


Université des sciences et technologies de lille

THÈSE

présentée et soutenue publiquement le 20 novembre 2006

pour obtenir le titre de

Docteur en informatique

par

Ouassila Labbani

Modélisation à haut niveau du contrôle dans des applications de traitement systématique à parallélisme massif

Composition du jury

Président :	Atika Menhaj-Rivenq	MdC, HdR	Université de Valenciennes
Rapporteurs :	Jean-Philippe Babau	MdC, HdR	INSA, Lyon
	Robert De Simone	DR	INRIA, Sophia Antipolis
Examinateurs :	Éric Rutten	CR, HdR	INRIA Rhône-Alpes
Directeurs :	Jean-Luc Dekeyser	Professeur	LIFL, Université de Lille I
	Pierre Boulet	Professeur	LIFL, Université de Lille I

UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE
Laboratoire d'Informatique Fondamentale de Lille — UPRESA 8022
U.F.R. d'I.E.E.A. – Bât. M3 – 59655 VILLENEUVE D'ASCQ CEDEX
Tél. : +33 (0)3 28 77 85 41 – Télécopie : +33 (0)3 28 77 85 37 – email : direction@lifl.fr

Remerciements

C'est avec une grande émotion et beaucoup de sincérité que je voudrais remercier toutes les personnes qui, par leur participation et leurs encouragements, m'ont permise de mener à bien ce travail.

Cette thèse n'aurait vu le jour sans la confiance, la patience et la générosité de mes directeurs de recherche, M. Jean-Luc DEKEYSER *et M. Pierre* BOULET*, que je veux vivement remercier. Leurs conseils précieux m'ont permis d'améliorer mes connaissances et d'aboutir à la production de ce travail. Je voudrais aussi les remercier pour le temps qu'ils m'ont accordé tout au long de ces années, d'avoir cru en mes capacités et de m'avoir fourni d'excellentes conditions de travail.*

Mes plus sincères remerciements vont également aux membres du jury, qui ont accepté d'évaluer mon travail de thèse. Merci à Mme. Atika MENHAJ-RIVENQ *d'avoir accepter de présider le jury de cette thèse, et à M. Jean-Philippe* BABAU *et M. Robert* DE SIMONE *d'avoir accepté d'être les rapporteurs de ce manuscrit et de m'avoir fait l'honneur de juger ce travail avec intérêt. Leurs remarques et suggestions lors de la lecture de mon rapport m'ont permis d'apporter des améliorations à la qualité de ce dernier. Merci également à M. Éric* RUTTEN *pour la gentillesse et la patience qu'il a manifestées à mon égard durant cette thèse, pour tous les conseils qu'il a bien voulu me donner, et aussi pour avoir accepté d'examiner mon mémoire et de faire partie de mon jury de thèse.*

Un grand remerciement est également adressé à tous les membres de l'équipe WEST pour l'ambiance très favorable qu'ils ont su créer autour de moi. C'est en partie grâce à eux que j'ai pu évoluer au cours de ma thèse et apprécier toutes ces années au sein d'un milieux familial très chaleureux. Merci à Philippe M., Cédric, Abdoulaye, Samy et Smaïl pour leur gentillesse et précieux conseils, à Lossan pour son grand cœur, à Éric P. pour son humour particulier, à Julien T. toujours prêt à aider, à Huafeng et Imran pour leur sympathie, à Rabie, Ashish et Calin toujours sérieux, et avec beaucoup d'émotions, je dit merci à mes confidents Antoine et Sébastien toujours à l'écoute, pour leur gentillesse et pour leur soutien moral. Je n'oublie pas non plus les anciens membres de l'équipe : Arnaud, Philippe D., Ali, Chadi, Luc, Joël, Julien S. et Stéphane. Ainsi que les nouveaux : Anne, Emmanuel, Abou El Hassan, Souha et Hajer.

Je tiens aussi à remercier les membres des équipes RD2P et SMAC avec qui j'ai partagé de très agréables moments. Merci également à tous les membres du laboratoire d'informatique fondamentale de Lille pour leur sympathie et leur grand sourire à chaque fois que nos chemins se croisent dans les couloirs du bâtiment M3. Je n'oublierai pas les aides permanentes

reçues du personnel administratif.

Je ne peux oublier à cette occasion tous les enseignants qui ont contribué à ma formation, de mes premières lettres d'alphabets jusqu'à ce manuscrit.

Je ne saurais exprimer ma profonde reconnaissance et gratitude à mes parents qui m'ont tout enseigné et ont suivi et soutenu mon évolution depuis mes premiers pas. Ils ont su me donner toutes les chances pour réussir. Leur amour et leurs perpétuels encouragements ont été la source et la force d'aboutissement de ce travail qui est en grande partie le leur, qu'il leur soit dédié. C'est avec les larmes aux yeux que j'écris ces phrases et une grande envie de les serrer dans mes bras pour leur dire : « je vous aime énormément ».

Je remercie toutes les personnes qui me sont chères et qui tiennent une grande place dans mon cœur : mes frères Fouad et Fayçal et ma soeur Ouafa qui ont toujours été présent et m'ont soutenu dans mes efforts, ma soeur Ouiddad et son mari Noureddine envers lesquels je suis très reconnaissante pour leur encouragements qui m'ont permis de traverser la méditerranée et d'arriver à mon objectif, ainsi que mes amis Nadia, Sara et Moncef, sans oublier Alexandre et toute sa famille, et tous les autres auxquels je pense et qui, je l'espère, ne m'en voudront pas de ne pas les citer.

Une dédicace spéciale à mes adorables petites nièces Lyna et Lamyce en leur souhaitant beaucoup de bonheurs.

Enfin, une pensée émue à mes grands parents et à mon oncle Saleh qui auraient pu être fières de moi s'il étaient toujours de ce monde.

إِلَى أَعَزِ النَاسِ إِلَى قَلبِي، أُمِي وَ أَبِي، مَعَ كُلِ حُبِي وَ إِمتِنَانِي.

وَسِيلَة

Table des matières

Introduction Générale

Chapitre 1

Introduction et motivations

1.1 Contexte

L'évolution rapide de la technologie informatique représente l'un des plus importants phénomènes techniques depuis plusieurs décennies. Tout le monde aura remarqué ces dernières années l'inéluctable course à la miniaturisation de tout appareil électronique. Prenons par exemple le cas d'évolution des téléphones portables, les premiers systèmes offrant le service de téléphonie mobile ont été introduits au début des années cinquante. Ces systèmes étaient souvent assez gros et avaient une antenne assez imposante. Ils étaient limités par une mobilité restreinte, une capacité basse, un service réduit et une mauvaise qualité du son. De plus, le prix de ces téléphones était souvent élevé, et leur autonomie bien faible. Aujourd'hui, le monde est transformé par la création des téléphones mobiles numériques plus petits, plus légers et plus performants. Ces téléphones sont également, de plus en plus, multimédia et fournissent une meilleure qualité et un grand nombre de service. Nous les utilisons dans des domaines très variés : pour passer des appels bien sûr, pour prendre des photos, pour écouter de la musique, et même pour visualiser des vidéos. De plus, les téléphones portable de nos jours offrent des fonctions de traitement d'image impressionnantes qui demandent des puissances de calcul similaires à celles des ordinateurs.

Ces grands progrès technologiques, permettant de développer des systèmes informatiques plus petits, plus compacts et plus rapides, sont en grande partie dus à une capacité d'intégration de plus en plus élevée. Les techniques de conception des systèmes embarqués permettent maintenant de regrouper des systèmes hétérogènes sur la même puce électronique, donnant ainsi naissance à un nouveau paradigme dans les systèmes embarqués : les systèmes sur puce (*SoC : System on Chip*). Ces systèmes se développent inévitablement et se retrouvent dans des domaines comme les applications de télécommunications et du traitement multimédia. Ces applications opèrent généralement dans des conditions temps réel et sont souvent basées sur des traitements complexes et fortement parallélisables. Les systèmes sur puce doivent être donc conçus de manière efficace afin d'assurer la fiabilité de ces applications et de minimiser leur temps d'arrivée sur le marché qui sont des facteurs importants de réussite. Ils doivent être également développés et vérifiés de manière sûre pour minimiser leur coût de fabrication.

Pour répondre à ces besoins, les industriels et les scientifiques doivent suivre cette révolution technologique au prix d'énormes efforts en recherche et développement. Dans ce contexte, plusieurs projets de recherche ont été lancés pour faciliter et assister le développement des systèmes sur puce. Le projet INRIA DaRT[1], dans lequel s'inscrit le travail présenté dans ce document, en fait partie. Un des principaux objectifs de ce projet est la mise en œuvre d'une méthodologie et d'un environnement de développement pour les systèmes sur puce à hautes performances. Ces systèmes sont particulièrement adaptés aux applications de traitement de données intensives qui comportent une partie de traitement du signal intensif ou de l'image. Il s'agit de fournir un cadre unifié pour le développement de ces systèmes en partant de leur modélisation au plus haut niveau d'abstraction jusqu'à la génération du code.

Le domaine d'application visé par le projet DaRT est donc celui des applications de traitement systématique à parallélisme massif. Les caractéristiques principales de ces applications sont qu'elles effectuent une grande quantité de calculs réguliers sur des données *multidimensionnelles*, elles opèrent dans des conditions temps réel et présentent un degré élevé de pa-

[1] `http://www.inria.fr/rapportsactivite/RA2005/dart/dart_tf.html`

rallélisme de données et de calcul. L'objectif du projet DaRT est donc de fournir un environnement de développement pour ces applications de traitement parallèles en se basant sur le modèle de spécification ARRAY-OL. Cependant, ce modèle ne permet pas la représentation des comportements de contrôle qui sont généralement indispensables pour la description de certaines applications de traitement parallèle plus évoluées. L'objectif de notre travail est donc d'étudier l'introduction des concepts de contrôle dans des applications massivement parallèles.

De façon générale, le contexte global de notre travail se situe à l'intersection de trois axes de recherche (figure 1.1). Le premier axe est relatif à l'étude des applications de traitement

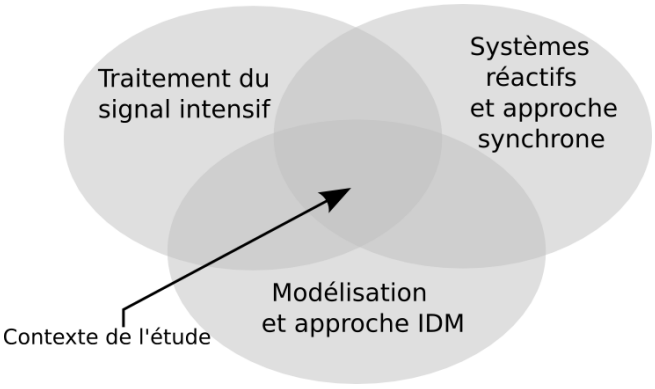


FIG. 1.1 – *Contexte de l'étude*

systématique à parallélisme massif, et en particulier celles dédiées au traitement du signal intensif, le domaine d'application privilégié sur lequel se basent nos travaux. Le deuxième axe recouvre les systèmes réactifs hybrides, il nous sert comme source d'inspiration pour l'introduction des comportements réactifs aux applications parallèles. Puisque notre étude se focalise sur la modélisation au plus haut niveau d'abstraction de ces systèmes, le troisième axe est lié à l'approche IDM qui offre un cadre conceptuel permettant de faciliter la modélisation, le développement modulaire et la réutilisation des différentes parties des applications étudiées. L'étude de l'intersection de ces trois axes de recherche à partir des travaux déjà existants et leur liens possibles est donc importante pour pouvoir atteindre notre objectif qui est la « *Modélisation à haut niveau du Contrôle dans des Applications de Traitement Systématique à Parallélisme Massif* ».

Applications de traitement systématique à parallélisme massif

Les applications de traitement systématique à parallélisme massif jouent un rôle crucial dans le domaine des télécommunications et des applications multimédia. Le développement de ces applications exige de grandes puissances de calcul afin de répondre aux différents besoins des utilisateurs. Leur principal objectif est d'assurer le traitement de grandes quantités d'information tout en exploitant au mieux le parallélisme présent dans leur description. Les applications industrielles d'imagerie embarquées, par exemple, sont basées sur des processus de filtrage en temps réel qui nécessitent une grande puissance de traitement parallèle pour atteindre les performances attendues.

L'étude des applications de traitement intensif exige des environnements de programmation spécialisés pour leur spécification, simulation, vérification et exécution, afin de réduire

les temps de développement et donc de mise sur le marché. Les techniques de modélisation et les modèles de calcul utilisés doivent alors proposer un cadre spécifique pour le développement de ces applications tout en respectant leurs caractéristiques de base. Une représentation de ces modèles pourrait être multidimensionnels, et riches en expression de parallélisme de données par la seule représentation de dépendances, temporelles ou spatiales, entre les différentes tâches de calcul. Parmi ces modèles de spécification, nous nous intéressons en particulier au modèle ARRAY-OL qui a été spécialement conçu pour la spécification des applications de traitement du signal intensif via l'expression de tout le parallélisme potentiel présent dans ces applications.

Systèmes réactifs synchrones

Le pouvoir d'expression du modèle ARRAY-OL n'est pas suffisant pour la spécification de certaines catégories d'applications de traitement intensif qui nous intéressent. Le modèle ARRAY-OL permet uniquement la représentation des traitements de données massivement parallèles, sans aucune prise en compte des notions de contrôle ou du changement de mode de fonctionnement. Cette limitation restreint l'étude des applications de traitement intensif à celles qui effectuent uniquement des calculs purs sur de grandes quantités de données. Cependant, une application de traitement du signal ou d'images suffisamment complexe, tels que les systèmes de radio logicielle, contient des parties de traitement numérique reliées par du contrôle. L'étude de ces systèmes nécessite alors le développement de nouvelles méthodologies de conception permettant de prendre en considération cette mixité de comportement. Ainsi, il s'avère important d'associer la notion de contrôle dans la description des modèles ARRAY-OL.

L'introduction du contrôle dans des applications de traitement de données correspond à l'association d'un comportement *réactif* à ces applications. Dans ce contexte, le système est non seulement descriptible par des relations transformationnelles, indiquant des sorties à partir des entrées, mais également par des relations entre les sorties et les entrées par l'intermédiaire de leurs combinaisons possibles dans le temps. En conséquence, la combinaison des descriptions comprenant des séquences complexes d'opérations, des actions, et des conditions de contrôle permet la définition du comportement global d'un *système réactif*.

La description du comportement des systèmes réactifs a été largement étudiée par la communauté du domaine. Plusieurs approches ont été proposées pour faciliter la conception sécurisée de ces systèmes, et en particulier de celle des systèmes hybrides mixant des traitements de données et du contrôle. Ces approches sont principalement basées sur une hypothèse *synchrone*, et proposent différents modèles de spécification permettant de mieux étudier et tester au plus haut niveau d'abstraction la fonctionnalité de ces systèmes avant leur réalisation définitive. Nous allons donc nous inspirer des études réalisées autour des systèmes réactifs pour introduire le contrôle dans le modèle de spécification ARRAY-OL.

Modélisation et approche IDM

Globalement, la complexité des systèmes parallèles et réactifs peut rendre la modélisation de leur comportement une activité difficile et exposée à des risques d'erreurs. Dans l'industrie, les soucis d'efficacité, de réutilisation, et de programmation sans erreur sont bien connus. D'une part, la première étape dans le développement de tout système informatique se résume en un ensemble d'idées et des besoins visant à définir exhaustivement les spé-

cifications de bases du système à réaliser. Ces besoins sont par nature non totalement formalisables car ils font partie du monde réel, et liés à des habitudes ou des opinions parfois mal conceptualisés. Réciproquement, le concepteur est étranger au monde du client, et doit prendre en considération tous ses besoins. Il s'agit donc de trouver un langage commun entre ces deux mondes différents. D'autre part, un défi supplémentaire est posé par le partage et la réutilisation du travail. Il est donc important de définir des formalismes communs permettant de faciliter l'échange et la réutilisation des différentes parties des systèmes étudiés.

Face à ces difficultés, et afin de pouvoir comprendre et agir sur le fonctionnement d'un système, il est nécessaire d'utiliser des techniques de modélisation et des langages communs permettant l'expression, l'échange, et la compréhension des principales fonctionnalités des systèmes étudiés. Dans ce domaine, l'*ingénierie dirigée par les modèles* (IDM) est considérée aujourd'hui comme l'une des approches les plus prometteuses dans le développement des systèmes. Cette technique offre un cadre méthodologique et technologique qui permet d'unifier différentes façon de faire dans un processus homogène. Son objectif est de favoriser l'étude séparée des différents aspects du système. Cette séparation permet d'augmenter la réutilisation et aide les membres de l'équipe de conception à partager et s'échanger leur travaux indépendamment de leur domaine d'expertise. Dans le cadre du travail présenté dans ce document, nous nous intéressons en particulier à la modélisation au plus haut niveau d'abstraction des systèmes parallèles selon une approche IDM. Il est à noter que cette approche fait généralement appel à l'utilisation de langages de modélisation communs et standardisés tel que UML pour mieux représenter et spécifier les différentes fonctionnalités du système étudié. Ce langage, largement utilisé dans les domaines industriels et académiques, offre une notation consensuelle permettant de privilégier la réutilisation et l'échange des différentes parties des systèmes étudiés.

1.2 Contribution

Notre contribution à la problématique d'introduction du contrôle dans les applications de traitement systématique à parallélisme massif, et en particulier, dans le modèle ARRAY-OL a suivi une démarche qui consiste en :

- la proposition d'une méthodologie de conception pour les systèmes hybrides basée sur la séparation contrôle/données,
- la proposition d'une approche permettant d'associer un comportement réactif aux applications décrites en ARRAY-OL, et
- la réalisation d'un profil UML permettant d'introduire nos résultats dans le cadre de l'environnement de développement GASPARD2.

Méthodologie de séparation contrôle/données

Comme nous nous intéressons dans ce travail à la conception des systèmes hybrides mixant des traitements de données et du contrôle, nous avons étudié les différentes approches existantes dans ce domaine. Cette étude nous a permis de constater que ces approches n'imposent aucune méthodologie de séparation entre la partie contrôle et les différentes parties de calcul. Ceci peut rendre difficile la compréhension de l'application, sa vérification, ainsi que l'échange d'informations et la réutilisation des différentes parties du

système. Nous avons donc proposé une méthodologie de conception basée sur une séparation claire entre la partie de contrôle et les parties de calcul. Cette méthodologie permet d'avoir un modèle de conception plus clair et plus facile à maintenir, et facilite l'étude séparée des différentes parties du système, notamment en ce qui concerne les processus de validation et de vérification formelle.

Introduction du contrôle dans ARRAY-OL

Le premier obstacle qui rend difficile la prise en compte des comportements réactifs dans un modèle ARRAY-OL est la différence de la sémantique de ce modèle avec celle des modèles de contrôle. Le comportement de contrôle est lié à un automate dont la sémantique est basée sur la notion de flot, tandis que le modèle ARRAY-OL est basé sur une sémantique de dépendance de données où le temps est banalisé. Nous avons donc proposé une solution permettant de mixer ces deux mondes différents et de définir les différents instants de prise en compte des valeurs de contrôle pour une application massivement parallèle.

Réalisation d'un profil UML

Dans le cadre du projet de développement de l'environnement GASPARD2, nous avons contribué à la réalisation d'un profil UML permettant la modélisation des comportements de contrôle tout en respectant la sémantique de base des modèles ARRAY-OL. Nous avons également proposé plusieurs améliorations de la première version du profil GASPARD2 en lui introduisant des nouveaux concepts et des contraintes OCL.

1.3 Plan

Le reste de ce manuscrit est organisé selon quatre parties principales : La première partie présente le contexte global et les problématiques abordés dans ce travail, et elle est composée de trois chapitres (chapitres 2, 3 et 4). Dans le chapitre 2, nous présentons la première partie du contexte de notre étude concernant les modèles de développement pour les applications de traitement systématique à parallélisme massif. Nous donnons un aperçu général sur les principaux modèles de calcul existants pour leur spécification, et nous décrivons en particulier le modèle ARRAY-OL qui est principalement conçu pour la description des applications de traitement du signal qui nous intéressent. À la fin de ce chapitre, nous posons la problématique principale de la thèse concernant le besoin de spécification des notions de contrôle dans les applications de traitement du signal intensif. Le chapitre 3 étudie la deuxième partie du contexte de notre travail concernant la conception des systèmes réactifs et hybrides. Dans ce chapitre, nous introduisons le principe de l'approche synchrone, et nous discutons les différentes approches existantes pour la conception et la modélisation des systèmes hybrides. Nous discutons également le lien entre la conception synchrone et les applications de traitement parallèle et intensif. Dans le chapitre 4, nous présentons la troisième partie du contexte de notre travail concernant la modélisation à haut niveau et la co-conception des systèmes sur puce selon l'approche IDM. Après avoir introduit les principes de base de cette approche, nous donnons un aperçu général sur l'environnement de développement GASPARD2, principalement conçu pour la co-conception des systèmes sur puce et basé sur le modèle ARRAY-OL, et sur la modélisation UML permettant de mieux représenter et spécifier les différentes fonctionnalités des systèmes sur puce. À la fin de ce chapitre, et comme

nous nous intéressons à la modélisation du contrôle, nous étudions le lien entre la modélisation UML et l'approche synchrone, notamment la modélisation des automates de contrôle en UML.

La deuxième partie présente notre méthodologie de séparation contrôle/donnée, et elle est structurée en deux chapitres (chapitres 5 et 6). Dans le chapitre 5, nous présentons le principe de notre méthodologie de séparation contrôle/données pour la conception des systèmes réactifs synchrones et hybrides. Nous discutons également l'intérêt et les avantages de cette méthodologie de séparation, notamment pour l'application des processus de vérification formelle. Le chapitre 6 illustre, à travers une étude de cas d'un système de limiteur et régulateur de vitesse intelligent avec GPS, les avantages de notre méthodologie de séparation, notamment en ce qui concerne les gains en temps de vérification formelle. Ce chapitre présente également les résultats d'un ensemble d'expérimentations et des tests réalisés sur le système étudié pour assurer son bon fonctionnement.

La troisième partie étudie l'introduction du contrôle dans le modèle ARRAY-OL et dans le profil GASPARD2, et elle est composée de trois chapitres (chapitres 7, 8 et 9). Dans le chapitre 7, nous étudions l'introduction du contrôle pour les applications massivement parallèles, et en particulier pour celles décrites selon le modèle de spécification ARRAY-OL. Nous montrons que l'introduction du contrôle dans ARRAY-OL nécessite la définition des différents moments de prise en compte des valeurs d'événements de contrôle. Pour ce faire, nous proposons un nouveau concept de *degré de granularité* permettant d'associer un comportement réactif aux applications parallèles en synchronisant l'arrivée des valeurs des données avec celles du contrôle, et nous discutons également la possibilité de prendre en considération plusieurs niveaux de contrôle. À la fin de ce chapitre, nous étudions la simulation de notre approche dans l'environnement de développement PTOLEMY II en se basant sur les concepts d'ARRAY-OL *for* PTOLEMY II et sur ceux du MODALMODEL. Le chapitre 8 présente la réalisation de notre approche dans le cadre du profil GASPARD2, et en particulier, la modélisation des concepts de contrôle tout en respectant la sémantique de base du modèle ARRAY-OL. Dans le chapitre 9, nous présentons une étude de cas détaillée d'un système de traitement de vidéo à multi-modes de fonctionnement. Cette étude permet d'illustrer l'utilisation des différents concepts introduits dans le profil GASPARD2, notamment en ce qui concerne la modélisation du comportement réactif des applications de traitement de données massivement parallèle.

Dans la quatrième et dernière partie nous concluons ce travail en présentant un bilan de notre contribution, et évoquant quelques perspectives envisageables.

Première partie

Contexte, Positionnement et Problématique

Chapitre 2

Traitement systématique et parallélisme de données massif

Ce chapitre présente la première partie du contexte de notre étude concernant les modèles de développement pour les applications de traitement systématique à parallélisme massif. Après l'introduction du principe général de ces applications, nous nous intéressons en particulier à celles de traitement du signal intensif, et nous présentons les principaux modèles de calcul existants pour leur spécification. Par la suite, nous étudions le principe du modèle ARRAY-OL, principalement conçu pour l'expression du parallélisme de données et des dépendances de tâches pour ce type d'applications. Nous montrons que la puissance d'expression de ce langage est suffisante pour la modélisation des applications de traitement du signal qui nous intéressent, ce qui justifie son choix. À la fin de ce chapitre, nous posons la problématique principale de la thèse concernant le besoin de spécification des notions de contrôle dans les applications de traitement du signal intensif.

2.1 Introduction

Les applications de traitement *systématique* et *intensif* sont de plus en plus présentes dans plusieurs domaines. Elles se retrouvent aussi bien dans le domaine du calcul scientifique que dans le domaine de traitement du signal intensif (télécommunications, traitement multimédia, traitement d'image et de la vidéo, etc.). Ces applications jouent un rôle crucial dans le développement des systèmes logiciels, et occupent une place importante dans le milieu de la recherche scientifique. Les caractéristiques principales de ces applications sont qu'elles effectuent une grande quantité de calculs réguliers, elles sont généralement complexes et opèrent dans des conditions temps réel.

Une application est dite *systématique* si son traitement consiste principalement en des calculs réguliers et indépendants, appliqués systématiquement sur des données en entrée pour fournir des résultats. Une application est dite *intensive* si elle opère sur une grande masse de données, et s'il faut fortement l'optimiser pour obtenir la puissance de calcul requise et répondre aux contraintes d'exécution du système [Bou02]. Généralement, les applications de traitement systématique et intensif nécessitent de grandes capacités de traitement de données faisant souvent appel à des techniques de calcul parallèle et distribué. Les difficultés de développement de ces applications sont donc principalement l'exploitation du parallélisme de données et de calcul, et le respect des contraintes de temps et de ressources d'exécution.

L'étude des applications de traitement systématique à parallélisme massif exige des environnements de programmation spécialisés pour leur spécification, simulation, vérification et exécution, afin de réduire les temps de développement et donc de mise sur le marché. Leur principal objectif est d'assurer le traitement de grandes quantités d'information tout en exploitant au mieux le parallélisme présent dans l'application. En outre, nous rencontrons les applications complexes de traitement systématique et intensif dans plusieurs domaines d'application tels que les réseaux de communication, le trafic aérien, l'automobile et les systèmes multimédias.

Depuis quelques années, le domaine des réseaux de communication, tels que les émissions satellites, les téléphones sans fil et les réseaux internet, subit une augmentation considérable du nombre d'utilisateurs et de la quantité de données à transporter [Owe04]. Le développement de ces système nécessite de grandes compétences en traitement systématique et intensif pour pouvoir répondre aux besoins d'accès rapide et direct aux informations. Ceci a conduit les compagnies de télécommunication de déployer encore plus de débit et de supporter un service temps réel sur leurs réseaux. Leur objectif est d'adopter les meilleurs

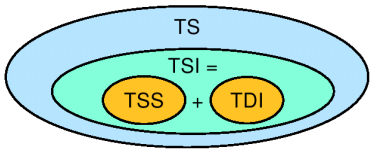
techniques de calcul pour pouvoir répondre aux exigences des consommateurs, notamment en terme de temps de réponse, de qualité et de quantité de services.

La sécurité et la sûreté des systèmes de transport, qu'il s'agisse du trafic aérien ou routier, constituent un aspect d'intérêt majeur pour l'industrie du transport. Dans le secteur spatial, la performance des systèmes de commande et de contrôle aérien ont comme principal objectif d'avoir plus de puissance de calcul afin de supporter des charges de circulation plus élevées et de maîtriser l'utilisation des technologies avancées [DDMS95]. L'amélioration du fonctionnement temps réel et de l'efficacité du transport aérien fait souvent appel à des algorithmes complexes, et nécessite la prise en compte d'une grande masse de données. Ces algorithmes sont relatifs à des applications de traitement systématique et intensif, et font souvent appel aux calculs parallèles et distribués pour améliorer leur capacité de traitement. De façon similaire, les applications de traitement systématique et intensif se situent au centre de développement des systèmes d'automobile. Ces systèmes sollicitent une grande quantité de traitement de l'information pour assurer la fiabilité et la sécurité de leur fonctionnement. Ils subissent également une grande évolution électronique et mécanique qui nécessite des algorithmes plus complexes sur une grande masse de données [Cha93]. L'industrie de l'automobile doit donc fournir un certain nombre de services nomades afin d'améliorer la qualité de leur service. Ainsi, le traitement intensif en temps réel joue un rôle important dans le domaine des systèmes de transport, ce qui nécessite le développement de solutions efficaces permettant une meilleure conception de ces systèmes tout en réduisant leur coût de développement.

Le besoin de calcul systématique et intensif est également indispensable dans le développement des applications multimédia qui nécessitent une grande capacité de stockage et de transmission de données. Dans ce domaine, l'utilisation d'images et de vidéo numériques a augmenté de façon considérable, et nécessite de plus en plus la prise en compte de grandes quantités d'information et des traitements de haute performance [ME05]. Ces applications ont de fortes contraintes temporelles, en particulier pour les applications interactives tel que la télévision en direct, qui doivent être prises en considération pour assurer la performance du système. Ainsi, les données manipulées par ces applications sont nombreuses et en voie d'explosion ce qui nécessite plusieurs processus de compression/décompression tout en assurant la fiabilité des informations traitées.

Les exemples présentés montrent le besoin d'utilisation de modèles de calcul bien efficaces, permettant la spécification, la simulation, et la vérification des applications de traitement systématique et intensif. Il est important de noter que le domaine de calcul systématique et intensif est très ambitieux, ce qui rend difficile, voire impossible, le développement de méthodes et de modèles généraux et efficaces permettant de prendre en considération toutes ces applications. Pour cette raison, nous focalisons notre étude sur un sous-ensemble précis de ce domaine relatif aux applications de *traitement du signal intensif*. Ces applications jouent un rôle important dans la description de plusieurs systèmes omniprésents dans notre vie quotidienne. Elles sont généralement *multidimensionnelles* dans le sens où elles manipulent une grande quantité de données multidimensionnelles structurées dans des tableaux, telles que les applications de traitement d'image et de la vidéo, et représentent un degré élevé du parallélisme de données et de calcul.

2.2 Traitement du signal intensif



*La partie de traitement du signal intensif est composée du traitement du signal systématique (*TSS*) et du traitement de données intensif (*TDI*) plus irrégulier. Le* TSS *correspond à la première phase de traitement des signaux et consiste principalement à l'application de filtres et à des traitements très réguliers (indépendants de la valeur des signaux) appliqués systématiquement aux signaux d'entrée pour en extraire les caractéristiques intéressantes. Celles-ci sont ensuite traitées par des calculs plus irréguliers (dépendants de la valeur de ces grandeurs) dans la phase de* TDI*. Ce schéma en deux phases se retrouve dans beaucoup d'applications de traitement du signal ou de l'image. En voici quelques exemples représentatifs.*

Récepteur de radio numérique. *Cette application en émergence fait appel à une partie de* TSS *consistant à la numérisation de la bande de réception, la sélection du canal et l'application de filtres permettant d'éviter les parasites. Les données fournies par ces traitements systématiques sont ensuite envoyées dans le décodeur dont le traitement est plus irrégulier (synchronisation, démodulation, etc.).*

Traitement sonar. *Une chaîne de traitement sonar classique se compose d'une première étape systématique : la veille bande large, suivie d'un traitement de données :la poursuite. La première phase prend en entrée les signaux produits par les hydrophones (microphones répartis autour du sous-marin) et, par une suite de traitements systématiques, produit des voies, couples (direction, intensité), représentant les échos captés. Ces échos sont ensuite analysés par la poursuite pour identifier et suivre au cours du temps les objets les produisant.*

***Encodeur/décodeur* JPEG-2000.** JPEG-2000 *est un nouveau format standard de compression d'image. Le fonctionnement de l'encodeur suit le même schéma en deux phases. La première partie (de prétraitement à la décomposition en ondelettes) est systématique. C'est dans la deuxième partie de l'encodage qu'apparaissent des traitements irréguliers (quantification, deux étages d'encodage). Le décodeur fonctionne exactement à l'inverse de l'encodeur et fait donc se suivre une phase de* TDI *et une phase de* TSS*.*

FIG. 2.1 – *Traitement du signal intensif*

Le traitement du signal est la discipline qui développe et étudie les techniques de traitement, d'analyse et d'interprétation des signaux. Un *signal* peut être défini comme le support physique de l'information. Le traitement du signal est alors un ensemble très vaste de techniques qui permettent d'extraire les éléments pertinents de l'information véhiculée, pour une application précise et dans un contexte donné. Ces techniques font largement appel aux résultats de la théorie de l'information, des statistiques ainsi qu'à de nombreux autres domaines de physique et de mathématiques appliquées[2].

Dans le contexte de notre travail, la partie de traitement du signal (TS) qui nous intéresse est sa partie la plus intensive, appelée *traitement du signal intensif* (TSI), définie par l'encadré

[2] `http://fr.wikipedia.org/wiki/Traitement_du_signal`

de la figure 2.1 [Bou02]. Les applications de traitement du signal intensif sont principalement basés sur des calculs numériques sophistiqués permettant d'améliorer les performances des systèmes étudiés. Bien que très diverses, les applications de traitement du signal réalisent toutes des traitements réguliers sur de grandes quantités de données, et certaines d'entre elles effectuent en plus des traitements irréguliers. Une autre caractéristique principale que partagent également ces applications est la manipulation de structures de données multidimensionnelles.

La régularité des traitements de la partie de traitement du signal systématique fait que l'ensemble des calculs effectués sur les données sont indépendants de leurs valeurs. Les valeurs des données traitées par les calculs sont ordonnées suivant des grandeurs physiques et se structurent en tableaux pouvant avoir des dimensions cycliques ou de taille infinie (pour représenter le temps par exemple). Les caractéristiques du traitement systématique montrent bien que la nature des calculs est moins importante que l'interaction de ces calculs avec les données. Une modélisation capable d'exprimer cette interaction est donc nécessaire pour mieux étudier l'application et effectuer de nombreux processus d'optimisation. Les enjeux ici sont les mêmes que ceux de la programmation classique : modéliser pour unifier et réutiliser. La seule différence vient du domaine de l'application lui même. Les techniques de modélisation et les modèles de calcul utilisés doivent donc proposer un cadre spécifique pour le développement d'applications de traitement du signal intensif à parallélisme massif. Ces modèles doivent être multidimensionnels, et riches en expression de parallélisme de données par la seule représentation de dépendances, temporelles ou spatiales, entre les différentes tâches de calcul. C'est dans ce contexte global que se situe notre étude.

2.3 Spécification multidimensionnelle et modèles de calcul pour le traitement du signal intensif

Pour étudier les applications de traitement du signal intensif, certains modèles de calcul ont été proposés pour la modélisation et la mise en œuvre de ces systèmes. Parmi ces modèles, nous citons MATLAB/SIMULINK[3], ALPHA [Mau89], MDSDF (MultiDimensional Synchronous Dataflow) [ML02], GMDSDF (Generalized MultiDimensional Synchronous Dataflow) [ML95], et le langage ARRAY-OL (Array Oriented Language) [DLB+95]. Malgré la diversité de ces modèles, leur occupation principale consistait à mieux modéliser les grandes quantités de données multidimensionnelles dont le but de faciliter la description et l'étude des applications de traitement intensif à parallélisme de données massif.

2.3.1 MATLAB/SIMULINK

MATLAB/SIMULINK est un langage de modélisation et de simulation pour les systèmes de traitement numérique dominés par les données.

Le logiciel de calcul MATLAB peut être vu comme un système interactif et convivial de calcul numérique et de visualisation graphique. La majorité de ces interactions (fonctions) sont basées sur un calcul matriciel simplifié. Nous distinguons plusieurs types de matrices : les matrices monodimensionnelles (vecteurs), les matrices bidimensionnelles, les matrices tridimensionnelles, etc. Grâce à ses fonctions spécialisées (analyse numérique, calcul matriciel, traitement du signal, etc.), MATLAB est considéré comme un langage de programmation

[3] `http://www.mathworks.com`

adapté pour les divers problèmes d'ingénierie. De plus, la méthodologie de conception basée sur MATLAB permet de réduire le temps de mise sur le marché en offrant le moyen pour la spécification des systèmes au plus haut niveau d'abstraction.

SIMULINK est l'extension graphique de MATLAB permettant de représenter les fonctions mathématique et les systèmes sous forme de *diagrammes de blocs* (ou diagrammes structurels). C'est un outil pour la modélisation, l'analyse, et la simulation d'une large variété de systèmes physiques et mathématiques, y compris ceux avec des éléments non-linéaires et ceux qui se servent du temps continu et discret. Le diagramme structurel dans SIMULINK met en évidence la structure du système et permet de visualiser les interactions entre les différentes grandeurs internes et externes. Les éléments qui composent le diagramme structurel représentent des opérations mathématiques, à savoir addition, soustraction, multiplication avec un coefficient, intégration et différentiation. L'environnement SIMULINK fournit également des outils pour la modélisation hiérarchique et la gestion des données qui permettent une représentation précise et concise de l'application indépendamment de la complexité du système étudié.

Pour l'expression des applications de traitement du signal et de la vidéo, un bloc spécifique, appelé *Signal Processing Blockset*, est introduit dans SIMULINK. Ce bloc offre une structure riche et efficace pour la conception, la simulation, et l'implémentation des algorithmes de traitement d'image et de la vidéo. Il inclut des primitives de base, et des algorithmes avancés conçus pour faciliter l'expression du comportement des applications de traitement d'image largement présentes dans le secteur industriel. Un exemple sur l'utilisation de SIMULINK pour la simulation de l'application d'un *Filtre de Kalman* est représenté dans [Gun01, Ala05]. Un autre exemple similaire pour la *Transformée de Fourier* est également disponible dans [Ade01].

Les logiciels MATLAB et SIMULINK sont généralement vus comme étant les outils quotidiens indispensables des techniciens et des ingénieurs dans diverses disciplines comme l'analyse numérique, le traitement du signal, et la modélisation de systèmes dynamiques (électricité, mécanique, thermique, etc.). Le choix d'utilisation de ces logiciels est devenu naturel puisqu'ils sont considérés comme des standards pour la conception des systèmes embarqués à traitement numérique et multidimensionnel. Cependant, MATLAB/SIMULINK est une pure création de numericiens et d'automaticiens, et n'a aucune des qualités informatiques requises pour la génération du code pour des systèmes informatiques parallèles. Ce sont des logiciels principalement conçus pour la simulation des applications de traitement numérique et ne prennent pas en considération l'expression et l'exploitation du parallélisme de données présent dans ces applications. Ainsi, la sémantique de SIMULINK est principalement basée sur un temps continu, et dépend du choix de la méthode de simulation effectuée. L'absence d'une sémantique claire, et la possibilité de non terminaison des programmes (boucles infinies, débordement de piles, etc.) rendent difficile, voire impossible, leur test et vérification formelle [CCM+03]. Les progrès des logiciels MATLAB/SIMULINK sont alors rapides et riches en expressivité, mais les tares initiales sont nombreuses pour les applications embarquées parallèles.

2.3.2 ALPHA

Le langage ALPHA, développé par Mauras en 1989 [Mau89], est un langage fonctionnel fondé sur le formalisme des systèmes d'équations récurrentes [KMW67]. Ce langage peut être vu à la fois comme un langage spécialisé pour la synthèse d'architectures systoliques, et

aussi comme un langage de parallélisme de données à « usage général » pour la description de haut niveau d'algorithmes de calcul réguliers (algorithmes d'algèbre linéaire, traitement du signal et d'image, etc.).

Le formalisme qui sous-tend le langage ALPHA est appelé le *modèle polyédrique*, et se trouve aujourd'hui, à la base de plusieurs méthodes de parallélisation automatique des boucles, et de la synthèse de réseaux systoliques. Dans ce langage, un algorithme est décrit par des équations sur des variables définies dans des domaines multidimensionnels. Par des transformations successives, telle que la parallélisation des instances, ces descriptions peuvent être raffinées jusqu'à leur interprétation par des outils de synthèse logique dans le but de générer des architectures VLSI[4]. Le langage ALPHA est donc proposé pour faciliter la synthèse d'architectures parallèles intégrées pour ces algorithmes afin de les exécuter en temps raisonnable (temps réel pour les applications de traitement du signal). Les transformations du langage ALPHA sont implémentées dans l'environnement MMALPHA[5] qui est une interface, principalement basée sur MATHEMATICA[6], pour la manipulation de ce langage.

La notion de temps, ou même d'exécution, n'existe pas en ALPHA : un programme décrit un ensemble de calculs. Les dépendances entre ces calculs imposent des conditions sur l'ordre dans lequel ils peuvent être exécutés, mais cet ordre n'est ni explicite ni nécessairement unique. Pour cette raison, ALPHA est un langage à *assignation unique* : pour tout point du domaine d'une variable, il existe une et une seule expression qui définit la valeur de cette variable à ce point.

Un programme en ALPHA est un *système* dont les variables sont définies à l'aide des fonctions de l'ensemble de points entiers d'un espace vectoriel vers un ensemble de valeurs[7] [de 97, dQRR99]. Les données manipulées par ALPHA sont multidimensionnelles : elles correspondent à des unions de polyèdres convexes. Leurs formes ne sont donc pas restreintes à de simples tableaux rectangulaires. Par exemple la déclaration suivante :

$$V : i, j | 1 \leq i \leq j; j \leq 3 \; of \; real;$$

déclare une variable V de type réel. Le domaine de cette variable est l'ensemble des points (i, j) dans le triangle : $1 \leq i \leq j; j \leq 3$, et qui représente la collection de valeurs suivantes :

$$\begin{array}{ccc} V_{1,1} & V_{1,2} & V_{1,3} \\ & V_{2,2} & V_{2,3} \\ & & V_{3,3} \end{array}$$

ALPHA se révèle donc être en mesure d'exprimer de façon simple des formes de données très complexes. Cependant, ce langage est basé sur des expressions affines, et s'avère incapable de gérer les accès cycliques indispensables pour la description de certaines applications de traitement du signal. En outre, La structure de données généralement manipulée par les applications de traitement du signal intensif sont de simples tableaux à plusieurs dimensions. Nous n'avons donc pas besoin de gérer des formes de données aussi complexes proposés par le langage ALPHA. En d'autres termes, la puissance d'expression du langage ALPHA est rarement bien exploitée dans le domaine du traitement intensif.

[4] Very Large Scale Integration : caractérise les circuits intégrés de très haute intégration.

[5] `http://www.irisa.fr/cosi/Alpha`

[6] `http://www.wolfram.com`

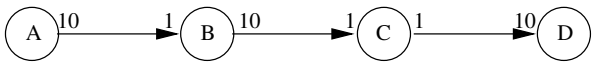
[7] Dans l'état actuel du langage, le type des valeurs peut être entier, réel ou booléen.

2.3.3 MDSDF : MultiDimensional Synchronous Dataflow

Le modèle MDSDF (MultiDimensional Synchronous Dataflow), proposé par Edward Lee en 2002 [ML02], étend le concept du modèle SDF (Synchronous DataFlow) [LM87b, LM87a] pour l'appliquer dans un contexte multidimensionnel. Le but est de permettre la spécification des applications de traitement du signal manipulant des données multidimensionnelles, tels que le traitement d'image et de la vidéo.

Le modèle SDF représente un cas spécial des modèles flot de données qui sont généralement utilisés pour décrire des applications de traitement du signal par des graphes, en représentant les fonctions par des nœuds et les données par les arêtes du graphe. L'ajout du terme « *synchrone* » implique que le nombre de données consommées et produites soit connu dès la conception de l'application, ce qui permet de réaliser des ordonnancements statiques. Le modèle SDF est intégré à PTOLEMY[8], l'environnement de modélisation et de simulation d'applications pour les systèmes embarqués.

En SDF, une application est décrite par un graphe orienté acyclique dont chaque nœud consomme des données en entrée pour produire des résultats en sortie. Dans PTOLEMY, ces données sont appelées jetons (*tokens*), et les nœuds sont appelés acteurs (*actors*). La figure 2.2 montre un exemple d'une application décrite en SDF. Les symboles associés aux entrées et aux sorties de chaque acteur dans un graphe SDF spécifient le nombre de jetons consommés ou produits par l'acteur en question. Ainsi, la relation entre les acteurs dans un graphe SDF doit satisfaire la relation suivante : $r_i O_i = r_{i+1} I_{i+1}$, où r_i représente le nombre de répétition d'un acteur i et O_i (resp. I_i) est le nombre de jetons produits (resp. consommés) par l'acteur i. Les répétitions des acteurs peuvent être regroupés dans un vecteur de la forme $\overrightarrow{r} = [r_1 \ \ r_2 \ \ ... \ \ r_n]$. Pour l'exemple de la figure 2.2, $\overrightarrow{r} = [1 \ \ 10 \ \ 100 \ \ 10]$ qui signifie que pour chaque exécution de l'acteur A, il y aura 10 exécutions de l'acteur B, 100 de l'acteur C et 10 de l'acteur D.



A, B, C, D : acteurs

FIG. 2.2 – *Exemple d'un graphe d'une application en* SDF

Le modèle SDF permet de modéliser facilement des applications basées sur des flots de données synchrones en spécifiant les dépendances entre les tâches. Cependant, ce modèle est réservé à l'expression des applications mono-dimensionnelles limitant ainsi le domaine des systèmes étudiés à celui manipulant des flots de données mono-dimensionnelles. Le modèle MDSDF est ainsi introduit pour résoudre ce problème et permettre la prise en compte des applications à flot de données multidimensionnelles. L'objectif de ce modèle est de donner un moyen pour la modélisation des traitements de données parallèles, et d'avoir une meilleure expression des dépendances de données entre les tâches.

Le fonctionnement de MDSDF est similaire à celui de SDF. Ainsi, il suffit juste de préciser pour chaque acteur le nombre de jetons consommés et produits sur chacune des dimensions du flot, sachant que ce nombre est représenté par un *n-uplet*. Dans ce cas, le nombre de répétitions de deux acteurs A et B d'un graphe MDSDF manipulant un flot de données de n dimensions est calculé comme suit : $r_{A,i} O_{A,i} = r_{B,i} O_{B,i} \ \ , i = [1, n]$.

[8] `http://ptolemy.eecs.berkeley.edu`

La figure 2.3 donne un exemple d'une application en MDSDF qui prend en entrée une image de 40×48 pixels et la divise en des blocs de 8×8. Dans cet exemple, le flot représenté est de forme bidimensionnelle, le premier acteur A produit un rectangle de jetons de taille 40 sur la première dimension, et de taille 48 sur la deuxième pour chaque itération. L'acteur B consomme sur les deux dimensions un rectangle de jetons de taille 8. Pour cet exemple, $r_{A,1} = r_{A,2} = 1, r_{B,1} = 5,\ r_{B,2} = 6$.

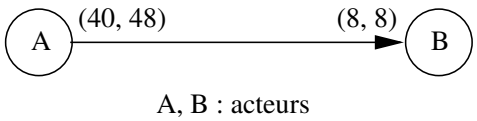


FIG. 2.3 – *Exemple d'un graphe d'une application en* MDSDF

L'utilisation de MDSDF permet une expression plus exacte d'une application multidimensionnelle, et rend possible la modélisation d'application qui ne pouvait être décrite en SDF. La figure 2.4 montre un exemple d'une application non représentable en SDF. Dans cet exemple, l'acteur A produit une colonne de deux jetons et l'acteur B consomme une ligne de trois jetons. Les deux jetons de la colonne ne seront donc pas consommés par la même itération du deuxième acteur comme le montre le graphe de dépendances. La modélisation de cette application en SDF n'est pas possible puisque ce dernier ne permet pas de spécifier le mode de construction du graphe de dépendances [ML96].

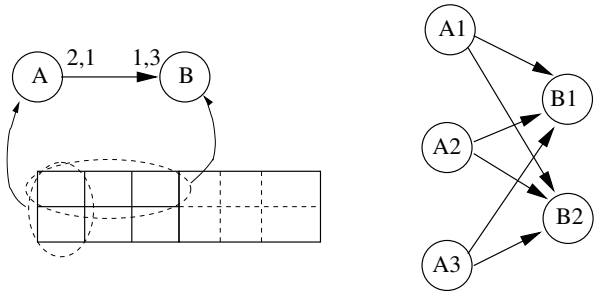


FIG. 2.4 – *Une application* MDSDF *non représentable en* SDF

Le modèle MDSDF rend alors possible l'utilisation de flot de données multidimensionnelles. Cependant, le nombre de dimensions du flot de données n'évolue jamais et il n'est d'ailleurs pas possible de créer ou de supprimer des dimensions. La création de dimension est utile pour certaines applications de traitement du signal comme dans le cas d'une FFT[9] qui crée une dimension fréquentielle. Une autre limitation dans le modèle MDSDF est que la consommation et la production des données doivent être parallèles aux axes. Pour résoudre ce problème, Murthy et Lee ont proposé une extension au modèle MDSDF appelé GMDSDF (Generalized MultiDimensional Synchronous Dataflow).

2.3.4 GMDSDF : Generalized MultiDimensional Synchronous Dataflow

Le but de GMDSDF [ML95] est de fournir un formalisme capable de modéliser des applications avec une consommation ou une production des données non parallèles aux axes.

[9] Fast Fourier Transformation

En GMDSDF, les jetons ne sont plus produits en ligne et en colonne comme avec MDSDF, mais ils sont placés sur des treillis de points. Seuls certains points du treillis sont consommés ou produits par les tâches de l'application. La forme et la manipulation de ces treillis sont faites par trois tâches spéciales appelées : *source*, *décimateur* et *expandeur*.

La tâche « source » sert à créer et à définir la forme du treillis. Cette tâche est associée à deux matrices : la matrice d'échantillonage (*sampling matrix*) et la matrice support (*support matrix*), notées respectivement V et W. La matrice d'échantillonage V permet de déterminer la forme du treillis en définissant les points qui en font partie, tandis que la matrice support W est utilisée pour déterminer quels points du treillis seront produits. Le calcul des positions des éléments du treillis s'effectue de la manière suivante [Mur96] :

- construire le « parallélogramme fondamental » (*fundamental parallelogram*) à partir de la matrice support W, en se basant à l'origine, et en prenant les vecteurs de la matrice W comme les deux premiers cotés. Tous les points à coordonnées entières se trouvant à l'intérieur de ce parallélogramme sont appelés les « points renumérotés » (*renumbered points*).
- La multiplication des coordonnées des « points renumérotés » par la matrice d'échantillonage V donne les coordonnées des points qui seront effectivement produits par la tâche « source ».

La figure 2.5 représente un exemple sur le fonctionnement de la tâche source. Dans cet exemple, les points du treillis sont générés par la matrice d'échantillonage $V = \left(\begin{smallmatrix} 1 & -1 \\ 1 & 2 \end{smallmatrix}\right)$, et la matrice support $W = \left(\begin{smallmatrix} 3 & 0 \\ 0 & 2 \end{smallmatrix}\right)$.

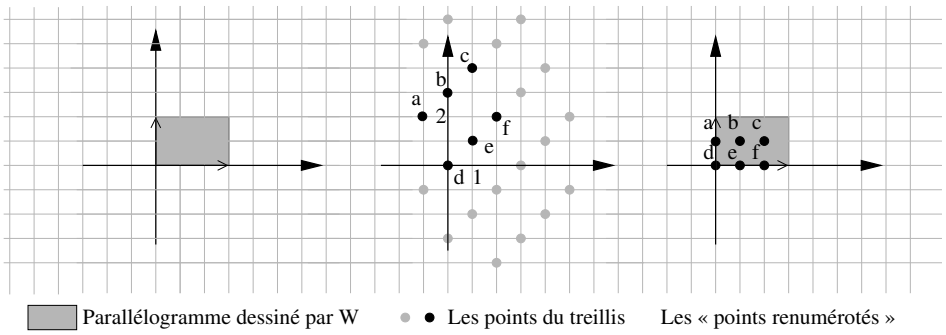


FIG. 2.5 – *Fonctionnement d'une tâche source*

Si la tâche « source » permet de créer des treillis, les tâches « décimateur » et « expandeur » permettent elles d'en modifier la forme : la première en enlevant des points, la deuxième en en rajoutant. Ces tâches sont respectivement définis par leur *matrice de décimation* (M) et *d'expansion* (L). Lee et Murthy notent respectivement V_e et V_f les matrices *sampling* d'entrées et de sorties. De la même façon, il note W_e et W_f les matrices *support*. Les liens unissant les treillis d'entrées de sorties sont données par les deux relations suivantes :

$$\begin{array}{lll} \text{« décimateur » :} & V_f = V_e.M & W_f = M^{-1}.W_e \\ \text{« expandeur » :} & V_f = V_e.L^{-1} & W_f = L.W_e \end{array}$$

Ainsi, un « décimateur » de matrice $M = \left(\begin{smallmatrix} 3 & 0 \\ 0 & 2 \end{smallmatrix}\right)$, utilisé sur un treillis de forme $M = \left(\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right)$, ne garde qu'un point sur trois sur la dimension horizontale et un point sur deux sur la verticale. Un « expandeur » de matrice $M = \left(\begin{smallmatrix} 3 & 0 \\ 0 & 2 \end{smallmatrix}\right)$ permet d'effectuer l'opération inverse.

Le fonctionnement global des autres tâches dans le modèle GMDSDF est similaire à celui du modèle MDSDF. Cependant elles n'utilisent plus un n-uplet comme en MDSDF mais une matrice *support*, et les points désignés ne sont plus nécessairement placés consécutivement sur une ligne ou sur une colonne, mais ils peuvent être choisis de manière non parallèle aux axes et avec un décalage.

Il est clair que le modèle GMDSDF est plus riche en expression que celui du MDSDF. Cependant, Edward Lee reconnaît lui même dans ses articles que l'utilisation de GMDSDF dans un environnement de développement ne serait pas sans poser de problèmes : « *...it is not clear at this point whether these principles will be easy to use in a programming environment* » [ML02]. Ainsi, le modèle GMDSDF oblige à définir, à un même niveau de description, les tâches spéciales de manipulation du treillis et les tâches ordinaires de manipulation des données. La manipulation des données sur les treillis se font de manière régulière, et le modèle GMDSDF ne peut pas s'appliquer lorsque le nombre de dimensions est supérieur à deux, ce qui limite le domaine des applications de traitement du signal à une catégorie bien particulière. Pour ces raisons, nous introduisons dans la section suivante un autre domaine multidimensionnel appelé ARRAY-OL qui ne présente pas ce genre d'inconvénients.

2.4 ARRAY-OL

ARRAY-OL (Array Oriented Langage) est un langage spécialisé dans la description d'applications de traitement du signal intensif. Ce type d'application est caractérisé par une manipulation de grandes quantités de données qui sont traitées par un ensemble de tâches de façon régulière. Le langage ARRAY-OL est basé sur la constatation que la complexité de telles applications vient des accès aux données (toujours des tableaux) et non des fonctions de calcul. C'est un langage à *assignation unique* où seules les dépendances de données sont exprimées et où les dimensions spatiales et temporelles des tableaux sont banalisées. Ainsi, le modèle ARRAY-OL est multidimensionnel et permet d'exprimer un parallélisme potentiel complet sur les applications, que ce soit un parallélisme de données ou un parallélisme de tâches.

Dans ARRAY-OL, chaque tâche de l'application consomme un ou plusieurs tableaux en le(s) découpant en « *morceaux* » de même taille appelés *motifs*. Un calcul est effectué sur ces motifs pour en produire d'autres qui seront rangés dans les tableaux résultats. La chaîne se poursuit, les tableaux produits étant à leur tour consommés. Les tâches dans ARRAY-OL sont reliées entre elles par des dépendances de données. L'expression de ces dépendances permet dans un premier temps de définir un ordre partiel minimal d'exécution des différentes tâches. Dans ce cas, une tâche ne peut produire ses résultats qu'à partir du moment où les données dont elle dépend sont présentes.

ARRAY-OL tire donc son nom du type de structure de données manipulés. Ce langage est basé sur la spécification des dépendances de données dans les applications, ce qui permet de déduire à la fois le parallélisme de tâches et le parallélisme de données. ARRAY-OL fournit donc un langage de spécification d'applications de traitement du signal intensif. À partir d'une telle spécification, il est naturel de chercher à exécuter l'application correspondante. Nous pouvons pour cela produire, à partir d'une description ARRAY-OL, un code source dans un langage cible (du type C, C++, etc.) exécutable sur une certaine plate-forme. Cette phase de transcription de l'application est appelée « *compilation* », ce terme est employé par abus de langage et ne doit pas induire le lecteur en erreur. Il s'agit en fait d'une « *traduction* »

de la description dans un langage de programmation et ce n'est que la compilation de ce dernier qui permettra l'exécution. Cette évolution consiste donc à remplacer le langage de programmation et la phase d'analyse des programmes par une spécification des algorithmes de plus haut niveau en facilitant la phase d'analyse de dépendances. La compilation du langage ARRAY-OL a été largement étudiée par Soula, Dumont *et al.* [Sou01, Dum05].

La description d'une application en ARRAY-OL fait successivement appel à deux niveaux. Le premier niveau, appelé *modèle global*, définit l'enchaînement des différentes parties de l'application dans son ensemble alors que le second niveau, appelé *modèle local*, précise les actions élémentaires à effectuer sur des éléments de tableaux et le parallélisme de données sur les différentes tâches.

2.4.1 Modèle global

Le modèle global permet de nommer et de définir les tableaux et les tâches de calcul. C'est un graphe dirigé acyclique où chaque nœud représente une tâche, et chaque arc représente un tableau multidimensionnel (figure 2.6). Le nombre de tableaux en entrée et en

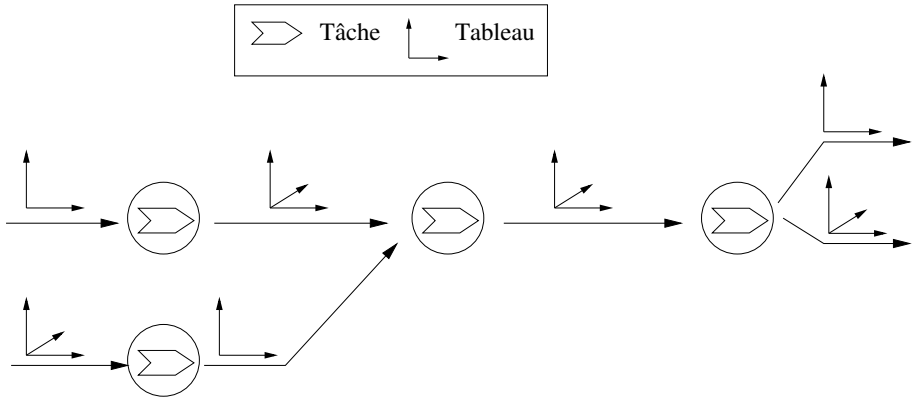


FIG. 2.6 – *Modèle global*

sortie est illimité. Il n'y a pas non plus de corrélation entre les nombres de dimensions de ces tableaux. Dans ce cas, il est possible pour une tâche de consommer deux tableaux bidimensionnels et de produire un tableau tridimensionnel. La création de dimensions est très utile, par exemple dans le cas d'une FFT qui crée une dimension fréquentielle. Cependant, la seule limitation sur les dimensions des tableaux est qu'il doit y avoir une seule dimension infinie par tableau qui représente généralement le *temps*. De plus, les tableaux utilisés dans ARRAY-OL sont considérés comme *toriques* dans le sens où la consommation ou la production de leurs éléments peut se faire modulo à leur taille.

Au moment de l'exécution, chaque tâche consomme un ou plusieurs tableaux, effectue un traitement quelconque sur leurs éléments, et produit un ou plusieurs tableaux résultats. Le nombre de tableaux consommés ou produits est égal au nombre d'arcs en entrée ou en sortie de la tâche. Le graphe relatif au modèle global représente alors un graphe de dépendance de tâches et non pas un graphe de flots de données.

À partir du modèle global, il est possible d'ordonnancer l'exécution des différentes tâches d'une application. Cependant, il est impossible d'exprimer le parallélisme de données présent dans cette applications puisqu'aucun détail sur les calculs réalisés n'est donné à ce niveau de spécification. Pour cette raison, l'introduction d'un autre niveau de spécification (le

modèle local) devient alors nécessaire pour pouvoir prendre en considération cette notion de parallélisme.

2.4.2 Modèle local

Le modèle local permet d'exprimer tout le parallélisme potentiel dans une tâche, et de définir l'interaction entre une tâche et ses tableaux opérandes et résultats. C'est un graphe dans lequel chaque tableau opérande et résultat est relié à la tâche de ce modèle en spécifiant comment les données sont consommées et produites par cette tâche. Dans ce modèle, une tâche est toujours composée d'un constructeur de répétitions, où chaque répétition est indépendante et appliquée sur un sous-ensemble fini de points des différents tableaux en entrée et en sortie (figure 2.7). Ainsi, pour chaque répétition, le rôle d'une tâche consiste à

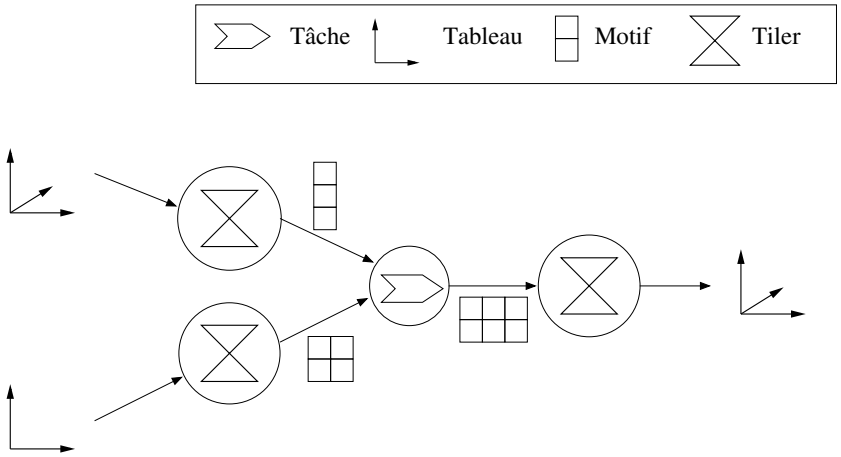


FIG. 2.7 – *Modèle local*

extraire des motifs à partir de chaque tableau en entrée, à appliquer une fonction de calcul qui produit des valeurs associées aux motifs en sortie, et à ranger ces derniers dans les tableaux résultats. La taille et la forme des motifs pour les différents tableaux en entrée ou en sortie peuvent être différentes. Cependant, pour les différentes répétitions sur un même tableau, les motifs doivent être réguliers (*ajustage du tableau*), et répartis régulièrement sur les tableaux (*pavage du tableau*).

La façon dont une tâche consomme et produit les tableaux qui lui sont associés peut être analysée à travers chaque couple (TÂCHE, TABLEAU). De tels couples sont appelés *demi-tâche*. Soit (T1, A1) un couple de demi-tâche, si A1 est un tableau d'entrée, T1 va alors prendre un sous ensemble fini des éléments de A1, puis va effectuer un traitement dessus. De façon similaire, si A1 est un tableau de sortie alors T1 va lui fournir un ensemble fini d'éléments qu'elle vient de calculer. Pour pouvoir exprimer des constructions *hiérarchiques*, chaque motif peut être lui même un tableau multidimensionnel. À chaque couple (TÂCHE, TABLEAU) est également associé un TILER. Ce dernier contient les informations nécessaires pour répondre aux questions suivantes : *comment le motif est-il construit ?*, et *comment la tâche passe-t-elle d'un motif à un autre ?* Ces informations sont :

- o : l'origine du motif référence
- $\overrightarrow{d}$: la taille des dimensions du motif

- P : une matrice appelée *matrice de pavage* (*paving matrix*) qui permet de décrire comment les motifs couvrent le tableau
- F : une matrice appelée *matrice d'ajustage* (*fitting matrix*) qui décrit comment remplir le motif par les éléments du tableau
- $\vec{m}$: la taille des dimensions du tableau

Pour énumérer les motifs, chaque demi-tâche dispose via son TILER d'une matrice de vecteurs de pavage et d'un point de départ appelé *origine*. À partir de ces deux éléments, il est possible d'identifier les origines de chaque motif à l'intérieur d'un tableau. Par exemple, si nous considérons un tableau bidimensionnels avec comme point d'origine $\binom{0}{0}$ et comme matrice de pavage $\left(\begin{smallmatrix}2 & 0\\ 0 & 3\end{smallmatrix}\right)$, alors les points origine des différents motifs du tableau sont montrés par la figure 2.8. Les coordonnées de ces points peuvent être calculés par un proces-

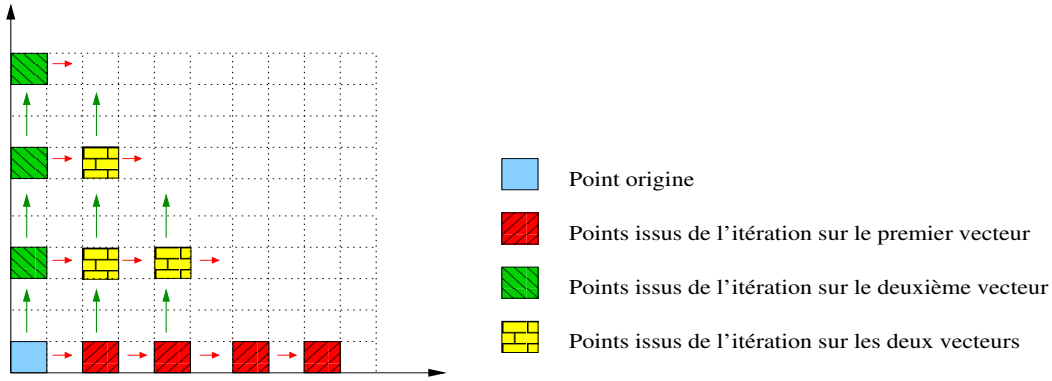


FIG. 2.8 – *Exemple de pavage*

sus répétitif correspondant à la somme des coordonnées de l'origine et d'une combinaison linéaire des vecteurs de pavage, le tout modulo la taille du tableau puisque les tableaux ARRAY-OL sont toriques comme il est exprimé par l'équation 7.1.

$$\forall \overrightarrow{x_q}, \overrightarrow{0} \leq \overrightarrow{x_q} < \overrightarrow{Q}, \overrightarrow{r_q} = (o + P \times \overrightarrow{x_q}) \mod \overrightarrow{m} \tag{2.1}$$

Dans cette équation, $\overrightarrow{x_q}$ représente le motif d'indice q, $\overrightarrow{Q}$ représente l'espace de répétition, et $\overrightarrow{r_q}$ représente l'origine du motif d'indice q.

La matrice d'ajustage est représentée par un ensemble de vecteurs où chaque vecteur est associé à une dimension du motif. Les vecteurs d'ajustage sont utilisés pour identifier les élément de tableau appartenant à un motif à partir du point d'origine. La figure 2.9 représente deux exemples simples utilisant un tableau unidimensionnel avec comme origine $\binom{0}{}$, et le motif associé est un tableau bidimensionnels de taille $\binom{3}{2}$. Cette exemple montre qu'il est possible d'avoir plus de dimensions dans le motif que dans le tableau. Dans le premier cas, la matrice d'ajustage est $\binom{1}{3}$. Chaque vecteur de cette matrice est utilisé pour remplir une dimension du motif. Dans le deuxième cas, la matrice d'ajustage est $\binom{2}{6}$, et les différents éléments du motif ne sont pas issus d'éléments consécutifs du tableau.

Ainsi, l'utilisation de la matrice d'ajustage s'effectue de manière similaire à celle de la matrice de pavage. Les éléments d'un tableau constituant un motif sont calculés comme la somme des coordonnées du premier élément de ce motif et d'une combinaison linéaire de la matrice d'ajustage, le tout modulo la taille du tableau puisque les tableaux ARRAY-OL sont

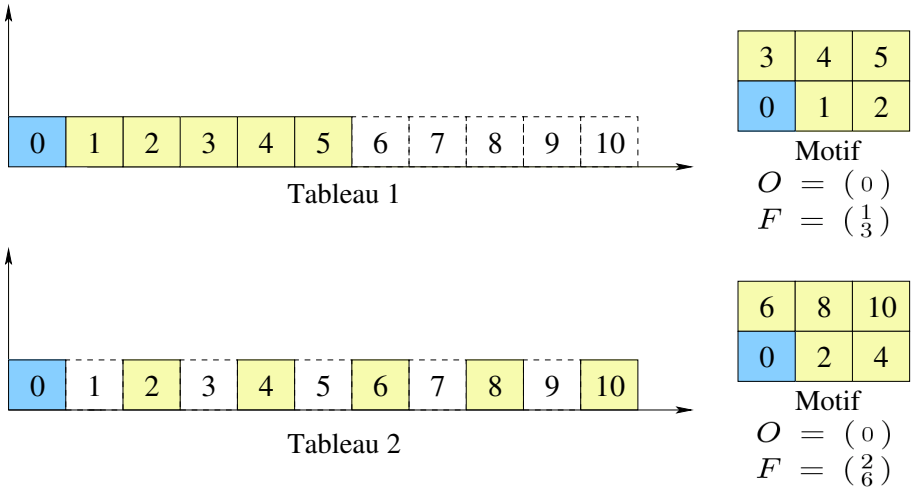


FIG. 2.9 – *Exemples d'ajustage*

toriques comme il est exprimé par l'équation 7.2.

$$\forall \overrightarrow{x_d}, \overrightarrow{0} \leq \overrightarrow{x_d} < \overrightarrow{d}, (\overrightarrow{r_q} + F \times \overrightarrow{x_d}) \mod \overrightarrow{m} \tag{2.2}$$

Dans cette équation, $\overrightarrow{x_d}$ représente l'élément d'indice d du motif.

La figure2.10 illustre la combinaison du pavage et de l'ajustage en montrant deux répétitions successives pour cinq cas différents :

1. Un cas basique, avec $o = \binom{0}{0}$, $\overrightarrow{d} = \binom{3}{2}$, $F = \binom{1\ 0}{0\ 1}$ et $P = \binom{3\ 0}{0\ 2}$
2. Des motifs qui ne sont pas nécessairement parallèles aux axes, avec $o = \binom{0}{0}$, $\overrightarrow{d} = \binom{2}{3}$, $F = \binom{1\ 1}{0\ 1}$ et $P = \binom{2}{0}$
3. Des motifs qui s'entrecroisent, avec $o = \binom{0}{0}$, $\overrightarrow{d} = \binom{2}{2}$, $F = \binom{1\ 0}{0\ 2}$ et $P = \binom{0}{1}$
4. Deux motifs successifs qui se chevauchent, avec $o = \binom{0}{0}$, $\overrightarrow{d} = (3)$, $F = (1)$ et $P = \binom{1}{0}$
5. Un tableau torique, avec $o = \binom{2}{0}$, $\overrightarrow{d} = (3)$, $F = (1)$ et $P = \binom{3\ 0}{0\ 1}$

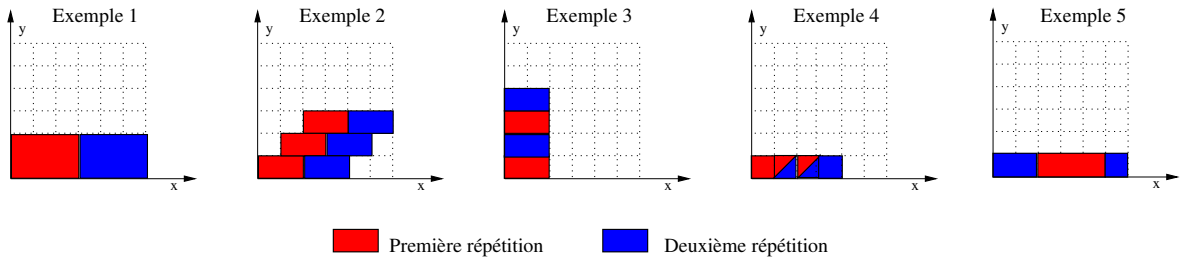


FIG. 2.10 – *Différentes répétitions*

Les deux niveaux, global et local, peuvent être hiérarchisés où toute tâche d'un niveau local peut être décrite comme un modèle global. Les tâches élémentaires du plus bas niveau sont implémentées dans le langage cible du compilateur. Une bibliothèque pour le TSS

comprendrait peu de ces tâches : des produits scalaires, des transformées de Fourier et des sommes.

Pour illustrer l'utilisation du langage ARRAY-OL, nous allons étudier la spécification d'un exemple académique d'un produit de matrices. Cet exemple permet de montrer la puissance du modèle local pour l'expression du parallélisme de données dans un algorithme. Soit A1 une matrice 3×5, et A2 une matrice 5×2. Nous calculons le produit $A1 \times A2 = A3$ avec A3 de taille 3×2. Le calcul d'un produit de matrice revient au calcul du produit scalaire de chaque ligne de A1 par chaque colonne de A2. Les différents produits scalaires sont indépendants et peuvent être alors effectués en parallèle.

Le langage ARRAY-OL permet d'exprimer tout ce parallélisme en identifiant les motifs nécessaires en entrées pour produire les motifs résultats. Soit la tâche élémentaire `ProduitScalaire` qui prend en entrée deux vecteurs de taille 5, et produit en sortie un scalaire correspondant au produit scalaire de ces deux vecteurs. Dans le modèle local, illustré par la figure 2.11, l'espace de répétition de la tâche `ProduitScalaire` est défini par le vecteur $[2, 3]$ (une tâche pour la production de chaque point du tableau en sortie). Cet exemple

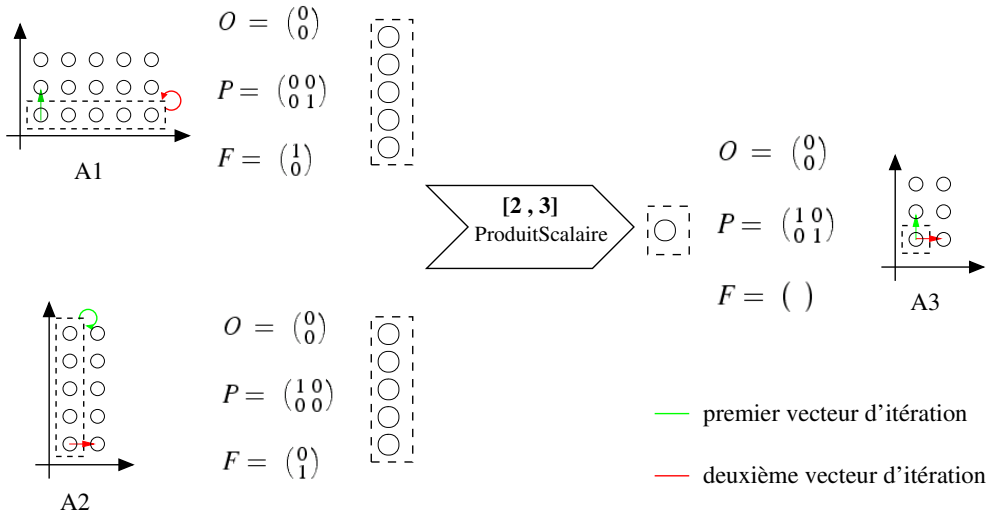


FIG. 2.11 – *Exemple d'un produit de matrice*

montre la possibilité d'exprimer tout le parallélisme potentiel dans une application, ce qui facilite son interprétation par des outils d'optimisation et de compilation.

La présentation que nous venons de faire montre qu'ARRAY-OL est parfaitement bien adapté à la modélisation des applications de traitement du signal systématique. Ainsi, le lecteur peut remarquer que seuls les modèles ARRAY-OL et GMDSDF permettant une description de haut niveau des applications multidimensionnelles pour le traitement du signal systématique. Bien que similaires, ARRAY-OL et GMDSDF n'autorisent pas la modélisation des mêmes applications. La désignation des points à consommer ou à produire ne se fait pas de la même façon dans ces deux modèles. De plus, l'utilisation des matrices support pour GMDSDF permet de désigner des ensembles de points de forme plus irrégulière qu'en ARRAY-OL. Cependant, nous ne sommes pas sûr que la désignation de tels ensembles de points représente un avantage quelconque pour GMDSDF. En effet, il est tout à fait possible de prendre une boite englobante pour effectuer la modélisation en ARRAY-OL et surtout il ne nous est jamais arrivé de rencontrer des tâches ayant besoin d'une manipulation de données

semblable.

Une comparaison plus détaillée, à travers un exemple sur l'analyse d'un signal vidéo, des deux modèles ARRAY-OL et GMDSDF est donnée dans [Dum05]. Cette comparaison a montré que, contrairement à GMDSDF, ARRAY-OL présente un ensemble d'avantages pour la description des applications de traitement du signal intensif. Parmi les avantages d'ARRAY-OL sur GMDSDF nous avons la possibilité de désigner des ensembles disjoints de points comme sur le troisième exemple de la figure 2.10 (page 28), et la possibilité d'utilisation des tableaux toriques comme sur le cinquième exemple de la même figure.

La différence la plus notable entre GMDSDF et ARRAY-OL est sans nul doute que certaines des démonstrations servant de support aux principes de GMDSDF ne sont valides que pour des applications ayant aux plus deux dimensions. ARRAY-OL fonctionne avec des applications multidimensionnelles sans aucune restrictions sur le nombre maximum de ces dimensions. Il est également à noter que le modèle MDSDF n'est pas limité du point de vue des dimensions, en revanche il ne permet qu'une manipulation rudimentaire des données, ce qui justifie notre choix pour l'utilisation du modèle ARRAY-OL.

2.5 Besoin du contrôle dans les applications de traitement du signal intensif

Nous avons présenté dans les sections précédentes le principe général des applications de traitement du signal intensif, et des différents modèles de calcul multidimensionnels existant pour leur spécification. La caractéristique principale de ces modèles est qu'ils permettent uniquement la représentation des traitements de données massivement parallèles, sans aucune prise en compte des notions de contrôle ou du changement de mode de fonctionnement. Il est clair que cette limitation restreint l'étude des applications de traitement du signal à celles qui effectuent uniquement des calculs purs sur de grandes quantités de données.

Cependant, une application de traitement du signal ou d'images suffisamment complexe contient généralement des parties de traitement numérique reliées par du contrôle. La programmation d'une application de compression/décompression d'image, la chaîne de codage vidéo à 34 Mbits/s, le système radio logicielle, ou encore le changement de processus par une télécommande représentent des exemples basiques de ce type d'applications.

Compression/décompression d'image. Un des premiers défis relevés par les industriels dans le domaine du multimédia numérique consiste à trouver un moyen de réduire au maximum les données informatiques décrivant les images. En effet, une image numérique non compressée pose dans la pratique, à cause de son poids important en octets, d'évidents problèmes de transfert et de stockage. La définition des capteurs augmentant sans cesse, et les fichiers deviennent de plus en plus encombrants. Afin de réduire la taille des images et de faciliter leur transfert et manipulation, plusieurs algorithmes de compression/décompression ont été proposés, et son bien connus dans la communauté de traitement d'images et de la vidéo tels que les formats JPEG (Joint Photographic Experts Group), MPEG (Moving Picture Experts Group), et JBIG (Joint Bi-level Image experts Group). La définition de ces algorithmes ainsi que leur utilisation pour certaines applications de traitement d'image font souvent appel à des traitements de contrôle. Un exemple d'utilisation d'automate de contrôle pour l'expression d'un algorithme de détection d'images biaisées (*skew*) dans

un fax ou télécopie (*facsimile*) est donné dans [BK97]. Un autre exemple sur l'utilisation d'automates à états fini dans la description d'un algorithme de compression/décompression d'images basé sur l'analyse de vaguelettes (wavelet) est donné dans [TNP+02].

Chaîne de codage vidéo à 34 Mbits/s. L'application de codage vidéo à 34 Mbits/s est constituée d'un ensemble de traitement numériques sur des séquences d'images de même dimension [Sma98]. Chaque image est divisée en un nombre fixe de macro-blocs sur lesquels des traitements de codage sont appliqués itérativement. Cette application contient une partie importante de contrôle reliant des opérations coûteuses en temps de calcul, et critiques du point de vue des ressources. Il est donc nécessaire d'utiliser de méthodologies bien adaptées pour la spécification de ces applications mixant des traitements numériques et du contrôle.

Radio logicielle. Dans un système radio classique, l'émission/réception est globalement assurée par des composant matériels spécifiques et adaptés aux systèmes auxquels ils sont destinés. Ceci rend difficile l'utilisation d'autres systèmes sans changer le matériel associé. Pour résoudre ce problème, la technologie de *radio logicielle* (*Software Radio* (SR) ou *Software Defined Radio* (SDR)) est introduite pour permettre le développement des radios principalement basés sur des logiciels facilement adaptables, et dans une moindre mesure sur des composants matériels. Cette technologie permet d'assurer l'inter-opérabilité entre différents types de radios ou appareils de communication dont l'exploitation est pilotée par un logiciel plutôt que par des dispositifs matériel. Le système de radio logicielle est principalement basé sur des applications de traitement du signal intensif. Ainsi, sa caractéristique adaptative lui associe un comportement de contrôle dont la fonctionnalité est de piloter les différents changements d'algorithmes du système. Cette partie de contrôle est généralement modélisée par des machines à état finis ou des diagrammes de flot de contrôle. Un exemple sur la représentation du système de radio logicielle adaptatif en utilisant des machines à états finis pour le changement d'algorithme de codage est donné par K. Ikemoto dans [Ike02]. Ainsi, dans [RRFL99], J. Razavilar *et al.* présentent la structure d'un algorithme de radio logicielle sous forme d'un diagramme FLOWCHART [NS73] pour la spécification des parties de contrôle dans le système.

