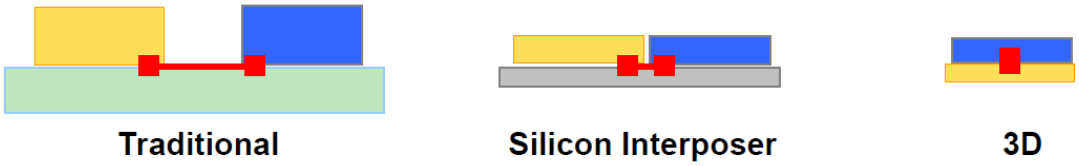


FIG. 7.1 – Evolution de la technologie des FPGAs Xilinx

100 fois la bande passante/watt de la connectivité entre les puces FPGAs avec 1/5 de latence en comparaison à une communication qui utilise des E/S. Cela permet aussi de réduire la consommation d'énergie. Le "silicon interposer" permet plus de 10000 connections à haut débit entre les puces FPGA, résolvant ainsi le problème lié au petit nombre d'I/O entre FPGAs, source de goulets d'étranglement. Après Virtex-7 2000T, la technologie SII a été utilisée pour construire un FPGA hétérogène. Alors que Virtex-7 2000T contient 4 puces d'FPGAs identiques, Virtex-7 HT580 combine les circuits FPGA et les circuits transceivers afin d'avoir une grande performance de communication ciblant les entreprises de communication utilisant des équipements très performants travaillant de 100 à 400 Gbps. Le fait de mettre les transceivers dans un circuit séparé permet d'optimiser leurs performances et de les isoler électriquement afin de diminuer les bruits [118]. Après la technologie 2.5D, Xilinx espère passer au 3D comme le montre la figure 7.1 [112].

7.2.4.2 La reconfiguration dynamique des FPGA 3D

Pour autant que nous sachions, la reconfiguration dynamique n'est pas encore supportée par des FPGAs 3D commercialisés. Elle n'est pas supportée non plus par les outils Xilinx pour les FPGA utilisant la technologie 2.5D [43]. La reconfiguration a été étudiée par quelques travaux traitant les FPGAs 3D. Dans [22], le FPGA est divisée en trois couches : mémoire (tout en dessous), interconnexion et logique. La couche mémoire permet la reconfiguration des deux autres couches. La couche logique contient les blocs logiques et une partie de l'interconnexion (les connexions courtes entre blocs logiques). Cette couche est organisée en clusters. La communication inter-clusters est assurée par la couche interconnexion qui est composée de boîtes de commutation. La plus petite unité de reconfiguration dans ce FPGA est le cluster puisque la reconfiguration d'un seul bloc du cluster nécessite aussi la reconfiguration d'une boîte de commutation (dans la couche interconnexion) qui demande un plus grand nombre de bits de reconfiguration. Vu la taille de l'information nécessaire pour la reconfiguration des boîtes de communication, leur reconfiguration en parallèle nécessite un bus de reconfiguration de taille très importante. Pour cette raison, dans un processus de reconfiguration, les boîtes de commutation sont reconfigurées l'une après l'autre alors que les blocs logiques sont reconfigurés en parallèle. Une approche ressemblante qui est basée sur trois couches a été présentée dans [68]. Dans [69], les auteurs proposent l'utilisation de deux couches mémoires à côté de la couche logique et la couche interconnexion. La première couche mémoire permet la programmation des blocs logiques. Cette couche est placée sur la couche logique. La couche interconnexion est placée en-dessus. Enfin, une deuxième

couche mémoire est placée tout en-dessus pour programmer les commutations dans la couche interconnexion. L'utilisation de deux couches mémoires offre une meilleure connectivité verticale et diminue le nombre de cases mémoires requises par couche par rapport au travail présenté dans [68].

Bien que la reconfiguration des FPGAs 3D ne soit pas étudiée et mise en oeuvre par un grand nombre de travaux, on peut envisager dans le futur l'application du modèle de contrôle proposé dans ce travail à ce type émergent d'FPGAs. Cette application serait bénéfique aux systèmes basés sur ce type d'FPGA vu la flexibilité de ce modèle de contrôle lui permettant de s'adapter à différentes architectures du système cible par rapport à une solution centralisée, que ce soit en termes de réutilisabilité et de scalabilité ou en termes d'efficacité de la communication.

7.2.4.3 Le contrôle semi-distribué pour les FPGAs 3D

L'adaptation du modèle de contrôle semi-distribué aux FPGAs 3D diffère entre un FPGA homogène et un FPGA hétérogène. Pour un FPGA homogène, où les couches ont la même structure, un ou plusieurs contrôleurs peuvent être implémentés par couche et coordonnés par un coordinateur ou plusieurs coordinateurs selon le besoin. Les coordinateurs peuvent communiquer si les configurations qu'ils gèrent ont un impact sur les contraintes ou objectifs globaux du système. Pour un FPGA hétérogène, où chaque couche est dédiée à un type de blocs, le contrôle peut être implémenté dans les couches logiques en utilisant différentes hiérarchies selon le besoin tout comme pour les FPGAs homogènes.

7.2.5 Intégration d'autres types de reconfiguration tels que le changement de contexte

Parmi les avantages de la RDP est qu'elle permet de remplacer une IP implémentant une tâche de l'application par une autre permettant ainsi une meilleure exploitation des ressources limitées d'un système. Dans certains cas, cette reconfiguration doit être accompagnée d'une sauvegarde du contexte de l'ancienne IP et le chargement de celui de la nouvelle IP. Cela permet de reprendre l'exécution de l'ancienne IP là où elle s'est arrêtée si elle est chargée de nouveau. Cet aspect a été traité dans le cadre du projet FAMOUS [135] en proposant une implémentation matérielle de ce service à travers des membranes pour la gestion de la RDP et du contexte des composants reconfigurables sur FPGA. Chaque membrane est allouée à un composant reconfigurable du système. Cette approche est proche de la notre dans la mesure où elle permet l'auto-adaptation des composants reconfigurables. Cependant, elle n'intègre pas des mécanismes d'observation et de prise de décision de reconfiguration. Pour cela, la combinaison de cette approche avec la notre sera bénéfique afin d'avoir un modèle de contrôle gérant à la fois l'observation, la décision, la RDP et le changement de contexte. Cette intégration en cours d'étude avec notre partenaire FAMOUS sera basée sur la fusion des concepts de membranes avec ceux des modules de reconfiguration de notre modèle de contrôle afin de permettre aux modules de décision de lancer les reconfigurations à travers les membranes et de gérer en même temps le contexte des régions contrôlées.

Bibliographie

[1] C. Ababei, Y. Feng, B. Goplen, H. Mogal, T. Zhang, K. Bazargan, and S. Sapatnekar. Placement and routing in 3d integrated circuits. *Design & Test of Computers, IEEE*, 22(6) :520–531, 2005. 217

[2] Acceleo. *Acceleo - transforming models into code*. http ://www.eclipse.org/acceleo/. 58, 127

[3] A.Deshmukh, F.Ponci, A.Monti, L.Cristaldi, R.Ottoboni, M.Riva, and M.Lazzaroni. Multi agent systems : an example of dynamic reconfiguration. In *Instrumentation and Measurement Technology Conference*, Sorrento, ITALY, 2006. 40

[4] A. Akoglu, A. Sreeramareddy, and J. G. Josiah. Fpga based distributed self healing architecture for reusable systems. *Cluster Computing*, 12 :269–284, September 2009. 3, 215

[5] A. Alexander, J. Cohoon, J. Colflesh, J. Karro, E. Peters, and G. Robins. Placement and routing for three-dimensional fpgas. In *Fourth Canadian Workshop on Field-Programmable Devices*, volume 71. Citeseer, 1996. 217

[6] Altera. Increasing design functionality with partial and dynamic reconfiguration in 28-nm fpgas. Technical Report WP-01137-1.0, 2010. 19

[7] K. Altisen, A. Clodic, F. Maraninchi, and E. Rutten. Using controller-synthesis techniques to build property-enforcing layers. In *Programming Languages and Systems*, pages 174–188. Springer, 2003. 46, 47

[8] S. Andre. Mda (model driven architecture) principes et états de l'art. Technical report, Technical report, CNAM, 05 Novembre 2004. 126 BIBLIOGRAPHIE, 2004. 49

[9] D. Andrews, R. Sass, E. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot, and E. Komp. The case for high level programming models for reconfigurable computers. *Engineering of Reconfigurabe Systems and Algorithms (ERSA)*, 2006. 35

[10] V. Aranega, A. Etien, and S. Mosser. Using feature model to build model transformation chains. In *Proceedings of the 15th international conference on Model Driven Engineering Languages and Systems*, MODELS'12, pages 562–578, Berlin, Heidelberg, 2012. Springer-Verlag. 127

[11] R. B. Atitallah. *Modèles et simulation des systèmes sur puce multiprocesseurs : estimation des performances et de la consommation d'énergie*. PhD thesis, PhD thesis, Université des Sciences et Technologie de Lille, 2008. 58

[12] S. Bansal. 3d-ic design : The challenges of 2.5d versus 3d. 9/14/2011. 17, 18

[13] S. Bayar and A. Yurdakul. Dynamic partial self-reconfiguration on spartan-iii fpgas via a parallel configuration access port (pcap). In *2nd HiPEAC workshop on Reconfigurable Computing*, HiPEAC 08, 2008. 20

[14] L. Besnard, H. Marchand, and E. Rutten. The sigali tool box environment. In *8th International Workshop on Discrete Event Systems*, pages 465–466. IEEE, 2006. 45

[15] J. Bézivin. On the unification power of models. *Software & Systems Modeling*, 4(2) :171–188, 2005. 48

[16] B. Blodget, S. McMillan, and P. Lysaght. A lightweight approach for embedded reconfiguration of fpgas. In *Design, Automation & Test in Europe*, DATE'03, 2003. 20

[17] B. Blodget, S. McMillan, and P. Lysaght. A lightweight approach for embedded reconfiguration of fpgas. In *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*, DATE '03, Washington, DC, USA, 2003. 33

[18] C. Bobda, M. Majer, D. Koch, A. Ahmadinia, and J. Teich. A dynamic noc approach for communication in reconfigurable devices. In J. Becker, M. Platzner, and S. Vernalde, editors, *Field Programmable Logic and Application*, volume 3203 of *Lecture Notes in Computer Science*, pages 1032–1036. Springer Berlin Heidelberg, 2004. 22

[19] C. Bolchini, L. Fossati, D. Codinachs, A. Miele, and C. Sandionigi. A reliable reconfiguration controller for fault-tolerant embedded systems on multi-fpga platforms. In *IEEE 25th International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT)*, pages 191–199, 2010. 215

[20] E. Canto, F. Fons, and M. Lopez. Reconfigurable opb coprocessors for a microblaze self-reconfigurable soc mapped on spartan-3 fpgas. In *32nd Annual Conference on IEEE Industrial Electronics (IECON)*, pages 4940–4944, 2006. 22

[21] E. Carvalho, N. Calazans, F. Moraes, and D. Mesquita. Reconfiguration control for dynamically reconfigurable systems. In *DCI'04*, pages 405–410, 2004. 33

[22] S. Chiricescu, M. Leeser, and M. Vai. Design and analysis of a dynamically reconfigurable three-dimensional fpga. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(1) :186–196, 2001. 217, 218

[23] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2) :244–263, 1986. 44

[24] C. Claus, R. Ahmed, F. Altenried, and W. Stechele. Towards rapid dynamic partial reconfigurationin video-based driver assistance systems. In *6th Internation Symposium of Reconfigurable Computing : Architectures, Tools and Applications*, pages 55–67, 2010. 20, 33

[25] C. Claus, F. H. Müller, J. Zeppenfeld, and W. Stechele. A new framework to accelerate virtex-ii pro dynamic partial self-reconfiguration. In *Parallel and Distributed Processing Symposium, IPDPS'07*, pages 1–7, 2007. 20

[26] C. Claus, J. Zeppenfeld, F. Muller, and W. Stechele. Using partial-run-time reconfigurable hardware to accelerate video processing in driver assistance system. In *Design, Automation Test in Europe Conference Exhibition, DATE '07*, pages 1–6, 2007. 22

[27] C. Claus, B. Zhang, W. Stechele, L. Braun, M. Hubner, and J. Becker. A multi-platform controller allowing for maximum dynamic partial reconfiguration throughput. In *Proceedings of the International Conference on Field Programmable Logic and Applications*, pages 535–539, 2008. 20

[28] V.-. F. Configuration. Ug360 (v3.5). *Xilinx Inc., September 11*, 2012. 20

[29] A. Cuoccio, P. R. Grassi, V. Rana, M. D. Santambrogio, and D. Sciuto. A generation flow for self-reconfiguration controllers customization. *IEEE International Workshop on Electronic Design, Test and Applications*, 0 :279–284, 2008. 20

[30] K. Danne. Operating systems for fpga based computers and their memory management. In *ARCS Organic and Pervasive Computing, Workshop Proceedings, volume P-41 of GI-Edition Lecture*, 2004. 35

[31] G. Delaval, H. Marchand, and E. Rutten. Contracts for modular discrete controller synthesis. *ACM Sigplan Notices*, 45 :57–66, 2010. 46, 47, 90, 166

[32] L. Devaux, S. B. Sassi, S. Pillement, D. Chillet, and D. Demigny. Flexible interconnection network for dynamically and partially reconfigurable architectures. *International Journal of Reconfigurable Computing*, 2010 :6, 2010. 35

[33] M. Dias, , and A. Stentz. Enhanced negotiation and opportunistic optimization for marketbased multirobot coordination. Technical Report CMU-RI-TR-02-18, Robotics Institute, Carnegie Mellon University, Pittsburgh (Pennsylvania), 2002. 38

[34] M. Dias, , and A. Stentz. Traderbots : A market-based approach for resource, role, and task allocation in multirobot coordination. 2003. 38

[35] J.-P. Diguet, Y. Eustache, and G. Gogniat. Closed-loop based self-adaptive hardware/software-embedded systems : Design methodology and smart cam case study. *ACM Transactions on Embedded Computing Systems*, 10(3) :38 :1–38 :28, May 2011. 38, 214

[36] P. Dorsey. Xilinx stacked silicon interconnect technology delivers breakthrough fpga capacity, bandwidth, and power efficiency. Technical Report Xilinx White paper : Vertex-7 FPGAs, 2011. 3, 217, 218

[37] V. G. Douros, G. C. Polyzos, and S. Toumpis. Negotiation-based distributed power control in wireless networks with autonomous nodes. In *IEEE 73rd Vehicular Technology Conference*, (VTC Spring), pages 1 –5, 2011. 37, 40, 42