Changement de processus par une télécommande. L'utilisation de la télécommande est considérée comme utile pour la plupart des systèmes technologiques afin de faciliter l'accès aux différents services disponibles proposés par ces systèmes. Parmi ces services, nous citons par exemple le changement de chaîne de télévision, le changement entre les mode de fonctionnement :TV, VCR, et DVD, et le fonctionnement du Zoom dans des logiciels d'aide à la vision [Cho05].

Les exemples présentés, ainsi que plusieurs d'autres [FW95, Coo95, Pal03, DaSHJJB05], montrent le grand besoin de spécification des comportements de contrôle dans les applications de traitement du signal et de l'image, et font souvent appel à des structures d'automates pour cette spécification. L'étude de ces systèmes nécessite alors le développement de nouvelles méthodologies de conception permettant de prendre en considération des applications de traitement du signal plus complexes mixant des traitement de données parallèles et du contrôle. Pour répondre à ces besoins, certaines études ont été lancées dont le but est de prendre en considération la spécification du contrôle dans des modèles de traitement de données intensif.

2.5.1 MATLAB/SIMULINK/STATEFLOW

Dans l'environnement MATLAB/SIMULINK, la prise en compte des aspects comportementaux et événementiels, nécessaire à la modélisation des machines à états, est introduite via le modèle STATEFLOW[10]. Ce modèle permet la représentation des machines à états parallèles et hiérarchiques, nécessaires pour la modélisation des systèmes plus complexes contenant des descriptions de contrôle, de surveillance, et de mode.

Le modèle STATEFLOW est principalement dérivé du formalisme des STATECHARTS de Harel [Har87], et suit sa sémantique de base. La principale différence est dans la nature du langage d'actions utilisé puisque dans le modèle STATEFLOW les actions sont plus étendues pour pouvoir référencer des fonctions MATLAB. Ainsi, STATEFLOW représente un outil interactif de conception de systèmes événementiels. Il est basé sur la théorie des machines à états finis, et permet de concevoir graphiquement des systèmes de logique de supervision ou de contrôle.

Dans l'environnement MATHWORKS, le modèle STATEFLOW est entièrement intégré à SIMULINK permettant ainsi de compléter cet environnement de simulation en prenant en compte des notions de contrôle. Dans ce contexte, un diagramme STATEFLOW peut être intégré dans un bloc de calcul SIMULINK. L'automate décrit en STATEFLOW est compilé en fonction C, puis intégré au graphe flot de données SIMULINK sous forme d'opérations dont les entrées sont les conditions des transitions et les sorties les actions à réaliser. L'utilisation de STATEFLOW avec SIMULINK permet alors la spécification des systèmes qui combinent des comportements logiques, tel que le changement de modes, avec des comportement algorithmiques, tel que le traitement du signal. L'intégration des diagrammes d'état dans SIMULINK permet également de lancer des simulations hybrides et d'étudier l'interaction entre ces deux modèles [Tiw02].

2.5.2 ALPHA/SIGNAL

Dans [SGG99], Smarandache *et al.* présentent la conception conjointe d'applications en utilisant les langages SIGNAL [BBGG85] et ALPHA. Dans leur approche, les traitements de données intensifs sont exprimés dans ALPHA, tandis que la partie contrôle, spécifiant des contraintes sur les horloges après une série de *transformations affines*, est exprimée dans SIGNAL. Le contexte de conception conjointe ALPHA/SIGNAL est basé, d'une part, sur l'utilisation du langage ALPHA pour la spécification des traitements de données numériques et des calculs sur des structures multidimensionnelles, et d'autre part, sur l'utilisation de l'environnement SIGNAL pour l'expression du contrôle et la validation de l'application à plusieurs niveaux d'abstraction.

ALPHA et SIGNAL disposent aussi d'un ensemble d'outils qui permettent la synthèse de l'application à partir de plusieurs niveaux de description. Ainsi, le problème de l'interfaçage entre ces deux formalismes a été étudié afin de pouvoir disposer d'une plateforme de spécification et de simulation conjointes du système complet. La communication entre les deux formalismes ALPHA et SIGNAL est réalisée par l'intermédiaire du langage C en deux phases principales [Sma98] : la première phase consiste à décrire le comportement de chaque système spécifié en ALPHA par une fonction en C, et la deuxième phase consiste à intégrer les fonctions C obtenues dans la spécification SIGNAL. Les langages ALPHA et SIGNAL sont ap-

[10] `http://www.mathworks.com`

parus ainsi comme des langages complémentaires pour la spécification d'applications contenant des traitements numériques et du contrôle.

2.5.3 PTOLEMY

Un autre exemple sur l'intégration du contrôle dans un modèle de calcul multidimensionnel peut être trouvé dans l'environnement PTOLEMY. En plus des modèles de spécification multidimensionnels MDSDF et GMDSDF, cet environnement fournit aussi un modèle de spécification pour les machines à état finis (FSM). Dans PTOLEMY, le domaine FSM représente un domaine particulier dans lequel les entités représentent les *états*, et les connexions représentent les *transitions* entre ces états. La puissance du modèle FSM dans PTOLEMY permet l'expression du contrôle logique, et la distinction des différents modes de fonctionnement dans un système.

Dans PTOLEMY II, le domaine FSM peut être combiné hiérarchiquement avec les autres domaines de calcul. Le formalisme résultant est appelé « $*$*charts* » (prononcé *star-charts*) [GLL99]. Contrairement aux automates hiérarchiques, $*$charts ne permet pas la description des modèles concurrents, mais montre plutôt la combinaison des FSMs hiérarchiques avec les modèles de calcul concurrents. Cette combinaison permet la modélisation et l'étude des systèmes mixant des traitements de données et du contrôle en spécifiant les différents modes de fonctionnement du système, et les conditions de changement entre ces modes. Le modèle de combinaison résultant est connu dans PTOLEMY II sous le nom du « MODALMODEL » [HLL+03]. Ce modèle permet une combinaison hiérarchique entre les états d'une FSM et d'un acteur du graphe flot de données. Cependant, la sémantique de la combinaison du modèle FSM a été étudié pour plusieurs modèles de calcul non multidimensionnels tels que SDF, DE (Discrete Event) et SR (Synchronous Reactive), mais elle n'a jamais été étudiée pour les modèles MDSDF et GMDSDF nécessaires pour la description des applications de traitement intensif multidimensionnel.

Plusieurs autres approches sur la combinaison des modèles de spécification du contrôle avec ceux du calcul intensif existent dans divers niveaux de conception, mais très peu d'entre elles ont couvert tout le cycle de développement des systèmes étudiés (de la spécification à la génération du code exécutable). Par exemple, dans [TNTBS00], une approche de simulation conjointe pour les applications numériques est introduite. Dans cette approche, les contrôleurs discrets sont modélisés dans le langage SIGNAL, tandis que les calculs continus sont représentés dans l'environnement SIMULINK. Un autre exemple peut être trouvé dans [CCM+03] qui étudie une approche pour la conception et l'implémentation des systèmes sur une plateforme distribuée en se basant sur SIMULINK pour la modélisation, et l'environnement SCADE[11] pour la validation.

2.6 Synthèse et conclusion

Dans ce chapitre, nous avons abordé la problématique de la spécification des applications de traitement systématique à parallélisme massif, et en particulier les applications de traitement du signal intensif. Ces applications nécessitent des calculs parallèles, distribués et à haute performances pour assurer leur bon fonctionnement, et optimiser leurs coûts de développement. Nous avons présenté les différents modèles de calcul existant pour la concep-

[11] `http://www.esterel-technologies.com`

tion de ces applications tels que MATLAB/SIMULINK, ALPHA, MDSDF et GMDSDF. Grâce à leur structure multidimensionnelle, ces modèles permettent l'expression des applications de traitement du signal intensif. Cependant, ils n'offrent pas de moyens efficaces pour l'exploitation du parallélisme de données, et peuvent avoir un certain nombre de contraintes limitant le domaine des applications étudiées, tels que le nombre de dimensions pouvant être pris en compte et les mécanismes d'accès aux données (cyclique/acyclique, etc.).

Pour faire face à ces problèmes, nous avons présenté le langage ARRAY-OL comme un modèle efficace et bien adapté pour la spécification des applications de traitement du signal intensif. Ce modèle répond bien à nos besoins de spécification puisqu'il permet l'expression de tout le parallélisme potentiel dans une application de traitement intensif. Il permet également des accès réguliers, cycliques et plus complexes aux données structurées dans des tableaux multidimensionnels. Nous avons donc considéré ce langage comme suffisant pour l'expression des applications de traitement du signal qui nous intéresse, ce qui justifie son choix.

À la fin de ce chapitre, nous avons posé la problématique principale de notre étude concernant le besoin de spécification des comportements de contrôle dans les applications de traitement du signal intensif. Nous avons montré que les travaux existant dans ce domaine proposent généralement la combinaison de deux formalismes différents, tels que SIMULINK/STATEFLOW et ALPHA/SIGNAL. Ceci nécessite la définition d'une interface de communication entre les deux formalismes, et le développement des techniques de transformation, afin de construire le système global, qui sont des tâches complexes pouvant conduire à des erreurs. Dans ce cas, le développeur doit bien maîtriser la sémantique des deux formalismes utilisés pour mieux définir leurs points de communication et assurer les fonctionnalités attendues du système étudié.

Le but de notre travail est donc de proposer un modèle commun introduisant la notion de contrôle dans le modèle ARRAY-OL pour permettre la prise en compte d'une plus grande catégorie d'applications, mixant des traitements de données parallèles et du contrôle. L'introduction du contrôle dans des applications de traitement de données correspond à l'association d'un comportement « *réactif* » à ces applications. Dans ce contexte, le système est non seulement descriptible par des relations transformationnelles, indiquant des sorties à partir des entrées, mais également par des relations entre les sorties et les entrées par l'intermédiaire de leurs combinaisons possibles dans le temps. En conséquence, la combinaison des descriptions comprenant des séquences complexes d'opérations, des actions, et des conditions de contrôle permet la définition du comportement global d'un *système réactif*. Pour cette raison, nous allons nous inspirer des études réalisées autour des systèmes réactifs pour introduire le contrôle dans le modèle de spécification ARRAY-OL. Le concept des systèmes réactifs ainsi que l'introduction du contrôle dans le modèle ARRAY-OL sont présentés dans les chapitres suivants (chapitres 3 et 7).

Chapitre 3

Systèmes réactifs synchrones et conception hybride

Dans ce chapitre nous présentons la deuxième partie du contexte de notre étude concernant la problématique générale de la conception et la modélisation des systèmes réactifs. Nous introduisons le principe de l'approche synchrone, et nous montrons que la plupart des systèmes réactifs sont hybrides dans le sens où ils mélangent des traitements de données et du contrôle. Nous citons et discutons ainsi les différentes approches existantes pour la conception et la modélisation de ces systèmes. Par la suite, nous présentons le lien entre la conception synchrone et les applications de traitement parallèle et intensif sur lesquelles se base notre travail. Cette étude nous permettra de nous inspirer de la technologie synchrone pour l'introduction du contrôle dans le modèle ARRAY-OL permettant de faciliter la conception et le développement des systèmes hybrides et parallèles.

3.1 Introduction

L'évolution technologique a conduit au développement de systèmes informatiques complexes, dont l'impact socio-économique est devenu de plus en plus important. De tels systèmes intègrent de nombreux composants logiciels et matériels et interagissent avec des environnements complexes. Ils sont devenus critiques tant par les conséquences de leur utilisation, que par la complexité de leur développement et de leur évolution. Ces systèmes possèdent aussi la caractéristique principale de fonctionner en temps réel, et doivent subir de sévères contraintes de leur environnement de fonctionnement.

Une classe importante des systèmes informatiques critiques est celle des *systèmes réactifs*, systèmes interagissant de façon continue avec leur environnement. En effet, de tels systèmes ne sont pas uniquement descriptibles par des relations transformationnelles, spécifiant des sorties à partir d'entrées, mais aussi par des liens entre sorties et entrées via leurs combinaisons possibles dans le temps. Dès lors, c'est la combinaison des descriptions englobant des séquences complexes d'événements, d'actions, de conditions et de flots d'informations qui permettent de définir le comportement global d'un système réactif.

Globalement, la complexité des systèmes réactifs découle essentiellement de la nature complexe des réactions aux différentes occurrences des événements discrets. Cette complexité peut rendre la modélisation du comportement de tels systèmes une activité difficile et exposée à des risques d'erreurs. Ce qui nécessitera impérativement l'introduction d'approches et de méthodes de conception rigoureuses permettant la vérification du comportement de ces systèmes par des méthodes formelles.

Face à ces difficultés, plusieurs industriels tels que AIRBUS[12], DASSAULT-AVIATION[13], ou encore SCHNEIDER ELECTRIC[14] se sont orientés vers une approche *synchrone* pour la conception d'équipements ou des sous-systèmes intégrés. Cette approche présente en effet l'intérêt d'asseoir la programmation des systèmes réactifs sur une sémantique déterministe, et sur des bases rigoureuses qui permettent, d'une part, la génération de code exécutable, et d'autre part, leur vérification formelle.

L'ensemble de langages et méthodes proposés pour l'étude des systèmes réactifs synchrones sont basés sur différents modèles, et peuvent être orientés *flot de contrôle* ou *flot de données* en fonction du comportement principal du système étudié qui peut être respectivement discret ou continu. Cependant, les systèmes embarqués les plus utilisés sont de nature

[12] `http://www.airbus.com`
[13] `http://www.dassault-aviation.com`
[14] `http://www.schneiderelectric.com`

hybride dans le sens où leur comportement représente un mélange de traitements de données et du contrôle. Il est donc important de proposer des méthodes efficaces prenant en compte ce mélange de comportement.

Produire des systèmes réactifs et hybrides répondant à des exigences de qualité données, à des coûts et dans des délais raisonnables, est donc un enjeu économique majeur et également un défi scientifique et technologique important. Pour répondre à ce défi il faut disposer entre autres d'outils et de méthodes pour améliorer l'efficacité du développement, et faciliter leur conception sécurisée. Dans ce qui suit, nous focalisons notre étude sur les systèmes réactifs synchrones. Pour cela, nous présentons plus en détails l'approche réactive synchrone, et nous donnons un aperçu des différentes approches de conception hybride existantes.

3.2 Systèmes réactifs synchrones

3.2.1 Systèmes réactifs

Selon D. Harel et A. Pnueli [HP85], les systèmes informatiques peuvent être catégorisés comme étant *transformationnels*, *interactifs*, ou *réactifs*, en fonction de leur degré d'interaction avec leur environnement, qui peut être un utilisateur humain ou un processus physique. Historiquement, les systèmes transformationnels et interactifs proviennent de la programmation classique, à laquelle sont ajoutés des mécanismes de gestion d'événements, de synchronisation et de programmation concurrente, alors que les systèmes réactifs sont issus des domaines du génie électrique, d'automatique et des systèmes embarqués.

Un **système transformationnel** est un programme classique généralement basé sur des structures de données complexes et des algorithmes. Un tel système effectue des calculs à partir des données fournies en entrée, pour produire des résultats en sortie avant de se terminer (voir figure 3.1). L'interaction avec l'environnement se limite donc à l'acquisition des données et à la production de résultats, comme dans le cas d'un compilateur par exemple. Un tel système n'a pas d'état interne, le résultat dépend ainsi uniquement des données en entrée (à moins d'utiliser des fonctions aléatoires).

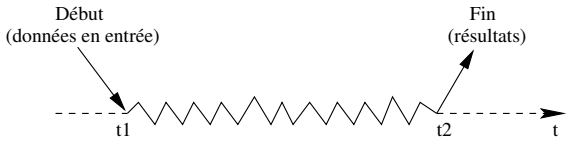


FIG. 3.1 – *Exécution d'un système de type transformationnel*

Les systèmes **interactifs** et **réactifs** sont des systèmes informatiques qui interagissent continuellement avec leur environnement, en produisant des résultats à chaque invocation. Ces résultats dépendent des données fournies par l'environnement lors de l'invocation, ainsi que de l'état interne du système. La différence entre ces deux types de système réside dans l'entité qui contrôle l'interaction. Dans un système interactif, comme une base de données, la prise en compte des requêtes et la production de réponses se font à l'initiative du système, qui impose ainsi son propre rythme comme le montre la figure 3.2. Un système réactif doit, quand à lui, être toujours en mesure de fournir une réponse immédiate quand l'environnement le sollicite. L'évolution d'un système réactif est donc une suite de réactions provoquées par l'environnement. Chaque réaction étant considérée instantanée par rapport

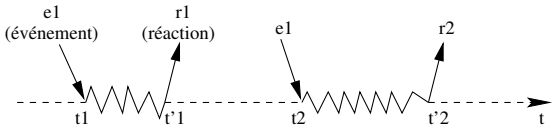


FIG. 3.2 – *Exécution d'un système de type interactif*

à l'échelle de temps propre à l'environnement comme il est illustré par la figure 3.3. Les interfaces homme-machine et les processus industriels représentent des exemples typiques des systèmes réactifs.

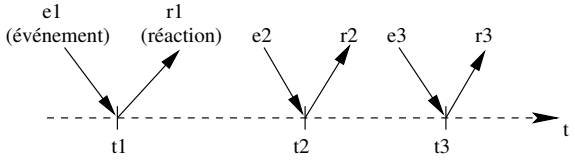


FIG. 3.3 – *Exécution d'un système de type réactif*

Les **systèmes réactifs** sont des systèmes à événements *discrets*. Leur comportement peut être représenté par une séquence de réactions à des *stimuli*. D. Harel et A. Pnueli ont donné aux systèmes réactifs l'image de boites noires qui réagissent de manière réflexe aux différents stimuli (figure 3.4).

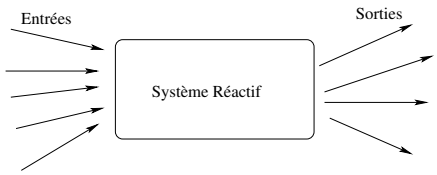


FIG. 3.4 – *Représentation d'un système réactif en tant que boite noire*

Pour mettre en évidence les aspects d'interaction d'un système réactif, nous pouvons considérer que l'exécution d'un tel système suit toujours le schéma de la figure 3.5. Dans ce schéma, la boucle est exécutée un grand nombre de fois, peut-être même indéfiniment. Puisque le système réactif est principalement basé sur la description des aspects d'interaction avec son environnement, son comportement reste toujours intéressant même si son exécution ne se termine pas [Mar97]. L'exécution de ce système est généralement divisée en plusieurs cycles correspondant à une échelle de temps discrète qui n'est pas nécessairement relative à l'échelle de temps physique.

Les caractéristiques critiques et importantes des systèmes réactifs font que la spécification, la programmation, et la vérification de ces systèmes est encore un problème de recherche important. La spécification du comportement des systèmes réactifs, qu'ils soient logiciels ou matériels, est complexe. Elle comporte des risques d'erreurs importants et difficiles à mettre en évidence. La complexité des systèmes réactifs découle essentiellement de la nature complexe des réactions conséquentes à des occurrences d'évènements discrets. La modélisation de ces systèmes est donc une activité difficile, et nécessite l'utilisation d'outils et de méthodes fiables et bien efficaces vu leurs fortes contraintes de fiabilité. Ces outils sont basés sur différents modèles selon leur hypothèses de base (synchrone ou asynchrone, flots

```
<Initialisations>
Tant que vrai
       <Acquérir des entrées>
       <Calculer des sorties>
       <Émettre les sorties>
```

FIG. 3.5 – *Schéma d'exécution d'un système réactif*

de contrôle ou flots de données, etc.), et utilisent des techniques formelles ayant une syntaxe bien définie accompagnée d'une sémantique rigoureuse. Dans ce qui suit, nous nous intéressons en particulier à la modélisation et l'étude des systèmes réactifs basés sur une *approche synchrone*.

3.2.2 Approche synchrone pour les systèmes réactifs

Le développement des applications réactives doit être un processus rigoureux, nécessitant des langages particuliers adaptés à la spécification de ces applications, et des outils efficaces pour leur analyse et vérification automatiques. Dans ce domaine, les premières approches utilisées pour la modélisation des systèmes réactifs ont été basées sur des structures d'automates, des langages de programmation classiques, et des langages asynchrones qui ont été largement étendus pour les adapter aux descriptions réactives. Cependant, l'extension des techniques traditionnelles n'apporte souvent pas de solutions satisfaisante à la programmation de systèmes réactifs. Il est donc largement préférable de développer des langages dédiés plutôt que d'adapter des langages existants. Ceci permettra de prendre directement en considération les spécificités de ces systèmes réactifs au plus haut niveau d'abstraction.

Au début des années 80, la famille des formalismes et langages *synchrones* a été une contribution très importante dans le domaine des systèmes réactifs. Les langages synchrones sont introduits pour faciliter la programmation des systèmes réactifs. Ils sont basés sur l'*hypothèse synchrone* qui ne prend pas en considération le temps de réaction des systèmes, et suppose que chaque réaction est *instantanée* et *atomique*. Cette hypothèse définit une échelle de temps logique et discret, constituée d'*instants* correspondant à chacune des réactions du système. Les évènements ayant déclenché la réaction sont considérés comme *simultanés*. De plus, le temps de réaction d'un composant particulier du système et le temps de communication entre les composants sont *nuls*. En d'autres termes, dans un langage synchrone, l'exécution du système se fait par *cycle* dont lequel les signaux émis pendant une réaction sont simultanés avec les signaux qui ont provoqué la réaction (la production de la réponse a lieu au même instant logique). Une réaction est donc par construction instantanée, ce qui évite les réactions concurrentes partielles d'un système, source d'indéterminisme. La figure 3.6 illustre cette propriété où plusieurs événements sont pris en compte à chaque réaction pour produire instantanément des résultats.

Dans [BB91], l'approche synchrone a été présentée comme une réponse au problème de la programmation réactive et temps-réel, via l'étude d'un exemple de contrôle d'automobile et d'un système de reconnaissance de voix. L'hypothèse synchrone a une vision abstraite des interactions et permet de simplifier l'expression des comportements réactifs. Cette hypothèse implique que les systèmes se composent très bien. Ils sont plus faciles à décrire, à simuler et à vérifier que les systèmes asynchrones.

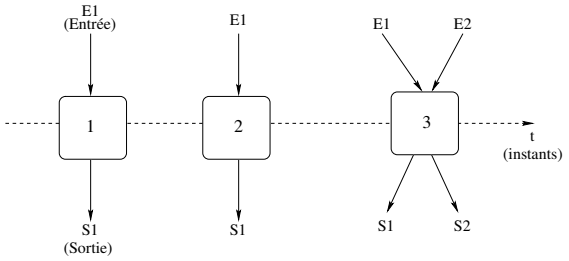


FIG. 3.6 – *Exemple d'exécution d'un système synchrone*

Les langages synchrones ont été introduits spécialement pour la programmation et la spécification des systèmes réactifs synchrones, et permettent de modéliser la manière dont le système réagit aux entrées émises par l'environnement en calculant et émettant des sorties. Un des objectifs de la programmation synchrone est d'apporter une « *programmation sûre* ». Les fondements mathématiques de ces langages leur confèrent élégance et rigueur, et permettent d'utiliser des méthodes formelles pour leur vérification. De plus, les langages synchrones sont généralement compilés dans des formats intermédiaires, ce qui leur permet une connexion à différents outils, notamment les outils de simulation, de vérification et de validation automatiques. Aujourd'hui, les langages synchrones sont considérés comme une technologie satisfaisante pour la modélisation, la spécification, la validation et l'implémentation des applications embarqués [BCE+03].

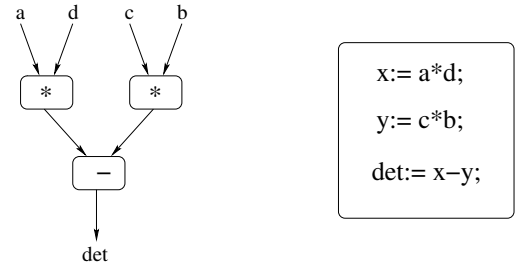
Il existe plusieurs langages synchrones permettant de programmer différents types de systèmes réactifs. Ces langages peuvent être classés en deux familles principales : les *langages déclaratifs* et les *langages impératifs*.

Les **langages déclaratifs** ou flots de données permettent de décrire les *systèmes réguliers*. Ils sont bien adaptés aux applications réactives manipulant essentiellement des flots de données (signaux continus) sous la forme de systèmes d'équations, comme dans un système de traitement du signal. Leurs tâches principales consiste à : consommer des données en entrées, effectuer des calculs, et produire des données en sortie. Parmi ces langages nous citons LUSTRE [CPHP87, HCRP91], SIGNAL [BBGG85, GGBM91] et SYNDEX [LSSS91, LS92].

Les **langages impératifs** ou flots de contrôles sont davantage appropriés pour la programmation des systèmes à *changements discrets* (signaux discrets) où le contrôle est dominant, comme dans le cas d'un distributeur de café. Leur objectif consiste à gérer le traitement des données en imposant l'ordre d'exécution des opérations (séquencement) et en choisissant une opération parmi plusieurs opérations exclusives (test et branchement). Les langages ESTEREL [FB91, BG92], ARGOS [Mar92, MR01], SYNCCHARTS [And96], et STATECHARTS [Har87] font partie de cette catégorie.

Afin d'illustrer les différences entre flot de contrôle et flot de données, la figure 3.7 représente un exemple de spécification d'un algorithme de calcul du déterminant d'une matrice carrée $A = \left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right)$, avec $det[A] = a * d - b * c$.

Chacun de ces deux types de langage, déclaratif ou impératif, peut être mieux adapté pour la spécification d'un système particulier de calcul ou de contrôle. Cependant, il est rare qu'un système ait un comportement exclusivement régulier ou exclusivement à changements d'états. Les systèmes embarqués les plus réalistes sont souvent hétérogènes au sens où ils regroupent des objets différents tels que les traitements de données et le contrôle. Ces systèmes *hybrides* peuvent être complètement décrits par des langages de flots de contrôles,



(a) Flot de données (b) Flot de contrôle

FIG. 3.7 – *Expression de calcul du déterminant d'une matrice*

mais les dépendances de données entre les différentes opérations ne peuvent pas être clairement spécifiées. Similairement, il est possible de décrire complètement ces systèmes dans un langage de flots de données, mais le contrôle est caché dans les dépendances de données, ce qui rend difficile la spécification des tests et des branchements nécessaires pour l'optimisation et la vérification de ces systèmes. Pour ces raisons, et compte tenu du fait que les systèmes hybrides sont souvent spécifiés par des partenaires différents, cette hétérogénéité de style induit souvent une hétérogénéité de langages, et il est donc important d'introduire de nouvelles techniques et outils permettant de prendre en compte cette diversité pour mieux étudier ces systèmes hybrides mixant des traitements de données et du contrôle.

3.3 Conception des systèmes hybrides

Les systèmes hybrides sont définis comme étant des sous-systèmes continus et discrets interagissant entre eux. Le fonctionnement de ces systèmes fait intervenir des aspect continus, modélisés en physique et automatique par des équations différentielles, et des aspects discrets, modélisés en informatique par des systèmes de transition et automates à états. Cette mixité pose le problème d'un découplage et d'un interfaçage permettant de s'abstraire, dans l'étude d'un aspect, de l'autre, tout en tenant compte des interactions entre les deux.

Généralement, les systèmes hybrides spécifient des processus continus (ou réguliers) pilotés par un contrôleur (ou superviseur) à événements discrets du type automates à états finis, Réseau de Petri ou Grafcet. En d'autres termes, les systèmes hybrides permettent de modéliser les systèmes discrets qui évoluent dans un environnement continu. Ces systèmes sont omniprésents dans notre vie quotidienne, ce qui a rendu leur étude un sujet de recherche important et intensif pendant toutes ces dernières années [IEE98, Aut98, SJSL00, IEE00]. Dans ce domaine, une importance particulière est donnée à la représentation unifiée des systèmes hybrides en se basant sur des modèles mathématiques rigoureux. Certains problèmes classiques tels que l'analyse, la stabilité et la sûreté de fonctionnement ont été étudiés, et des outils et méthodes de développement ont été proposés. Cependant, puisque la classe des problèmes des systèmes hybrides est extrêmement large, il est très difficile de concevoir une stratégie générale et efficace pour assurer le bon fonctionnement de tous ces systèmes.

Dans notre étude, nous nous intéressons uniquement à l'étude des systèmes réactifs basés sur l'approche synchrone. Comme nous l'avons introduit dans la section 3.2.2, les deux familles de langages synchrones sont principalement dédiées à la spécification des systèmes réactifs ayant un comportement exclusivement continu ou exclusivement discret. L'utili-

sation séparée de ces langages n'est donc pas suffisante pour la description des systèmes réactifs hybrides, ce qui rend nécessaire l'introduction de nouvelles approches spécifiques permettant de prendre en compte cette mixité de comportement.

Dans le domaine synchrone, plusieurs approches *multi-formalismes* ont été proposées pour permettre l'étude des systèmes réactifs hybrides, et d'aller au-delà des frontières posées par les langages synchrones spécifiques. Parmi ces approches, nous citons l'approche *multi-langages*, l'approche *transformationnelle*, et l'approche *multi-styles*.

3.3.1 Approche multi-langages

Comme son nom l'indique, l'approche *multi-langages* combine les deux types de langages, impératif et déclaratif, pour la description des comportements hybrides. Dans cette approche, la partie continue du système est décrite dans un langage déclaratif tel que LUSTRE, tandis que sa partie discrète est décrite dans un langage impératif telque ARGOS. De façon générale, la combinaison de ces deux styles de programmation est basée sur un mécanisme d'édition de lien. Cette approche permet une bonne réutilisation des codes déjà existants où chaque sous-système du système global peut être spécifié en utilisant le langage le mieux adapté, qui correspond aux diagrammes d'états ou aux flots de données. Cependant, cette approche oblige l'utilisateur à connaître assez finement les différents langages utilisés, ainsi que la description du système dans plusieurs langages rend souvent difficile la compréhension du comportement global. De plus, l'édition de lien de deux langages différents peut engendrer des problèmes de sémantique, et ne permet pas d'assurer que l'ensemble des codes générés satisfait la spécification globale du système étudié.

La combinaison des langages LUSTRE et ARGOS, proposée par M. Jourdan et al., est un exemple de l'approche multi-langages qui permet d'associer un nœud LUSTRE à un état d'ARGOS [JLMR94]. Très schématiquement, l'objectif de cette approche est d'introduire une structure de contrôle hiérarchique dans un langage déclaratif de façon à permettre l'écriture de programmes à la fois impératifs et déclaratifs. L'idée consiste à définir un langage unificateur dénommé ARGOLUS dont la sémantique est exprimée, dans un style commun, par l'union des sémantiques de LUSTRE et d'ARGOS [Jou94]. Dans le langage résultant, les constructeurs des deux langages LUSTRE et ARGOS sont réellement mixés et non seulement juxtaposés. Le point important ici, est que la fusion de ces deux formalismes est faite au niveau *source* dans le sens où elle n'utilise aucun type d'édition de lien pour établir un code exécutable à partir de la compilation séparée des deux langages. Le code résultat est alors obtenu en utilisant la structure d'un des deux langages pour enrichir l'autre langage. Cette approche peut donc éviter le problème d'édition de lien pour différents styles de programmation, mais pas celui de la maîtrise de plusieurs langages, ni de la définition d'une sémantique de base globale.

Cette démarche a été complétée par A. Poigné et al. dans un outil dénommé SYNCHRONIE qui associe les langages LUSTRE, ESTEREL, et ARGOS dans un même environnement de spécification [PMM+98]. L'objectif de cette approche est également d'offrir un cadre formel qui permet la construction de spécifications synchrones mélangeant des traitements de données et du contrôle. Pour ce faire, la solution retenue par A. Poigné, voisine de celle développée par M. Jourdan, consiste à traduire chaque bloc LUSTRE, ESTEREL, et ARGOS dans un format commun qui repose lui-même sur une sémantique synchrone.

Les travaux de M. Jourdan et A. Poigné apportent une réponse intéressante au besoin de la programmation multi-langages. Cependant, ils reposent sur une même démarche qui

passe par la construction d'un nouveau langage synchrone, par fusion de LUSTRE, ESTEREL, ou ARGOS, compilé ensuite dans un code synchrone unique et centralisé. En conséquence, ces approches souffrent des mêmes limites de la programmation multi-langages telle que la maîtrise de plusieurs langages, et en particulier répondent mal aux exigences de la compilation séparée.

Une autre approche de conception multi-formalismes a été présenté dans [BBG+99] en utilisant les langages ESTEREL, LUSTRE, et SIGNAL. Ce mécanisme est principalement basé sur une représentation commune de ces langages, et leur co-simulation via un format commun pour les langages synchrones. Une génération automatique du code distribué a été aussi expérimentée dans le contexte de l'environnement graphique du langage SIGNAL. Cependant, cette approche nécessite en plus de la maîtrise des différents langages utilisés, une bonne connaissance du format commun utilisé pour leur fusion.

En 1997, un langage multi-formalismes, dénommé LEA, est introduit pour la programmation des systèmes réactifs synchrones et hybrides [Bon97, HP98]. Ce formalisme est basé, d'une part, sur l'intégration des trois langages synchrones : LUSTRE, ESTEREL, et ARGOS, et d'autre part sur une sémantique de la concurrence et de la communication faiblement synchrone [Bon95]. Cette approche est à la fois similaire aux travaux mentionnés ci-dessus puisqu'elle associe les langages LUSTRE, ESTEREL, et ARGOS pour la conception de systèmes hybrides, et plus simple puisqu'il s'agit plus d'une approche par coopération de programmes que par fusion de langages. L'ambition de LEA est d'offrir les moyens syntaxiques et sémantiques qui permettent la description d'une vue global d'un système réactif hybride, tout en respectant ses exigences de déterminisme et de modularité. Cependant, l'approche sur laquelle est basée LEA n'offre pas de moyens directs qui permettent la vérification modulaire d'une spécification [Bon98].

L'approche multi-langages s'avère intéressante pour la conception séparée des différentes parties du système hybride. Or, la compilation séparée des sous-modules d'un programme synchrone, c'est à dire la capacité à pouvoir transformer séparément et indépendamment chaque sous-module en code exécutable qui sera intégré en suite au sein de l'architecture cible à l'aide d'une opération d'édition de lien, est un problème encore mal maîtrisé. L'utilisation de plusieurs langages ne permet pas d'assurer que l'ensemble des codes générés satisfait la spécification globale. Ainsi, puisque les spécifications des différentes parties du système peuvent être modifiées, le cycle de développement de l'application est plus long par rapport à l'utilisation d'un seul langage de spécification. La description d'un système global amené à être structuré et réalisé par plusieurs formalismes nécessite alors une approche différente.

3.3.2 Approche transformationnelle

Une autre approche utilisée pour la conception des systèmes hybride est l'approche *transformationnelle*. Cette approche consiste à utiliser les deux types de langage de spécification, impératif et déclaratif, mais avant la génération du code global, les spécifications impératives doivent être transformées en spécifications déclaratives ou vice-versa, permettant ainsi de produire un code unique au lieu de plusieurs. Cette approche permet d'assurer que le code global est conforme à la spécification du système. Cependant, la définition des règles de transformation demeure une tâche difficile et peut induire à plusieurs erreurs. Un exemple de l'approche transformationnelle est données dans [PS03] où les spécifications de contrôle décrites en SYNCCHARTS sont transformées vers le langage flots de données SYNDEX. Cette

approche permet l'optimisation du code généré pour les systèmes complexes, et peut réduire leur cycle de développement.

Une démarche similaire est présentée dans [BRG+01] qui étudie une méthode de transformation d'une spécification impérative en une spécification déclarative au travers les langages STATECHARTS, ACTIVITYCHARTS, et SIGNAL. Cette traduction rend possible la spécification multi-formalismes, et fournit un support pour l'inter-opérabilité des langages.

Dans [MH96], les auteurs présentent une méthode pour la compilation du langage ARGOS vers une machine MEALY implicitement représentée par un ensemble d'équations booléennes. L'implémentation du compilateur produit du code dans un format équationnel commun pour les langages synchrones. Ainsi, la transformation d'un langage déclaratif tel que LUSTRE vers ce même format permet de fusionner des langages impératifs et déclaratifs.

Une autre approche transformationnelle pour la conception des systèmes hybrides peut être trouvée dans l'environnement de développement SCADE [Tec03a]. Cet environnement est basé sur l'utilisation des deux langages synchrones ESTEREL et LUSTRE, où le code ESTEREL est transformé en LUSTRE avant tout processus de simulation, de vérification, ou de génération de code. Dans ce qui suit, nous présentons un peu plus en détail cet environnement de développement puisque nous allons l'utiliser pour la représentation de certains concepts de notre étude.

SCADE (Safety Critical Application Development Environment) est un environnement de développement graphique commercialisé par Esterel Technologies[15]. Cet environnement a été défini pour aider et assister le développement des spécifications logicielles formelles, ainsi que pour la production automatique du code embarqué pour ces applications. SCADE est composé de plusieurs outils tels que l'éditeur graphique, le simulateur, le vérificateur, et le générateur de code qui transforme automatiquement les spécifications graphique en code C. Le code généré est correct par construction [Dio04], ce qui permet d'éliminer la phase du test de code qui peut prendre jusqu'à 50% du coût de développement.

SCADE est construit sur de bonnes fondations formelles : il est déterministe, facile à apprendre, et fournit des solutions efficaces pour le développement des systèmes complexes et critiques. Cette environnement est utilisé dans de grands projets européens en avionique (Airbus A340-600, Airbus A380, Eurocopter, etc.), et également considéré comme étant une norme standard pour ce domaine.

L'environnement SCADE est basé sur une approche synchrone dans laquelle le programme examine cycliquement ses entrées pour calculer et fournir des résultats. Pendant ce cycle, le programme ne voit pas les changements qui peuvent arriver à son environnement. Ces changements ne seront pris en compte que dans le prochain cycle d'exécution. Le langage sur lequel est basé SCADE est une spécification graphique flot de données qui peut être traduite en code LUSTRE ayant une sémantique unique et précise. Ainsi, pour la représentation et l'étude des systèmes hybrides, SCADE utilise deux formalismes de spécification : les *diagrammes de blocs* pour le contrôle continu, et les *machines d'états* pour le contrôle discret.

Le « contrôle continu » sous entend le prélèvement des valeurs en entrée à des intervalles de temps réguliers, l'exécution des calculs sur ces valeurs, et la production des résultats en utilisant souvent des formules mathématiques complexes. Dans SCADE, le contrôle continu est spécifié graphiquement en utilisant des *diagrammes de blocs* (block diagrams) comme il est montré par l'exemple de la figure 3.8. Les blocs correspondent aux fonctions de calcul mathématique, aux filtrages et aux délais, tandis que les arcs représentent les flots de données

[15] `http://www.esterel-technologies.com`

entre les différents blocs. Les données circulent de façon continue entre les blocs permettant ainsi le calcul des valeurs en sortie en fonction de celles en entrées. Il est aussi à noter que les blocs dans SCADE sont complètement hiérarchiques. A chaque niveau de description, un bloc peut être composé d'un ensemble de petits blocs inter-connectés par des flots locaux. En comparant cette notion avec d'autres formalismes de diagrammes de blocs, la hiérarchie dans SCADE est purement architecturale dans le sens où elle n'implique pas des règles complexes d'évaluation hiérarchique.

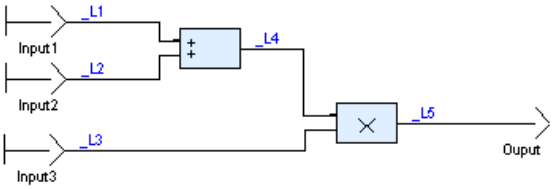


FIG. 3.8 – *Exemple Simple d'un diagramme de bloc dans* SCADE

Le « contrôle discret » correspond aux différents changements de comportement du système en fonction des évènements qui peuvent être externes ou internes au programme. Ce contrôle est généralement représenté par des machine à états qui ont été largement étudiées dans les 50 dernières années. Elles sont basées sur de bonnes fondations mathématiques et leur théorie est bien connue. Cependant, dans la pratique et pour de grandes applications, l'utilisation des machines à état peut induire des problèmes d'explosion combinatoire et devient très complexe à gérer. Pour limiter ce problème, un concept riche de machines à états *hiérarchiques* a été introduit. La structure de machine à états hiérarchique utilisée dans SCADE est dénommée SSM (Safe State Machines) [And03] qui représente une extension du langage de programmation ESTEREL et du modèle SYNCCHARTS. La figure 3.9 représente un exemple simple d'une SSM dans SCADE. Dans cet exemple, l'automate est composé de

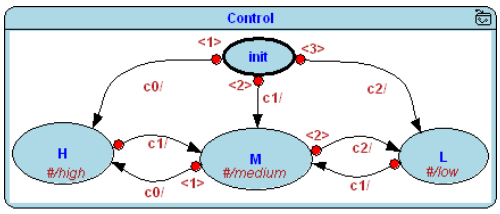


FIG. 3.9 – *Exemple Simple d'une* SSM *dans* SCADE

quatre états init, H, M et L, avec init comme état initial. Le passage entre les différents états de l'automate est fait en fonction de la valeur des conditions c_k, $k = 0..2$. Les chiffres spécifiés sur les transitions permettent de représenter la priorité entre les différentes transitions en sortie du même état pour assurer un comportement déterministe de l'automate.

Puisque la plupart des applications contiennent un mélange du contrôle discret et continu. SCADE permet de coupler le traitement de données et les machines à états pour faciliter la conception de tels systèmes. Dans ce contexte, les signaux discrets envoyés par une SSM se transforme en un flot de données booléen qui sera facilement consommé par les diagrammes de blocs.

L'approche transformationnelle est efficace pour la description des systèmes réactifs combinant des traitements de données et du contrôle. Cependant, elle nécessite toujours

la maîtrise de deux formalismes différents, et une bonne définition des règles de transformation entre les deux styles de langage.

3.3.3 Approche multi-styles

Il existe des systèmes dont le comportement est principalement régulier, mais peut passer brutalement d'un comportement régulier à un autre. Ce sont des systèmes à *modes de fonctionnement*. Par exemple, pour un avion, nous pouvons distinguer les modes `au sol`, `décollage`, `en vol`, et `atterrissage` [Rém01]. Ces modes représentent des comportements différents du système. Ils sont complètement *exclusifs*, dans le sens où ils se succèdent séquentiellement sans aucun recouvrement. La méthode la plus adaptée pour la description de ce type de systèmes consiste à utiliser une approche *multi-styles* permettant de décrire avec un seul langage, qui comporte des structures adaptées, les différents comportements du système. L'un des avantages de cette approche est qu'il n'y a qu'un seul langage, ce qui facilite son apprentissage et sa maîtrise. Dans ce domaine, les *automates de modes* [MR98] représentent une contribution remarquable qui consiste à ajouter une structure impérative dans le langage flots de données LUSTRE. Dans notre étude, nous nous intéressons en particulier à cette catégorie de système à modes de fonctionnement en se basant sur le principe des automates de modes qui ont prouvé leur efficacité pour la conception de ces systèmes.

Les automates de modes représentent des modèles de programmation basés sur des opérations sur les automates, inspirées du langage ARGOS, et des équations flots de données en LUSTRE. La notion de modes de fonctionnement correspond au fait que plusieurs définitions (équations) peuvent exister pour la même sortie du système dans des périodes de temps disjointes, et en décomposant sa spécification en plusieurs tâches « indépendantes ».

Un automate de modes est un automate d'entrée/sortie, avec un ensemble fini d'états appelés *modes*. À chaque instant, le système doit être dans un et un seul mode, et peut changer son mode si un événement apparaît. Pour chaque mode, une fonction de transfert détermine les valeurs des flots de sorties en fonction de celles des flots d'entrées. Les automates de modes permettent d'augmenter la lisibilité et facilitent la compréhension du comportement du système puisque leur structure spécifie clairement les différents modes de fonctionnement et les conditions de commutation entre ces modes.

Pour mieux expliquer ce concept, considérons l'exemple d'un système de valve présenté dans [Rau02] (figure 3.10.(a)). Ce système peut être soit `ouvert`, ou `fermé`. Dans ces deux modes de fonctionnement, le système peut aussi être en mode `coincé`. Le système passe du mode `ouvert` vers le mode `fermé` (resp. de `fermé` vers `ouvert`) s'il n'est pas coincé, et si l'événement `fermer` (resp. `ouvrir`) apparaît. Il peut aussi être en mode `coincé` si l'événement `échouer` apparaît. Si la valve est ouverte, son flot de sortie est égale à son flot d'entrée. Autrement, son flot de sortie est nul. La figure 3.10.(b) décrit l'automate de modes relatif à ce système.

De nombreux autres travaux sur l'utilisation du concept de modes de fonctionnement ont été introduits pour faciliter la conception des systèmes réactifs hybrides. Un premier exemple de ces travaux est celui proposé par J. Colaço *et al.*, et qui consiste à définir un sous ensemble des SSMs dans Scade, appelé SSM♯, prenant en compte des descriptions flot de données en se basant sur le principe des automates de modes [CP03]. Nous trouvons également dans [CPP05] une proposition pour étendre les langages synchrones flots de données, tel que LUSTRE, par des structures impératives sous forme d'automates à la SYNCCHARTS. De point de vue syntaxique, cette extension consiste à définir une machine à états hiérar-

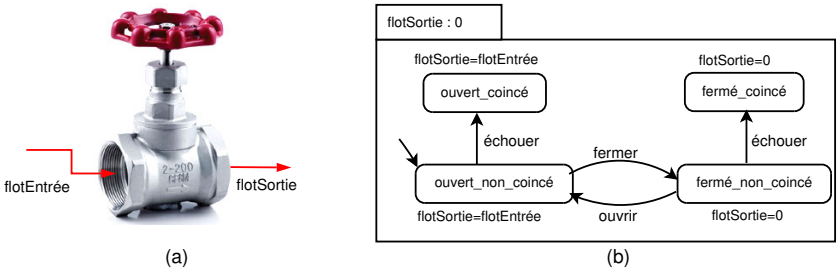


FIG. 3.10 – *Exemple simple d'un système de valve*

chique qui peut contenir à chaque niveau de la hiérarchie une description d'un ensemble d'équations de calcul. Cette extension est également basée sur l'utilisation des horloges permettant ainsi de définir plus précisément la sémantique du modèle flot de données de base. Cette approche permet donc de mixer des équations flot de données avec une structure d'automate en proposant un concept est plus riche que celui des automates de modes en ce qui concerne la forme des transition. Cette étude a été principalement intégrée dans l'outil industriel SCADE-V6, et dans le compilateur de LUCID SYNCHRONE [Pou06]. Une autre approche similaire pour l'extension des langages flot de données synchrones, tel que LUSTRE, par une structure de machine à états plus générale appelée « *parametrized state machines* » est également proposée dans [CHP06]. Cette approche est principalement basée sur la gestion des signaux évalués (valued signals) à la ESTEREL pour permettre la spécification des systèmes mixant des traitements de données et du contrôle. La sémantique de cette proposition a été clairement définie et introduite dans le compilateur de LUCID SYNCHRONE.

Une autre proposition moins récente et similaire au concept des automates de modes peut être trouvée dans le concept du MODALMODEL du projet PTOLEMY [HLL+03] (référencé dans la section 2.5, page 30). Cette proposition permet l'utilisation des machines d'états finis hiérarchiques avec différents modèles de concurrence. Dans ce modèle, il est possible d'écrire des programmes flots de données attachés aux états, à n'importe quel niveau de la hiérarchie. Le concept de ce modèle est similaire syntaxiquement à celui des automates de modes. Cependant, la sémantique est différente puisque, dans le modèle PTOLEMY, la transmission des informations d'un composant flots de données à un autre doit être construite explicitement en utilisant un mécanisme de communication.

3.4 Traitement intensif à parallélisme massif et approche synchrone

Nous avons présenté dans les sections précédentes l'utilisation de l'approche synchrone, et de la combinaison de modèles continus et discrets pour la spécification des systèmes réactifs hybrides. Ces travaux représentent une contribution importante dans le domaine de la conception hybride, et ils sont utilisés par certains industriels pour l'étude et la spécification de leur systèmes critiques. Cependant, dans le domaine réactif synchrone, les approches existantes ne prennent pas en considération la modélisation des applications de traitement intensif et parallèle sur des données multidimensionnelles, et par conséquent, les approches hybrides existantes ne peuvent pas être directement appliquées pour ce type d'applications.

Nous avons montré dans la section 2.5 (page 30) que la plupart des applications de traitement du signal intensif et parallèle peuvent contenir, dans la description de leur comportement, des parties de traitement de contrôle qui doivent être bien spécifiées pour garantir le comportement global du système. En se basant sur les résultats de la technologie synchrone, nous pensons qu'il est intéressant de s'inspirer des concepts de l'approche synchrone et hybride pour le développement de méthodes de conception plus puissantes prenant en compte les applications de traitement de données parallèles. Ceci permettra, d'une part, de bénéficier de la simplicité et la puissance d'expression des modèles synchrones, et d'autre part, de tirer profit des résultats obtenus et des outils développés autour de ces systèmes, notamment les outils de vérification formelle et de génération automatique du code.

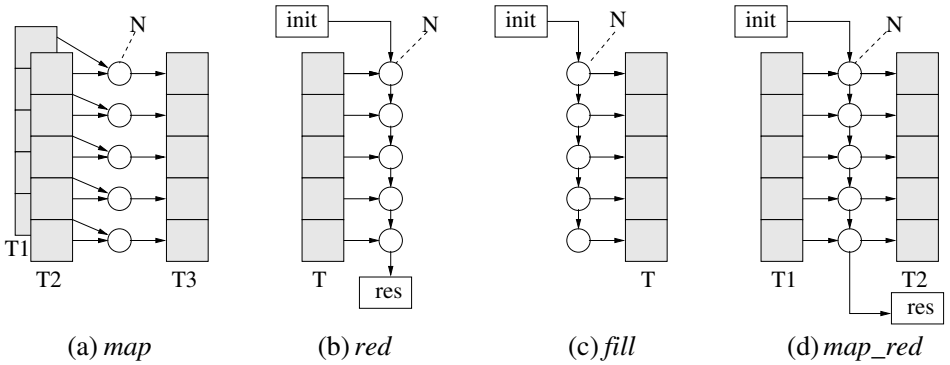
Dans la littérature, un exemple sur le lien entre la modélisation synchrone et la spécification multidimensionnelle a été étudié via l'introduction des tableaux dans le langage de programmation synchrone LUSTRE. La première réalisation de cette étude a été proposée dans LUSTRE V4 pour faciliter la description de circuits réguliers dans l'environnement POLLUX développé pour la conception à haut niveau d'architectures matérielles [RH91]. Les techniques de compilation mises en place pour cette version étendue du langage LUSTRE ont été particulièrement adaptées à la programmation de circuits. Dans ce contexte, le compilateur traduit un tableau de n éléments en n variables indépendantes, ce qui peut être naturel pour la programmation de systèmes matériels qui se ramènent au cas d'une programmation sans tableaux, mais pas forcément adapté pour la programmation de systèmes logiciels. La compilation par exemple d'un tableau de n éléments booléens par le compilateur LUSTRE[16] donne n variables booléennes indépendantes. Le code C généré pour ce programme aura donc n variables indépendantes à la place du tableau ce qui n'est pas efficace dans la programmation logicielle.

L'utilisation du langage LUSTRE dans le développement logiciel a montré que les primitives de manipulation des tableaux ne sont pas pratiques pour le développement logiciel. Ce qui a poussé au développement des *itérateurs* de tableaux dont le but est de fournir des outils plus faciles à manipuler, et pour lesquels le code généré est plus efficace [Hal99]. La manipulation des tableaux dans LUSTRE se fait via quatre itérateurs : `map`, `red`, `fill` et `map_red`.

1. `map` : l'opération de mapping consiste à appliquer la même fonction à tous les éléments d'un ou plusieurs tableaux. Les résultats obtenus spécifient les éléments d'autres tableaux (voir figure 3.11.(a))
2. `red` : l'opérateur de réduction permet de calculer un accumulateur en parcourant un ou plusieurs tableaux (voir figure 3.11.(b))
3. `fill` : l'opérateur `fill` permet de remplir un ou plusieurs tableaux en effectuant des itérations sur une fonction à partir d'une valeur initiale (voir figure 3.11.(c))
4. `map_red` : l'opérateur `map_red` représente un itérateur générique duquel tous les autres éléments peuvent être déduits (voir figure 3.11.(d))

La compilation des tableaux dans LUSTRE a été optimisée par L. Morel [Mor02] dans LUSTRE-V6 en proposant l'utilisation des itérateurs de tableaux pour la génération de boucles lors de la compilation. Cette proposition ne cherche pas à remplacer LUSTRE-V4 pour la programmation de circuits, mais plutôt à donner des outils assez puissants pour la programmation de systèmes logiciels.

[16] `http://www-verimag.imag.fr/~raymond/tools/lv4-distrib.html`



(a) *map* (b) *red* (c) *fill* (d) *map_red*

FIG. 3.11 – *Les itérateurs de tableaux dans* LUSTRE

Le langage LUSTRE est donc en mesure de fournir un support pour la manipulation des tableaux multidimensionnels. Cependant cette utilisation est restreinte aux quatre itérateurs présentés, et aux tableaux dont la taille est connue statiquement. Il n'est donc pas possible de prendre en considération des tableaux de taille infinie. Cette dimension infinie est généralement utilisée pour la représentation du temps et permet la prise en compte de tout le parallélisme potentiel présent dans l'application. Ainsi, à la différence du langage ARRAY-OL qui permet une expression graphique et simplifiée des motifs dans un tableau, la manipulation des éléments d'un tableau dans LUSTRE doit être explicitement exprimée par des indices.

Un autre exemple sur la combinaison d'un modèle de calcul multidimensionnel avec le paradigme synchrone peut être trouvé dans la spécification conjointe ALPHA/SIGNAL [SGG99] (présentée dans la section 2.5.2, page 32). Dans cette approche, la description des structures de données multidimensionnelles est faite dans ALPHA, tandis que le paradigme synchrone est spécifié via le langage synchrone SIGNAL.

Nous rappelons que dans le cadre de notre travail, nous nous intéressons en particulier à l'étude des applications de traitement du signal intensif à parallélisme massif, et que la problématique principale consiste à intégrer les concepts de contrôle dans la spécification de ces applications. Nous avons présenté dans le chapitre 2 que le langage ARRAY-OL permet de mieux exprimer les applications de traitement du signal intensif et répond bien à nos besoin de spécification. Cependant, ce langage ne donne aucun moyen pour la spécification des comportements de contrôle et des changements de modes de fonctionnement. La question à poser est alors : *comment introduire le contrôle dans le modèle* ARRAY-OL *?*. Pour répondre à cette question, nous allons nous inspirer des travaux effectués autour des systèmes réactifs synchrones à modes de fonctionnement, mais avant de décrire plus en détail notre solution dans le chapitre 7, nous proposons d'étudier la relation entre l'approche synchrone et le modèle ARRAY-OL.

Comme nous l'avons décrit dans la section 2.4 (page 24), le modèle ARRAY-OL représente globalement un graphe de dépendance acyclique dans lequel l'ordre minimal des calculs est bien connu. C'est un modèle déterministe qui assure toujours le même résultat pour les mêmes entrées. En d'autres termes, en respectant les dépendances de données explicites entre les différentes tâches du modèle, l'ordre d'exécution des tâches parallèles n'a aucune influence sur le résultat final. Cette description fait partie des caractéristiques de base d'un modèle synchrone, ce qui facilite l'orientation du modèle ARRAY-OL vers un modèle synchrone.

La particularité du modèle ARRAY-OL se situe dans sa puissance d'expression de tout le parallélisme potentiel, et éventuellement infini, présent dans la spécification du système étudié. Cependant, cette description est rapidement brisée au niveau de l'implémentation puisque le nombre réel de ressources utilisées est souvent incapable d'assurer une telle exécution parallèle. Pour cette raison, une sérialisation du modèle ARRAY-OL devient nécessaire. De plus, les données en entrée et en sortie manipulées par les tâches ARRAY-OL sous forme de tableaux infinis sont généralement produites ou consommées par des composants réels tels que les capteurs ou les actionneurs. Il est donc naturel de représenter les tableaux ARRAY-OL par des flots de données produits ou consommés selon une échelle de temps logique en se basant sur un mécanisme synchrone.

En 2005, P. Dumont a étudié la projection du modèle ARRAY-OL sur des modèles de calcul génériques tels que le modèle séquentiel, SPMD[17] et pipeline, et sur des modèles de calcul concrets tels que SDF[18] et KPN[19], pour pouvoir l'exécuter sur des plateformes réelles [Dum05]. Une première proposition consistait en une utilisation conjointe du séquentiel et du SPMD. Le modèle global est ainsi exécuté séquentiellement alors que le modèle local bénéficie de l'expression du parallélisme de données présent dans ARRAY-OL. Cependant, une exécution séquentielle du modèle global reste peu intéressante puisqu'elle considère l'exécution d'une seule tâche à la fois. De plus, l'utilisation de dimensions infinies dans les tableaux implique une exécution également infinie des tâches et donc provoque un blocage de l'exécution du système global.

Pour résoudre ce problème, P. Dumont propose une séquentialisation des tableaux pour permettre la création d'un *flots de données* et donc de bénéficier d'une exécution de type pipeline. Dans ce contexte, deux types de flots de données ont été proposés : les *flots de motifs* et les *flots de tableaux*. La création des flots de motifs ou de tableaux est réalisée par un certain nombre de processus tel que la définition des niveaux de hiérarchie par le processus de *fusion* [Dum05]. Les processus appliqués changent la forme de l'application mais ne peuvent pas changer son sens global puisqu'elle produit toujours les mêmes résultats. Il est aussi à noter que la création des flots de motifs ou de tableaux peut engendrer certains problèmes tels que la difficulté de la définition d'un ordre pour la production et la consommation des données, et les interblocages dus aux tableaux infinis.

La construction de flot de données pour un modèle ARRAY-OL est inévitable pour son exécution réelle. De plus, cette méthode rapproche le modèle ARRAY-OL des modèles à flots de données synchrones traditionnels, ce qui lui offre l'avantage de bénéficier des résultats développés pour ces modèles, notamment pour la représentation des applications de traitement du signal systématique. Une mise en œuvre concrète de cette projection a été réalisée pour les modèles SDF et KPN qui sont déjà utilisés pour la simulation des applications de traitement systématique visées par ARRAY-OL [ABD05]. De plus, une implémentation dans PTOLEMY, dénommée « ARRAY-OL *for* PTOLEMY II[20] », est réalisée en se basant sur les implémentations existantes du modèle SDF pour faciliter la simulation des applications ARRAY-OL [DB05].

Une autre mise en œuvre synchrone du modèle ARRAY-OL a été étudié par A. Gamatié et al. [GRY+06] en 2006. Cette étude est proche de celle proposée par P. Dumont dans la création de flots à partir des tableaux infinis. Elle est uniquement basée sur les applications

[17] Simple Program Multiple Data
[18] Synchronous DataFlow
[19] Kahn Process Network
[20] `http://www.lifl.fr/west/aoltools`

initialement transformées par des processus de fusion qui permettent la définition d'une seule tâche à partir de plusieurs en créant des niveaux de hiérarchie supplémentaires. À partir de la tâche de plus haut niveau, deux modèles différents sont définis pour la représentation synchrone du modèle ARRAY-OL : le *modèle en parallèle* (parallel model), et le *modèle en série* (serialized model).

Le principe du modèle en parallèle consiste à modéliser les tableaux ARRAY-OL par des flots de tableaux, et chaque répétition d'une tâche T par un modèle de calcul synchrone. Ainsi, l'application répétée de la tâche T est considérée comme étant la composition synchrone des différents modèles relatif à chaque répétition. Cette représentation respecte la caractéristique du modèle ARRAY-OL, qui spécifie que seul les dépendances de données peuvent définir un ordre d'exécution, puisque la composition synchrone n'introduit aucun ordre d'exécution sur les différents modèles de répétition.

Le modèle en série permet de fournir une vue plus raffinée des spécifications ARRAY-OL décrites par le modèle en parallèle. Ceci est réalisé par l'introduction de deux processus de base : *Array to Flow* et *Flow to Array* qui permettent respectivement de transformer les tableaux ARRAY-OL d'entrée vers des flots de données en entrée, et les flots de données en sortie vers des tableaux ARRAY-OL de sortie. Ceci nécessite un certain nombre de processus de mémorisation, d'ordonnancement, et surtout la définition d'une *horloge logique* pour permettre une représentation synchrone des applications ARRAY-OL plus facile à analyser. Une extension de ces travaux préliminaires consiste à proposer une transformation des applications ARRAY-OL vers des équations synchrones, et en particulier vers le langage LUSTRE [YGR+06]. Les modèles synchrones résultants permettent et facilitent l'analyse et la vérification du système en utilisant les outil formels offerts par la technologie synchrone.

Les travaux effectués sur la définition d'une représentation synchrone pour les modèles ARRAY-OL sont très récents et toujours en cours de développement. Les premiers résultats sont appliqués pour des cas particuliers d'applications ARRAY-OL, et posent un challenge remarquable sur la définition exacte du lien entre le modèle ARRAY-OL et la définition d'une horloge logique synchrone.

3.5 Synthèse et conclusion

Nous avons présenté dans ce chapitre le principe d'utilisation de l'approche synchrone pour la modélisation des systèmes réactifs critiques. Nous avons aussi montré que ces applications peuvent contenir un mélange de contrôle et des traitements de données ce qui rend nécessaire le développement de méthodes bien adaptées pour l'étude de ces systèmes.

Dans notre travail, nous nous intéressons principalement à l'étude des systèmes réactifs à modes de fonctionnement. Les deux approches multi-langages et transformationnelle, mentionnées ci-dessus, permettent la spécification de tels systèmes. Cependant, ces études n'imposent aucune méthodologie de conception pour soit, séparer clairement la partie contrôle de la partie données, ou séparer les différents modes de fonctionnement du système. Ces approches donnent plus de liberté aux développeurs qui ont souvent tendance à construire leur applications selon leur point de vue personnel, ce qui peut rendre difficile la compréhension, l'étude, et la réutilisation des différentes parties du système. Ainsi, l'application des techniques de vérification formelle pour de telles spécifications peut être plus difficile, et ne facilite pas la localisation des erreurs. Nous pensons alors qu'une séparation claire entre la partie contrôle et les différents modes de fonctionnement du système permet d'avoir un

modèle plus clair et plus simple à maintenir et à vérifier. Sachant que le rôle principal de la partie contrôle consiste à piloter les différents modes de fonctionnement du système qui sont par nature exclusifs, la spécification séparée des différentes parties du système permet alors l'application séparée des processus de test et de vérification, et peut assurer la vérification du système global. L'étude et le développement de ce concept de séparation contrôle/données pour cette catégorie de système fait partie de notre contribution, et il est présenté plus en détails dans le chapitre 5.

Dans [MR00], F. Maraninchi et al. montrent, à travers un exemple de chaîne de production, que les applications industrielles peuvent être mieux spécifiées en utilisant la structure de modes si leur comportement est principalement régulier. Les automates de modes représentent alors une structure bien conçue pour la spécification des systèmes à mode de fonctionnement. Cependant, ce concept a été uniquement réalisé dans le contexte du langage LUSTRE limitant ainsi le domaine des applications étudiées à celles pouvant être uniquement spécifié par ce langage. Pour cette raison, nous proposons d'étendre le concept des automates de mode pour l'appliquer aux applications de traitement intensif à parallélisme massif, présentées dans le chapitre 2, et qui représentent le contexte de base de notre étude. En d'autres termes, nous allons nous inspirer du concept des automates de modes pour introduire la spécification du contrôle pour les applications parallèles. Ce travail est réalisé dans le contexte du langage ARRAY-OL, et il est présenté plus en détail dans le chapitre 7.

En résumé, la problématique à laquelle contribue ce travail est alors essentiellement la proposition d'une méthodologie de séparation contrôle/données, et l'introduction du contrôle dans le modèle parallèle ARRAY-OL en se basant sur l'approche réactive synchrone et le concept des automates de modes. Mais avant de discuter plus en détails ces deux concepts, il est important de noter que notre étude est basée sur la modélisation à haut niveau des applications selon une approche d'Ingénierie Dirigée par les Modèles (IDM). Elle fait partie d'une démarche de *co-design* dans un environnement de développement particulier dénommé GASPARD2 dont nous allons présenter les principes dans le chapitre 4.

Chapitre 4

Ingénierie dirigée par les modèles pour la co-conception des systèmes sur puce

Dans ce chapitre, nous présentons la troisième partie du contexte de notre étude concernant la modélisation à haut niveau et la co-conception des systèmes sur puce. Notre démarche est basée sur une approche d'ingénierie dirigée par les modèles (IDM) qui permet de faciliter l'étude et le développement de ces systèmes. Après avoir introduit les principes de base de cette démarche, nous allons présenter l'environnement de développement GASPARD2, principalement conçu pour la co-conception des systèmes sur puce définis autour des applications et d'architectures parallèles basés sur ARRAY-OL. Puisque l'approche IDM fait généralement appel à l'utilisation de langages de modélisation communs et standardisés tel que UML pour mieux représenter et spécifier les différentes fonctionnalités du système étudié, nous allons étudier l'utilisation de ce langage dans le domaine de la conception des systèmes sur puce. L'objectif principal de notre travail est d'introduire la modélisation du contrôle pour les applications de traitement parallèles dans GASPARD2 en se basant sur une approche synchrone. Une étude de la combinaison du langage UML et la technologie synchrone, et en particulier la représentation des automates de contrôle en UML sont également présentées.

4.1 Introduction

Dans l'industrie, les soucis d'efficacité, de réutilisation, et de programmation sans erreurs sont bien connus. D'une part, le besoin exprimé par le client, souvent incompétent dans le domaine de la conception informatique, et les contraintes d'environnement d'un futur système sont généralement flous et incomplets. Ces besoins sont par nature non totalement formalisables car ils font partie du monde réel, et liés à des habitudes ou des opinions parfois mal conceptualisés. Réciproquement, le concepteur est étranger au monde du client, et doit prendre en considération tous ses besoins. Il s'agit donc de trouver un langage commun entre ces deux mondes différents. D'autre part, un défi supplémentaire est posé par le partage du travail. Le développement est généralement effectué à plusieurs puisque les différentes parties d'un système sont toujours plus ou moins interdépendantes. Les développeurs et concepteurs doivent pouvoir se mettre d'accord sur les fonctionnalités en interaction et les formalismes utilisés pour faciliter l'échange et la réutilisation des différentes parties du système étudié.

Pour ces raisons, il est nécessaire d'utiliser des techniques de modélisation et des langages communs permettant l'expression, l'échange, et la compréhension des principales fonctionnalités des systèmes étudiés. Dans ce domaine, l'*ingénierie dirigée par les modèles* (IDM[21]), appelée en anglais MDE (Model Driven Engineering) ou aussi MDD (Model Driven Development), est considérée aujourd'hui comme l'une des approches les plus prometteuses dans le développement des systèmes. Cette technique a pour ambition de fournir un cadre conceptuel, technologique et méthodologique dans lequel les modèles sont au centre des activités du processus de développement des systèmes informatiques étudiés, facilitant ainsi la modélisation et la conception de ces systèmes.

Dans le cadre de notre travail, nous allons nous positionner dans un contexte particulier de systèmes informatiques appelés *systèmes sur puce* (SOC : System On Chip). Ces systèmes regroupent des fonctionnalités logicielles et des composants matériels embarqués dans une même puce. Ils opèrent souvent dans des conditions temps réel, et sont soumis à de sévères contraintes de bon fonctionnement. Il est donc nécessaire d'utiliser des méthodes de

[21] `http://idm.imag.fr`

développement bien rigoureuses pour assurer la fiabilité de ces systèmes et faciliter leur *co-conception*.

Le terme « co-conception » (co-design) est utilisé pour désigner la conception conjointe matériel/logiciel d'un système. Ce concept est né suite à la nécessité de produire simultanément des descriptions de type logiciel et des composants matériels afin d'optimiser les coûts de développement et d'améliorer les performances. Comme tout processus de conception de système, la conception conjointe doit passer par plusieurs processus de spécification, de validation et de synthèse. Dans ce cadre, le processus de modélisation conjointe matériel/logiciel à différents niveaux d'abstractions est appelé *co-spécification* ou *co-modélisation*, la validation du comportement du système par des processus de simulation ou de vérification formelle est appelée *co-simulation*, et le processus d'exploitation, d'évaluation et de génération du code et des circuits matériels correspondants est appelé *co-synthèse*.

Dans notre étude, nous allons nous intéresser uniquement au processus de co-modélisation des systèmes sur puce en étudiant la modélisation au plus haut niveau d'abstraction de ces systèmes selon l'approche IDM et via l'environnement de développement GASPARD2[22]. Cet environnement est spécialement conçu pour le développement des systèmes sur puce en se basant sur le modèle ARRAY-OL, permettant ainsi l'étude et la co-conception des applications de traitement du signal intensif, présentées dans le chapitre 2, et sur lesquelles se base notre travail.

4.2 L'Ingénierie Dirigée par les Modèles (IDM)

Afin de pouvoir comprendre et d'agir sur le fonctionnement d'un système, il est nécessaire de disposer d'un *modèle* de ce dernier. Un modèle est une simplification et/ou une abstraction de la réalité. Modéliser consiste à identifier les caractéristiques intéressantes ou pertinentes d'un système dans le but de pouvoir l'étudier du point de vue de ces caractéristiques. Un bon modèle doit, d'une part, faciliter la compréhension et réduire la complexité du système étudié, et d'autre part, permettre la simulation et la vérification de ses différents concepts et fonctionnalités.

La modélisation permet de mieux visualiser et contrôler la construction du système en clarifiant les différentes relations et interactions possibles. Les abstractions de la réalité que propose un modèle identifient les caractéristiques intéressantes d'une entité en masquant les détails dans le but de disposer d'éléments exploitable par des outils mathématiques et/ou informatiques.

Les systèmes étudiés peuvent être modélisés selon différentes approches adaptées à la nature de ces systèmes et au contexte d'application dans lequel ils sont définis. Parmi ces approches, l'*Ingénierie Dirigée par les Modèles* présente une contribution importante dans le domaine de la conception de systèmes logiciels [Ken02]. Elle représente une forme d'ingénierie générative, par laquelle tout ou partie d'une application est générée à partir de modèles.

L'IDM peut être vue comme une famille d'approches qui se développent à la fois dans les laboratoires de recherche et chez les industriels impliqués dans les projets de développement logiciels. Cette approche offre un cadre méthodologique et technologique qui permet d'unifier différentes façons de faire dans un processus homogène. Il est ainsi possible d'utiliser la technologie la mieux adaptée pour chacune des étapes du développement, tout en ayant un processus de développement global et unifié.

[22] `http://www.lifl.fr/west/gaspard`

Dans l'IDM, le code source n'est plus considéré comme l'élément central d'un système, mais comme un élément dérivé de la fusion d'éléments de modélisation. Dans cette approche, l'étude du système est basée sur la construction de modèles qui occupent la place principale dans le cycle de développement du système, et doivent en contrepartie être suffisamment précis afin de pouvoir être interprétés ou transformés par des machines vues comme un ensemble de transformations de modèles. De façon générale, l'IDM peut être défini autour de trois concepts de base : les *modèles*, les *métamodèles* et les *transformations*. La figure 4.1 illustre la mise en relation de ces trois concepts.

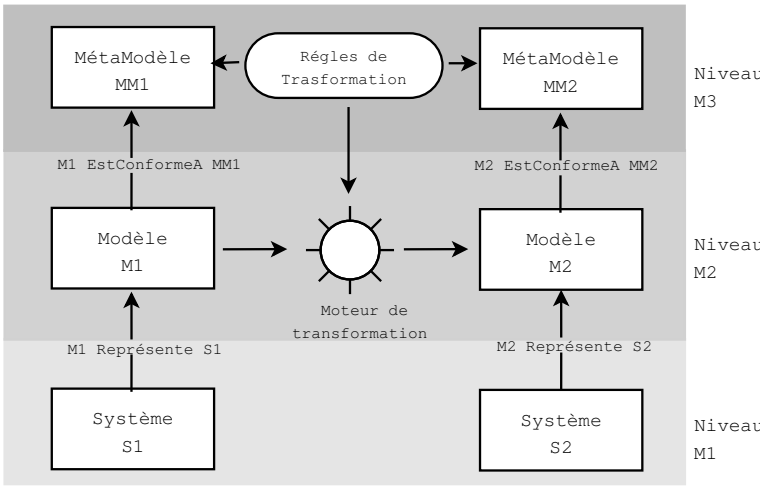


FIG. 4.1 – *Concepts de base de l'approche IDM*

Un des aspects essentiels de l'approche IDM, consiste à automatiser la transformation de modèles. Ponctuellement, il s'agit de transformer le code d'un langage en un autre, ou une modélisation abstraite en une structure de classes, ou même un modèle de données en un autre modèle tout en assurant que les propriétés des données sont conservées lors de la transformation. Dans ce contexte, la vérification de modèles basée sur des approches formelles donnant des spécifications rigoureuses a pour objectif de garantir que les modèles possèdent toutes les qualités attendues, en particulier lorsqu'ils ont en charge de modéliser la fabrication de systèmes embarqués développés dans des secteurs comme les transports ferroviaires, l'avionique, le spatial, les télécommunications ou l'énergie. L'approche IDM ouvre ainsi de nouvelles perspectives dans le cadre de la conception des systèmes embarqués. De plus, une caractéristique principale de l'approche IDM consiste à favoriser l'étude séparée des différents aspects du système. Cette séparation augmente la réutilisation et aide les membres de l'équipe de conception à partager et s'échanger leur travaux indépendamment de leur domaines d'expertise. Ce concept est particulièrement intéressant dans le domaine de conception des systèmes sur puce où les deux technologies logicielle et matérielle vont interagir pour la définition du comportement global du système.

4.2.1 Model Driven Architecture (MDA)

Une « variante[23] » de l'approche IDM définie par l'OMG[24] (Object Management Group) est appelée MDA[25] (Model Driven Architecture) [Poo01, MM03].

L'approche MDA est basée sur l'utilisation de modèles pour la description des systèmes logiciels à développer. Elle fournit un ensemble de directives permettant de séparer les contraintes fonctionnelles des contraintes techniques. L'idée de base de cette approche consiste à permettre la description du système à plusieurs niveaux d'abstractions en séparant sa spécification fonctionnelle des détails de sa plateforme d'exécution et des contraintes techniques. Dans ce contexte, MDA fournit une approche et des outils pour :

- spécifier les fonctionnalités du système indépendamment de sa plateforme d'exécution (niveau PIM : Platform Independent Model),
- spécifier les plateformes d'exécution,
- choisir une plateforme particulière pour le système, et
- transformer la spécification du système vers une spécification plus adaptée à la plateforme choisie (niveau PSM : Platform Specific Model)

La première étape du processus MDA est relative au niveau PIM qui consiste à réaliser un modèle indépendant de toutes plateformes d'exécution. Le PIM représente uniquement les caractéristiques fonctionnelles et le comportement du système sous forme d'un modèle simple et clair. La clarté de ce modèle permet aux concepteurs de mieux comprendre les fonctionnalités du système, et donc de pouvoir vérifier ses caractéristiques et sa cohérence. Après avoir choisi la plateforme d'exécution du système, la deuxième phase dans le processus MDA consiste à transformer le PIM vers un PSM. Le PSM permet d'obtenir un modèle spécifique à la plateforme d'exécution utilisée pour la mise en œuvre du système. À ce niveau, les caractéristiques d'exécution et les informations technologiques sont prises en compte dans la description du modèle.

Les transformations entre les niveaux PIM et PSM sont généralement effectuées via des outils automatisés tel que QVT (Queries/Views/Transformations) [Gro05a], le langage de transformation de modèles standardisé par l'OMG. L'approche MDA se présente sous la forme d'un ensemble de standards utilisés pour créer un modèle et l'affiner jusqu'à obtenir, idéalement, un produit fini, comme du code source. Cette approche fait généralement appel à l'utilisation de langages de modélisation et d'outils de développement communs et standardisés comme SDL(Specification and Description Language) [tTSSot02], UML[26] (Unified Modeling Language) et SA-RT (Structured Analysis for Real-Time systems) [HP88]. En pratique, nous rencontrons le plus souvent l'approche MDA en définissant la spécification de l'application avec UML et en générant automatiquement l'exécutable lié, grâce à des générateurs de code Java, SystemC, etc.

Dans le cadre de notre étude, nous allons nous intéresser uniquement aux aspects de modélisation à haut niveau selon l'approche MDA, et en utilisant le langage UML pour la spécification des systèmes sur puce qui nous intéressent. Les aspects de transformation de modèles et de génération du code ne seront donc pas abordés dans ce travail.

[23] Historiquement, l'approche MDA a été proposée avant l'approche IDM.
[24] `http://www.omg.org`
[25] `http://www.omg.org/mda`
[26] `http://www.uml.org`

4.2.2 Modélisation UML

Pour répondre aux besoins de documentation et de spécification de logiciels, de nombreux langages graphiques ont vu le jour, en particulier au sein de la communauté des objets. Afin de privilégier la réutilisation de composants, d'architectures, ou de favoriser l'interconnexion entre des systèmes modélisés avec des notions différentes, il apparaît évident qu'une notation consensuelle est nécessaire. De ce constat est né la notation UML [Gro04] qui représente un des formalismes les plus utilisés dans l'approche IDM.

UML (Unified Modeling Language) est un langage de modélisation visuelle utilisé pour documenter, spécifier, et visualiser graphiquement les aspects d'un système logiciel. Comme son nom l'indique, ce langage est le résultat d'une longue maturation. Il est né de la fusion de plusieurs langages de modélisation qui ont le plus influencé la modélisation objet au milieu des années 90, notamment les méthodes OMT [RBP+91], Booch [Boo91] et OOSE [JCJO92]. L'ambition d'UML est de rassembler en une seule notation les meilleures caractéristiques des différents langages de modélisation à objets. Cette unification a aussi pour effet de donner une masse critique à UML.

En tant que standard de l'OMG, et en tant que successeur de notations déjà bien implémentées, UML jouit d'une popularité à la fois dans l'industrie du logiciel et dans le monde de la recherche scientifique. L'approche UML ne représente pas une méthode à proprement parler, mais plutôt un langage de représentation de processus. En d'autres termes, UML n'impose pas une démarche pour l'enchaînement des activités d'une organisation, mais essentiellement un support de communication, qui facilite la représentation et la compréhension de ces activités. Sa notation graphique permet d'exprimer visuellement une solution, et facilite la comparaison et l'évaluation des différentes solutions proposées. Son indépendance par rapport aux langages de programmation, aux domaines d'application et aux processus, en fait un langage « universel ».

Le langage UML facilite la communication entre clients et concepteurs, et entre équipes de concepteurs. Sa syntaxe étant bien définie dans [JBR97] sous forme de diagrammes UML, cela accélère le développement des outils, et permet le passage d'une spécification UML de haut niveau vers la génération d'un code exécutable. Principalement, UML définit deux catégories de diagrammes : *structurels* et *comportementaux* [Dou04]. L'aspect structurel regroupe les notions de base permettant la représentation de la structure statique du système. Il spécifie l'organisation interne des différents éléments du système, ainsi que leurs interactions possibles. Parmi les diagrammes structurels les plus utilisés dans UML nous citons les *diagrammes de classe* et les *diagrammes de composant*. L'aspect comportemental permet quant à lui de modéliser la dynamique du système en spécifiant comment les valeurs ou les états des objets changent dans le temps. Dans ce contexte, UML permet la spécification du comportement d'un objet individuel, ou d'un ensemble d'objets coopérant. Le premier type de comportement est généralement représenté par des *Statecharts* ou des *diagrammes d'activités*, tandis que le deuxième est représenté par des *diagrammes de séquence* ou des *diagrammes de temps*.

L'aspect formel des notations dans UML permet généralement de limiter les ambiguïtés et les incompréhensions des modèles. La véritable force de la modélisation UML est due au fait qu'elle repose sur un métamodèle qui normalise la sémantique des différents concepts proposés. Cependant, la modélisation graphique proposée par le langage UML n'est généralement pas suffisante pour donner une spécification bien précise et non ambiguë du système étudié. Il y a toujours un besoin pour spécifier des contraintes supplémentaires sur

les différents objets du modèle et leurs modes d'interaction. De telles contraintes ont été souvent décrites dans un langage naturel, mais cette description peut conduire à des confusions et des incompréhensions du modèle. Elles peuvent être également décrites dans des langages formels, mais l'inconvénient de ces langages est qu'ils sont utilisables par les personnes ayant une formation mathématique et formelle, mais difficile à manipuler par les industriels ou les concepteurs et développeurs des systèmes. Pour résoudre ce problème, un langage appelé OCL (Object Constraint Language) [Gro05b] est développé pour définir un compromis entre le langage naturel et les descriptions formelles. L'introduction du langage OCL a permis d'enrichir la modélisation UML en spécifiant des contraintes supplémentaires sur le comportement du système. C'est un langage déclaratif, textuel et formel qui joue un rôle important dans la phase d'analyse dans le cycle de développement du système.

Depuis sa standardisation par l'OMG en 1997, plusieurs versions d'UML ont été proposées. La dernière version est UML2.0 standardisée en juin 2003 qui fournit des éléments de modélisation et une sémantique plus étendue et systématique que les versions précédentes [Sel04]. Pour la spécification des contraintes sur les objets UML2.0, une version plus adaptée du langage OCL (OCL2.0) est aussi développée en 2004 pour assurer la conformité des deux langages.

Vue l'importance de la modélisation UML et sa large utilisation dans les domaines industriels et académiques, nous avons choisi d'adopter les diagrammes UML pour l'étude et la co-conception de la structure et du comportement des systèmes sur puce étudiés selon une approche MDA.

4.3 Systèmes sur puce et l'environnement GASPARD2

Les systèmes électroniques sont de plus en plus présents dans notre quotidien. Cette technologie peut être trouvée aussi bien dans les systèmes multimédia que dans les applications de réseau et de télécommunication. Dans ce domaine, l'évolution du marché exige plus de fonctionnalités et de puissance de calcul, une grande fiabilité et une réduction des coûts.

Selon le Dr. Gordon E. Moore, cofondateur d'Intel, la complexité des circuits intégrés suit une loi empirique (*la loi de Moore*) dans laquelle le nombre et la puissance des transistors doublaient presque tous les deux ans.

> *En fait on constate que tous les 18 à 24 mois nous assistons à une progression technologique exponentielle depuis la fin des années '60 ayant pour résultat de doubler 20 fois ou d'un facteur d'environ un million la capacité des chips. Mais combien de temps ce progrès peut-il continuer ?*
>
> Le Dr. Moore interviewé lors de la Conférence Internationale sur les Circuits Intégrés (ISSCC) en 2003[a]
>
> [a] `http://www.intel.com/pressroom/archive/photos/isscc_2003.htm`

Ces progrès technologiques ont permis d'intégrer sur une même puce plus de fonctionnalités ce qui a conduit à la définition d'un nouveau paradigme de systèmes embarqués : les *systèmes sur puce* (SOC : System on Chip). Ainsi, les constructeurs des systèmes et les scientifiques doivent prendre en compte cette évolution technologique au cours de développement

de leur projets. Le développement de ces systèmes nécessite des méthodes et des outils rigoureux pour assurer leur fonctionnalité tout en diminuant leur coûts de production.

Dans ce qui suit, nous allons présenter le concept des systèmes sur puce, notamment le processus de co-conception de ces systèmes. Nous présentons également un ensemble d'environnements et de projets destinés à l'étude de ces systèmes en consacrant notre étude à l'environnement de développement GASPARD2.

4.3.1 Co-conception des systèmes sur puce

La technologie des systèmes sur puce permet d'embarquer plusieurs composants matériels et logiciels incluant des microprocesseurs programmables, des mémoires, des fonctions de traitement du signal numériques ou analogiques, des interfaces de communication, etc. Ainsi, la densité d'intégration de transistors permet de construire des puces avec plus d'un milliard de transistors[27]. Cette technologie ne permet pas uniquement de rassembler les traitements informatiques nécessaires pour la réalisation d'une fonction complexe sur une même puce, mais aussi de réutiliser les IPs (*Intellectual Property*) matériels et logiciels existantes. Ces systèmes permettent alors une réduction importante des coûts de production et une augmentation considérable de la fiabilité des systèmes développés. La réalisation des différents composants d'un système sur puce peut être faite par plusieurs équipes de développement, où chaque équipe utilise son propre environnement de conception pour décrire la fonctionnalité du système étudié.

Ce nouveau type de circuit intégré nécessite de nouveaux outils et de nouvelles méthodes de conception, de réalisation et de vérification. La conception des systèmes sur puce est soumise à un ensemble de contraintes fortes pour assurer leur bon fonctionnement. Comme ces systèmes regroupent des descriptions logicielles et matérielles, il est important de prendre en considération l'étude conjointe de ces deux concepts et leurs différentes interactions au plus haut niveau d'abstraction pour faciliter et réduire le cycle de développement. Il est alors nécessaire de maîtriser la conception matériel/logiciel des systèmes sur puce tout en respectant les contraintes de mise sur le marché et les objectifs de qualité. Pour ce faire, le processus de *co-conception* (*co-design*) des systèmes sur puce est apparu. C'est un processus complexe qui nécessite de grandes compétences de développement et un grand travail de test et de vérification afin d'assurer le bon fonctionnement du système global. Dans ce domaine, des outils et des environnements de développement sont conçus pour faciliter et assister l'étude des systèmes sur puce. Parmi ces environnement nous citons SCE[28] (SOC Environment) qui fournit un environnement graphique pour la modélisation, la synthèse et la validation des systèmes sur puce, et l'environnement GASPARD2 développé pour la modélisation, la simulation et la génération du code pour des systèmes sur puce en se basant sur le modèle parallèle ARRAY-OL.

Comme nous l'avons introduit dans le chapitre 2, le contexte de base de notre travail s'inscrit dans le cadre de la description des applications de traitement du signal intensif à parallélisme massif en utilisant le modèle ARRAY-OL. Notre travail fait donc partie du projet de développement de l'environnement GASPARD2 où nous nous intéressons en particulier à la modélisation des applications au plus haut niveau d'abstraction.

[27] `http://public.itrs.net/Home.htm`
[28] `http://www.ics.uci.edu/~cad/sce.html`

4.3.2 L'environnement GASPARD2

GASPARD2 est un environnement de développement pour la co-conception et la co-modélisation visuelles des systèmes sur puce, développé dans le contexte du projet INRIA DART[29]. Cet environnement étend le langage ARRAY-OL, présenté dans le chapitre 2, et permet la modélisation, la simulation, le test et la génération du code pour les applications et les architectures matérielles des systèmes sur puce.

L'environnement GASPARD2 est principalement destiné à la spécification des applications de traitement du signal intensif. Il est basé sur une méthodologie *orientée modèle* selon le flot de conception en Y inspiré du modèle de conception *Y-chart* [GK83]. Le flot de conception GASPARD2, décrit par la figure 4.2, est principalement défini autour de trois axes : l'*application* qui spécifie les fonctionnalités du système, l'*architecture matérielle* utilisée comme support d'exécution et de réalisation des fonctionnalités de l'application, et l'*association* qui spécifie le placement de l'application sur une architecture matérielle donnée. Dans la figure 4.2, les boites représentes les différents métamodèles de la chaîne de

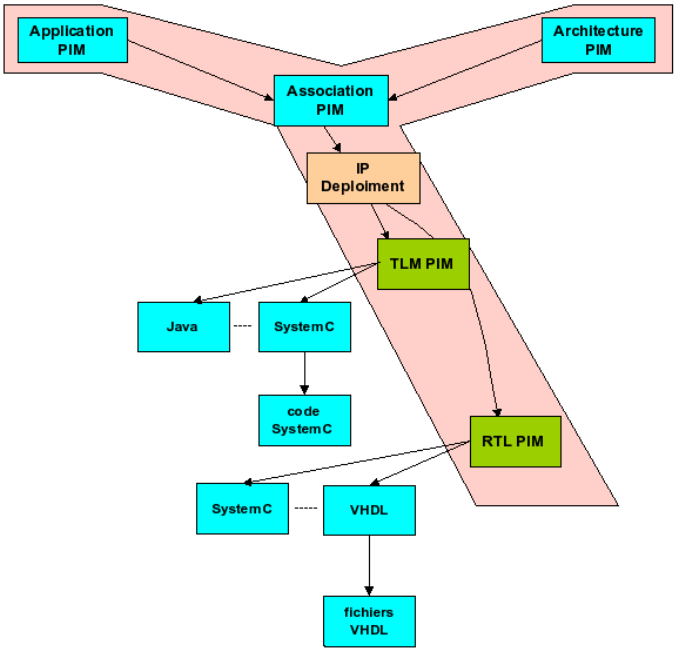


FIG. 4.2 – *Modèle* Y *relatif au flot de conception* GASPARD2

conception, et les arcs spécifient le lien entre ces métamodèles. Ainsi, l'environnement GASPARD2 est basé sur des modèles représentant le parallélisme potentiel de l'algorithme et le parallélisme physique de l'architecture matérielle. Le processus d'association consiste à distribuer et ordonnancer le modèle d'application sur le modèle d'architecture tout en respectant les contraintes du système. La caractéristique principale de cet environnement est que les concepts et la sémantique de chaque niveau de conception (application, architecture matérielle et association) sont indépendants de toute plateforme d'exécution (PIM : Platform Independent Model) qui peut être un composant SoC ou un simulateur logiciel ou matériel. Aucun composant de ces niveaux n'est associé à une technologie d'exécution, de simulation

[29] `http://www.lifl.fr/west`

ou de synthèse. Une fois que tous ces composants sont définis, il est possible de les associer à une certaine technologie (Java, SystemC RTL, SystemC TLM, VHDL, etc.) en raffinant le modèle d'association PIM vers un modèle TLM PIM (Transaction Level Model) en premier, et par la suite vers un modèle RTL PIM (Register Transfer Level).

La contribution du projet INRIA DaRT dans le flot de conception de GASPARD2 peut être définie autour de trois domaines : la *co-modélisation*, les *techniques d'optimisation*, et la *simulation*. Ces trois concepts visent un objectif commun qui consiste à réduire le temps du cycle de développement, et proposent des solutions prenant en compte les aspects réguliers et parallèles des systèmes étudiés. L'aspect de co-modélisation permet la modélisation conjointe des parties logicielles et matérielles du système étudié. Son objectif est de proposer un environnement de modélisation, au plus haut niveau d'abstraction, commun pour toutes les parties du système. Les techniques d'optimisation représentent principalement des transformations utilisées pour optimiser le placement et l'ordonnancement d'une application sur une architecture donnée, tandis que le processus de simulation permet de simuler le fonctionnement du système à plusieurs niveaux d'abstraction et de vérifier l'adéquation du placement et de l'ordonnancement.

Il est important de noter que le Y de GASPARD2 se situe à la frontière de deux domaines de recherche : la *conception des systèmes sur puce*, et l'*approche* MDA, décrits dans les sections précédentes. Ce flot de conception correspond principalement à un flot de co-conception des systèmes sur puce, ou plus généralement des systèmes embarqués incluant des parties logicielles et des parties matérielles. L'approche MDA intervient par contre dans la mise en œuvre de la méthodologie proposée par l'environnement GASPARD2. Cette méthodologie est définie autour de trois niveaux d'abstraction : *fonctionnel* correspondant au modèles application PIM, architecture PIM et association PIM, *transactionnel* correspondant au modèle TLM PIM et *registre* correspondant au modèle RTL PIM. Elle permet de gérer la complexité du flot de conception en mettant en œuvre plusieurs types de raffinement automatique ou semi-automatique basés sur des techniques de transformation de modèle.

Le contexte de notre étude se situe au niveau fonctionnel qui consiste à modéliser les parties application, architecture et association du système en utilisant le langage de modélisation UML2.0. Les modèles développés seront par la suite importés dans des plugins Eclipse[30] via le moteur de transformation ModTransf[31], et traités en appliquant des algorithmes de placement, d'ordonnancement et de génération du code. Dans le cadre de notre travail, nous allons nous intéresser uniquement au processus de modélisation au plus haut niveau d'abstraction des trois parties du système : *Application* PIM, *Architecture* PIM et *Association* PIM spécifiés dans le flot de conception en Y de la figure 4.2. Les autres niveaux d'abstraction et les technologies de simulation, d'optimisation et de génération du code ne seront pas abordés dans ce travail.

4.4 UML pour la co-conception des systèmes sur puce

Les contraintes de bon fonctionnement des systèmes embarqués, et en particulier des systèmes sur puce, ont imposé une démarche méthodique pour le développement de ces systèmes. Dans ce contexte, les processus itératifs et les méthodes orientés objets ont remplacé les processus en cascade et les méthodes procédurales. En tant que notation orientée

[30] `http://www.eclipse.org`
[31] `http://www.lifl.fr/west/modtransf`

objet standard, UML est souvent utilisé comme un terme commun dans le développement des systèmes informatiques, autant sur le plan industriel que sur le plan académique. Le langage UML peut donc être vu comme un modèle « universel » pour l'analyse et la conception des systèmes informatiques, avec comme ambition de couvrir tous les domaines d'application.

Il s'avère que si quelques modèles peuvent être directement étudiés par le langage UML, d'autres nécessitent une spécialisation liée à leur classe d'application. Cette nécessité a conduit vers l'introduction des extensions au langage UML qui sont reconnues par la communauté UML et rendues possibles via la notion du « *profil* » (profile). Un profil correspond au regroupement d'extensions et de spécialisation du langage UML du point de vue notation et sémantique. Ceci est principalement réalisé via le concept de *stéréotype* qui représente la définition d'une propriété supplémentaire appliquée sur un élément standard d'UML (classe, association, attribut, etc.).

Dans le domaine de la co-conception des systèmes sur puce, plusieurs profils ont été proposés pour permettre la prise en compte des caractéristiques de base de ces systèmes dans un modèle UML, aussi bien sur le plan modélisation et analyse que sur le plan validation et génération du code exécutable.

Dans le cadre de la modélisation et l'analyse des systèmes sur puce, deux profils de base sont en cours de standardisation à l'OMG : SysML (System Modeling Language) [Tea06], et MARTE (Modeling and Analysis of Real-Time and Embedded systems) [Gro05e].

SysML : Le profil SysML définit une proposition d'un langage générique offrant un support de modélisation pour l'ingénierie des systèmes sans aucune particularité pour les systèmes embarqués. L'objectif de SysML consiste à supporter la spécification, l'analyse, la conception, la vérification et la validation d'une large catégorie de systèmes complexes pouvant inclure à la fois du matériel, du logiciel, des processus, des personnes, des informations personnalisées, etc. SysML réutilise un sous-ensemble d'UML2.0 et fournit des extensions pour répondre aux besoins du langage, notamment pour faciliter la spécification des structures, des comportements, des allocations et des contraintes d'analyse. Son objectif de base est de proposer des mécanismes permettant de faciliter la modélisation des caractéristiques attendues du système étudié. Dans ce profil, et en plus de l'extension de certains packages UML, trois packages de base ont été introduits : `Requirements`, `Parametrics` et `Allocation`. Le package `Requirements` est utilisé pour la modélisation des caractéristiques attendues d'un système. Le package `Parametrics` définit les relations paramétriques entre les éléments d'un système qui sont généralement utilisés pour le processus d'analyse. Le package `Allocation` désigne un mécanisme permettant de définir des liens entre les éléments des différents modèles. Ce concept est vu comme une étape importante dans la conception du système qui permet de garantir sa cohérence puisque la spécification de ce lien définit un type de relation plus concret entre les différents éléments du système.

MARTE : Le profil MARTE est défini spécialement pour la modélisation et l'analyse des systèmes embarqués temps-réel. Son objectif est de fournir un moyen commun pour la représentation de l'application, de la plateforme d'exécution (architecture matérielle) et du placement (association) de l'application sur la plateforme définie. La principale motivation du profil MARTE est de remplacer le profil SPT (Schedulability, Performance, and Time) [Gro05d] d'UML1.x en proposant une modélisation plus adaptée, notamment pour la représentation du temps logique discret et continu. Ce profil permet aussi de définir un catalogue de qualités de service pour la prise en compte des contraintes des systèmes embarqués, et un mécanisme commun pour la représentation des structures répétitives, à la fois

pour l'application, l'architecture et l'association. La description des modèles selon MARTE garde un aspect plus abstrait puisqu'elle ne prend pas en considération les propriétés fonctionnelles dans l'expression et l'analyse du système. Il est aussi à noter que l'équipe INRIA DaRT participe à la définition du profil MARTE, notamment en ce qui concerne la définition des mécanismes de répétition et d'un modèle d'association tout en respectant l'approche orientée composant[32].

UML for SoC : Un autre exemple de profils utilisés pour la conception des systèmes sur puce est le profil *UML for SoC* [Gro05c]. Ce profil est défini comme un standard OMG pour la modélisation des systèmes sur puce et l'évaluation de leurs performances. Il est principalement basé sur l'extension du diagramme de structure composite sous forme de représentations visuelles particulières introduisant les concepts de base nécessaires pour la modélisation des architectures matérielles. Le profil UML for SoC permet la modélisation hiérarchique des *modules* et des *canaux*, éléments de base dans un système sur puce, ainsi que la modélisation des *ports* et des informations communiquées entre les modules. Il permet aussi de générer automatiquement des squelettes SystemC ou autres langages équivalents. Cependant, UML for SoC prend uniquement en compte la modélisation structurelle des systèmes étudiés. Les aspects comportementaux sont généralement négligés ou, dans certains cas, suggérés par les informations associées aux éléments structurels.

UML for SystemC : Un autre profil répandu dans le domaine de la co-conception des systèmes sur puce est le profil *UML for SystemC* [RSRB05]. Le but de ce profil est de définir dans la modélisation UML tous les éléments nécessaires pour la génération automatique du code SystemC. Ce profil inclut les concepts défini dans le profil UML for SoC, et permet en plus la définition des aspects comportementaux des modules et des canaux de façon non ambiguë. Le profil UML for SystemC peut être logiquement structuré en trois parties : *structure et communication*, *comportement et synchronisation* et *types de données*. La partie structure et communication permet la définition des stéréotypes sur la construction de blocs en SystemC, ainsi que la représentation hiérarchique des structures et des blocs de communication à base de module, d'interface, de ports et de canaux. La partie comportement et synchronisation définit des stéréotypes sur des variantes de machine à états dans UML pour permettre la description à haut niveau de spécification le comportement des processus SystemC (méthodes et threads) à l'intérieur des modules et des canaux. Enfin, la partie types de données permet de définir les classes UML utilisées pour la représentation des types de données utilisées dans le code SystemC.

Nous remarquons que les profils SysML et MARTE peuvent être principalement utilisés pour la modélisation à haut niveau d'abstraction des systèmes sur puce et faciliter leur analyse, tandis que les profils UML for SoC et UML for SystemC sont principalement dédiés à la modélisation des systèmes au niveau électronique ESL (Electronic System Level) dont le but est la génération du code pour ces systèmes. Ainsi, nous pouvons imaginer une approche MDA pour la co-conception des systèmes sur puce basée sur l'utilisation des profils UML introduits. Dans ce contexte, nous pouvons modéliser le système au plus haut niveau d'abstraction en utilisant le profil SysML, analyser ses fonctionnalités et la séparation matériel/logiciel selon le profil MARTE, et enfin générer des code de simulation (niveau TLM ou RTL) en utilisant les profils UML for SoC ou UML for SystemC.

Nous avons brièvement présenté les profils les plus utilisés et les plus remarquables dans

[32]Les propositions de l'équipe INRIA DaRT dans le cadre du profil MARTE sont principalement inspirés des travaux développé pour le profil GASPARD2.

le domaine de la co-conception des systèmes sur puce. Cependant, il est important de noter que l'étude de la combinaison de la modélisation UML et de la co-conception des systèmes sur puce est un travail de recherche récent et qui est toujours en cours d'évolution. Dans ce domaine, plusieurs autres travaux existent déjà dans la littérature, tandis que d'autres débutent leur développement. Par exemple, P. Green *et al.* ont développé une méthode appelée HASOC (Hardware and Software Objects on Chip) pour le développement des systèmes sur puce [GEE02, EG03]. C'est une méthode orientée objet basée sur une extension d'UML, et sur la décomposition du système en sous-systèmes matériel/logiciel pouvant être placés sur une plateforme réelle. Les auteurs ont également proposé la modélisation d'une plateforme basée sur l'utilisation d'UML et SystemC dont le but est d'enrichir UML pour mieux supporter la conception des systèmes embarqués jusqu'à la génération du code [GE02]. En 2003, M. Pauwels *et al.* ont proposé une méthodologie de conception pour le développement des systèmes sur puce en utilisant UML [PVS+03]. Le but principal de leur travail est de fournir un environnement commun pour la spécification et la documentation de l'architecture du système et de ses contraintes. Dans la même année, L. Baresi *et al.* ont développé un profil UML prenant en compte un ensemble d'informations nécessaires pour la génération des squelettes SystemC à partir des spécifications UML [BBDS03]. Une autre plateforme de modèle d'exécution générique, appelée MEP (Model Execution Platform), est introduite pour la conception des systèmes matériel/logiciel [SMR05]. Cette plateforme est également basée sur l'utilisation d'UML pour faciliter le passage de la spécification du système vers son implémentation en définissant une sémantique d'exécution bien précise. Plusieurs autres travaux ont été proposés dans le même domaine définissant ainsi un nouveau thème de recherche sur l'utilisation d'UML pour la co-conception des systèmes sur puce (UML for SoC Design). C'est un domaine qui bénéficie d'un grand intérêt académique et industriel, et regroupe plusieurs communautés scientifiques allant de la modélisation UML jusqu'au développement des circuits électroniques [MM05].

Dans le cadre de notre travail, nous allons nous intéresser uniquement à la modélisation au plus haut niveau d'abstraction des systèmes sur puce en se basant sur des concepts proches de celles proposés dans MARTE. Ce travail est réalisé dans le cadre du projet INRIA/DaRT, et propose la définition d'un profil UML pour l'environnement de développement GASPARD2 sur lequel se base notre étude (le profil sera présenté plus en détails dans le chapitre 8). Nous rappelons que le but principal de notre travail consiste à introduire la modélisation des concepts de contrôle dans les applications de traitement du signal intensif et parallèle en se basant sur une approche synchrone. L'étude de la relation entre la modélisation UML et la technologie synchrone, ainsi que la modélisation du contrôle dans UML sont alors intéressantes et inévitables.

4.5 Modélisation du contrôle dans UML

Dans la modélisation UML, les termes *contrôle* et *réactif* sont utilisés pour désigner les objets qui répondent dynamiquement aux événements, et dont le comportement est contrôlé par leur ordre d'arrivée. De tels objets sont généralement modélisés selon deux formalismes différents de machine à états : les *diagrammes de machine à états* (state machine diagrams) et les *diagrammes d'activités* (Activity diagrams) [Gro04]. Ces deux formalismes diffèrent selon la situation dans laquelle ils sont appliqués, ainsi que le comportement et la sémantique à vouloir représenter. Les diagrammes de machine à états sont généralement utilisés lorsque la

transition d'un état vers un autre est réalisée suite à l'occurrence d'un événement, tandis que les diagrammes d'activités sont plus adaptés pour les situations où le changement d'états se fait suite à l'accomplissement de l'exécution d'une activité à l'intérieur d'un état.

4.5.1 Diagrammes de machines à états

Le diagramme de machine à états peut être utilisé pour la modélisation d'un comportement discret via un système d'*étas-transitions* en représentant l'ensemble d'états possibles pour un objet et les transitions entre ces états. Ce diagramme est défini autour de trois éléments de base : les *états*, les *transitions* et les *actions*. Les états représentent les conditions d'existence de l'objet défini ou les différentes étapes de son modèle de comportement, les transitions spécifient les changements d'états et les progressions dans le comportement de l'objet en réponse à des événements, et les actions définissent un comportement atomique qui peut être exécuté par la machine à états (liées aux transitions ou aux états). L'objet doit être donc en mesure d'exécuter des actions soit au moment de déclenchement d'une transition ou à l'intérieur d'un état. Ces actions sont relatives à n'importe quel type d'action en UML qui peut être une opération simple telle que l'addition, ou une opération plus complexe telle que l'invocation de méthodes.

En UML, le diagramme de machine à états est défini comme une variante orientée objet des STATECHARTS de Harel [Har87]. Dans ce modèle, les états sont représentés par des rectangles arrondis et les transitions par des arcs orientés comme il est montré par la figure 4.3. Dans le diagramme de machine à état, la syntaxe de l'étiquette de la transition est définie

FIG. 4.3 – *Notations des états et des transitions dans un diagramme de machine à états en UML*

comme suit :

event-name (parameter-list) [guard-condition] / action-expression

L'event-name représente le nom d'une occurrence d'un événement. Si le nom de l'événement n'est pas spécifié, la transition est immédiatement affranchie[33]. parameter-list représente une liste de paramètres envoyée à l'objet pour exécuter les actions, action-expression définit les actions à exécuter lorsque la transition est affranchie, et [guard-condition] spécifie une expression booléenne qui définit une garde sur la transition. Si la garde évalue à false, l'événement est ignoré et la transition ne sera pas affranchie. Il est aussi à noter que tous les éléments présentés pour étiqueter la transition sont optionnels.

Ainsi, les machines à états en UML proposent des constructions de composition de choix, de parallélisme et de hiérarchie, permettant la description des comportements plus complexes des objets spécifiés. Les états décrivant un comportement de *choix* pour un objet sont généralement connus sous le nom des *états-ou* (*or-states*). Si A et B représentent l'ensemble des états-ou possible pour un objet, alors cet objet doit à tout moment être dans l'un de ces deux états. La description du *parallélisme* dans le modèle de machine à états UML peut être réalisée via la notion des *états-et* (*and-states*). Ce concept permet la représentation de la

[33] Dans le cas où il y a une garde, cette dernière doit évaluer à *true* pour que la transition soit affranchie.

simultanéité dans le sens où chaque état dans l'ensemble des états-et peut opérer indépendamment des autres états. La sémantique associée à ce modèle permet d'attribuer à chaque état dans l'ensemble des états-et une *copie personnelle* des événements reçus par l'objet. L'état global de l'objet sera défini par le produit cartésien de tous les états actifs de l'ensemble états-et. Il est important de noter que c'est l'*objet* qui reçoit les événements en entrée et non pas les états de son diagramme de comportement, et que l'objet ne peut traiter qu'un seul événement à la fois. La description *hiérarchique* dans le diagramme de machine à états permet une construction plus structuré des états d'un objet en les organisant en des ensembles de sous-états.

La principale caractéristique des machines à états en UML est qu'ils ont une sémantique « *run-to-completion* » qui impose qu'aucun autre événement ne peut être pris en considération avant que l'événement précédent soit complètement traité. Ceci signifie qu'une fois qu'un événement est reçu par un objet qui lui répond, l'objet doit d'abord exécuter les actions de sortie de son état actuel, suivies des actions de transition, et finalement les actions d'entrée de l'état suivant. Cet ordre précis d'exécution doit être respecté et l'objet ne peut pas traiter d'autres événements jusqu'à ce que cet ensemble d'actions soit accompli [Cra06].

UML2 propose également un ensemble de *pseudo-états* (pseudostate) permettant la définition d'une sémantique particulière au comportement du modèle. Ces pseudo-états sont représentés par la figure 4.4 et leur fonctionnalité est définie comme suit [Gro04, Dou04] :

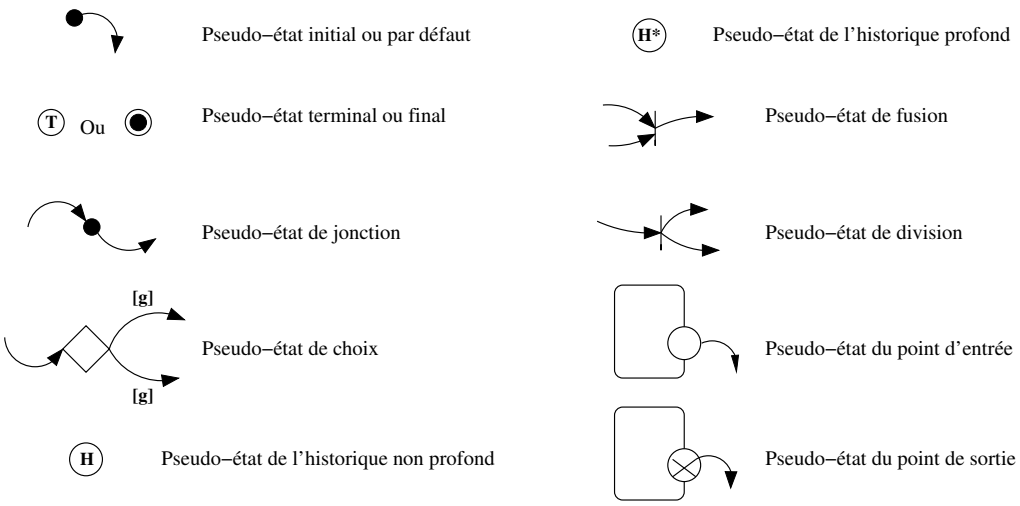


FIG. 4.4 – *Notations des pseudo-états dans un diagramme de machine à états en UML*

- **Pseudo-état initial ou par défaut (initial/default pseudostate).** Permet de spécifier le point de départ ou l'entrée par défaut, notamment dans les états hiérarchiques.
- **Pseudo-état terminal ou final (terminal/final pseudostate).** Représente le point final dans un état composé. Si cet état est atteint, l'objet n'acceptera aucun événement car il peut être considéré comme détruit.
- **Pseudo-état de jonction (junction pseudostate).** Utilisé pour connecter plusieurs transitions ou pour diviser une transition en des segments de transitions séquentiels.
- **Pseudo-état de choix (choice pseudostate).** C'est un type particulier de jonction qui permet l'exécution d'un ensemble d'actions avant de passer aux transitions suivantes. La priorité d'exécution est faite selon l'évaluation des gardes sur les transitions.
- **Pseudo-état de l'historique non profond (shallow history pseudostate).** Indique que

l'état par défaut dans un état composé est le dernier état visité de cet état composé sans inclure les sous-états.

- **Pseudo-état de l'historique profond (deep history pseudostate).** Indique que l'état par défaut dans un état composé est le dernier état visité de cet état en prenant en compte les sous-états.
- **Pseudo-état de fusion (join pseudostate).** Permet de fusionner les transitions venant des états en parallèle en une seule transition.
- **Pseudo-état de division (fork pseudostate).** Branche plusieurs transitions à des états en parallèle à partir d'une seule transition.
- **Pseudo-état du point d'entrée (entry point pseudostate).** Représente un point d'entrée dans un état composé.
- **Pseudo-état du point de sortie (exit point pseudostate).** Représente un point de sortie dans un état composé.

La figure 4.5 représente un exemple simple d'un diagramme de machine à états en UML. Ce diagramme est réalisé en utilisant l'outil de modélisation UML2 appelé MAGIC DRAW[34].

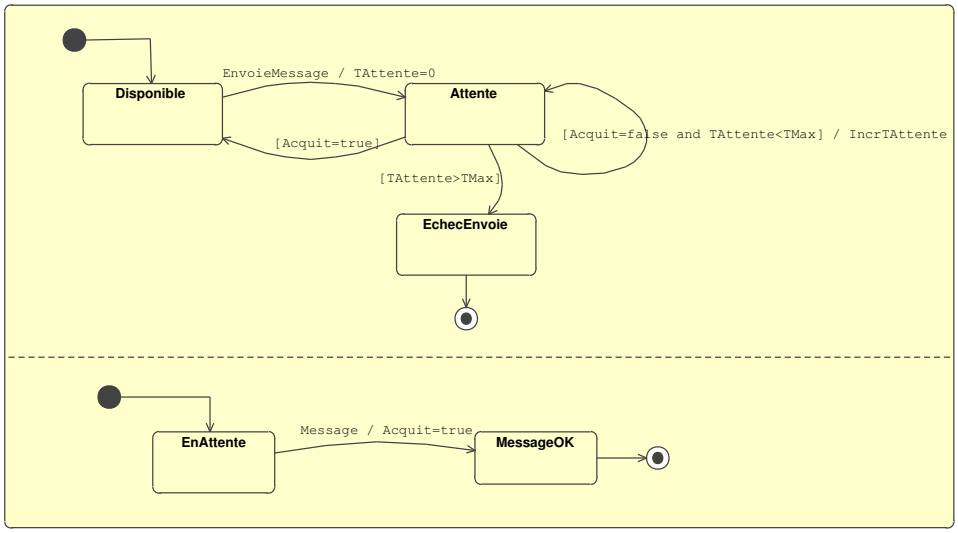


FIG. 4.5 – *Exemple Simple d'un diagramme de machine à états en UML*

4.5.2 Diagrammes d'activités

Le diagramme d'activités représente un cas particulier du diagramme de machine à états dans lequel les états sont définis par des activités et les transitions sont déclenchées par la fin d'une activité et provoquent le début immédiat d'une autre. Un diagramme d'activités est généralement attaché à un objet UML pour représenter son comportement dynamique. Plus spécifiquement, les diagrammes d'activités d'UML2 contiennent des *nœuds* connectés par des *arcs* pour former un graphe de flot complet. Leur sémantique est basée sur la notion de *flot de jeton* (token flow) qui circulent à l'intérieur du graphe. Ainsi, un diagramme d'activités est dit qu'il est dans un certain état (ou nœud) si cet état contient un jeton. Les jetons circulent d'un nœud vers un autre via les arcs et sont manipulées par les actions des nœuds du graphe.

[34] `http://www.magicdraw.com`

Il existe trois types de nœuds dans un modèle de diagramme d'activités : les *nœuds d'actions* (action nodes), les *nœuds de contrôle* (control nodes) et les *nœud d'objets* (object nodes) [Boc03]. La figure 4.6 représente les notations utilisées pour la spécification de ces nœuds dans un diagramme d'activités. Les nœuds d'actions représentent les actions qui

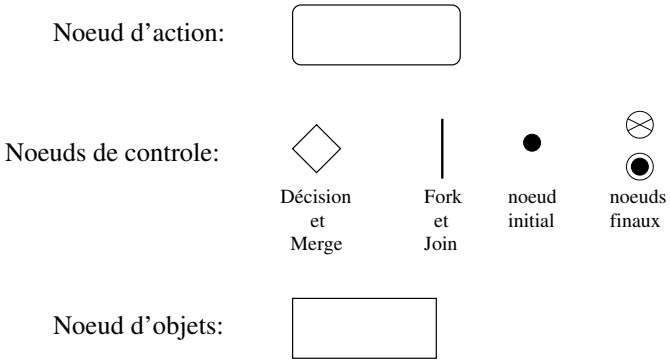


FIG. 4.6 – *Représentation des nœuds dans un diagramme d'activités*

opèrent sur des valeurs de jetons en entrée pour fournir des valeurs en sortie. Les nœuds de contrôle permettent d'orienter les jetons de données et du contrôle dans le graphe. Ces nœuds regroupent des descriptions pour des flots alternatifs (nœud de décision), des flots parallèles (fork), etc. Le fonctionnement des nœuds d'objets consiste à garder temporairement les jetons de données en attendant leur circulation dans le graphe.

De façon générale, les nœud d'un diagramme d'activités peuvent être connectés par deux types d'arcs : les *arcs de flot de contrôle* (control flow edges) et les *arcs de flot d'objets* (object flow edges) [Boc03]. Les arcs de flot de contrôle permettent de connecter les actions en spécifiant que l'action en fin de la connexion (action de destination) ne peut pas commencer son exécution avant que l'action au début de la connexion (action source) soit terminée. Ainsi, ce type d'arc permet uniquement le passage des jetons de type contrôle. Les arcs de flot d'objets quant à eux permettent la spécification des connexions entre les nœuds d'objets pour fournir les entrées aux actions. Les jetons pouvant circuler sur ce type d'arc sont uniquement des jetons de données et d'objets. La figure 4.7 représente les notations utilisées pour la spécifi-

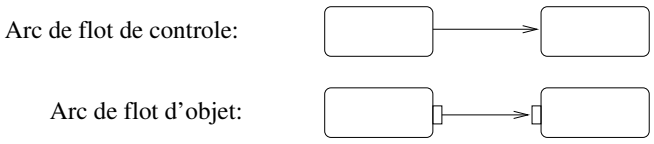


FIG. 4.7 – *Représentation des arcs dans un diagramme d'activités*

cation de ces deux types d'arc. Ainsi, les arcs de flot de contrôle connectent directement les actions, tandis que les arcs de flot d'objets connectent les entrées et les sorties des actions.

Un exemple simple d'une application décrite par un diagramme d'activités est donné par la figure 4.8. Cet exemple est réalisé en utilisant l'outil MAGIC DRAW. Il est à noter que cet outil permet la spécification des deux types d'arcs de flot de contrôle et d'objets, mais le schéma correspondant est le même pour les deux types d'arcs (une simple flèche). Pour des raison de clarté et pour différencier les deux types d'arcs, nous avons proposé de représenter l'arc de flot d'objets par une flèche en pointillé dans notre exemple.

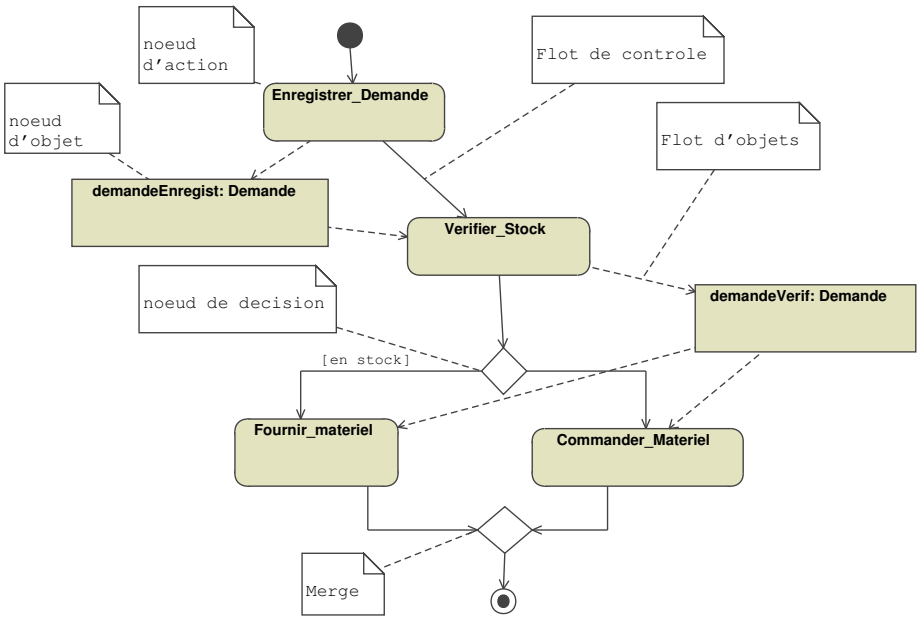


FIG. 4.8 – *Exemple d'un diagramme d'activités*

UML permet donc de représenter graphiquement le comportement d'un objet à l'aide des diagrammes de machines à états et des diagrammes d'activités. Comme nous l'avons spécifié à l'introduction de cette section, le choix d'utilisation de ces diagrammes dépend fortement de la sémantique à vouloir représenter. Ainsi, nous rappelons que les diagrammes de machines à états sont utilisés pour spécifier le comportement d'un objet qui change suite à l'occurrence d'un événement, tandis que les diagrammes d'activités sont plus appropriés pour la description des changement d'états suite à la fin d'une activité. Ces diagrammes sont souvent utilisés pour la description algorithmique du comportement des objets.

4.6 UML et la technologie synchrone

L'étude du rapprochement entre UML et la technologie synchrone, notamment la modélisation UML du comportement des systèmes réactifs synchrones, est récemment apparue comme un sujet de recherche intéressant. Ces travaux résultent principalement de la faiblesse sémantique d'UML et de la puissance d'expression de la technologie synchrone. Un premier travail dans ce domaine est étudié par le constructeur aéronautique Dassault Aviation[35] en 2000 [BNLD00][36]. Ce travail propose l'intégration des modèles synchrones et d'UML en définissant des extensions orientés objets du langage ESTEREL appelé ESTEREL++. Cette extension utilise le modèle SYNCCHARTS pour la représentation du comportement dynamique des objets, et elle est principalement basée sur les diagrammes de structure définis dans ROOM (Real-time Object-Oriented Modeling) [Sel96] et UML/RT (UML for Real-Time) [SR98] en les adaptant aux modèles synchrones. Une possibilité de génération du

[35] `http://www.dassault-aviation.com`
[36] Ce papier est cité dans [BCE+03]

code ESTEREL++ est également intégrée dans l'environnement Rational Rose[37]. Cependant, puisque Rational Rose ne permet pas la représentation des SYNCCHARTS, la simulation et la vérification des modèles sont faites en utilisant l'environnement Esterel Studio[38]. La combinaison de ces outils est supportée par la compagnie Dassault Aviation. C'est une approche jeune et prometteuse, mais qui nécessite toujours plus de développement et d'amélioration pour assurer une bonne intégration d'UML et du paradigme synchrone.

Toujours dans le but de rapprocher le langage UML et la technologie synchrone, C. André *et al.* étudient la conception des systèmes de contrôle en utilisant UML et l'approche synchrone [APR01]. Ils proposent la possibilité d'exprimer des scénarios et des propriétés de vérification pour ces systèmes en étendant les diagrammes de séquences UML, et en utilisant le modèle SYNCCHARTS pour la représentation de leur comportement. Le modèle développé est appelé SIB (Synchronous Interface Behavior) qui dispose de syntaxes graphique et textuelle bien définies et pouvant être mixées pour la spécification de certaines applications. Une traduction du modèle SIB vers le langage ESTEREL est également proposée. Cette traduction est principalement structurelle avec peu de processus d'optimisation, et par conséquent, la taille du code généré est non optimisé et peut être excessive pour de grandes applications réelles.

Dans le même domaine et pour les mêmes objectifs, des travaux ont été entrepris à l'IRISA[39] pour définir un lien entre la modélisation UML et la technologie synchrone. Dans ce contexte, les STATECHARTS et les diagrammes de séquence UML sont traduits vers le formalisme synchrone BDL (Behavioral Description Language) [TBC+98] permettant ainsi la définition d'une liaison entre UML et le paradigme synchrone [WTBG00, Wan01]. Cette transformation autorise des manipulations formelles pour les modèles UML, et définit une passerelle comportementale entre les deux formalismes.

En 2002 et en se basant sur les deux derniers travaux, C. André *et al.* étendent leur approche d'intégration d'UML et du paradigme synchrone en l'appliquant dans le cadre de la conception de contrôleurs embarqués [APR02, PAR02]. Cette approche permet une conception à base d'objets des systèmes étudiés en prenant en compte la sémantique synchrone qui facilite leur vérification formelle. Le principe de base consiste à bénéficier de l'approche synchrone tout en conservant une démarche UML dans l'étude des systèmes. Ceci est réalisé en substituant les diagrammes de séquence UML par le modèle synchrone SIB, et les STATECHARTS UML par le modèle des SYNCCHARTS. L'intérêt de cette approche consiste à proposer des modèles cohérents issus de la même sémantique synchrone et qui respectent la philosophie UML. Ces modèles peuvent aussi être traduits vers le même langage exécutable ESTEREL, et par conséquent bénéficier des outils déjà existants autour de ce langage.

Dans [dA03], R. de Simone *et al.* ouvrent de nouvelles perspectives pour le développement d'un profil UML permettant la représentation du comportement réactif synchrone. Le but de leur approche est, d'une part, de proposer une solution synchrone aux limitations des machines à états dans UML, en particulier la description des événements simultanés et de l'absence d'occurrence d'événements, et d'autre part, de bénéficier des avantages de la technologie synchrone tel que la représentation du temps logique par des instants discrets. Le profil proposé est basé sur une version synchrone des diagrammes d'états et d'activités, et définit une notion d'*instant* (*Clock*) qui est généralement nécessaire pour la spécification

[37] `http://www-306.ibm.com/software/rational`
[38] `http://www.esterel-technologies.com`
[39] `http://www.irisa.fr`

du comportement des systèmes réactifs.

En 2004, 2005, P. Green *et al.* étudie la représentation des formalismes flots de données synchrone en UML2.0, ainsi que leur intégration avec les machines à états UML [GE04, Gre05]. Le but de ce travail est de fournir un langage orienté objet permettant la description des systèmes sur puce plus complexes, dont le comportement est basés sur des machines à état et des modèles flots de données. L'approche définie est principalement inspirée de HASoC et d'UML/RT, et elle est appliquée pour le modèle synchrone SDF (Synchronous DataFlow) [ML02] (voir section 2.3.3, page 21). Cette approche propose l'intégration des SDFGs (Synchronous DataFlow Graphs) dans UML via les diagrammes d'activités et l'introduction d'un nouveau concept appelé *Shell*. Les *Shells* désignent un type particulier d'objet actif et abstrait communicant à travers des ports, et ayant une sémantique particulière d'exécution et de communication. Ce concept est l'élément de base dans le profil développé, et peut avoir trois types différents : *CShell* (Control oriented Shell) dont la description du comportement est basé sur les machines à état et communique à travers des ports de signaux (*CPort*), *DShell* (Data oriented Shell) qui décrit les flots de données et communique à travers des ports de flots de données (*DPort*), et *HShell* (Hybrid Shell) qui identifie un comportement hybride contrôle/données et peut avoir les deux types de ports *CPort* et *DPort*. Il est à noter que le comportement du contrôle est basé sur la sémantique des machines à états UML. Ainsi, pour le modèle hybride, le lien entre SDF et la machine à états est tout simplement défini en représentant les activités SDF sous forme d'activités à l'intérieur d'un état UML, et par conséquent, la machine à état contrôle l'exécution des activités SDF. Le concept du *Shell* peut être appliqué pour des composants matériel ou logiciel, et il est intégré dans la méthode HASoC pour permettre la simulation et la synthèse des systèmes représentés en UML.

Une autre approche combinant UML et la technologie synchrone a été proposé par Esterel Technologies et intégré dans l'environnement de développement SCADE [LD06]. Le but de cette approche est de faciliter l'étude et la conception des systèmes embarqués en utilisant, d'une part, UML pour la spécification à haut niveau des fonctionnalités et de l'architecture du système, et d'autre part, Scade pour la description formelle et la vérification du comportement attendu. Le lien entre ces deux modèles a été établi en faisant référence à la similarité syntaxique des diagrammes de bloc dans UML et ceux dans SCADE. La principale partie de l'approche consistait à étudier les différences sémantiques des deux modèles, notamment pour la représentation des facteurs de temps. Ainsi, l'idée de base de cette approche est de pouvoir raffiner des diagrammes UML par des spécifications de comportement à la SCADE permettant ainsi la spécification des systèmes embarqués à différents niveaux d'abstractions.

Un autre travail intéressant qui rapproche la modélisation UML et la technologie synchrone est introduit via la modélisation du temps logique en UML [ACdT06]. L'objectif de ce travail consiste à proposer un profil UML prenant en compte la notion d'horloge pour la modélisation des événements, des actions et des objets. La notion d'horloge introduite est similaire de celle utilisée pour la définition du comportement réactif synchrone, et peut être utilisée dans différentes vues UML.

La partie qui nous intéresse dans notre travail est la modélisation du contrôle et du comportement réactif en UML, et leur intégration dans la chaîne de développement de GASPARD2. Pour cela, nous proposons d'étudier la modélisation des automates de contrôle, voire les automates synchrones, dans UML.

4.7 Synthèse et conclusion

Nous avons présenté dans ce chapitre l'intérêt et les principes de base de l'approche IDM pour le développement des systèmes informatiques, ainsi que l'utilisation du langage UML pour la co-modélisation des systèmes sur puce. Dans ce domaine, nous avons présenté l'environnement de développement GASPARD2, conçu pour la co-conception des systèmes embarqués sur puce, et basé sur une extension du modèle ARRAY-OL pour faciliter l'étude des applications de traitement du signal intensif. Cet environnement représente le contexte général de notre travail, il suit la démarche MDA, et produit une chaîne de conception en Y pour les systèmes sur puce. La contribution de ce travail se situe en haut de cette chaîne de conception. Un de ses objectifs consiste à proposer une modélisation à haut niveau, via un profil UML, des concepts de contrôle et leur intégration dans des applications de traitement parallèles. Nous rappelons que le but principal de notre travail est d'introduire la notion du contrôle et des changements de mode dans des applications de traitement intensif à parallélisme massif en se basant sur la technologie synchrone. Pour cela, nous avons étudié le lien entre la modélisation UML et la technologie synchrone, notamment la modélisation des automates de contrôle en UML.

Dans le cadre de notre travail, nous nous intéressons en particulier à la modélisation des changements de comportement d'un système selon l'occurrence des événements discrets. Les diagrammes de machine à états paraissent donc plus appropriés pour répondre à nos besoins. Nous avons mentionné que ces diagrammes sont basés sur une sémantique « *run-to-completion* » qui impose une atomicité dans l'exécution des événements. Cette hypothèse d'atomicité permet de simplifier la fonction de transition puisque les conflits de simultanéité sont évités pendant le traitement des événements. Cependant, dans la plupart des systèmes réactifs, cette notion d'événements simultanés est importante et doit être prise en considération pour la spécification du comportement de ces systèmes. De plus, les machines à états en UML ne prennent pas en considération l'absence des événements ce qui peut rendre difficile l'expression du comportement des systèmes réactifs plus complexes. La sémantique des machines à états dans UML est donc insuffisante pour représenter touts les comportements des systèmes réactifs synchrones. Par ailleurs la relation entre les différents types de diagrammes n'est pas toujours bien formalisée en UML, notamment en ce qui concerne le lien entre les diagrammes de structure et les diagrammes de machine à états. Or, pour les applications que nous envisageons, cette cohérence entre structure statique et comportement réactif est intéressante pour comprendre la fonctionnalité des systèmes étudiés.

Il est donc nécessaire d'introduire de nouvelles approches et d'étendre la sémantique des diagrammes de machine à états en UML pour, d'une part, faciliter l'expression du comportement des systèmes réactifs synchrones, et d'autre part, définir un lien cohérent entre le modèle structurel et le comportement dynamique de ces système. Comme nous l'avons mentionné dans la section précédente, le travail de R. de Simone *et al.* part dans cette direction [dA03].

Il est aussi à noter que la réalisation de notre approche est principalement basée sur le modèle de description parallèle ARRAY-OL, ce qui rend nécessaire l'étude de la relation entre ce modèle et la représentation du contrôle en UML. L'étude de la modélisation du contrôle en UML ainsi que la modélisation du lien entre la partie contrôle et les applications parallèles en GASPARD2 sont présentées plus en détails dans le chapitre 8.

Deuxième partie

Méthodologie de Séparation Contrôle/Données

Chapitre 5

Méthodologie de séparation contrôle/données pour la conception des systèmes hybrides

Nous allons présenter dans ce chapitre notre méthodologie de séparation contrôle/données pour la conception des systèmes réactifs synchrones et hybrides. Cette méthodologie est principalement basée sur le concept des automates de modes. Son objectif est de permettre, d'une part, une meilleure lisibilité et une bonne réutilisation des différentes parties du système, et d'autre part, de faciliter le processus de vérification et de tirer profit des différents outils déjà existant, dédiés exclusivement au traitement de la partie contrôle ou de la partie données. Nous discutons également l'intérêt et les avantages de cette méthodologie de séparation, notamment pour l'application du processus de vérification formelle. Ainsi, pour mieux comprendre et illustrer ces concepts, nous allons présenter notre démarche en nous basant sur un exemple réel, et en utilisant l'environnement de développement SCADE.

5.1 Introduction

Dans le chapitre 3, nous avons mentionné que la majorité des systèmes réactifs synchrones sont de nature hybride dans le sens où ils regroupent des traitements de données et du contrôle. L'étude et la conception de ces système nécessitent alors des méthodes et des outils spécifiques et rigoureux permettant la prise en compte de cette mixité de traitement. Les approches multi-langages, transformationnelles et multi-styles, introduites dans le chapitre 3, représentent des exemples de base de méthodes de conception pour les systèmes hybrides. Cependant, ces approches n'imposent aucune méthodologie de séparation entre la partie contrôle et les différentes parties de calcul, et donnent plus de liberté aux développeurs pour la conception de leurs systèmes. Ceci rend généralement difficile la compréhension de l'application, sa vérification, ainsi que l'échange d'informations et la réutilisation des différentes parties du système.

Nous rappelons que dans le cadre de notre travail, nous nous intéressons en particulier à l'étude des systèmes réactifs dont le comportement est principalement régulier, mais qui peuvent changer instantanément d'un comportement à un autre en répondant à des événements de contrôle. Ce sont des systèmes à mode de fonctionnement où chaque mode représente un comportement différent du système. Nous avons mentionné dans la section 3.3.3 (page 47) que les automates de modes [MR98] permettent la description des systèmes réactifs à modes de fonctionnement en ajoutant une structure d'automate au langage flot de données LUSTRE. Ceci correspond à la description du système par un automate dont les états représentent les modes et les transitions les changements entre ces modes. Ainsi, chaque état correspond à une description d'un programme en LUSTRE pour le mode en question. Les automates de modes permettent donc la prise en compte des systèmes réactifs hybrides. Cependant, leur structure ne permet pas de bien isoler l'automate de contrôle des différentes parties de traitement, notamment pour des automates plus complexes ayant des descriptions hiérarchiques et parallèles. De plus, le concept des automates de modes a été uniquement réalisé dans le contexte du langage LUSTRE limitant ainsi le domaine des applications étudiées à celles pouvant être spécifiées par ce langage.

Dans notre étude, nous proposons une méthodologie de conception inspirée du principe des automates de modes, non liée à un langage de spécification particulier, et basée sur une séparation claire entre la partie de contrôle et les parties de calcul. Notre démarche part du constat que les différents modes des systèmes étudiés sont complètement indépendants et exclusifs, et que ces systèmes sont généralement pilotés par une partie de contrôle qui joue le rôle d'un chef d'orchestre pour coordonner l'exécution des différentes parties de

l'application. Le caractère indépendant de ces différentes parties du système permet donc de distinguer les modes de fonctionnement et de les séparer de la partie contrôle. Cette séparation donne un modèle plus clair, plus facile à maintenir et à vérifier, et favorise l'échange et la réutilisation des applications existantes.

Dans [LDB05], nous avons proposé un modèle de séparation contrôle/données permettant la distinction des différents modes d'un système, et dans lequel l'activation de ces modes est faite par un automate de contrôle en fonction des valeurs des événements. Ces événements peuvent être fournis par l'environnement de l'application (appui sur un bouton, changement de température, etc.), ou par la partie de traitement (résultat d'un calcul interne, dépendance entre tâches, etc.). Une vue globale de ce modèle est donnée par la figure 5.1.

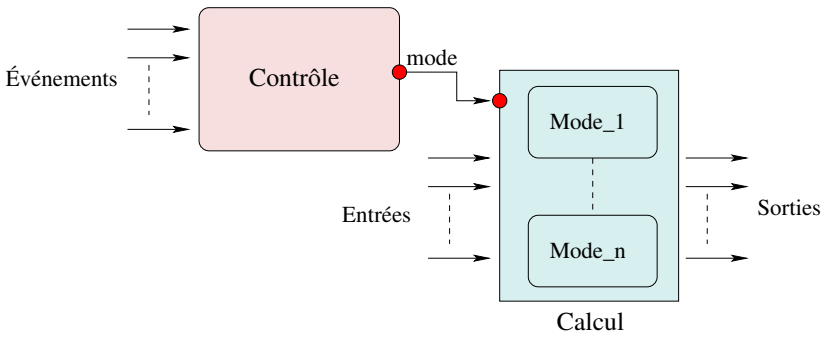


FIG. 5.1 – *Vue globale du modèle de séparation contrôle/données*

Dans ce qui suit, et pour mieux illustrer notre méthodologie de séparation entre la partie contrôle et les différents modes de fonctionnement d'un système, nous allons présenter notre démarche via une étude de cas et en prenant comme support de développement l'environnement SCADE [Tec03a] (présenté dans la section 3.3.2, page 44).

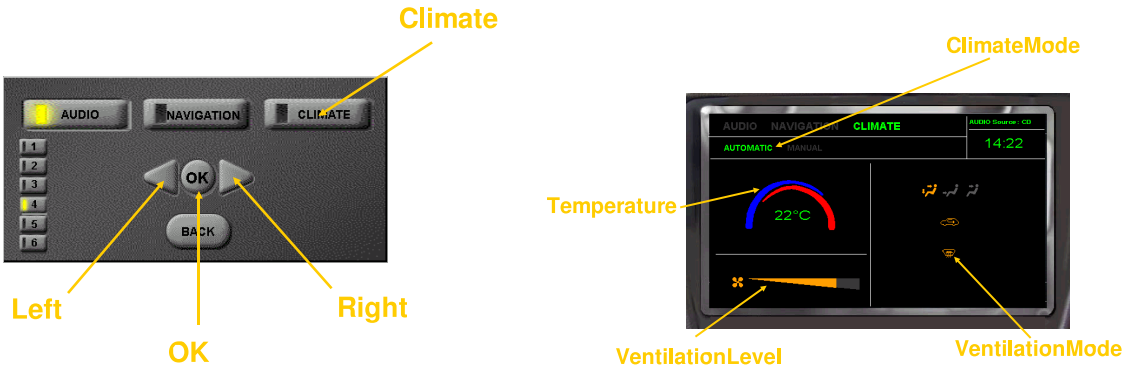
5.2 Problématique de la combinaison contrôle/données

Dans cette section, nous allons montrer, à travers l'étude d'un exemple de climatisation dans une voiture, que le développement des applications hybrides selon les approches précédemment citées, et en particulier dans l'environnement de développement SCADE, n'impose pas de méthodologie de séparation claire entre la partie contrôle et les différents modes de fonctionnement d'un système. Nous allons aussi montrer que cette combinaison peut poser un certain nombre de problèmes, ce qui peut rendre difficile l'étude et la maintenance de l'application, et par conséquent, augmenter son coût de développement.

Le choix de l'environnement SCADE comme outil de développement vient, d'une part, du fait que le principe de fonctionnement de SCADE est similaire à celui des autres approches de conception hybride dans le sens où ils n'imposent pas de contraintes sur la combinaison contrôle/données, et d'autre part, du fait que SCADE facilite la modélisation graphique des applications étudiées, et propose un ensemble d'outils de simulation et de vérification qui nous sont utiles pour l'étude et la validation du comportement des applications étudiées.

5.2.1 Exemple d'application : climatisation dans une voiture

Nous avons choisi d'étudier un exemple simplifié d'un système de climatisation dans une voiture, appelé `Climate`, combinant des traitements de données et du contrôle. Dans cet exemple, le système répond à quatre entrées différentes de type booléen. Ces entrées correspondent aux boutons `Climate`, `Left`, `Right` et `Ok` comme il est montré par la figure 5.2.(a). Comme valeurs en sortie, le système affiche le mode de climatisation (`ClimateMode`), la valeur de la température (`Temperature`), le niveau de ventilation (`VentilationLevel`) et le mode de ventilation (`VentilationMode`) comme il est montré par la figure 5.2.(b). Les types des valeurs en sorties sont comme suit :



(a) Entrées (b) Sorties

FIG. 5.2 – *Représentation des entrées/sorties dans `Climate`*

- `ClimateMode` est de type énuméré ayant comme valeur `Auto` ou `Manual`, initialisé à `Auto`,
- `Temperature` est un entier qui prend ses valeurs dans l'intervalle $[17, 27]$, initialisée à 19,
- `VentilationLevel` est un entier qui prend ses valeurs dans l'intervalle $[0, 100]$, initialisé à 0,
- et `VentilationMode` est de type énuméré qui prend ses valeurs dans l'ensemble {`CAR`, `FACE`, `FEET`, `DEFROST`, `CIRCULATION`}, initialisé à `CAR`.

À l'initialisation, le système de climatisation est en mode `Auto`. Le changement entre les modes `Auto` et `Manual` est toujours précédé par l'état `Adjust` qui permet de confirmer le choix du mode via le bouton `Ok` (figure 5.3).

Dans le mode `Auto`, le système permet de changer la valeur de la température et de basculer vers l'état `Adjust`. Le fonctionnement des différents boutons dans ce mode est comme suit :

- le bouton `Left` décrémente la température par 1 jusqu'à 17,
- le bouton `Right` incrémente la température par 1 sans dépasser 27,
- et le bouton `Climate` permet de passer à l'état `Adjust`.

Le mode `Adjust` permet de basculer à travers les boutons `Left` et `Right` entre les différents modes de ventilation et de climatisation dans l'ordre suivant : `CAR`, `FACE`, `FEET`, `DEFROST`, `CIRCULATION`, `AUTO`, `MANUAL`. Dans ce mode, le bouton `Ok` sélectionne le mode à activer et quitte l'état `Adjust`. Ainsi, le système bascule vers le mode `Auto` si `ClimateMode=AUTO`, sinon il passe au mode `Manual`. Dans le mode `Manual`, le système est capable de modifier le niveau

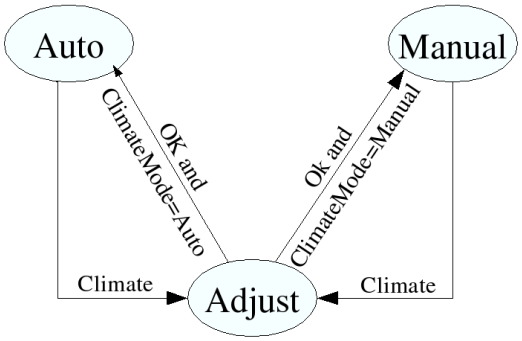


FIG. 5.3 – *Les différents états du système* `Climate`

de ventilation et passer en mode `Adjust`. Le fonctionnement des différents boutons dans le mode `Manual` est comme suit :

- le bouton `Left` décrémente le niveau de ventilation par 1 jusqu'à 0,
- le bouton `Right` incrémente le niveau de ventilation par 1 sans dépasser 100,
- et le bouton `Climate` permet de passer au mode `Adjust`.

5.2.2 Conception du système de climatisation dans SCADE

La spécification du système de climatisation décrite dans la section précédente montre que ce système contient un mélange de traitement de données et du contrôle. Une solution proposée par Esterel Technologies[40] pour la conception du système `Climate` en utilisant l'environnement de développement SCADE est représentée par la figure 5.4 [Tec03b]. Ce système

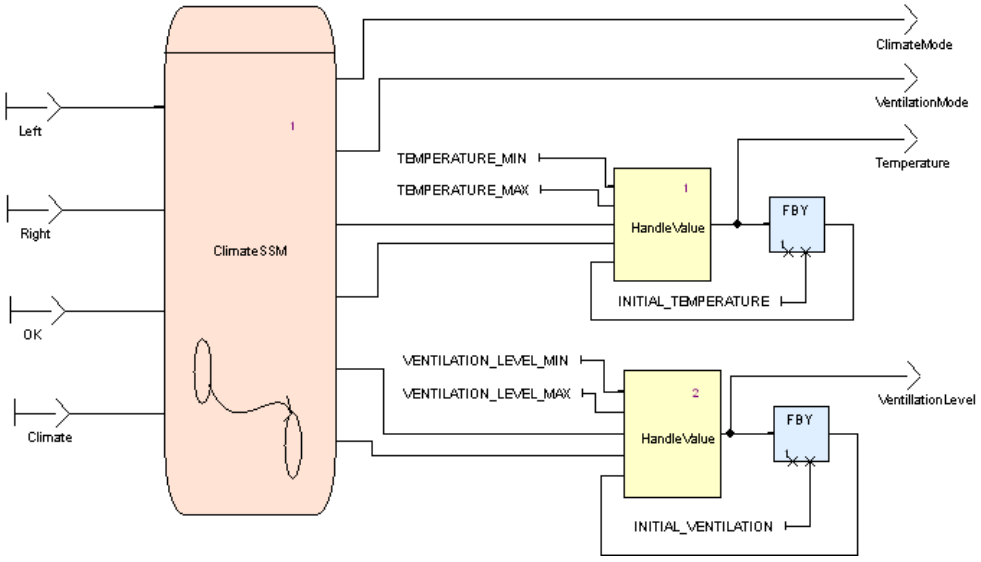


FIG. 5.4 – *Spécification du* `Climate` *dans* SCADE

possède quatre entrées relatives aux boutons `Left`, `Right`, `Ok` et `Climate`. En sortie, il four-

[40] `http://www.esterel-technologies.com`

nit quatre résultats : `ClimateMode`, `VentilationMode`, `Temperature` et `VentilationLevel`. Les valeurs en entrées sont consommées par une partie de contrôle représentée par la machine à état (SSM : Safe State Machine [And03]) `ClimateSSM` de la figure 5.5. Chaque état dans cette

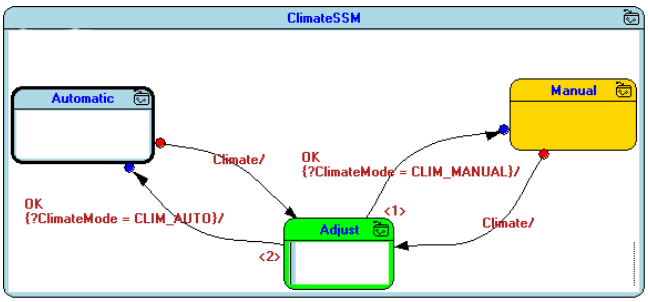


FIG. 5.5 – *ClimateSSM*

SSM est représenté par un *macro-état* (macro-state) spécifiant le comportement d'un état hiérarchique. Il est à noter que la structure de cette SSM est similaire à celle de l'automate représenté par la figure 5.3. Cependant, le comportement de `ClimateSSM` est complètement différent puisque les macro-états ne représentent pas uniquement des états simple de l'automate, mais ils décrivent également des structures plus complexes spécifiant le comportement du système. De façon générale, le fonctionnement de la SSM `ClimateSSM` consiste à fournir comme résultat les valeurs de modes de climatisation et de mode de ventilation (`ClimateMode` et `VentilationMode`). Elle permet aussi d'activer la partie de calcul `HandleValue` par deux signaux différent : `Incr` et `Decr` correspondant respectivement à l'incrémentation et à la décrémentation de la valeur de température ou de celle du niveau de ventilation. L'activation du composant `HandleValue` dépend des valeurs en entrées (les boutons appuyés) et de l'état courant du système.

L'opérateur `FBY` (followed by), apparaissant dans la figure 5.4, est un opérateur temporel prédéfini dans SCADE qui permet de conserver la valeur d'une expression donnée sur plusieurs cycles. Dans SCADE, `FBY`(E, n, Init) est équivalent en LUSTRE à

$$\text{Init} \rightarrow \text{pre}(\text{Init} \rightarrow \text{pre}(\cdots \rightarrow \text{pre}(\text{E})))$$

où E est une expression qui définit une séquence $(e_1, e_2, \ldots e_n)$ et n est une expression statique dont la valeur est strictement positive. L'opérateur pre (pour précédent) de LUSTRE permet de mémoriser et de faire référence à la valeur dans l'instant précédent. Dans le système `Climate`, `FBY` permet de garder la valeur précédente de `Temperature` ou de `VentilationLevel` pour les transmettre à l'opérateur `HandleValue`. À l'initialisation, l'opérateur `FBY` transmet la valeur initiale de la température (`INITIAL_TEMPERATURE`) ou celle du niveau de ventilation (`INITIAL_VENTILATION`).

La figure 5.4 montre que le système est composé d'une partie de contrôle `ClimateSSM` et de deux instances d'une même partie de calcul `HandleValue`. Dans ce modèle, il n'est pas possible de distinguer les trois modes de fonctionnement du système de climatisation (`Auto`, `Adjust` et `Manual`). Ainsi, si nous souhaitons modifier le comportement d'un des trois modes, ajouter ou supprimer un mode, l'application doit être complètement réécrite, ce qui est souvent très coûteux en terme de temps et de ressources de développement.

En descendant dans un niveau plus bas de la hiérarchie, le schéma de conception correspondant à l'opérateur `HandleValue` est représenté par la figure 5.6. Cet opérateur per-

met d'incrémenter ou de décrémenter une valeur donnée en fonction des valeurs des signaux `Incr` et `Decr`[41]. Dans ce schéma, nous remarquons que la partie de calcul `HandleValue`

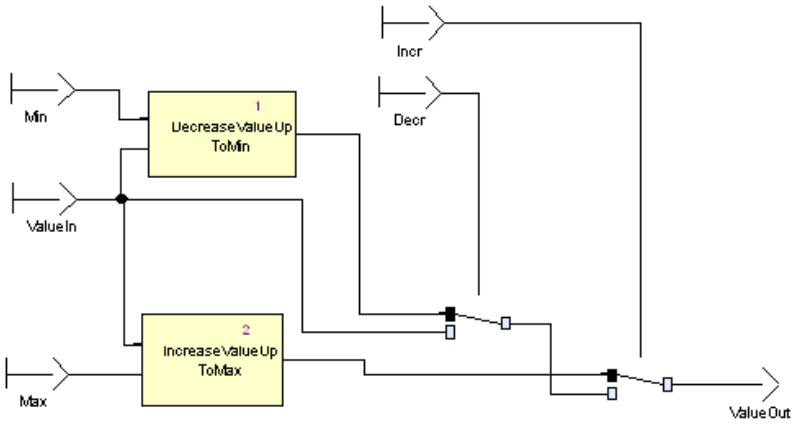


FIG. 5.6 – *Spécification de l'opérateur* `HandleValue`

contient un mélange de calcul (`DecreaseValueUpToMin` et `IncreaseValueUpToMax`) et du contrôle (`If_Then_Else`). Ce mélange peut rendre difficile la compréhension, la manipulation et la réutilisation de différentes parties de l'application, ainsi que l'utilisation des outils déjà existants, dédiés exclusivement au traitement de la partie contrôle ou de la partie calcul. Il est aussi à noter que, indépendamment des valeurs des signaux `Inc` et `Dec`, les deux parties de calcul `DecreaseValueUpToMin` et `IncreaseValueUpToMax` sont activées et la valeur en sortie sera choisie par la suite en fonction des valeurs de `Inc` et `Dec`. Ceci correspond à la nature ternaire et stricte de la structure conditionnelle `If_Then_Else` dans LUSTRE. Dans ce cas, les deux branches de la structure conditionnelle sont toujours évaluées ce qui peut induire des problèmes d'effets de bord.

L'exemple de `Climate` que nous avons présenté montre que la spécification du système dans SCADE ne montre pas clairement l'aspect indépendant et exclusif des différents modes de fonctionnement du système, et ne représente pas la partie de contrôle comme module global pilotant le changement entre ces modes. Le but de notre travail est donc de proposer un modèle de conception permettant d'avoir une séparation claire entre la partie contrôle et les différentes parties de calcul, et de faciliter l'étude séparée et la réutilisation des différentes parties du système.

5.3 Méthodologie de séparation contrôle/données

Dans cette section, nous allons présenter la démarche suivie pour la définition de notre méthodologie de conception permettant de séparer clairement entre la partie contrôle et les différentes parties de calcul.

5.3.1 Séparation contrôle/données en utilisant SCADE

En premier lieu, nous avons essayé d'appliquer le concept de séparation contrôle/données en utilisant l'environnement de développement SCADE. Pour ce faire, nous avons étu-

[41] `Incr` et `Decr` ne peuvent pas être valides en même temps.

dié l'exemple de la climatisation dans une voiture (`Climate`) en séparant dès le départ la partie contrôle de la partie calcul. Il est à noter que le modèle proposé peut être hiérarchique où, à chaque niveau de la hiérarchie, les éléments de contrôle qui permettent de piloter l'activation des différents parties de traitement seront séparés de ces parties de traitement et regroupés dans une seule partie de contrôle. Le schéma de conception correspondant à notre démarche est représenté par la figure 5.7. Dans cet exemple, nous avons divisé le système

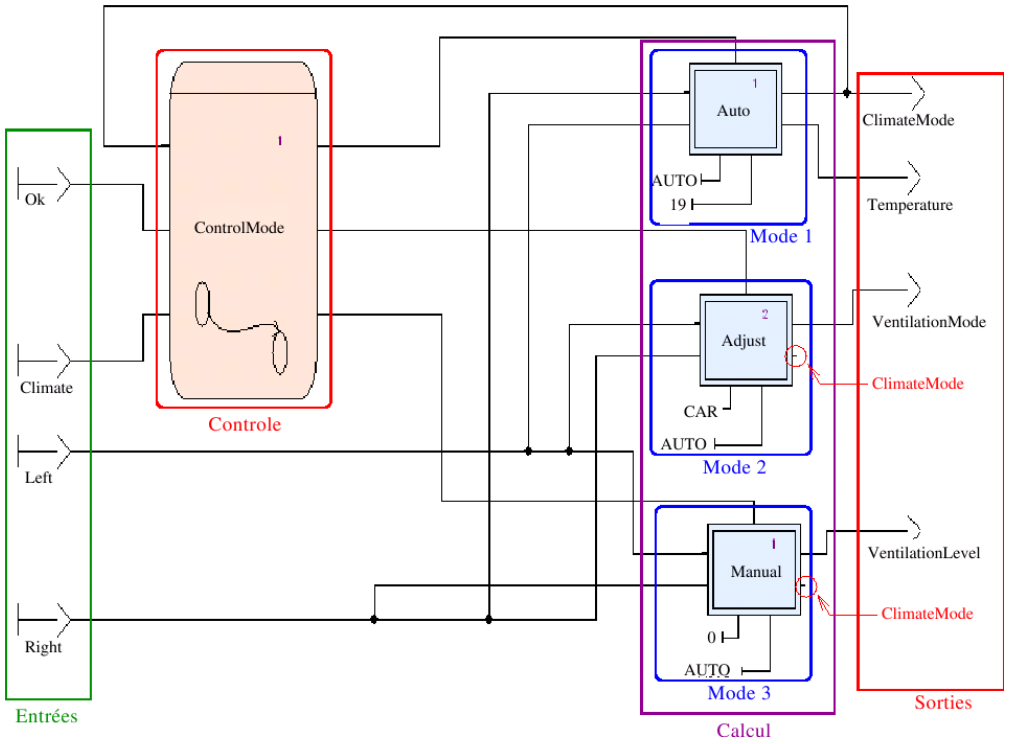
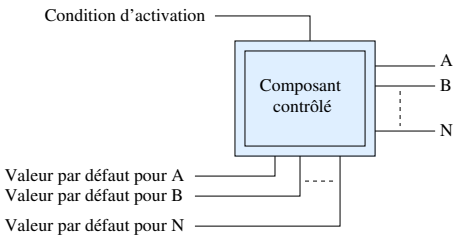


FIG. 5.7 – *`Climate` : séparation contrôle/données dans* SCADE

en trois sous-systèmes qui correspondent aux trois différents modes de l'application : `Auto`, `Adjust` et `Manual`.



Les différents modes de l'application sont représentés par un cadre doublé dans la figure 5.7 relatif à l'expression des *opérateurs conditionnés* dans SCADE. Un opérateur conditionné ne peut être activé que si sa condition d'activation est évaluée à *vrai*. Ainsi, si cet opérateur n'est pas activé (la condition d'activation vaut *faux*), l'opérateur fournit comme résultat les valeurs par défaut spécifiées pour chacune de ses sorties.

L'activation de chaque mode de fonctionnement du système `Climate` est faite par la machine à états `ControlMode`, représentée par la figure 5.8, en fonction des valeurs en entrées de `Ok` et de `Climate`. À la différence de la machine à états `ClimateSSM` représentée par la figure 5.5, la structure de `ControlMode` est composée des états simples dont le rôle consiste uniquement à activer un des trois modes de fonctionnement du système sans effectuer d'autres calculs.

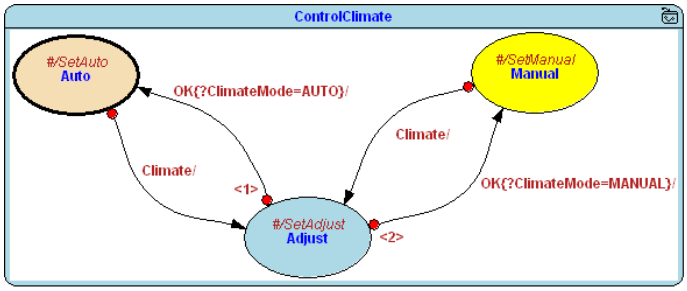


FIG. 5.8 – *Automate de contrôle* `ControlMode`

L'idée de base de notre approche consiste alors à diviser la spécification du système en plusieurs parties contrôlables par une machine à état. Dans ce contexte, Les différentes parties du système représentent les modes de fonctionnement qui sont indépendants et exclusifs, tandis que la machine à états représente la partie qui pilote l'activation de ces modes en fonction des valeurs des événements et de l'état courant du système. Dans cette approche, nous pouvons clairement distinguer les entrées et les sorties du système, la partie contrôle et la partie de calcul. Contrairement à ce que son nom indique, la partie de calcul ne dé-

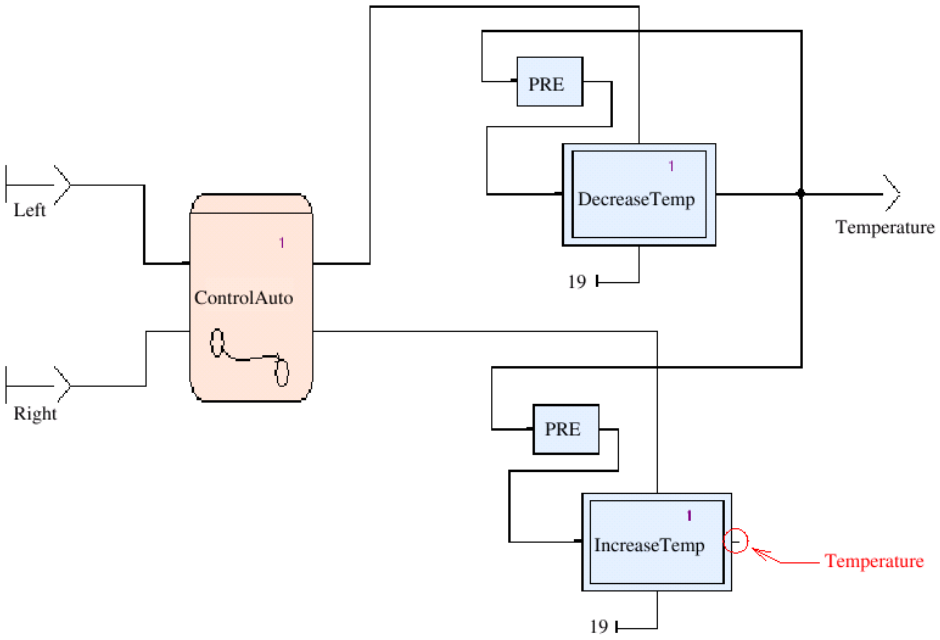


FIG. 5.9 – *Description de l'opérateur* `Auto`

signe pas uniquement un traitement de données exclusif. Elle peut aussi contenir une partie de contrôle suivie par d'autres sous-parties de calcul comme dans le cas du mode `Auto` représenté par la figure 5.9, ou uniquement du contrôle comme dans le cas du mode `Adjust` représenté par la figure 5.10. Il est aussi à noter que la partie de contrôle peut être précédée par une partie de traitement de données. Ceci est utile dans le cas où la machine à états prend comme valeurs en entrée les résultats d'une partie de calcul. Ainsi, notre modèle est complètement hiérarchique, où le niveau le plus bas de la hiérarchie représente une partie homogène contenant exclusivement une partie de contrôle ou une partie de calcul.

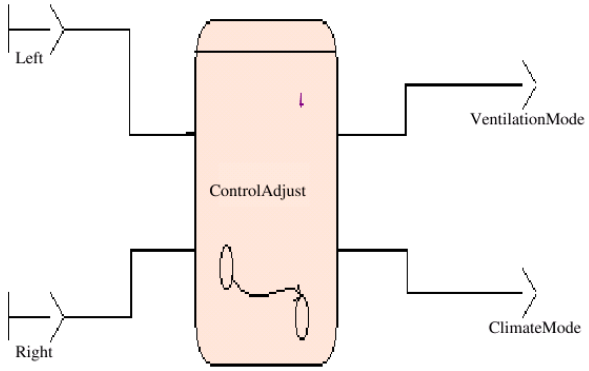


FIG. 5.10 – *Description de l'opérateur* `Adjust`

L'application de notre approche de séparation contrôle/données dans SCADE nous a confrontés à un certain nombre de problèmes. Par exemple, la valeur de `ClimateMode` peut être modifiée par les trois modes du système : `Auto`, `Adjust` et `Manual` (figure 5.7), et de même, la valeur de `Temperature` dans le mode `Auto` peut être modifiée par deux opérateurs différents : `DecreaseTemp` et `IncreaseTemp` (figure 5.9). Cependant, dans `Scade` il n'est pas possible de lier une même sortie à deux opérateurs différents. Dans cet environnement, chaque donnée ne peut et ne doit avoir qu'une unique définition à un instant donné, ce qui rend impossible la connexion d'une même sortie à plusieurs opérateurs différents. La résolution possible de ce problème dans SCADE nécessite l'utilisation des opérateurs `If_Then_Else`, ce qui peut compliquer le modèle de conception et casse notre concept de séparation contrôle/données.