[38] Y. Du, C. Jiang, and M. Zhou. A petri net-based model for verification of obligations and accountability in cooperative systems. *IEEE Transactions on Systems, Man and Cybernetics, Part A : Systems and Humans*, 39(2) :299–308, 2009. 43

[39] Eclipse. *Papyrus*. http ://www.eclipse.org/modeling/mdt/papyrus/, 2012. 53, 58

[40] Y. Eustache and J.-P. Diguet. Specification and os-based implementation of self-adaptive, hardware/software embedded systems. In *Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, pages 67–72. ACM, 2008. 34, 35, 38, 64, 214

[41] M. Faugere, T. Bourbeau, R. D. Simone, and S. Gerard. Marte : Also an uml profile for modeling aadl applications. In *12th IEEE International Conference on Engineering Complex Computer Systems*, pages 359–364. IEEE, 2007. 51

[42] A. Gamatie, S. L. Beux, E. Piel, R. B. Atitallah, A. Etien, P. Marquet, and J.-L. Dekeyser. A model driven design framework for massively parallel embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 10, 2011. 8, 46, 47, 61

[43] P. R. U. Guide. Ug702 (v14.3). *Xilinx Inc., October 16*, 2012. 15, 19, 26, 184, 193, 196, 218

[44] S. Guillet, F. de Lamotte, N. L. Griguer, E. Rutten, G. Gogniat, and J.-P. Diguet. Designing formal reconfiguration control using uml/marte. In *7th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, pages 1–8. IEEE, 2012. 46, 47, 62, 64

[45] S. Guillet, F. de Lamotte, E. Rutten, G. Gogniat, and J.-P. Diguet. Modeling and formal control of partial dynamic reconfiguration. *International Conference on Reconfigurable Computing and FPGAs*, 0 :31–36, 2010. 166

[46] H. Yu. *A MARTE based reactive model for data-parallel intensive processing : Transformation toward the synchronous model*. PhD thesis, USTL, 2008. V, 46, 47, 58, 60, 61, 63, 72, 73, 74

[47] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. In *Proceedings of the IEEE*, pages 1305–1320, 1991. 43

[48] N. Harb, S. Niar, J. Khan, and M. A. R. Saghir. A reconfigurable platform architecture for an automotive multiple-target tracking system. *SIGBED Rev.*, 6(3) :13 :1–13 :5, Oct. 2009. 33

[49] M. Held, P. Wolfe, and H. P. Crowder. Validation of subgradient optimization. *Mathematical programming*, 6(1) :62–88, 1974. 42

[50] E. Horta and J. W. Lockwood. Parbit : A tool to transform bifitles to implement partial reconguration of field programmable gate arrays (fpgas). Technical report wucs-01-13, July 2001. 20

[51] J. Huang, M. Parris, J. Lee, and R. F. Demara. Scalable fpga-based architecture for dct computation using dynamic partial reconfiguration. *ACM Transactions on Embedded Computing Systems (TECS)*, 9(1) :9, 2009. 33

[52] K. Huang, S. Srivastava, and D. Cartes. Decentralized reconfiguration for power systems using multi agent system. In *Annual IEEE Systems Conference Waikiki Beach*, Waikiki Beach, Honolulu, Hawaii, USA, 2007. 37, 40

[53] M. Hubner, K. Paulsson, and J. Becker. Parallel and flexible multiprocessor system-on-chip for adaptive automotive applications based on xilinx microblaze soft-cores. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, pages 149a–149a, 2005. 34

[54] M. Hübner, C. Schuck, and J. Becker. Elementary block based 2-dimensional dynamic and partial reconfiguration for virtex-ii fpgas. In *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, pages 196–196, 2006. 21

[55] M. Hübner, C. Schuck, M. Kuhnle, and J. Becker. New 2-dimensional partial dynamic reconfiguration techniques for real-time adaptive microelectronic circuits. In *Proceedings of the IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures*, ISVLSI '06, pages 97–, 2006. 21

[56] Imran Rafiq Quadri. *MARTE based model driven design methodology for targeting dynamically reconfigurable FPGA based SoCs*. PhD thesis, USTL, 2010. V, 46, 47, 58, 61, 63, 64, 76, 77

[57] M. Iqbal and U. Awan. Run-time reconfigurable instruction set processor design : Rt-risp. In *2nd International Conference on Computer, Control and Communication, IC4*, pages 1–6, 2009. 21

[58] Z. E. A. A. Ismaili and A. Moussa. Self-partial and dynamic reconfiguration implementation for aes using fpga. *arXiv preprint arXiv :0909.2369*, 2009. 33

[59] N. R. Jennings. On agent-based software engineering. *Artificial intelligence*, 117(2) :277–296, 2000. 40

[60] M. Khalgui and H. M. Hanisch. Reconfiguration protocol for multi-agent control software architectures. *IEEE Transactions on Systems, Man, and Cybernetics, Part C : Applications and Reviews*, 41, 2011. 37, 40

[61] R. Koch, T. Pionteck, C. Albrecht, and E. Maehle. A lightweight framework for runtime reconfigurable system prototyping. *IEEE International Workshop on Rapid System Prototyping*, 0 :61–64, 2007. 33

[62] O. Labbani. *Modélisation à haut niveau du contrôle dans des applications de traitement systématique à parallélisme massif*. PhD thesis, USTL, 2006. V, 46, 47, 59, 60, 63, 68, 69, 70, 71

[63] S. Lamonnier, M. Thoris, and M. Ambielle. Accelerate partial reconfiguration with a 100% hardware solution. *Xcell Journal*, (79), 2012. 33, 151, 198

[64] B. M. LaPedus. Chip makers intensify race in 3d dram market. 15/10/2011. 17

[65] S. Lee, S. Yoo, and K. Choi. Reconfigurable soc design with hierarchical fsm and synchronous dataflow model. In *Proceedings of the tenth international symposium on Hardware/software codesign*, CODES '02, pages 199–204. ACM, 2002. 43

[66] M. Leeser, W. Meleis, M. Vai, S. Chiricescu, W. Xu, and P. Zavracky. Rothko : A three-dimensional fpga. *Design & Test of Computers, IEEE*, 15(1) :16–23, 1998. 217

[67] R. Lewis and R. Lewis. Modelling distributed control systems using iec 61499 : Applying function blocks to distributed systems. Institution of Electrical Engineers, 2001. 43

[68] M. Lin, A. El Gamal, Y. Lu, and S. Wong. Performance benefits of monolithically stacked 3-d fpga. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2) :216–229, 2007. 3, 217, 218, 220

[69] M. Lin and A. E. Gamal. A routing fabric for monolithically stacked 3d-fpga. In *Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*, pages 3–12, 2007. 218

[70] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgford. Invited paper : Enhanced architectures, design methodologies and cad tools for dynamic reconfiguration of xilinx fpgas. In *Field Programmable Logic and Applications, FPL'06*, pages 1–6, 2006. 15

[71] D. G. M., Hubner, V. Schatz, and J. Becker. Runtime adaptive multi-processor system-on-chip : Rampsoc. In *IEEE International Symposium on Parallel and Distributed Processing, IPDPS*, pages 1–7, 2008. 23

[72] I. Mansouri, F. Clermidy, P. Benoit, and L. Torres. A run-time distributed cooperative approach to optimize power consumption in mpsocs. In *IEEE International SOC Conference (SOCC)*, pages 25–30, 2010. 42

[73] F. Maraninchi and Y. Rémond. Mode-automata : a new domain-specific construct for the development of safe critical systems. *Science of Computer Programming*, 46(3) :219–254, Mar. 2003. 8, 43, 44, 80, 83, 84

[74] C. Maxfield. New xilinx virtex-7 2000t fpga provides equivalent of 20 million asic gates. 10/25/2011. 18, 218

[75] C. Maxfield. Fpga architectures from 'a' to 'z' : Part 1. *EE Times Design*, 2006. 15

[76] C. Maxfield. 2d vs. 2.5d vs. 3d ics 101. 4/8/2012. 17

[77] C. Maxfield. The state of the art in 3d ic technologies. 4/8/2012. 17

[78] M. Maxfield. The world before 3d ics. 7/3/2012. 15, 16

[79] S. McMillan and S. Guccione. Partial run-time reconfiguration using jrtr. In *Proceedings of the The Roadmap to Reconfigurable Computing, 10th International Workshop on Field-Programmable Logic and Applications*, FPL '00, pages 352–360, 2000. 19

[80] T. Mens and P. V. Gorp. A taxonomy of model transformation. *Proceedings of the International Workshop on Graph and Model Transformation, GraMoT 2005*, 152 :125 –142, 2006. 49

[81] P. Merino, J. C. Lopez, and M. F. Jaco. A hardware operating system for dynamic reconfiguration of fpgas. In *Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications, From FPGAs to Computing Paradigm*, pages 431–435, London, UK, 1998. 35

[82] L. Ming, W. Kuehn, L. Zhonghai, and A. Jantsch. Run-time partial reconfiguration speed investigation and architectural design space exploration. In *Proceedings of the International Conference on Field Programmable Logic and Applications*, pages 498–502, 2009. 20, 33

[83] A. Montone, V. Rana, M. Santambrogio, and D. Sciuto. Harpe : A harvard-based processing element tailored for partial dynamic reconfigurable architectures. In *IEEE International Symposium on Parallel and Distributed Processing, IPDPS*, pages 1–8, 2008. 21

[84] V. J. Mooney III and D. M. Blough. A hardware-software real-time operating system framework for socs. *Design & Test of Computers, IEEE*, 19(6) :44–51, 2002. 35

[85] G. Moore. No exponential is forever : but "forever" can be delayed ! [semiconductor industry]. In *IEEE International Solid-State Circuits Conference, ISSCC*, pages 20 – 23 vol.1, 2003. 3

[86] S. Muhlbach and A. Koch. A dynamically reconfigured multi-fpga network platform for high-speed malware collection. *International Journal of Reconfigurable Computing*, 2012, 2012. 3

[87] F. Muller, J. L. Rhun, F. Lemonnier, B. Miramond, and L. Devaux. A flexible operating system for dynamic applications. *Xcell journal*, (73) :30–34, 2010. 35, 64

[88] S. Narayanan, D. Chillet, S. Pillement, and I. Sourdis. Hardware os communication service and dynamic memory management for rsocs. In *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pages 117–122, 2011. 35

[89] P. S. B. Nascimento, P. R. M. Maciel, M. E. Lima, R. E. Sant'ana, and A. G. S. Filho. A partial reconfigurable architecture for controllers based on petri nets. In *Proceedings of the 17th symposium on Integrated circuits and system design*, SBCCI '04, pages 16–21, New York, NY, USA, 2004. ACM. 33

[90] J. F. Nash. Equilibrium points in n-person games. In *Proceedings of the National Academy of Sciences of the United States of America*, 1950. 41

[91] B. T. J. Ng, L. Q. Zhuang, J. B. Zhang, and Y. Z. Zhao. Consortium based negotiation for distributed multi-equipment execution control. In *IEEE International Symposium on Industrial Electronics*, volume 1, pages 757 – 762, 2004. 40

[92] X. Y. Niu, K. H. Tsoi, and W. Luk. Reconfiguring distributed applications in fpga accelerated cluster with wireless networking. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 545 –550, 2011. 3, 216

[93] R. Olfati-Saber, J. Fax, and R. Murray. Consensus and cooperation in networked multi-agent systems. *Proceedings of the IEEE*, 95(1) :215–233, 2007. 42

[94] OMG. Object modeling group. http://www.omg.org/. 47

[95] OMG. Uml profile for schedulability, performance, and time specification, version 1.1, 2005. http://www.omg.org/spec/SPTP/1.1/. 50

[96] OMG. Uml profile for system on a chip (soc) available specification, version 1.0.1, 2006. http://www.omg.org/spec/SoCP/1.0.1/. 50

[97] OMG. Meta object facility (mof) 2.0 query/view/transformation specification, version 1.1, 2011. http://www.omg.org/spec/QVT/1.1/. 50

[98] OMG. Omg unified modeling language (omg uml) infrastructure version 2.4.1, 2011. http://www.omg.org/spec/UML/2.4.1/. 50, 123, 238

[99] OMG. Uml profile for modeling and analysis of real-time and embedded systems, version 1.1, 2011. http://www.omg.org/spec/MARTE/1.1/. 47, 52, 53, 115

[100] OMG. Omg systems modeling language (omg sysml), version 1.3, 2012. http://www.omg.org/spec/SysML/1.3. 50

[101] M. J. Osborne and A. Rubinstein. *A course in game theory*. MIT Press, 1994. 41

[102] O. I. Partnership. *OCP-IP*. http ://www.ocpip.org. 129

[103] O. I. Partnership. Open core protocol specification. Technical Report Release 2.0, 2008. 129, 130

[104] K. Paulsson, M. Hübner, and J. Becker. On-line optimization of fpga power-dissipation by exploiting run-time adaption of communication primitives. In *Proceedings of the 19th annual symposium on Integrated circuits and systems design*, SBCCI '06, pages 173–178, New York, NY, USA, 2006. ACM. 19, 22

[105] K. Paulsson, M. Hübner, and J. Becker. Dynamic power optimization by exploiting self-reconfiguration in xilinx spartan 3-based systems. *Microprocessors and Microsystems*, 33(1) :46–52, Feb. 2009. 19, 23

[106] K. Paulsson, M. Hubner, J. Becker, J.-M. Philippe, and C. Gamrat. On-line routing of reconfigurable functions for future self-adaptive systems - investigations within the aether project. In *International Conference on Field Programmable Logic and Applications*, FPL 2007, 2007. 36, 37, 39

[107] J.-M. Philippe, B. Tain, and C. Gamrat. A self-reconfigurable fpga-based platform for prototyping future pervasive systems. In *Evolvable Systems : From Biology to Hardware*, volume 6274 of *Lecture Notes in Computer Science*, pages 262–273. Springer Berlin / Heidelberg, 2010. 36, 37, 39, 64

[108] D. Puschini, F. Clermidy, P. Benoit, and G. S. L. Torres. Temperature-aware distributed run-time optimization on mp-soc using game theory. In *IEEE Computer society annual Symposium on VLSI*, ISVLSI '08, Montpellier, France, 2008. 37, 40, 41, 42

[109] D. Puschini, F. Clermidy, P. Benoit, and G. S. L. Torres. Adaptive energy-aware latency-constrained dvfs policy for mpsoc. In *IEEE International SOC Conference*, SOCC 2009, Belfast, Northern Ireland, UK, 2009. 37, 40, 41, 42