Pour éviter ce problème, nous avons proposé d'ajouter des opérateurs spéciaux qui jouent le rôle de `Fork` (F) et `Join` (J) pour permettre le partage des données entre plusieurs opérateurs. Nous avons aussi ajouté un opérateur spécial appelé `Selector` (S) qui reçoit en entrée une valeur produite par la partie contrôle, en fonction de laquelle il peut choisir le mode à activer. L'application de ces concepts pour l'exemple de `Climate` donne le modèle de conception de la figure 5.11. Dans ce modèle, le fonctionnement de l'opérateur `Fork` consiste à diffuser la valeur reçue en entrée sur toutes ses branches en sortie, tandis que le rôle de l'opérateur `Join` consiste à choisir une valeur en sortie parmi celles reçues en entrée, et en fonction de la valeur fournie par la machine à états.

Les opérateurs `Selector` et `Fork` représentent uniquement une optimisation des notations utilisées dans SCADE puisque, dans cet outil, il est possible de connecter la même valeur à plusieurs entrées d'opérateurs différents. Cependant, l'opérateur `Join` permet de remplacer les structures conditionnelles `If_Then_Else` et `Switch_Case` utilisées dans SCADE. Dans ce contexte, un opérateur `Join` avec n entrées peut être utilisé pour remplacer $n-1$ opérateurs `If_Then_Else` ou un seul opérateur *Switch_Case* avec n entrées.

Dans le cas des opérateurs `If_Then_Else`, il est évident que la complexité du modèle augmente avec le nombre de valeurs en entrées ce qui rend difficile la compréhension et la manipulation du modèle. Ainsi, si nous utilisons l'opérateur `Switch_Case`, les différents blocs de calcul ne sont pas conditionnés, et par conséquent, toutes les entrées doivent être calculées avant que l'opérateur puisse choisir la valeur sélectionnée selon la condition du `Switch`. Ce comportement mène souvent à des problèmes difficilement contrôlables et peut être très coûteux en terme de temps et de capacité mémoire. L'utilisation des opérateurs

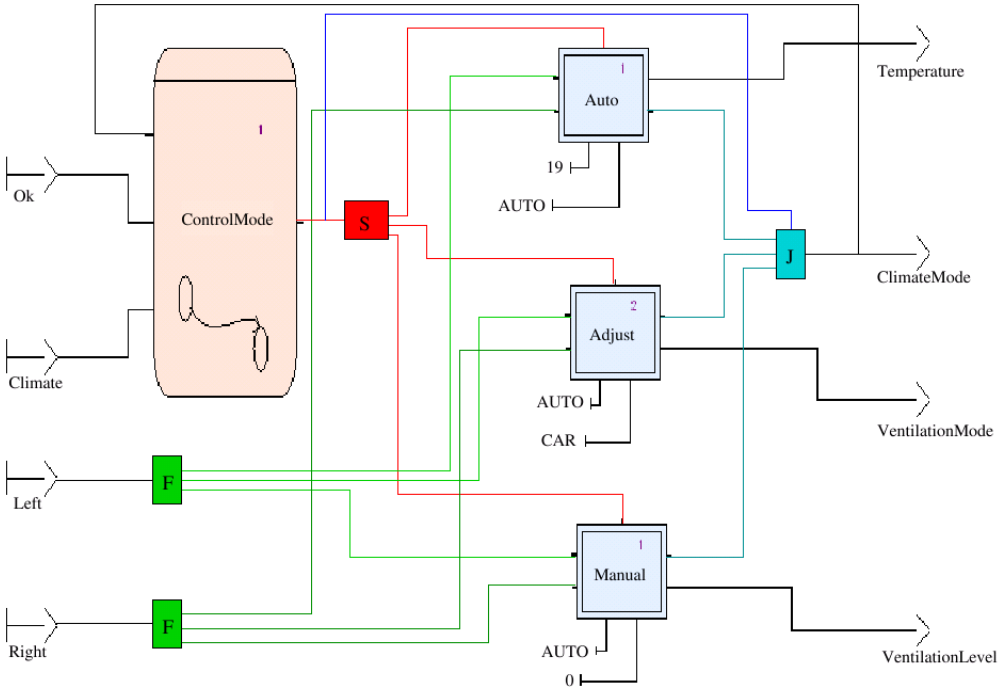
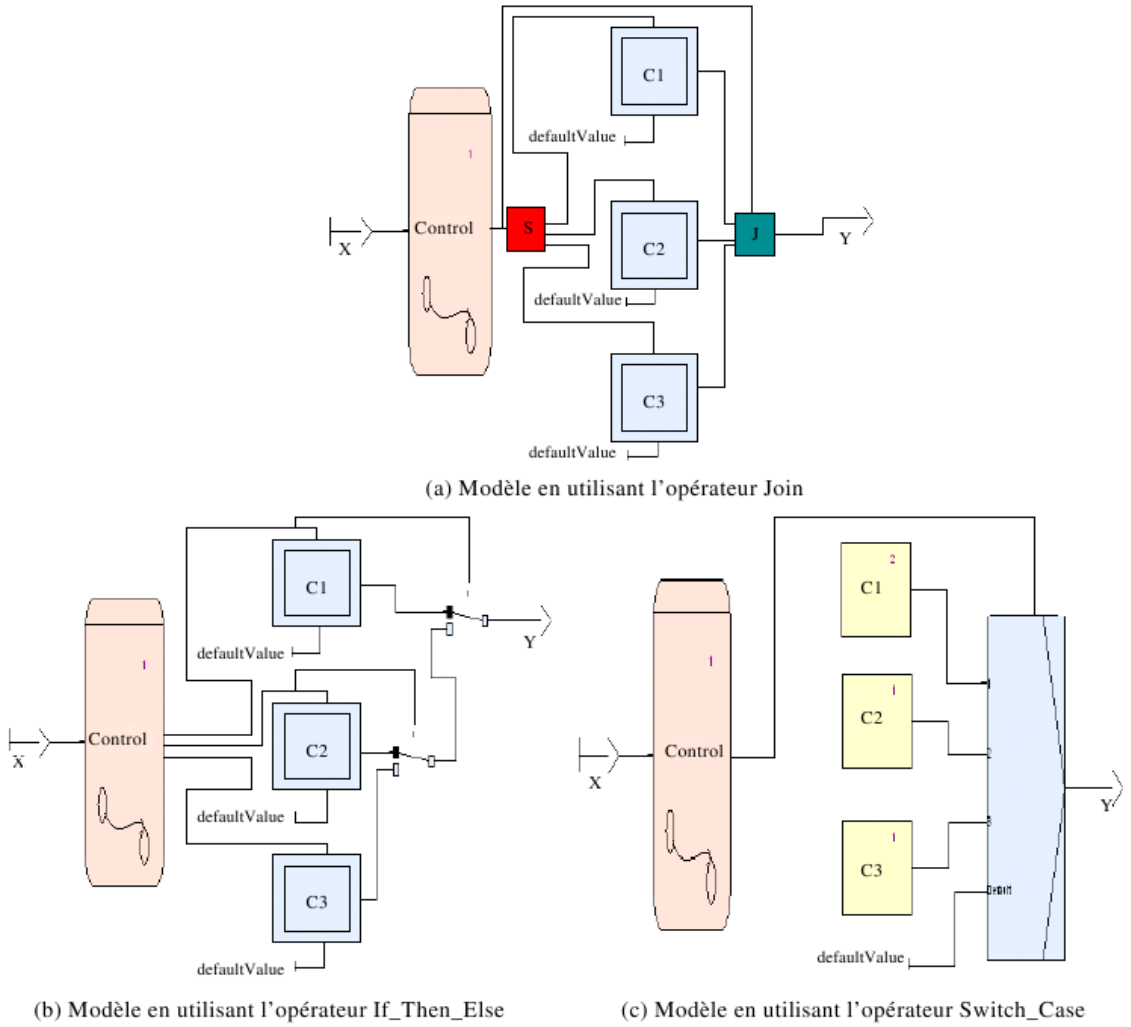


FIG. 5.11 – *Modèle de séparation contrôle/données en utilisant les opérateurs* `Fork`, `Join` *et* `Selector`

`Selector` et `Join` permet d'éviter ces problèmes en proposant une utilisation implicite des structures conditionnelles qui facilite la compréhension du modèle. La figure 5.12 donne un exemple d'un opérateur `Join` et ses représentations équivalentes dans SCADE.

Notre démarche consiste alors à étudier les systèmes à modes de fonctionnement en divisant leur spécification en plusieurs parties indépendantes et exclusives contrôlables par une machine à état. Cette dernière permet de choisir le mode de fonctionnement à activer en fonction des événements en entrées et de l'état du système. Ainsi, la structure de l'automate de contrôle que nous proposons d'utiliser est inspirée de celle des automates de modes et des automates de Moore [Moo56]. L'utilisation du principe des automates de modes permet de bien spécifier les différents modes de fonctionnement d'un système et les conditions de changement de modes. Tandis que le principe des automates de Moore est utilisé pour limiter la sortie de l'automate à l'état dans lequel il se trouve. Dans ce type d'automates, la relation de transition spécifie uniquement des événements permettant le passage d'un état à un autre sans avoir aucun effet sur le résultat en sortie de l'automate.

Dans le modèle de séparation contrôle/données que nous avons présenté pour le système `Climate`, nous avons divisé dès le départ la spécification globale du système en plusieurs modes de fonctionnement. Une telle séparation stricte est intéressante pour la modélisation de certains systèmes dont le comportement est principalement régulier et dont la distinction des différents modes est évidente. Cependant, il existe des systèmes dont la partie contrôle ne concerne pas le système global mais plutôt des sous parties de ce système. Dans ce cas, la séparation de la spécification globale du système en plusieurs parties pilotées par un automate de contrôle devient complexe, et peut introduire des problèmes de redondance et conduire à des erreurs incontrôlables. Pour faire face à ce problème, nous proposons d'utiliser notre méthodologie de séparation contrôle/données uniquement pour les sous-parties



(a) Modèle en utilisant l'opérateur Join

(b) Modèle en utilisant l'opérateur If_Then_Else

(c) Modèle en utilisant l'opérateur Switch_Case

FIG. 5.12 – *Exemple d'un opérateur* `Join` *et ses équivalents dans* SCADE

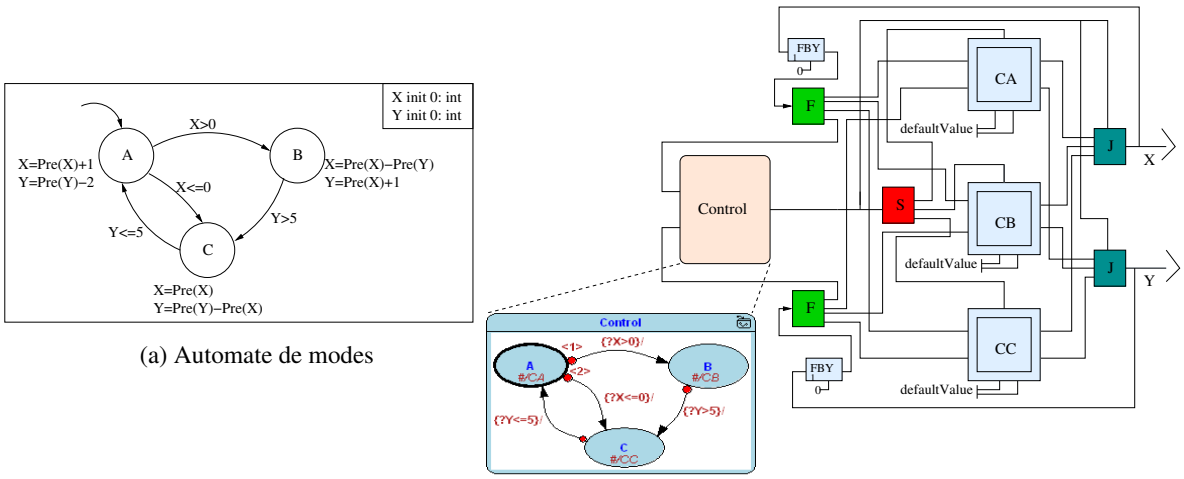
du système contenant des changements de modes. Dans ce cas, la description du système sera composée d'un ensemble de composants qui sont soit contrôlés ou non contrôlés. Ceci correspond à une séparation moins stricte, mais bien organisée, entre la partie de contrôle et les différentes parties de calcul tout en définissant un modèle plus général et plus simple à utiliser. Il est aussi à noter que ce modèle supporte la description hiérarchique sur tous ses niveaux de conception.

Comme nous l'avons mentionné, notre méthode de séparation contrôle/données est principalement basée sur le principe des automates de modes. Ainsi, il est possible de développer un modèle de conception basée sur notre méthodologie de séparation à partir d'un modèle décrit en automate de modes. L'idée de base consiste à représenter chaque mode de fonctionnement (les équations LUSTRE liées à un état de l'automate) par une partie de calcul qui sera pilotée par un automate de contrôle ayant une structure équivalente à celle de l'automate de modes en question. Ce processus peut être résumé comme suit :

1. Extraire les entrées et les sorties du système
2. Construire l'automate de contrôle dont la structure est équivalente à celle de l'auto-

mate de modes

3. Modéliser chaque mode de fonctionnement dans l'automate de modes par un bloc de calcul indépendant
4. Connecter la partie de contrôle (l'automate) aux différentes parties de calcul en utilisant les opérateurs `Fork`, `Join` et `Selector`
5. Ajouter si nécessaire des opérateurs spéciaux tels que les opérateurs temporels (`PRE`, `FBY`, etc.)



(a) Automate de modes

(b) Modèle de séparation contrôle/données

FIG. 5.13 – *Automate de modes et son équivalent dans notre modèle de séparation contrôle/données*

Cette démarche est proche de celle utilisée par l'outil MATOU[42] pour l'implémentation des automates de modes en équations booléennes [MRR00]. La figure 5.13 représente un exemple d'un automate de modes et son équivalent dans notre modèle de séparation contrôle/données. Dans cet exemple, les modes de fonctionnement A, B et C de l'automate de modes sont respectivement remplacés par les blocs de calculs CA, CB et CC. Le passage entre les différents modes est réalisé par l'automate de contrôle `Control` qui, en fonction des valeurs de X et Y et de l'état courant du système, peut choisir la partie de calcul à activer. Dans ce modèle, les équation `Lustre` de l'automate de modes sont remplacées par des composants de calcul indépendants, tandis que la structure de l'automate de modes est remplacée par un automate de contrôle responsable de l'activation des différents modes de fonctionnement.

5.3.2 Nouveau formalisme pour le modèle de séparation contrôle/données

Dans cette section, nous allons proposer un nouveau formalisme pour notre modèle de séparation contrôle/données basé sur une *structure d'onglets* (*tab structure*). Le but de ce formalisme est de fournir un modèle plus clair et plus facile à utiliser, notamment en ce qui concerne l'ajout, la suppression et la modification des différents modes du système. Pour ce faire, nous avons introduit le concept de composants ayant une *même interface* pour tous les modes de fonctionnement d'un système. Dans ce contexte, les différents modes du système,

[42] `http://www-verimag.imag.fr/~maraninx/MATOU`

contrôlés par un automate et définis dans un niveau de hiérarchie donné, doivent avoir le même nombre et les mêmes types d'entrées et de sorties. Une vue globale du modèle proposé est donnée par la figure 5.14. Dans ce modèle, chaque mode du système est représenté

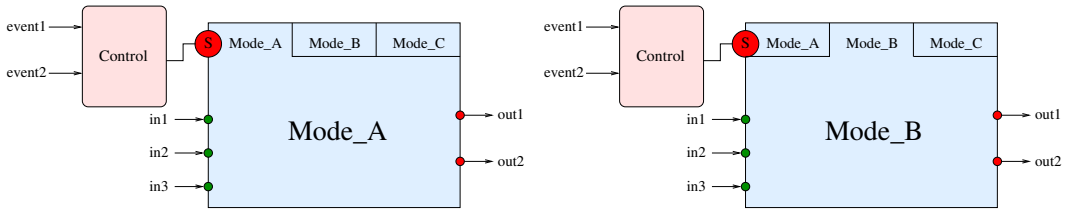
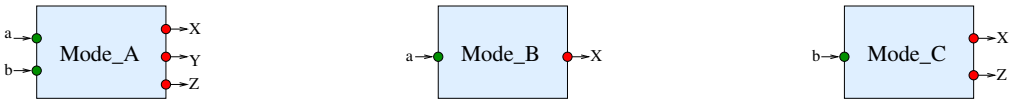


FIG. 5.14 – *Définition des différents modes de fonctionnement en utilisant la structure d'onglets*

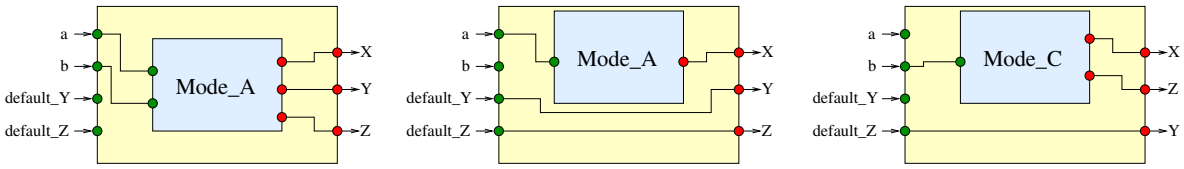
par un onglet. L'activation des modes est réalisée en fonction de la valeur donnée par la partie de contrôle via l'opérateur de sélection (`Selector`).

Cependant, il est rare que tous les modes d'un système aient la même interface. Le système de climatisation `Climate` de la figure 5.11 représente un exemple dans lequel les modes de fonctionnement du système ne donnent pas les mêmes sorties. Dans ce modèle, la question que nous nous posons est : *dans un cycle d'exécution donné, quelle valeur aura une sortie d'un mode non activé ?* La sémantique de l'opérateur conditionné dans SCADE permet de répondre à cette question en associant aux sorties des opérateurs non activés les valeurs définies par défaut pour ces sorties.

Dans notre modèle, nous répondons à cette question en imposant la définition d'une interface unique pour tous les modes de fonctionnement du système étudié. Ceci peut être réalisé en ajoutant un niveau de hiérarchie supplémentaire qui permet d'associer des valeurs par défaut aux sorties non calculées par le mode activé comme il est expliqué par la figure 5.15. Dans ce contexte, les sorties du composant hiérarchique englobant le mode de



(a) Différents modes de fonctionnement d'un système



(b) Ajout d'un niveau de hiérarchie pour avoir la même interface

FIG. 5.15 – *Définition des modes avec une même interface : introduction d'un niveau de hiérarchie*

fonctionnement représentent l'union des sorties de tous les modes de fonctionnement, tandis que l'ensemble de ses entrées est défini par l'union des entrées des différents modes et l'ensemble des valeurs par défaut pour les sorties qui peuvent être non définies. Dans l'exemple de la figure 5.15, les entrées/sorties du composant hiérarchique, introduit pour assurer la même interface aux différents modes du système, sont définis comme suit :

- $E_E = \{a, b\} \bigcup \{a\} \bigcup \{b\} \bigcup \{default\_Y, default\_Z\} \bigcup \{default\_Y\}$
  $= \{a, b, default\_Y, default\_Z\}$

(E_E : ensemble d'entrée, default_Y : la valeur par défaut de Y (la valeur de Y peut ne pas être définie dans les modes Mode_B et Mode_C), default_Z : la valeur par défaut de Z (la valeur de Z peut ne pas être définie dans le mode Mode_B))

– $E_S = \{x, y, z\} \bigcup \{x\} \bigcup \{x, z\}$
$= \{x, y, z\}$
(E_S : ensemble de sorties)

Nous remarquons également qu'il est possible que certaines entrées du composant hiérarchique ne soient pas consommées dans tous les modes de fonctionnement du système. Ceci ne pose aucun problème dans notre modèle de conception puisque nous considérons que le système est capable d'ignorer les valeurs en entrée dont il n'a pas besoin pour son processus comme il est le cas pour le langage LUSTRE. Il est également à noter que le processus permettant la définition d'une interface unique pour tous les modes de fonctionnement, en ajoutant des niveaux de hiérarchie supplémentaires, peut être complètement automatisé. Dans ce cas, l'utilisateur n'as pas à se soucier de son application, et peut définir les différents modes de fonctionnement de son système avec une interface différente.

L'application du formalisme de notre modèle de conception pour l'exemple de climatisation dans une voiture (`Climate`) est donnée par la figure 5.16. Dans cet exemple, la partie

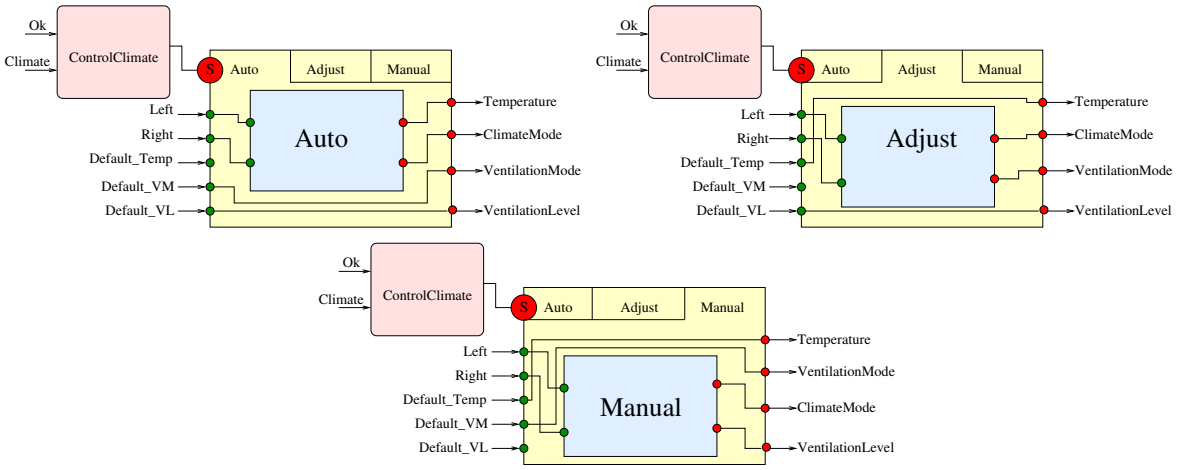


FIG. 5.16 – *Représentation du système `Climate` par un modèle à structure d'onglets*

de calcul contrôlée est représentée par trois onglets correspondant aux différents modes de fonctionnement du système. Pour assurer la même interface pour les différents modes, nous avons ajouté un niveau de hiérarchie supplémentaire pour fournir toutes les valeurs en sortie quelque soit le mode activé. Cette représentation permet la définition d'un modèle plus facile à maintenir et à réutiliser. Dans ce modèle les différents modes du système sont facilement distingués et peuvent être traités séparément. L'ajout et la suppression de modes est également facile grâce à la nature exclusive et indépendante des modes, et à l'hypothèse de l'interface unique pour tous les modes.

5.4 Spécification structurelle du modèle de séparation contrôle/-données

Comme nous l'avons discuté dans la section précédente, nous proposons un modèle de conception permettant de séparer de façon claire entre la partie contrôle et les différents modes de fonctionnement de l'application étudiée. En premier lieu, nous distinguons les composants non contrôlés des composants contrôlés de l'application, et nous appliquons sur ces derniers notre méthodologie de séparation contrôle/données. L'idée de base est de regrouper, pour chaque composant contrôlé, tout le contrôle présent à un certain niveau de hiérarchie dans un et un seul composant de contrôle, de définir les différents modes de fonctionnement sous forme de composants avec la même interface, et d'utiliser le composant de contrôle comme un chef d'orchestre pour piloter l'activation des différents modes de fonctionnement. Un composant relatif à un mode de fonctionnement peut être à son tour contrôlé, et sa structure sera décrite en respectant notre méthodologie de séparation contrôle/données.

La structure de notre modèle de séparation contrôle/données est basée sur la notion de composant et peut être formalisée en une grammaire principalement composée des règles suivantes :

1. `Component` $\longrightarrow$ `Controlled` | `Not-Controlled`
2. `Controlled` $\longrightarrow$ `Control Modes`
3. `Modes` $\longrightarrow$ `Empty` | `Mode Mode`$^{+}$
4. `Mode` $\longrightarrow$ `Component`
5. `Not-Controlled` $\longrightarrow$ `Component`$^{+}$

La première règle de la grammaire spécifie que la structure de notre modèle de séparation contrôle/données est basée sur la notion de composant (`Component`) où chaque composant peut être soit contrôlé (`Controlled`) ou non contrôlé (`Not-Controlled`). La deuxième règle de grammaire spécifie que chaque composant contrôlé contient forcément une partie de contrôle (`Control`) qui est suivie par une partie décrivant les modes de fonctionnement du composant (`Modes`). La partie de contrôle représente un automate sous forme d'une machine à états fini dont nous n'allons pas détailler la structure, tandis que la partie relative aux modes de fonctionnement peut être soit vide (`Empty`) ou composée d'au moins deux modes de fonctionnement (`Mode`) comme il est décrit par la troisième règle de la grammaire. Le cas où la partie `Modes` est vide permet la description de la structure d'un composant constitué uniquement d'une partie de contrôle comme dans le cas du mode de fonctionnement `Adjust` représenté par l'exemple de la figure 5.10 (page 89). Ainsi, chaque mode de fonctionnement est à son tour un composant qui peut être contrôlé ou pas permettant ainsi la description hiérarchique de notre modèle de conception comme il est décrit par la quatrième règle de la grammaire. La cinquième règle de la grammaire spécifie que tout composant non contrôlé peut être vu comme un graphe dirigé acyclique des différents composants de notre modèle de conception.

La figure 5.17 représente des exemples de modèles qui sont acceptés ou pas dans notre modèle de séparation contrôle/données selon la grammaire spécifiée. Le premier exemple, montre un cas simple dans lequel l'application est décrite uniquement par un composant non contrôlé et ne contient aucun comportement de contrôle. À l'inverse, le deuxième exemple représente le cas d'une application qui est uniquement décrite par un comportement de contrôle. La représentation de ces deux types d'application est possible dans notre

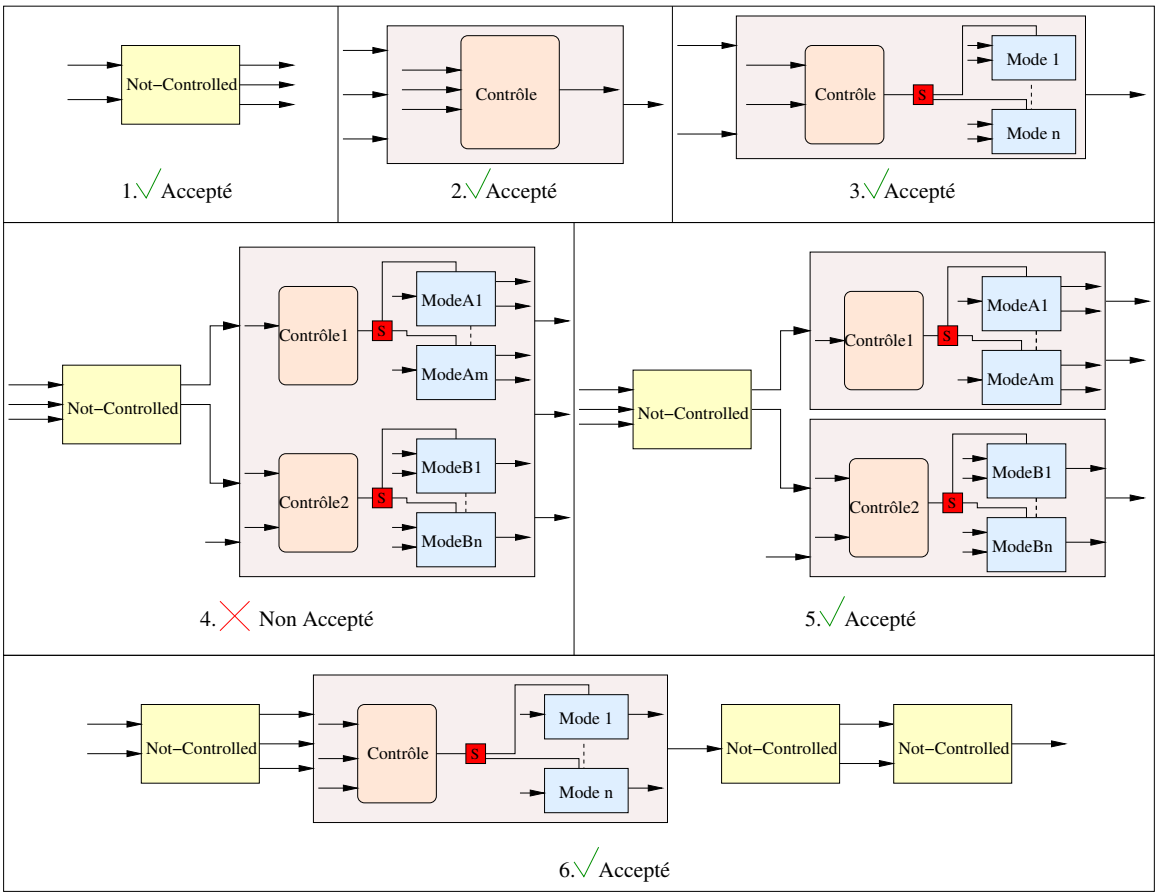


FIG. 5.17 – *Exemples de modèles acceptés ou pas dans notre modèle de séparation contrôle/données*

modèle de séparation contrôle/données et respecte bien les règles de la grammaire décrite ci-dessus. Le troisième exemple représente une application composée d'un seul composant contrôlé. La structure de ce dernier est définie par un composant de contrôle qui pilote l'activation d'un ensemble de modes de fonctionnement. Cette description suit les règles de notre grammaire et elle est bien acceptée par le modèle de séparation contrôle/donnée. Le quatrième exemple montre un cas qui ne respecte pas notre modèle de séparation. La raison est que dans le même niveau de hiérarchie sont définis plusieurs composants de contrôle qui pilotent différents modes de fonctionnement. Nous rappelons que le principe de notre méthodologie de séparation est de regrouper tout le contrôle présent dans un niveau de hiérarchie dans un et un seul composant de contrôle qui va piloter l'activation des différents modes de fonctionnement, et cette contrainte n'est pas respectée par le quatrième exemple. Le cinquième exemple représente une application composée d'un composant non contrôlé suivi par deux composants contrôlés pouvant s'exécuter en parallèle. La description des composants contrôlés respecte bien le modèle de séparation contrôle/données dans le sens où le contrôle présent à chaque niveau de hiérarchie est regroupé dans un seul composant de contrôle. Le sixième exemple représente un cas plus général d'une application composée d'un enchaînement de composants non contrôlés ou contrôlés tout en respectant la structure de notre modèle de séparation contrôle/données.

Ainsi, pour des raisons de simplification, nous considérons que la structure de la partie

contrôle correspond à un automate de mode plat. Ce dernier permet de déterminer l'état de l'automate en fonction des événements de contrôle et sans aucune construction parallèle ou hiérarchique.

5.5 Vers l'utilisation des automates hiérarchiques et parallèles

Dans cette section nous allons présenter nos premiers résultats concernant l'utilisation des automates parallèles et/ou hiérarchique dans notre modèle de séparation contrôle/-données. Pour ce faire, nous proposons de définir informellement le sens d'utilisation de ces automates, ainsi que leur équivalent dans le modèle de séparation de base à automates plats.

5.5.1 Utilisation des automates hiérarchiques

Comme nous l'avons décrit dans la section précédente, notre modèle de conception permet des constructions hiérarchiques où certains modes de fonctionnement d'un composant contrôlé sont également contrôlés comme il est montré par l'exemple de la figure 5.18. Dans

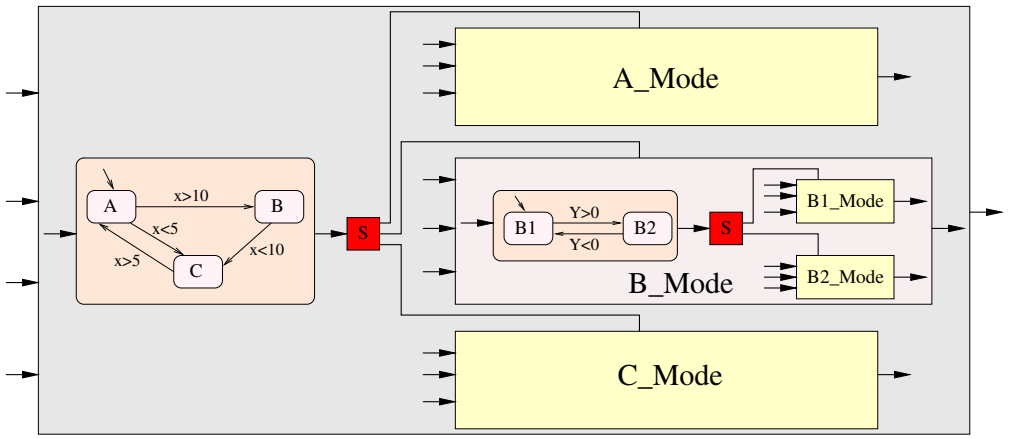


FIG. 5.18 – *Exemple d'un modèle d'application à plusieurs niveaux de hiérarchie*

cet exemple, l'application décrite peut fonctionner dans trois modes différents dont l'activation est pilotée par un automate de contrôle. Les deux modes `A_Mode` et `C_Mode` sont des composants non contrôlés, tandis que le mode `B_Mode` représente un composant contrôlé qui est, à son tour, composé d'une partie de contrôle et de deux modes de fonctionnement `B1_Mode` et `B2_Mode`.

La représentation hiérarchique permet une structuration simple et séparée sur différents niveaux dans la description de l'application étudiée. Elle facilite également la manipulation et la modification des modes de fonctionnement, et permet une bonne réutilisation des composants déjà existants. Cependant, pour certaines applications, il est possible d'avoir besoin de regrouper tout le contrôle présent sur plusieurs niveaux de hiérarchie dans un seul automate de contrôle hiérarchique. Ceci peut être utile dans l'application des processus de test et de vérification formelle sur toute la structure de l'automate hiérarchique. Pour cette raison, nous introduisons l'utilisation des automates hiérarchiques dans notre modèle de conception. La construction de tels automates à partir d'un modèle de conception hiérarchique à automates plats peut être réalisée comme suit :

1. Pour chaque composant contrôlé relatif à un niveau de hiérarchie de profondeur n, l'automate de contrôle de ce composant sera défini à l'intérieur de l'état correspondant à son activation au niveau de hiérarchie supérieur (de profondeur $n-1$)
2. Les différents modes de fonctionnement définis au niveau de hiérarchie de profondeur n seront introduits comme étant des modes de fonctionnement supplémentaire au niveau de hiérarchie supérieur (de profondeur $n-1$)

L'application de ce processus, qui peut être complètement automatisé, pour l'exemple de la figure 5.18 fournit le modèle représenté par la figure 5.19. Dans ce modèle, nous avons

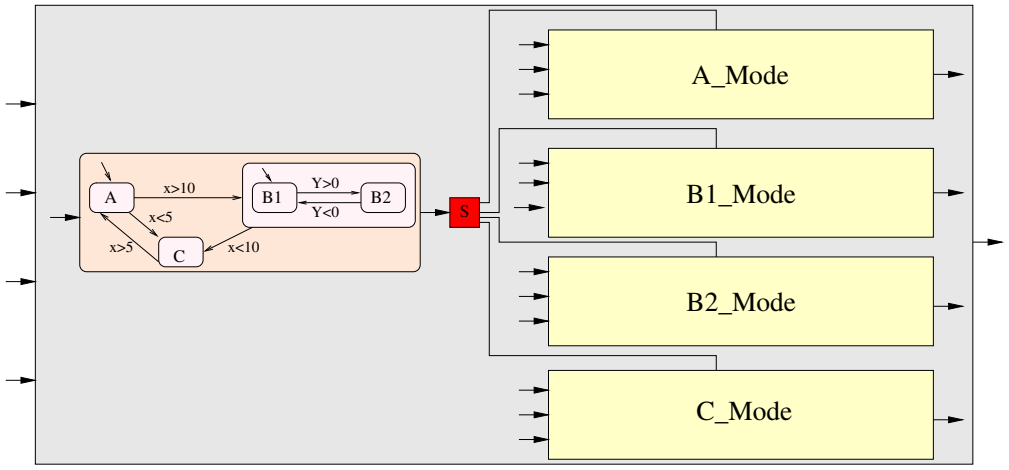


FIG. 5.19 – *Exemple d'utilisation d'un automate hiérarchique pour la description du modèle de la figure 5.18*

défini l'automate de contrôle relatif au mode de fonctionnement `B_Mode` à l'intérieur de l'état `B` de l'automate de contrôle responsable de l'activation de ce mode et défini au niveau supérieur de la hiérarchie. Ainsi, les modes de fonctionnement `B1_Mode` et `B2_Mode` sont décrits comme étant des modes alternatifs aux modes de fonctionnement `A_Mode` et `C_Mode` dans le même niveau de la hiérarchie puisque cette dernière est représentée au niveau de l'automate de contrôle.

Le comportement des deux modèles, représentés par les figures 5.18 et 5.19, est équivalent. Il est également à noter que le passage entre ces deux modèles est possible dans les deux sens et peut être complètement automatisé, ce qui donne plus de liberté aux utilisateurs dans la description de la structure de l'automate de contrôle, et définit une sémantique à l'utilisation des automates de contrôle hiérarchiques dans notre modèle de séparation contrôle/données.

5.5.2 Utilisation des automates parallèles

Selon les règles de la grammaire décrivant la structure de notre modèle de séparation contrôle/données, il est possible de représenter des exécutions parallèles pour les composants non liés par des dépendances de données comme il est montré par l'exemple de la figure 5.20. Dans cet exemple, les deux composants `Controlled_A` et `Controlled_B` sont de type contrôlé et leur structure est composée d'une partie de contrôle suivie par les modes de fonctionnement relatifs à chaque composant. Cet exemple montre également que ces deux composants peuvent s'exécuter en parallèle puisqu'ils ne sont pas liés par des dépendances

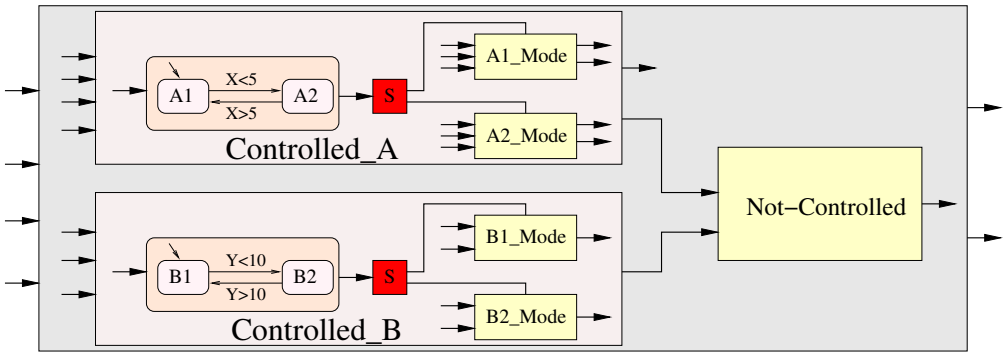


FIG. 5.20 – *Exemple d'un modèle d'application avec des composants parallèles*

de données. Dans ce cas, et à chaque instant, il est possible d'exécuter en parallèle un des modes du composant `Controlled_A` avec un autre mode du composant `Controlled_B`. Ceci est relatif à l'application du produit cartésien qui donne l'ensemble des composants pouvant s'exécuter en parallèle. Pour l'exemple de la figure 5.20, l'application du produit cartésien donne les couples de composants suivant : (`A1_Mode`, `B1_Mode`), (`A1_Mode`, `B2_Mode`), (`A2_Mode`, `B1_Mode`) et (`A2_Mode`, `B2_Mode`).

De façon similaire au modèle hiérarchique, la représentation parallèle facilite la manipulation et la réutilisation des différents composants de l'application. Nous remarquons également que la sémantique de la composition parallèle est similaire à celle des automates parallèles. Il est donc possible de regrouper le contrôle présent dans les composants parallèles dans un seul automate de contrôle parallèle. Le modèle résultant doit respecter la structure de notre modèle de séparation dans le sens où il doit être composé d'une seule partie de contrôle qui fournit une seule information sur le mode à activer. Pour ce faire, nous considérons que si l'automate parallèle regroupe le contrôle présent dans n composants parallèles, cet automate doit fournir en sortie un *n-uplet* permettant de sélectionner le composant contenant l'ensemble des modes à activer en parallèle. Le résultat fourni par l'automate peut être décrit sous la forme $(m_1, m_2, ..., m_n)$ où m_i décrit le mode de fonctionnement à activer pour le composant i. La construction de l'automate parallèle à partir d'un modèle de conception parallèle à automates plats peut être réalisée comme suit :

1. Regrouper de façon parallèle les automates de contrôle présents dans les différents composants parallèles dans un seul automate hiérarchique
2. Définir les différents modes de fonctionnement par des composants contenant les modes pouvant s'exécuter en parallèle selon l'application du produit cartésien

L'application de ce processus pour l'exemple de la figure 5.20 donne le modèle de la figure 5.21. Dans ce modèle l'automate de contrôle est relatif à la composition parallèle des deux automates de contrôle des composants `Controlled_A` et `Controlled_B`, tandis que les modes de fonctionnement sont relatif à l'application du produit cartésien où chaque mode représente une composition parallèle d'un des modes du composant `Controlled_A` et d'un autre mode du composant `Controlled_B`. Il est clair que l'application de ce processus donne un modèle plus complexe et redondant dans le cas des systèmes plus complexes contenant plusieurs composants contrôlés et parallèles.

Les comportements des deux modèles représentés par les figures 5.20 et 5.21 sont équivalents, et le processus permettant le passage entre ces deux modèle peut être complètement

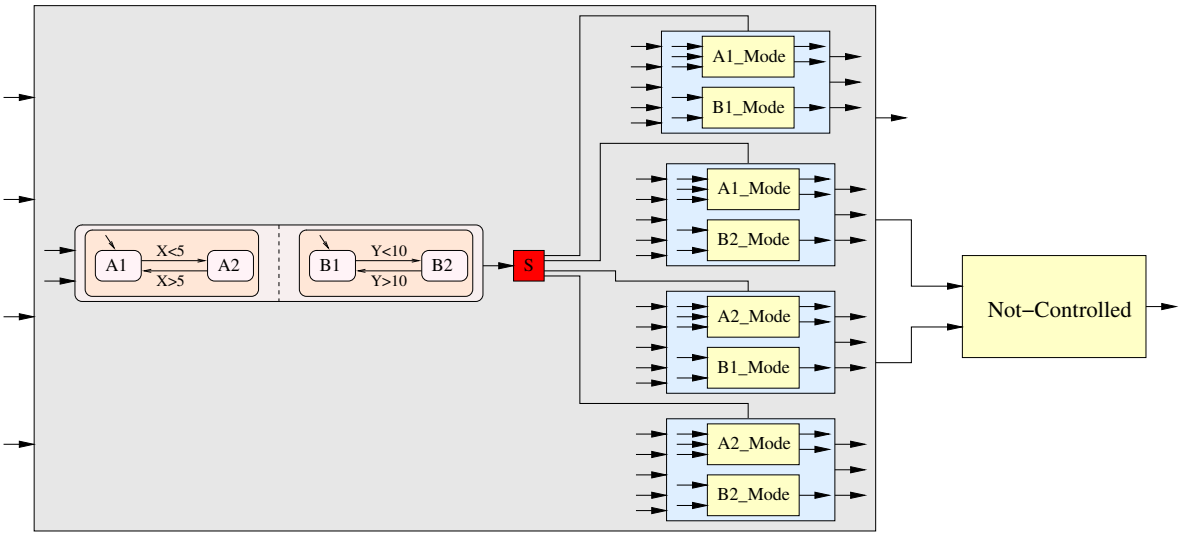


FIG. 5.21 – *Exemple d'utilisation d'un automate parallèle pour la représentation du comportement du modèle de la figure 5.20*

automatisé. Ainsi, la construction de l'automate de contrôle parallèle peut être intéressante pour l'application des processus de vérification formelle, et donne plus de liberté aux utilisateurs pour la description de la structure de leur automates.

5.6 Intérêts de la méthodologie de séparation contrôle/données

Notre méthodologie de séparation contrôle/données permet d'avoir un modèle de conception plus clair et plus facile à maintenir. Dans ce modèle, la nature indépendante et exclusive des différentes parties du système permet leur étude séparée en utilisant les outils les plus appropriés pour chacune d'entre elles. Ainsi, la spécification modulaire du système permet de bénéficier des avantages de la programmation modulaire, notamment en ce qui concerne le temps de développement et la réutilisation des différents composants déjà existants. De plus, le formalisme proposé pour notre modèle, basé sur une structure d'onglets et imposant la même interface pour les modes du système étudié, facilite la modification, l'ajout ou la suppression de modes.

Le principal avantage de notre méthodologie de séparation est de permettre la simulation et la vérification séparées des différentes parties du système. Sachant que le rôle principal de la partie contrôle consiste à piloter les différents modes de fonctionnement du système qui sont par nature indépendants et exclusifs, l'application séparée des processus de simulation et de vérification peut assurer la vérification du comportement global du système étudié. En d'autres termes, la nature exclusive de la structure de l'automate de contrôle qui associe à chaque état un seul mode de fonctionnement fait que les différents modes sont exclusifs et peuvent donc être étudiés séparément. Ceci peut être généralement fait en remplaçant la partie contrôlée par le mode de fonctionnement à vouloir vérifier. La vérification séparée des différentes parties du système donne un gain considérable en terme de temps de vérification et de capacité mémoire. Dans ce contexte, le nombre d'états d'un module indépendant du système est largement plus petit que celui du système global.

Il est à noter que le développement séparé est devenu de plus en plus une contrainte industrielle inhérente aux systèmes embarqués critiques. La démonstration de la correction de ces systèmes doit également être modulaire dans le sens où, à partir des vérifications locales effectuées séparément sur chaque composant du système, il est possible de vérifier les propriétés globales recherchées sur le système en raisonnant pas à pas sur ses différentes parties. Cette approche qui consiste à profiter de la modularité du système de manière à vérifier les propriétés globales a été prouvée par N. Halbwachs *et al.* [HLR93]. Leur idée repose sur le constat que toute propriété satisfaite par une partie du système P_{i1} permet de restreindre le champ des comportement pour la partie P_{i2} ($i1 \neq i2$).

Dans notre étude, nous nous intéressons en particulier au processus de vérification séparée des différentes parties du système. Ce processus permet également de localiser plus facilement les erreurs pour pouvoir les corriger tout en évitant la modification de l'application entière qui peut être très coûteuse en terme de temps et de ressources. Ainsi, pour le processus de vérification, nous faisons appel à la technique de vérification formelle de type « *model checker* ».

Un model checker est un algorithme qui permet de vérifier si une machine M satisfait (ou est un modèle d') une propriété φ. La machine M fait partie d'une certaine classe de modèles mathématiques sous forme de machines à états finis tels que la structure de Kripke [Gup93], tandis que la propriété φ est exprimée dans une certaine logique telle que la logique temporelle CTL [BAMP83]. Le processus du model checker consiste à confronter un automate, appelé *modèle*, à une formule de logique temporelle. L'automate représente les futurs comportements du système à développer, tandis que la formule permet d'exprimer la propriété que l'utilisateur veut vérifier pour son système. L'algorithme du model checker représente l'application d'une procédure de recherche qui permet de déterminer si les propriétés sont satisfaites par le modèle du système.

Le premier algorithme du model checker a été développé par Clarke et Emerson en 1981 [CE81]. Cet algorithme était polynomial aussi bien dans la taille du modèle étudié que dans la longueur des spécifications exprimées en logique temporelle. Plusieurs améliorations de cet algorithme ont été proposées dans la littérature, et le monde de la recherche dans le domaine du model checker n'a pas été stationnaire [QS82, CES86]. Il existe également différents types de model checker en fonction de la sémantique des modèles étudiés, la logique utilisée pour l'expression des propriétés, et leur algorithme d'implémentation [BCM$^+$90, SL03].

L'avantage du model checker par rapport aux techniques de preuves est, d'une part qu'il est complètement automatique : une fois que le programme est modélisé, aucune intervention humaine n'est nécessaire, et d'autre part, en cas de réponse négative (si le modèle ne satisfait pas la propriété), il retourne un contre exemple : une exécution précise du modèle qui ne satisfait pas la propriété. Ce contre exemple est particulièrement utile pour pouvoir situer la source de l'erreur dans un modèle souvent complexe.

Pour montrer l'avantage de notre méthodologie de séparation en terme de vérification formelle, nous avons comparé les temps de vérification pour les deux modèle du système de climatisation `Climate`. Le premier modèle est celui proposé par Esterel Technologies et présenté dans la section 5.2.2, noté M1, tandis que le deuxième est notre modèle de séparation contrôle/données présenté dans la section 5.3.1, noté M2. Nous proposons de vérifier pour ces deux modèles les deux propriétés suivantes :

- $\varphi 1$: à chaque instant, le système est dans un et un seul mode à la fois
- $\varphi 2$: dans le mode `Manual`, l'appui sur le bouton `Right` permet d'incrémenter le niveau

de ventilation par 1 sans dépasser 100

Comme nous l'avons montré ci-dessus, le modèle M1 contient un mélange de traitement de données et du contrôle. Dans ce modèle, la vérification d'une propriété donnée pour un certain mode nécessite la vérification du système complet puisque la distinction des différents modes est impossible. Contrairement à ce modèle, le modèle M2 permet une séparation claire entre les différents modes du système. Dans ce contexte, si nous souhaitons vérifier une propriété pour un certain mode, il suffit de vérifier la partie du système ou le mode concerné. Il est également à noter que les formules des propriétés vérifiées pour le modèle de séparation sont beaucoup moins complexes que celles utilisées pour vérifier le système complet. Par exemple, la vérification de la propriété $\varphi 2$ pour le modèle M2 ne nécessite pas de vérifier que le mode concerné est `Manual` puisque cette propriété est vérifiée à l'intérieur de ce mode. Le tableau 5.1 donne le résultat des temps de vérification obtenus pour les deux propriétés $\varphi 1$ et $\varphi 2$ pour les deux modèles M1 et M2. Ces résultats sont obtenus par l'outil de

Propriété	***Module vérifié pour*** M1	***Module vérifié pour*** M2	***Temps*** M1	***Temps*** M2	***Accélération***[a]
$\varphi 1$	Tout le système	SSM `ControlMode`	0.36*s*	0.03*s*	**12.00**
$\varphi 2$	Tout le système	Mode `Manual`	1.69*s*	0.09*s*	**18.77**

[a] L'accélération est calculée comme suit : $\frac{t_{M1}}{t_{M2}}$, où t_{M1} t_{M2} représentent respectivement les temps de vérification pour les modèles M1 et M2.

TAB. 5.1 – *Temps de vérification des propriétés* $\varphi 1$ *et* $\varphi 2$ *pour les modèles* M1 *et* M2

vérification formelle de SCADE. Cet outil est utilisé pour la vérification des propriétés fonctionnelles des systèmes complexes. C'est un model checker, appelé *Design Verifier*[43], basé sur un moteur de preuve puissant développé par Prover Technology[44]. Dans ce contexte, la propriété à vérifier est généralement décrite dans un nœud séparé qui sera lié au système sans modifier son comportement, et agit comme un *observateur* pour ce système. Ainsi, les résultats donnés par le tableau 5.1 représentent un exemple simple pour montrer les avantages de notre technique de séparation contrôle/données. Nous présentons dans le chapitre 6 nos résultats de vérification pour un système réel plus complexe.

5.7 Synthèse et conclusion

Dans ce chapitre, nous avons présenté notre méthodologie de conception pour les systèmes réactifs synchrones et hybrides. Cette méthodologie est basée sur une séparation claire entre la partie de contrôle et les différentes parties de calcul. Elle est principalement basée sur le concept des automates de modes, et permet, d'une part, d'avoir un modèle plus facile à étudier, et d'autre part, de tirer profit des différents outils déjà existant dédiés exclusivement au traitement de la partie contrôle ou de la partie calcul.

[43] `http://www.esterel-technologies.com/products/scade-drive/design-verifier.html`
[44] `http://www.prover.com`

Nous avons présenté notre démarche en nous basant sur un exemple réel d'un système de climatisation dans une voiture, et en utilisant l'outil SCADE comme environnement de développement. Cet environnement n'impose aucune méthodologie de séparation entre le contrôle et les calculs, et mène souvent à un mélange de leur spécification qui peut rendre difficile la compréhension de l'application, ainsi que l'étude et la réutilisation des différentes parties du système. Il est donc intéressant de proposer une méthodologie de conception basée sur une séparation claire entre la partie de contrôle et les différentes partie de calcul.

En premier lieu, nous avons étudié la possibilité de séparation entre la partie contrôle et les parties de traitement de données dans SCADE. Cette étude nous a montré qu'il est difficile d'avoir une séparation stricte contrôle/données dans SCADE puisque, dans cet outil, chaque variable en sortie doit avoir une seule définition à un instant donné, et il est donc impossible de partager la même variable par plusieurs opérateurs différents. Pour faire face à ce problème, nous avons proposé l'ajout des opérateurs spéciaux jouant le rôle de `Fork`, `Join` et `Selector`, et permettant d'avoir un modèle de séparation plus clair.

Nous avons également montré que notre approche est similaire à celle des automates de modes qui consiste à séparer la spécification globale du système en plusieurs modes de fonctionnement contrôlables par un automate de contrôle. Cette séparation stricte peut être intéressante pour certains systèmes dont la distinction des différents modes est évidente. Cependant, il existe des systèmes où le comportement de contrôle est peu présent et concerne uniquement des sous-parties de ces systèmes. Dans ce cas, la distinction des modes de fonctionnement pour le système global est difficile et peut induire des erreurs. Pour résoudre ce problème, nous avons proposé la possibilité de généraliser notre modèle de séparation contrôle/données en permettant l'utilisation de ce concept localement dans les sous-parties du système contenant une description de contrôle.

Par la suite, et pour améliorer la représentation de notre modèle de séparation, nous avons proposé un formalisme basé sur l'utilisation d'une structure d'onglets. Ce formalisme impose l'utilisation d'une interface unique pour les différents modes du système pour faciliter la modification, l'ajout ou la suppression des modes. Nous avons montré que cette hypothèse d'interface unique peut être facilement respectée en utilisant des niveaux de hiérarchie supplémentaires pour « habiller » les modes qui n'ont pas la même interface par une structure à interface unique.

À la fin de ce chapitre, nous avons discuté les avantages de notre méthodologie de séparation en terme de lisibilité et de facilité de maintenance et de réutilisation. Nous avons par la suite consacré notre étude au processus de vérification formelle pour montrer le gain en temps de vérification pour notre modèle de séparation contrôle/données en utilisant la technique du model checker pour l'exemple de système de climatisation dans une voiture.

Nous rappelons que le but principal de notre étude consiste à introduire la notion de contrôle dans les applications de traitement de données massivement parallèle, et en particulier dans le modèle de spécification ARRAY-OL. L'introduction de contrôle dans ARRAY-OL sera basée sur notre méthodologie de séparation contrôle/données. Ceci nous permettra, d'une part, de tirer profit des avantages de cette technique de séparation, et d'autre part, d'introduire les notions de contrôle dans le modèle ARRAY-OL sans avoir besoin de modifier la structure ou les concepts de base de ce modèle. L'introduction du contrôle dans le modèle de spécification ARRAY-OL est présentée dans le chapitre 7.

Chapitre 6

Étude de cas : limiteur et régulateur de vitesse intelligent avec GPS

Dans ce chapitre nous allons présenter une étude de cas d'un limiteur et régulateur de vitesse intelligent avec GPS. L'objectif de cette étude est d'illustrer l'utilisation et les avantages de notre méthodologie de séparation contrôle/données présentée dans le chapitre 5.

6.1 Introduction

Selon plusieurs statistiques et d'après les experts du domaine de la sécurité routière, les principales causes des accidents de la route sont l'excès de vitesse et le non respect des vitesses limites. Par exemple, et selon la sécurité routière française[45], chaque année en France, le nombre d'accidents de la route dépasse les 120000 accidents, faisant plus de 7500 morts (7720 en 2001). Plus de 40% de ces accidents sont dus aux excès de vitesse, et plus de 60% des conducteurs ne respectent pas les vitesses limites. Ceci montre d'une façon flagrante l'implication de la vitesse et du non respect des limitations de vitesse dans l'augmentation du nombre de morts et de blessés sur les routes.

L'excès de vitesse est généralement dû au manques de responsabilité des conducteurs, et d'informations précises et suffisantes sur les limitations de vitesse sur les routes. Pour ces raisons, et pour donner aux utilisateurs la possibilité de contrôler la vitesse de leur voitures, plusieurs constructeurs ont développé différents systèmes tel que le *limiteur* ou *régulateur de vitesse* déjà existant dans certains véhicules. Le but de ce système consiste à refléter la vitesse normale à laquelle le conducteur souhaite voyager. Ce régulateur de vitesse, souvent connu sous le nom de *Cruise Control*, est un dispositif qui permet à un véhicule de garder une vitesse constante. Facteur de sécurité accrue pour les uns, il est accusé par d'autres d'être à l'origine d'accidents graves. Dans ce domaine, la recherche est toujours en cours de développement pour proposer des systèmes plus efficaces et évolués, et récemment, nous parlons souvent des systèmes de régulation de vitesse avec GPS (Global Positioning System). L'objectif de ces systèmes est d'offrir un moyen automatique permettant d'adapter la vitesse de la voiture à celle autorisée dans sa zone de localisation.

Dans ce domaine, plusieurs pays européens ont lancé différents projets de recherches et d'expériences sur les systèmes de régulation de vitesse. Ces systèmes sont généralement connus sous le nom d'ISA pour *Intelligent Speed Adaptation*, ou sous le nom d'EVSC pour *External Vehicle Speed Control*. Les résultats obtenus par ces projets sont généralement diversifiés. Leur comparaison reste néanmoins difficile puisque chaque expérience est basée sur des protocoles particuliers et vise des objectifs bien spécifiques. Ainsi, les technologies appliquées et la nature des systèmes varient d'un pays à un autre. Cependant, certaines conclusions communes indiquent le bénéfice d'utilisation de tels systèmes pour réduire le nombre d'accidents et leur dangerosité.

Parmi ces projets, nous citons le projet français LAVIA[46] lancé en septembre 2001[MES03]. Le système LAVIA est un régulateur de vitesse intelligent qui permet d'adapter automatiquement la vitesse d'une voiture à celle autorisée dans sa zone de localisation déterminée par un dispositif de navigation de type GPS. Pendant la période $1999 - 2002$, le ministère de la sécurité routière suédois s'est retrouvé en tête d'un projet de type ISA[47] dont l'objectif est d'étudier de plus près le comportement des conducteurs, leur interaction avec le système, ainsi que l'impacte de ces systèmes sur la sécurité routière [BV02]. Un autre

[45] `http://www.securite-routiere.org`
[46] `http://www.lavia.fr`
[47] `http://www.vv.se/isa`

projet Européen, dénommé PROSPER[48], a été lancé en 2002 [Bey04]. L'objectif principal de ce projet a été d'évaluer les bénéfices en coût et l'efficacité des méthodes ISA par rapport aux méthodes traditionnelles en effectuant une analyse complète des différentes stratégies de mise en œuvre convenables et possibles pour la gestion et le contrôle des vitesses. Entre octobre 2002 et décembre 2003, un projet de type ISA a été lancé dans la ville de Gand en Belgique [Pag04]. Ce projet a progressé en parallèle avec celui du PROSPER, mais il ne représente pas une partie de ce dernier malgré leurs objectifs communs. Un autre projet similaire a été développé en Danemark. C'est le projet INFATI[49], un acronyme danois pour *INtelligent FArtTIlpasning* qui signifie « adaptateur de vitesse intelligent » [JLPR04]. Ce projet est principalement lié à l'étude des techniques et des moyens pour la réalisation des systèmes de régulation de vitesse, leur spécification, le développement des prototypes et les processus de test.

Le caractère critique des systèmes de régulation de vitesse nécessite l'utilisation d'outils et de méthodes fiables et bien efficaces permettant d'assurer leur sûreté de fonctionnement. Tout plantage ou défaillance dans ces systèmes peut conduire à des incidents généralement catastrophiques. La modélisation et la vérification de ces systèmes est donc une activité importante et difficile.

Une des caractéristiques principales des systèmes de régulation de vitesse est qu'ils combinent les traitements de données et le contrôle. Dans ce qui suit, nous allons présenter l'étude d'un système de limiteur et régulateur de vitesse intelligent avec GPS dénommé ICCG (Intelligent Cruise Control with GPS) [LRD06]. Le but de cette étude est d'illustrer les avantages de notre méthodologie de séparation contrôle/donnée, présentée dans le chapitre 5, notamment en ce qui concerne le développement modulaire et la vérification formelle. Il est à noter que cette étude de cas représente une simplification des systèmes réels dans le sens où nous faisons abstraction des détails techniques. Ces derniers ont été largement étudiés par la communauté scientifique dans le domaine du transport et ne font pas partie de notre sujet de travail.

6.2 Description de l'application

Le système étudié décrit un limiteur et régulateur de vitesse intelligent conçu pour régulariser la vitesse d'un véhicule à un point donné sélectionné par le conducteur et/ou donné par le GPS. L'application ICCG représente un système de type ISA dont le rôle principal est de contrôler automatiquement la vitesse d'un véhicule pour assurer la sécurité des passagers.

Le système ICCG peut être vu comme une aide électronique permettant de faciliter la conduite d'un véhicule en contrôlant sa vitesse. Il informe le conducteur des différents changements dans les limitations de vitesse et, dans certains cas, l'oblige à les respecter. L'étude d'un tel système s'avère importante puisqu'il permet d'augmenter de façon considérable la sécurité des automobilistes.

Dans la description du système ICCG, nous considérons que ce dernier est mis en service lorsque le conducteur actionne le bouton de régulation `On_ICCG` qui génère un signal de déclenchement pour ce système. Ceci est permis uniquement pour les véhicules à une

[48] `http://www.prosper-eu.nl`
[49] `http://www.infati.dk`

vitesse de $50km/h$ ou plus. La terminaison du système peut être achevée de quatre façons différentes :

- implicitement, en pressant sur la pédale de freinage
- explicitement, en actionnant le bouton de régulation `Off_ICCG`
- en arrêtant le moteur (via un *starter*)
- ou lorsque la vitesse du véhicule est inférieur à $50km/h$

Nous considérons également que lorsque les actions d'accélération et de ralentissement apparaissent simultanément, la priorité est donnée à l'action de ralentissement. Dans ce cas, la fonction de régulation est complètement désactivée.

Dans ce qui suit, nous allons décrire le fonctionnement du système ICCG une fois déclenché. De façon générale, le système de régulation de vitesse que nous proposons d'étudier peut opérer dans cinq modes de fonctionnement différents : `Alarm` (alerte de survitesse), `Limit` (limiteur de vitesse), `Cruise` (régulateur de vitesse), `LimitGPS` (limiteur de vitesse avec GPS) et `CruiseGPS` (régulateur de vitesse avec GPS). Initialement, le système est en mode `Alarm_Mode`. L'interaction avec le système, ainsi que l'activation et le passage entre les différents modes se font en fonction du choix de l'utilisateur et via les boutons :

- `On` : active le système en mode limiteur et fixe la vitesse limite à la vitesse courante du véhicule
- `Off` : arrête le système et retourne en mode d'alerte de survitesse
- `Set` : fixe la vitesse limite à la vitesse courante du véhicule
- `Resume` : réactive le système après une interruption (appui sur la pédale de frein par exemple)
- `QuickAccel` (+) : augmente la vitesse limite fixée par le conducteur d'une constante SPEEDINC
- `QuickDecel` (-) : diminue la vitesse limite fixée par le conducteur d'une constante SPEEDINC
- `GPS` : active ou désactive le GPS
- `Cruise` : fait passer le système en mode régulateur ou retourne en mode limiteur

Une description plus détaillée des conditions de passage entre les différents modes de fonctionnement est donnée par l'automate de la figure 6.1.

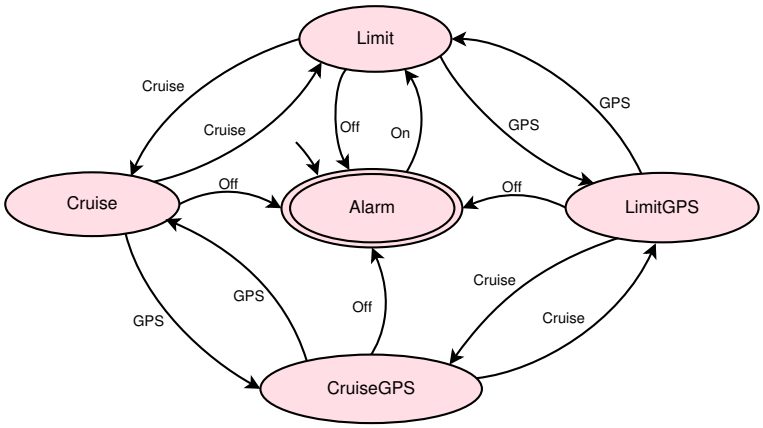


FIG. 6.1 – *Description des différents modes de fonctionnement de l'application ICCG*

Pour faciliter l'interaction avec le système, le système ICCG est équipé par un écran LCD permettant l'affichage des informations sur le mode de fonctionnement activé, sur la vitesse

limite à ne pas dépasser et sur la vitesse actuelle de la voiture comme il est montré par l'exemple de la figure 6.2. Dans ce qui suit, nous allons décrire plus en détails le comporte-

FIG. 6.2 – *Représentation du système ICCG*

ment des différents modes de fonctionnement de l'application ICCG.

6.2.1 Le mode `Alarm`

Le mode `Alarm` (Alerte de survitesse) est un mode purement informatif. Son rôle consiste uniquement à indiquer au conducteur, par un signal sonore ou lumineux, s'il a dépassé la vitesse limite autorisée. Cette dernière est donnée par le GPS et représente la vitesse limite autorisée dans la zone actuelle du véhicule.

L'alerte de survitesse n'influe en aucun cas sur la vitesse réelle du véhicule. Le conducteur reste toujours le maître de sa conduite et le signal sonore ou lumineux ne représente qu'un simple avertisseur. Dans ce mode, les pédales agissent normalement sur le comportement du véhicule et n'ont aucune influence sur l'état du système. Cependant, en cas de perte du signal GPS, la fonctionnalité du mode `Alarm` est interrompu, un signal sonore ou lumineux est envoyé pour indiquer la perte des signaux satellite et le système passe en état `GPS_Fail`. Cette description montre que le mode `Alarm` peut fonctionner en deux sous-modes différents : `Alarm_GPS` qui représente le comportement normal du système en présence du signal GPS, et `Alarm_GPS_Fail` qui décrit le comportement du système dans le cas d'une perte du signal GPS. Les conditions d'activation et le passage entre ces deux sous-modes sont donnés par l'automate de la figure 6.3.

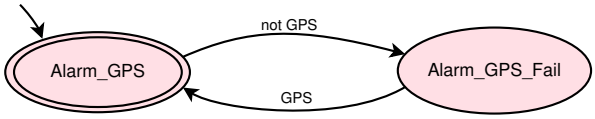


FIG. 6.3 – *Les différents sous-modes de fonctionnement du mode* `Alarm`

6.2.2 Le mode `Limit`

Le mode `Limit` (Limiteur de vitesse) est le mode le plus utilisé dans les voitures actuellement commercialisées. Dans ce mode, le GPS est désactivé et le système joue le rôle

d'un limiteur de vitesse qui empêche le dépassement d'une limite préalablement fixée par le conducteur. Ce dernier peut toujours contrôler sa voiture en accélérant ou en freinant, mais il ne peut en aucun cas dépasser la vitesse limitée.

Le comportement de ce mode est proche de celui du mode `Alarm`. La seule différence est que la voiture ne peut pas dépasser la vitesse limite puisque la pédale d'accélération devient systématiquement inactive lorsque le conducteur atteint cette limite. Cependant, en cas de nécessité (dépassement d'une voiture par exemple), le conducteur peut dépasser la limite fixée (dans le domaine du transport ce phénomène est connu sous le nom de *kick-down*). Pour ce faire, nous introduisons une constante Min_Accel au delà duquel l'accélérateur peut être pris en compte. Dans ce cas, le fonctionnement du système est interrompu et il passe en état d'attente `STDB`. La description du mode de fonctionnement `Limit` montre qu'il peut opérer dans deux sous-modes différents : `Limit_On` qui représente le comportement normal du système en mode `Limit`, et `Limit_STDB` qui définit le comportement du système en cas d'un *kick-down*. Les conditions d'activation et le passage entre ces deux sous-modes sont représentés par l'automate de la figure 6.4.

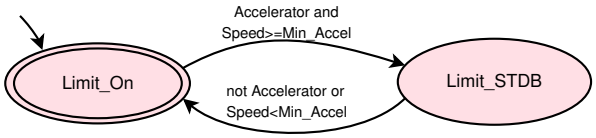


FIG. 6.4 – *Les différents sous-modes de fonctionnement du mode* `Limit`

6.2.3 Le mode `Cruise`

Le mode `Cruise` (Régulateur de vitesse) permet de maintenir la voiture à une vitesse constante donnée par le conducteur. Dans ce mode, le système ne prend pas en considération les pédales et force la voiture à atteindre progressivement la vitesse limite et à la maintenir. Dans ce contexte, le conducteur ne contrôle plus la vitesse de sa voiture via la pédale d'accélération mais plutôt à l'aide d'un ensemble de boutons : les deux boutons `QuickAccel` et `QuickDecel` permettent respectivement d'augmenter ou de diminuer cette vitesse d'une valeur prédéfinie SPEEDINC et le bouton `Set` permet d'affecter la vitesse courante de la voiture à la vitesse limite.

Si le conducteur appuie sur l'accélérateur tandis que le régulateur de vitesse est en mode `Cruise`, le fonctionnement du système est interrompu et le système passe en état `STDB` jusqu'à ce que l'accélérateur soit relâché. Ainsi, un appui sur la pédale de frein désactive complètement le système qui passe en état `Off` et ne peut être réactivé qu'à travers le bouton `Resume`. Cette description montre que le mode `Cruise` peut opérer en trois sous-modes de fonctionnement : `Cruise_On` qui représente le fonctionnement normal du mode `Cruise`, `Cruise_STDB` qui représente l'interruption du mode `Cruise` en cas d'accélération, et `Cruise_Off` qui définit la désactivation du mode `Cruise` en cas d'un freinage. La figure 6.5 représente les conditions d'activation et le passage entre les différents sous-modes de fonctionnement du mode `Cruise`.

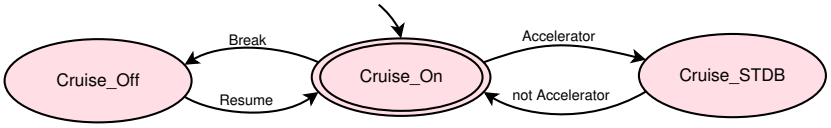


FIG. 6.5 – *Les différents sous-modes de fonctionnement du mode* `Cruise`

6.2.4 Le mode `LimitGPS`

Le comportement du mode `LimitGPS` (Limiteur de vitesse avec GPS) est similaire à celui du mode `Limit`. Dans le mode `LimitGPS`, le véhicule ne peut pas dépasser une vitesse limite définie par le système. Cette limite dépend de la vitesse spécifiée par le conducteur, de la vitesse maximale autorisée dans la zone courante du véhicule et de la vitesse maximale autorisée dans la zone suivante. Dans ce cas, la vitesse limite pris en considération est toujours définie par la valeur minimum de ces trois vitesses pour assurer au maximum la sécurité des passagers. Cependant, en cas de perte du signal GPS, le fonctionnement du mode `LimitGPS` est interrompu et le système passe en état `GPS_Fail`. Le comportement du mode `LimitGPS` est donc composé de deux sous-modes de fonctionnement : `LimitGPS_GPS` qui décrit le comportement par défaut du mode `LimitGPS`, et `LimitGPS_GPS_Fail` qui décrit le comportement du mode `LimitGPS` en cas de perte du signal GPS. Les conditions d'activation et le passage entre ces deux sous-modes sont donnés par l'automate de la figure 6.6.

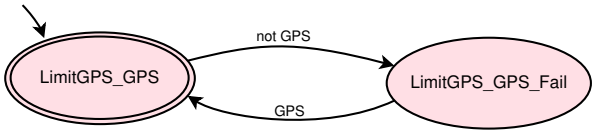


FIG. 6.6 – *Les différents sous-modes de fonctionnement du mode* `LimitGPS`

6.2.5 Le mode `CruiseGPS`

Le comportement du mode `CruiseGPS` (Régulateur de vitesse avec GPS) est similaire à celui du mode `Cruise`. La seule différence est que la vitesse limite est calculée de la même façon que dans le mode `LimitGPS` qui prend en considération la vitesse spécifiée par le conducteur et celles données par le GPS. En cas de perte du signal GPS le système s'arrête automatiquement et passe en état `GPS_Fail`. La description du mode `CruiseGPS` est donc composée de deux sous-modes de fonctionnement : `CruiseGPS_GPS`qui décrit le comportement du mode `CruiseGPS` en présence du signal GPS, et `CruiseGPS_GPS_Fail` qui décrit le comportement du mode `CruiseGPS` en cas de perte du signal GPS. La figure 6.7 représente les conditions d'activation et le passage entre ces deux sous-modes de fonctionnement.

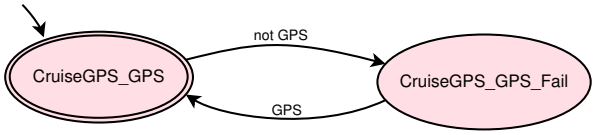


FIG. 6.7 – *Les différents sous-modes de fonctionnement du mode* `CruiseGPS`

6.3 Modélisation de l'application ICCG selon la méthodologie de séparation contrôle/données

Dans cette section, nous allons décrire le comportement de l'application ICCG en utilisant l'environnement de développement SCADE. Le but principal de cette étude est d'illustrer l'utilisation et les avantages de notre méthodologie de séparation contrôle/données dans le cas d'un système réel.

La description de l'application ICCG est principalement composée de deux parties : `ICCG_Prototype` et `CarSimple` comme il est montré par le modèle de la figure 6.8. La partie

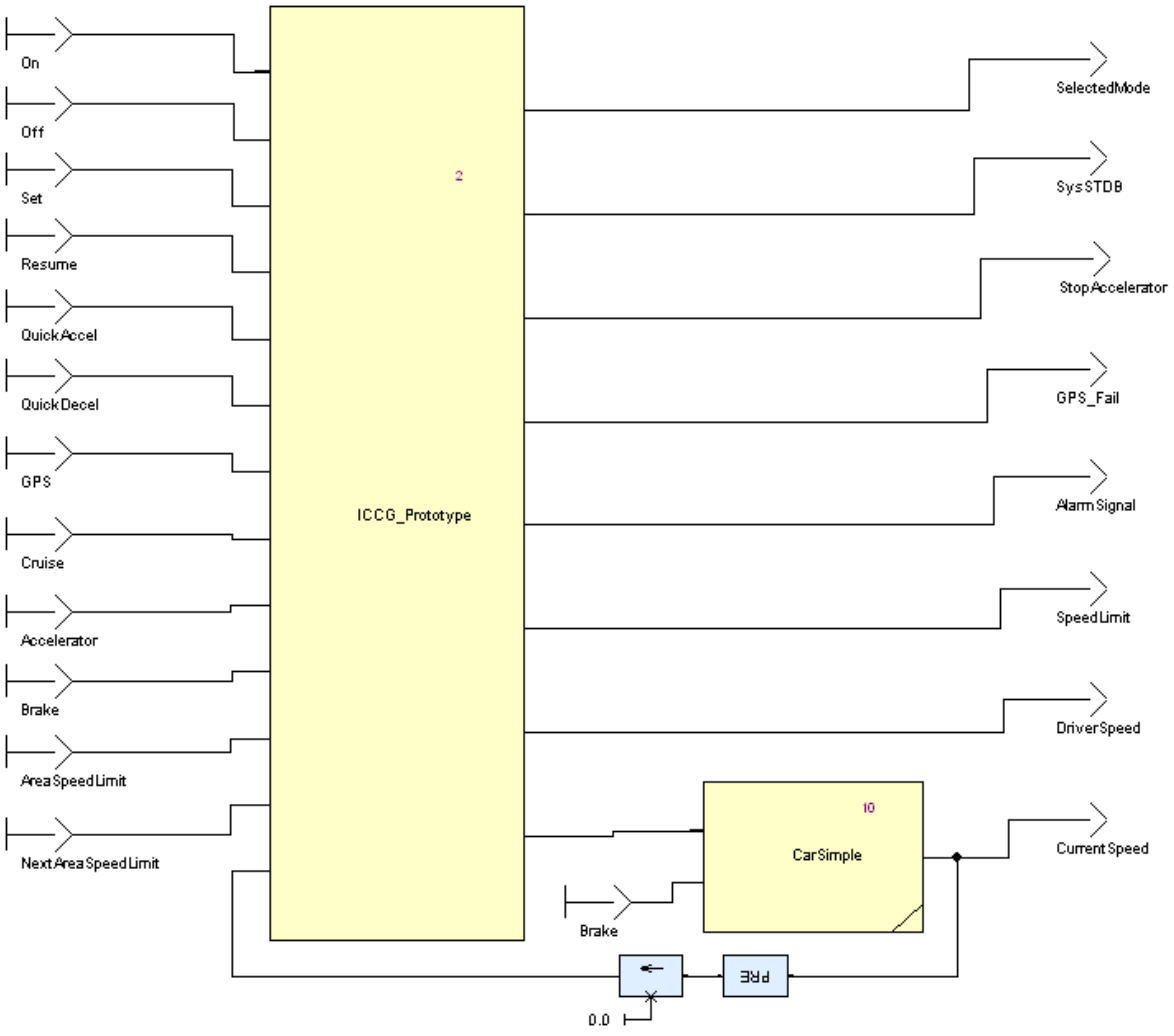


FIG. 6.8 – *Modèle global relatif à l'application ICCG dans* SCADE

de `ICCG_Prototype` représente la comportement du limiteur et régulateur de vitesse étudié, tandis que celle de `CarSimple` spécifie le comportement simplifié d'un véhicule avec lequel notre système de régulation de vitesse va interagir.

Le fonctionnement de base du système ICCG est de permettre, via les boutons de commande, de spécifier le mode de fonctionnement à activer, de sauvegarder la valeur de la vitesse limite spécifiée par le conducteur, de prendre en considération les informations don-

nées par le GPS, et de réagir sur la vitesse du véhicule en fonction du mode sélectionné. Dans le cadre de notre étude, nous nous intéressons uniquement à la description du comportement de la partie `ICCG_Prototype`. Nous basons notre étude sur la méthodologie de conception de cette application en faisant abstraction des détails techniques qui sortent du sujet de notre travail. Il est à noter que l'application `CarSimple` utilisée est celle définie par ESTEREL TECHNOLOGIES comme étant une librairie interne dans SCADE pour la définition de leur système de `Cruise Control`[50]. Le comportement de cette application ne sera donc pas présenté dans cette étude.

L'application ICCG_Prototype prend en entrée un ensemble de valeurs relatives aux différents boutons de commande (`On`, `Off`, `Set`, `Resume`, `QuickAccel`, `QuickDecel`, `GPS`, `Cruise`), aux pédales d'accélération et de freinage du véhicule (`Accelerator`, `Brake`), à la vitesse actuelle du véhicule (`CurrentSpeed`), à la vitesse limite de sa zone de localisation (`AreaSpeedLimit`) et à celle de la zone suivante (`NextAreaSpeedLimit`). Cette dernière représente une anticipation sur la prochaine position du véhicule qui sera calculée en fonction de sa position courante et de sa vitesse. C'est une mesure de sécurité qui peut être utile uniquement lorsque `NextAreaSpeedLimit` <`AreaSpeedLimit`. Par exemple, si le véhicule est sur une autoroute limité à $130km/h$ et se dirige vers une sortie limitée à $90km/h$, la vitesse de cet véhicule doit descendre à $90km/h$ avant d'être sur la sortie d'autoroute. Comme résultat en sortie, l'application ICCG_Prototype fournit un ensemble d'informations permettant d'indiquer le mode de fonctionnement sélectionné (`SelectedMode`), l'état du système en cas d'interruption (`SysSTDB`), l'état de l'accélérateur en cas de blocage (`StopAccelerator`), l'état du signal GPS en cas de perte d'émission ou de non identification de la zone (`GPS_Fail`), le taux d'accélération ou de décélération transmis vers le véhicule pour atteindre la vitesse limite (`Throttle`), et le dépassement de la vitesse limite par un signal d'alarme (`AlarmSignal`).