[110] J.-P. Queille and J. Sifakis. A temporal logic to deal with fairness in transition systems. In *23rd Annual Symposium on Foundations of Computer Science, SFCS'08*, pages 217–225. IEEE, 1982. 44

[111] QVTO. *Eclipse Model To Model (M2M) Project*. http ://m2m.eclipse.org/. 50, 58, 124

[112] S. Ramalingam. 3d ic development and key role of supply chain collaboration. In *IEEE Components, Packaging, and Manufacturing Technology Conference*, 2011. 218

[113] V. Rana, M. Santambrogio, D. Sciuto, B. Kettelhoit, M. Koester, M. Porrmann, and U. Ruckert. Partial dynamic reconfiguration in a multi-fpga clustered architecture based on linux. In *IEEE International Parallel and Distributed Processing Symposium, IPDPS*, pages 1 –8, 2007. 3, 215

[114] F. Redaelli, M. Santambrogio, D. Sciuto, and S. O. Memik. Reconfiguration aware task scheduling for multi-fpga systems. *Reconfigurable Computing*, page 57, 2010. 215

[115] E. Riccobene, P. Scandurra, A. Rosti, and S. Bocchio. A model-driven design environment for embedded systems. In *Proceedings of the 43rd annual Design Automation Conference*, pages 915–918. ACM, 2006. 50

[116] W. Rodrigues. *A Methodology to Develop High Performance Applications on GPGPU Architectures : Application to Simulation of Electrical Machines*. PhD thesis, PhD thesis, Université des Sciences et Technologie de Lille, 2012. 58

[117] M. D. Santambrogio, V. Rana, and D. Sciuto. Operating system support for online partial dynamic reconfiguration management. In *International Conference on Field Programmable Logic and Applications*, 2008. 34, 35

[118] M. Santarini. Stacked & loaded : Xilinx ssi, 28-gbps i/o yield amazing fpgas. (74), 2011. 218

[119] C. Schuck, B. Haetzer, and J. Becker. An interface for a decentralized 2d reconfiguration on xilinx virtex-fpgas for organic computing. *International Journal of Reconfigurable Computing*, 2009 :7 :3–7 :3, January 2009. 36, 37, 39, 64

[120] S. Sendall and W. Kozaczynski. Model transformation : The heart and soul of model-driven software development. *IEEE Software*, 20(5) :42 –45, 2003. 49

[121] H. Sidiropoulos, K. Siozios, and D. Soudris. A framework for architecture-level exploration of 3-d fpga platforms. *Integrated Circuit and System Design. Power and Timing Modeling, Optimization, and Simulation*, pages 298–307, 2011. 217

[122] H. Tan, R. F. DeMara, A. J. Thakkar, A. Ejnioui, and J. Sattler. Complexity and performance evaluation of two partial reconfiguration interfaces on fpgas : A case study. In *ERSA'06*, pages 253–256, 2006. 20

[123] Tezzaron. 3d fpga demo/test, 2012. http://www.tezzaron.com/about/PhotoAlbum/Products/3D_FPGA.html. 217

[124] S. Toscher, R. Kasper, and T. Reinemann. Implementation of a reconfigurable hard real-time control system for mechatronic and automotive applications. In *20th International Parallel and Distributed Processing Symposium, IPDPS*, pages 4 pp.–, 2006. 36, 37

[125] S. Toscher, T. Reinemann, and R. Kasper. An adaptive fpga-based mechatronic control system supporting partial reconfiguration of controller functionalities. In *First NASA/ESA Conference on Adaptive Hardware and Systems, AHS*, pages 225–228, 2006. 36, 37

[126] C. Trabelsi, R. B. Atitallah, S. Meftali, J.-L. Dekeyser, and A. Jemai. A model-driven approach for hybrid power estimation in embedded systems design. *Eurasip Journal on Embedded Systems*, 2011. 58, 82

[127] C. Trabelsi, S. Meftali, R. B. Atitallah, A. Jemai, J.-L. Dekeyser, and S. Niar. An mde approach for energy consumption estimation in mpsoc design. In *2nd Workshop on Rapid Simulation and Performance Evaluation : Methods and Tools*, Pisa, Italy, 2010. 58, 82

[128] C. Trabelsi, S. Meftali, and J.-L. Dekeyser. Distributed control for reconfigurable fpga systems : a high-level design approach. In *7th International Workshop on Reconfigurable Communication-centric Systems-on-Chip*, 2012. 6, 135

[129] C. Trabelsi, S. Meftali, and J.-L. Dekeyser. Semi-distributed control for fpga-based reconfigurable systems. In *15th Euromicro Conference on Digital System Design*, 2012. 6, 64, 79

[130] C. Trabelsi, S. Meftali, and J.-L. Dekeyser. Decentralized control for dynamically reconfigurable fpga systems. *Microprocessors and Microsystems*, sera publié en 2013. 6, 64

[131] A. Tumeo, M. Monchiero, G. Palermo, F. Ferrandi, and D. Sciuto. A self-reconfigurable implementation of the jpeg encoder. In *IEEE International Conference on Application -specific Systems, Architectures and Processors (ASAP)*, pages 24–29, 2007. 22, 33

[132] J. Vidal, F. de Lamotte, G. Gogniat, J.-P. Diguet, and P. Soulard. Uml design for dynamically reconfigurable multiprocessor embedded systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, 2010. 33, 62, 64

[133] K. Vipin and S. Fahmy. A high speed open source controller for fpga partial reconfiguration. In *International Conference on Field-Programmable Technology (FPT)*, pages 61–66, 2012. 22

[134] V. Vyatkin, M. Hirsch, and H.-M. Hanisch. Systematic design and implementation of distributed controllers in industrial automation. In *IEEE Conference on Emerging Technologies and Factory Automation*, ETFA '06, pages 633 –640, 2006. 36

[135] P. Wattebled, J.-P. Diguet, and J.-L. Dekeyser. Membrane-based design and management methodology for parallel dynamically reconfigurable embedded systems. In *7th International Workshop on Reconfigurable Communication-centric Systems-on-Chip*, 2012. 220

[136] N. Weste and D. Harris. *Principles of CMOS VLSI design : A circuits and systems perspectives*. Addison-Wesley, 2004. 217

[137] G. Wigley, D. Kearney, and M. Jasiunas. Reconfigme : a detailed implementation of an operating system for reconfigurable computing. In *20th International Parallel and Distributed Processing Symposium, IPDPS*, pages 8 pp.–, 2006. 34, 35

[138] Xilinx. *CoreConnect Architecture*. http://www.xilinx.com/ipcenter/processor_central/coreconnect/coreconnect.htm. 20, 22

[139] Xilinx. *FSL-V20*. http://www.xilinx.com/products/ipcenter/FSL.htm. 22

[140] Xilinx. Virtex 2.5 v field programmable gate arrays product specification. Technical Report DS003-1 (v2.5), 2001. 3

[141] Xilinx. *Two Flows for Partial Reconfiguration : Module Based or Difference Based*, 2003. 19

[142] Xilinx. Opb hwicap product specification. Technical Report DS 280 (v1.3), 2004. 20, 21

[143] Xilinx. *Two Flows for Partial Reconfiguration : Module Based or Difference Based*, 2004. 19, 26

[144] Xilinx. Synthesis and simulation design guide. Technical Report UG626 (v 11.4), 2009. 24

[145] Xilinx. Virtex-5 family overview product specification. Technical Report DS100 (v5.0), 2009. 3

[146] Xilinx. *Early Access Partial Reconfigurable Flow*, 2010. 19

[147] Xilinx. Virtex-4 family overview product specification. Technical Report DS112 (v3.1), 2010. 3

[148] Xilinx. Logicore ip xps hwicap (v5.01a) product specification. Technical Report DS586, 2011. 20

[149] Xilinx. Virtex-ii pro and virtex-ii pro x platform fpgas : Complete data sheet product specification. Technical Report DS083 (v5.0), 2011. 3

[150] Xilinx. 7 series fpgas overview advance product specification. Technical Report DS180 (v1.13), 2012. 3, 180, 207

[151] Xilinx. Logicore ip axi bus functional models (v3.00.a) product specification. Technical Report DS824, 2012. 21, 22

[152] Xilinx. Logicore ip axi hwicap (v2.02.a) product specification. Technical Report DS817, 2012. 20, 21

[153] Xilinx. Virtex-5 fpga configuration user guide. Technical Report UG191 (v3.11), 2012. 20

[154] Xilinx. Virtex-6 family overview product specification. Technical Report DS150 (v2.4), 2012. 3

[155] C. Ykman-Couvreur, E. Brockmeyer, V. Nollet, T. Marescaux, F. Catthoor, and H. Corporaal. Design-time application exploration for mp-soc customized run-time management. In *Proceedings of International Symposium on System-on-Chip*, pages 66–69, 2005. 214

[156] C. Ykman-Couvreur, V. Nollet, T. Marescaux, E. Brockmeyer, F. Catthoor, and H. Corporaal. Pareto-based application specification for mp-soc customized run-time management. In *International Conference on Embedded Computer Systems : Architectures, Modeling and Simulation, IC-SAMOS*, pages 78–84, 2006. 214

[157] C.-H. Yu and R. Nagpal. Self-adapting modular robotics : A generalized distributed consensus framework. In *IEEE International Conference on Robotics and Automation, ICRA'09*, pages 1881–1888, 2009. 40

[158] C.-H. Yu, J. Werfel, and R. Nagpal. Collective decision-making in multi-agent systems by implicit leadership. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems : volume 3*, AAMAS '10, pages 1189–1196, 2010. 37, 40, 42

[159] H. Yu, A. Gamatié, É. Rutten, and J.-L. Dekeyser. Safe design of high-performance embedded systems in an mde framework. *Innovations in Systems and Software Engineering*, 4(3) :215–222, 2008. 45

Annexe A

L'utilisation du package *RSM* pour la modélisation des systèmes de contrôle

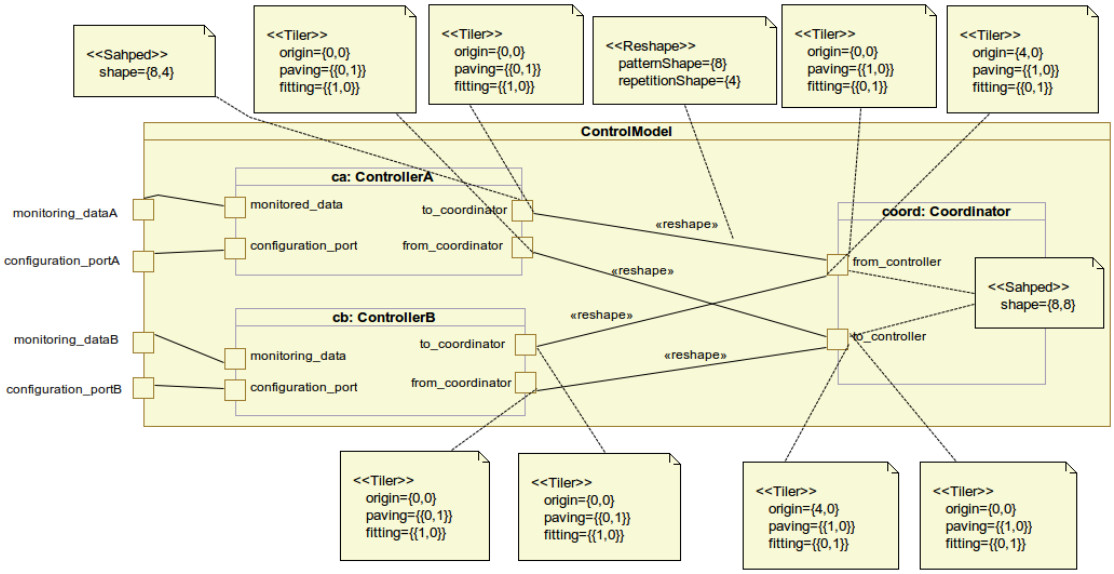


FIG. A.1 – Exemple générique sur l'utilisation des «*Reshape*» et «*Reshape*» pour la connexion entre contrôleurs et coordinateur

La figure A.1 représente un exemple plus générique de système de contrôle par rapport à celui présenté dans le chapitre 5. Ici, nous supposons que la dimension de l'interface *to_coordinator* du contrôleur *ca* est $\{8, 4\}$ (un tableau a deux dimensions). Pareil pour l'interface *to_coordinator* du contrôleur *cb*. Cela peut correspondre physiquement dans un langage HDL (dans la phase de génération du code) à un tableau de taille $\{8, 4\}$, où chaque élément du tableau est représenté par le même ensemble de ports physiques. L'interface *from_controller* du coordinateur *coord* aura comme taille $8x4 + 8x4 = 64$. Ici, on a choisi la dimension $\{8, 8\}$ pour cette interface. Pour le connecteur entre le port *to_coordinator* du contrôleur *ca* et le port *from_controller* du coordinateur

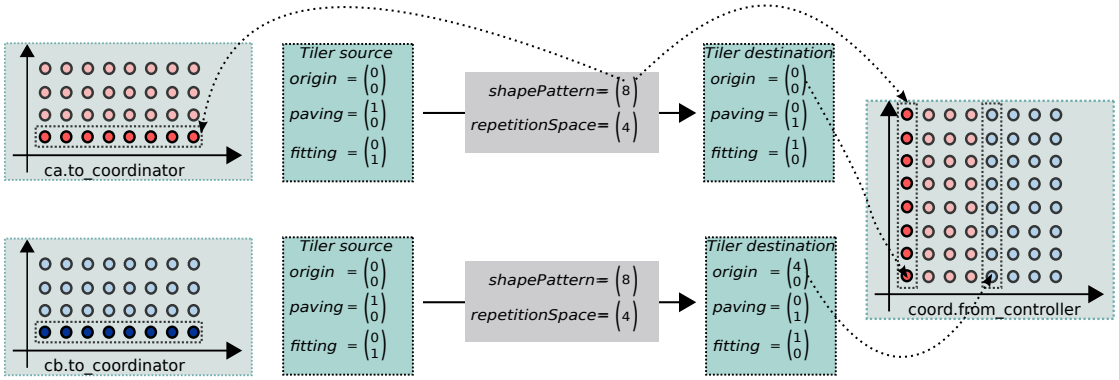


FIG. A.2 – Détails sur les connexions de la figure A.1

coord, les valeurs des attributs *patternShape* et *repetitionShape* du stéréotype «*Reshape*» sont {8} et {4}. Cela indique qu'à chaque répétition (des {4} répétitions), {8} éléments du tableau *to_coordinator* seront connectés au coordinateur.

Pour indiquer comment les éléments du tableau source seront parcourus et comment ils seront rangés dans le tableau destination, on utilise les stéréotypes «*Tiler*» qu'on applique sur les extrémités des connecteurs. Un *Tiler* a trois attributs : *origin*, *paving* et *fitting*. L'*origin* permet d'indiquer l'origine à partir de laquelle on commence à extraire (pour un *Tiler* source) ou remplir (pour un *Tiler* destination) les éléments du tableau. Ici, pour la connexion entre le port *to_coordinator* du contrôleur *ca* et le port *from_controller* du coordinateur *coord*, le *Tiler* source est du côté du port *to_coordinator* du contrôleur *ca* puisqu'il s'agit d'un port de sortie. L'extraction des éléments du tableau de ce port commence à l'origine $\{0, 0\}$. Le pavage (*paving*) indique comment se déplace l'origine du *shape* en passant d'une répétition à la suivante. Le pavage est une matrice ayant comme nombre de lignes, la dimension du tableau (ici, 2 pour un tableau $\{8, 4\}$) et comme nombre de colonnes la dimension de l'espace de répétitions (ici, 1 pour un espace de répétition {4}). Ici, la valeur $\{\{0, 1\}\}$ indique que la matrice contient une colonne ayant deux éléments ayant la valeur 0 pour la première dimension et 1 pour la deuxième. Cela indique qu'en passant d'une répétition à une autre, l'origine se déplace sur la deuxième dimension d'un élément. L'ajustage (*fitting*) indique comment le *shape* est construit à partir des éléments du tableau. L'ajustage est une matrice ayant comme nombre de lignes, la dimension du tableau (ici, 2 pour un tableau *8,4*) et comme nombre de colonnes la dimension du *shape* (ici, 1 pour un *shape* de taille *8*). Ici, la valeur $\{\{1, 0\}\}$ indique que pour construire le *shape*, il faut se déplacer 8 fois d'un élément sur la première dimension du tableau.

La figure A.2 donne une explication graphique du pavage et l'ajustage. Le port *to_coordinator* du contrôleur *ca* est représenté par un tableau ayant 8 colonnes et 4 lignes. A chaque répétition, on prend une ligne du tableau source, en commençant de l'indice $\{0, 0\}$ (*origin={0,0}* dans le *Tiler* source), pour la mettre dans une colonne, en commençant de l'indice $\{0, 0\}$ (*origin={0,0}* dans le *Tiler* destination), du tableau destination. Ici, il s'agit de connecter chaque ligne du tableau source à une colonne du tableau de sortie. Pour connecter le port *to_coordinator* du contrôleur *cb* au port *from_controller* du coordinateur *coord*, il faut commencer à l'indice $\{0, 4\}$ dans tableau destination, ce qui correspond à un *origin={0,4}* dans le *Tiler* destination.

Annexe B

La chaîne de transformations des modèles de contrôle

Cet annexe donne les détails de la chaîne de transformations de modèles de contrôle présentée dans le chapitre 5 et illustrée par la figure 5.11. Cette chaîne permet d'ajouter progressivement des détails au modèle UML en entrée afin de se rapprocher peu à peu de la cible VHDL. Les transformations de cette chaîne se basent sur le mécanisme du merge de Gaspard2, où les métamodèles sources et cibles sont des extensions du métamodèle MARTE. Ainsi, pour pouvoir exécuter une transformation donnée, il faut écrire ses règles de transformation et définir son métamodèle cible qui étend le métamodèle MARTE. Dans le reste de cet annexe, les métamodèles et les transformations utilisés dans cette chaîne sont présentés.

B.1 Extension du métamodèle MARTE pour supporter le contrôle semi-distribué

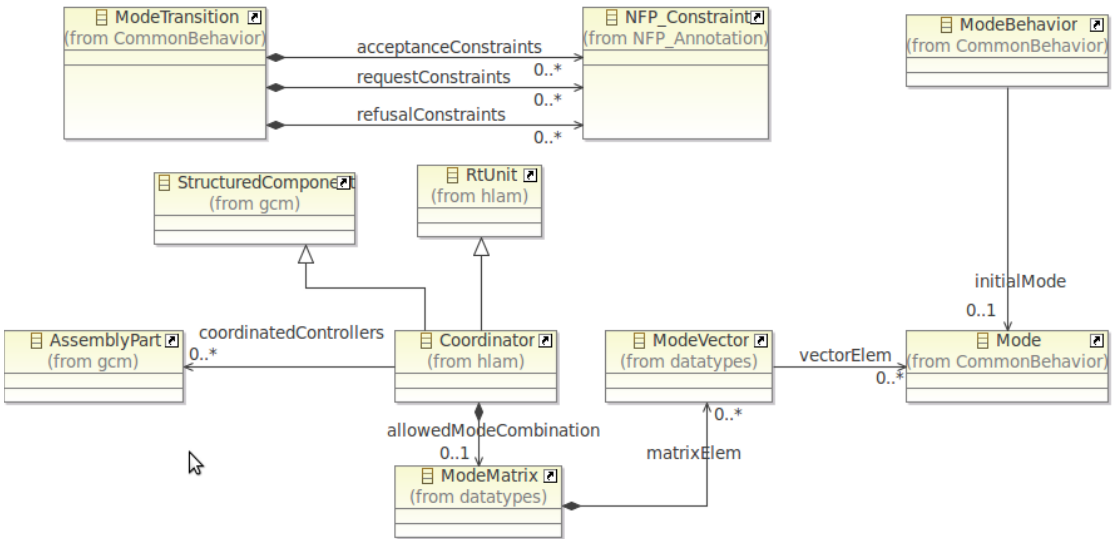


FIG. B.1 – Extension du métamodèle MARTE pour supporter le contrôle semi-distribué

La chaîne de transformations liée au contrôle commence par une transformation UML vers MARTE. Le métamodèle cible de cette transformation est une version étendue du métamodèle MARTE en introduisant les concepts du contrôle semi-distribué. La figure B.1 montre les concepts ajoutés au métamodèle MARTE pour supporter le contrôle semi-distribué. Trois associations avec la métaclasse *NFP_Constraint* du package *NFP_Annotation* sont ajoutées à la métaclasse *ModeTransition* du package *CommonBehavior*. Ces associations permettent d'ajouter des contraintes aux transitions pour spécifier les conditions déclenchant une requête de reconfiguration, ou une acceptation/refus d'une proposition. Bien que dans le profil MARTE le *ModeBehavior* étend la machine d'états UML, dans le métamodèle MARTE, il n'y a aucune indication sur l'état/mode initial du *ModeBehavior*. Pour cela, nous avons ajouté une association entre ce dernier et la métaclasse *Mode* pour spécifier le mode initial. Une métaclasse *Coordinator* a été également ajoutée au package *HLAM* pour représenter le coordinateur. Cette métaclasse a une association avec la métaclasse *AssemblyPart* du package *GCM* qui hérite de la métaclasse *Property* d'UML. Une *AssemblyPart* référence une instance d'un *StructuredComponent* qui est fait parti de la structure d'un autre. La figure B.2 illustre un extrait du package *GCM* de MARTE. Dans la transformation d'UML vers MARTE, les classes du modèle en entrée sont transformées en des *StructuredComponent*s MARTE et les ports et connecteurs de ces classes UML en des ports et connecteurs de la classe *StructuredComponent* suivant les concepts de la figure B.2. Les instances des classes sont transformées en des *AssemblyPart*s. Puisque les *coordinatedControllers* d'un coordinateur représentent les instances des contrôleurs coordonnés, elles sont représentées par une association entre la métaclasse *Coordinator* et la métaclasse *AssemblyPart*. Pour la table *GC* du coordinateur, elle est représentée par l'association *allowedModeCombination* entre *Coordinator* et *ModeMatrix*, qui a été ajouté tout comme *ModeVector* au package *DataTypes* de MARTE. Le résultat du *merge* entre le métamodèle MARTE original et les extensions illustrées dans la figure B.1 donne un métamodèle MARTE étendu. Le package *CommonBehavior* de ce métamodèle contient, en plus des éléments d'origine, les associations entre *ModeTransition* et *NFP_Constraint* et l'association entre *ModeBehavior* et *Mode*. Il en est de même pour les packages *HLAM* et *DataTypes* qui sont étendus par le merge.

B.2 La transformation d'UML vers MARTE

La transformation UML vers MARTE permet de traduire le modèle en entrée selon les concepts du métamodèle MARTE en transformant, principalement les composants UML en des *StructuredComponent*, les propriétés en des *AssemblyPart* et les ports stéréotypés «*FlowPort*» en des *FlowPort*. Cette transformation permet aussi de transformer les données de déploiement modélisés en utilisant le profil déploiement de Gaspard2 en des classes compatibles avec le métamodèle MARTE. Le métamodèle cible de cette transformation est le métamodèle MARTE intégrant l'extension proposée dans ce travail pour le contrôle semi-distribué et l'extension faite par Gaspard2 pour le déploiement. Cette dernière est illustrée dans la figure B.3. La figure B.4 montre un extrait du résultat de la transformation UML vers MARTE appliquée au modèle UML étudié dans le chapitre 5. Le modèle résultant contient un package *Deployment Model* contenant les données de déploiement. La figure B.5 illustre les données de déploiement du module de reconfiguration selon les concept introduit par le métamodèle de déploiement de la figure B.3. A côté des données de déploiement, le modèle de sortie contient la traduction des composants du modèle d'entrée en *StructuredComponent*s et la traduction des types en *PrimitiveType* de MARTE comme le montre la figure B.4. Cette figure montre aussi que la classe stéréotypée «*Coordinator*» du modèle UML en entrée est transformée en une classe *Coordinator* (qui hérite de *StructuredComponent*).

Chaque contrôleur est représenté, dans le modèle de sortie, par un *StructuredComponent* contenant trois *AssemblyPart*s liés à ses trois modules (observation, décision et reconfiguration)