Une première solution pour la spécification de l'application `ICCG_Prototype` a été proposée dans [LDR06] en utilisant l'environnement de développement SCADE. Le modèle de conception correspondant, représenté par la figure 6.9, a été réalisé de façon très intuitive puisque le but principal était uniquement de développer un modèle fonctionnel sans suivre aucune méthodologie de conception permettant de séparer clairement entre le contrôle et les traitements de données.

Comme il est montré par le modèle de la figure 6.9, la description de l'application `ICCG_Prototype` contient un mélange de traitement de données et du contrôle. Ce modèle ne permet pas de distinguer clairement les différents modes de fonctionnement de l'application, ni les conditions de passage entre ces modes. Sa structure est très ambiguë et il est difficile, voir impossible, d'extraire le modèle correspondant à la description du comportement d'un certain mode. Dans ce cas, toute modification dans le comportement d'un mode de fonctionnement particulier nécessite la modification de l'application entière. Il est également difficile d'introduire un nouveau mode de fonctionnement ou de supprimer un mode déjà existant. Ainsi, l'application des techniques de validation et de vérification formelle sur un tel modèle est compliquée et ne permet pas de détecter les causes des erreurs qui deviennent de plus en plus sérieuses. Le système résultant est par conséquent instable et sa conception est inacceptable dans le domaine des systèmes critiques.

Pour ces raisons, nous avons modifié la spécification de l'application `ICCG_Prototype` en adoptant notre méthodologie de séparation contrôle/données présentée dans le chapitre 5. Cette méthodologie permet d'avoir une séparation claire entre la partie contrôle et la par-

[50] `http://www.esterel-technologies.com/technology/demos`

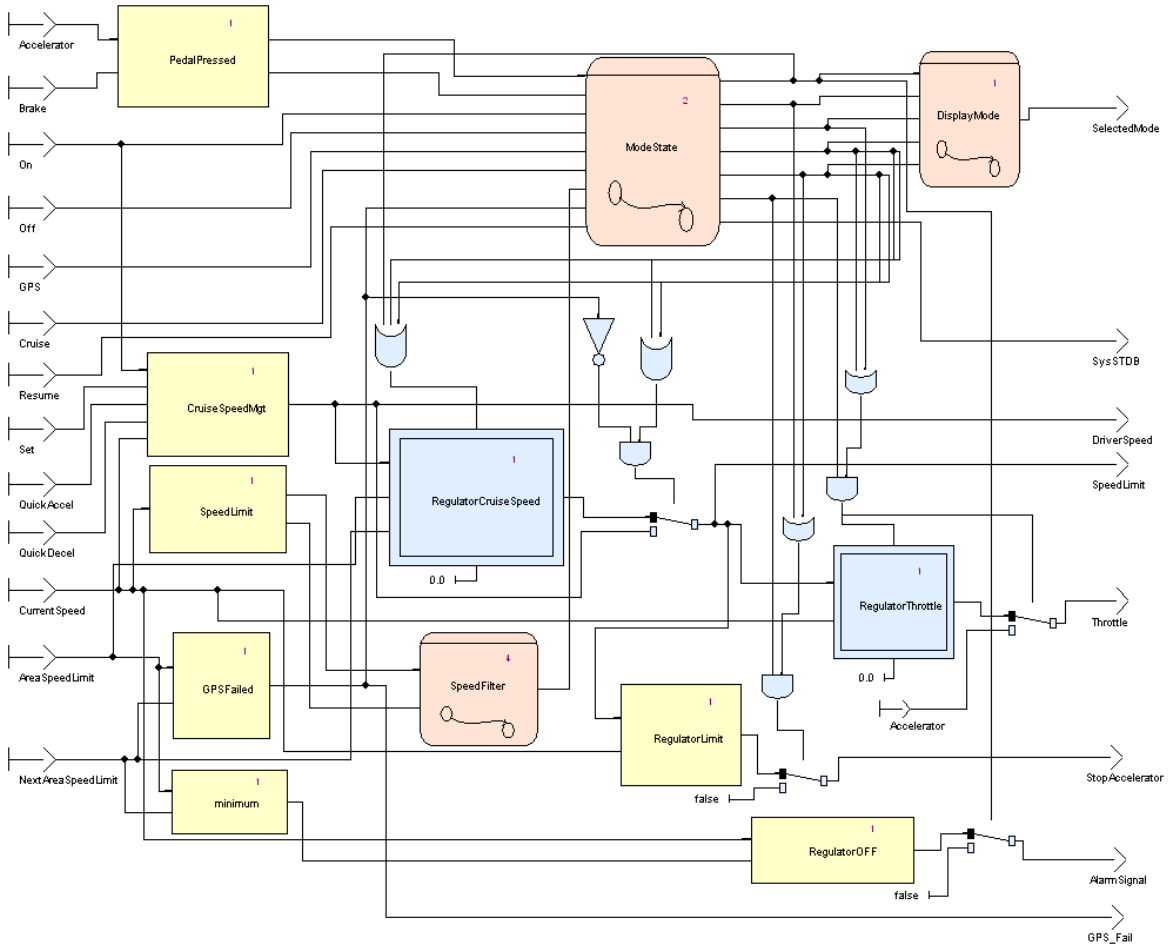


FIG. 6.9 – *Modèle de conception de l'application* `ICCG_Prototype` *sans suivre aucune méthodologie de séparation contrôle/données*

tie données, ainsi qu'une spécification séparée des différents modes de fonctionnement. Le modèle de conception relatif à l'application `ICCG_Prototype` selon notre méthodologie est représenté par la figure 6.10. Ce modèle est composé de quatre parties principales : une partie de pré-traitement, une partie de contrôle, une partie des modes de fonctionnement et une partie représentant le `Join` pour les résultats finals.

La première partie est relative au composant `Pre_Compuation` qui regroupe les parties de calcul communes aux différents modes de fonctionnement de l'application. Le composant `Pre_Computation`, dont la structure est donnée par la figure 6.11, permet de déterminer le taux de pression sur les pédales du véhicule, de calculer la vitesse définie selon les choix du conducteur, et de déterminer la vitesse limite à prendre en considération selon la zone de localisation du véhicule. Cette partie de pré-traitement est toujours exécutée indépendamment du mode de fonctionnement sélectionné. Elle fournit en sortie un ensemble de valeurs qui peuvent être utilisées par les différents modes de fonctionnement de l'application, ou être directement affichées sur le tableau de bord.

La deuxième partie représente la partie contrôle de l'application `ICCG_Control`. Cette partie permet de définir les conditions de passage entre les différents modes et de piloter leur

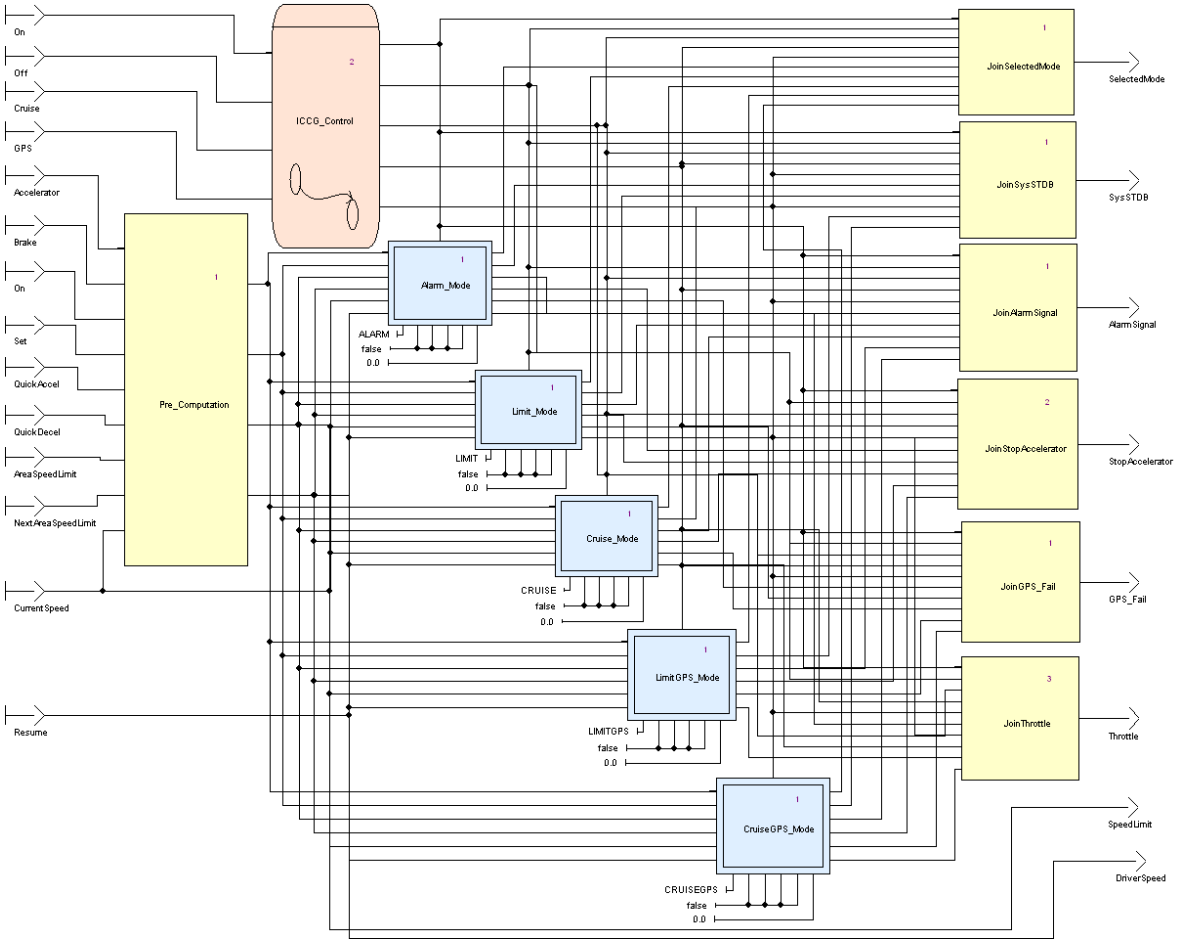


FIG. 6.10 – *Modèle de conception de l'application* `ICCG_Prototype` *selon la méthodologie de séparation contrôle/données*

activation selon les boutons de commande appuyés. Cette partie de contrôle est modélisée par l'automate de la figure 6.12.

La troisième partie regroupe les cinq modes de fonctionnement de l'application qui sont représentés par les composants conditionnés `Alarm_Mode`, `Limit_Mode`, `Cruise_Mode`, `LimitGPS_Mode` et `CruiseGPS_Mode`. Ces composants sont par nature exclusifs et peuvent être activés uniquement par l'automate de contrôle `ICCG_Control`. Ainsi, nous avons défini la même interface pour les différents modes de fonctionnement pour faciliter leur manipulation. Dans ce cas, les entrées non consommées par un certain mode sont ignorées, tandis que les sorties non définies sont remplacées par les valeurs par défaut. Par exemple, la figure 6.13 représente le modèle relatif à la description du mode de fonctionnement `Alarm`. Dans ce modèle, les entrées `DriverSpeed`, `Resume`, `Accelerator` et `Brake` ne sont pas utilisées, et aux sorties `SelectedMode`, `StopAccelerator` et `Throttle` sont associées les valeurs par défaut. Ainsi, le comportement du mode `Alarm` est composé de deux sous-modes : `Alarm_GPS` et `Alarm_GPS_Fail` dont l'activation est pilotée par l'automate de contrôle `Alarm_Control`. Le modèle de conception du mode `Alarm` est similaire à celui de l'application globale dans le sens où il est basé sur note méthodologie de séparation contrôle/donnée. Ceci est également vrai pour les modèles relatifs aux autres modes de fonctionnement de l'application

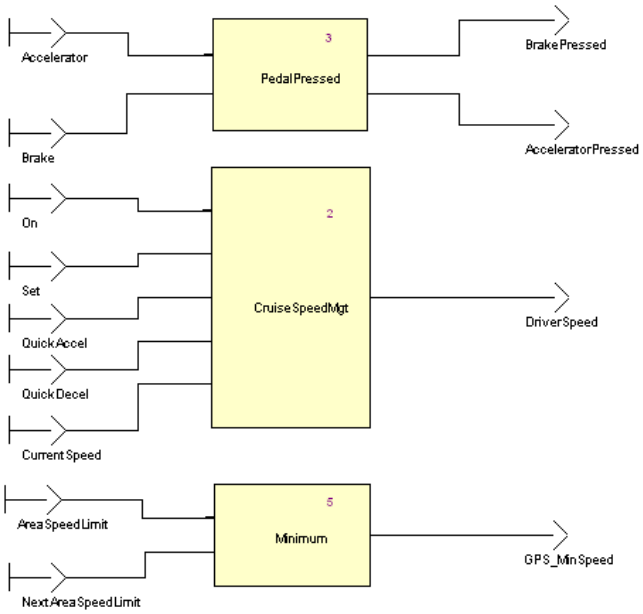


FIG. 6.11 – *Modélisation de la partie de pré-traitement pour l'application* `ICCG_Prototype`

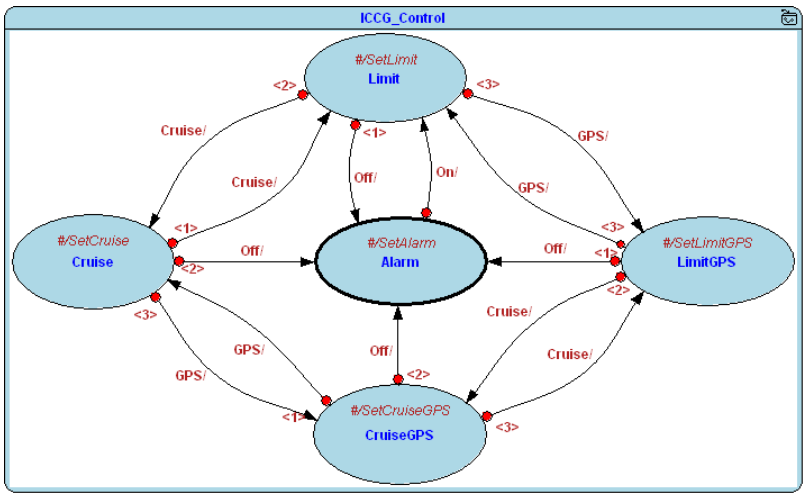


FIG. 6.12 – *Modélisation de l'automate de contrôle pour l'application* `ICCG_Prototype`

`ICCG_Prototype`.

La quatrième partie représente une codification des opérateurs `Join` utilisés pour la définition des valeurs en sortie. Puisque l'opérateur `Join` n'est pas implémenté dans SCADE, nous allons définir explicitement son comportement en utilisant l'opérateur `If_Then_Else` comme il est montré par le modèle de la figure 6.14 qui définie la structure du `Join` utilisé pour la définition de la sortie `AlarmSignal`.

Le modèle de la figure 6.10 représente un modèle de conception plus clair et plus facile à utiliser. La distinction des différents modes de fonctionnement est évidente ce qui rend possible leur développement et étude séparés grâce à leur nature indépendante et exclusive.

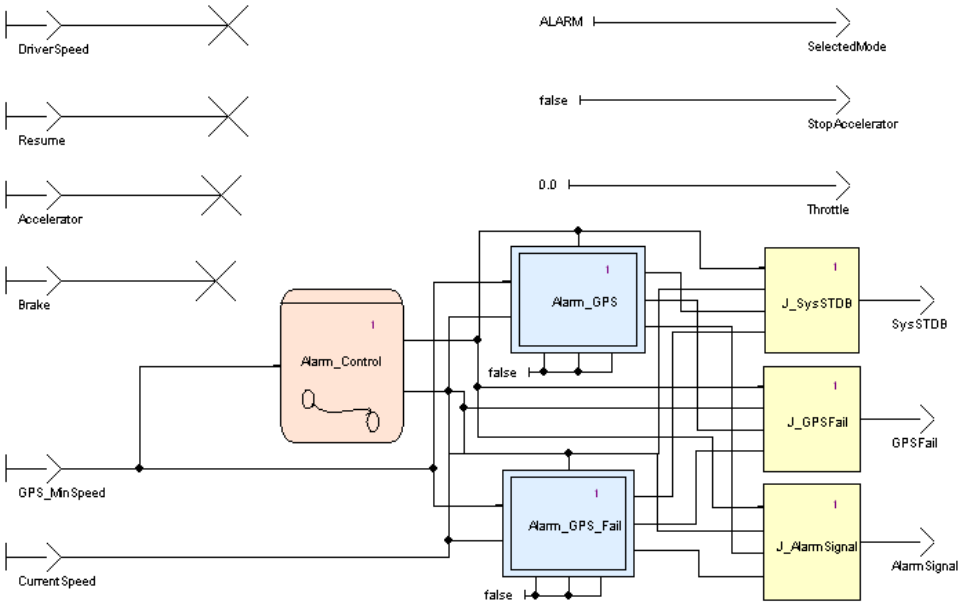


FIG. 6.13 – *Modélisation du mode de fonctionnement* `Alarm`

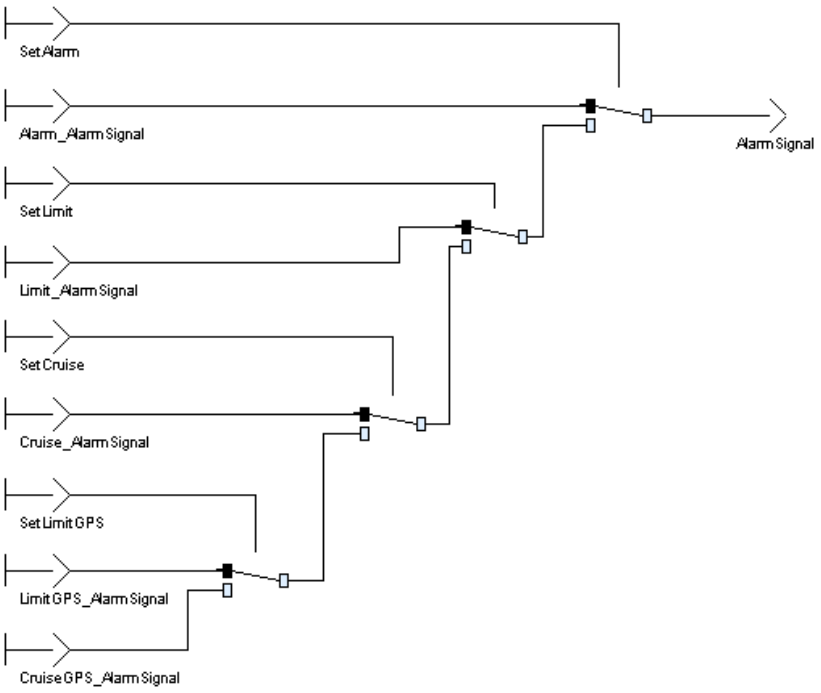


FIG. 6.14 – *Modélisation de l'opérateur* `Join` *pour la sortie* `AlarmSignal`

6.4 Expérimentation du système ICCG

Dans ce qui suit, nous allons présenter quelques études expérimentales sur le fonctionnement du système ICCG. Le but principal de cette étude est de montrer les avantages de la modélisation séparée des différents composants du système. En utilisant l'outil de déve-

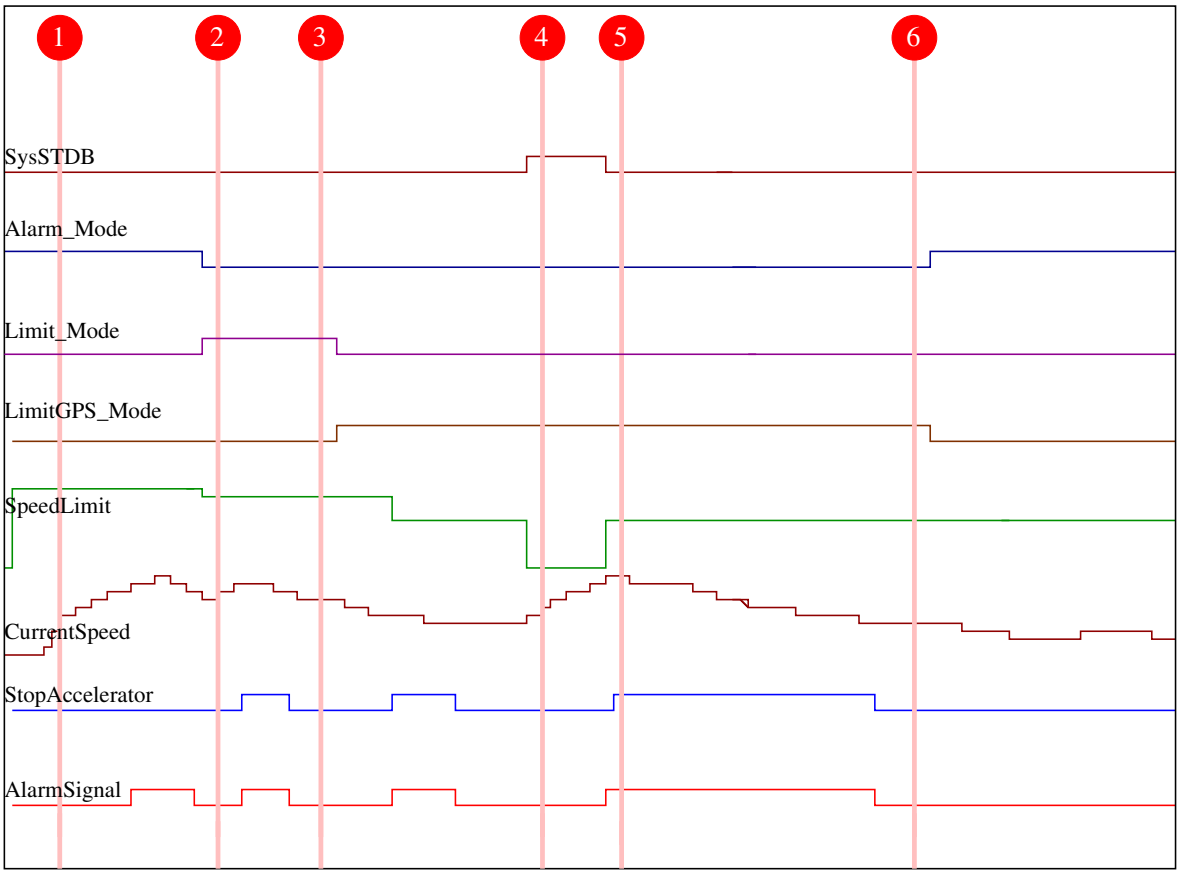


FIG. 6.15 – *Exemple de simulation des modes de fonctionnement : `Alarm`, `Limit` et `LimitGPS`*

loppement SCADE, nous avons effectué des processus de simulation et de vérification pour assurer le bon fonctionnement du système.

6.4.1 Simulation du système ICCG

Pour tester le bon fonctionnement du système ICCG, nous avons simuler le fonctionnement des différents modes du système en utilisant l'outil de simulation de SCADE. Pour des raisons de simplification, nous avons divisé notre processus de simulation en deux parties : la première partie étudie la simulation des modes de fonctionnement `Alarm`, `Limit` et `LimitGPS`, tandis que la deuxième partie étudie la simulation des modes de fonctionnement `Alarm`, `Cruise` et `CruiseGPS`.

La figure 6.15 représente le résultat de simulation de la première partie permettant de tester les modes de fonctionnement `Alarm`, `Limit` et `LimitGPS`. Ce processus représente six étapes de simulation :

1. Le mode `Alarm` est activé et la valeur de `SpeedLimit` est fixée à la valeur minimum de la vitesse courante et celles données par le GPS. Lorsque la vitesse courante (`CurrentSpeed`) dépasse la vitesse limite (`Speedlimit`), le signal d'alarme (`AlarmSignal`) est activé jusqu'à ce que la vitesse courante devient inférieur à la vitesse limite

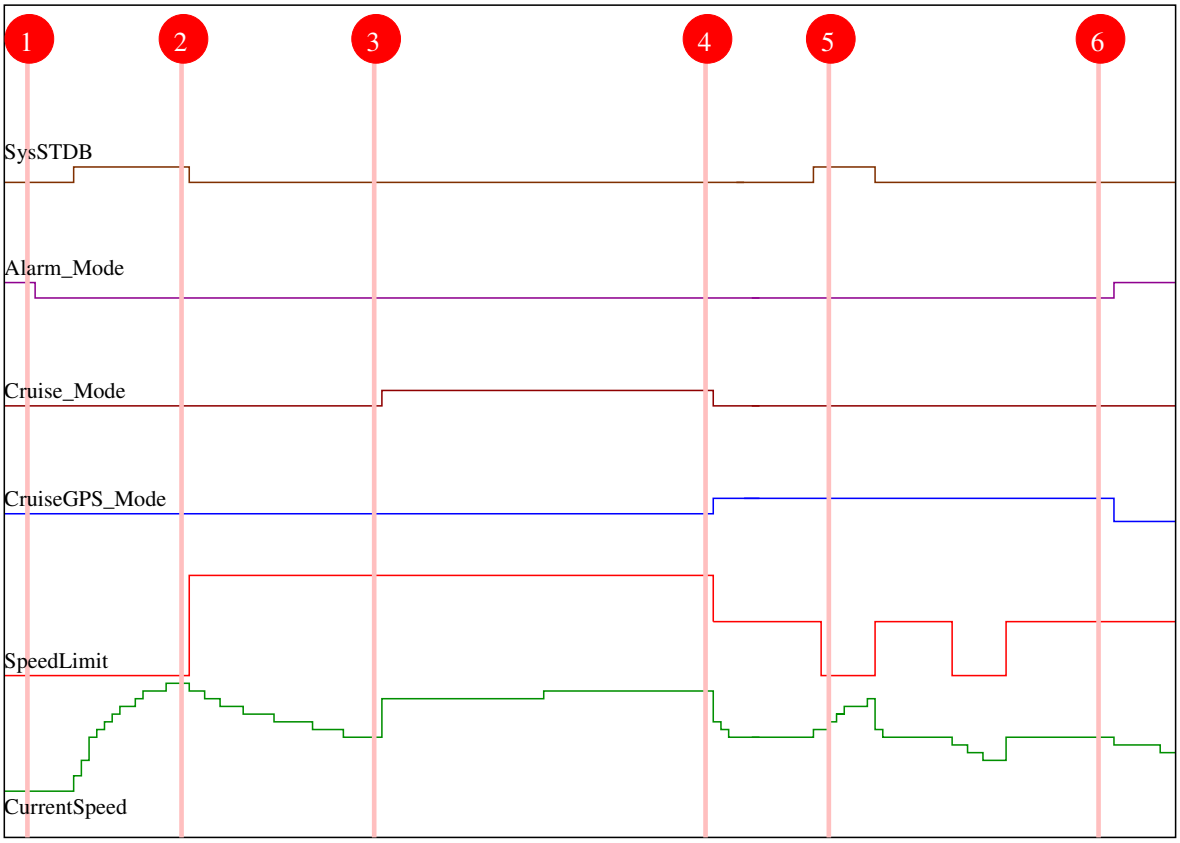


FIG. 6.16 – *Exemple de simulation pour les modes de fonctionnement :* `Alarm`, `Cruise` *et* `CruiseGPS`

2. Le bouton `On` est appuyé et le mode `Limit` est activé. Dans ce cas, la valeur de `CurrentSpeed` est inférieur à celle de `SpeedLimit`, et si le conducteur veut dépasser cette vitesse, l'accélérateur est bloqué et le signal d'alarme est envoyé
3. Le bouton GPS est appuyé et le mode `LimitGPS` est activé. Dans ce mode, la valeur de `SpeedLimit` représente le minimum des valeurs de la vitesse spécifiée par le conducteur et de celles données par le GPS. Ainsi, si la valeur de `CurrentSpeed` dépasse celle de `SpeedLimit`, alors l'accélérateur est bloqué et le signal d'alarme est envoyé
4. La pédale d'accélération est appuyé dans le cas d'un *kick-down* et le système est interrompu et passe en état STDB
5. La pédale d'accélérateur est relâché et le fonctionnement du système est réactivé. Dans ce cas, la valeur de `SpeedLimit` prend la dernière valeur avant l'interruption du système. Ainsi, l'accélérateur est bloqué et le signal d'alarme est envoyé puisque la valeur de `CurrentSpeed` est supérieur à celle de `SpeedLimit`
6. Le bouton `Off` est appuyé et le système passe en mode `Alarm`

De façon similaire, la figure 6.16 représente le résultat de simulation pour la deuxième partie permettant de tester les modes de fonctionnement `Alarm`, `Cruise` et `CruiseGPS`. Dans ce processus, nous avons également représenté six étapes de simulation :

1. Le bouton `On` est appuyé et le système est activé en mode `Alarm`. Par la suite, la pédale d'accélération est pressée en cas d'un *kick-down* ce qui interrompt le fonctionnement

du système qui passe en état STDB

2. La pédale d'accélération est relâchée, le bouton `On` est appuyé et le système passe en mode `Limit`
3. Le bouton `Cruise` est appuyé permettant d'activer le mode `Cruise`. Dans ce cas, la valeur de `CurrentSpeed` atteint celle de `SpeedLimit`
4. Le bouton GPS est appuyé et le système passe en mode `CruiseGPS`. Dans ce cas, la valeur de `SpeedLimit` change lorsque la valeur de la vitesse limite dans la zone de localisation du véhicule est différente de celle de `SpeedLimit`
5. La pédale d'accélération est pressée dans le cas d'un *kick-down* et le système passe en état STDB. Lorsque la pédale d'accélération est relâchée, le système est réactivé et la valeur de `SpeedLimit` prend la dernière valeur avant l'interruption
6. La pédale de frein est pressé et le système est désactivé. Dans ce cas, seul le bouton `Resume` permet de réactiver le système

6.4.2 Vérification formelle du système ICCG

Le but de cette étude est de vérifier un ensemble de propriétés sur le comportement du système ICCG pour assurer son bon fonctionnement. Nous allons appliqué le processus de vérification pour les deux modèles de conception : le modèle sans séparation contrôle/données représenté par la figure 6.9, et notre modèle basé sur la méthodologie de séparation représenté par la figure 6.10. L'objectif est de comparer les résultats obtenus par chaque modèle et montrer les avantages en terme de vérification formelle de notre méthodologie de séparation contrôle/données. Dans ce qui suit, nous allons noter le premier modèle de conception par `Model1` et le deuxième par `Model2`. L'ensemble de propriétés que nous proposons de vérifier est comme suit :

- *Propriété P1* : à chaque instant, un et un seul mode de fonctionnement est activé à la fois
- *Propriété P2* : si le système est en mode `Alarm` alors la valeur de `AlarmSignal` est *True* si et seulement si `CurrentSpeed`>`SpeedLimit`
- *Propriété P3* : si le système est en mode `Alarm` alors le système n'a aucun effet sur l'accélérateur (`StopAccelerator`=*False*)
- *Propriété P4* : si le système est en mode `Limit` et `CurrentSpeed`>`SpeedLimit` alors l'accélérateur est bloqué (`StopAccelerator`=*True*)
- *Propriété P5* : si le système est en mode `LimitGPS` ou en mode `CruiseGPS` alors la valeur de `SpeedLimit` est le minimum des valeurs de `DriverSpeed`, `AreaSpeedLimit` et `NextAreaSpeedLimit`

Comme nous l'avons spécifié dans la section 6.3, le modèle de conception `Model1` contient un mélange de traitement de données et du contrôle ce qui complique la compréhension du modèle et la distinction des différents modes de fonctionnement du système. Dans ce cas, la vérification d'une certaine propriété pour un mode particulier nécessite la vérification de l'application entière puisque la distinction du mode concerné est très difficile, voir impossible. Contrairement à ce modèle, le modèle de conception `Model2` permet une séparation claire entre la partie contrôle et les traitements de données, et par conséquent, facilite la distinction des différents modes de fonctionnement de l'application. Dans ce cas, la vérification d'une certaine propriété pour un mode donné nécessite uniquement la vérification du mode concerné grâce au caractère exclusif et indépendant des différents modes de l'application.

Ainsi, la formule de la propriété à vérifier est moins compliquée que celle utilisée pour la vérification du premier modèle. Par exemple, la vérification de la propriété *P2* pour le modèle `Model1` nécessite de vérifier que le système est en mode `Alarm`. Ceci n'est pas le cas pour la vérification de la même propriété pour le modèle `Model2` puisque nous savons dès le départ que le mode vérifié est le mode `Alarm`.

Le tableau 6.1 représente le résultat de la comparaison des temps de vérification obtenus lors la vérification des différentes propriétés pour les deux modèles de conception `Model1` et `Model2`. Ces résultats sont obtenus en utilisant l'outil de vérification formelle de SCADE appelé *Design Verifier*[51], et qui est basé sur la technique de vérification de type *model checker*. Dans ce cas, la propriété à vérifier est généralement décrite dans un nœud séparé qui sera lié au modèle de l'application sans modifier sa structure, et qui représente un *observateur* pour cette application.

Propriété	***Module vérifié pour `Model1`***	***Module vérifié pour `Model2`***	***Temps `Model1`***	***Temps `Model2`***	***Accélération***[a]
P1	Tout le système	SSM `ICCG_Control`	0.46*s*	0.08*s*	**5.75**
P2	Tout le système	Mode `Alarm`	0.43*s*	0.03*s*	**14.33**
P3	Tout le système	Mode `Alarm`	3.42*s*	0.01*s*	**342.00**
P4	Tout le système	Mode `Limit`	0.44*s*	0.04*s*	**11.00**
P5	Tout le système	Mode `LimitGPS`	8.70*s*	0.08*s*	**108.75**
		Mode `CruiseGPS`		0.09*s*	**96.66**

[a] L'accélération est calculée comme suit : $\frac{t_{Model1}}{t_{Model2}}$, où t_{Model1} t_{Model2} représentent respectivement les temps de vérification pour les modèles `Model1` et `Model2`.

TAB. 6.1 – *Temps de vérification des différentes propriétés pour les modèles de conception `Model1` et `Model2`*

Les résultats donnés par le tableau 6.1 montrent que le temps de vérification pour le modèle `Model2` est largement inférieur à celui donné par la vérification du modèle `Model1`. Ceci est dû au fait que le nombre d'états de l'automate vérifié pour un seul mode de fonctionnement est largement inférieur à celui du système complet. Ainsi, cette différence en temps de vérification est également relative au fait que les formules des propriétés vérifiées sont moins compliquées dans le cas de la vérification séparée des modes. Cette technique de vérification que nous appelons « *vérification modulaire* » permet de mieux localiser les erreurs, et d'avoir un gain considérable en temps de vérification et en espace mémoire tout en minimisant le problème d'explosion combinatoire. Cette technique facilite la spécification et la vérification séparées et modulaires des systèmes critiques tout en assurant le bon fonctionnement du système global.

[51] `http://www.esterel-technologies.com/products/scade-drive/design-verifier.html`

6.4.3 Prototype et test sur le terrain du système ICCG

Pour tester le fonctionnement du système ICCG dans des conditions réelles, nous avons développé[52] un prototype de simulation en utilisant les langages Visual Studio .Net et C#[53].

Le prototype développé est principalement composé de deux applications : la première application est relative à une carte de simulation permettant de définir une feuille de route et de créer la base de données de cette carte. Généralement, cette application consiste à représenter graphiquement sur une feuille de route les vitesses limites des différentes zones comme il est montré par la figure 6.17. La deuxième application est développée pour la si-

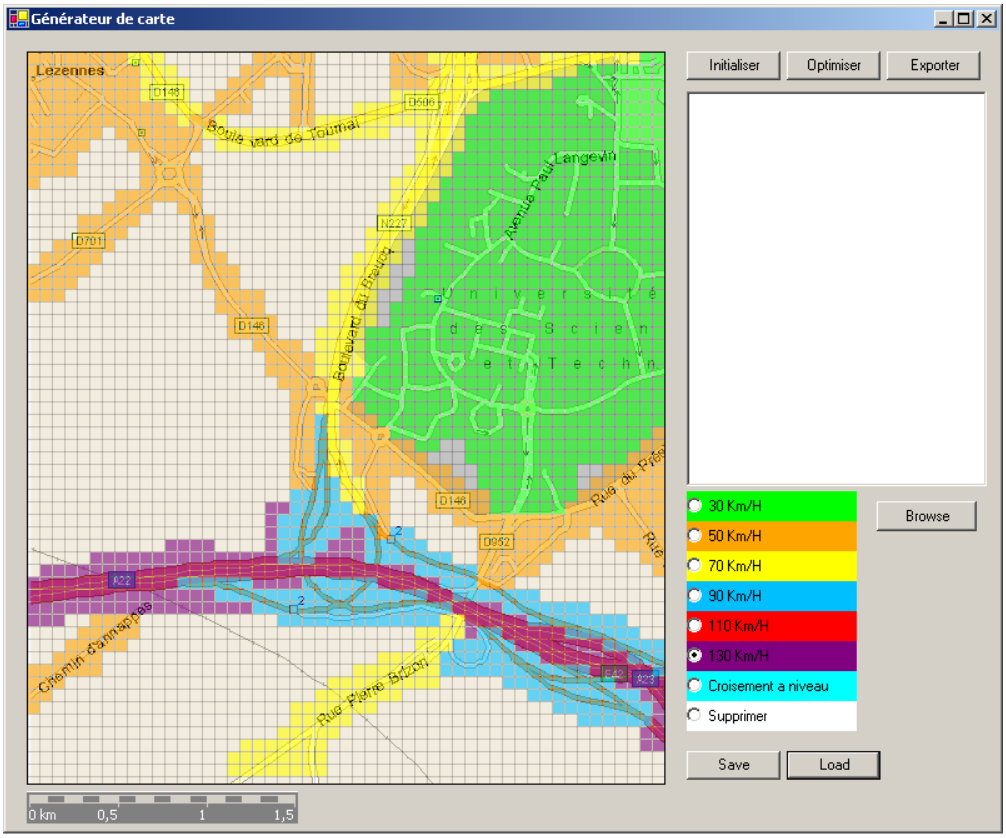


FIG. 6.17 – *Représentation de l'application de simulation de la feuille de route*

mulation du fonctionnement du système ICCG dans des conditions réelles. Pour ce faire, nous avons relié notre système à un GPS de type Garmin[54]. Ce dernier utilise le protocole de transmission NMEA (National Marine Electronic Association) [MGH02] pour envoyer des informations chaque seconde sur la position du véhicule au prototype ICCG. Dans ce cas, le prototype calcule en temps réel la vitesse du véhicule et sa direction en fonction de sa position courante et en utilisant des données graphiques (latitude et longitude). En fonction de ces informations et de la base de données de la feuille de route produite par le simulateur,

[52] Je me dois de rajouter que l'implémentation du prototype a été réalisée par les étudiants Ahmed Jerbi et Yousri Miled.

[53] `http://www.hitmill.com/programming/dotNET/csharp.html`

[54] `http://www.garmin.com`

le prototype localise les positions courante et suivante du véhicule et les limites de vitesse à respecter.

Nous avons également effectué un test sur le terrain de notre système de régulation de vitesse ICCG. Ce test a été effectué à Villeneuve d'Ascq[55] en utilisant un ordinateur portable et un GPS de type Garmin comme il est montré par la figure 6.18. Cette application repré-

FIG. 6.18 – *Test réel du système ICCG*

sente un test simplifié dans le sens où notre système ne réagit pas directement sur la vitesse du véhicule. Dans ce cas, le système ICCG prévient uniquement le conducteur par un signal d'alarme si la vitesse limite est dépassée.

Pour afficher les différentes informations sur l'état du système et sur les changements de vitesse, nous avons proposé une interface graphique permettant d'afficher la feuille de route, les positions courante et suivante du véhicule, la vitesse du véhicule, la vitesse limite, etc. Nous avons également proposé une interface pour la simulation des différents boutons du système ICCG comme il est montré par la figure 6.18.

Il est également à noter que, dans le processus de test effectué, nous nous somme beaucoup plus intéressé par la prise en compte des vitesses limites données par le GPS. Pour ce faire, nous avons fixé la vitesse spécifiée par le conducteur à $140km/h$. Dans ce cas, la vitesse limite autorisée et qui sera prise en compte est toujours celle de la zone de localisation du véhicule. Le test effectué a donné des résultats satisfaisants et a prouvé la précision de la détection des différents changements des zones. Par exemple, le passage d'une zone limité à $90km/h$ à une zone limité à $70km/h$ a été très rapidement détecté par le système et le signal d'alarme a été activé jusqu'à ce que la vitesse du véhicule va au-dessous de $70Km/h$.

6.5 Synthèse et conclusion

Dans ce chapitre, nous avons présenté l'étude d'un système de limiteur et régulateur de vitesse intelligent avec GPS. Le but principal de cette étude était de montrer les avantages de l'application de notre méthodologie de séparation contrôle/données notamment en ce qui concerne l'application des processus de vérification formelle. Pour ce faire, nous avons comparé les modèles de conception ainsi que les résultats de vérification de certaines propriétés

[55] 59650, France

pour deux réalisation différentes de notre système : selon et sans méthodologie de séparation. Les résultats obtenus ont montré les avantages de l'utilisation de la méthodologie de séparation contrôle/données pour assurer le bon fonctionnement des systèmes critique. Le développement modulaire proposé par cette méthodologie et le caractère indépendant et exclusif des différents modes de fonctionnement de l'application favorise le développement séparé et permettent d'avoir un un modèle plus claire et plus facile à étudier, à vérifier et à réutiliser. Il est également facile d'introduire de nouveaux modes de fonctionnement sans avoir à modifier le comportement global de l'application. L'introduction d'un mode de fonctionnement utilisant un radar au système ICCG est présentée dans [LMLD06].

Nous avons également étudié quelques expérimentations de notre système pour tester son bon fonctionnement via le développement d'un prototype et d'un simulateur. Un test réel sur le terrain a été également effectué et a permis d'assurer que le système ICCG répond bien à ses fonctionnalités.

Troisième partie

Introduction du Contrôle dans le Profil GASPARD2

Chapitre 7

Introduction du contrôle dans les applications de traitement systématique à parallélisme massif

Dans ce chapitre, nous allons étudier l'introduction du contrôle pour les applications massivement parallèles, et en particulier pour celles décrites dans le langage de spécification ARRAY-OL. Nous allons montrer que l'introduction du contrôle dans ARRAY-OL nécessite la définition des différents moments de prise en compte des valeurs d'événements responsables des changements de modes. Pour ce faire, et en se basant sur les fonctionnalités attendues de l'application, nous allons introduire la notion de « degré de granularité » permettant d'associer un comportement réactif aux applications parallèles. Nous présentons le principe de cette notion à travers des exemples académiques, et selon le type des événements de contrôle qui peuvent parvenir de l'environnement externe ou d'un résultat interne de l'application. Nous montrons par la suite qu'il est possible que différentes parties d'une même application soient contrôlées, et que les événements de contrôle soient pris en compte selon des degrés de granularité différents pour chacune de ces parties. Nous allons donc étendre la notion de degré de granularité à celle de « multi-degrés de granularité » pour permettre la définition du comportement global des applications plus complexes à plusieurs niveaux de contrôle. À la fin de ce chapitre, nous étudions la simulation de notre approche dans l'environnement de développement PTOLEMY II en se basant principalement sur les concepts d'ARRAY-OL *for* PTOLEMY II et sur ceux du MODALMODEL. Cette implémentation permet de bien illustrer le concept de degré de granularité, et de faciliter l'étude de comportement des applications mixant des traitements de données parallèle et du contrôle. L'approche que nous allons présenter est une approche orientée fonctionnelle, elle est basée sur la technologie synchrone, et elle est fortement inspirée du concept des automates de modes pour la représentation des systèmes hybrides à modes de fonctionnement.

7.1 Introduction

Comme nous l'avons discuté dans le chapitre 2, le contexte de notre travail est basé sur l'étude des applications de traitement intensif et systématique, et en particulier sur celles de traitement du signal intensif. Nous avons montré, à travers quelques exemples, que les applications de traitement du signal complexes contiennent généralement un mélange de traitement de données et du contrôle, ce qui nécessite le développement de nouvelles méthodologies de conception permettant de prendre en considération cette mixité de comportement.

Nous avons également montré l'intérêt du modèle ARRAY-OL et sa puissance d'expression pour la spécification des applications de traitement du signal intensif sur lesquelles se base notre étude. Cependant, les applications modélisées dans ARRAY-OL sont purement fonctionnelles, et traitent uniquement des données intensives sans aucune prise en compte des aspects non fonctionnels. Ces derniers, tels que les contraintes temporelles et les concepts de contrôle, sont induits par les différentes configurations ou modes de fonctionnement du système, et permettent la description des comportements plus complexes reflétant des systèmes plus réels.

Le but principal de notre travail est donc d'introduire, dans le modèle de spécification ARRAY-OL, des mécanismes permettant la spécification des notions de contrôle et des changements de modes de fonctionnement pour les applications de traitement du signal intensif en fonction de leur contexte d'exécution. Il est également à noter que l'introduction du contrôle dans le modèle de spécification ARRAY-OL sera basée sur notre méthodologie de séparation contrôle/données présentée dans le chapitre 5. Ceci nous permettra de tirer pro-

fit des différents avantages de cette méthodologie sans avoir besoin de modifier le modèle ARRAY-OL de base. Il est donc facile de réutiliser, d'une part, les applications ARRAY-OL déjà existantes et de les enrichir par des comportements de contrôle, et d'autre part, les outils et les résultats déjà développés autour du modèle de spécification ARRAY-OL.

7.2 Introduction du contrôle dans les applications parallèles en ARRAY-OL

Dans la section 2.4 (page 24), nous avons montré que la description du modèle ARRAY-OL peut être principalement basée sur la notion de demi-tâche (TÂCHE, TABLEAU) qui spécifie les tâches de calcul associées aux tableaux en entrée et en sorties, et celle du TILER qui permet la définition de la relation entre les tableaux et les tâches de calcul. À partir de cette description, nous pouvons distinguer deux situations possibles de changement de modes de fonctionnement dans lesquelles des événements de contrôle peuvent être pris en compte.

1. Changement des tâches de calcul globales ou locales
2. Changement des TILERS

Pour illustrer ces deux situations de changement de modes, nous présentons dans la figure 7.1 un exemple simple d'une application ARRAY-OL. Dans cet exemple, la tâche de

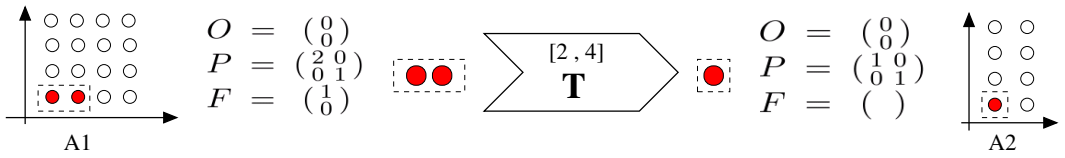


FIG. 7.1 – *Exemple d'une application* ARRAY-OL

calcul T est répétée 2×4 fois. Pour chaque répétition, cette tâche prend en entrée un motif de 2 éléments pour produire un motif d'1 seul élément. La construction des motifs en entrée à partir du tableau d'entrée A1, ainsi que le rangement des motifs en sortie dans le tableau résultat A2 sont spécifiés en fonction des informations données par le TILER d'entrée et celui de sortie.

Nous considérons maintenant que l'application décrite dans la figure 7.1 est sensible à son environnement d'exécution, dans le sens où elle peut changer son comportement en fonction d'un certain événement de contrôle. En premier lieu, nous imaginons une situation

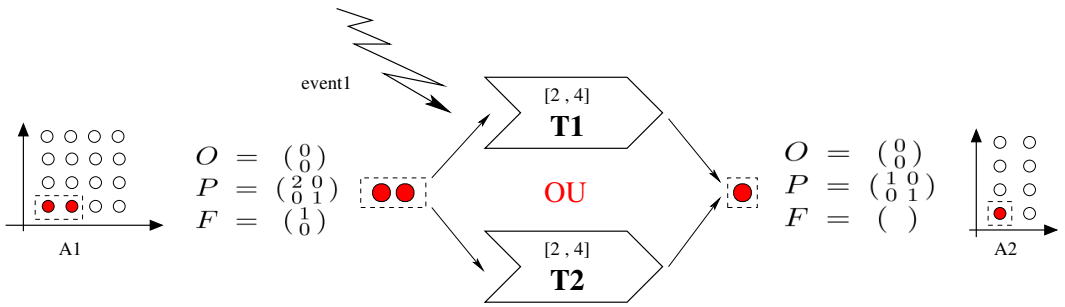


FIG. 7.2 – *Changement de tâche de calcul dans une application* ARRAY-OL

dans laquelle la tâche de calcul T peut être remplacée, en fonction de la valeur d'événement event1, par deux tâches de calcul différentes, T1 ou T2, ayant la même interface (même type et taille des motifs en entrée et en sortie) comme il est montré par la figure 7.2.

Une autre situation possible, dans laquelle l'application peut changer son mode de fonctionnement, consiste à modifier une ou plusieurs informations du TILER en entrée et/ou en sortie. La figure 7.3 représente un exemple dans lequel le point d'origine pour le TILER en

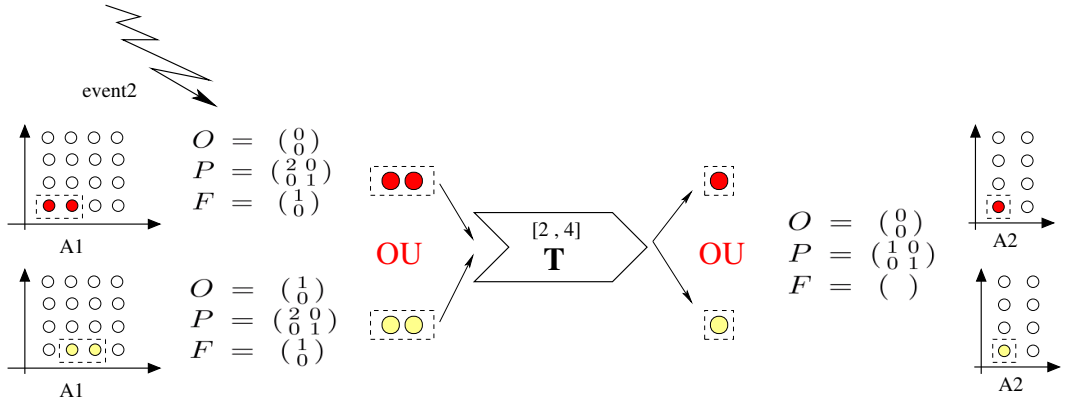


FIG. 7.3 – *Changement de* TILER *dans une application* ARRAY-OL

entrée peut être modifié en fonction de la valeur d'événement event2. En d'autre termes, le tableau en entrée peut être parcouru différemment en fonction de la valeur de cet événement.

Nous pouvons également imaginer d'autres situations plus complexes dans lesquelles il est possible de changer les matrices de pavage, les matrices d'ajustage, ou encore la taille des motifs. Ces différentes situations dépendent fortement du comportement attendu des applications étudiées, et doivent avoir une sémantique bien claire et déterministe pour leur fonctionnement.

Les exemples présentés montrent la possibilité de changement de modes de fonctionnement pour des applications spécifiées en ARRAY-OL. Cependant, dans tous les cas, la problématique principale pour le changement de mode est la spécification des instants ou moments dans lesquels les événements de contrôle peuvent être pris en compte. Nous rappelons que la notion de temps dans le modèle ARRAY-OL est banalisée. Dans ce modèle, toutes les tâches indépendantes peuvent s'exécuter en parallèle, et il est donc difficile de spécifier les instants de changement de modes dans une application parallèle décrite en ARRAY-OL.

De façon générale, nous considérons que l'automate de contrôle produit un **tableau de modes** qui sera consommé par la partie contrôlée de l'application parallèle, et utilisé pour déterminer le mode de fonctionnement à activer. Il est donc nécessaire de coordonner et de synchroniser les rythmes de fonctionnement de l'automate de contrôle et de la partie contrôlée pour assurer les fonctionnalités attendues de l'application étudiée (figure 7.4). Sachant que la sémantique de l'automate de contrôle est basée sur une notion de flot, et celle des modèles ARRAY-OL est basée sur la notion de dépendances de données où le temps est banalisé, la principale question à poser est : *comment mélanger et faire communiquer ces deux mondes différents ?*.

L'introduction du contrôle dans le modèle ARRAY-OL exige donc la définition d'un modèle clair, et d'une sémantique rigoureuse permettant la spécification des moments dans lesquels la prise en compte des événements de contrôle devient possible. Ce concept corres-

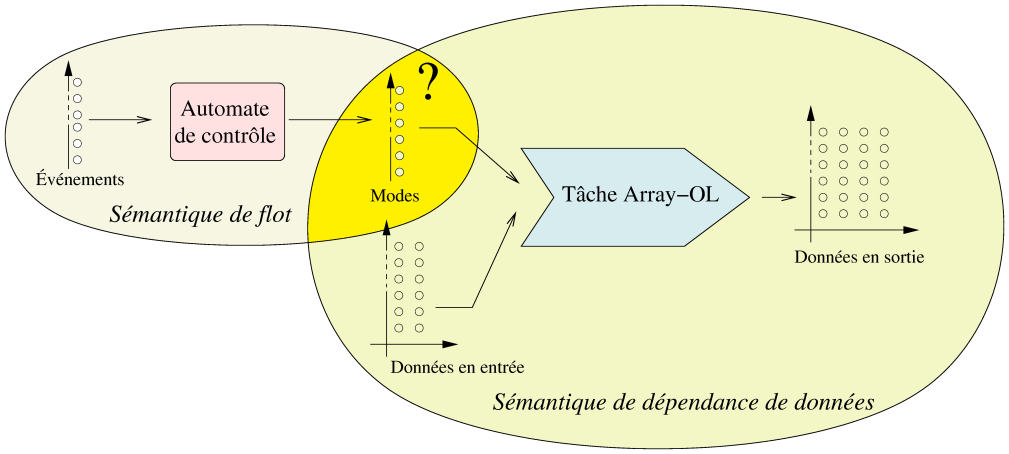


FIG. 7.4 – *Comment faire communiquer un automate de contrôle et une tâche* ARRAY-OL *?*

pond à l'association d'un comportement « *réactif* » aux applications de traitement parallèle pour permettre la description des relations entre les sorties et les entrées par l'intermédiaire de leurs combinaisons possibles dans le temps. Pour cette raison, nous allons nous inspirer des études réalisées autour des systèmes réactifs, présentées dans le chapitre 3, et en particulier celles des automates de modes pour introduire le contrôle et la notion de « *flot* » dans le modèle de spécification ARRAY-OL.

Dans la section 3.4 (page 48), nous avons discuté quelques travaux pour la séquentialisation du modèle ARRAY-OL en lui introduisant une notion de flot. Ces travaux peuvent se résumer en deux notions de base : la première considère que les données manipulées par les tâches ARRAY-OL sous forme de tableaux infinis sont généralement produites ou consommées par des composants réels tels que les capteurs ou les actionneurs. Ces composants fonctionnent à des rythmes bien déterminés, et permettent d'introduire la notion de flot dans le modèle. Cependant, définir les instants de changement de modes en fonction du rythme de fonctionnement des capteurs et des actionneurs peut être très limitatif pour l'application, et peut éventuellement conduire à des comportements différents pour la même application en cas d'un changement dans le rythme de fonctionnement de ces composants. La deuxième notion est relative à la création des flots de tableaux ou de motifs à partir des tableaux infinis. Dans cette approche, les tableaux en entrée pour une application peuvent être représentés par des flots de tableaux ou de motifs. Cependant, cette approche peut engendrer certains problèmes tels que la difficulté de la définition d'un ordre pour la production et la consommation des données, et les inter-blocages dûs aux tableaux infinis qui peuvent être produits et consommés de manières différentes. Ainsi, cette technique peut être considérée comme restrictive dans le sens où la création des flots ne prend pas en considération le résultat final ou la fonctionnalité de l'application.

Dans ce qui suit, nous allons présenter notre approche pour l'introduction du contrôle dans le modèle de spécification ARRAY-OL. Notre présentation sera limitée à l'étude des changements de modes de fonctionnement pour les tâches de calcul selon une approche réactive synchrone. Le changement des TILERs ne sera donc pas abordé dans ce travail.

7.3 Notion de degré de granularité pour le contrôle des applications parallèles

Pour l'introduction du contrôle dans les applications massivement parallèles décrites en ARRAY-OL, nous avons introduit le concept de *degré de granularité* pour ces application [LDBR05]. Ce concept permet de délimiter les différents *cycles d'exécution* ou les *instants* dans lesquels la prise en compte des événements de contrôle devient possible. Il permet également d'introduire une sémantique de flot dans la description des applications ARRAY-OL pour faciliter l'étude de leur comportement réactif en synchronisant les valeurs des données en entrée avec celles du contrôle.

De façon générale, le concept de degré de granularité permet la spécification des comportements réactifs pour les applications de traitement de données massivement parallèle décrite en ARRAY-OL. Comme nous l'avons discuté dans le chapitre 3, l'hypothèse synchrone est considérée comme une contribution importante et largement utilisée dans le domaine des systèmes réactifs. Nous allons donc nous inspirer de cette hypothèse pour décrire la réaction des applications de traitement de données parallèles aux événements de contrôle. L'idée est de synchroniser les instants d'arrivée des valeurs des données et celles du contrôle en supposant que ces valeurs seront prises en compte, par la tâche contrôlée, en même temps et selon une *horloge de base* commune. La description de l'application contrôlée contient un composant de contrôle dont le rôle est de produire un *tableau de modes* qui sera utilisé par cette application pour déterminer les modes de fonctionnement pour les différentes répétitions de la tâche contrôlée. Dans ce contexte, le tableau de modes n'est rien qu'une donnée en entrée comme toutes autres données de calcul. Le modèle proposé peut donc avoir une première impression « simpliste ». Cependant, cette approche impose un bon choix de degré de granularité pour pouvoir ramener les valeurs de calcul en même temps que celles du contrôle tout en respectant la fonctionnalité attendue de l'application étudiée. Ceci revient généralement à construire des flots de blocs de motifs en entrée et en sortie en créant des niveaux de hiérarchie supplémentaires et, par conséquent, modifier les matrices de pavage et d'ajustage de l'application parallèle décrite en ARRAY-OL.

7.3.1 Définition du concept de degré de granularité

Un degré de granularité (noté DG) définit un sous-ensemble du domaine de répétition correspondant à l'exécution, dans le même mode de fonctionnement, d'une tâche ARRAY-OL contrôlée. En d'autres termes, le degré de granularité est décrit par un bloc d'éléments représentant des points dans le domaine de répétition, et il peut être donc défini par sa forme $\overrightarrow{d_{DG}}$, par son origine o_{DG}, par une matrice de pavage P_{DG}, et par une matrice d'ajustage F_{DG} permettant de remplir les éléments de ce bloc (figure 7.5).

L'exécution du sous-ensemble du domaine de répétition défini par le degré de granularité nécessite en entrée un bloc de motifs d'entrée pour produire comme résultat un bloc de motifs de sortie. Il est donc nécessaire de définir les informations de TILER en entrée et en sortie permettant de déterminer les éléments de ces blocs de motifs.

Les informations de TILER pour le bloc de motifs en entrée regroupent la forme de ce motif, les matrices d'ajustage et de pavage, ainsi que l'origine du bloc référence. La forme du bloc de motifs en entrée, lié au domaine de répétition défini par le degré de granularité

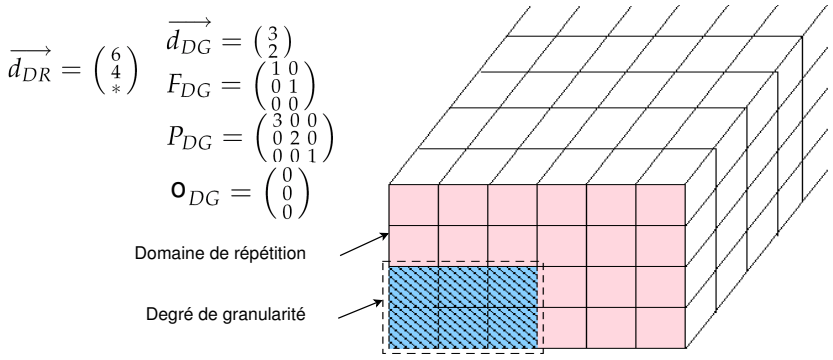


FIG. 7.5 – *Exemple de degré de granularité de taille 3 par 2 dans un espace de répétition sur 3 dimensions*

DG, est définie comme suit :

$$\overrightarrow{d_{inDG}} = \begin{pmatrix} \overrightarrow{d_{DG}} \\ \overrightarrow{d_{in}} \end{pmatrix} \tag{7.1}$$

où $\overrightarrow{d_{in}}$ représente la forme du motif en entrée pour la tâche contrôlée. La forme du bloc de motifs en entrée permet de déterminer le nombre d'éléments du tableau en entrée appartenant à ce bloc, et correspond au produit cartésien des dimensions du bloc de degré de granularité et de celles du motif en entrée de la tâche contrôlée. La sélection des éléments du tableau en entrée, utilisés pour remplir le bloc de motifs en entrée lié au degré de granularité DG, est réalisée en utilisant une matrice d'ajustage en entrée qui est définie comme suit :

$$F_{inDG} = \begin{pmatrix} P_{in} \times F_{DG} & F_{in} \end{pmatrix} \tag{7.2}$$

où P_{in} (resp. F_{in}) représente la matrice de pavage (resp. d'ajustage) en entrée pour la tâche contrôlée. La matrice d'ajustage F_{inDG} est relative à l'application linéaire d'un espace dans un autre pour permettre le passage de l'ensemble des éléments du domaine de répétition vers l'ensemble des éléments du tableau en entrée. Ainsi, le nombre de colonnes de cette matrice est égal au nombre de dimensions du bloc de motifs en entrée, tandis que le nombre de lignes est relatif au nombre de dimensions du tableau en entrée. La définition des différents blocs de motifs en entrée, pour permettre l'exécution de toutes les répétitions du domaine de répétition de la tâche contrôlée, est réalisée en utilisant une matrice de pavage en entrée qui est définie comme suit :

$$P_{inDG} = P_{in} \times P_{DG} \tag{7.3}$$

Cette matrice est relative au produit de la matrice de pavage qui permet de couvrir le tableau en entrée avec celle qui permet de couvrir l'espace de répétition selon le degré de granularité défini. Finalement, le point d'origine pour le bloc référence de motifs en entrée est défini comme suit :

$$o_{inDG} = o_{in} + P_{in} \times o_{DG} \tag{7.4}$$

où o_{in} représente le point d'origine du motif référence en entrée de la tâche contrôlée.

De façon similaire, nous pouvons déduire les informations de TILER liées au bloc de motifs en sortie relatif à l'exécution du sous-ensemble du domaine de répétition défini par le degré de granularité DG. Ces informations regroupent également la forme du bloc de motifs en sortie, les matrices d'ajustage et de pavage, et le point origine du bloc référence, et elles

sont définies comme suit :

$$\begin{cases} \overrightarrow{d_{outDG}} = \begin{pmatrix} \overrightarrow{d_{DG}} \\ \overrightarrow{d_{out}} \end{pmatrix} \\ F_{outDG} = \begin{pmatrix} P_{out} \times F_{DG} & F_{out} \end{pmatrix} \\ P_{outDG} = P_{out} \times P_{DG} \\ o_{outDG} = o_{out} + P_{out} \times o_{DG} \end{cases} \tag{7.5}$$

où $\overrightarrow{d_{out}}$ représente la forme du motif en sortie pour la tâche contrôlée, P_{out} (resp. F_{out}) représente la matrice de pavage (resp. d'ajustage) en sortie pour cette tâche, et o_{out} représente l'origine du motif référence en sortie de la tâche contrôlée.

Le concept de degré de granularité permet de créer un niveau de hiérarchie supplémentaire pour décrire la répétition autour du sous-ensemble du domaine de répétition défini par le degré de granularité, et auquel sera attribué une seule valeur de mode. Ce niveau de hiérarchie prend donc en entrée un bloc de motifs d'entrée et une seule valeur de mode, pour fournir comme résultat un bloc de motifs de sortie comme il est expliqué par l'exemple de la figure 7.6.

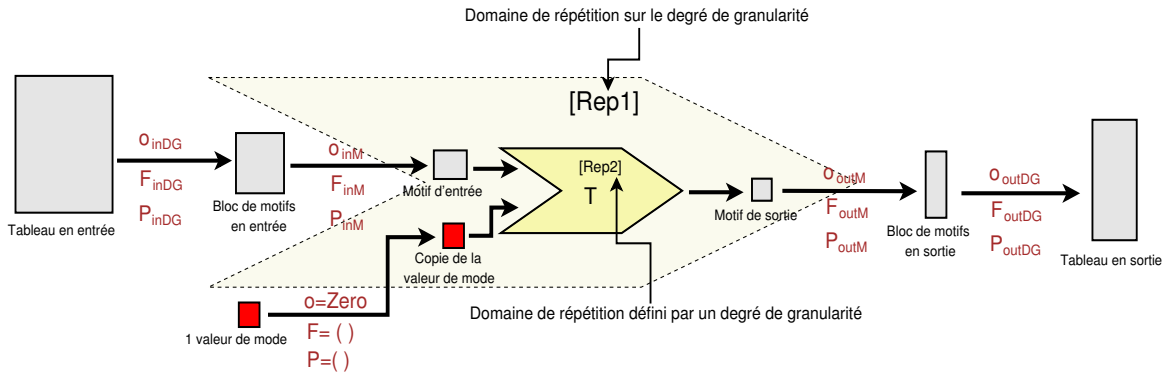


FIG. 7.6 – *Introduction d'un niveau de hiérarchie selon le concept de degré de granularité*

L'introduction du niveau hiérarchique lié au degré de granularité permet, d'une part de simplifier le modèle de l'application en créant des flots de blocs de motifs en entrée et en sortie, et d'autre part, de respecter notre méthodologie de séparation contrôle/données, présentée dans le chapitre 5, et de mieux exprimer la synchronisation des valeurs de mode avec celles des données. Cette représentation favorise la réutilisation des différentes parties de l'application, ainsi que l'étude et la maintenance de son comportement, notamment en ce qui concerne les processus de vérification formelle et la génération automatique du code.

Après avoir construit le bloc de motifs en entrée et celui en sortie liés à un degré de granularité, il est nécessaire de définir les informations de TILER permettant de déterminer les éléments du motif en entrée et de celui en sortie. Ces informations sont relatives à l'exécution d'une seule répétition du domaine de répétition défini par le degré de granularité. La forme du motif en entrée correspondant à cette répétition est définie comme suit :

$$\overrightarrow{d_{inM}} = \overrightarrow{d_{in}} \tag{7.6}$$

La forme de ce motif est la même que celle de la tâche contrôlée avant la construction du bloc de motifs lié au degré de granularité puisque, à ce stade, nous modélisons l'exécution

de chaque répétition de la tâche contrôlée. La matrice d'ajustage qui permet de sélectionner les éléments du motif en entrée à partir du bloc de motifs d'entrée est définie comme suit :

$$F_{inM} = \begin{pmatrix} Zero_{d_{DG}} \\ Id \end{pmatrix} \tag{7.7}$$

où $Zero_{d_{DG}}$ représente une matrice de zéros dont le nombre de lignes est égal au nombre de dimensions du degré de granularité, et Id : $gaspardControl.tex, v1.302006/12/1317 : 56 : 37labbaniExp$ représente la matrice identité dont le nombre de lignes est égal au nombre de dimensions du bloc de motifs en entrée moins celui du degré de granularité. Le nombre de colonnes pour la matrice d'ajustage F_{inM} est égal au nombre de dimensions du motif en entrée puisque chaque vecteur d'ajustage correspond à une dimension du motif, tandis que le nombre de lignes pour cette matrice est relatif au nombre de dimensions pour le bloc de motifs en entrée à partir duquel les éléments du motif sont sélectionnés. Pour déterminer les motifs en entrée manipulés par les répétitions du sous-ensemble du domaine de répétition défini par le degré de granularité, nous utilisons une matrice de pavage qui est définie comme suit :

$$P_{inM} = \begin{pmatrix} Id_{d_{DG}} \\ Zero \end{pmatrix} \tag{7.8}$$

où $Id_{d_{DG}}$ représente la matrice identité dont le nombre de lignes est égal au nombre de dimensions du degré de granularité, et $Zero$ représente une matrice de zéros dont le nombre de lignes est égal au nombre de dimensions du bloc de motifs en entrée moins celui du degré de granularité. Le nombre de colonnes pour la matrice de pavage P_{inM} est égal au nombre de dimensions de la matrice d'ajustage du degré de granularité F_{DG}, tandis que le nombre de lignes pour cette matrice est relatif au nombre de dimensions pour le bloc de motifs en entrée. Finalement, le point d'origine pour le motif référence en entrée est défini comme suit :

$$o_{inM} = Zero \tag{7.9}$$

De façon similaire, nous pouvons définir les informations de TILER correspondant au motif en sortie pour chaque répétition du sous-ensemble du domaine de répétition défini par la degré de granularité comme suit :

$$\begin{cases} \overrightarrow{d_{outM}} = \overrightarrow{d_{out}} \\ F_{outM} = \begin{pmatrix} Zero_{d_{DG}} \\ Id \end{pmatrix} \\ P_{outM} = \begin{pmatrix} Id_{d_{DG}} \\ Zero \end{pmatrix} \\ o_{outM} = Zero \end{cases} \tag{7.10}$$

Pour illustrer le concept de degré de granularité, nous proposons d'étudier un exemple simple d'une application ARRAY-OL représentée par la figure 7.7. Dans cet exemple, la tâche

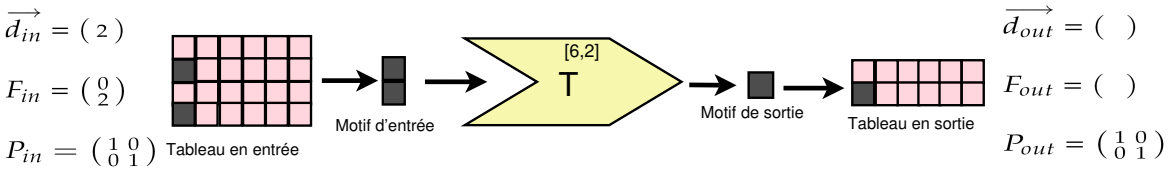


FIG. 7.7 – *Exemple simple d'une application* ARRAY-OL

T prend en entrée un tableau de taille 6×4 pour fournir en sortie un tableau de taille 6×2. Cette tâche est répétée 6×2 fois, où pour chaque répétition elle prend en entrée un motif de deux éléments pour fournir en sortie un motif d'un seul élément. Il est à noter que, dans cet exemple, nous ne prenons pas en considération le comportement réactif puisque l'objectif de cet exemple est uniquement de montrer l'utilisation du concept de degré de granularité.

Si nous considérons que le degré de granularité choisi pour l'application représentée par la figure 7.7 est de la forme $\overrightarrow{d_{DG}} = \binom{2}{2}$, avec $F_{DG} = \left(\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right)$, $P_{DG} = \binom{2}{0}$ et $o_{DG} = \binom{0}{0}$, alors l'application correspondante est représentée par la figure 7.8. Dans cette application,

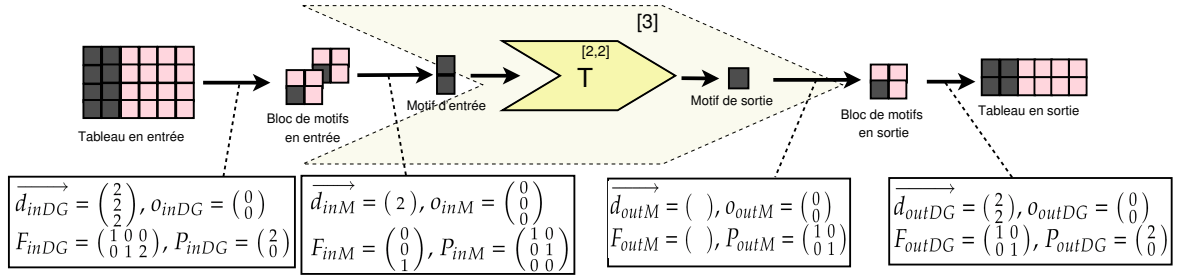


FIG. 7.8 – *Exemple1 d'utilisation du concept de degré de granularité*

nous avons créer un niveau de hiérarchie supplémentaire qui est répété 3 fois, où chaque répétition est relative au sous-ensemble du domaine de répétition défini par le degré de granularité.

La figure 7.9 représente un autre cas pour l'exemple de la figure 7.7, et dans lequel le degré de granularité choisi est de la forme $\overrightarrow{d_{DG}} = (3)$, avec $F_{DG} = \binom{2}{0}$, $P_{DG} = \left(\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right)$ et $o_{DG} = (0)$. Dans ce cas, le niveau de hiérarchie introduit est répété 2×2 fois, où chaque répétition correspond à l'exécution du sous-ensemble du domaine de répétition défini par le degré de granularité.

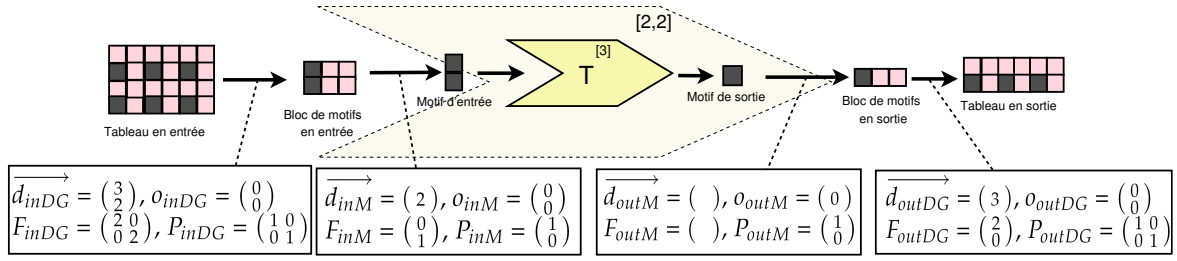


FIG. 7.9 – *Exemple2 d'utilisation du concept de degré de granularité*

Dans ce qui suit, nous allons utiliser le concept de degré de granularité pour la description des comportements de contrôle dans une application parallèle spécifiée en ARRAY-OL. Ce concept permet d'associer une sémantique de flot au comportement global de l'application étudiée en synchronisant l'arrivée des données en entrée d'une tâche contrôlée avec celle des valeurs de contrôle. L'idée est de déterminer les différents instants de prise en compte des valeurs de contrôle de telle sorte que les instances d'une tâche contrôlée, relatives au sous-ensemble du domaine de répétition d'un degré de granularité, soient exécutées dans le même mode de fonctionnement. En d'autres termes, le concept de degré de granularité permet de définir un bloc de motifs auquel sera attribué une seule valeur de mode. Les

motifs appartenant à ce bloc vont alors subir les mêmes traitements en s'exécutant dans le même mode de fonctionnement, et le changement de modes peut être pris en considération uniquement entre chaque deux blocs.

De façon générale, le domaine de répétition correspondant à un degré de granularité est choisi selon les fonctionnalités et le comportement attendus de l'application étudiée. L'utilisateur a donc la totale responsabilité et la liberté de choisir le degré de granularité pour son application. Cependant, il est important de noter que ce choix doit respecter la sémantique de base du modèle ARRAY-OL dans le sens où les blocs de motifs construits selon le concept de degré de granularité doivent permettre la génération de tous les éléments des tableaux en sortie de façon régulière et en respectant l'hypothèse d'assignation unique. En d'autres termes, le degré de granularité choisi doit couvrir tout le domaine de répétition de l'application étudiée sans aucun recouvrement.

Le changement de mode de fonctionnement pour une application de traitement de données parallèle exprime sa réaction à son contexte d'exécution qui peut dépendre d'un événement de son environnement externe, ou d'un résultat d'un calcul interne. Dans les deux cas, il est nécessaire de définir précisément les moments dans lesquels ces événements seront pris en compte pour permettre les changements de modes de fonctionnement. Le modèle que nous proposons doit, sur tous ses niveaux de hiérarchie, respecter la concurrence et le parallélisme, le déterminisme (même sorties pour les mêmes entrées) et la compositionnalité (pour la réutilisation) du modèle ARRAY-OL. Dans ce qui suit, nous allons illustrer l'utilisation du concept de degré de granularité et des composants contrôlés dans le modèle ARRAY-OL en fonction des types des événements responsables des changements de modes, et qui peuvent être externes ou internes.

7.3.2 Changement de modes en fonction de l'environnement externe

Dans cette section, nous considérons que l'application parallèle décrite en ARRAY-OL est sensible à son environnement externe qui peut être un utilisateur humain tel que l'appui sur un bouton, ou un processus physique tel que le changement de température. Ceci correspond aux différents changements de modes de fonctionnement de l'application selon les valeurs de contrôle fournies par son environnement. Pour décrire ce comportement, nous allons utiliser la notion de degré de granularité permettant la définition des différents moments de prise en compte des valeurs d'évènement de contrôle.

Pour mieux comprendre cette notion, nous considérons un exemple simple d'une application ARRAY-OL représentée par la figure 7.10. Dans cet exemple, le système prend en

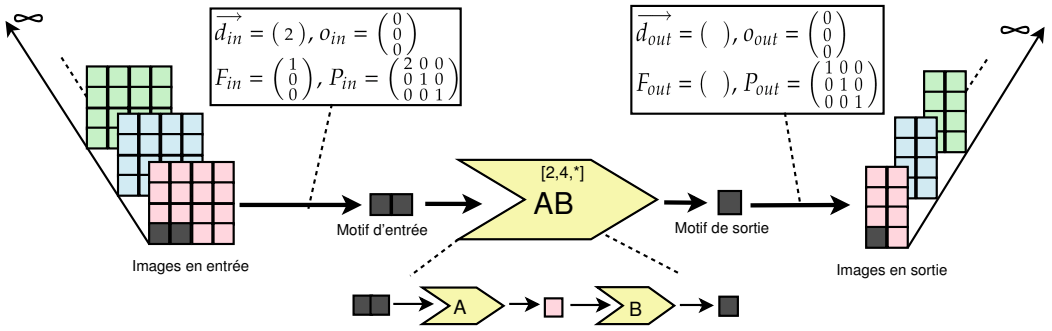


FIG. 7.10 – *Exemple simple d'un modèle* ARRAY-OL

entrée une infinité d'images de taille 4×4, et retourne en sortie une infinité d'images de taille 2×4. La tâche AB est répétée $2 \times 4 \times *$ fois[56]. Pour chaque répétition, cette tâche prend un motif de deux éléments en entrée, avec $\overrightarrow{d_{in}} = (2)$, $F_{in} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$, $P_{in} = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$ et $o_{in} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$, pour fournir un motif d'un seul élément en sortie, avec $\overrightarrow{d_{out}} = (\ )$, $F_{out} = (\ )$, $P_{out} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$ et $o_{out} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$.

Dans ce qui suit, nous allons introduire un automate de contrôle qui permet de changer les modes de fonctionnement pour la tâche élémentaire A. Cette dernière est remplacée par une tâche contrôlée CA qui, en fonction des valeurs fournies par l'automate de contrôle, peut choisir le mode de fonctionnement approprié. Nous rappelons que la sémantique de l'automate de contrôle est basée sur la notion de *flot*. Ce concept n'existe pas dans la description des modèles ARRAY-OL qui sont basés sur une sémantique de dépendance de données où le temps est banalisé. Dans ce qui suit, nous considérons que la structure de l'automate de contrôle représente un cas particulier d'une répétition dans ARRAY-OL avec une dépendance de données entre ces différentes répétitions. Une représentation plus appropriée de la structure de l'automate de contrôle tout en respectant la sémantique du modèle ARRAY-OL sera présentée dans le chapitre 8.

Ainsi, nous considérons que l'automate de contrôle produit un tableau infini de valeurs de modes qui seront utilisées pour déterminer les modes de fonctionnement de l'application contrôlée. Puisque ces valeurs de modes sont utilisées pour piloter l'exécution de l'application contrôlée, la définition du rythme de production de ces valeurs dépendra toujours du comportement attendu de l'application étudiée. Plusieurs situations possibles peuvent alors se présenter.

Un seul événement, un seul mode de fonctionnement, une seule image en sortie

Dans ce premier cas, nous considérons que le degré de granularité de l'application étudiée est de la forme $\overrightarrow{d_{DG}} = \begin{pmatrix} 2 \\ 4 \end{pmatrix}$, avec $F_{DG} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}$, $P_{DG} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ et $o_{DG} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$, ce qui correspond à l'exécution de 8 répétitions de la tâche contrôlée CA. Dans ce cas, pour chaque partie du domaine de répétition correspondant au degré de granularité choisi sera associée une et une seule valeur de mode comme il est montré par la figure 7.11. Cette valeur de mode sera utilisée par toutes les répétitions du sous-ensemble du domaine de répétition défini par le degré de granularité pour sélectionner le mode de fonctionnement à activer.