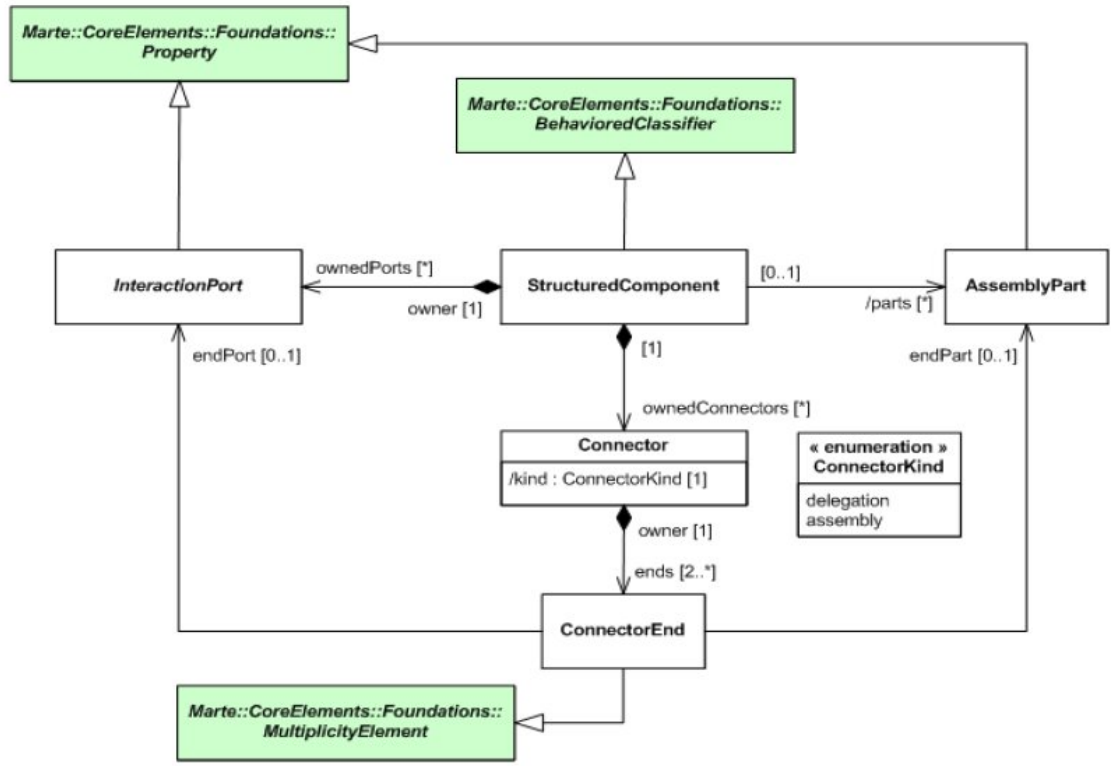


FIG. B.2 – Extrait du package *GCM* de MARTE

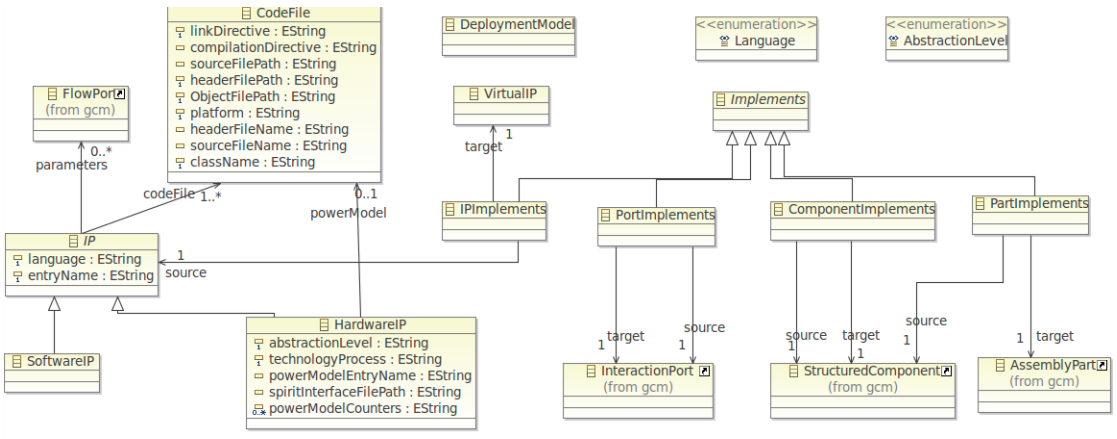


FIG. B.3 – Métamodèle de déploiement

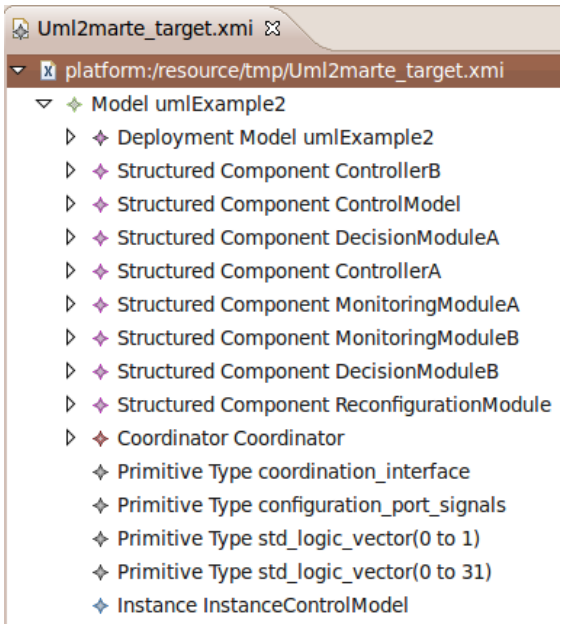


FIG. B.4 – Structure du modèle résultant de la transformation UML vers MARTE

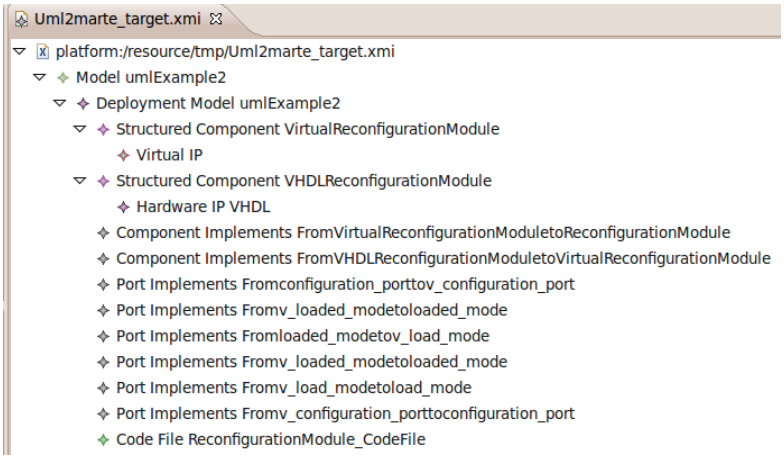
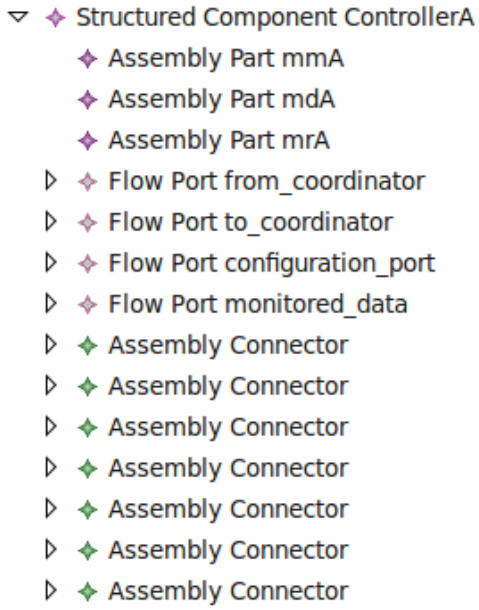
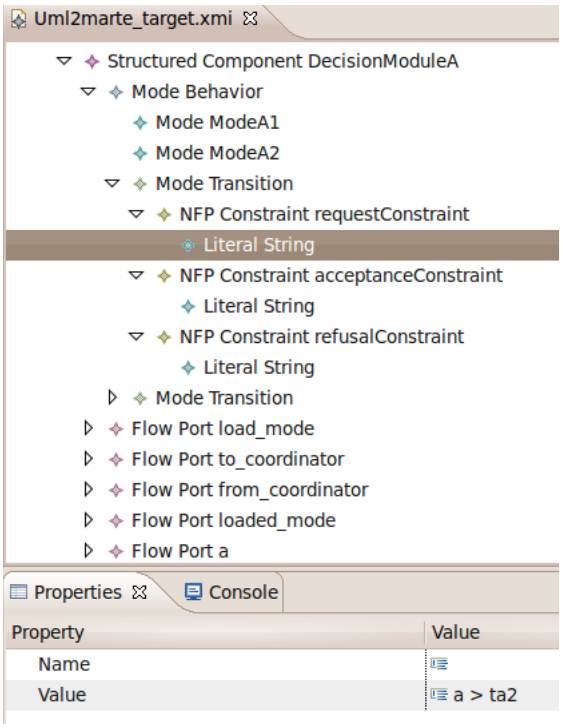


FIG. B.5 – Extrait du modèle MARTE décrivant les données de déploiement du module de reconfiguration



(a) Structure du contrôleur *ControllerA*



(b) Structure du *ModeBehavior* du contrôleur *ControllerA*

FIG. B.6 – Extrait de la partie du modèle MARTE décrivant les contrôleurs distribués

comme le montre la figure B.6(a). Ces *StructuredComponent*s contiennent aussi les *FlowPort*s représentant les ports du contrôleur ainsi que des connecteurs entre les *AssemblyPart*s. Les classes représentant les modules de décision des contrôleurs sont transformées en *StructuredComponent* contenant, en plus des ports, des *ModeBehavior* comme le montre la figure B.6(b). Ces *ModeBehavior* contiennent des *Mode*s et des *ModeTransition*s. Chaque *ModeTransition* a un ensemble de contraintes liées à la requête, l'acceptation ou le refus de cette transition. La figure B.7 montre la partie du modèle MARTE liée au coordinateur. La classe *Coordinator* contient, en plus des ports, une *ModeMatrix* décrivant sa table *GC*.

B.3 La transformation MARTE2PortInstance

Cette transformation fait parti des transformations génériques (utilisées pour des chaînes ciblant différentes plates-formes) de Gaspard2. Nous décrivons brièvement cette transformation puisqu'elle ne fait pas parti de notre contribution dans l'environnement Gaspard2. Cette transformation permet de créer des instances de ports puisque selon la spécification UML [98], les ports sont des points d'interaction entre composants mais les *Property* ayant comme type un composant qui a des ports n'ont pas des instances de ces ports. Pour ce faire, Gaspard2 propose une extension du métamodèle MARTE illustrée dans la figure B.8. Ici, la notion de *PortPart* permet de désigner une instance de port. Les notions de *PortConnector* et *PortConnectorEnd* seront utilisés au lieu des anciens concepts de MARTE pour les connecteurs. Cette extension introduit

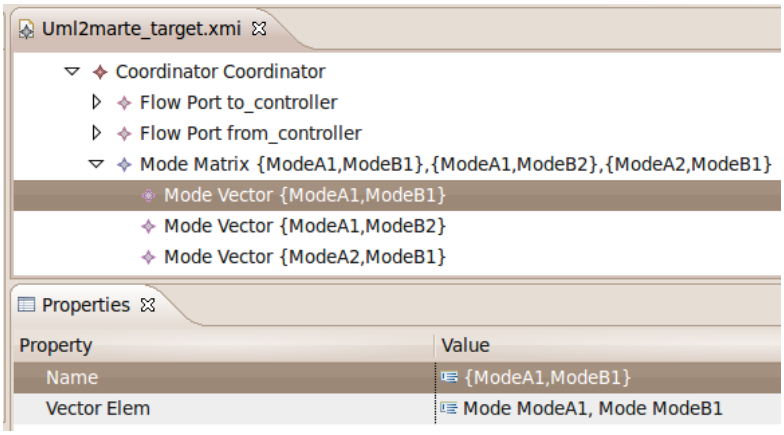


FIG. B.7 – Extrait de la partie du modèle MARTE décrivant le coordinateur

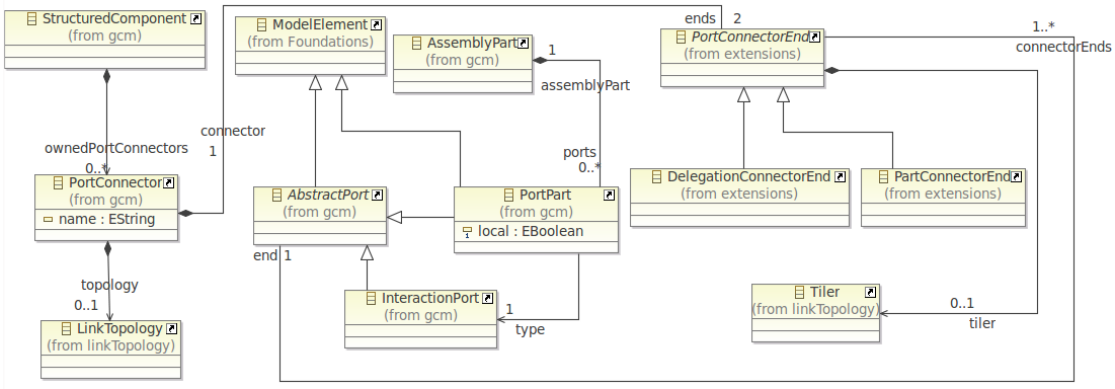


FIG. B.8 – Métamodèle des instances de ports

aussi les concepts de*PartConnectorEnd* et *DelegationConnectorEnd* pour faire la différence entre une extrémité de connecteur liée à une instance de port ou à un port, respectivement.

La figure B.9 illustre l'association des instances de ports aux *AssemblyParts* du contrôleur *ControllerA*. Cette figure montre l'ajout de ces instances par rapport au modèle de la figure B.6(a). La transformation des connecteurs est illustrée dans les figures B.10(a) et B.10(b). La figure B.10(a) montre la représentation du connecteur entre le port *to_coordinator* du contrôleur *ControllerA* et le port *to_coordinator* de son module de décision *mdA* dans le modèle résultant de la transformation UML vers MARTE. Comme le montre la figure, la connexion se faisait entre deux *FlowPort*s alors qu'il s'agit d'une connexion entre un port et une instance de port. Ce connecteur est transformé par MARTE2PortInstance en un connecteur entre un *FlowPort* et un *PortPart* comme le montre la figure B.10(b).

B.4 La transformation ClockReset

Cette transformation permet d'ajouter des ports pour le signal d'horloge et de réinitialisation aux composants puisque la plate-forme cible ici est la plate-forme VHDL. En fait, chaque transformation de la chaîne de transformation permet d'ajouter un ensemble de détails de bas-

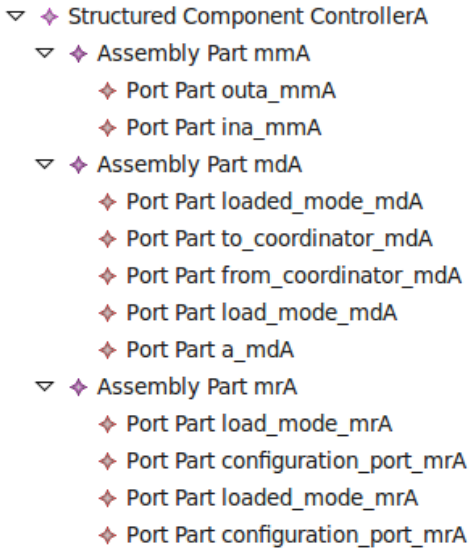
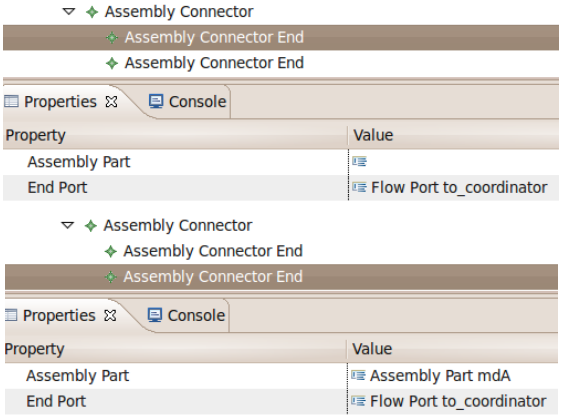
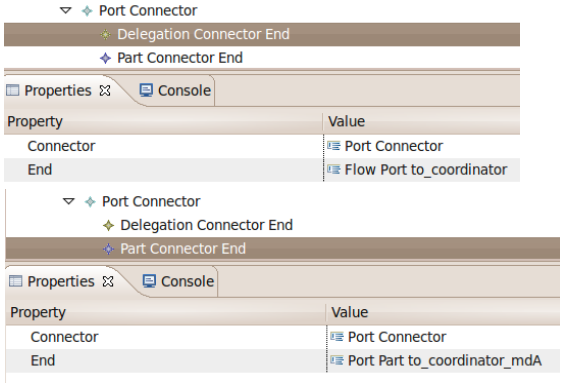


FIG. B.9 – Association des instances de ports aux *AssemblyParts*



(a) Les connecteurs après la transformation UML2MARTE



(b) Les connecteurs après la transformation MARTE2PortInstance

FIG. B.10 – La transformation des connecteurs par MARTE2PortInstance

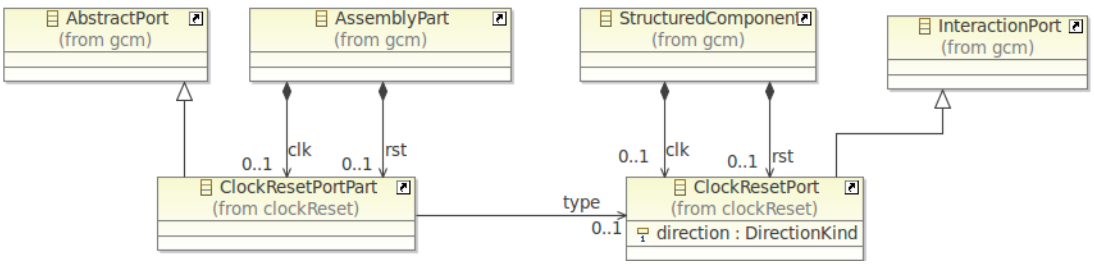


FIG. B.11 – Métamodèle des ports clock et reset

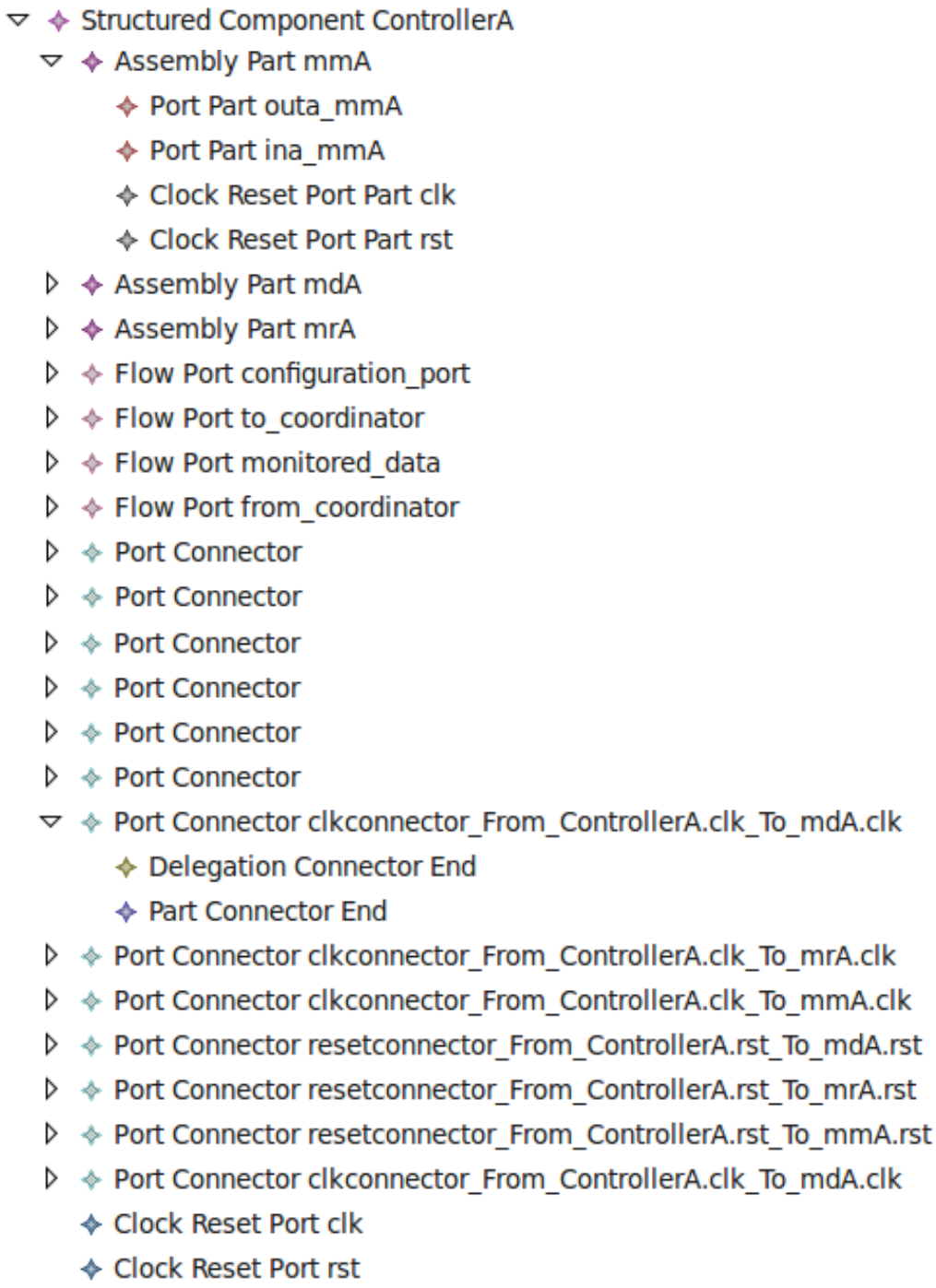


FIG. B.12 – Extrait du modèle résultant de la transformation *ClockReset*

niveau jusqu'à arriver à un modèle qui sera facilement traduit en code VHDL. La transformation ClockReset permet donc de passer d'un ensemble de composants abstraits à des composants qui ont la notion d'horloge et réinitialisation des composants matériels. La figure B.11 illustre l'extension du métamodèle MARTE proposée pour cela. Cette extension introduit les notions de *ClockResetPort ClockResetPortPart* désignant respectivement un port d'horloge/réinitialisation et une instance de port d'horloge/réinitialisation. La transformation *ClockReset* permet la création de ce type de ports/ instances de ports ainsi que les connexions correspondantes entre eux. Cela facilitera la génération de tout le système avec les connexions nécessaires entre ces ports spéciaux. La figure B.12 illustre l'ajout des ports, instances de ports et connecteurs par la transformation *ClockReset*. Cette figure montre l'ajout des concepts des ports d'horloge et réinitialisation au composant représentant le contrôleur *ControllerA*. Chaque *StructuredComponent* a un port d'horloge *clk* et un port de réinitialisation *rst*. Ces ports sont de type *ClockReset*. Les *AssemblyPart*s d'un *StructuredComponent* ont des instances de ces ports. Ces instances sont de type *ClockResetPortPart*. La transformation *ClockReset* génère aussi des connecteurs entre les ports d'horloge et de réinitialisation *rst* d'un *StructuredComponent* et les instances correspondantes des *AssemblyPart*s qu'il contient. Les extrémités des connecteurs ont le type *DelegationConnectorEnd* du côté du port d'un *StructuredComponent* et le type *PartConnectorEnd* du côté de l'instance du port d'un *AssemblyPart*.

B.5 La transformation VHDLSyntax

Cette transformation permet de traduire les éléments du modèle en entrée selon les concepts VHDL et d'avoir un modèle contenant toutes les notions nécessaires permettant de faciliter la génération du code en VHDL. Pour cela, nous proposons un métamodèle VHDL qui est compatible avec le métamodèle MARTE ce qui permet de faire la relation entre les composants du modèle d'entrée de la transformation et les concepts VHDL introduits par le métamodèle cible. Cette extension du métamodèle consiste en un package nommé *vhdl* et qui contient trois sous-packages : *foundations*, *design* et *statements*. Le sous-package package *foundations* contient les concepts de base de VHDL sans entrer dans les détails des vues structurelle et comportementale d'un système. La figure B.13 illustre le contenu de ce package et ses relations avec les autres sous-packages de package *vhdl* ainsi que ceux de MARTE (ici la métaclasse ModelElement fait parti du package Foundations de MARTE). Les concepts de base introduits par le package *foundations* sont le *TopLevel* désignant le module englobant tout le système, le *TypedElement* pour les éléments typés, le *LabeledElement* pour les éléments ayant des labels, le *AssignementElement* pour les éléments pouvant faire parti d'une affectation (la figure B.13 donne l'exemple du port VHDL, qui fait parti du sous-package *design*, en tant qu'élément d'affectation) et le *InstantiableUnit* pour les éléments qui peuvent être instanciés (la figure B.13 donne l'exemple du concept Entity de VHDL, qui fait parti du sous-package *design*, en tant qu'élément qui peut être instancié).

Le package *design* contient les éléments de structure d'un système VHDL. La figure B.14 illustre son contenu. Le concept *Entity* est parmi les principaux concepts introduits dans ce package. Ce concept considère un module VHDL en tant que boîte noire dont on ne connaît que l'interface et les paramètres s'il en existe. L'interface d'une *Entity* est décrite par un ensemble de ports. Comme le montre la figure B.14, la métaclasse *Entity* hérite de la métaclasse *InstantiableUnit*. Cette dernière a un ensemble de ports dont ceux de l'horloge et de réinitialisation comme illustré dans la figure B.13. Cette figure montre aussi que la métaclasse *InstantiableUnit* peut avoir un ensemble de *Generic* permettant de spécifier les paramètres. La description de l'*Entity* d'un point de vue structurelle ou comportementale se fait à l'aide d'*Architecture*s. Une *Architecture* contient un ensemble de déclarations qui peuvent être liées à des *Signal*s, *Variable*s, *Constant*s ou des *Components* comme le montre la figure B.14. Les valeurs des *Signal*s, *Variable*s, *Constant*s sont spécifiées à l'aide d'*Expression*s. En plus des déclarations, une *Architecture* contient un ensemble

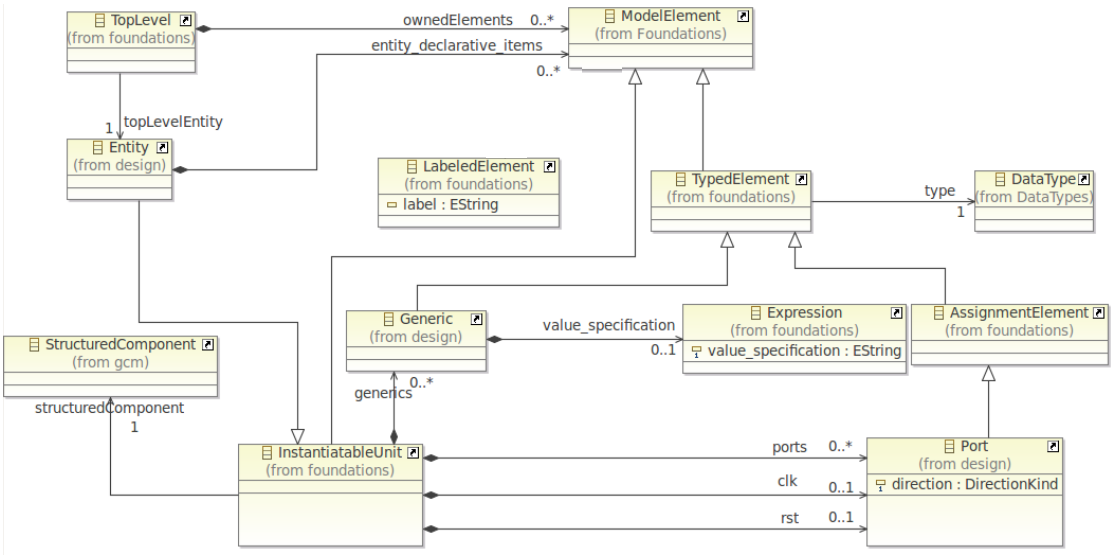


FIG. B.13 – Extrait du package *foundations* du métamodèle VHDL

d'instructions concurrentes qui peuvent être par exemple des déclarations de processus, ou des instanciations de composants. Ces instructions font parti du package *Statements*.

Pour la structure du package *Statements*, nous proposons l'utilisation de sous-packages *concurrent* et *sequential* pour séparer les instructions concurrentes, qu'on trouve par exemple dans le corps d'une architecture, et les instructions séquentielles, qu'on trouve par exemple à l'intérieur d'un processus. Le package *Statements* contient aussi des métaclasses qui ne font parti d'aucun de ces deux sous-packages et qui désignent des concepts communs pour les deux. Ces concepts communs sont illustrés dans la figure B.15 et sont représentés par les schémas d'instructions qui peuvent être utilisées en tant qu'instructions concurrentes ou séquentielles telles que les instructions avec des *generate*, des *if*, des *for* ou des *while*. Ces instructions sont désignées, respectivement, par les métaclasses *GenerateScheme*, *IfScheme*, *ForScheme* et *WhileScheme*. Une instruction d'affectation peut être aussi utilisée en tant qu'instruction concurrente ou séquentielle. Elle est désignée par la métaclasse *AssignementStatement*. Cette instruction contient une source (l'élément à affecter) et une cible (l'élément qui reçoit le contenu de la source). Puisque la source peut avoir n'importe quelle expression, elle est spécifiée à l'aide d'une *Expression* alors que la cible ne peut être qu'un élément simple ou un élément d'un tableau. Les indices du tableau correspondant à l'élément cible sont spécifiés par une expression, alors que le nom de la structure de la cible (un tableau ou un élément simple) est désigné par un *AssignementElement*.

Le contenu du sous-package *concurrent* des instructions concurrentes est illustré par la figure B.16. Parmi les principaux concepts utilisés pour nos systèmes de contrôle, les notions *GenerateStatement*, *ComponentInstantiation* et *ConcurrentAssignementStatement*. Une *GenerateStatement* permet de répéter les mêmes instructions avec des indices différents un nombre déterminé de fois. Cette instruction est utilisée entre autres pour la répétition des instanciations de composants (une *ComponentInstantiation*) ou des affectations des éléments d'un tableau. Le sous-package *concurrent* contient aussi la métaclasse *ConcurrentAssignementStatement* qui représente une instruction d'affectation qu'on peut trouver dans le corps d'une *Architecture*.

Un extrait du sous-package *sequential* est illustré par la figure B.17. Parmi les instructions de ce sous-package, on trouve les instructions avec des *if* (les *IfStatement*s) et les instructions avec des *case* (les *CaseStatement*s). L'instruction *IfStatement* peut être utilisée pour les automates de modes des contrôleurs et du coordinateur pour spécifier les conditions de requêtes, d'acceptation ou de