L'application du concept de degré de granularité permet de créer un niveau de hiérarchie supplémentaire qui, selon le degré de granularité défini, prend un bloc de motif en entrée pour fournir un bloc de motifs en sortie. Dans l'exemple de la figure 7.11, le niveau de hiérarchie introduit est répété une infinité de fois. Chaque répétition de ce niveau hiérarchique correspond à un sous-ensemble du domaine de répétition défini par le degré de granularité. L'exécution de ce bloc de degré de granularité fournit en sortie un bloc de motifs de sortie de la forme $\overrightarrow{d_{outDG}} = \begin{pmatrix} 2 \\ 4 \end{pmatrix}$, avec $F_{outDG} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}$, $P_{outDG} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ et $o_{outDG} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$ (équation 7.5). Ces informations de TILER sont relatives au bloc de motifs en sortie, et permettent de ranger les éléments de ce bloc dans le tableau de sortie correspondant. La production du bloc de motifs en sortie nécessite en entrée un bloc de motifs d'entrée de la forme $\overrightarrow{d_{inDG}} = \begin{pmatrix} 2 \\ 4 \\ 2 \end{pmatrix}$ (équa-

[56] Le symbole $*$ décrit l'infini.

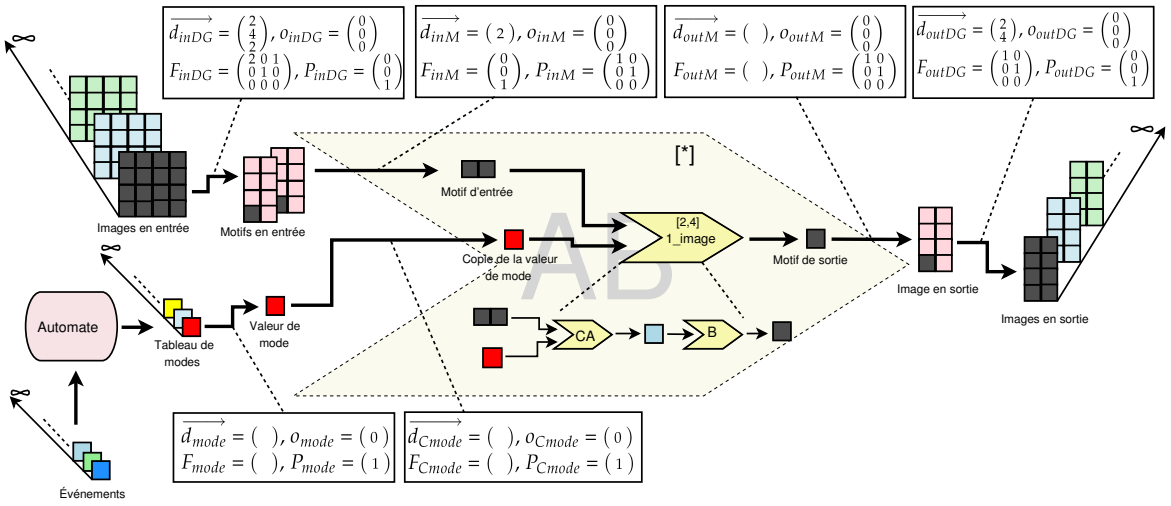


FIG. 7.11 – *Exemple d'introduction du contrôle pour toute l'image en sortie*

tion 7.1), avec $F_{inDG} = \begin{pmatrix} 2 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$ (équation 7.2), $P_{inDG} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ (équation 7.3) et $o_{inDG} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$ (équation 7.4). Ces informations de TILER permettent de sélectionner les éléments du tableau en entrée appartenant au bloc de motifs en entrée correspondant.

Après avoir construit le bloc de motifs en entrée et celui en sortie relatifs au degré de granularité choisi, il est possible de décrire les informations de TILER qui permettent de sélectionner (resp. de ranger) les éléments du motif en entrée (resp. en sortie) lié à une seule répétition de la tâche contrôlée. Pour l'exemple étudié, $\overrightarrow{d_{inM}} = (2)$ (équation 7.6), $F_{inM} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ (équation 7.7), $P_{inM} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}$ (équation 7.8), et $o_{inM} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$ (équation 7.9) représentent les informations de TILER en entrée pour l'exécution d'une seule répétition de la tâche contrôlée, tandis que $\overrightarrow{d_{outM}} = (\ )$, $F_{outM} = (\ )$, $P_{outM} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}$ et $o_{outM} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$ (équation 7.10) représentent les informations de TILER en sortie pour l'exécution d'une seule répétition de la tâche contrôlée.

Nous remarquons que le degré de granularité choisi pour cet exemple correspond à la génération d'une seule image en sortie pour l'application globale. Dans ce cas, l'ensemble des points d'une image en sortie résultent de l'exécution de la tâche contrôlée dans le même mode de fonctionnement. En d'autres termes, pour chaque répétition du sous-ensemble du domaine de répétition relatif au degré de granularité est associée une *copie* de la valeur de mode associée à ce degré de granularité. Cette valeur de mode est fournie par un automate de contrôle dont l'exécution correspond naturellement à un processus itératif prenant en entrée un flot d'événements pour produire en sortie un flot de valeurs de modes. Cette notion de flot va donc être propagée pour l'application globale puisque la tâche de calcul doit avoir toutes ses données en entrée pour pouvoir commencer son exécution, et la valeur de mode en fait partie. Dans ce contexte, le tableau infini d'images en entrée (resp. en sortie) sera considéré comme un flot de de bloc de motifs en entrée (resp. en sortie) correspondant au degré de granularité choisi.

Ainsi, dans l'exemple de la figure 7.11, et pour chaque valeur de mode de fonctionnement, la tâche 1_image est répétée 2×4 fois, et elle correspond à l'exécution des répétitions du sous-ensemble du domaine de répétition lié à au degré de granularité choisi. Cette tâche

répétée est englobée dans la tâche AB qui est répétée une infinité de fois, et elle représente le niveau de hiérarchie introduit pour la représentation de la répétition autour du degré granularité.

Un seul événement, un seul mode de fonctionnement, un seul point de l'image en sortie

Une autre situation possible consiste à autoriser les changements de mode de fonctionnement entre chaque répétition de la tâche contrôlée. Dans ce cas le degré de granularité est de la forme $\overrightarrow{d_{DG}} = (\ )$, avec $F_{DG} = (\ )$, $P_{DG} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$ et $o_{DG} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$. Ainsi, pour chaque répétition de la tâche contrôlée sera associée une seule valeur de mode comme il est représenté par la figure 7.12. .

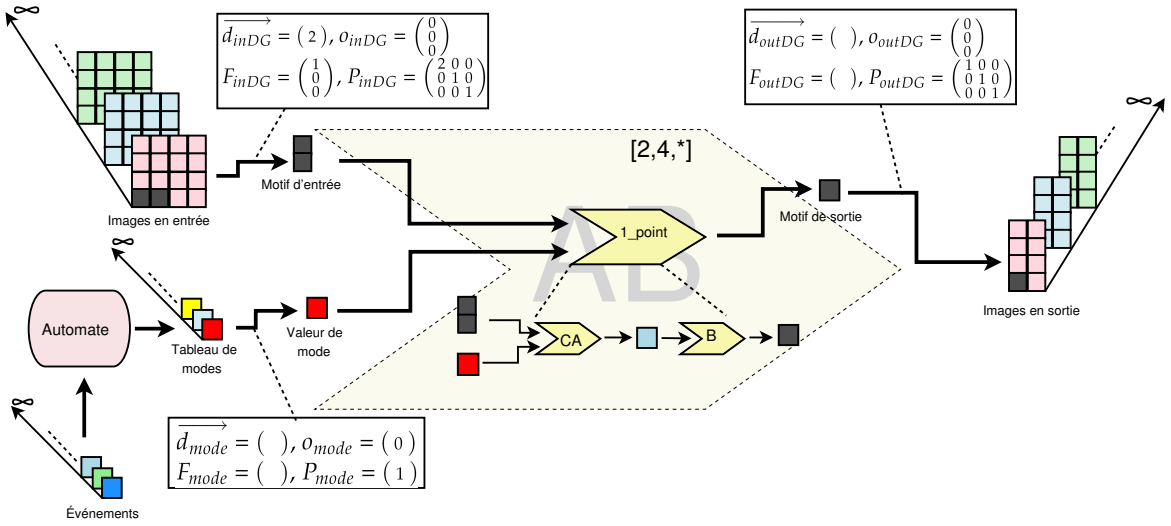


FIG. 7.12 – *Exemple d'introduction du contrôle pour un point de l'image en sortie*

Dans cet exemple, et en raisonnant de façon similaire que l'exemple précédent, le degré de granularité défini permet la génération d'un bloc de motifs en sortie de la forme $\overrightarrow{d_{outDG}} = (\ )$, avec $F_{outDG} = (\ )$, $P_{outDG} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$ et $o_{outDG} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$.La production de ce bloc de motifs en sortie nécessite un bloc de motifs en entrée de la forme $\overrightarrow{d_{inDG}} = (2)$, avec $F_{inDG} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$, $P_{inDG} = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$ et $o_{inDG} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$.

Nous remarquons que le degré de granularité choisi pour cet exemple correspond au calcul d'un seul point de l'image en sortie. Dans ce cas, il est possible d'effectuer des calculs différents pour chaque point de l'image en sortie en associant une valeur de mode de fonctionnement par une répétition de la tâche contrôlée. Ainsi, nous constatons que la notion de flot (ou de dépendance) dans l'automate de contrôle fait que le tableau infini d'images en entrée (resp. en sortie) se transforme en un flot de bloc de motifs en entrée (resp. en sortie) relatif au degré de granularité choisi (dans cet exemple, le bloc de motifs contient un seul motif). La tâche 1_point représentée dans la figure 7.12 est relatif à l'exécution du sous-ensemble du domaine de répétition défini par le degré de granularité, et qui représente une seule répétition de la tâche contrôlée. Cette tâche est englobée dans la tâche AB qui est répétée $2 \times 4 \times *$ fois pour la production de tous les points des images en sortie.

La principale différence entre les deux exemples présentés ci-dessus est dans le rapport NbVM / NbRep, où NbVM représente le nombre de valeurs de modes, et NbRep représente le nombre de répétitions relatives au degré de granularité choisi. Dans le premier exemple (figure 7.11), à chaque 8 répétitions de la tâche contrôlée, définies par le degré de granularité de la forme $\overrightarrow{d_{DG}} = \binom{2}{4}$, correspond une seule valeur de mode (NbVM / NbRep $= 1/8$), tandis que pour le deuxième exemple (figure 7.12), à chaque répétition de la tâche contrôlée, définie par le degré de granularité de la forme $\overrightarrow{d_{DG}} = (1)$, correspond une seule valeur de mode (NbVM / NbRep $= 1/1 = 1$). Dans ce cas, la génération d'une seule image en sortie nécessite la prise en compte d'un seul mode de fonctionnement pour le premier exemple, et 8 modes de fonctionnement pour le deuxième exemple comme il est montré par la figure 7.13. Ceci correspond également à des valeurs différentes pour les matrices de pavage

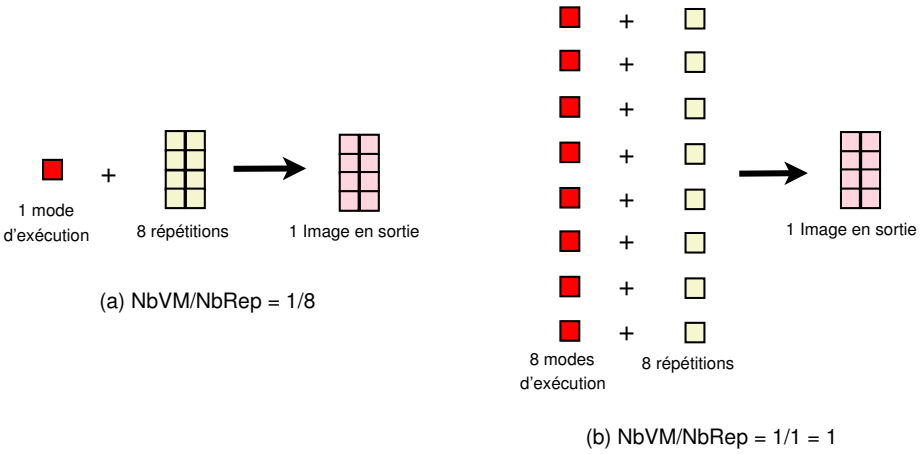


FIG. 7.13 – *Représentation du rapport* NbVM / NbRep

et d'ajustage puisque les données sont consommées et produites différemment.

Les deux exemples présentés montrent des cas simples de définition de degré de granularité dans une application pour permettre les changements de modes de fonctionnement selon des événements de contrôle. Il est également possible de considérer le changement de modes pour des cas plus complexes selon le degré de granularité choisi et selon les fonctionnalités attendues de l'application. Ce concept permet donc d'associer un comportement réactif aux applications décrites en ARRAY-OL, et par conséquent, rend possible l'étude des applications mixant des traitements de données parallèles et du contrôle tout en respectant la sémantique du modèle ARRAY-OL.

Un seul événement, un seul mode d'exécution, deux images en sortie

La définition de degré de granularité des tâches peut également dépendre d'un ensemble de critères extérieurs tels que l'implémentation ou l'association de l'application à une plate-forme particulière. Par exemple, si nous savons dès le départ que notre système est capable de calculer 16 répétitions en parallèle, et nous souhaitons que ces répétitions correspondent au calcul de deux images en sortie, nous pouvons considérer que le degré de granularité pour cette application est de la forme $\overrightarrow{d_{DG}} = \binom{2}{8}$, avec $F_{DG} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}$, $P_{DG} = \begin{pmatrix} 0 \\ 0 \\ 2 \end{pmatrix}$ et

$o_{DG} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$ comme il est montré par la figure 7.14. Dans cet exemple le degré de granu-

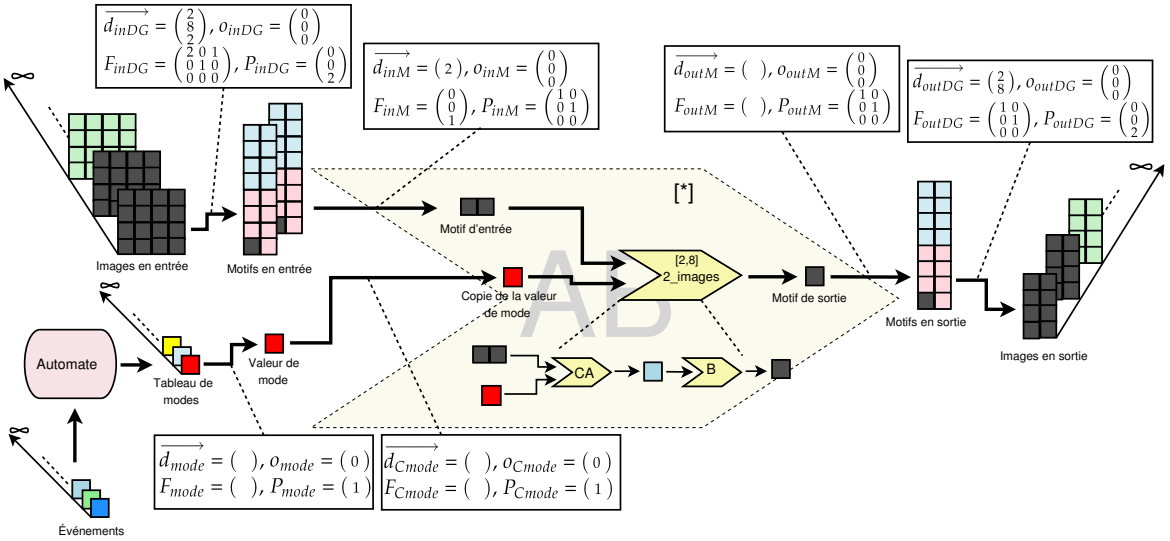


FIG. 7.14 – *Exemple d'exécution en parallèle de deux images avec la même valeur de mode*

larité défini un niveau de hiérarchie qui sera répété une infinité de fois et qui fournit en sortie un bloc de motifs de la forme $\overrightarrow{d_{outDG}} = \begin{pmatrix} 2 \\ 8 \end{pmatrix}$, avec $F_{outDG} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}$, $P_{outDG} = \begin{pmatrix} 0 \\ 0 \\ 2 \end{pmatrix}$ et $o_{outDG} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$.De façon similaire que les exemples précédents, la production de ce bloc nécessite un bloc de motifs en entrée de la forme $\overrightarrow{d_{inDG}} = \begin{pmatrix} 2 \\ 8 \\ 2 \end{pmatrix}$, avec $F_{inDG} = \begin{pmatrix} 2 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$, $P_{inDG} = \begin{pmatrix} 0 \\ 0 \\ 2 \end{pmatrix}$ et $o_{inDG} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$.

Il est également possible de déduire les informations de TILER permettant de récupérer (resp. de ranger) les éléments d'un motif en entrée (resp. en sortie) pour une seule répétition de la tâche contrôlée. Dans ce cas, $\overrightarrow{d_{inM}} = (2)$, $F_{inM} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$, $P_{inM} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}$ et $o_{inM} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$ représentent les informations de TILER en entrée d'une répétition de la tâche contrôlée, tandis que $\overrightarrow{d_{outM}} = (\ )\ F_{outM} = (\ )\ P_{outM} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}$ et $o_{outM} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$ représentent les informations de TILER en sortie pour cette répétition.

Dans cet exemple, une seule valeur de mode de fonctionnement est utilisée pour le calcul de deux images en sortie. La tâche 2_images, présentée dans la figure 7.14, est répétée 2×8 fois, et elle correspond à l'exécution des répétitions du sous-ensemble du domaine de répétition lié au degré de granularité choisi pour cette application. Cette tâche répétée est englobée dans la tâche AB qui est répétée une infinité de fois, et elle représente le niveau de hiérarchie introduit pour la représentation de la répétition autour du degré granularité.

7.3.3 Changement de modes en fonction d'un résultat interne

Dans les exemples présentés ci-dessus, nous avons supposé que les événements responsables du changement de modes viennent de l'environnement externe de l'application, et n'ont aucune relation avec son comportement ou sa fonctionnalité. Cependant, il est possible que les événements de contrôle dépendent d'un résultat de calcul interne. Pour illustrer ce

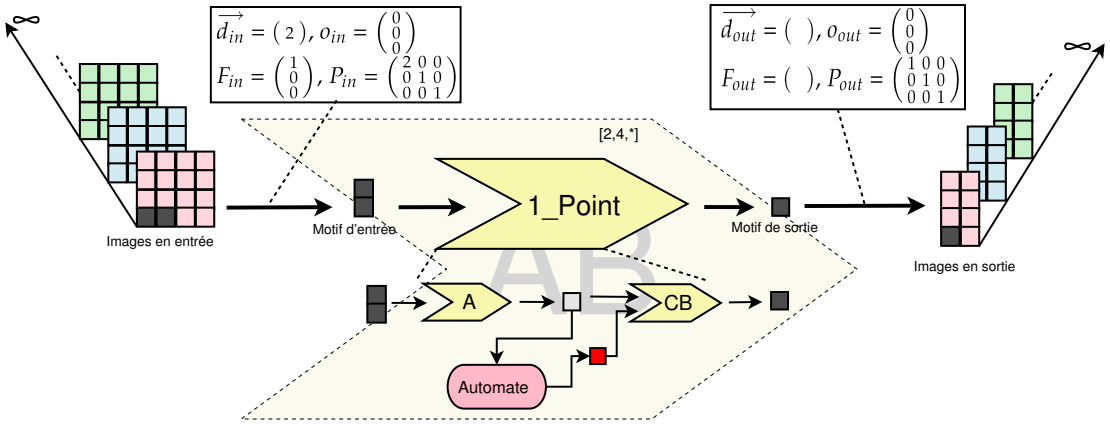


FIG. 7.15 – *Exemple d'un contrôle interne*

concept, nous considérons l'exemple de l'application représentée par la figure 7.10, et nous remplaçons la tâche élémentaire B par une tâche contrôlée CB qui peut changer son mode de fonctionnement selon le résultat fourni par la tâche élémentaire A comme il est montré par la figure 7.15. Dans cet exemple, le calcul de la tâche contrôlée CB dépend du résultat donné par la tâche élémentaire A. Pour ce faire, nous introduisons un automate de contrôle entre les deux tâches A et CB dont le rôle est de fournir à la tâche CB la valeur de mode d'exécution en fonction du résultat donné par A. Le degré de granularité choisi pour cet exemple est de la forme $\overrightarrow{d_{DG}} = (\ )$, avec $F_{DG} = (\ )$ et $P_{DG} = \begin{pmatrix}1&0&0\\0&1&0\\0&0&1\end{pmatrix}$. Ceci correspond au calcul d'un seul point de l'image en sortie auquel est associée une seule valeur de mode de fonctionnement comme dans le cas de l'exemple présenté par la figure 7.12.

Nous pouvons également imaginer des situations dans lesquelles le calcul de la tâche contrôlée CB dépend uniquement d'un certain points du résultat fourni par la tâche A. La figure 7.16 donne un exemple dans lequel, pour chaque image en sortie, le calcul de la tâche contrôlée CB dépend uniquement du point d'origine de l'image résultant de l'application répétée de la tâche A. Dans cet exemple, le degré de granularité de l'application est de la forme $\overrightarrow{d_{DG}} = \begin{pmatrix}2\\4\end{pmatrix}$, avec $F_{DG} = \begin{pmatrix}1&0\\0&1\\0&0\end{pmatrix}$ et $P_{DG} = \begin{pmatrix}0\\0\\1\end{pmatrix}$. Ceci correspond au calcul d'une seule image en sortie dans le même mode de fonctionnement comme dans le cas de l'exemple présenté par la figure 7.11.

Il est également possible de prendre en considération les deux types de contrôle, externe et interne, pour la même application comme il est montré par l'exemple de la figure 7.17. Dans cet exemple, le degré de granularité choisi est de la forme $\overrightarrow{d_{DG}} = \begin{pmatrix}2\\4\end{pmatrix}$, avec $F_{DG} = \begin{pmatrix}1&0\\0&1\\0&0\end{pmatrix}$ et $P_{DG} = \begin{pmatrix}0\\0\\1\end{pmatrix}$, et qui permet le calcul des points de chaque image en sortie dans le même mode de fonctionnement. Ce concept permet d'étudier des comportements plus complexes pour les systèmes mixant des traitements de données parallèles et du contrôle, et dont les différentes parties contrôlées de l'application peuvent être pilotées par des automates de contrôle différents. Ainsi, il est possible de choisir des degrés de granularité différents pour chaque partie contrôlée de l'application. Ce concept sera étudié en détail dans la section 7.4.

Les exemples présentés montrent la possibilité de décrire plusieurs scénarios introduisant le contrôle dans des applications de traitement parallèle en ayant la possibilité de dé-

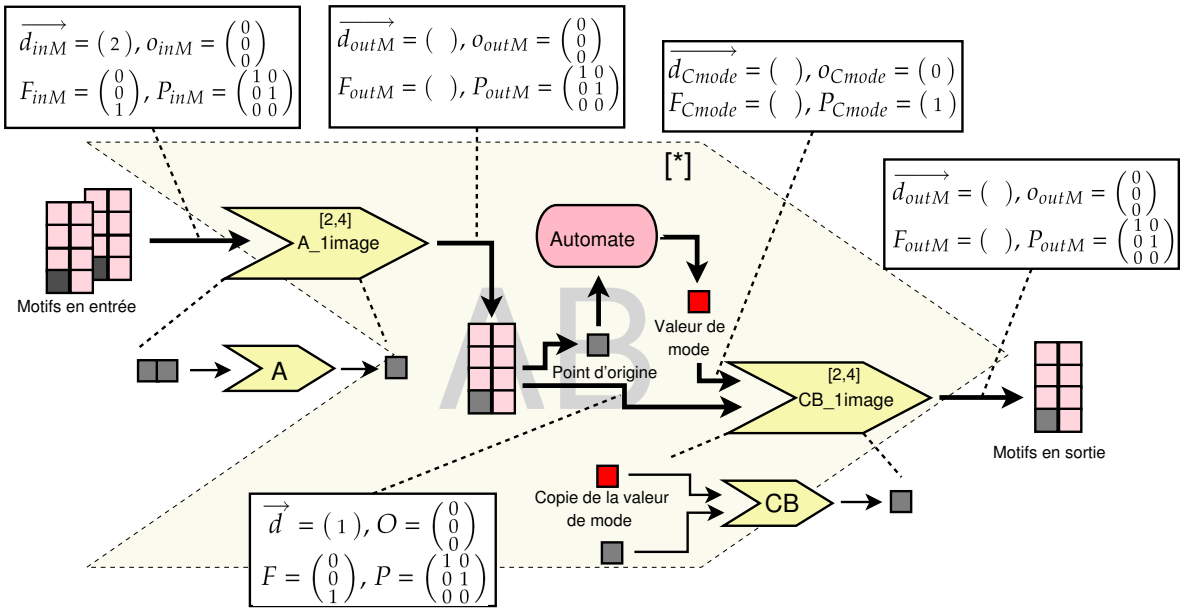


FIG. 7.16 – *Exemple d'un contrôle interne en fonction du point d'origine*

finir plusieurs types de degré de granularité pour ces applications. Cependant, nous rappelons que le degré de granularité choisi pour une application doit respecter l'hypothèse de l'*assignation unique* du modèle ARRAY-OL. Dans cette hypothèse, chaque point en sortie doit être calculé une et une seule fois sachant que les tableaux en sortie ne sont pas toriques. Ainsi, la répétition autour du degré de granularité choisi doit permettre le calcul de tous les points du tableau en sortie sans aucun recouvrement. La figure 7.18 donne des exemples de degré de granularité qui sont autorisés ou pas dans notre modèle. Ces exemples sont relatifs à une application dont le domaine de répétition est de la forme $\overrightarrow{d_{Rep}} = \begin{pmatrix}4\\4\end{pmatrix}$. Le premier exemple montre un cas où le degré de granularité choisi est de la forme $\overrightarrow{d_{DG}} = (4)$, avec $F_{DG} = \begin{pmatrix}0\\1\end{pmatrix}$ et $P_{DG} = \begin{pmatrix}1\\0\end{pmatrix}$. Ce cas est accepté puisqu'il couvre tout le domaine de répétition sans aucun recouvrement. De même pour le deuxième exemple où le degré de granularité choisi est de la forme $\overrightarrow{d_{DG}} = (2)$, avec $F_{DG} = \begin{pmatrix}1\\0\end{pmatrix}$ et $P_{DG} = \begin{pmatrix}2&0\\0&1\end{pmatrix}$. Le troisième exemple représente un cas dans lequel le degré de granularité choisi est de la forme $\overrightarrow{d_{DG}} = (3)$, avec $F_{DG} = \begin{pmatrix}0\\1\end{pmatrix}$ et $P_{DG} = \begin{pmatrix}1&0\\0&3\end{pmatrix}$. Ce cas n'est pas accepté puisqu'il ne permet pas de couvrir tout le domaine de répétition sans recouvrement.

De façon générale, le choix du degré de granularité pour une application dépend fortement de sa fonctionnalité et de son comportement attendus. Notre approche est par conséquent une approche *orientée fonctionnelle* dans laquelle plusieurs applications, mixant des traitement de données parallèles à la ARRAY-OL et du contrôle, peuvent être spécifiées en choisissant le « bon » degré de granularité.

En conclusion, l'introduction de la notion de degré de granularité dans un modèle de traitement parallèle permet la définition du top d'horloge dans lequel la prise en compte des différents changements de modes pour une application parallèle devient possible. Notre approche est basée sur la technologie réactive synchrone dans laquelle les tableaux de modes sont considérés comme des simples données en entrée, et l'application consomme et produit ses données en même temps.

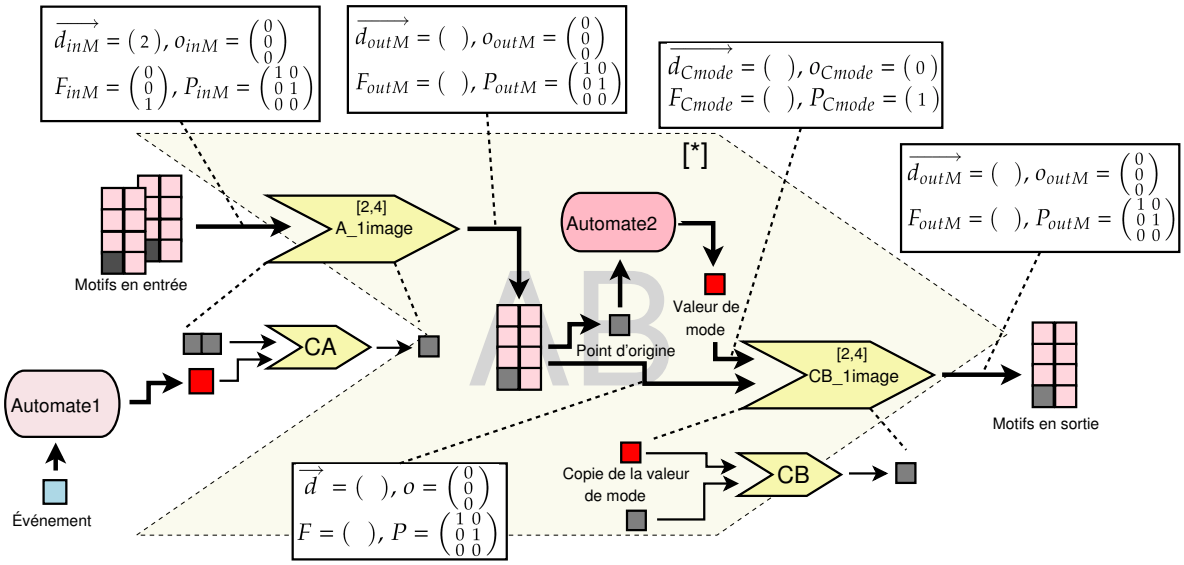


FIG. 7.17 – *Exemple d'une application avec du contrôle externe et interne*

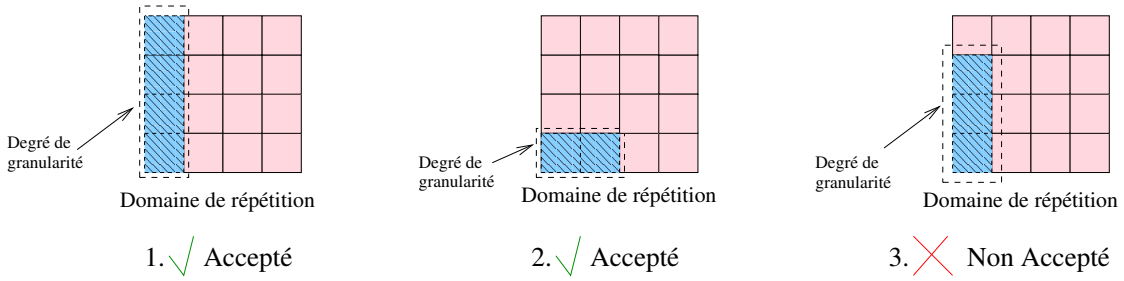


FIG. 7.18 – *Exemples de degrés de granularité respectant ou pas l'hypothèse d'assignation unique*

Il est à noter que l'hypothèse synchrone et la définition du concept de degré de granularité peuvent imposer un ordre partiel (pas de séquentialisation) sur l'exécution des différentes tâches parallèles en leur introduisant une notion de flot de données. Cette notion de flot n'empêche pas que notre approche respecte bien le parallélisme et la concurrence du modèle ARRAY-OL pour les tâches non séquentialisées, qu'elle soit déterministe en donnant les mêmes résultats pour les mêmes entrées, compositionnelle pour favoriser le processus de réutilisation, et suit notre méthodologie de séparation contrôle/données introduite dans le chapitre 5.

7.4 Vers un multi-degrés de granularité pour les applications parallèles

Dans la section précédente, nous avons introduit la notion de degrés de granularité pour pouvoir délimiter les différents moments de prise en compte des événements de contrôle dans une application parallèle décrite en ARRAY-OL. Dans les exemples présentés, nous avons étudié la définition d'un seul degré de granularité fixé dés le départ et utilisé pour la description du comportement global de toute l'application. Cependant, il est également

possible d'imaginer d'autres scénarios plus complexes dans lesquels plusieurs composants dans l'application parallèle peuvent être contrôlés par des automates de contrôle différents, et selon des degrés de granularité variés. Dans ce cas, l'effet des événements de contrôle, ainsi que les moments de leur prise en compte peuvent différer d'un composant à un autre. Pour pouvoir étudier ce type de comportement, nous allons étendre notre concept de degré de granularité en définissant la notion de *multi-degrés de granularité*. Cette notion permet de prendre en considération plusieurs niveaux de granularité dans une application parallèle en interprétant différemment les événements de contrôle à des instants variés. Dans ce contexte, la question à poser est la suivante : *comment définir le comportement global des applications à multi-degrés de granularité ?*

Le concept de multi-degrés de granularité suppose que les différentes parties d'une même application peuvent réagir différemment aux événements de contrôle dans le sens où elles changent leur modes de fonctionnement selon des degrés de granularité différents, et à des échelles de temps variées. Il est donc important de pouvoir synchroniser les différents automates de contrôle pour fournir toutes les valeurs de modes utilisées pour le calcul du même point dans le tableau en sortie.

Dans ce qui suit, et pour illustrer le concept de multi-degrés de granularité, nous allons l'étudier selon deux points de vue différents en fonction si les degrés de granularité choisis pour l'application sont *complètement imbriqués* ou pas.

Définition du multi-degrés de granularité complètement imbriqués

Soit $D = \{ DG_1, DG_2, \ldots, DG_n\}$ un ensemble de degrés de granularité choisis pour la prise en compte des événements de contrôle dans différentes parties d'une même application. Nous disons que les éléments de l'ensemble D sont complètement imbriqués si et seulement si :

$$\forall i, j, i \neq j, D_i \subseteq D_j \; ou \; D_j \subseteq D_i$$

où D_i représente les éléments du domine de répétition défini par le degré de granularité DG_i.

Le concept de degrés de granularité complètement imbriqués permet de créer un ordre partiel d'inclusion pour les différents éléments appartenant à l'ensemble D. Par exemple, pour un domaine de répétition sur deux dimensions de taille 4×4, si nous considérons trois degrés de granularité pour l'application avec $\overrightarrow{d_{DG_1}} = (\ )$, $F_{DG_1} = (\ )$, $P_{DG_1} = \left(\begin{smallmatrix}1 & 0\\ 0 & 1\end{smallmatrix}\right)$, $o_{DG1} = \left(\begin{smallmatrix}0\\ 0\end{smallmatrix}\right)$; $\overrightarrow{d_{DG_2}} = \left(\begin{smallmatrix}2\\ 4\end{smallmatrix}\right)$, $F_{DG_2} = \left(\begin{smallmatrix}1 & 0\\ 0 & 1\end{smallmatrix}\right)$, $P_{DG_2} = \left(\begin{smallmatrix}2\\ 0\end{smallmatrix}\right)$, $o_{DG2} = \left(\begin{smallmatrix}0\\ 0\end{smallmatrix}\right)$; et $\overrightarrow{d_{DG_3}} = (4)$, $F_{DG_3} = \left(\begin{smallmatrix}0\\ 1\end{smallmatrix}\right)$, $P_{DG_3} = \left(\begin{smallmatrix}1\\ 0\end{smallmatrix}\right)$, $o_{DG2} = \left(\begin{smallmatrix}0\\ 0\end{smallmatrix}\right)$, alors DG_1, DG_2 et DG_3 sont complètement imbriqués puisque $DG_1 \subseteq DG_3 \subseteq DG_2$.

7.4.1 Multi-degrés de granularité complètement imbriqués

Dans le cas où les éléments appartenant à l'ensemble de degrés de granularité pour une application sont complètement imbriqués, le degré de granularité qui sera utilisé pour la description de l'application globale est relatif au degré de granularité qui *englobe* le reste de l'ensemble.

Pour illustrer ce concept, nous proposons d'étudier l'exemple de l'application parallèle présentée par la figure 7.10 (page 140). Dans cet exemple, nous allons remplacer les tâches élémentaires A et B par les tâches contrôlées CA et CB respectivement. Les deux tâches

contrôlées changent leur mode de fonctionnement selon la valeur de l'événement de contrôle Event1. Ainsi, nous supposons que le changement de mode pour la tâche CA a un degré de granularité de la forme $\overrightarrow{d_{DG}} = \binom{2}{4}$, avec $F_{DG} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}$, $P_{DG} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ et $o_{DG} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$(relatif au calcul d'une image en sortie), tandis que le degré de granularité pour la tâche CB est de la forme $\overrightarrow{d_{DG}} = (\ )$, avec $F_{DG} = (\ )$, $P_{DG} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$ et $o_{DG} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$ (relatif au calcul d'un seul point de l'image en sortie) comme il est montré par la figure 7.19. Dans cet exemple,

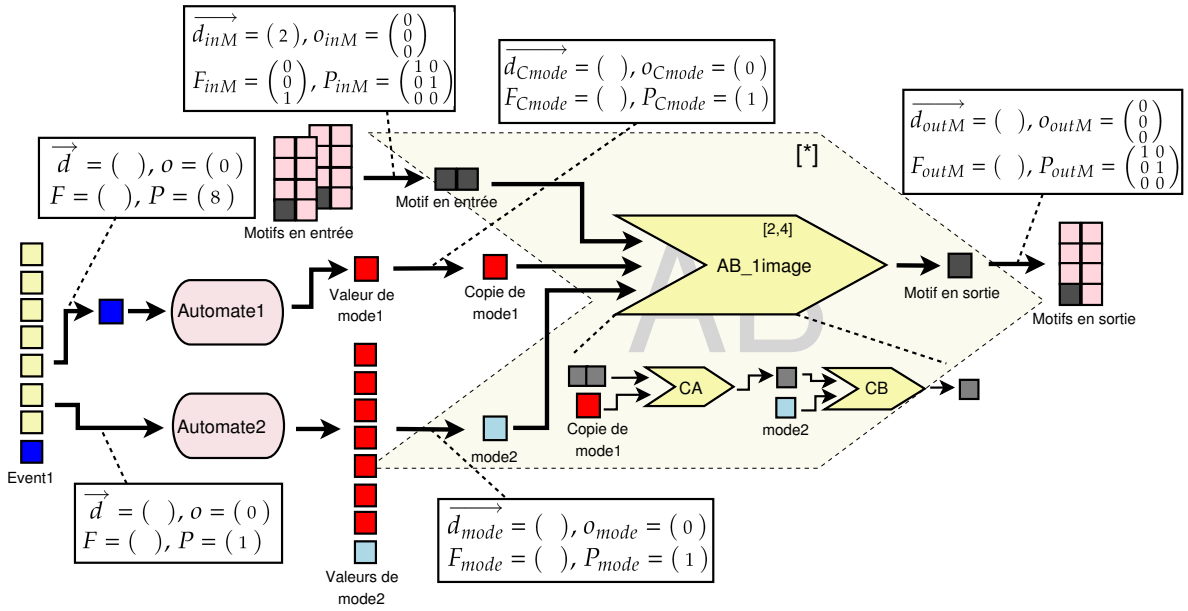


FIG. 7.19 – *Multi-degrés de granularité liés au même événement de contrôle avec perte d'événements*

les éléments des sous-ensemble du domaine de répétition définis par les deux degrés de granularité choisis pour l'application sont complètement imbriqués. Dans ce cas, le degré de granularité utilisé pour l'application globale est relatif à celui de la plus grande dimension (la partie du domaine de répétition englobante), et il est de la forme $\overrightarrow{d_{DG}} = \binom{2}{4}$, avec $F_{DG} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}$, $P_{DG} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ et $o_{DG} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$. Nous remarquons que le degré de granularité global permet le calcul d'une seule image en sortie, où les différents points de cette image résultent de l'exécution de la tâche CA dans le même mode de fonctionnement et de la tâche CB dans différents modes de fonctionnement.

Pour assurer le comportement attendu de l'application, il est nécessaire de définir les rythmes de fonctionnement des différents automates de contrôle et leurs points de synchronisation. Dans l'exemple de la figure 7.19, le comportement attendu de l'application impose d'avoir une seule valeur de mode mode1 et 8 valeurs de mode mode2 pour le calcul des répétitions du degré de granularité relatif à l'application globale. Ainsi, nous avons supposé dans cet exemple que pour chaque 8 événements de contrôle pris en compte par l'automate `Automate2`, l'automate `Automate1` prend en considération un seul événement de contrôle (le premier événements de chaque bloc de 8 événements), et par conséquent, l'évolution de l'automate `Automate1` rate ou ne « voit » pas les événements non pris en compte (pour chaque exécution, 7/8 valeurs d'événement sont perdues). En d'autre termes, les deux automates de contrôle évoluent à des rythmes différents et doivent être synchronisés puisque le calcul

d'une image en sortie de taille 2×4 nécessite 1 seule valeur de mode fournie par l'automate `Automate1` et 8 valeurs de mode fournies par l'automate `Automate2` en fonction du même événement de contrôle Event1.

Cet exemple montre que le concept de multi-degrés de granularité peut imposer des décalages dans la prise en compte des différents événements de contrôle. Dans ce cas, l'horloge liée à l'automate de contrôle `Automate2` est plus fine que celle liée à l'automate `Automate1`. Ainsi, à chaque transition dans l'automate `Automate1` correspondent 8 transitions pour l'automate `Automate2` comme il est expliqué par la figure 7.20.

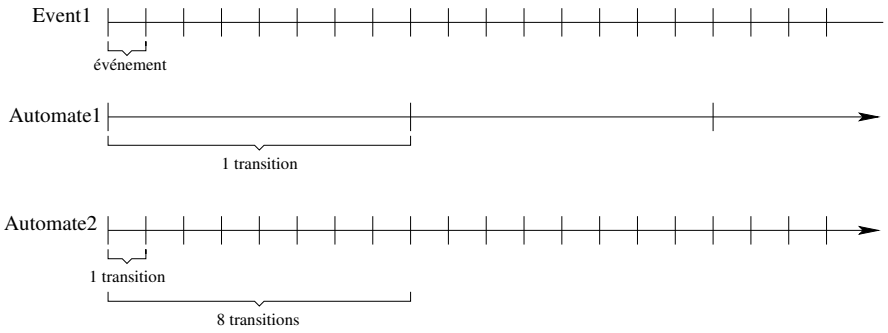


FIG. 7.20 – *Évolution à des rythmes différents pour les deux automates* `Automate1` *et* `Automate2`

L'expression de ce comportement dans le modèle ARRAY-OL est réalisée en associant à la spécification de l'application les informations appropriées des TILERs pour la prise en compte des événements de contrôle. Dans l'exemple présenté, et comme nous considérons que l'automate de contrôle est un composant répété particulier en ARRAY-OL, la différence dans le rythme de fonctionnement des deux automates de contrôle est exprimée par deux matrice de pavage différente : $P = (8)$ pour la répétition en entrée de l'automate `Automate1`, et $P = (1)$ pour la répétition en entrée de l'automate `Automate2` (figure 7.19).

Cependant, si nous remplaçons la matrice de pavage de la répétition en entrée de l'automate `Automate1` par $P' = (1)$, le comportement de l'application va être complètement différent puisque l'automate de contrôle `Automate1` évolue à un rythme différent que celui de l'application représentée par la figure 7.19, et par conséquent, cet automate peut donner des valeurs de mode complètement différentes puisque le résultat donné par un automate après 8 transitions est différent de celui donné par le même automate après une seule transition (l'automate ne doit pas être nécessairement dans le même état). La figure 7.21 représente un exemple dans lequel les deux automates de contrôle `Automate1` et `Automate2` fonctionnent au même rythme. Dans cet exemple, et pour assurer l'exécution du même mode de fonctionnement de la tâche CA pour tous les points de l'image en sortie et respecter les degrés de granularité de l'application, une seule valeur de mode sera utilisée parmi les 8 valeurs calculées (dans l'exemple nous avons choisi la première valeur de mode). Dans ce contexte, l'automate de contrôle `Automate1` ne rate aucun événement de contrôle à la différence de celui représenté dans l'exemple de la figure 7.19.

7.4.2 Multi-degrés de granularité non complètement imbriqués

Il est possible d'imaginer d'autres situations plus complexes où les degrés de granularité choisis pour une application contrôlée ne sont pas complètement imbriqués.

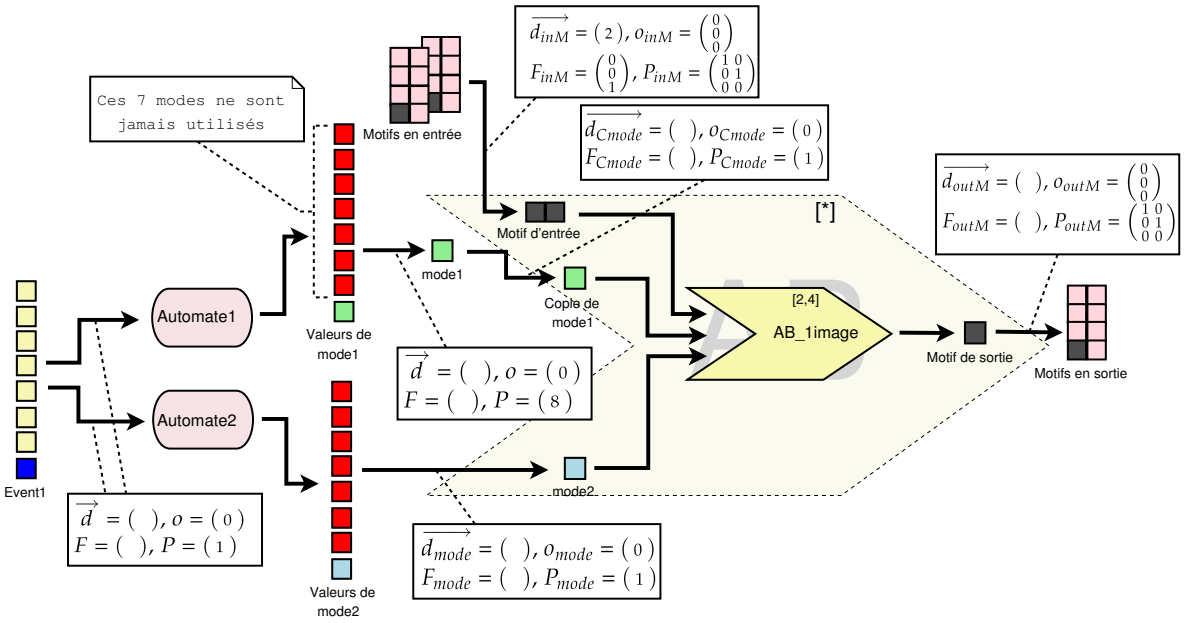


FIG. 7.21 – *Multi-degrés de granularité liés au même événement de contrôle sans perte d'événements*

Pour illustrer ce concept, nous proposons d'étudier l'exemple d'une application similaire[57] à celle présentée par l'exemple de la figure 7.10 (page 140), et qui prend en entrée une infinité d'image de taille 4×3 pour fournir en sortie une infinité d'image de taille 2×3. Pour cet exemple, nous remplaçons la tâche élémentaire A par une tâche contrôlée CA avec un degré de granularité de la forme $\overrightarrow{d_{DG}} = (2)$, avec $F_{DG} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$, $P_{DG} = \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}$ et $o_{DG} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$, et nous remplaçons la tâche élémentaire B par une tâche contrôlée CB avec un degré de granularité de la forme $\overrightarrow{d_{DG}} = (3)$, avec $F_{DG} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$, $P_{DG} = \begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{pmatrix}$ et $o_{DG} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$. Le degré de granularité pour l'application globale est donc de la forme $\overrightarrow{d_{DG}} = \begin{pmatrix} 2 \\ 3 \end{pmatrix}$, avec $F_{DG} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}$, $P_{DG} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ et $o_{DG} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$. La représentation de cette application est donnée par la figure 7.22. Dans cet exemple, nous avons introduit deux niveaux de hiérarchie pour permettre l'exécution des répétitions du sous-ensemble du domaine de répétition défini par le degré de granularité pour l'application globale. Le premier niveau de hiérarchie correspond à la tâche AB'_2points. Cette tâche est répétée 2 fois pour permettre l'exécution des répétitions relatives à la première dimension du sous-ensemble du domaine de répétition défini par le degré de granularité de l'application globale. Le deuxième niveau de hiérarchie est relatif à la tâche AB'_1image. Cette tâche est répétée 3 fois pour permettre l'exécution des répétitions relatives à la deuxième dimension du sous-ensemble du domaine de répétition défini par le degré de granularité de l'application globale. Ainsi, pour respecter le comportement de l'application globale, le degré de granularité défini pour cette application nécessite 3 valeurs de modes pour l'exécution de la tâche CA' et 2 valeurs de modes pour l'exécution de la tâche CB'. Ces valeurs de modes sont respectivement calculées par les deux automates

[57] La seule raison d'avoir changer la taille des images en entrée et en sortie par rapport aux exemples étudiés précédemment est d'avoir un exemple qui permet de mieux illustrer le concept de multi-degrés de granularité en terme de perte d'événements de contrôle.

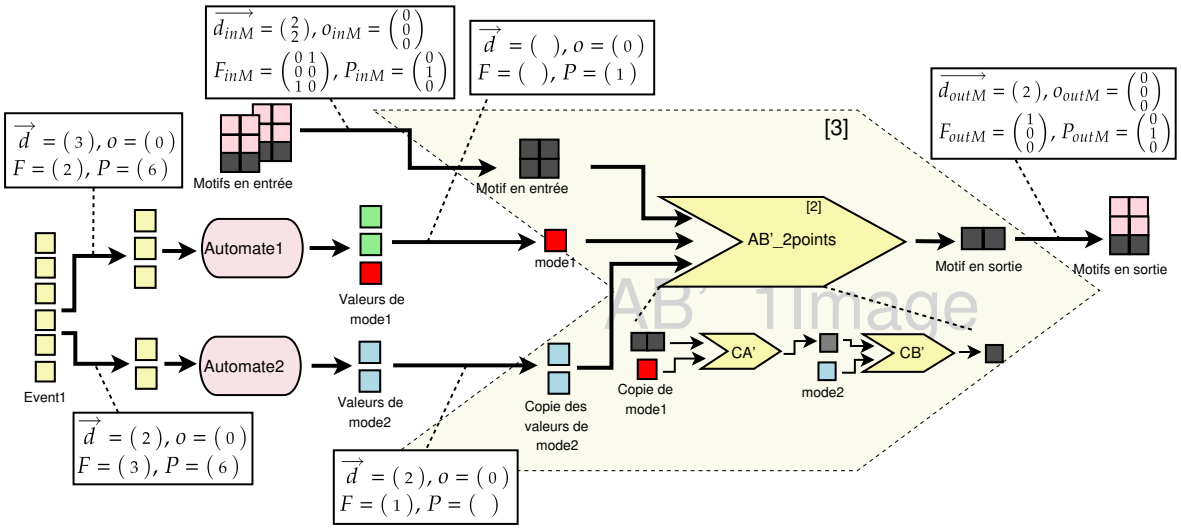


FIG. 7.22 – *Multi-degrés de granularité non complètement imbriqués*

de contrôle `Automate1` et `Automate2` en fonction des valeurs de l'événement Event1. Pour assurer le bon fonctionnement de l'application globale, il est nécessaire de synchroniser les deux automates de contrôle pour fournir les valeurs de mode nécessaires pour l'exécution des répétitions définies par le degré de granularité global de l'application. Pour ce faire, une solution possible consiste à prendre en considération 6 valeurs d'événements Event1 pour chaque répétition de la tâche AB'_1Image[58]. Ces événements sont utilisés selon des rythmes différents par les deux automates de contrôle comme il est expliqué par la figure 7.23. Ce

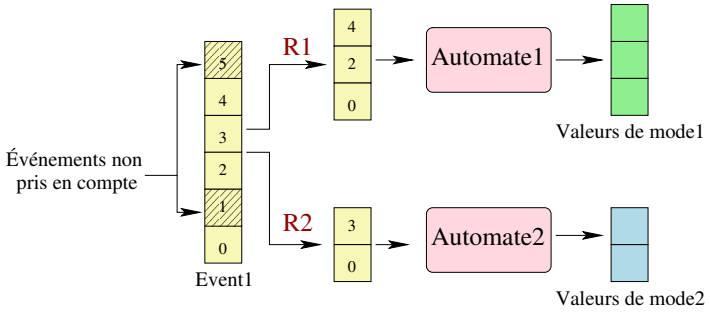


FIG. 7.23 – *Consommation des événements par les automates de contrôle pour l'exemple de la figure 7.22*

comportement est réalisé par l'expression des informations de TILER pour la consommation des valeurs d'événements par les différentes répétitions des automates de contrôle. Dans le cas de notre exemple, l'automate `Automate1` prend en considération un événement sur deux avec comme informations de TILER pour la répétition R1 : $o = (0)$, $P = (6)$ et $F = (2)$, tandis que l'automate `Automate2` prend en considération un événement sur trois avec comme informations de TILER pour la répétition R2 : $o = (0)$, $P = (6)$ et $F = (3)$.

Dans cet exemple, il est possible que dans la progression des deux automates de contrôle certaines valeurs d'événement ne sont pas prises en compte. Ceci est dû à la synchronisation

[58] La tâche AB'_1Image est définie à l'intérieur de la tâche AB' qui est répétée une infinité de fois pour assurer le calcul de toutes les images en sortie.

des deux automates pour assurer la fonctionnalité de l'application globale qui impose, d'une part, l'exécution du même mode de la tâche CA' pour tous les points d'une même ligne de l'image en sortie, et d'autre part, l'exécution du même mode de la tâche CB' pour une colonne de l'image en sortie.

La définition des différents niveaux de hiérarchie, des blocs de motifs en entrée et en sortie correspondant, ainsi que les valeurs de modes utilisées pour l'exécution des répétitions du sous-ensemble du domaine de répétition, défini par le degré de granularité de l'application globale, est un problème complexe qui nécessite plusieurs traitements de fusion et des processus de calcul d'horloge. Les opérations de fusion [Sou01, Dum05] doivent permettre la construction des blocs de motifs en entrée et en sortie correspondant à chaque niveau de hiérarchie, ainsi que la déduction des informations de TILER permettant de remplir ces blocs de motifs. Les processus de calcul d'horloge doivent permettre la synchronisation des rythmes de fonctionnement des différents automates de contrôle pour avoir les valeurs de modes appropriées pour le degré de granularité de l'application globale. L'étude de la prise en compte des processus de calcul d'horloge dans ARRAY-OL, ainsi que leur utilisation avec des processus de fusion représentent des problèmes de recherche importants qui sont récemment lancés au sein de l'équipe West[59] et sortent du cadre de notre travail.

Les exemples présentés ci-dessus supposent que les changements de mode pour les deux tâches CA et CB dépendent du même événement de contrôle Event1. Cependant, il est également possible que ces changements de mode dépendent des événements différents. Ces derniers sont généralement interprétés différemment pour chaque partie de l'application, et par conséquent, sont traités par des automates de contrôle différents pouvant fonctionner à des rythmes variés. Pour illustrer ce concept, nous supposons que les tâches CA et CB peuvent changer leur mode de fonctionnement selon deux événements différents Event1 et Event2. Le degré de granularité choisi pour le changement de mode de la tâche CA est de la forme $\overrightarrow{d_{DG}} = \binom{2}{4}$, avec $F_{DG} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}$, $P_{DG} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ et $o_{DG} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$ (relatif au calcul d'une image en sortie), tandis que le degré de granularité pour la tâche CB est de la forme $\overrightarrow{d_{DG}} = (1)$, avec $F_{DG} = (\ )$, $P_{DG} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$ et $o_{DG} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$ (relatif au calcul d'un seul point de l'image en sortie). Le modèle relatif à cette spécification est représenté par la figure 7.24. La seule différence entre le modèle de cet exemple et celui représenté par la figure 7.19 est que les événements de contrôle sont différents. Ainsi, à chaque valeur d'événement Event1 va correspondre 8 valeurs d'événements Event2. Dans ce cas, en fonction du rythme d'arrivée des événements Event1 et Event2, les deux automates de contrôle doivent être synchronisés, où l'automate `Automate1` peut éventuellement rater certains événements de contrôle (tout dépend du comportement attendu et des TILERs définis pour chaque répétition).

L'ensemble des exemples étudiés représentent des cas particuliers pour la spécification du concept de multi-degrés de granularité dans une application parallèle décrite en ARRAY-OL. Dans ces exemples, nous avons montré qu'il est possible de prendre en considération plusieurs degré de granularité pour la même application en définissant un degré de granularité pour l'application globale. Il est important de noter que ce concept de multi-degrés de granularité est valide uniquement si le degré de granularité défini pour l'application globale permet de calculer tous les points des tableaux en sortie et respecte l'hypothèse d'assignation unique du modèle ARRAY-OL.

Ainsi, nous avons présenté le concept de multi-degrés de granularité en prenant en consi-

[59] `http://www.lifl.fr/west`

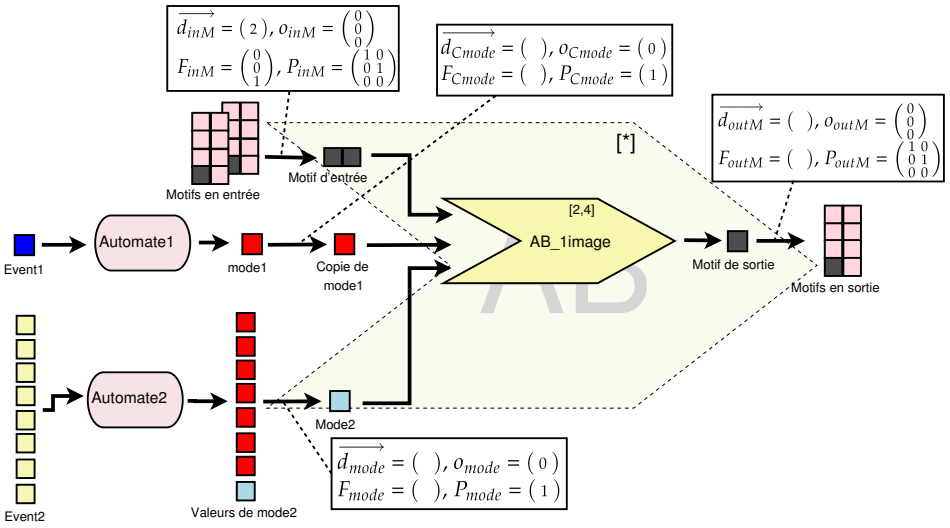


FIG. 7.24 – *Multi-degrés de granularité liés à des événements de contrôle différents*

dération le cas de deux automates de contrôle. Cependant, il est possible d'imaginer des scénarios plus complexes où les changements de modes dépendent de plusieurs automates de contrôle sur plusieurs niveaux de hiérarchie et selon des degrés de granularité différents. Ce sont des cas complexes pour lesquels des processus de *calcul d'horloge* sont indispensables pour la synchronisation des différents automates de contrôle. L'étude de ce problème ainsi que l'introduction des mécanismes de calcul d'horloge en ARRAY-OL font récemment partie d'un sujet de recherche lancé au sein de l'équipe West.

De façon générale, nous avons proposé un modèle offrant la possibilité aux utilisateurs de spécifier des comportements complexes mixant des traitements de données parallèle et des automates de contrôle via la notion de multi-degrés de granularité. La seule contrainte de notre approche est que le bloc de degré de granularité relatif à l'application globale doit respecter l'hypothèse de l'assignation unique du modèle ARRAY-OL. Ceci permet, d'une part, de respecter la régularité du modèle, et d'autre part, d'assurer le comportement et la fonctionnalité attendus de l'application en spécifiant les `Tilers` appropriés pour chaque situation. Ainsi, notre approche est similaire à celles utilisées généralement dans les langages de programmation qui donnent aux utilisateurs la possibilité de spécifier plusieurs scénarios indépendamment de leur faisabilité réel. Les utilisateurs ont donc la responsabilité pour gérer leurs applications pour avoir le comportement prévu et atteindre les objectifs attendus.

7.5 Validation expérimentale du modèle dans PTOLEMY II

L'introduction de la notion de degré de granularité pour les applications contrôlées décrites en ARRAY-OL conduit à la construction des flots de données pour ces application. Cette notion de flot est basée sur un mécanisme réactif synchrone permettant ainsi d'étudier le comportement des applications contrôlées ARRAY-OL en utilisant les différents outils développés autour des systèmes réactifs, et qui prennent en considération les applications mixant des traitements de données et du contrôle.

Dans ce qui suit, nous allons utiliser l'environnement PTOLEMY II pour la simulation des

applications parallèles décrites en ARRAY-OL avec un comportement de contrôle. Ce processus de simulation ne peut être réalisé qu'après avoir défini le degré de granularité pour l'application globale, et construire les flots de données correspondant. Le choix de l'environnement PTOLEMY II vient du fait que, d'une part, la projection du modèle ARRAY-OL sur un modèle SDF, ainsi que son implémentation dans PTOLEMY II (« ARRAY-OL *for* PTOLEMY II ») ont été étudié [DB05], et d'autre part, l'environnement PTOLEMY II offre la possibilité pour la description des automates de contrôle (domaine FSM) ainsi que le lien entre le modèle de contrôle et les modèles de calcul via le concept du MODALMODEL [HLL+03].

Nous avons présenté dans la section 2.3.3 (page 21) le modèle SDF comme étant un modèle à flot de données synchrones destiné à la représentation des applications monodimensionnelles. La projection d'ARRAY-OL sur SDF peut donc paraître limitative car la plupart des applications ARRAY-OL ne peuvent pas être décrites en SDF. Cependant, le but d'« ARRAY-OL *for* PTOLEMY II » ne consiste pas à modéliser des applications ARRAY-OL en SDF mais plutôt d'exécuter à l'aide du modèle SDF une application préalablement décrite en ARRAY-OL. Il semblerait plus naturel d'utiliser les modèles MDSDF et GMDSDF comme support pour la représentation des applications multidimensionnelles. Cependant, ces modèle n'existent que sur un plan théorique et n'ont fait l'objet d'aucune implémentation dans PTOLEMY II.

Le concept du MODALMODEL permet l'utilisation des machines à états finis (FSM) avec différents modèles de concurrence, et par conséquent, il est possible de les lier au modèle SDF décrivant des applications ARRAY-OL. L'idée de base consiste à écrire des programmes flot de données attachés aux différents états de l'automate de contrôle. Il est à noter que ce concept est syntaxiquement similaire à celui des automates de modes. Cette représentation ne respecte pas complètement notre méthodologie de séparation contrôle/données, présentée dans le chapitre 5, puisque les différents modes de fonctionnement sont décrits à l'intérieur des états et non pas de façon complètement séparée, mais elle permet de simuler le fonctionnement de nos applications qui est un processus intéressant pour valider leur comportement.

Nous allons illustrer l'implémentation des applications parallèles et contrôlées décrite en ARRAY-OL en utilisant l'environnement PTOLEMY II à travers un exemple simple représenté par la figure 7.25. Dans cet exemple, l'application prend en entrée deux tableaux infinis

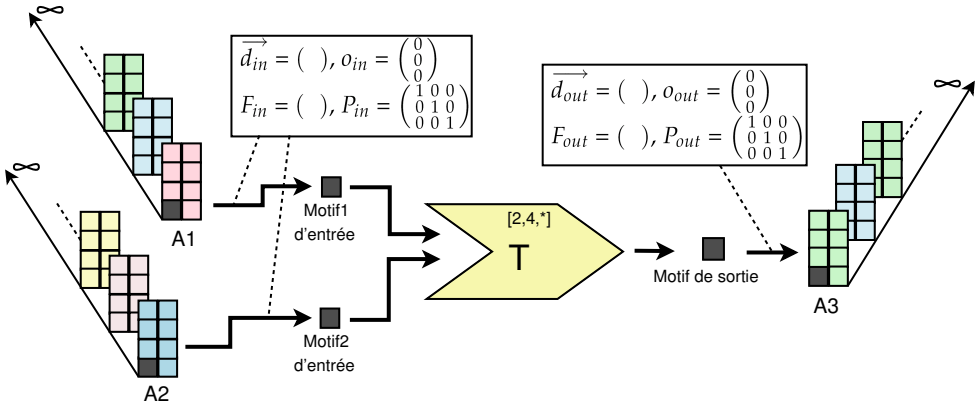


FIG. 7.25 – *Exemple simple d'un modèle* ARRAY-OL

d'images de taille 2×4 et retourne en sortie un tableau infini d'images de même taille. Pour

ce faire, la tâche T est répétée $2 \times 4 \times *$ fois, où pour chaque répétition elle prend en entrée deux motifs d'un seul élément pour fournir en sortie un motif d'un seul élément.

Dans ce qui suit, nous considérons que la tâche T peut changer son mode de fonctionnement selon l'événement externe event. Pour ce faire, nous remplaçons la tâche T par la tâche contrôlée CT qui peut opérer en trois modes de fonctionnement différents : A, B et C. L'activation et le passage entre ces modes se font via l'automate de contrôle représenté par la figure 7.26. Ainsi, pour des raisons de simplification, nous allons considérer que les processus des trois modes de fonctionnement sont définis comme suit :

- dans le mode A la tâche retourne en sortie les valeurs du tableau A2 (A3 = A2),
- dans le mode B la tâche retourne en sortie les valeurs du tableau A1 (A3 = A1), et
- dans le mode C la tâche retourne en sortie le résultat d'addition des éléments des tableaux A1 et A2 (A3 = A1 + A2).

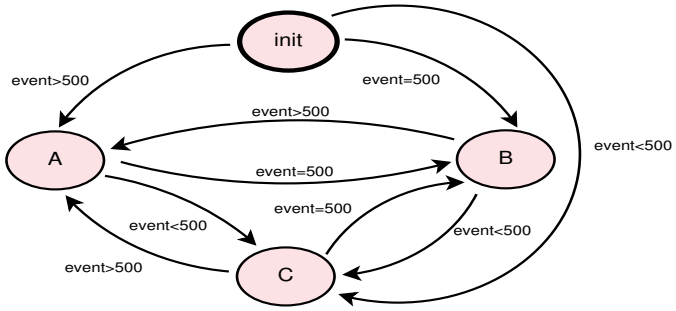


FIG. 7.26 – *Automate de contrôle pour la tâche* CT

Pour étudier le comportement réactif de cette application parallèle, nous allons considérer que le degré de granularité choisi pour cette application de la forme $\overrightarrow{d_{DG}} = \binom{2}{4}$, avec $F_{DG} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}$, $P_{DG} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ et $o_{DG} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$. Ce degré de granularité est relatif au traitement d'une seule image en sortie de taille 2×4. Dans ce cas, à chaque image en entrée pour chacun des deux tableaux A1 et A2 va correspondre une seule valeur de mode comme il est expliqué par la figure 7.27. Cette notion de degré de granularité permet donc de construire un modèle à flot de données, et par conséquent, rend possible la modélisation de l'application dans l'environnement PTOLEMY II. La figure 7.28 montre le modèle global relatif à cette application.

Comme nous l'avons décrit dans la section 2.4 (page24), le modèle global dans ARRAY-OL représente un graphe de dépendances de données orienté acyclique. Dans ce graphe, les sommets représentent les tâches, tandis que les arêtes représentent les tableaux et le sens d'exécution dans le graphe. Une telle représentation est possible en SDF puisque ce dernier suppose que chaque nœud du graphe représente un acteur qui doit produire et consommer des données. La projection du modèle global d'ARRAY-OL sur SDF, proposé dans [DB05], suppose que les nœuds SDF représentent uniquement des tâches, et les arêtes représentent en même temps les tableaux et le sens d'exécution du graphe.

Les différents nœuds du modèle global sont hiérarchiques puisqu'ils sont amenés à contenir les modèles locaux de l'application. Dans l'exemple de la figure 7.28, le graphe du modèle global est composé d'un ensemble de composants hiérarchiques : `Produire_A1`, `Produire_A2`, `Produire_event` et `Transformation`, d'un composant de type MODALMODEL relatif à la tâche `CT_1image`, et d'un ensemble d'acteurs prédéfinis dans PTOLEMY II per-

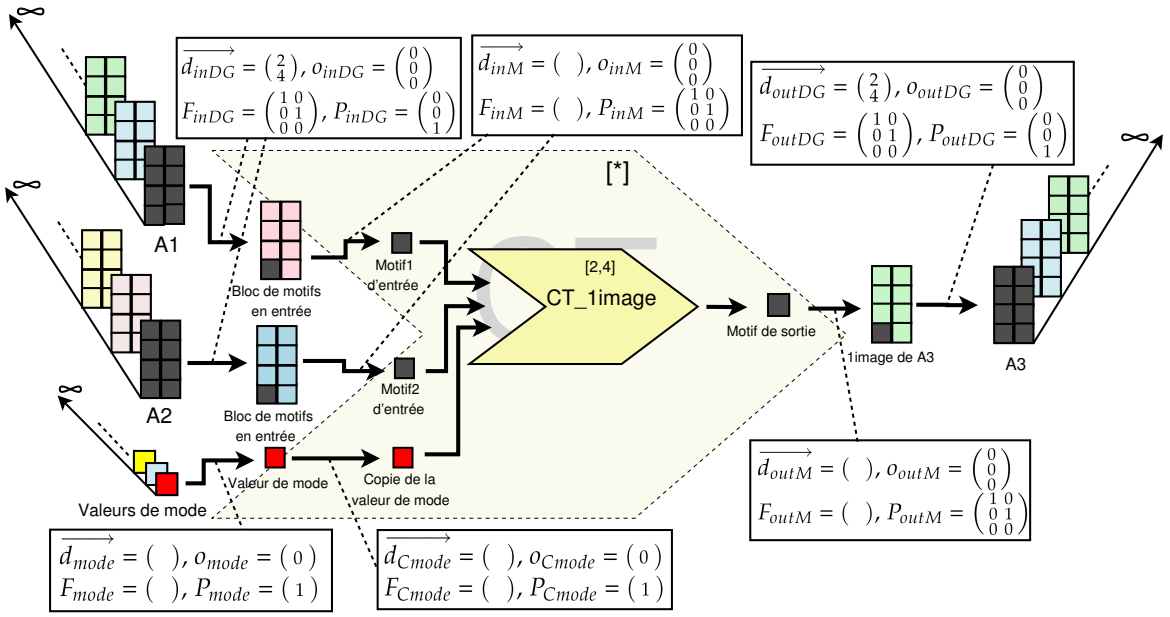


FIG. 7.27 – *Introduction du contrôle dans l'application de la figure 7.25 avec un degré de granularité de dimension* $[2,4]$

mettant l'affichage des valeurs des données : `Valeurs_A1`, `Valeurs_A2`, `Valeurs_event` et `Valeurs_A3`. Il est également à noter que la sémantique utilisée pour le modèle de calcul global est celle relative au domaine SDF (SDF *Director*).

Les composants hiérarchiques `Produire_A1`, `Produire_A2` et `Produire_event` permettent respectivement de produire des valeurs aléatoires pour les tableaux A1, A2 et event. La seule différence entre ces composants est dans la spécification des dimensions des tableaux produits. Le modèle local relatif au composant `Produire_A1` par exemple est donné par la figure 7.29.

La représentation du modèle local ARRAY-OL dans PTOLEMY II a nécessité la définition d'un ensemble d'acteurs dont le fonctionnement est résumé par le tableau de la figure 7.35 (page 165). Ainsi, un nouveau domaine appelé AOL *Director* a été introduit pour spécifier le modèle local des applications ARRAY-OL. Ce domaine hérite de celui du SDF, et permet de récupérer au niveau du TILER les informations nécessaire pour l'exécution de la tâche. Nous rappelons que ces différents concepts sont implémentés dans l'environnement ARRAY-OL *for* PTOLEMY II[60] développé par P. Dumont et *al*. [DB05].

Le composant `CT_1image` représente un nœud MODALMODEL. Le comportement de ce nœud est décrit par un automate de contrôle selon le domaine FSM défini dans PTOLEMY II comme il est montré par la figure 7.30. La structure de cet automate est équivalente à celle représentée par la figure 7.26. Ainsi, le principe de fonctionnement des nœuds de type MODALMODEL est similaire à celui des automates de modes. Dans ce modèle, à chaque état du système est associé le mode de fonctionnement correspondant sous forme d'un graphe de calcul. Dans le cas de notre exemple, nous allons associer aux états de l'automate les différentes parties de calcul pour chaque mode de fonctionnement. La figure 7.31 montre le modèle local relatif au mode de fonctionnement C, et dans lequel l'application retourne en

[60] `http://www.lifl.fr/west/aoltools`

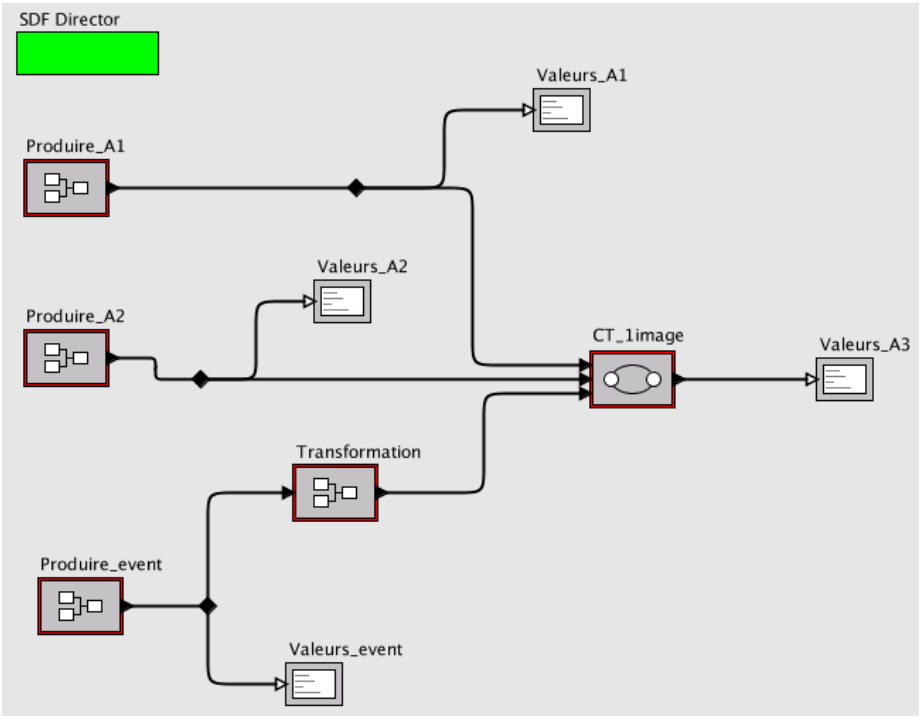


FIG. 7.28 – *Représentation du modèle global de l'application de la figure 7.27 dans* PTOLEMY II

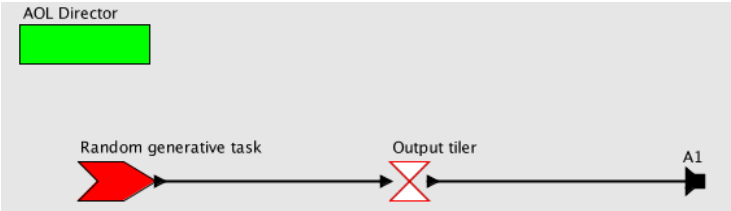


FIG. 7.29 – *Représentation du modèle local relatif au composant* `Produire_A1`

sortie le résultat de l'addition des différents éléments des tableaux en entrée.

Pour étudier le comportement de notre application, nous avons simulé son fonctionnement pour 5 répétitions en utilisant l'environnement PTOLEMY II. Le résultat obtenu est donné par la figure 7.32. Nous rappelons que dans cet exemple, le degré de granularité choisi correspond à une seule image en sortie de taille 2×4. Dans ce cas, en fonction de la valeur de l'événement event, l'automate de contrôle déclenche le mode de fonctionnement correspondant pour chaque répétition calculant une image en sortie de taille 2×4, et mène par conséquent à la construction d'un flot d'images en entrée et en sortie. Les résultats donnés par le processus de simulation confirment bien ce comportement (figure 7.32). Ces résultats montrent que pour les répétitions d'ordre 1, 2 et 5, les différents points des images en sortie sont calculés en mode A puisque event > 500. Dans ce mode, l'application donne comme résultat les valeurs du tableau A2 (A3 $=$ A2). Pour les répétitions d'ordre 3 et 4, event < 500, ce qui fait que les différents points des images en sortie pour ces répétitions sont calculés en mode C. Dans ce mode, l'application retourne en sortie le résultat de l'addition des éléments des tableaux A1 et A2 (A3 $=$ A1 $+$ A2).

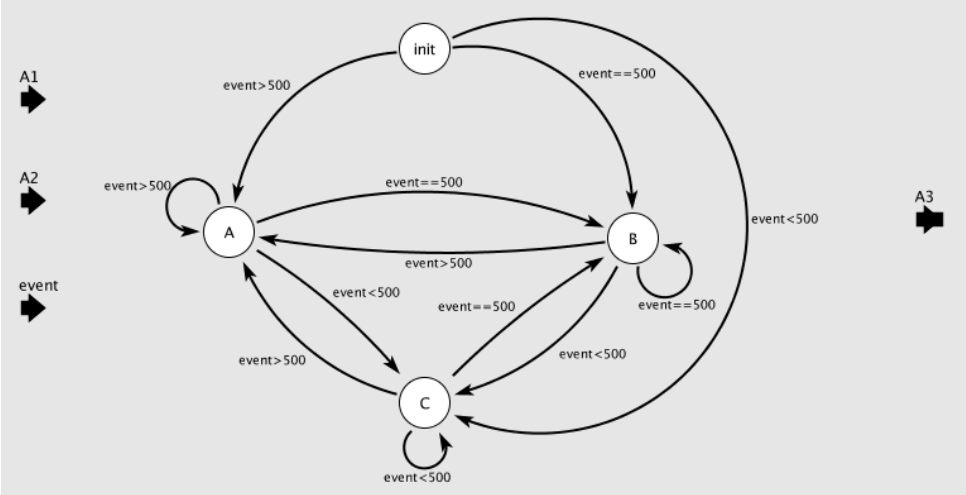


FIG. 7.30 – *Représentation de l'automate de contrôle relatif à la tâche* `CT_1image` *dans* PTOLEMY II

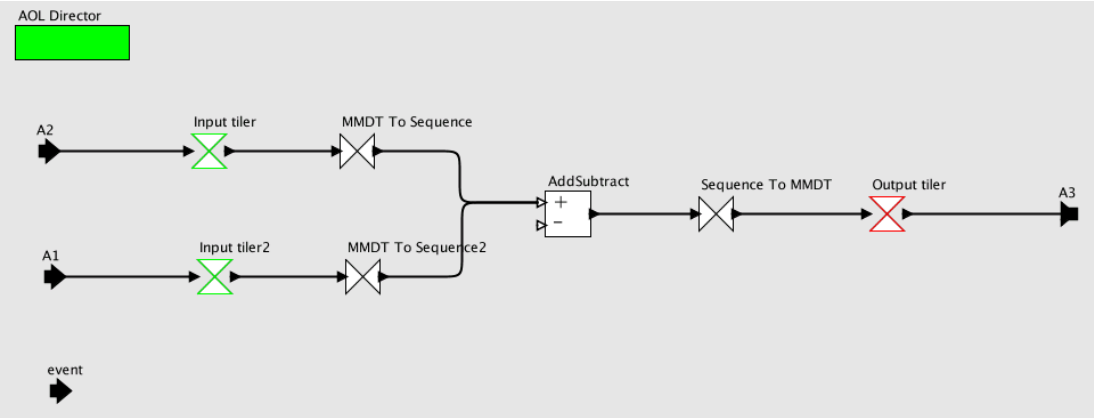


FIG. 7.31 – *Modèle de calcul local associé à l'état* C *de l'automate de la figure 7.30*

Nous pouvons également imaginer un autre scénario dans lequel le degré de granularité choisi est de la forme $\overrightarrow{d_{DG}} = (2)$, avec $F_{DG} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$, $P_{DG} = \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}$ et $o_{DG} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$. Ce degré de granularité est relatif au calcul d'une seule ligne de l'image en sortie à laquelle sera attribué une seule valeur de mode comme il est expliqué par la figure 7.33. Le modèle PTOLEMY II correspondant à cette application est très similaire à celui présenté précédemment. La seule différence est dans la spécification des informations des TILERs en entrée et en sortie puisque, dans ce cas, l'application associe une valeur de mode pour chaque ligne de l'image en sortie. Les résultats de processus de simulation pour cette application sont donnés par la figure 7.34. Dans cet exemple, nous remarquons que les tableaux infinis en entrée et en sortie sont transformés en des flots de données de deux éléments sur une seule dimension représentant les lignes de chaque image. Ainsi, le résultat de la simulation pour 8 répétitions permet de calculer les points de deux images en sortie. Ces résultats montrent bien qu'au calcul de chaque ligne de l'image en sortie est associé une seule valeur de mode qui permet d'exécuter le même mode de fonctionnement pour les deux points de la même ligne. Par exemple, la répétition d'ordre 2 permet de calculer la deuxième ligne de la première

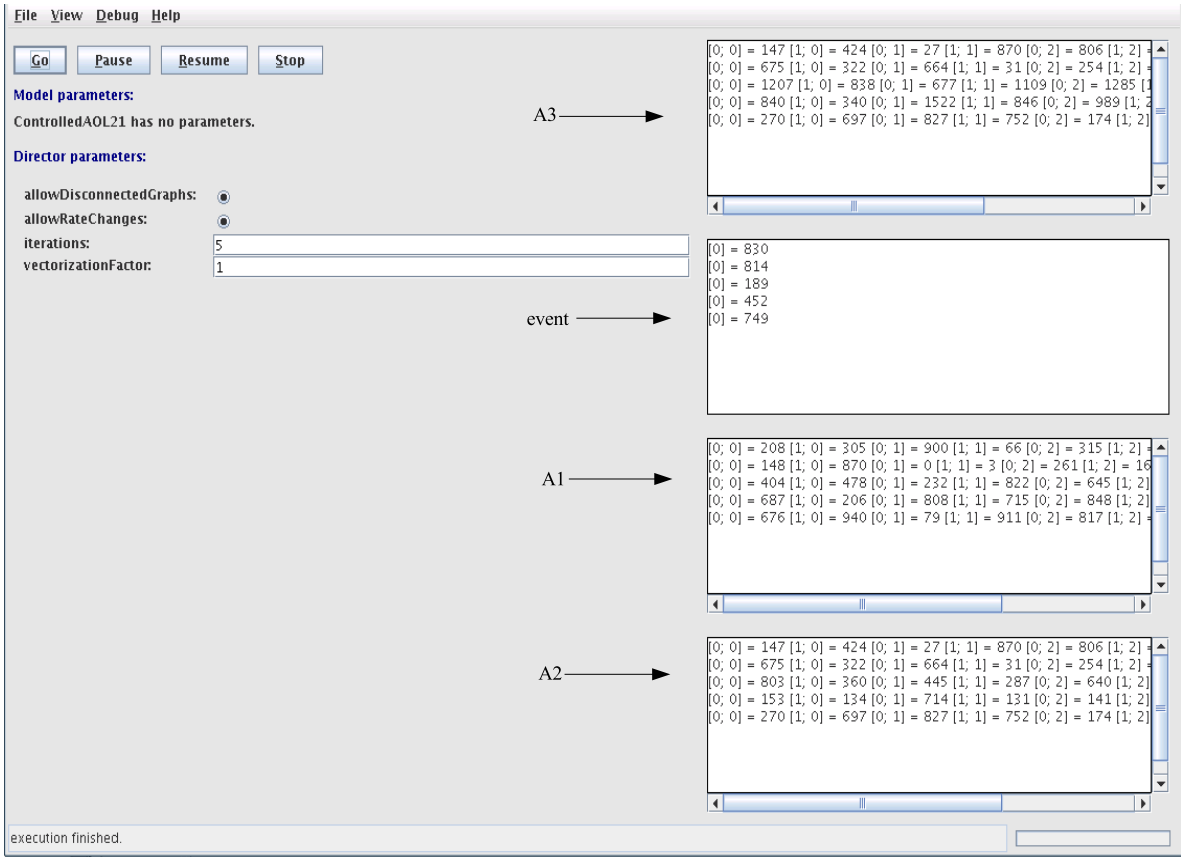


FIG. 7.32 – *Résultat de la simulation de l'exemple de la figure 7.27 dans* PTOLEMY II

image en sortie. Pour cette répétition, event $>$ 500, et par conséquent les deux points de la deuxième ligne sont exécutés en mode A qui donne en sotie les valeurs de tableau A2. La répétition d'ordre 3 permet de calculer la troisième ligne de la première image en sortie. Pour cette répétition, les deux points de la ligne sont exécutés en mode C, qui retourne le résultat de l'addition des éléments de A1 et A2, puisque event $<$ 500. Cet exemple montre effectivement la possibilité d'exécuter les différentes lignes de la même image dans des modes de fonctionnement différents.