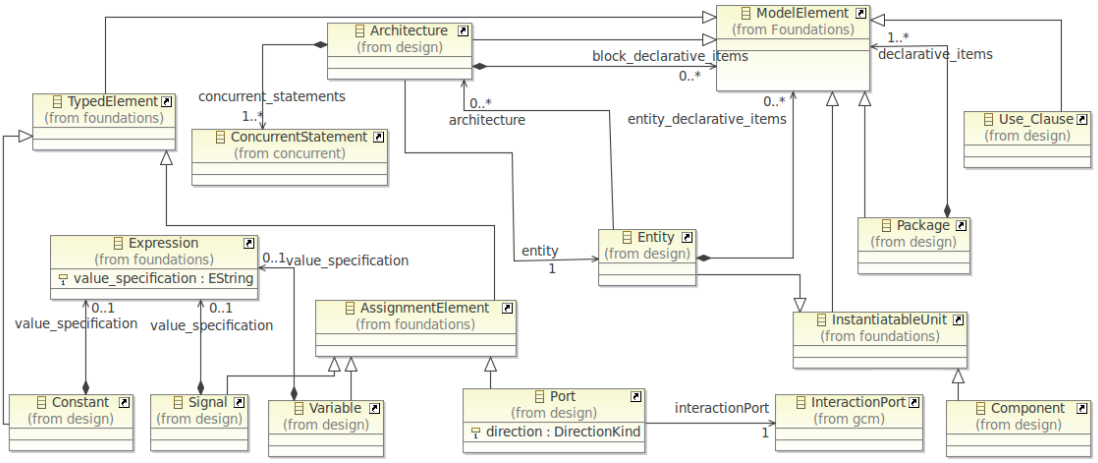


FIG. B.14 – Extrait du package *design* du métamodèle VHDL

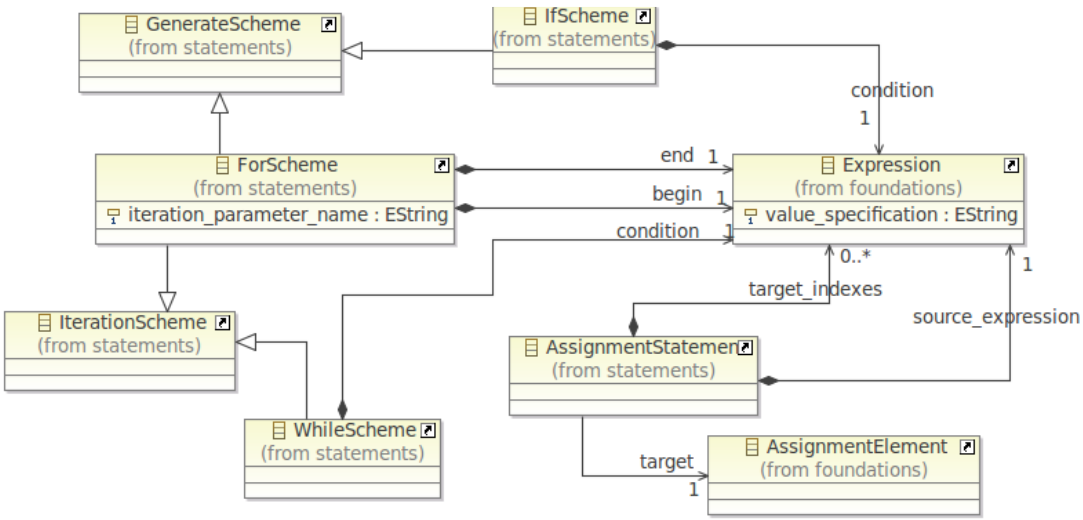


FIG. B.15 – Extrait du package *statements*

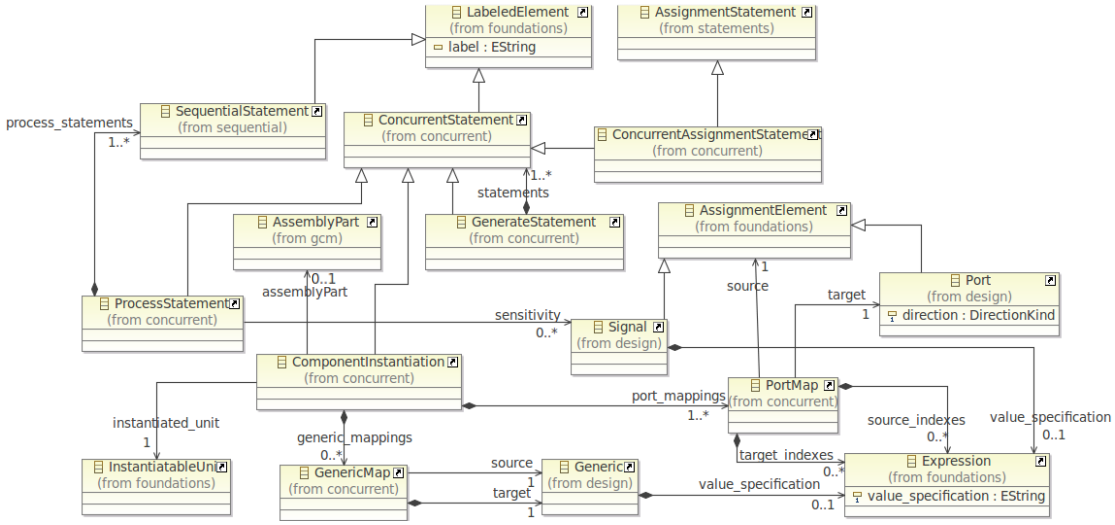


FIG. B.16 – Extrait du package *concurrent*

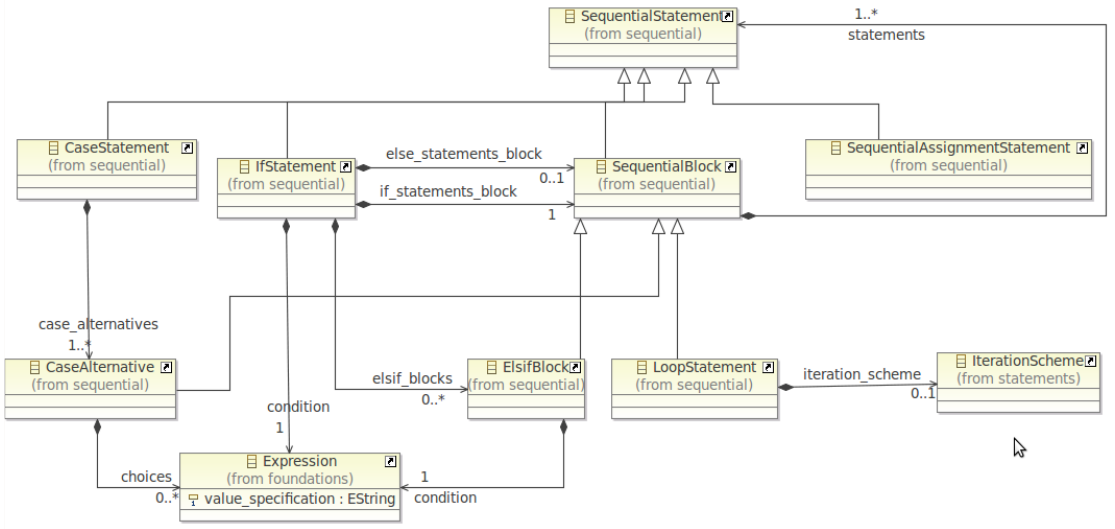


FIG. B.17 – Extrait du package *sequential*

refus des transitions. L'instruction *CaseStatement* peut être utilisée pour spécifier le traitement à faire selon le mode courant de l'automate. Or, nous avons choisi de ne pas utiliser ce sous-package pour la transformation *VHDLSyntax* parce le fait de passer par cette transformation pour avoir le contenu des automates, oblige à mettre beaucoup de détails de bas-niveau dans le modèle de sortie pour gérer, par exemple, la communication avec le coordinateur. Cela rend la transformation *VHDLSyntax* dépendante d'une implémentation donnée des contrôleurs, du coordinateur et de la communication entre eux. Il est donc préférable de laisser cela pour la transformation du modèle vers texte *VHDLCode*, et garder un certain niveau d'abstraction pour la transformation *VHDLSyntax*. Quant au sous-package *sequential*, il a été essentiellement proposé pour couvrir différentes possibilités offertes par le langage VHDL indépendamment du fait que la transformation cible le contrôle semi-distribué ou pas.

La figure B.18 montre un extrait du modèle résultant de la transformation *VHDLSyntax*. Cette transformation ajoute un *TopLevel* à l'instance du système. Dans ce *TopLevel*, on trouve les types, les composants, les entités et les architectures VHDL ajoutés par la transformation. Les types ajoutés par cette transformation sont sous-forme de tableaux VHDL. Ils sont utilisés pour les ports VHDL résultants de la transformation des ports MARTE ayant le stéréotype «*Shaped*». Comme nous avons précisé précédemment, ce stéréotype permet de spécifier qu'il s'agit d'une répétition du même port, et le nombre de répétitions est donné par l'attribut *shape* de ce stéréotype. Comme le montre la figure B.18, les deux types VHDL ajoutés correspondent aux types des ports du coordinateur, modélisé dans la figure 5.6(b), qui ont un *shape* = 2. Ces deux types VHDL sont des tableaux de 2 éléments. La figure B.19 illustre la description du type *array_2_Of_coordination_interface* qui est un tableau de 2 éléments de type *coordination_interface*.

La transformation *VHDLSyntax* crée aussi un *Component* VHDL pour chaque *StructuredComponent* du modèle d'entrée. La figure B.20 illustre le composant correspondant au système de contrôle modélisé dans la figure 5.7 et donne les détails de l'un des ports (nom, type, port correspondant dans le modèle d'entrée).

Comme le montre la figure B.18, la transformation *VHDLSyntax* crée un *Component* pour chaque *StructuredComponent*, y inclus ceux des composants de déploiement tels que *VHDLCoordinator* et *VHDLMonitoringModuleA*. Par contre, les *Entity* et les *Architecture* sont créées pour les composants sauf ceux dédiés aux déploiement. En effet, une architecture d'une entité liée à un composant élémentaire contient une instanciation du composant de déploiement. Par exemple, l'architecture du composant *MonitoringModuleA* contient une instanciation du composant *VHDLMonitoringModuleA* comme le montre la figure B.21. Cette instanciation est représentée par une *ComponentInstantiation* qui contient un ensemble de *PortMap* permettant le mapping entre les ports de l'entité et ceux de l'instance du composant.

La figure B.22 illustre l'architecture VHDL liée à un composant UML composé représentant le contrôleur *ControllerA* de la figure 5.2. Les connecteurs à l'intérieur de ce composant sont traduits en des signaux VHDL comme le montre la figure. Les connexions liées directement aux ports du composant global sont traduites en des instructions d'affectation (*ConcurrentAssignment*s) entre ces ports et un sous-ensemble de signaux créés par la transformation. La figure B.22 illustre l'instruction d'affectation pour la connexion entre le port *from_coordinator* du *ControllerA* et celui du composant de décision *mdA* à travers le signal *from_coordinatorTofrom_coordinator_mdA*.

La figure B.23 illustre l'architecture liée au système de contrôle de la figure 5.7. A la différence des connexions non stéréotypées, celles qui sont stéréotypées «*Reshape*» impliquent la création d'un signal pour chaque extrémités. L'architecture du système de contrôle contient, en plus des deux signaux liés aux connexions non stéréotypées, 6 signaux permettant la connexion entre les deux contrôleurs et le coordinateur. La figure B.23 donne les détails d'un signal lié à un port du coordinateur et un autre lié à un port de l'un des contrôleurs. Le type d'un signal prend en compte le nombre de répétitions de l'instance du composant et le nombre de répétitions du port dans une instance. Par exemple, si une instance du composant est répétée 2 fois et qu'un port est répété 3 fois dans chaque instance, le signal correspond est un tableau de taille