Il est donc possible de simuler le comportement des applications contrôlées ARRAY-OL dans l'environnement PTOLEMY II en utilisant les concepts d'ARRAY-OL *for* PTOLEMY II et de MODALMODEL. Le processus de simulation peut être également réalisé pour des applications plus complexes. Dans le cadre de notre travail, nous avons également étudié le comportement de plusieurs exemples d'applications prenant en compte différents événements de contrôle et/ou faisant appel au concept de multi-degrés de granularité. L'idée est toujours la même : il suffit de bien définir le degré de granularité de l'application globale, ainsi que les valeurs pour les TILERs en entrée et en sortie pour respecter le comportement et les fonctionnalités attendus de l'application.

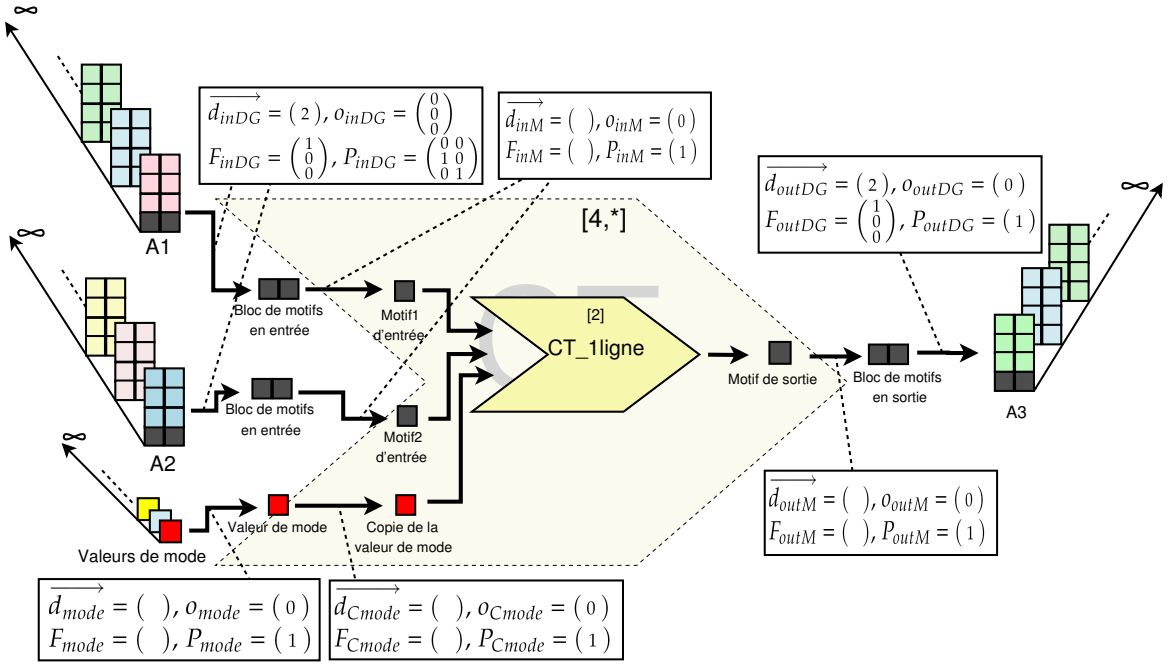


FIG. 7.33 – *Introduction du contrôle dans l'application de la figure 7.25 avec un degré de granularité de dimension* $[2,1]$

7.6 Synthèse et conclusion

Dans ce chapitre, nous avons étudié l'introduction du contrôle dans les applications de traitement parallèle décrites en ARRAY-OL. Nous avons montré que l'introduction du contrôle dans le modèle ARRAY-OL lui associe un comportement réactif qui nécessite la définition précise des différents moments de prise en compte des valeurs d'événement de contrôle. Pour ce faire, nous avons introduit la notion de degré de granularité pour les applications parallèles. Le but est de permettre la définition des sous-ensembles du domaine de répétition qui vont être exécutées dans le même mode de fonctionnement. Ceci correspond à la construction des flots de bloc de motifs entre lesquels les changements de modes de fonctionnement sont autorisés. Cette notion permet également de projeter le modèle ARRAY-OL sur un modèle flot de données puisque les valeurs de mode sont produites par un automate de contrôle, et dont la sémantique de base est une sémantique de flot. Le traitement des tâches ARRAY-OL doit donc suivre le rythme de fonctionnement de l'automate de contrôle puisque chaque tâche a besoin de toutes ses données en entrée pour son processus, et la valeur de mode produite par l'automate de contrôle en fait partie. Il est également à noter que notre approche est basée sur la technologie synchrone, et elle est fortement inspirée du concept des automates de mode.

Nous avons montré que notre approche est une approche orientée fonctionnelle dans le sens où le degré de granularité de l'application est défini en fonction du comportement et de la fonctionnalité attendus de cette application. Ce concept est assez riche puisqu'il permet la modélisation de plusieurs scénarios prenant en compte des événements externes venant de l'environnement de l'application, ou des événements internes résultant d'un calcul interne de l'application.

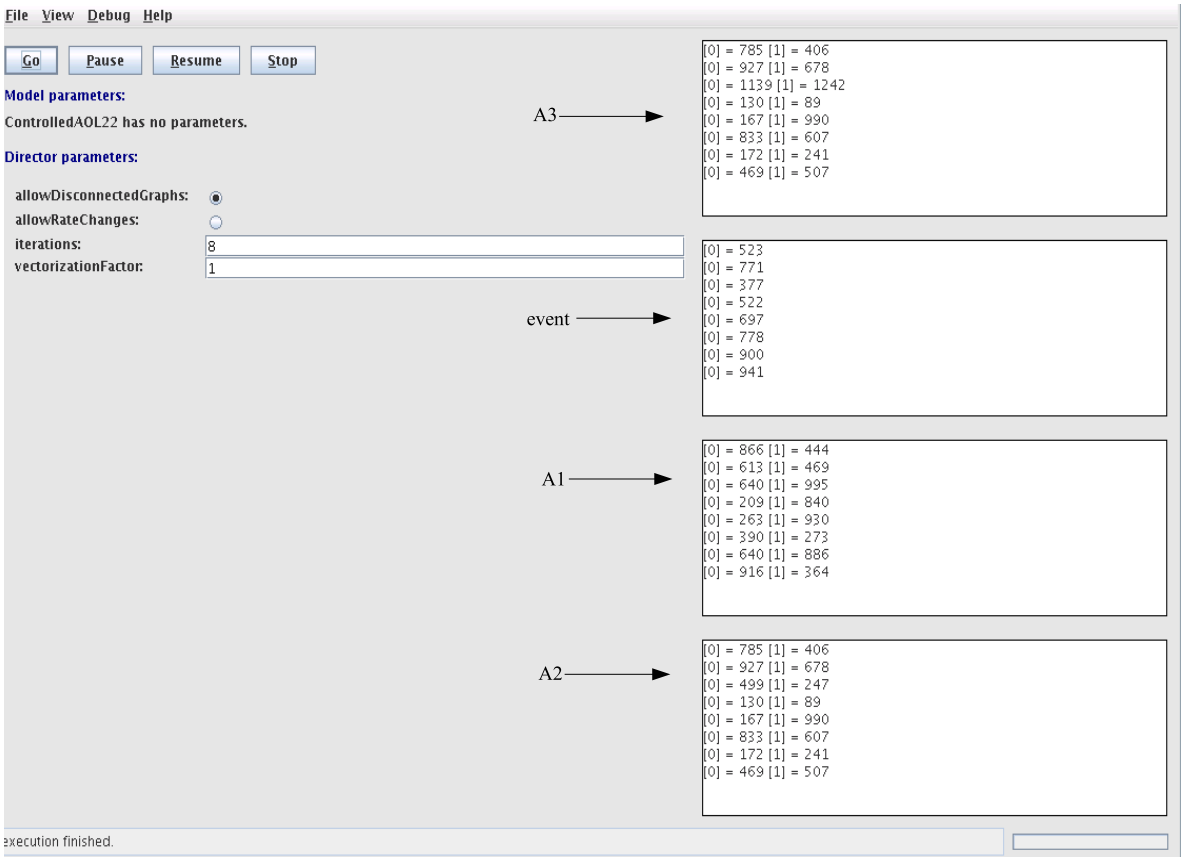


FIG. 7.34 – *Résultat de la simulation de l'exemple de la figure 7.33 dans* PTOLEMY II

Nous avons également étudié la notion de multi-degrés de granularité pour les applications parallèles. Ce concept permet la modélisation des applications plus complexes qui peuvent changer leurs modes de fonctionnement selon des degrés de granularité différents. Les automates de contrôle correspondants peuvent évoluer à des rythmes variés et il est nécessaire de les synchroniser pour pouvoir définir le comportement global de l'application. C'est un problème assez compliqué qui nécessite l'introduction des processus de fusion et de calcul d'horloge.

Notre concept de degré de granularité permet donc l'introduction des comportements de contrôle dans les applications ARRAY-OL. Le modèle proposé offre la possibilité aux utilisateurs de spécifier des comportements complexes mixant des traitements de données parallèles et du contrôle. Il est donc possible de spécifier plusieurs scénarios indépendamment de leur faisabilité réelle. Les utilisateurs ont donc la responsabilité de gérer leurs applications et d'assurer leurs fonctionnalités attendues.

À la fin de ce chapitre, nous avons étudié la simulation de comportement des applications contrôlées ARRAY-OL en utilisant l'environnement PTOLEMY II. Ceci est réalisé grâce à l'application ARRAY-OL *for* PTOLEMY II qui permet la projection du modèle ARRAY-OL sur le modèle SDF, et à la notion de MODALMODEL qui permet d'associer des modèles de calcul aux différents états d'un automate de contrôle. Cette représentation est syntaxiquement similaire à celle des automates de modes permettant ainsi de distinguer les différents modes de fonctionnement de l'application étudiée. Il est à noter que la simulation de notre modèle

dans PTOLEMY II ne respecte pas complètement notre méthodologie de séparation contrôle/données, présentée dans le chapitre 5, puisqu'elle ne sépare pas la structure de l'automate des différents modes de calcul. Cependant, elle permet de bien simuler le comportement des modèles ARRAY-OL étudiés, de mieux comprendre la notion de degré de granularité et les liens entre les automates de contrôle et l'application parallèle, et de confirmer l'adéquation des modèles définis aux applications visés.

Acteur	*Description*
Hierarchical task 	Définit un composant hiérarchique pour la spécification du modèle local. Le domaine décrivant le composant hiérarchique doit être un `AOL Director`
Input tiler 	Définit un acteur pour les TILERs en entrée qui permet de découper les tableaux d'entrée en motifs. Les informations de TILER peuvent être spécifiées via une interface graphique
Output tiler 	Définit un acteur pour les TILERs en sortie qui permet de regrouper les motifs de sortie pour construire les tableaux. Les informations de TILER peuvent être spécifiées via une interface graphique
MMDT To Sequence 	Sert de convertisseur entre les motifs multidimensionnels d'ARRAY-OL et les motifs traditionnels manipulas par les acteurs de PTOLEMY II
Sequence To MMDT 	Sert de convertisseur entre les motifs traditionnels manipulas par les acteurs de PTOLEMY II et les motifs multidimensionnels d'ARRAY-OL
Random generative task 	Représente une tâche ARRAY-OL particulière qui permet de générer des motifs aléatoires d'une certaine dimension. La dimension des motifs peut être introduite via une interface graphique
Constant generative task 	Représente une tâche ARRAY-OL particulière qui permet de générer des motifs avec des valeurs constantes d'une certaine dimension. La dimension des motifs et la constante peuvent être introduites via une interface graphique
Identity task 	Représente une tâche ARRAY-OL particulière qui ne fait rien (une tâche « fantôme »)

FIG. 7.35 – *Définitions des acteurs introduits pour la spécification du modèle local* ARRAY-OL

Chapitre 8

Profil UML2 pour la modélisation des applications massivement parallèles et contrôlées

Dans ce chapitre, nous allons présenter les différents concepts définis dans le profil UML2 pour l'environnement de développement GASPARD2. Ces concepts offrent un moyen efficace pour la modélisation au plus haut niveau d'abstraction des applications de traitement intensif à parallélisme massif, l'architecture matérielle associée, ainsi que le processus de placement d'une application sur une architecture donnée. Après avoir présenté ces différents concepts et leurs contraintes et contexte d'utilisation, nous allons nous intéresser à la modélisation de l'introduction des notions de contrôle dans le profil GASPARD2.

Nous rappelons que le but principal de notre travail est de proposer un modèle de conception mixant des traitements de données parallèles et du contrôle, et en particulier, l'introduction du contrôle dans l'environnement de développement GASPARD2 basé sur le modèle ARRAY-OL. Comme nous l'avons discuté dans le chapitre 7, la sémantique du modèle ARRAY-OL est basée sur une notion de temps triviale ou banalisée, et elle est différente de celle des automates de contrôle qui sont basés sur une sémantique de flot. Il est donc nécessaire de proposer un moyen de modélisation permettant de mixer ces deux mondes différents. Nous allons donc proposer un ensemble de concepts permettant la modélisation de l'automate de contrôle et des différents changements de modes tout en respectant la sémantique de base des modèles dans GASPARD2.

L'approche que nous proposons est fortement liée à celle de la notion de degré de granularité, et respecte bien notre méthodologie de séparation contrôle/données. Nous pouvons également considérer cette étude comme une mise en œuvre des différentes approches présentées dans les chapitres 5 et 7.

8.1 Introduction

Nous rappelons que le contexte de base de notre travail consiste à étudier la spécification des applications de traitement intensif à parallélisme massif, l'introduction des comportements de contrôle pour ces applications selon une approche réactive synchrone, ainsi que leur modélisation à haut niveau d'abstraction selon une approche IDM. Nous rappelons également que notre travail s'inscrit dans le cadre du projet de développement de l'environnement GASPARD2, basé sur le langage de spécification ARRAY-OL, et principalement conçu pour la co-conception et la co-modélisation visuelles des applications de traitement du signal pour les systèmes sur puce.

Dans la section 4.3 (page 62), nous avons montré que l'environnement GASPARD2 est basé sur un modèle en Y défini autour de trois concepts de base : l'*application* permettant de spécifier les fonctionnalités du système, l'*architecture matérielle* utilisée pour la représentation de la plateforme d'exécution, et l'*association* permettant de spécifier le placement d'une application sur une architecture donnée. La définition de ces trois modèles dans l'environnement GASPARD2 est complètement abstraite dans le sens où aucun détail d'implémentation ou d'exécution n'est pris en compte à ce niveau de modélisation. C'est une approche *orientée composant* qui permet de séparer clairement les différentes parties du modèle Y, et par conséquent, de faciliter la réutilisation des IPs matériels et logiciels déjà existants.

Le point de départ dans la chaîne de conception relative à l'environnement de développement GASPARD2 consiste à modéliser à haut niveau d'abstraction les concepts de base du modèle Y (application, architecture matérielle et association) en définissant un profil UML pour la représentation de ces concept. Cette chaîne de conception est basée sur une approche MDA dans laquelle des processus de transformation et de raffinement de modèles sont ap-

pliqués en partant des modèles abstraits UML jusqu'à la génération automatique du code exécutable.

Dans le cadre de notre travail, nous nous intéressons en particulier à la modélisation au plus haut niveau d'abstraction des trois modèles de base du Y. Notre but consiste à définir un profil UML2 pour la modélisation des concepts utilisés dans l'environnement GASPARD2, et en particulier proposer une solution pour la modélisation des concepts de contrôle introduits dans la spécification des applications de traitement parallèle. Le profil que nous proposons étend la sémantique d'UML2, et permet d'étudier une plus grande catégorie d'application mixant des traitements de données parallèles à la ARRAY-OL et des notions de contrôle. Ce profil respecte notre méthodologie de séparation contrôle/données, et représente également une mise en œuvre de notre approche d'introduction du contrôle dans les applications parallèles selon la notion de degré de granularité. Cette mise en œuvre représente un moyen efficace pour la description et l'échange des applications, et offre aux utilisateurs la liberté de décrire plusieurs comportements possibles de leur applications de manière bien organisée.

8.2 Profil UML2 pour la modélisation des concepts dans GASPARD2

Pour la modélisation des différents concepts utilisés dans l'environnement de développement GASPARD2, une première version d'un profil UML2 a été proposée [CDMB05, Cuc05]. Ce profil est défini autour de cinq packages : `component`, `factorization`, `application`, `hardwareArchitecture` et `association` comme il est montré par la figure 8.1. Dans cet ensemble,

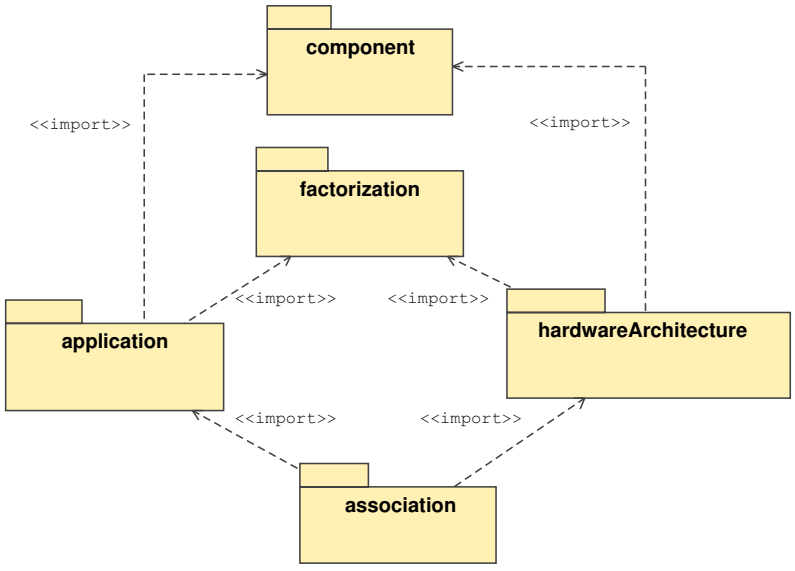


FIG. 8.1 – *Différents packages du profil UML2 dans l'environnement* GASPARD2

nous retrouvons les trois concepts de base du modèle Y dans GASPARD2 : `application`, `hardwareArchitecture` et `association`. Les packages `component` et `factorization` sont introduits pour regrouper les concepts de base communs pour toutes les parties du modèle Y.

Dans ce profil, les packages `application` et `hardwareArchitecture` spécifient respectivement l'application logicielle d'un système et l'architecture matérielle utilisée pour la réalisation des fonctionnalités de cette application. Ces deux packages partagent la même définition de la structure de composant et des mécanismes de répétition introduits dans les packages `component` et `factorization`. Le but d'introduction de ces deux derniers packages est de regrouper les concepts communs utilisés par les différentes parties du modèle Y pour favoriser leur processus de réutilisation. Le package `association` quand à lui, est utilisé pour introduire certaines directives de base pour le placement d'une application sur une architecture matérielle donnée.

Dans le cadre de notre travail, nous avons proposé plusieurs améliorations[61] du profil GASPARD2 en lui introduisant des nouveaux concepts et des contraintes OCL [LDBR06]. Ces améliorations permettent d'enrichir et de fortifier la description au plus haut niveau d'abstraction des différentes notions utilisées dans les modèles GASPARD2. Dans ce qui suit, nous allons présenter plus en détail les différents packages du profil GASPARD2. La réalisation de ce profil est faite en utilisant MAGICDRAW[62], l'outil visuel de modélisation UML écrit en java, et qui tourne sur plusieurs plateformes.

8.2.1 Le package `component`

Le package `component` regroupe les concepts et les mécanismes communs utilisés pour la description de l'application et de l'architecture matérielle dans l'environnement GASPARD2. L'objectif principal de ce package consiste à définir un support pour la méthodologie *orientée composant* en permettant la représentation de la structure d'un composant indépendamment de son environnement d'utilisation. Le but de cette approche est de favoriser autant que possible l'aspect de réutilisation d'IPs logiciels et matériels.

Dans ce package, nous avons introduit le stéréotype abstrait `GaspardComponent` qui étend la métaclasse `Component` d'UML2 en lui associant une interprétation particulière relative aux caractéristiques des composants dans GASPARD2. L'introduction de ce stéréotype permet de spécifier de façon claire les contraintes et les règles d'utilisation des composants UML2 pour les rendre compatible à la description des composants dans GASPARD2.

Le concept de *composant* représente la structure de base réutilisable dans le profil GASPARD2. Un composant de type `GaspardComponent` est vu comme une adaptation ou une restructuration des mécanismes utilisés pour la définition d'un composant en UML2. Chaque composant dans GASPARD2 est donc considéré comme un élément indépendant qui peut communiquer avec son environnement externe via la notion de `port`, et dont sa structure et sa composition hiérarchique peuvent être définies via la notion de `part` et de `connector`.

De façon générale, la structure d'un composant de type `GaspardComponent` peut être *élémentaire*, *composée* ou *répétitive*. Pour décrire ces trois types de composant nous avons introduits trois stéréotypes de base dérivés du concept abstrait `GaspardComponent` : `ElementaryComponent`, `CompoundComponent`, et `RepetitiveComponent` comme il est montré par la figure 8.2. Ces stéréotypes étendent la métaclasse `Component` d'UML2 pour permettre la spécification des composants ayant une sémantique particulière relative aux modèles GASPARD2. Ainsi, dans la description des modèles GASPARD2, nous imposons que chaque composant doit être exclusivement de type `ElementaryComponent`, `CompoundComponent`, ou

[61] Je me dois de rajouter qu'une partie d'amélioration du profil GASPARD2 est le fruit de plusieurs discussions de réunion d'équipe West.

[62] `http://www.magicdraw.com`

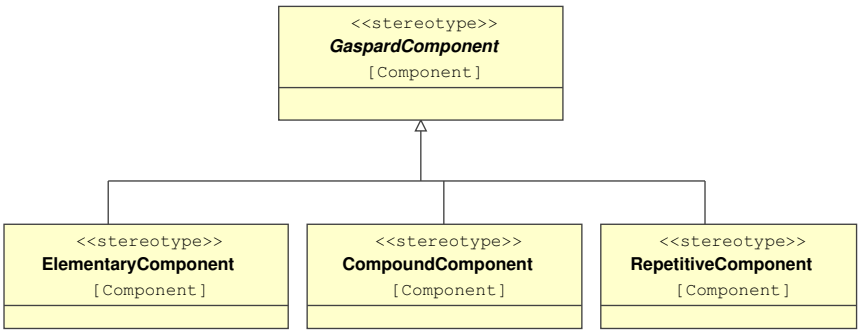


FIG. 8.2 – *Le package* `component`

`RepetitiveComponent` comme il est exprimé par la contrainte OCL :

inv *: self.stereotype → exists(name='*`ElementaryComponent`*' **xor** name='*`CompoundComponent`*' **xor** name='*`RepetitiveComponent`*')*

Cette contrainte représente un invariant qui doit être valide pour tous les composants de type `GaspardComponent`. Elle permet de vérifier que tout composant défini dans le profil GASPARD2 doit être exclusivement stéréotypé `ElementaryComponent`, `CompoundComponent`, ou `RepetitiveComponent`.

Un composant de type `ElementaryComponent` est utilisé pour la représentation d'un composant particulier qui est généralement vu comme une boite noire. La description de ce composant ne doit donc contenir aucun composant de type `GaspardComponent` comme il est spécifié par la contrainte OCL :

inv *: self.part → select(stereotype → exists(name='*`GaspardComponent`*')) → size()=0*

Cette contrainte représente un invariant qui doit assurer que chaque composant de type `ElementaryComponent` ne doit avoir aucune `part` de type `GaspardComponent`. Le comportement de ces composants élémentaires est généralement décrit dans un langage source telque du code SystemC ou Java. La figure 8.3 représente un exemple simple d'un composant de type `ElementaryComponent` qui prend en entrée un motif de deux éléments de type entier pour fournir en sortie un motif d'un seul élément de même type.

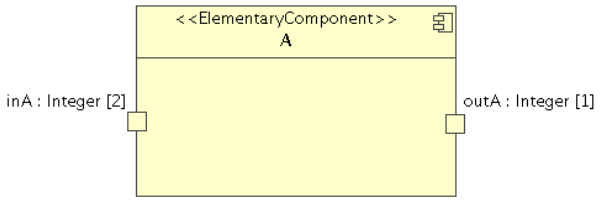


FIG. 8.3 – *Exemple d'un composant de type* `ElementaryComponent`

Le concept de `CompoundComponent` est utilisé pour la définition des composants composés dans GASPARD2 et faciliter leur représentation hiérarchique. Un composant de type `CompoundComponent` doit uniquement contenir des composants de type `GaspardComponent` comme il est exprimé par la contrainte OCL :

inv : self.part → forAll(p | p.stereotype → exists (name='GaspardComponent'))

Cette contrainte représente un invariant qui impose que toutes les `parts` d'un composant composé dans GASPARD2 doivent être de type `GaspardComponent`. Par exemple si le résultat de la tâche élémentaire A, représentée par la figure 8.3, est utilisé par une autre tâche élémentaire B qui produit un motif de deux éléments de type booléen, alors le composant AB de type `CompoundComponent` est utilisé pour regrouper et représenter la relation entre les deux composants A et B comme il est montré par la figure 8.4.

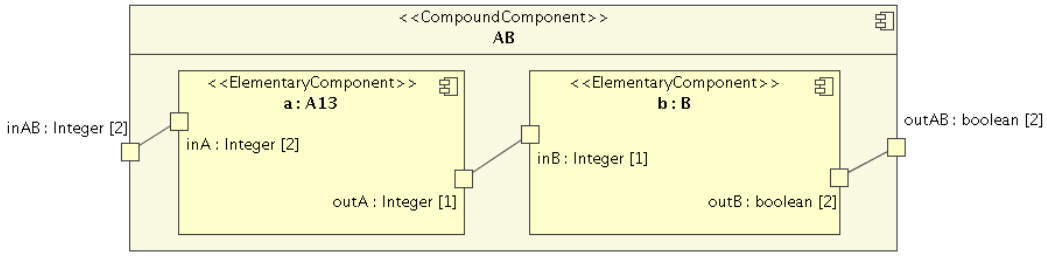


FIG. 8.4 – *Exemple d'un composant de type `CompoundComponent`*

Un composant de type `RepetitiveComponent` permet de décrire la répétition de la `part` modélisée à l'intérieur de ce composant. Cette notion de répétition est définie en se basant sur les concepts du modèle ARRAY-OL, et représente un espace de répétition sur un seul composant de type `GaspardComponent` comme il est exprimé par la contrainte OCL :

inv : self.part → select(stereotype → exists(name='GaspardComponent'))→ size()=1

Cette contrainte représente un invariant qui spécifie que chaque composant répétitif doit contenir une et une seule `part` de type `GaspardComponent`. La figure 8.5 représente un exemple simple d'une répétition sur la tâche élémentaire A. Dans cet exemple, le compo-

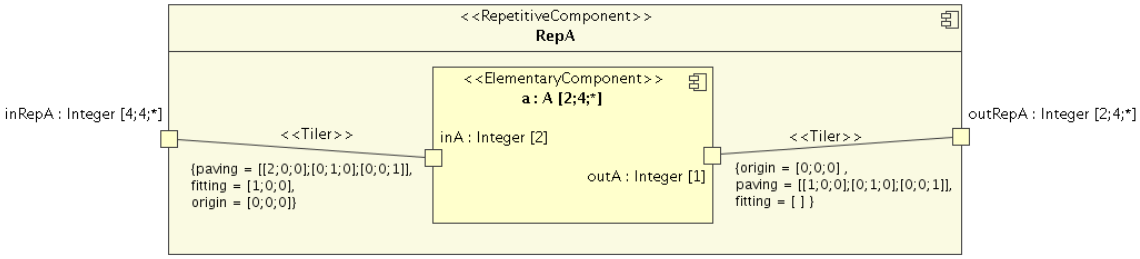


FIG. 8.5 – *Exemple d'un composant de type `RepetitiveComponent`*

sant RepA reçoit en entrée un tableau infini d'entiers de dimension $[4,4,*]$, et retourne en sortie un tableau infini d'entiers de dimension $[2,4,*]$. À l'intérieur de ce composant, une instance de la tâche A est répétée une infinité de fois sur la dimension $[2,4,*]$. Ainsi, le lien entre le composant RepA et la `part a : A` est spécifié par des connecteurs spéciaux permettant d'exprimer la répétition dans GASPARD2 selon les bases du modèle ARRAY-OL. Les concepts de ces connecteurs seront présentés plus en détails dans la description du package `factorization`.

La particularité des composants définis dans le profil GASPARD2 se résume principale-

ment à l'introduction du mécanisme de *multiplicité multidimensionnelle*. Ce concept permet de spécifier des directives sur le nombre de répétition des éléments liés à un composant de type `GaspardComponent`. Les éléments dans GASPARD2 qui peuvent avoir une multiplicité multidimensionnelle sont uniquement les `ports` et les `parts`. La multiplicité multidimensionnelle définie sur les `ports` permet la représentation des tableaux multidimensionnels, tandis que celle définie sur les `parts` est utilisée pour la spécification de l'espace de répétition de la `part` en question. Par exemple, dans la figure 8.5, le concept de multiplicité multidimensionnelle a été utilisé pour exprimer les tableaux multidimensionnels en entrée et en sortie, ainsi que l'espace de répétition de la `part` répétée.

Il est à noter que le concept de multiplicité existe déjà dans la spécification d'UML2, et qui peut être vu comme un tableau d'éléments à une seule dimension. La définition de la multiplicité multidimensionnelle revient donc à généraliser le concept de multiplicité en UML2 pour permettre la représentation des tableaux multidimensionnels. Ceci est relatif à une extension du métamodèle UML2 et ne va pas apparaître dans la description du profil GASPARD2. Dans ce qui suit, nous allons définir implicitement le concept de multiplicité multidimensionnelle en donnant une interprétation particulière au concept de multiplicité défini en UML2. Dans ce contexte, la multiplicité multidimensionnelle d'un élément est décrite sous la forme : $[n_1; n_2; \ldots ; n_m]$, où n_i représente le nombre d'éléments pour la dimension i.

La figure 8.6 représente un exemple simple d'un composant GASPARD2 avec des `ports` et une `part` de multiplicité multidimensionnelle et son équivalent en UML2. Dans le cas du

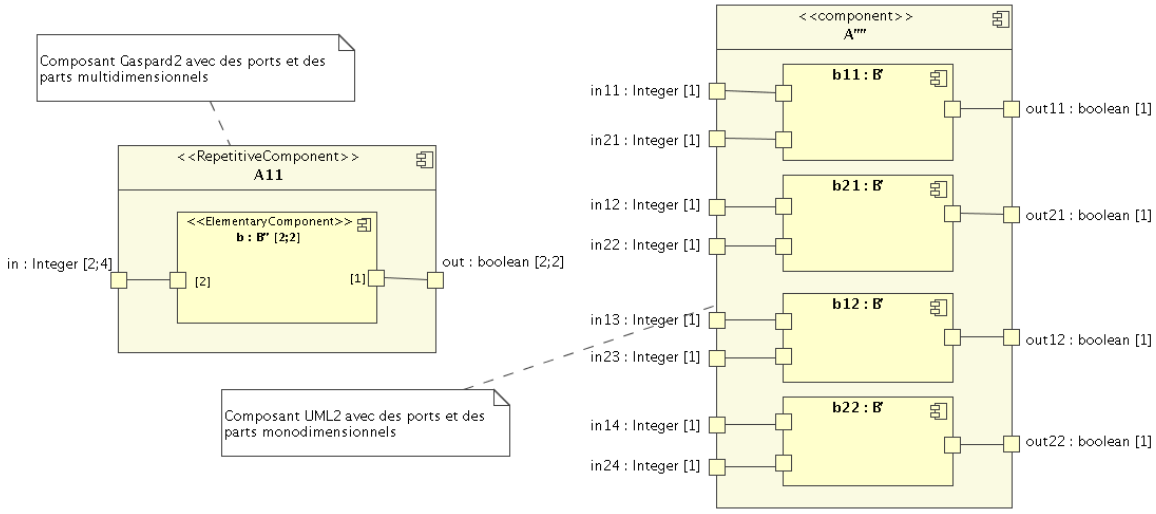


FIG. 8.6 – *Représentation d'un composant* GASPARD2 *avec une multiplicité multidimensionnelle et son équivalent en UML2*

composant de type `GaspardComponent`, les `ports` ont une multiplicité multidimensionnelle et peuvent être décrit comme suit : port_name : type[multiDim_multiplicity], où port_name spécifie le nom du `port`, type représente le type des différents éléments qui peuvent circuler à travers ce `port`, et [multiDim_multiplicity] représente la multiplicité multidimensionnelle du `port`. Le `port in` représenté dans l'exemple du composant GASPARD2 de type `RepetitiveComponent` spécifie que les données pouvant passer par ce `port` sont des tableaux d'entiers de dimension $[2, 4]$. Dans le cas de la `part`, la multiplicité multidimensionnelle per-

met de représenter l'espace de répétition de la `part` b : B qui est répétée 2×2 fois sur un espace bidimensionnels, avec deux répétitions sur chaque dimension. De façon générale, l'exemple de la figure 8.6 montre que la notion de multiplicité multidimensionnelle permet d'avoir une représentation plus compact des composants définis dans le profil GASPARD2. Ce concept permet également de spécifier différentes topologies de représentation des éléments sur plusieurs dimensions, ce qui n'est pas le cas pour la modélisation UML2.

8.2.2 Le package `factorization`

Le package `factorization` regroupe des mécanismes de factorisation structurelle inspirés du modèle ARRAY-OL. Ces mécanismes permettent d'exprimer l'aspect multidimensionnel, ainsi que la relation entre les tâches de calcul et les éléments des motifs en entrée et ceux en sortie. La description du package `factorization` est basée sur la définition des topologies de lien principalement utilisées pour l'expression de la répétition dans les modèles GASPARD2.

La définition des différentes topologies de lien est introduite via le stéréotype abstrait `LinkTopology` qui étend la métaclasse `Connector` d'UML2. Ce concept est utilisé pour la spécification d'un ensemble d'informations associées aux liens entre les différents composants de type `GaspardComponent`.

La topologie de lien définie dans le profil GASPARD2 prend en considération la multiplicité multidimensionnelle, ainsi que la représentation des informations des TILERs utilisés dans le modèle ARRAY-OL pour la spécification des applications de traitement de données massivement parallèle. La figure 8.7 représente les différents types de topologie de lien définis dans le package `factorization`. La `tagged value modulo` associée au stéréotype

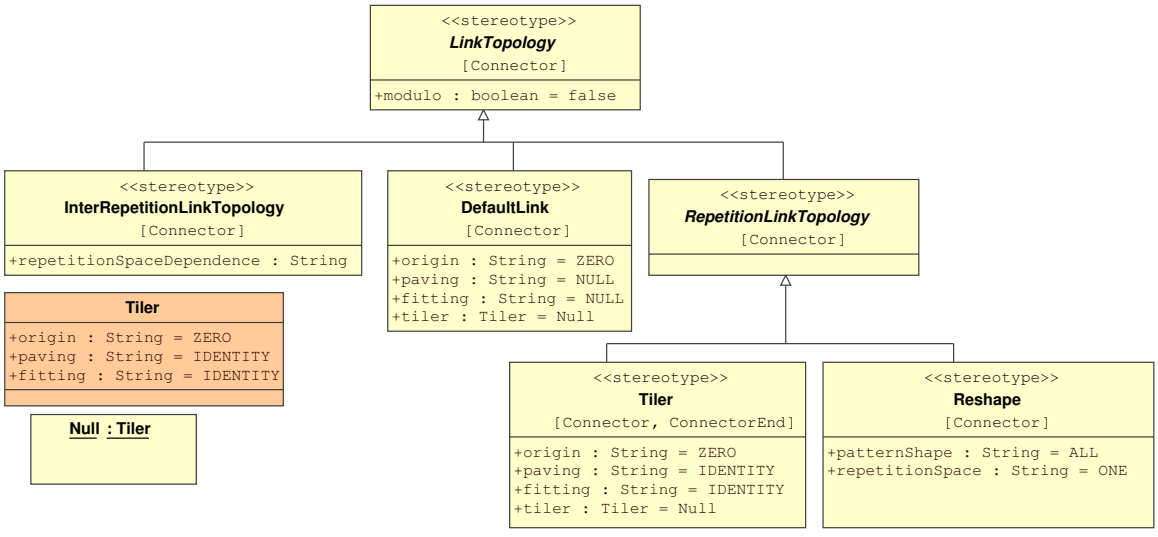


FIG. 8.7 – *Le package* `factorization`

`LinkTopology` permet de spécifier si les tableaux d'éléments manipulés sont toriques (s'il est possible de parcourir les éléments d'un tableau modulo sa taille).

Parmi les topologies de lien définies dans le package `factorization`, nous avons le concept d'`InterRepetitionLinkTopology` qui étend la métaclasse `Connector` d'UML2. Ce

concept est utilisé pour la spécification d'un type particulier de lien régulier entre les différentes répétitions des instances du même composant comme il est exprimé par la contrainte OCL :

inv *: self.participant → size()=2* ***and*** *self.participant[0]=self.participant[1]*

Cette contrainte représente un invariant qui spécifie que les deux participants dans un lien de type `InterRepetitionLinkTopology` (les deux éléments entre lesquels le lien est établi) représentent le même composant.

Un connecteur de type `InterRepetitionLinkTopology` peut être utilisé pour la représentation de plusieurs topologies de lien entre les différentes répétitions d'un même composant GASPARD2. La figure 8.8 représente un exemple dans lequel le concept d'`InterRepetitionLinkTopology` est utilisé pour la spécification d'une topologie de type grille cyclique à deux dimensions. Dans cette topologie, le système est composé de 6 instances d'un composant G, où chaque instance est connectée à ses quatre voisins. La `tagged value repetition-`

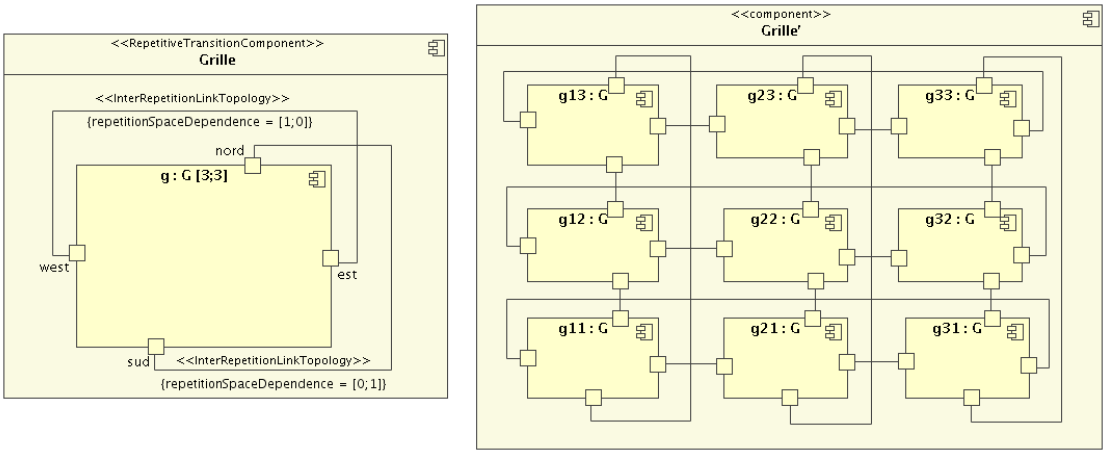


FIG. 8.8 – *Modélisation d'une topologie en grille cyclique*

`SpaceDependence` associée aux connecteurs de type `InterRepetitionLinkTopology` permet la spécification des vecteurs de dépendance entre les différentes répétitions de l'instance d'un composant pour déterminer le lien entre ces répétitions ou la position des voisins pour chaque instance potentielle.

La notion d'`InterRepetitionLinkTopology` peut être également utilisée pour la modélisation des fonctions de calcul particulières telle que :

$$\begin{cases} Y_n = f(Y_{n-1}, X) \\ Y_0 = \text{initValue} \end{cases}$$

Dans ce cas, le calcul de chaque répétition d'ordre n de la fonction f dépend du résultat fourni par le calcul de la répétition d'ordre $n-1$. Un exemple simple d'une telle fonction est le calcul d'intégrale représenté par la figure 8.9. Dans cet exemple, le calcul de chaque répétition d'une instance de la tâche Somme necessite le résultat de calcul de la répétition précédente. Cette relation est spécifiée par l'utilisation d'un connecteur de type `InterRepetitionLinkTopology` avec `repetitionSpaceDependence` = 1.

Nous remarquons également que cette notion de dépendance inter-répétition permet

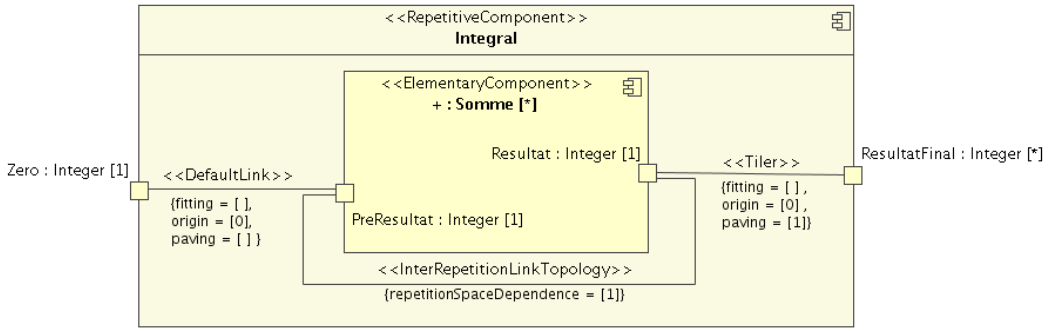


FIG. 8.9 – *Exemple de modélisation d'une fonction d'intégral*

d'introduire une sémantique de *flot* pour les différentes répétitions des instances du composant en question. Ainsi, un connecteur de type `InterRepetitionLinkTopology` doit être défini entre deux `ports` de même type et de même multiplicité comme il est décrit par la contrainte OCL :

$$\textbf{\textit{inv}} : self.connectorEnd \rightarrow forAll(x,y \mid x.multiplicity = y.multiplicity \ \textbf{\textit{and}} \ x.type = y.type)$$

Cette contrainte représente un invariant qui impose que les deux extrémités (les `ports`) d'un connecteur de type `InterRepetitionLinkTopology` doivent avoir la même multiplicité et faire circuler le même type de données.

Puisque le connecteur de type `InterRepetitionLinkTopology` est utilisé pour la spécification d'une relation de dépendance régulière entre les différentes répétitions d'un même composant, ce connecteur doit être uniquement décrit à l'intérieur d'un composant de type `RepetitiveComponent` comme il est exprimé par la contrainte OCL :

$$\textbf{\textit{inv}} : self.owner.stereotype \rightarrow exists(name = \text{'}\texttt{RepetitiveComponent}\text{'})$$

Cette contrainte représente un invariant qui permet d'assurer que le contenant d'un connecteur de type `InterRepetitionLinkTopology` est un composant de type `RepetitiveComponent`.

Un autre type de connecteur défini dans le package `factorization` est celui de `DefaultLink` qui étend la métaclasse `Connector` d'UML2. L'objectif d'utilisation d'un lien de type `DefaultLink` est de définir les valeurs des données par défaut produites ou consommées par une tâche en dehors de son espace de répétition. Un connecteur de type `DefaultLink` peut donc avoir sens uniquement en relation avec un connecteur de type `InterRepetitionLinkTopology` comme il est exprimé par la contrainte OCL :

$$\textbf{\textit{inv}} : self.owner.connector \rightarrow exists(c \mid c.stereotype \rightarrow exists(name=\text{'}\texttt{InterRepetitionLinkTopology}\text{'}) \ \textbf{\textit{and}} \ self.connectorEnd=c.connectorEnd)$$

Cette contrainte représente un invariant qui permet d'assurer que tout connecteur de type `DefaultLink` doit être défini avec un connecteur de type `InterRepetitionLinkTopology` sur le même `port` (défini par le concept du `connectorEnd`).

Ainsi, puisqu'un connecteur de type `DefaultLink` est toujours associé à un connecteur de type `InterRepetitionLinkTopology` pour permettre la définition des valeurs de données dans un contexte répétitif. Ce connecteur doit être uniquement décrits à l'intérieur d'un com-

posant de type `RepetitiveComponent` comme il est exprimé par la contrainte OCL :

inv *: self.owner.stereotype → exists(name =' `RepetitiveComponent`')*

Cette contrainte représente un invariant qui permet d'assurer que le contenant du connecteur de type `DefaultLink` doit être un composant de type `RepetitiveComponent`.

À chaque connecteur de type `DefaultLink` sont associées des `tagged values` pour la spécification des informations d'un TILERs selon le concept d'ARRAY-OL (`origin`, `paving`, `fitting` et `tiler`). Ces informations permettent de retrouver les valeurs des données par défaut dans le cas où il yen a plusieurs.

La figure 8.10 représente un exemple dans lequel un lien de type `DefaultLink` est utilisé pour la spécification des valeurs par défaut pour des données en entrée utilisées par une tâche répétée avec une dépendance inter-répétition. Dans cet exemple, pour les quatre pre-

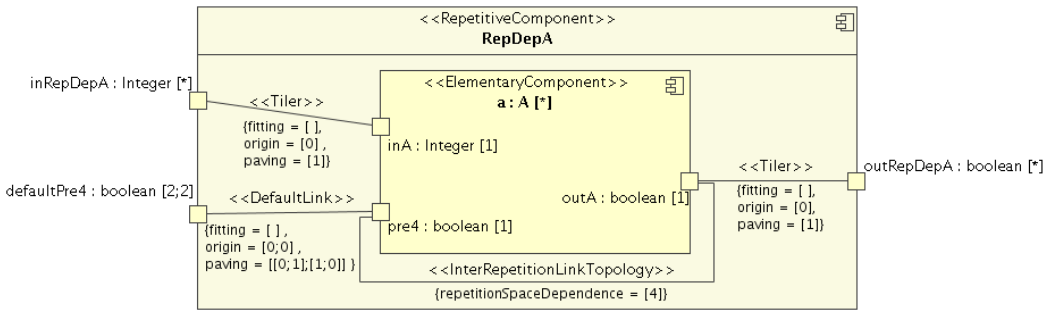


FIG. 8.10 – *Exemple d'utilisation d'un connecteur de type* `DefaultLink`

mières répétitions de l'instance de tâche a : A, la valeur en entrée de pre4 est donnée par le tableau defaultPre4 de dimension $[2, 2]$. Les informations de `Tiler` définies sur le connecteur de type `InputDefaultLink` permettent d'extraire la valeur de donnée pre4 correspondante pour chacune de ces répétitions.

Une autre topologie de lien définie dans le package `factorization` est celle de `RepetitionLinkTopology`. Ce concept abstrait étend la métaclasse `Connector` d'UML2, et il est principalement basé sur les mécanismes de répétition définis en ARRAY-OL. La notion de `RepetitionLinkTopology` permet d'introduire deux concepts de base : le `Tiler` et le `Reshape`.

Le concept de `Tiler` étend les métaclasses `Connector` et `ConnectorEnd` d'UML2, et il est utilisé pour la spécification d'un lien particulier de répétition basé sur le modèle ARRAY-OL. À ce concept est associé des `tagged values` permettant d'exprimer les informations liées à la définition d'un TILER en ARRAY-OL. Ces informations peuvent être définies de deux façons

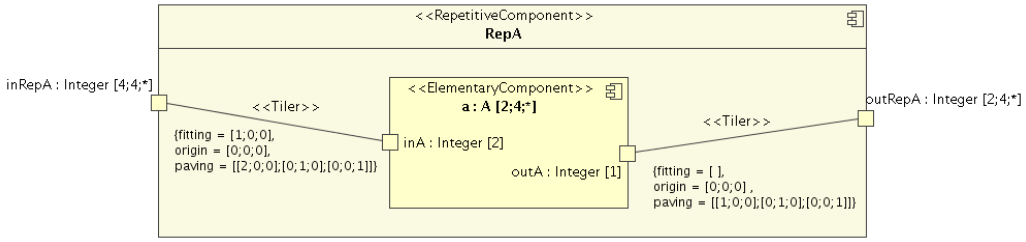


FIG. 8.11 – *Utilisation des* `tagged values` `origin`, `paving` *et* `fitting`

différentes : la première est la plus classique qui consiste à associer des valeurs aux `tagged values` `origin`, `paving` et `fitting` comme il est montré par l'exemple de la figure 8.11, tandis que la deuxième consiste à définir une instance de la classe prédéfinie `Tiler`, lui associer des valeurs aux `tagged values` `origin`, `paving` et `fitting`, et utiliser cette instance de classe comme valeur pour la `tagged value` `tiler` associée au stéréotype `Tiler`. Cette deuxième façon de faire est utile dans le cas où plusieurs connexions de type `Tiler` ont les mêmes valeurs pour l'origine, la matrice de pavage et celle d'ajustage. Dans ce cas, une seule instance de la classe `Tiler` est définie et peut être utilisée pour la définition de tous les `Tilers` correspondant comme il est montré par l'exemple de la figure 8.12.

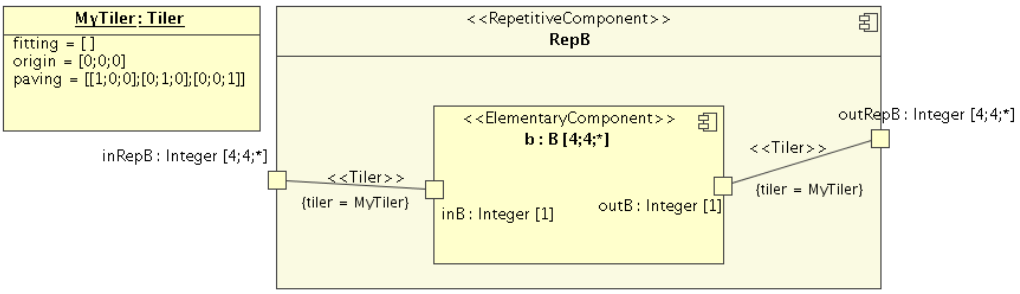


FIG. 8.12 – *Utilisation de la `tagged value tiler`*

Pour éviter des conflits dans l'utilisation des informations associées aux `tagged values` des `Tilers`, nous imposons une utilisation exclusive entre la `tagged value` `tiler` et les trois `tagged values` `origin`, `paving` et `fitting` comme il est exprimé par la contrainte OCL :

inv *: (self.taggedValue → select(name='`tiler`' and value<>'NULL')* ***implies*** *self.taggedValue → exclude(name='`origin`' or name='`paving`' or name='`fitting`'))* ***and*** *(self.taggedValue → select(name='`tiler`'* ***and*** *value='NULL')****implies*** *self.taggedValue → exclude(name='`tiler`'))*

Cette contrainte représente un invariant qui spécifie que si la valeur de `tagged value` `tiler` est différente de NULL, alors cette `tagged value` est utilisée pour la spécification des informations de `Tiler`, et les autres `tagged values` (`origin`, `paving` et `fitting`) sont ignorées (ou exclues). Dans le cas contraire (la valeur de la `tagged value` `tiler` est NULL), la `tagged value` `tiler` est ignorée puisque nous considérons que les informations de `Tiler` sont définies par les `tagged values` `origin`, `paving` et `fitting`.

Les exemples des figures 8.11 et 8.12 montrent également l'utilité de l'utilisation des connecteurs de type `Tiler` pour la représentation des liens répétés dans les modèles GASPARD2. Un connecteur de type `Tiler` doit être donc défini à l'intérieur d'un composant répétitif comme il est exprimé par la contrainte OCL :

inv *: self.oclIsTypeOf('`Connector`')* ***implies*** *self.owner.stereotype → exists (name='`RepetitiveComponent`')*

Cette contrainte représente un invariant permettant d'assurer que dans le cas où le stéréotype `Tiler` est appliqué sur un connecteur, ce connecteur doit être défini à l'intérieur d'un composant de type `RepetitiveComponent`.

La définition du stéréotype `Tiler` sur les fins de connexion (`ConnectorEnd`) est uniquement utilisée pour la représentation d'un type particulier de lien répétitif lié au concept de `Reshape`. Le stéréotype `Reshape` étend la métaclasse `Connector` d'UML2. Son objectif est de

permettre la représentation des topologies de lien plus complexe dans lesquelles les éléments d'un tableau multidimensionnel sont redistribués ou réorganisés dans un autre tableau comme il est expliqué par l'exemple de la figure 8.13. Dans cet exemple chaque ligne

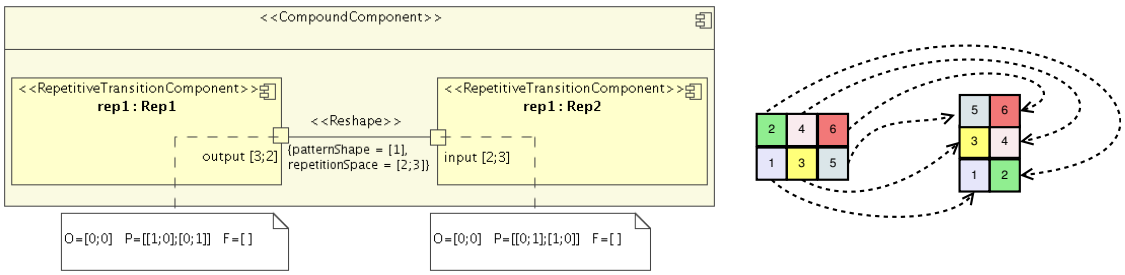


FIG. 8.13 – *Exemple d'utilisation d'un connecteur de type* `Reshape`

du tableau en sortie de l'instance de composant Rep1 est rangé en une colonne dans le tableau en entrée de l'instance de composant Rep2. Le connecteur `Reshape` offre donc une représentation plus compacte permettant d'éviter l'introduction d'une tâche « fantôme » qui ne fait que consommer les motifs du tableau en entrée et les ranger dans le tableau en sortie via des connecteurs de type `Tiler` comme il est montré par l'exemple de la figure 8.14.

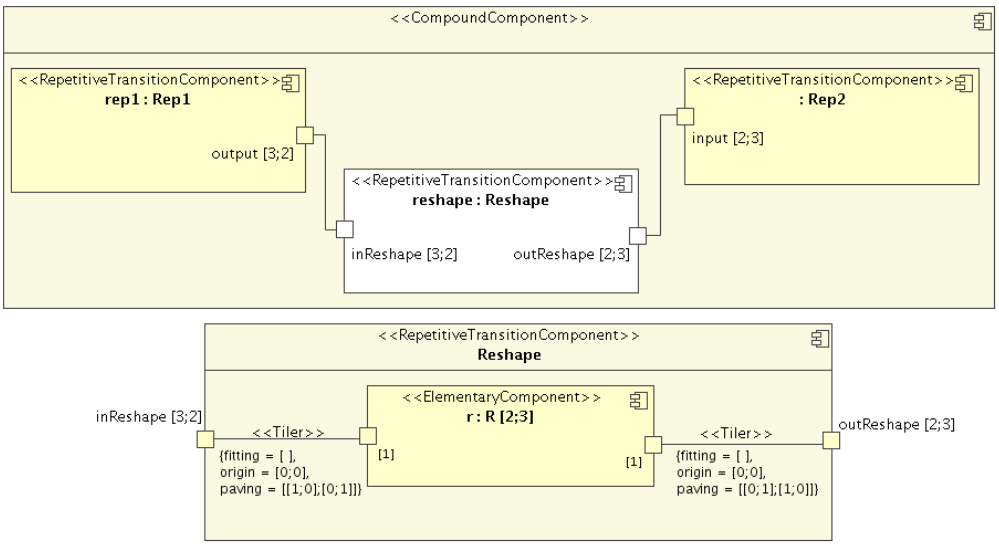


FIG. 8.14 – *Représentation équivalente à celle de la figure 8.13 sans utilisation d'un connecteur* `Reshape`

La définition d'un connecteur de type `Reshape` doit être accompagnée par la spécification des informations de TILER sur les deux fins de connexion. Ces informations sont utilisées pour déterminer comment sélectionner les éléments du tableau source et comment les ranger dans le tableau destination. En d'autres termes, chaque fin de connexion d'un connecteur de type `Reshape` doit être stéréotypée `Tiler` comme il est exprimé par la contrainte OCL :

inv *: self.connectorEnd → forAll(stereotype → exists(name='`Tiler`'))*

Il est également à noter que nous avons associé deux `tagged values` à la définition du stéréotype `Reshape` : `patternShape` et `repetitionSpace` qui représentent respectivement la forme du motif et son espace de répétition.

8.2.3 Le package `application`

Le package `application`, représenté par la figure 8.15, est utilisé pour introduire certains concepts spécifiques pour la description de la partie liée à l'application. Dans ce package,

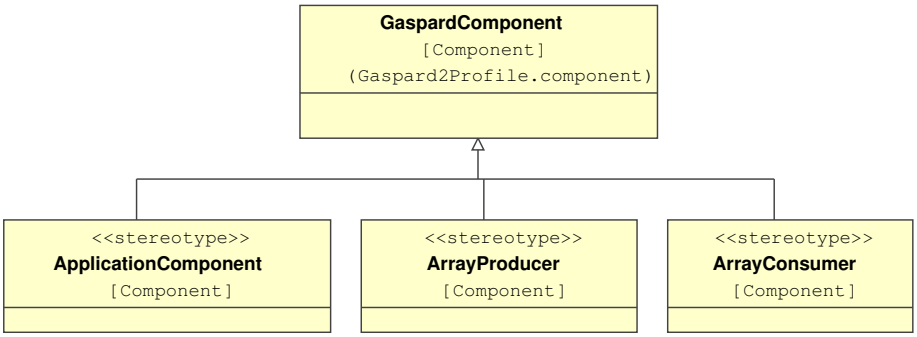


FIG. 8.15 – *Le package `application`*

nous avons introduit trois stéréotypes de base : `ApplicationComponent`, `ArrayProducer` et `ArrayConsumer` qui étendent la métaclasse `Component` d'UML2.

Un composant de type `ApplicationComponent` représente un cas particulier d'un composant `GaspardComponent` en lui ajoutant une connotation de type applicatif comme il est montré par l'exemple de la figure 8.16. Un tel composant peut être vu comme une fonction manipulant des données en entrée, provenant de son environnement externe via des `ports` d'entrée, pour fournir des résultats en sortie à cet environnement via des `ports` de sortie. Puisque le concept d'`ApplicationComponent` est lié à celui de `GaspardComponent`, chaque composant de type `ApplicationComponent` peut être de type `ElementaryComponent`, `CompoundComponent`, ou `RepetitiveComponent`, et doit respecter les mêmes contraintes définies pour les composants de type `GaspardComponent`.

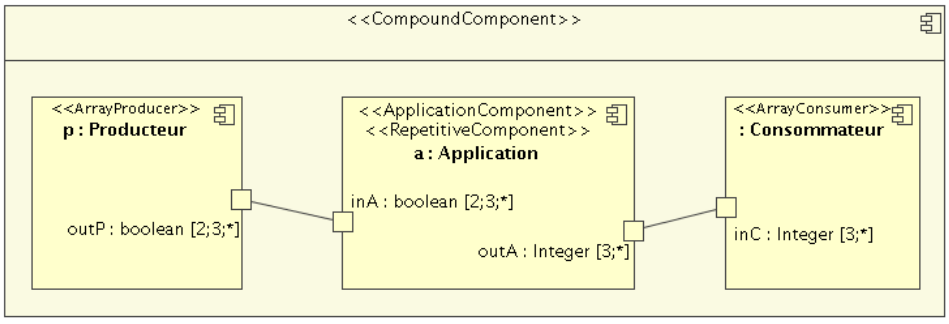


FIG. 8.16 – *Exemple de composants de type `ApplicationComponent`, `ArrayProducer` et `ArrayConsumer`*

Un composant de type `ArrayProducer` est introduit pour la modélisation d'un type particulier de composant dans GASPARD2. Le fonctionnement d'un `ArrayProducer` consiste

uniquement à produire des tableaux de données multidimensionnels et éventuellement de taille infinie comme il est montré par l'exemple de la figure 8.16. Un tel composant ne doit donc avoir que des `ports` de sortie comme il est exprimé par la contrainte OCL :

inv *: self.ownedPort.provided → size()=0* ***and*** *self.ownedPort.required → size()>0*

Le fonctionnement inverse d'un `ArrayProducer` est assuré par un composant de type `ArrayConsumer`. Le concept d'un `ArrayConsumer` est introduit pour la modélisation d'un type particulier de composant dans GASPARD2, et dont le rôle consiste uniquement à consommer des tableaux de données multidimensionnels et éventuellement de taille infinie comme il est montré par l'exemple de la figure 8.16. Ainsi, un tel composant ne doit avoir que des `ports` d'entrée comme il est exprimer par la contrainte OCL :

inv *: self.ownedPort.required → size()=0* ***and*** *self.ownedPort.provided → size()>0*

Les deux types de composant : `ArrayProducer` et `ArrayConsumer` peuvent être vus comme étant une représentation logique au niveau application des composants physique de type capteur et actionneur. Le but principal de ces deux concepts est de spécifier le point de départ et le point final pour chaque application. En d'autres termes, l'utilisation des composants `ArrayProducer` et `ArrayConsumer` permet de répondre à la question : *dans le déroulement d'une application, d'où viennent les données en entrée consommées par les premières tâches et où vont les données en sortie fournies par les dernières tâche ?*.

8.2.4 Le package `hardwareArchitecture`

Le concept de `hardwareArchitecture` regroupe des composants particuliers de type `GaspardComponent` en leur ajoutant une connotation d'architecture matérielle. L'objectif de ce package est d'introduire des concepts liés à la description des différents composants dans une architecture matérielle utilisée comme support d'exécution pour les applications dans GASPARD2. Une vue globale de ce package est donnée par la figure 8.17.

Le concept de `HardwareComponent` étend la métaclasse `Component` d'UML2, et représente un cas particulier d'un `GaspardComponent`. À ce titre, ce concept hérite des différents mécanismes définis autour des composants de type `GaspardComponent`, et peut être de type `ElementaryComponent`, `CompoundComponent`, ou `RepetitiveComponent`. Ces différents types de composants représentent une abstraction des ressources matérielles physiques : `Processor`, `Memory`, `Communication`, etc. Ces ressources peuvent avoir des structures très variées et ont la capacité de communiquer selon différentes topologies de lien.

Dans le package `hardwareArchitecture`, un composant de type `Processor` est introduit pour la représentation des ressources matériels utilisées comme support d'exécution pour les composants applicatifs. Un composant de type `Memory` est utilisé comme un support de stockage pouvant contenir des données ou des instructions. Un composant de type `Communication` est utilisé pour la représentation des différentes connexions matérielles entre les composants de type `HardwareComponent`. Un composant de type `IO` est introduit pour la représentation des protocoles de communication entre les composants matériels et leur environnement externe. Cette notion permet également de modéliser une partie de l'architecture de l'environnement qui est généralement de type `Actuator` ou `Sensor`.

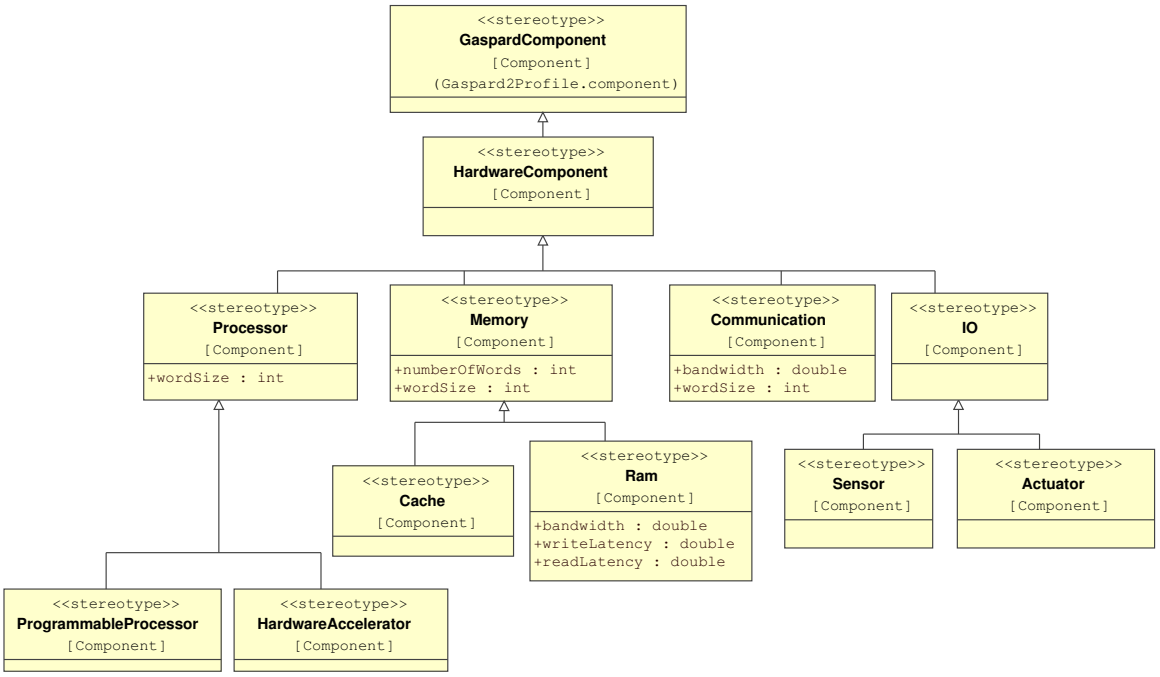


FIG. 8.17 – *Le package* `hardwareArchitecture`

La figure 8.18 représente un exemple simple d'une architecture matérielle d'un « QuadriPro ». Cet exemple montre l'utilisation des mécanismes définis dans les packages `factorization` et `component`. Un ensemble de `tagged values` est également ajouté pour les différents composant matériels pour permettre la spécification des caractéristiques de ces composants telles que leur capacité, leur performances, etc.

8.2.5 Le package `association`

L'objectif du package `association`, présenté par la figure 8.19, est de regrouper les différents mécanismes utilisés pour le placement d'une application sur une architecture matérielle donnée. Dans ce package, nous avons introduit le concept abstrait `Allocation` qui étend le concept `Dependency` d'UML2. L'objectif d'`Allocation` est de définir le lien entre les composants applicatifs et les composants matériels associés. Dans ce contexte, nous distinguons trois types d'allocation : `DataAllocation`, `TaskAllocation` et `CommunicationAllocation`. Ainsi, une `Allocation` peut être exclusivement de type `DataAllocation`, `TaskAllocation` ou `CommunicationAllocation` comme il est exprimé par la contrainte OCL :

*inv : self.stereotype → exists(name='DataAllocation' **xor** name='TaskAllocation' **xor** name='CommunicationAllocation')*

Le concept de `DataAllocation` est utilisé pour spécifier sur quelles mémoires sont stockés les tableaux de données. Dans le profil GASPARD2, les tableaux sont référencés par les `ports` sur lesquels ils sont définis. Le placement des données revient donc à placer les différentes instances des `ports` en entrée et ceux en sortie liés aux composants applicatifs.

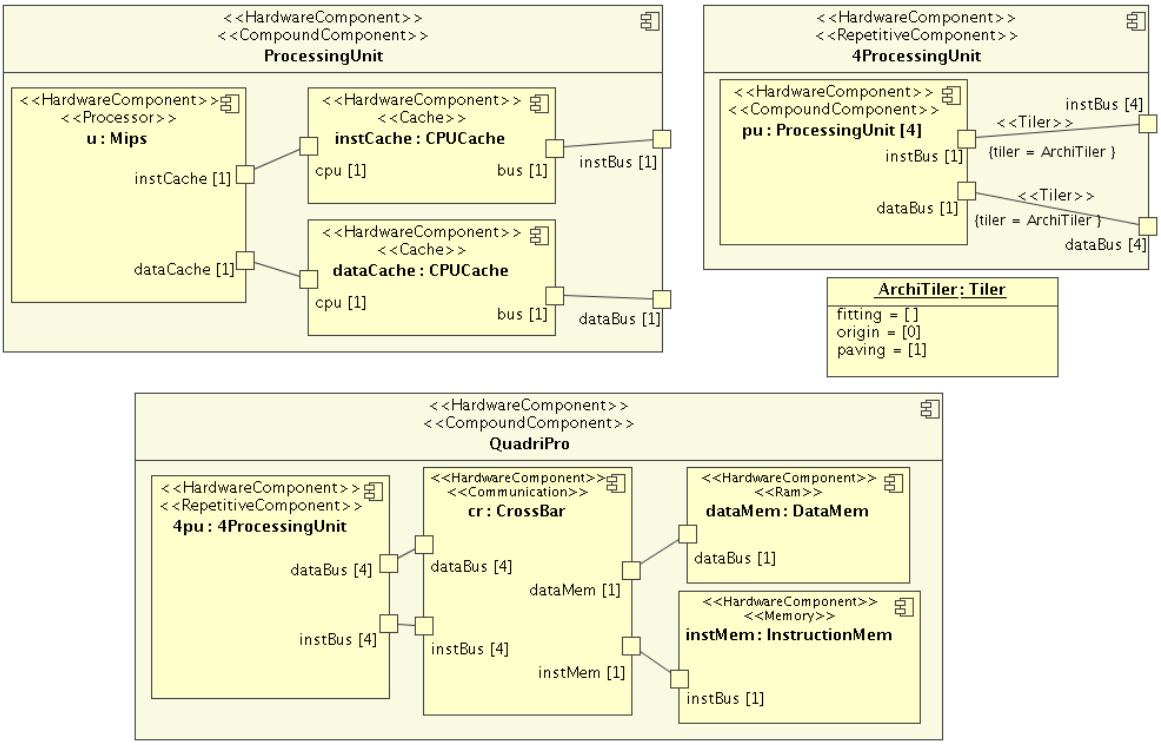


FIG. 8.18 – *Exemple d'une architecture « QuadriPro »*

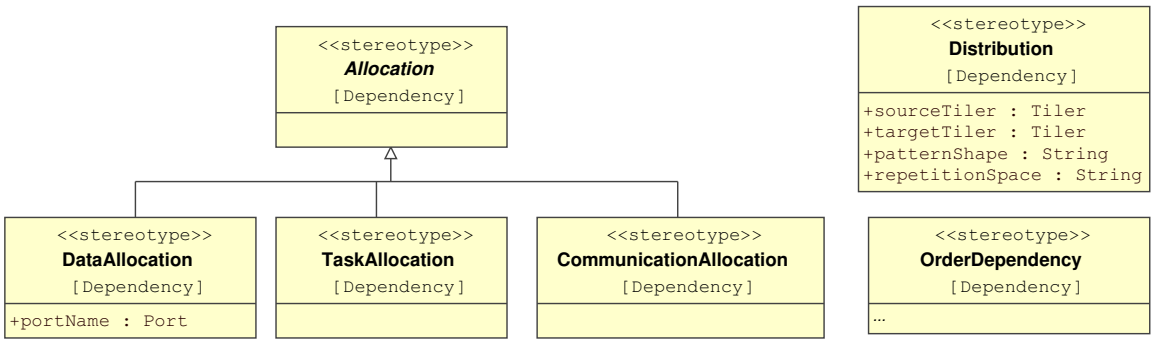


FIG. 8.19 – *Le package* `association`

Cependant, en UML2, la notion d'instance de `port` n'existe pas, ce qui oblige le placement des données, définies sur le même `port` et relatives à des instances différente du même composant, sur la même mémoire. Pour faire face à ce problème, nous avons associé au concept `DataAllocation` une `tagged value portName` qui permet la spécification de nom du `port` placé. Dans ce cas, le lien de dépendance `DataAllocation` sera tracé entre l'instance du composant applicatif et la mémoire comme il est montré par l'exemple de la figure 8.20. Dans cet exemple, les données définies sur le `port` inA sont placées sur la mémoire m1 : Memoire1 pour l'instance de composant a1 : A, et sur la mémoire m2 : Memoire2 pour l'instance de composant a2 : A. De façon similaire, les données définies sur le `port` outA sont placées sur la mémoire m2 : Memoire2 pour l'instance de composant a1 : A, et sur la mémoire

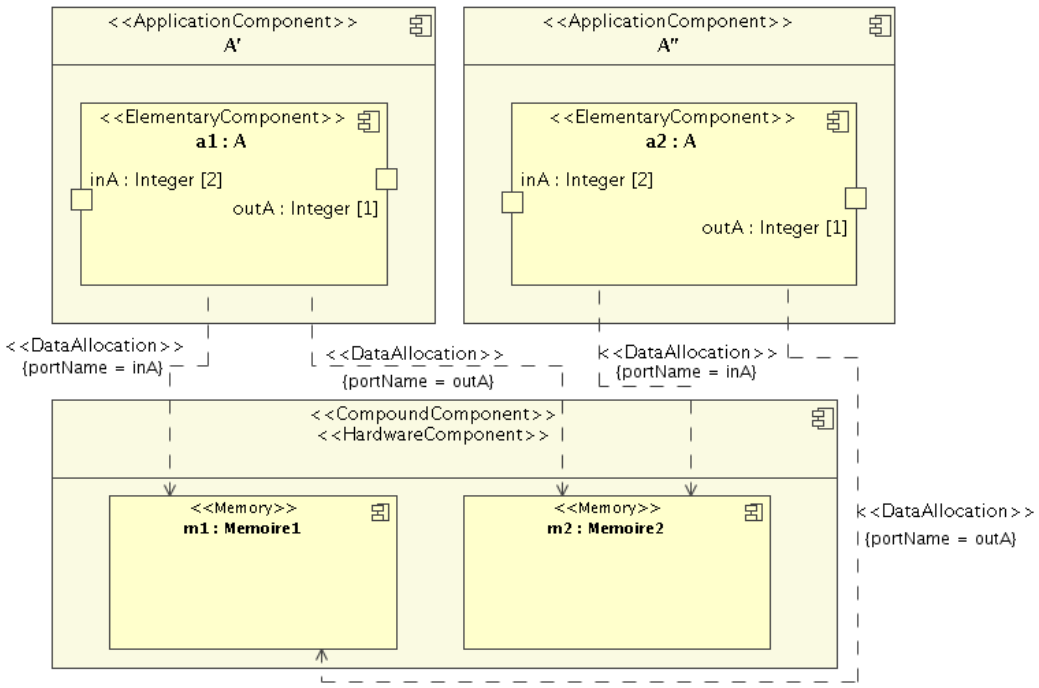


FIG. 8.20 – *Exemple d'allocation de données en utilisant* `DataAllocation`

m1 : Memoire1 pour l'instance de composant a2 : A. Cet exemple montre qu'il est possible de placer les mêmes données liées à des instances différentes de composants sur des mémoires différentes en utilisant le concept de `DataAllocation`.

Le concept de `TaskAllocation` est introduit pour définir le placement des composants applicatifs sur des processeurs matériels comme il est montré par l'exemple de la figure 8.21. Dans cet exemple, l'instance de tâche a : A est placée sur le processeur p1 : Processeur1, tandis que l'instance de tâche b : B est placée sur le processeur p2 : Processeur2.

Le concept de `CommunicationAllocation` est utilisé pour le placement de la communication des données sur des composants de communication matériels tel que les bus. Ce type de placement est uniquement utile lorsque les données des `ports` communiquant ne sont pas placées dans la même mémoire comme il est montré par l'exemple de la figure 8.22. Dans cet exemple, les données définies sur les `ports` outA et inB sont placées sur des mémoires différentes. Le lien entre ces `ports` doit être donc placé sur un dispositif de communication via le concept `CommunicationAllocation`.

Le concept d'`OrderDependency` étend la métaclasse `Dependency` d'UML2, et il est introduit pour la définition d'un ordre sur les liens de placement de plusieurs composants applicatifs sur le même composant matériel. La figure 8.23 représente un exemple qui montre l'utilité d'utilisation d'un lien de type `OrderDependency`. Dans cet exemple, les deux instances de tâche a1 : A et a2 : A sont indépendantes et peuvent être exécuter de manière concurrente. Le placement de ces deux instances de tâche sur le même processeur ne permet pas de savoir quelle instance de tâche sera exécutée en premier. Le concept d'`OrderDependency` peut donc être utilisé pour forcer un ordre d'exécution entre ces deux tâches. Dans l'exemple, l'instance de tâche a2 : A sera exécutée avant l'instance de tâche a1 : A.