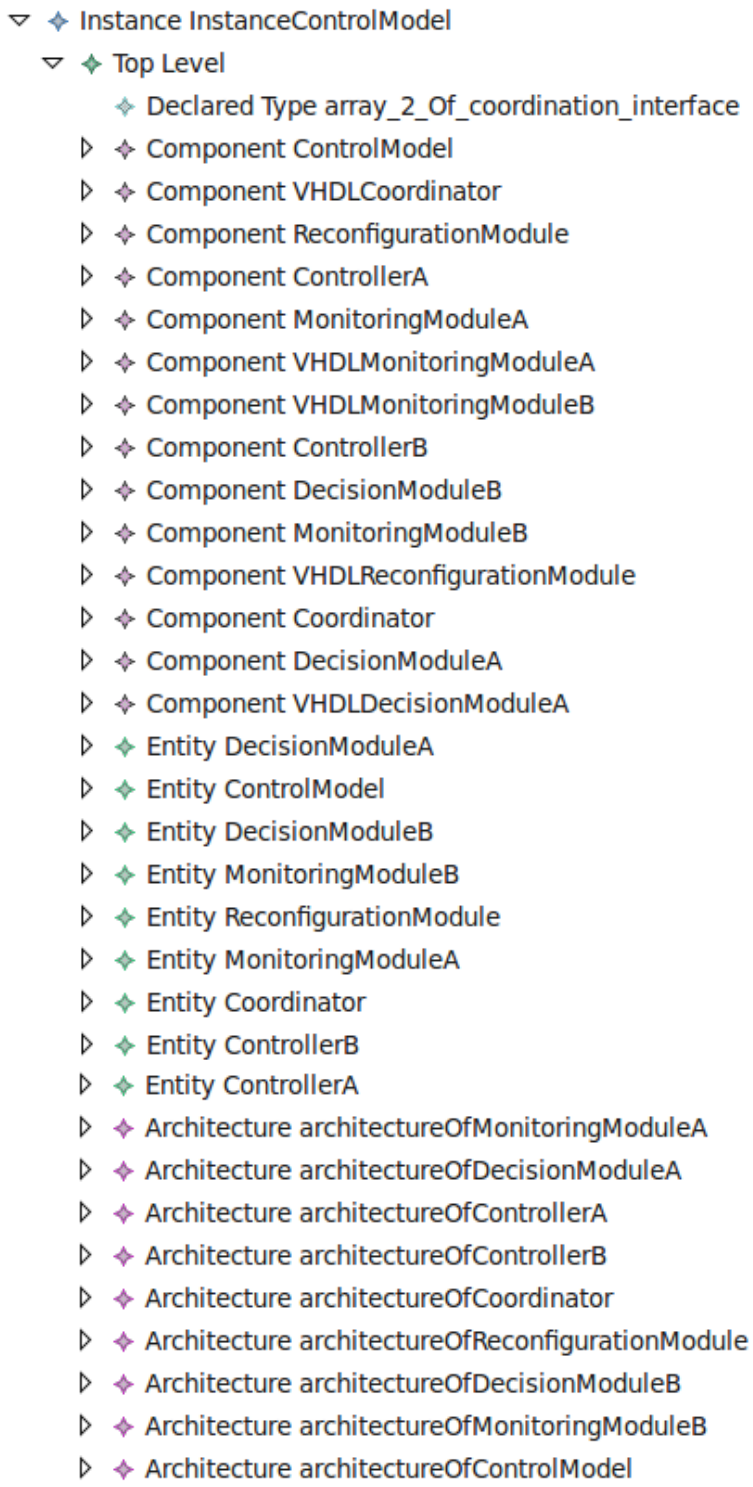


FIG. B.18 – Extrait du résultat de la transformation *VHDLSyntax*

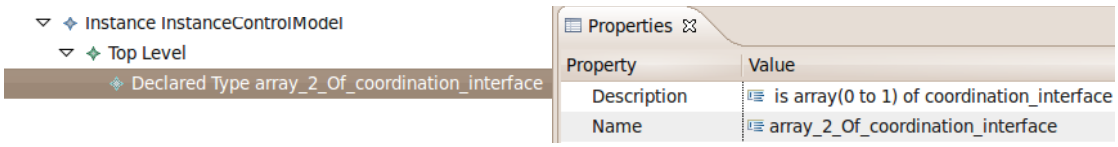


FIG. B.19 – Création de types VHDL par la transformation *VHDLSyntax*

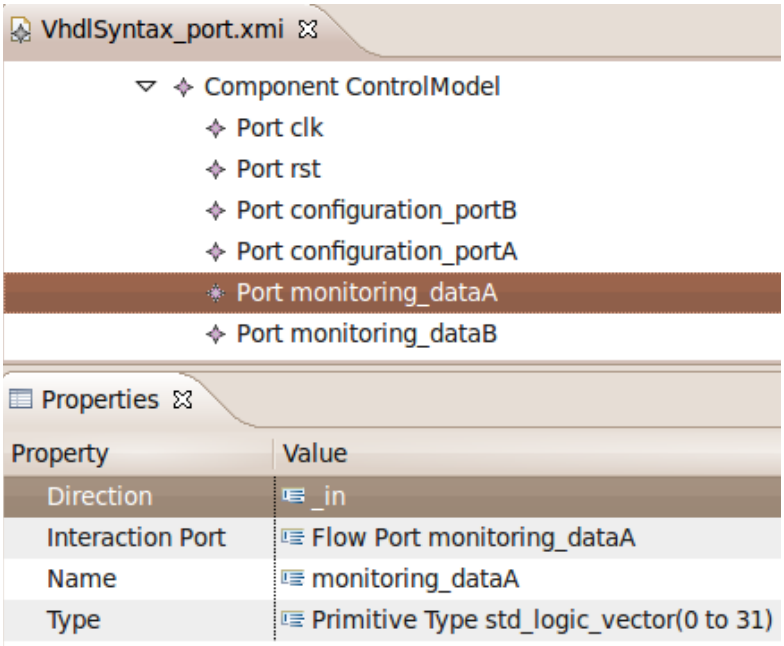


FIG. B.20 – Création des *Component*s VHDL par la transformation *VHDLSyntax*

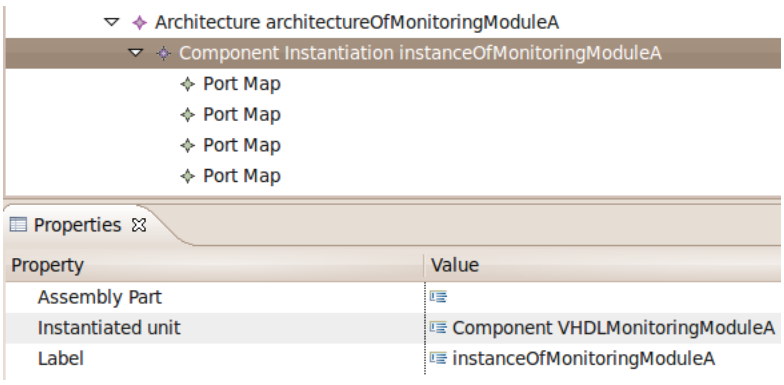


FIG. B.21 – Exemple d'instanciation de composant dans une architecture VHDL

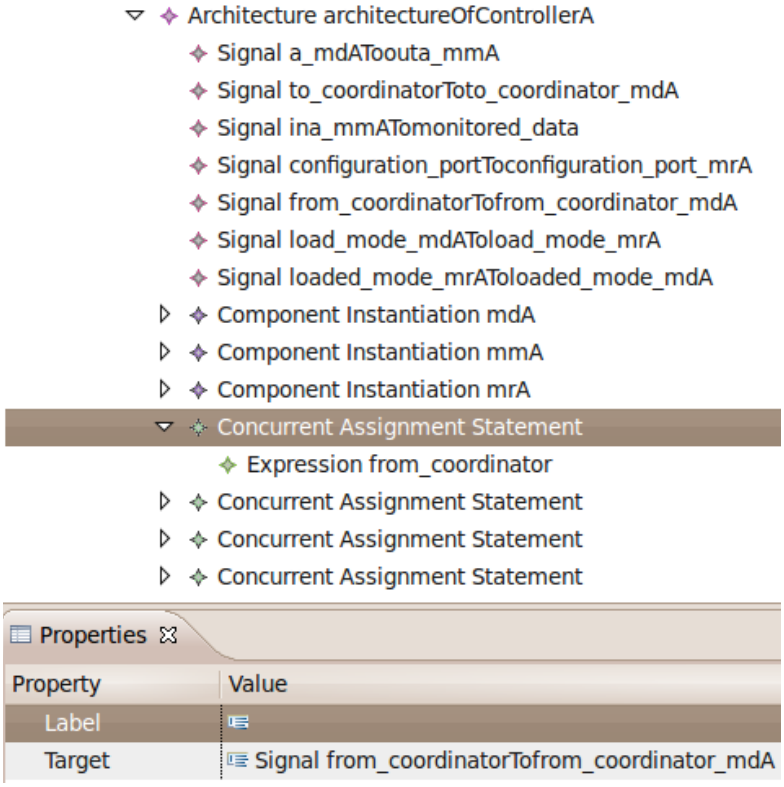


FIG. B.22 – Exemple de la gestion des connexions dans une architecture VHDL

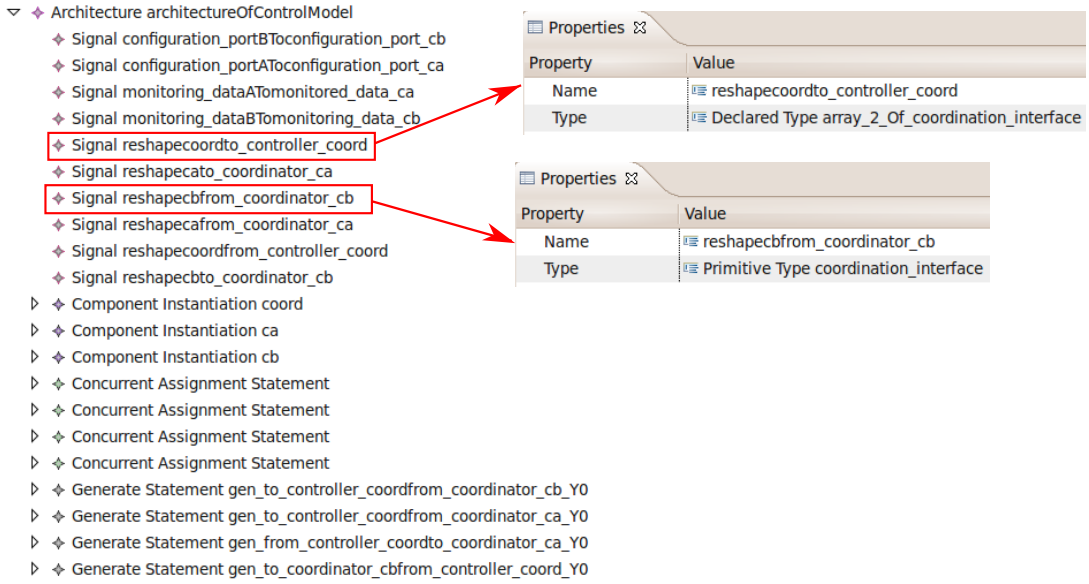


FIG. B.23 – L'architecture VHDL du système du contrôle

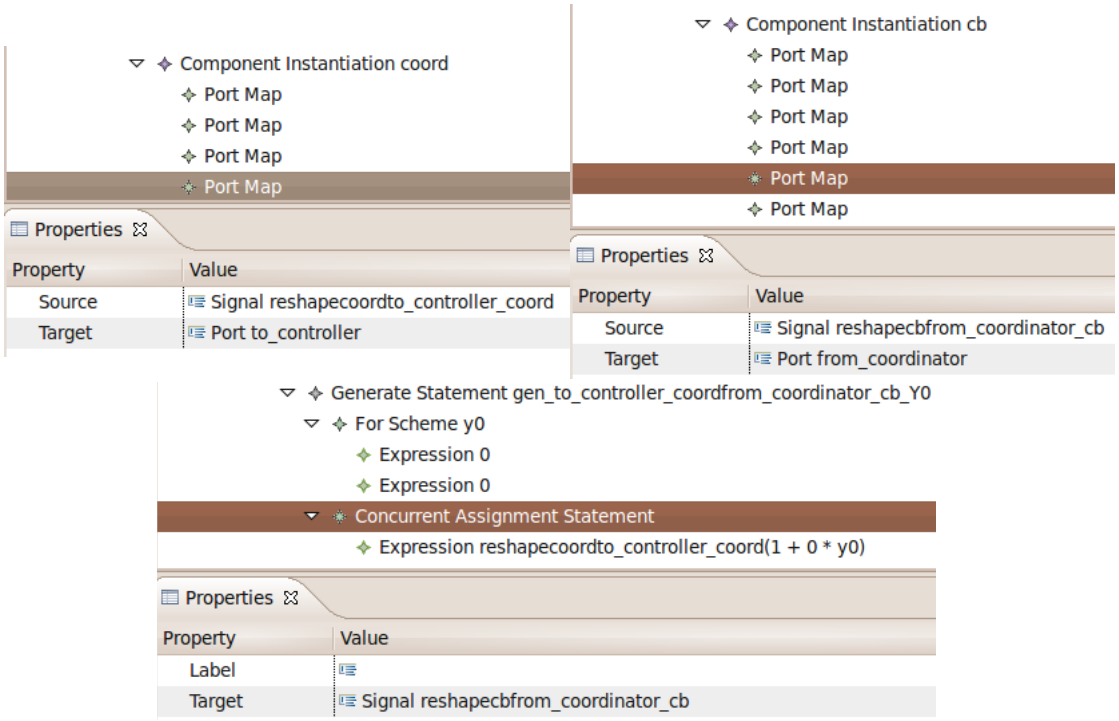


FIG. B.24 – Traduction des *reshape*s MARTE selon la syntaxe VHDL

[2, 3]. Comme le montre la figure B.23, le signal *reshapecoordto_controller_coord* est un tableau de 2 éléments traduisant un *shape* = 2 du port *to_controller* du coordinateur. Le signal *reshapecbfrom_coordinator_cb* est de type de *coordination_interface* puisque le port *from_coordinator* du contrôleur *ControllerB* est de ce type. La figure B.24 illustre les *port map* de l'instance du coordinateur (*coord* à gauche) et celle du contrôleur *ControllerB* (*cb* à droite) utilisant respectivement les signaux *reshapecoordto_controller_coord* et *reshapecbfrom_coordinator_cb* de la figure B.23. Comme expliqué dans la figure 5.7, le «*Reshape*» entre le port *to_controller* du coordinateur *coord* et le port *from_coordinator* du contrôleur *cb* a comme tiler source qui a une origine égale à {1}. Cela indique que le port *from_coordinator* du contrôleur *cb* est lié au deuxième élément du tableau représentant le port *to_controller* du coordinateur *coord*. Cette connexion est traduite en une instruction *generate* VHDL (*GenerateStatement*) comme le montre la partie inférieure de la figure B.24. Le nombre de répétitions gérées par cette instruction correspond au nombre de répétitions du port du côté du tiler source. Ici, il s'agit du port *from_coordinator* du contrôleur *cb* qui n'est pas répété (le port n'est pas stéréotypé «*Shaped*» et le contrôleur *cb* n'est pas répété non plus). La boucle *for* du *generate* commence à 0 et finit à 0 indiquant qu'il s'agit d'une seule répétition. L'affectation entre les signaux liés aux deux ports de la connexion est gérée par une instruction d'affectation *ConcurrentAssignmentStatement* comme le montre la figure B.24. La source de cette affectation est le deuxième élément du tableau du signal lié au port *to_controller* du coordinateur *coord*. La cible est le signal lié au port *from_coordinator* du contrôleur *cb*.

Contrôle matériel des systèmes partiellement reconfigurables sur FPGA : de la modélisation à l'implémentation

Résumé : Ce travail propose une méthodologie de conception du contrôle pour les systèmes reconfigurables sur FPGA, visant à améliorer la productivité des concepteurs et assurer l'efficacité de l'implémentation. Cette méthodologie est basée sur un modèle de contrôle semi-distribué qui se compose d'un ensemble de contrôleurs distribués modulaires assurant chacun les tâches d'observation, de prise de décision et de reconfiguration pour une région reconfigurable du système, et d'un coordinateur entre les décisions des contrôleurs distribués afin de respecter les contraintes et objectifs globaux du système. Cette prise de décision semi-distribuée est basée sur le formalisme des automates de modes. Cette combinaison entre modularité, division du contrôle et formalisme permet d'améliorer la flexibilité, réutilisabilité et scalabilité de la conception du contrôle. Un autre point peut être ajouté à cette combinaison pour améliorer la productivité des concepteurs, qui est l'automatisation. Pour cela, la méthodologie proposée est basée sur une approche d'Ingénierie Dirigée par les Modèles permettant d'automatiser la génération du code à partir de modèles de haut-niveau d'abstraction. Cette approche fait usage du profil standard MARTE (Modeling and Analysis of Real-Time and Embedded Systems), permettant de rendre les détails techniques de bas niveau transparents aux concepteurs et d'automatiser la génération du code VHDL pour une implémentation matérielle des systèmes de contrôle modélisés afin d'assurer leur performance. Les systèmes de contrôle générés ont été validés par simulation. Les résultats de synthèse ont montré un coût acceptable en termes de temps d'exécution et de ressources pour des systèmes ayant différents nombres de contrôleurs. Un système de contrôle composé de quatre contrôleurs et d'un coordinateur a été également validé par implémentation physique dans un système FPGA pour une application de traitement d'images.

Mots-clés : Contrôle distribué, Auto-adaptation, Coordination, Reconfiguration dynamique partielle, FPGA, Automates de modes, Ingénierie Dirigée par les Modèles, UML, MARTE.

Hardware control of partially reconfigurable FPGA systems : from modeling to implementation

Abstract : This work proposes a control design methodology for FPGA-based reconfigurable systems aiming at increasing control design productivity and guaranteeing implementation efficiency. This methodology is based on a semi-distributed control model composed of a set of modular distributed controllers executing each observation, decision-making and reconfiguration tasks for a reconfigurable region of the system, and a coordinator between the distributed controllers decisions in order to respect global systems constraints and objectives. This semi-distributed decision-making is based on the mode-automata formalism. The proposed combination between modularity, control splitting and formalism-based design allows to enhance the flexibility, reusability and scalability of the control design. Another point that can be added to this combination, to enhance design productivity, is design automation. For this, the proposed methodology is based on Model-Driven Engineering approach allowing to automate code generation from high-level models. This approach makes use of the UML MARTE (Modeling and Analysis of Real-Time and Embedded Systems) standard profile, allowing to make low-level technical details transparent to designers and to automate the VHDL code generation for hardware implementation of the modeled control systems in order to guarantee their performance. The generated control systems were validated using simulation. Synthesis results showed an acceptable time and resource overhead for systems having different numbers of controllers. A control system composed of four controllers and a coordinator was also validated through physical implementation in an FPGA system for an image processing application.

Keywords : Distributed control, Auto-adaptivity, Coordination, Partial dynamic reconfiguration, FPGA, mode-automata, Model-Driven Engineering, UML, MARTE.