Les trois concepts d'allocation présentés ci-dessus représente un type d'allocation simple

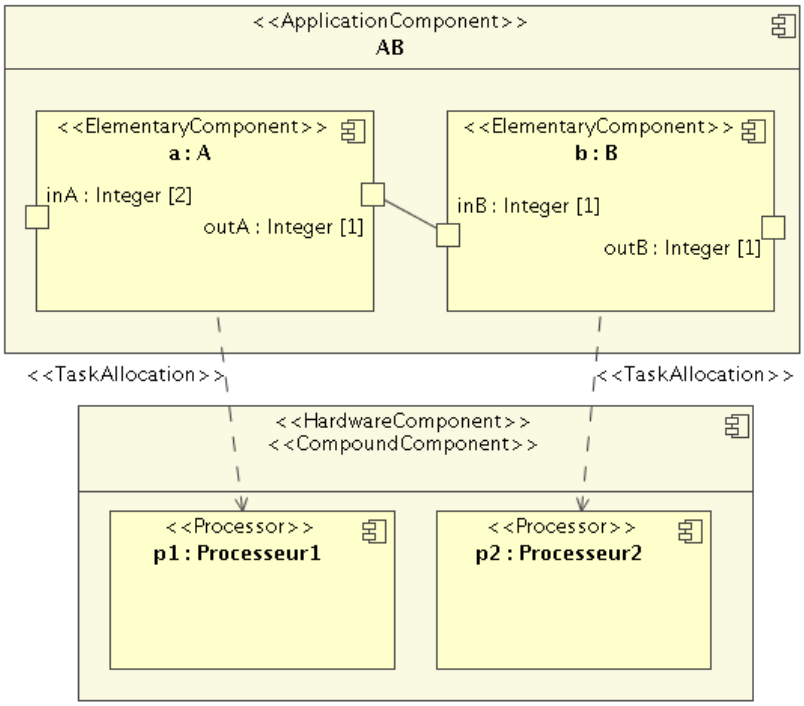


FIG. 8.21 – *Exemple d'allocation de tâche en utilisant* `TaskAllocation`

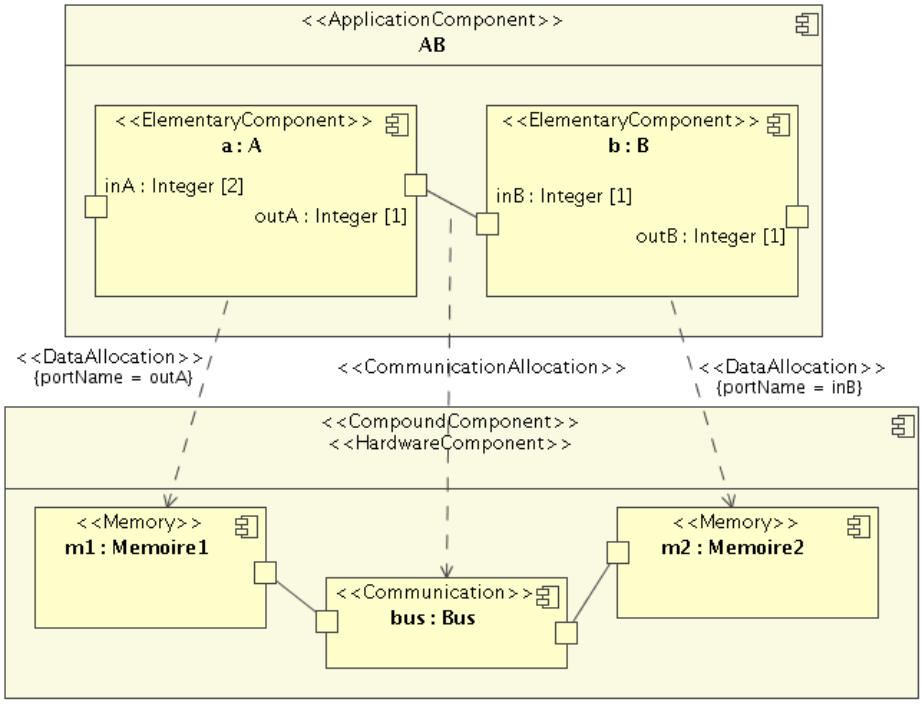


FIG. 8.22 – *Exemple d'allocation de communication en utilisant* `CommunicationAllocation`

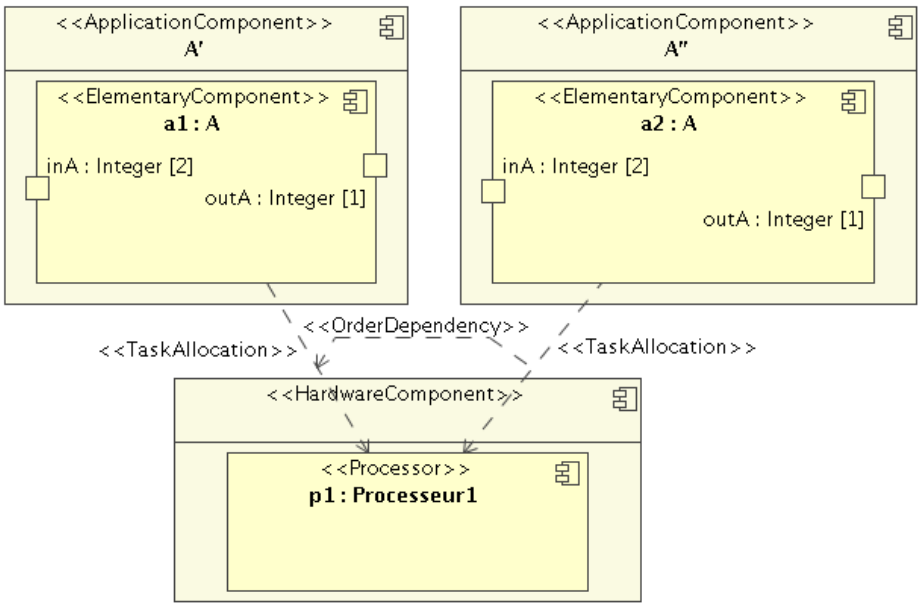


FIG. 8.23 – *Exemple d'utilisation du concept d'`OrderDependency`*

entre un composant applicatif et un composant matériel. Cependant, il est également possible de distribuer les différents éléments d'un composant applicatif répété sur les éléments d'un composant matériel également répété. Pour cette raison, nous avons introduit le concept de `Distribution` qui étend la métaclasse `Dependency` d'UML2. L'utilisation du concept de `Distribution` est basée sur les notions de répétition d'ARRAY-OL pour la description des différents liens entre les composants applicatifs et les composants matériels. Pour ce faire, nous avons associé à ce concept les `tagged values` : `sourceTiler`, `targetTiler`, `patternShape` et `repetitionSpace`. Les `tagged values sourceTiler` et `targetTiler` sont de type `Tiler` (défini dans le package `factorization`), et permettent de spécifier les informations de TILER liées respectivement à la partie source qui est généralement l'application et à la partie target qui représente généralement l'architecture. La `tagged value patternShape` permet de spécifier la forme du motif, tandis que la `tagged value repetitionSpace` permet de déterminer l'espace de répétition de la distribution. La figure 8.24 représente un exemple d'utilisation du concept `Distribution`.

Le concept de `Distribution` peut être uniquement appliqué sur des relations de dépendance de type `DataAllocation`, `TaskAllocation` ou `CommunicationAllocation` comme il est exprimé par la contrainte OCL :

inv *: self.stereotype → exists(name='`Allocation`')*

Une autre contrainte sur l'utilisation du `Distribution` est que le composant du côté application et celui du côté architecture doivent être de type répétitif comme il est exprimé par la contrainte OCL :

inv *: self.participant → size()=2* ***and*** *self.participant → forAll(stereotype → exists(name='`RepetitiveComponent`'))*

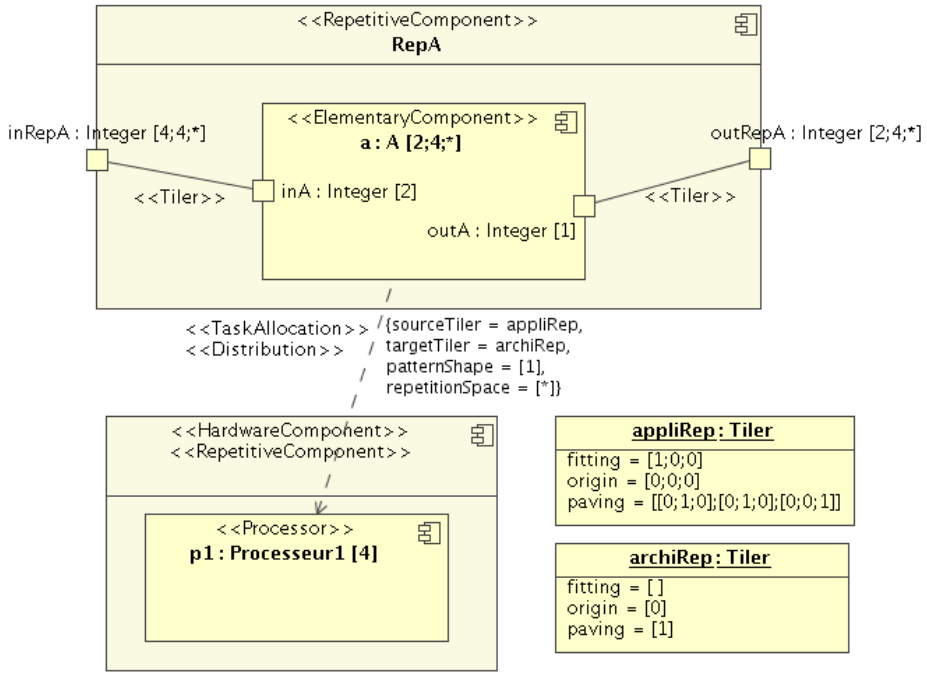


FIG. 8.24 – *Exemple d'utilisation du concept d'*`Distribution`

Nous avons présenté l'ensemble de concepts utilisés pour la modélisation à haut niveau d'abstraction des différents éléments dans les modèle GASPARD2. Ces concepts permettent la représentations des applications de traitement de données massivement parallèle en se basant sur les mécanismes de répétition du langage ARRAY-OL. Cependant, ils ne permettent pas la représentation des notions de contrôle et des changements de modes de fonctionnement pour ces applications. Le but de notre travail est donc de proposer une solution UML pour la modélisation des différents concepts liés à la notion de contrôle et leur introduction dans le profil GASPARD2. Dans ce qui suit, nous allons étudier la modélisation de l'automate de contrôle et du « switch » entre les différents modes de fonctionnement dans le profil GASPARD2 pour permettre la représentation d'une plus grande catégorie d'applications mixant des traitements de données parallèles et du contrôle.

8.3 Introduction du contrôle dans le profil GASPARD2

Dans cette section, nous allons décrire notre approche pour l'introduction des notions de contrôle dans le profil GASPARD2 [LDBR06] en se basant sur la méthodologie de séparation contrôle/données présentée dans le chapitre 5, et sur la notion de degré de granularité étudiée dans le chapitre 7. Nous rappelons que le principe d'introduction du contrôle dans les applications de traitement parallèle décrites en ARRAY-OL est basé sur la technologie synchrone, et il est principalement inspiré du concept des automates de modes permettant d'activer un seul mode de fonctionnement à la fois selon la valeur donnée par l'automate de contrôle.

Comme nous l'avons discuté dans le chapitre 7, La sémantique du langage de spécification ARRAY-OL, et par conséquent de son environnement de développement GASPARD2,

considère que les tableaux en entrée et ceux en sortie représentent une infinité d'éléments avec une dimension de temps triviale ou banalisée. Cette sémantique prend uniquement en considération la description des dépendances de données sans aucune représentation de la notion de flot. Cependant, la sémantique de flot est indispensable pour la modélisation des automates de contrôle qui, dans notre cas, génèrent un flot de modes en sortie en fonction des flots d'événements en entrée. Il est donc nécessaire de trouver une solution permettant d'introduire la notion de contrôle dans le modèle GASPARD2 tout en respectant sa sémantique de base. Dans ce contexte, la principale question à poser est : *comment modéliser l'automate de contrôle et le « switch » entre les différents modes de fonctionnement dans le profil* GASPARD2 *?*.

Dans ce qui suit, nous allons répondre à cette question en limitant notre étude à l'introduction du contrôle pour la partie application du modèle Y de GASPARD2. Les différents concepts utilisés pour la représentation des comportements de contrôle sont regroupés dans un nouveau package appelé `control`, et qui peut être utilisé par la partie `application` comme il est montré par la figure 8.25. L'objectif des différents concepts introduits dans le

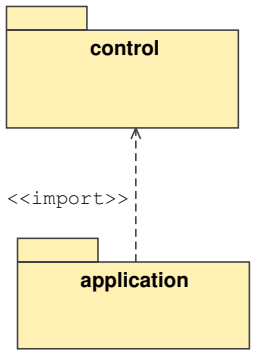


FIG. 8.25 – *Introduction du contrôle pour la partie* `application` *du profil* GASPARD2

package `control` est de permettre la modélisation de l'automate de contrôle et de la partie contrôlée tout en respectant la sémantique de base du profil GASPARD2.

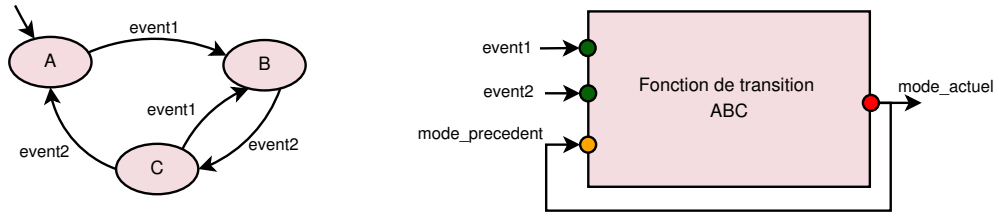
8.3.1 Modélisation de l'automate de contrôle

La structure d'automate de contrôle que nous proposons d'utiliser est inspirée de celle des automates de modes [MR98] et des automates de Moore [Moo56]. L'utilisation du principe des automates de modes permet de bien spécifier les différents modes de fonctionnement d'un système et les conditions de changement de modes, tandis que le principe des automates de Moore est utilisé pour limiter la sortie de l'automate à l'état dans lequel il se trouve, et qui représente le mode de fonctionnement pour l'application étudiée. Dans ce type d'automates, la relation de transition spécifie uniquement des événements permettant le passage d'un état à un autre sans avoir aucun effet sur le résultat en sortie de l'automate (à part son état). Dans ce contexte, nous considérons que l'automate de contrôle fournit en sortie uniquement un flot de valeurs de modes qui sera utilisé par la partie contrôlée pour choisir le mode de fonctionnement à activer parmi plusieurs exclusifs.

Comme nous l'avons introduit ci-dessus, la définition du comportement d'un automate de contrôle est basée sur une sémantique de flot qui n'existe pas dans la description des modèles dans GASPARD2. Pour cette raison, il est nécessaire de proposer une « codification »

de la notion de flot dans le profil GASPARD2 pour permettre, d'une part, de représenter la structure de l'automate de contrôle, et d'autre part, de respecter la sémantique de base des modèles dans GASPARD2.

Une solution possible consiste à modéliser l'automate de contrôle en spécifiant de façon explicite la relation de dépendance entre les différentes répétitions de sa fonction de transition. Dans ce contexte, nous considérons que l'automate de contrôle peut être représenté par une fonction de transition et une dépendance inter-répétition sur cette fonction comme il est montré par l'exemple de la figure 8.26. Dans cet exemple, la structure de l'automate est



(a) Structure de l'automate ABC

(b) Dépendance inter-répétition sur la fonction de transition ABC

FIG. 8.26 – *Représentation d'un automate de contrôle par une dépendance inter-répétition sur sa fonction de transition*

spécifiée par une répétition sur la fonction de transition ABC relative au comportement d'un cycle d'exécution de l'automate. Cette fonction prends en entrée la liste des événements en entrée (event1 et event2), et l'état précédent de l'automate (mode_precedent), et retourne en sortie la valeur de l'état actuel de l'automate (mode_actuel) qui représente le mode de fonctionnement de l'application, et sera utilisée dans le prochain cycle d'exécution comme étant l'état précédent de l'automate. Il est à noter qu'au premier cycle d'exécution, l'état précédent de l'automate représente son état initial (l'état A dans l'exemple).

Cette représentation de la structure d'automate est en complète adéquation avec le principe des modèles dans GASPARD2 qui sont basés sur la description des dépendances de données entre les différentes tâches du modèle. Ainsi, la relation de dépendance liée à la modélisation de l'automate de contrôle est relative à une dépendance de données particulière entre les différentes répétitions de la même instance de composant. C'est le concept de dépendance inter-répétition qui peut être facilement modélisée en utilisant le concept d'`InterRepetitionLinkTopology` introduit dans le package `factorization`. L'expression d'un tel lien de dépendance sur les différentes répétitions d'un automate permet de mémoriser son état précédent pour l'utiliser comme valeur en entrée pour le cycle suivant, et par conséquent, d'introduire la notion de flot, liée à la description d'un automate de contrôle, dans le profil GASPARD2 sans avoir à changer ou modifier sa sémantique de base.

Dans le cadre de la définition du profil GASPARD2, nous allons considérer que l'automate de contrôle est un composant particulier autour duquel est défini un ensemble de contraintes permettant de bien encadrer son contexte d'utilisation. La figure 8.27 représente les concepts introduits dans le package `control` pour la modélisation de l'automate de contrôle tout en respectant la sémantique des modèles GASPARD2.

Pour la modélisation d'un automate de contrôle dans un modèle GASPARD2, nous avons introduit le concept abstrait `AutomatonComponent` qui étend la métaclasse `Component` d'UML2. Comme nous l'avons discuté ci-dessus, la structure d'automate que nous introdui-

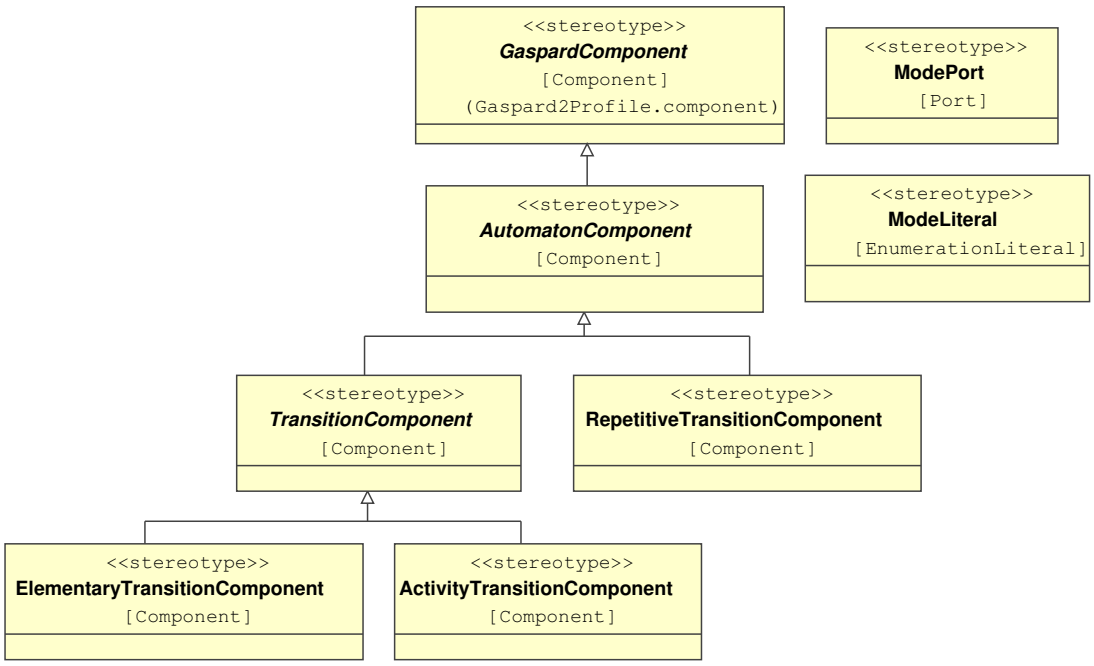


FIG. 8.27 – *Représentation des concepts du package* `control` *introduits pour la modélisation des automates de contrôle*

sons dans les modèles GASPARD2 est fortement inspirée de celles des automates de mode et des automates de Moore, et donne uniquement en sortie les valeurs de modes de fonctionnement pour l'application contrôlée. Pour spécifier ce comportement, nous avons introduit un type particulier de `port` dans GASPARD2 appelé `ModePort`. Le concept de `ModePort` étend la métaclasse `Port` d'UML2, et permet uniquement de faire circuler des valeurs de modes qui peuvent être énumérées sous forme de littéraux de type `ModeLiteral` comme il est exprimé par la contrainte OCL :

inv *: self.feature → exists(f | f.oclIsTypeof('ModeLiteral'))*

Cette contrainte représente un invariant qui doit assurer que tout `ModePort` est de type `ModeLiteral` (fait circuler des données de type `ModeLiteral`). L'objectif d'introduction du concept `ModeLiteral` est de faciliter la représentation et l'énumération des différentes valeurs de modes pour une application contrôlée.

En utilisant le concept de `ModePort`, nous pouvons définir une contrainte sur la structure d'un composant de type `AutomatonComponent` en imposant qu'un tel composant doit fournir en sortie uniquement des valeurs de mode comme il est spécifié par la contrainte OCL :

inv *: self.ownedPort.required → size()=1* ***and*** *self.ownedPort.required.stereotype → exists (name='ModePort')*

Cette contrainte représente un invariant qui permet d'assurer qu'un composant de type `AutomatonComponent` dispose d'un et un seul `port` en sortie de type `ModePort`.

Le concept d'`AutomatonComponent` peut être de deux types différents : `TransitionComponent` utilisé pour la spécification de la fonction de transition d'un

automate, et RepetitiveTransitionComponent utilisé pour la modélisation de la répétition autour de la fonction de transition de l'automate. Un composant de type RepetitiveTransitionComponent doit être donc défini autour d'un seul composant de type TransitionComponent comme il est exprimé par la contrainte OCL :

inv : self.part → size()=1 ***and*** *self.part.stereotype → exists(s | s.name='TransitionComponent')*

Cette contrainte représente un invariant qui assure que tout composant de type RepetitiveTransitionComponent doit contenir une et une seule part de type TransitionComponent. De façon similaire, pour qu'un composant de type TransitionComponent ait un sens d'utilisation, il doit être défini à l'intérieur d'un composant de type RepetitiveTransitionComponent comme il est exprimé par la contrainte OCL :

inv : self.owner.stereotype → exists(name='RepetiveTransitionComponent') ***and*** *self.owner → size()>0*

Cette contrainte représente un invariant qui assure qu'une part de type TransitionComponent doit être uniquement définie à l'intérieur d'un composant de type RepetiveTransitionComponent.

Ainsi, pour la description de la dépendance entre les différentes répétitions de la fonction de transition, il est indispensable d'associer un et un seul lien de type InterRepetitionLinkTopology à tout composant de type TransitionComponent comme il est exprimé par la contrainte OCL :

inv : self.connector → select(stereotype → exists(name='InterRepetitionLinkTopology)→ size()=1)

Cette contrainte représente un invariant qui doit assurer la définition d'un lien de type InterRepetitionLinkTopology sur tout composant de type TransitionComponent. Nous pouvons également imposer que la valeur de la tagged value repetitionSpaceDependence liée au concept d'InterRepetitionLinkTopology doit être égale à 1 (prendre toujours la valeur de l'état précédent).

La description de la fonction de transition peut être réalisée de deux façons différentes : en utilisant un composant de type ElementaryTransitionComponent ou en utilisant un composant de type ActivityTransitionComponent. Dans le cas d'un composant de type ElementaryTransitionComponent, le comportement de la fonction de transition est vu comme une boite noire qui ne doit contenir aucune spécification d'un composant de type GaspardComponent comme il est exprimé par la contrainte OCL :

inv : self.part → select(stereotype → exists(name=GaspardComponent)) → size()=0

Dans le cas d'un composant de type ActivityTransitionComponent, le comportement de la fonction de transition doit être représenté par un diagramme d'activités en UML2 comme il est exprimé par la contrainte OCL :

inv : self.feature → forAll(oclIsKindof(Activity)) ***and*** *self.part → size()=0*

La figure 8.28 représente un exemple de modélisation de l'automate de contrôle de la figure 8.26 en utilisant les concepts introduits dans la package control. Dans cet exemple, la fonction de transition est de type ElementaryTransitionComponent dont le comportement

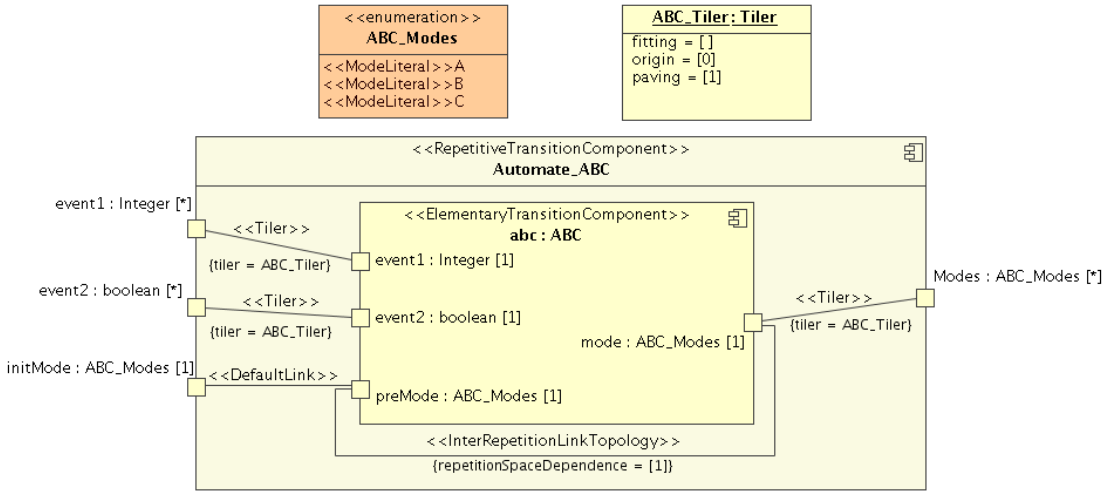


FIG. 8.28 – *Représentation d'un automate de contrôle en utilisant les concepts du package* `control`

peut être défini dans le langage source comme dans le cas de composants GASPARD2 de type `Elementarycomponent`. Il est également possible de définir la même fonction de transition en utilisant le concept d'`ActivityTransitionComponent`. Dans ce cas, le comportement de la fonction de transition doit être spécifié par un diagramme d'activités comme il est montré par l'exemple de la figure 8.29.

De façon générale, nous avons défini un ensemble de concepts permettant la modélisation de la structure d'automate par une fonction de transition et une dépendance inter-répétition sur cette fonction. La fonction de transition est spécifiée par une tâche qui prend en entrée la liste des événements et l'état précédent de l'automate, et retourne en sortie son état actuel qui sera utilisé comme une entrée dans le prochain cycle d'exécution de l'automate. La relation de dépendance inter-répétition permet d'exprimer la notion de flot sur les différentes répétitions de la fonction de transition dans l'automate, et par conséquent, d'introduire cette notion de flot dans le modèle GASPARD2 en spécifiant une relation de dépendance sur l'espace temporel. Ceci est généralement relatif au codage classique d'un automate de contrôle dans un langage purement flot de données.

8.3.2 Modélisation des différents modes de fonctionnement

La valeur de mode en sortie fournie par l'automate de contrôle est utilisée par la partie de l'application contrôlée par cet automate pour déterminer le mode de fonctionnement à activer. La partie contrôlée de l'application regroupe donc les différents modes de fonctionnement possibles pour son exécution, et doit être capable d'activer le bon mode parmi plusieurs exclusifs en fonction de la valeur fournie par l'automate.

Comme nous l'avons introduit dans le chapitre 5, à chaque mode de fonctionnement du système correspond un composant de calcul avec un certain nombre d'entrées et de sorties. Pour simplifier la manipulation et la réutilisation des différents modes du système, ces composants de calcul doivent avoir la *même interface*. En d'autres termes, les modes de fonctionnement d'un composant contrôlé doivent avoir, à un certain niveau de hiérarchie, le même nombre et le même type d'entrées et de sorties comme il est montré par l'exemple de

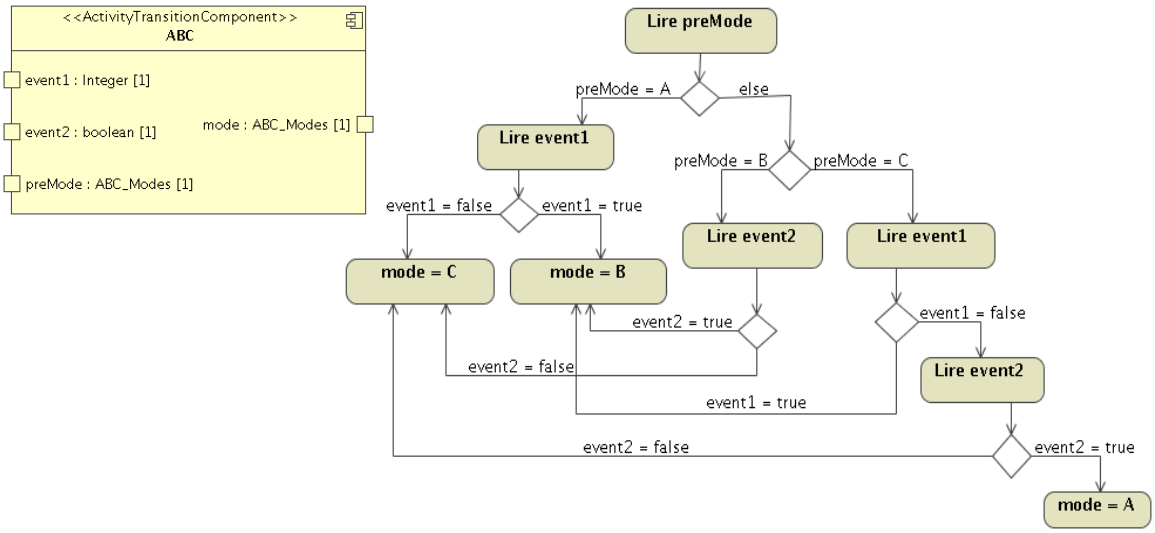


FIG. 8.29 – *Représentation de la fonction de transition* ABC *par un diagramme d'activités*

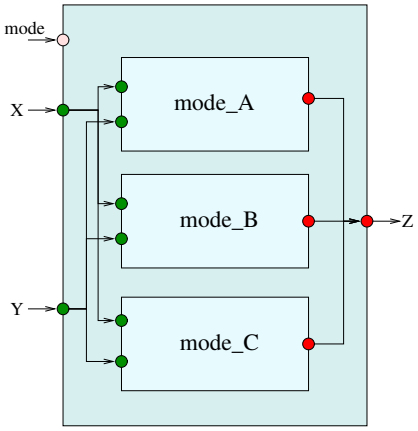


FIG. 8.30 – *Représentation des modes de fonctionnement avec une interface unique*

la figure 8.30.

La partie contrôlée représente donc un composant particulier permettant la modélisation du comportement d'un « switch » entre les différents modes de l'application. Pour la représentation d'un tel comportement, nous avons introduit un ensemble de concepts dans le package `control` du profil GASPARD2 comme il est montré par la figure 8.31.

Le concept de `ControlledComponent` étend la métaclasse `Component` d'UML2, et permet de définir un type particulier de composant dans GASPARD2 utilisé pour la représentation de la partie contrôlée de l'application. L'objectif d'un composant de type `ControlledComponent` est de regrouper les différents modes de fonctionnement liés à son exécution. Pour la représentation des différents modes de fonctionnement, nous avons introduit le concept d'`AlternativeComponentPart` qui s'applique uniquement sur les `parts` d'UML2. À chaque `part` de type `AlternativeComponentPart` est liée une `tagged value activationCondition` de type `ModeLiteral`, et qui permet de spécifier la condition d'activation de la `part` en ques-

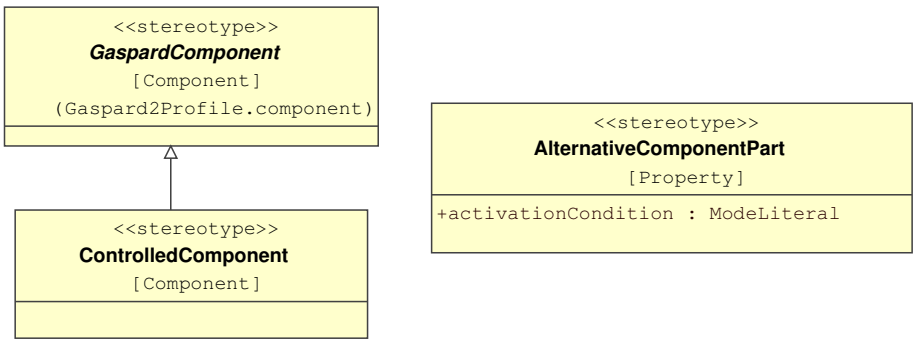


FIG. 8.31 – *Représentation des concepts du package* `control` *introduits pour la modélisation de la partie contrôlée*

tion.

Un composant de type `ControlledComponent` ne doit donc contenir que des `parts` de type `AlternativeComponentPart` comme il est spécifié par la contrainte OCL :

inv *: self.part → forAll(stereotype → exists(name='`**`AlternativeComponentPart`**`'))*

Cette contrainte représente un invariant qui impose que toutes les `parts` dans un composant de type `ControlledComponent` doivent être stéréotypées `AlternativeComponentPart`. Les différente `parts` dans un composant contrôlé doivent également avoir la même interface pour faciliter leur manipulation et réutilisation comme il est exprimé par la contrainte OCL :

inv *: self.part → forAll(p1,p2 | (p1.ownedPort.provided=p2.ownedPort.provided)* ***and*** *(p1.ownedPort.required=p2.ownedPort.required))*

Cette contrainte représente un invariant qui impose que toutes les `parts` dans un composant de type `ControlledComponent` doivent avoir les mêmes `ports` en entrée et en sortie (même type et même multiplicité). Ainsi, puisque les différentes `parts` dans un composant contrôlé représentent les différents modes de fonctionnement de la partie contrôlée de l'application, et qui sont par nature exclusifs, les conditions d'activation des différentes `parts` doivent être également exclusives comme il est exprimé par la contrainte OCL :

inv *: self.part → select(taggedValue.name=*`activationCondition` ***implies*** *forAll(p1,p2 | p1.taggedValue.value<>p2.taggedValue.value))*

Cette contrainte représente un invariant qui assure que les différentes `parts` dans un composant de type `ControlledComponent` sont exclusives dans le sens où elles ne peuvent pas être activées en même temps. Le concept d'`AlternativeComponentPart` permet alors d'associer un comportement particulier aux `parts` d'un composant GASPARD2 en spécifiant que la `part` peut être prise en compte ou exécutée si et seulement si sa valeur d'`activationCondition` est égale à celle du mode de fonctionnement fournie par l'automate de contrôle. Le « switch » entre les différentes `parts` représentant les modes de fonctionnement doit être assuré par le composant `ControlledComponent` selon la valeur de mode fournie par l'automate de contrôle comme il est exprimé par la contrainte OCL :

inv *: self → includes(p :part | p.activationCondition.value=ownedPort.provided → select(stereotype → exists(name=ModePort)).value)* ***and*** *self→ excludes(p :part | p.activationCondition.value<>ownedPort.provided → select(stereotype → exists(name=ModePort)).value)*

Cette contrainte représente un invariant qui spécifie qu'un composant de type `Controlled-Component` doit prendre en considération uniquement la `part` dont la valeur de sa condition d'activation est égale à celle de son `port` de type `ModePort`, et ignorer toutes les autres `parts`.

La figure 8.32 représente un exemple simple de la modélisation de la partie contrôlée dans une application. Dans cet exemple, l'application est composée de trois modes de fonc-

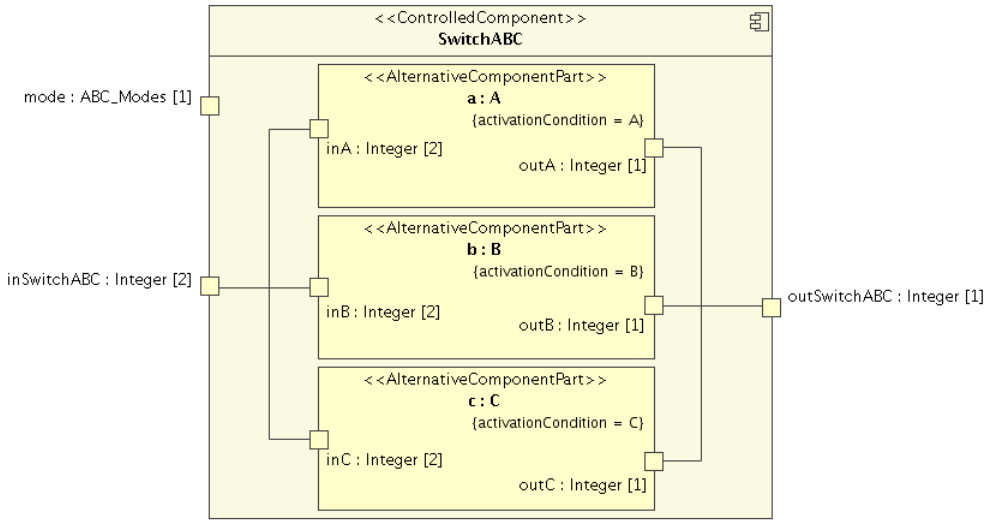


FIG. 8.32 – *Exemple de modélisation de la partie contrôlée*

tionnement A, B et C. L'activation de ces modes est faite en fonction de la valeur de mode et selon la condition d'activation de la `part` du mode en question. Cet exemple montre également que la sémantique du composant de type `ControlledComponent` est différente de celle des autres composants dans GASPARD2 dans le sens où c'est le seul type de composants qui permet de connecter un `port` en entrée à plusieurs `ports` d'entrée de ses `parts`, et de connecter plusieurs `ports` de sortie des différentes `parts` à son même `port` de sortie. Ce comportement est similaire à celui utilisant des opérateurs de type `Fork` et `Join`. Dans ce contexte, il ne va pas y avoir des conflits de connexion puisque nous considérons que le comportement d'un composant de type `ControlledComponent` se résume à celui de la seule `part` active de ce composant, et par conséquent, les liens relatifs aux autres `parts` sont ignorés.

8.3.3 Modélisation du lien entre l'automate de contrôle et la partie contrôlée

La communication entre l'automate de contrôle et la partie contrôlée se fait via la valeur de mode fournie par l'automate. En fonction de cette valeur, la partie contrôlée peut choisir le bon mode de fonctionnement à activer. Dans ce contexte, la partie contrôlée doit absolument avoir un et un seul `port` de type `ModePort` sur lequel elle reçoit les différentes valeurs de modes comme il est exprimé par la contrainte OCL :

inv *: self.ownedPort.provided → select(stereotype → exists(name='ModePort') → size()=1)*

Cette contrainte représente un invariant qui permet d'assurer que tout composant de type `ControlledComponent` doit avoir un seul `port` en entrée de type `ModePort`. La particularité de ce `port` est qu'il n'est jamais connecté, et sa valeur est uniquement utilisée pour la détermination du mode de fonctionnement à exécuter.

Comme nous l'avons discuté dans le chapitre 7, la partie contrôlée doit suivre le rythme de fonctionnement de l'automate de contrôle en fonction du comportement de l'application à vouloir modéliser. Dans ce contexte, la notion de degré de granularité est utilisée pour déterminer le lien entre l'automate de contrôle et la partie contrôlée via les différents instants de prise en compte des valeurs d'événements.

En se basant sur la notion de degrés de granularité et en utilisant les différents concepts introduits dans le package `control`, il est donc possible de modéliser selon le profil GASPARD2 différents comportements d'applications mixant des traitements de données parallèles à la ARRAY-OL et du contrôle. Pour illustrer ce concept, nous proposons de modéliser l'exemple de l'application définie par la figure 7.12 (page 143). Nous rappelons que cette application prend en entrée une infinité d'images de taille 4×4 et retourne en sortie une infinité d'images de taille 2×4. Son comportement est relatif à la répétition d'une tâche composée AB, où la tâche CA est une tâche contrôlée avec un degré de granularité relatif à un seul point de l'image en sortie. Le modèle UML lié à cette application selon le profil GASPARD2 est donné par la figure 8.33. Dans cet exemple, la fonction de transition de l'automate de contrôle et la partie contrôlée sont représentés dans le même composant répétitif. Nous remarquons également que le lien de dépendance inter-répétition relatif à l'automate de contrôle est défini sur un plus haut niveau de hiérarchie (sur l'instance de composant) puisque ce type de lien doit être uniquement défini sur le composant autour duquel est défini la répétition (nous rappelons qu'un lien de type `InterRepetitionLinkTopology` doit être uniquement défini à l'intérieur d'un composant de type `RepetitiveComponent`).

La représentation de la fonction de transition et de la partie contrôlée dans le même composant répétitif peut être intéressante dans certains cas de modélisation liés par exemple à la réutilisation de certains composants modélisés de cette façon. Cependant, cette représentation ne respecte pas complètement notre méthodologie de séparation contrôle données puisque l'automate de contrôle n'est pas complètement isolé de la partie calcul. Pour éviter ce problème, la modélisation séparée de l'automate de contrôle et de la répétition de la partie contrôlée est également possible comme il est montré par l'exemple de la figure 8.34. Cet exemple décrit le même comportement de l'application modélisée par la figure 8.33. Ainsi, l'utilisateur a complètement la liberté de choisir la façon de modélisation la plus appropriée pour son application.

De façon générale, nous remarquons que les différents concepts introduits dans la package `control` permettent la modélisation de plusieurs comportements mixant des traitements de données et du contrôle en se basant sur la notion de degré de granularité pour la définition des différents instants de prise en compte des valeurs d'événement. Ces concepts de modélisation respectent également notre approche de séparation contrôle/données et offrent aux utilisateurs différents moyens pour décrire plusieurs scénarios, basés sur des événement de contrôle interne ou externe, et sur la notion degré de granularité simple ou multiple, sans se soucier de leur faisabilité réel.

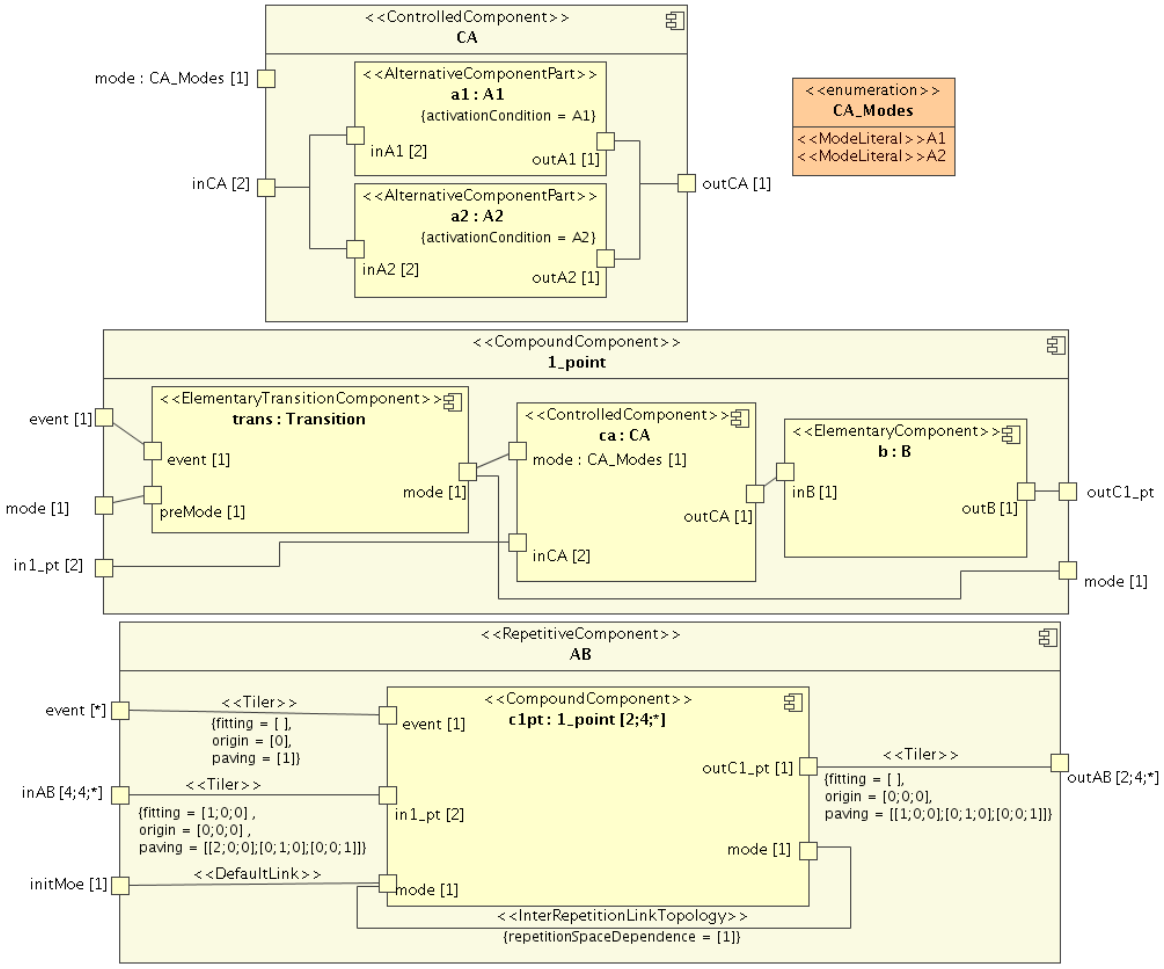


FIG. 8.33 – *Représentation de la fonction de transition et de la partie contrôlée dans le même composant répétitif*

8.4 Synthèse et conclusion

Nous avons présenté dans ce chapitre les différents concepts introduits dans le profil GASPARD2 pour la modélisation au plus haut niveau d'abstraction des applications de traitement parallèle et intensif pour les systèmes sur puce. Après avoir étudié les différents concepts du profil sans contrôle et leurs contraintes d'utilisation, nous avons consacré notre étude à la modélisation de l'automate de contrôle et le « switch » entre les différents modes de fonctionnement pour la partie application dans le profil GASPARD2.

L'introduction du contrôle dans le modèle d'application est basée sur la technologie synchrone, et suppose que, pour pouvoir lancer l'exécution d'une tâche, les valeurs de modes fournies par l'automate de contrôle doivent être présentes en même temps que celles des données. Dans ce contexte, le rythme de fonctionnement de la partie contrôlée d'une application doit suivre celui de son automate de contrôle.

De façon générale, le fonctionnement d'un automate de contrôle est basé sur une sémantique de flot, où à chaque cycle d'exécution, l'évolution de l'automate dépend des valeurs des événements et de son état courant. La modélisation telle quelle d'un automate sous

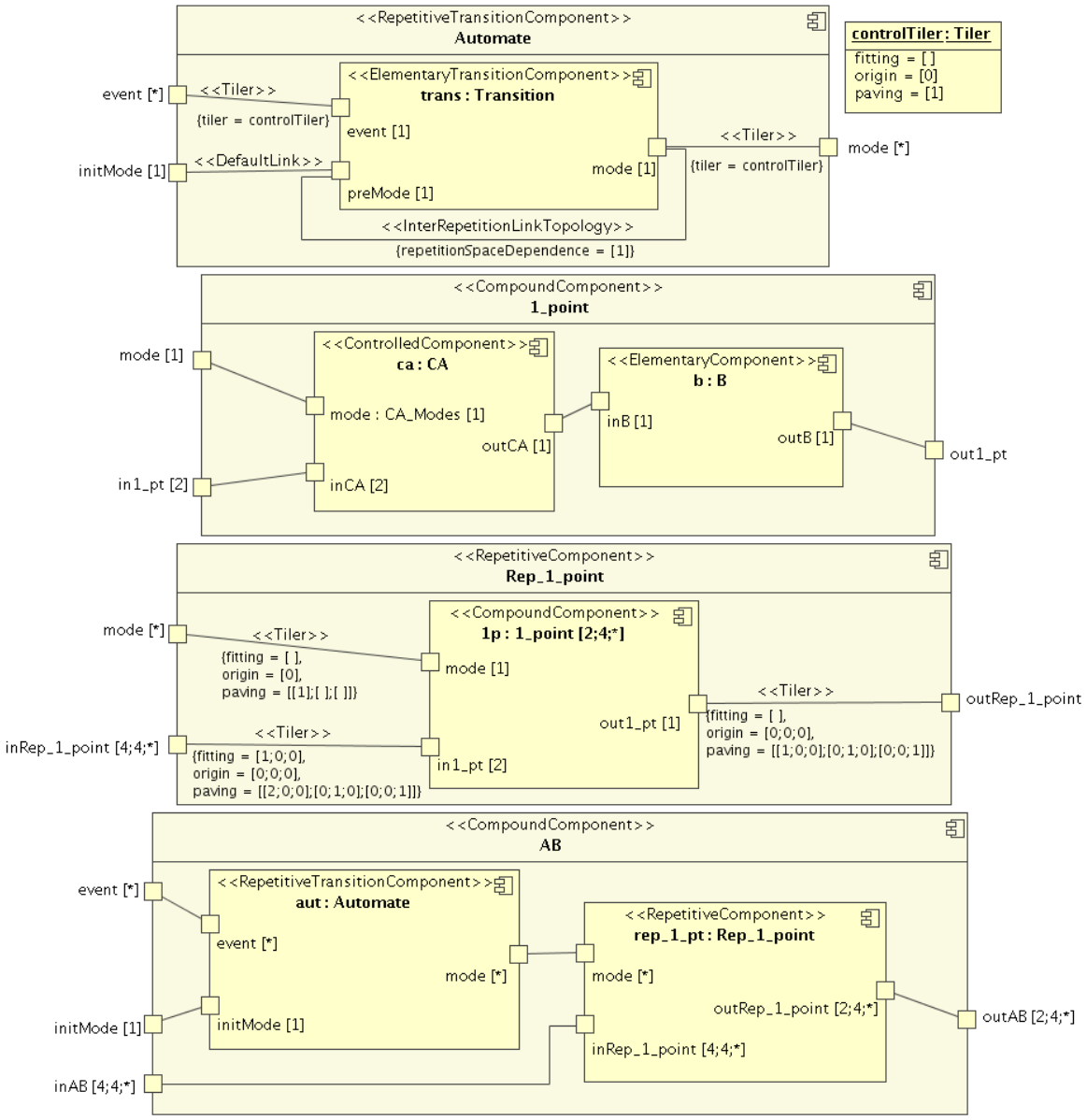


FIG. 8.34 – *Représentation de la fonction de transition et de la partie contrôlée dans des composants répétitifs différents*

forme d'états et de transitions est impossible dans l'environnement GASPARD2. Ce dernier est basé sur une sémantique de temps banalisé manipulant des tableaux de taille infinie avec des dimensions qui représentent de façon indifférente le temps et l'espace. Pour cette raison, nous avons proposé une codification de l'automate sous forme d'une fonction de transition et une dépendance explicite entre ses différentes répétitions. Cette représentation permet la modélisation d'un automate de contrôle, et d'introduire une notion de flot dans les modèles GASPARD2 sans modifier leur sémantique de base. Nous avons également proposé des concepts pour la représentation des différents modes de fonctionnement, et le lien entre l'automate de contrôle et la partie contrôlée.

L'introduction du contrôle dans le profil GASPARD2 est principalement basée sur notre méthodologie de séparation contrôle/données, et sur la notion de degrés de granularité utilisée pour la définition des différents instants de prise en compte des valeurs de contrôle. Le profil proposé offre un moyen efficace et complet pour la modélisation à haut niveau d'abstraction de plusieurs comportements mixant des traitements de données parallèles et du contrôle.

Chapitre 9

Étude de cas : traitement vidéo à multi-modes de fonctionnement

Dans ce chapitre, nous allons étudier la modélisation d'une application de traitement de vidéos à multi-modes de fonctionnement. L'objectif de cette étude est d'illustrer l'utilisation des différents concepts introduits dans le profil GASPARD2, notamment en ce qui concerne la modélisation du comportement réactif des applications de traitement de données massivement parallèle. L'application étudiée représente un cas simplifié des applications de traitement de vidéos réelles dans le sens où nous nous intéressons pas aux détails techniques ni aux algorithmes d'optimisation ou de traitement d'images. Ce chapitre représente donc un exemple de modélisation des parties de traitement parallèle, des parties de contrôle et des changements de modes de fonctionnement. Il offre aux développeurs un exemple de référence dont ils peuvent s'inspirer pour la modélisation et la réalisation de leur applications réelles mixant des traitements de données parallèles et du contrôle.

9.1 Introduction

Les applications de traitement d'image et de vidéo sont de plus en plus présentes dans notre vie quotidienne. La particularité de ces applications est qu'elles opèrent sur de grandes masses de données et font souvent appel aux processus de calcul parallèle à haute performance.

La qualité des images traitées et le nombre de services proposés par ces applications augmentent significativement grâce aux techniques de traitement de signal intensif et parallèles. Les téléviseurs de nos jours par exemple proposent de plus en plus des fonctionnalités permettant de répondre aux différents besoins des utilisateurs tels que l'affichage en plusieurs fenêtres, la mémorisation des séquences vidéos, etc. L'objectif des développeurs de ces systèmes est donc d'offrir plus de fonctionnalités aux utilisateurs, d'augmenter les performances de leurs applications en assurant leur qualité de service, et de réduire leurs coûts de développement.

Comme nous l'avons discuté dans la section 2.5 (page 30), les applications de traitement d'image et de la vidéo les plus utilisées contiennent dans leur description des comportements de contrôle. Ce type de comportement doit être pris en considération au plus haut niveau dans la chaîne de conception de ces applications pour assurer leurs fonctionnalités. Dans ce qui suit, nous allons présenter une étude de cas d'un système de traitement de vidéo à multi-modes de fonctionnement (MMVP : Multi-Modes Video Processing). Cette application permet à chaque téléspectateur de contrôler directement les modes d'affichage de son téléviseur. La description du comportement de l'application MMVP représente un mélange de traitement de données parallèle et du contrôle. Le but de cette étude est d'illustrer l'utilisation des concepts du profil GASPARD2, présenté dans le chapitre 7, pour la modélisation du contrôle et des différents modes de fonctionnement des applications parallèles et réactives.

Dans cette étude, nous ne nous intéressons pas aux détails techniques de l'application tels que les algorithmes de mise à l'échelle, de compression, de décompression et d'entrelacement. Ces derniers seront considérés comme des boites noires qui peuvent être remplacées par les algorithmes les plus adéquats selon le choix et les besoins des utilisateurs. Les problèmes de sélection et du choix des meilleurs algorithmes pour ce genre de problème ont été largement étudiés par la communauté de traitement d'image et de la vidéo et ne font donc pas partie de notre étude. Il est également à noter que l'application étudiée respecte bien notre méthodologie de séparation contrôle/données, présentée dans le chapitre 5, permet-

tant de faciliter l'étude séparée des différentes partie de cette application et de tirer profit des avantages de la méthodologie de séparation.

9.2 Description de l'application

Dans cette section, nous allons décrire informellement une application de traitement de vidéo à multi-modes de fonctionnement que nous appelons MMVP (Multi-Modes Vides Processing). L'objectif de base de cette application est d'illustrer notre approche d'introduction du contrôle dans les applications de traitement de données parallèles et leur modélisation selon les concepts du profil GASPARD2. Il est à noter que l'application MMVP est principalement inspirée de celle présentée par M. Schu *et al.* pour décrire le comportement d'une application de traitement de vidéo intégrée sur un système sur puce [SWT+01]. Nous avons apporté quelques simplifications, ajouts et modifications à la description de l'application de base pour faciliter sa compréhension et l'adapter à nos besoins de modélisation en faisant abstraction des détails techniques.

Le but principal de l'application MMVP est de permettre l'affichage en plusieurs modes de fonctionnement des images vidéo sur un écran de télévision en se basant sur des algorithmes d'optimisation permettant d'améliorer les performances de l'affichage, et de répondre aux différents besoins des utilisateurs. Dans ce qui suit, nous allons considérer que l'affichage des images vidéo se fait sur un écran de résolution 768 × 576 pixels, relative aux standards européens PAL (Phase Alternated Line), SÉCAM (SÉquentiel Couleur À Mémoire) et NTSC-4.43 (National Television System Committee). Nous considérons également que le système est capable de prendre en considération quatre vidéos différentes (quatre chaînes de télévision) où les images de chaque vidéo sont constituées de 1920 × 1080 pixels comme il est montré par la figure 9.1.

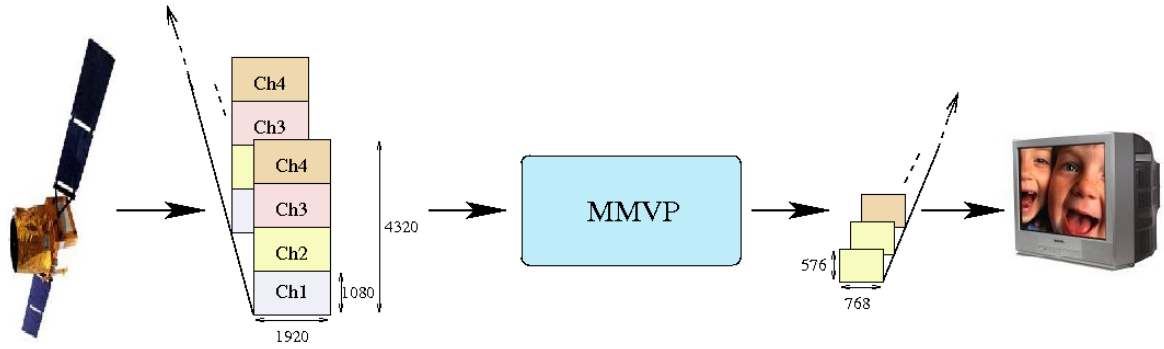


FIG. 9.1 – *Schéma global de l'application* MMVP

De façon générale, l'application MMVP peut fonctionner en six modes différents[63] : `FSM` (Full Screen Mode), `PIP` (Picture In Picture mode), `SSC` (Split SCreen mode), `MUP` (MUlti Picture mode), `SPS` (SnaPShot mode) et `DSPS` (Display SnaPShot mode). À l'initialisation, le système est en mode `FSM`, l'activation, la désactivation et le passage entre les différents modes se font via un ensemble de boutons d'une télécommande : `Full`, `PiP`, `Split`, `Multi`, `Snap` et `DSnap` comme il est montré par l'automate de contrôle de la figure 9.2.

[63]Les noms des modes de fonctionnement sont ceux utilisés par M. Schu *et al.* pour la description de leur application [SWT+01].

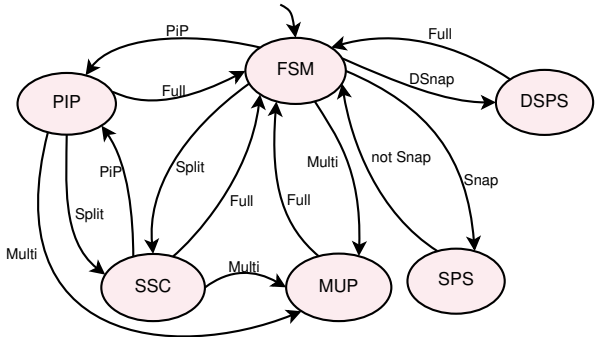


FIG. 9.2 – *Les différents modes de fonctionnement de l'application* MMVP

La description de l'application MMVP contient donc un mélange de traitement de données massivement parallèle et du contrôle. Pour la modélisation de ce comportement, nous allons utiliser le concept de degré de granularité présenté dans le chapitre 7. Dans ce contexte, le degré de granularité choisi pour la description du comportement de l'application MMVP est relatif au traitement d'une seule image en sortie de taille 768×576 pixels. Ce comportement peut être réalisé en associant à toute image en entrée de taille 1920×4320 une seule valeur de mode comme il est expliqué par la figure 9.3. Dans ce qui suit, nous allons

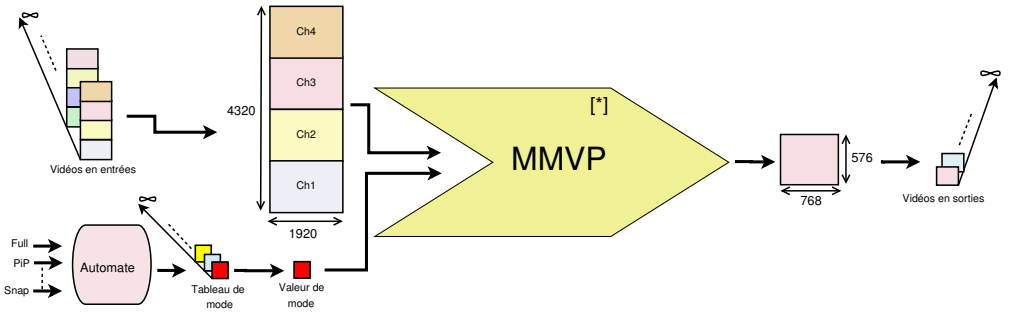


FIG. 9.3 – *Schéma global de l'application* `MMVP` *avec un degré de granularité de dimension* $[768, 576]$

présenter plus en détails le fonctionnement des différents modes de l'application MMVP, et les conditions de passage entre les modes. Il est à noter que l'application présentée définit un cas particulier et simplifié d'un système réel de traitement de vidéo.

9.2.1 Le mode `FSM`

Le mode de fonctionnement `FSM` (Full Screen Mode) est relatif au cas le plus répandu qui permet l'affichage en plein écran d'une séquence vidéo comme il est montré par l'exemple de la figure 9.4. La réalisation de ce processus nécessite une mise à l'échelle via une application de type DownScaler [LLLK00]. Cette application consiste principalement à l'application d'un filtre horizontal et d'un filtre vertical sur les images en entrée pour la construction des images en sortie. Dans le cas de l'application MMVP, les images en entrée sont de taille 1920×1080 pixels et celles en sortie sont de taille 768×576 pixels. Nous avons donc un facteur d'échelle horizontal de $5/2$, et un facteur d'échelle vertical de $15/8$. Pour l'application du filtre horizontal, 11 pixels de l'image en entrée sont nécessaires pour le calcul des

FIG. 9.4 – *Exemple d'un affichage en mode* `FSM`

2 pixels de l'image en sortie. Ce processus correspond à une première étape qui consiste à calculer la moyenne des 6 pixels adjacents à chaque 5 pixels en entrée. Par la suite, 2 pixels sont sélectionnés en fonction d'un ensemble de coefficients et critères bien définis par la communauté de traitement d'image et de la vidéo. De façon similaire, l'application du filtre vertical nécessite 21 pixels de l'image en entrée pour la génération de 8 pixels de l'image en sortie. Le principe de fonctionnement de cette application de mise à l'échelle est illustré par la figure 9.5.

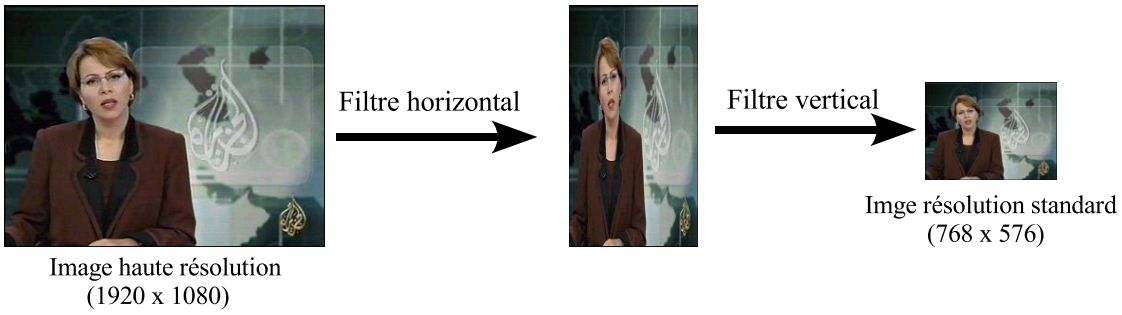


FIG. 9.5 – *Principe du DownScaler*

Le processus de mise à l'échelle est relatif au traitement des images d'une seule vidéo particulière. Cependant, dans le cas de l'application MMVP, nous avons quatre vidéos différentes et il est donc nécessaire de savoir à quelle vidéo va correspondre l'image en sortie. Pour ce faire, nous introduisons dans le mode `FSM` une partie de prétraitement, appelée `Select_Video`, permettant de sélectionner l'image de la vidéo correspondante. Dans ce contexte, l'application `Select_Video` peut fonctionner en quatre sous-modes différents[64] : `Ch1`, `Ch2`, `Ch3` et `Ch4`. À l'initialisation (lorsque la télévision est allumée), l'application est en mode `Ch1`, la navigation entre les différents sous-modes de `Select_Video` peut se faire via les boutons `Left` et `Right` de la télécommande comme il est montré par l'automate de contrôle de la figure 9.6. Ainsi, dans le cas où l'utilisateur n'appuie pas sur les boutons `Left` ou `Right`, le système doit être capable de garder trace de la vidéo en cours (le mode fournit

[64]Cette application peut être plus facilement réalisable s'il est possible de décrire les changements des TILERs.

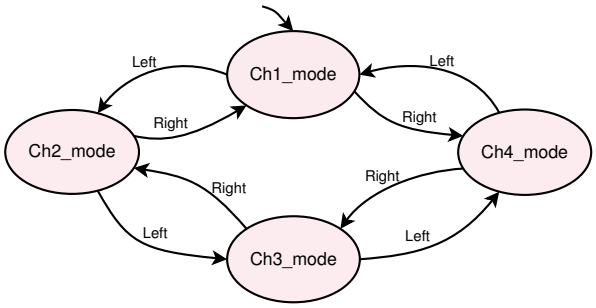


FIG. 9.6 – *Les différents sous-modes de fonctionnement du mode* `FSM`

par `Select_Video`) pour continuer l'affichage des différentes images de cette même vidéo. La figure 9.7 représente le schéma global de fonctionnement du mode `FSM`.

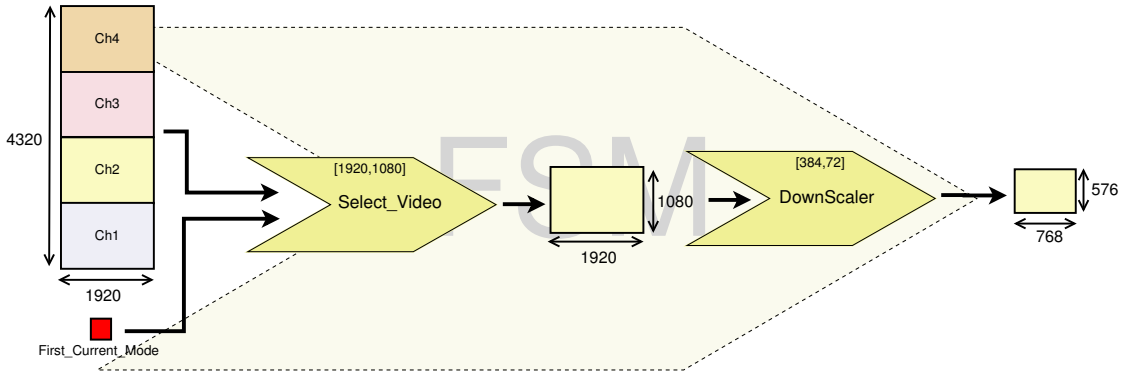


FIG. 9.7 – *Schéma global de fonctionnement du mode* `FSM`

9.2.2 Le mode `PIP`

Dans le mode `PIP` (Picture In Picture), le système est capable d'afficher une image à l'intérieur d'une autre dont le but est de visualiser en même temps des vidéos différentes comme il est montré par l'exemple de la figure 9.8. Dans cette application, le système est capable d'afficher sur une résolution de 256×288 pixels ($1/6$ de la résolution totale de l'écran) une image correspondant à une autre vidéo. Dans ce contexte, un processus de DownScaler est indispensable pour la mise à l'échelle de l'image de taille 256×288 pixels. Le principe du DownScaler est le même que celui décrit précédemment avec un facteur d'échelle horizontal de $15/2$ (21 pixels de l'image en entrée pour la génération de 2 pixels de l'image en sortie), et un facteur d'échelle vertical de $15/4$ (21 pixels de l'image en entrée pour la génération de 4 pixels de l'image en sortie). Ainsi, il est possible de naviguer entre les différentes vidéos sur la fenêtre de 256×288 pixels en utilisant la tâche `Select_Video` décrite dans la section précédente. Pour les autres points de l'image en sortie, l'application affiche les points relatifs à l'image en cours selon le mode de la vidéo actuelle (`First_Current_Video`). Pour respecter la régularité des traitements et l'hypothèse d'assignation unique des points de l'image en sortie, nous allons diviser l'image en sortie en trois parties régulières comme il est montré par la figure 9.9. Pour les deux parties `P1` et `P2`, les points de l'image en sortie sont relatifs à ceux de l'image de la vidéo actuelle, tandis que les points de la partie `P3` correspondent à

FIG. 9.8 – *Exemple d'un affichage en mode* `PIP`

FIG. 9.9 – *Découpage de l'image en sortie pour le traitement en mode* `PIP`

ceux de l'image de taille 256 × 288 pixels. Il est également à noter que le traitement du mode `PIP` est toujours précédé par un traitement dans le mode `FSM` pour la construction de l'image de base. La figure 9.10 représente le schéma global de l'application relative au mode de fonctionnement `PIP`. Ainsi, pour des raisons de simplification, nous considérons que le système ne peux pas afficher directement en mode `FSM` une image sélectionnée dans la fenêtre de 256 × 288 pixels. Dans ce cas, l'appui sur le bouton `Full` permet uniquement de fermer la petite fenêtre et passer en mode `FSM`.

9.2.3 Le mode `SSC`

Le fonctionnement du mode `SSC` (Split SCreen mode) permet l'affichage de deux images relatives à des vidéos différentes en découpant l'écran en deux parties. Ce découpage peut être fait de manière horizontale ou verticale comme il est montré par l'exemple de la figure 9.11. Dans ce contexte, la description du comportement du mode `SSC` est composée de deux sous-modes : `SSCH` et `SSCV`. À l'initialisation, nous considérons que les images s'affichent en en mode `SSCH`, et le passage entre les deux modes peut se faire via les boutons `Hr` et `Vr` de la télécommande comme il est illustré par l'automate de contrôle de la figure 9.12.

De façon générale, et pour les deux sous-modes de fonctionnement `SSCH` et `SSCV`, le comportement de l'application consiste à sélectionner l'image de la vidéo à afficher, et à réaliser par la suite un processus de mise à l'échelle. Ce processus est similaire à celui du mode

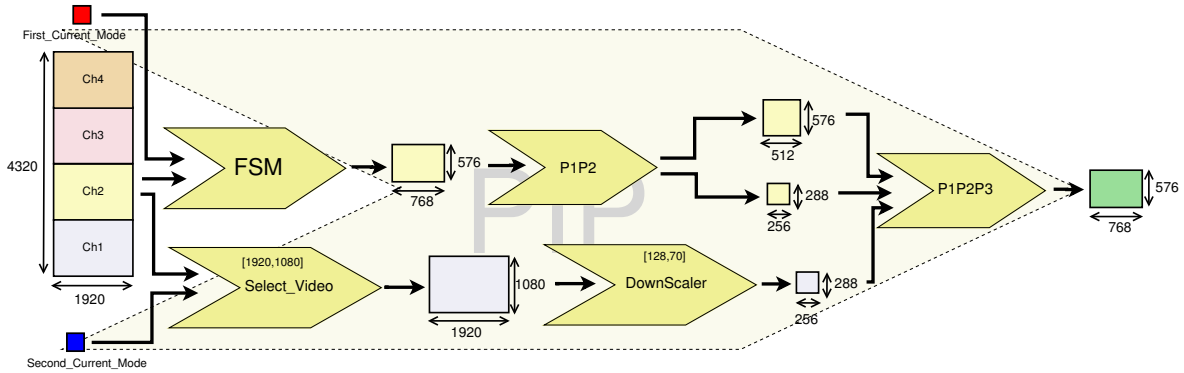


FIG. 9.10 – *Schéma global de fonctionnement du mode* `PIP`

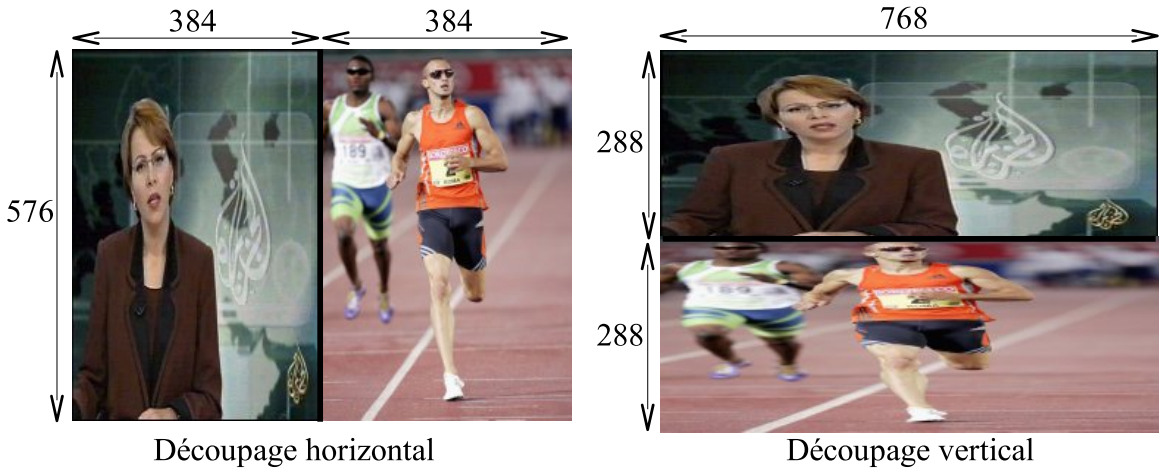


FIG. 9.11 – *Exemple d'un affichage en mode* `SSC`

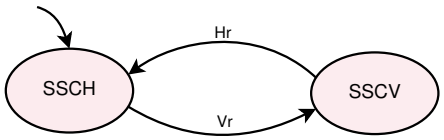


FIG. 9.12 – *Les différents sous-modes de fonctionnement du mode* `SSC`

`FSM` avec des facteurs d'échelle différents. Dans le cas d'un découpage horizontal, le facteur d'échelle horizontal pour chacune des deux images est de 5/1 (11 pixels de l'image en entrée sont nécessaires pour le calcul d'1 pixel de l'image en sortie), tandis que le facteur d'échelle vertical est de 15/8 (21 pixels de l'image en entrée sont nécessaires pour le calcul de 8 pixels de l'image en sortie). Dans le cas d'un découpage vertical, le facteur d'échelle horizontal pour chacune des deux images est de 5/2 (11 pixels de l'image en entrée sont nécessaires pour le calcul de 2 pixels de l'image en sortie), tandis que le facteur d'échelle vertical est de 15/4 (21 pixels de l'image en entrée sont nécessaires pour le calcul de 4 pixels de l'image en sortie). La figure 9.13 représente le schéma global de l'application relative au sous-mode de fonctionnement `SSCH`. Ainsi, pour des raisons de simplification, nous considérons que

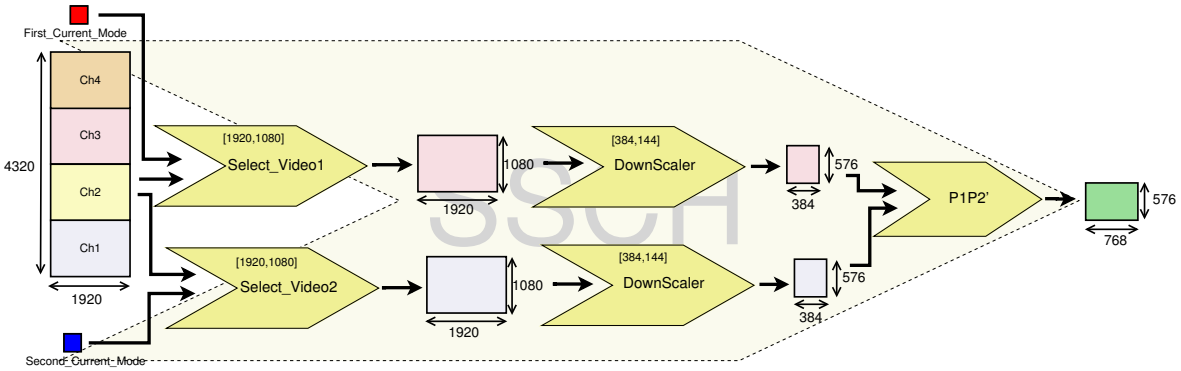


FIG. 9.13 – *Schéma global de fonctionnement du sous-mode* `SSCH`

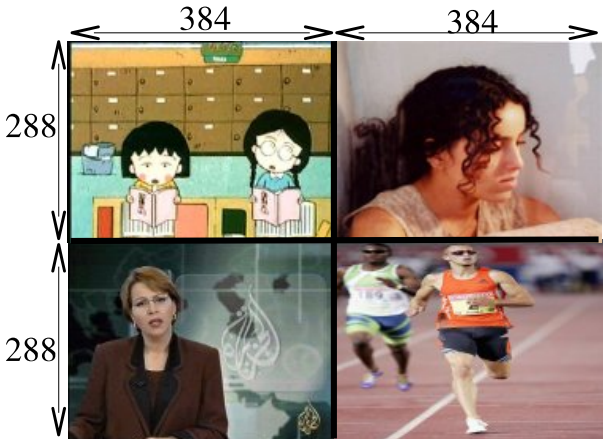


FIG. 9.14 – *Exemple d'un affichage en mode* `MUP`

lorsque le système revient en mode `FSM`, l'image qui sera affichée en plein écran est celle relative à la vidéo donnée par la valeur de mode `First_Current_Video` (l'image à gauche de l'écran dans le cas du sous-mode `SSCH`, et celle en haut de l'écran dans le cas du sous-mode `SSCV`).

9.2.4 Le mode `MUP`

Dans le mode de fonctionnement `MUP` (MUlti Picture mode), le système est capable d'afficher toutes les vidéos sur l'écran en sortie sous une forme mosaïque. Puisque nous considérons que l'application MMVP peut manipuler quatre vidéos différentes, l'écran en sortie pour cette application sera découpé en quatre parties de telle sorte que les parties de gauche à droite et de bas en haut affichent dans l'ordre les vidéos `Ch1`, `Ch2`, `Ch3` et `Ch4` comme il est montré par l'exemple de la figure 9.14.

La réalisation de l'application en mode `MUP` consiste tout simplement à appliquer un processus de mise à l'échelle (un DownScaler) pour les quatre vidéos comme il est expliqué par la figure 9.15. Pour la définition de chaque image de taille 384×288, il est nécessaire d'appliquer un filtre horizontal avec un facteur d'échelle horizontal de $5/1$ (11 pixels de l'image en entrée sont nécessaires pour le calcul d'1 pixel de l'image en sortie), et un filtre vertical avec

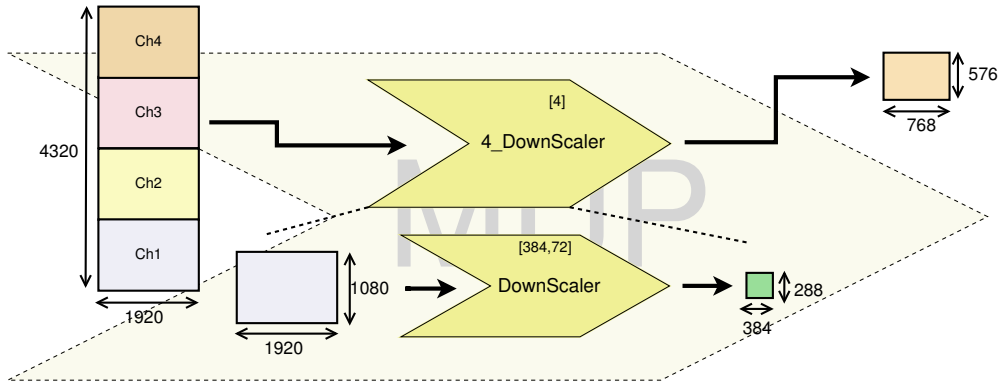


FIG. 9.15 – *Schéma global de fonctionnement du mode* `MUP`

un facteur d'échelle vertical de 15/4 (21 pixels de l'image en entrée sont nécessaires pour le calcul de 4 pixels de l'image en sortie). Ainsi, pour des raisons de simplification, nous considérons que le passage en mode `FSM` permet d'afficher la vidéo correspondant à la valeur de `First_Current_Video`.

9.2.5 Le mode `SPS`

Le mode de fonctionnement `SPS` (SnaPShot mode) représente un cas particulier dans lequel le système offre la possibilité de sauvegardé une image en mémoire interne en même temps que la vidéo en cours continue à s'exécuter. Ce mode de fonctionnement est utile pour sauvegarder une image d'une vidéo qui contient une information importante tel qu'un numéro de téléphone comme il est montré par l'exemple de la figure 9.16.

FIG. 9.16 – *Exemple de fonctionnement en mode* `SPS`

Pour des raisons de simplification, nous considérons que le mode `SPS` peut être uniquement activé si le système est en mode `FSM`. Dans ce cas, l'appui sur le bouton `Snap` permet de sauvegarder l'image courante du mode `FSM` en mémoire. Une fois le bouton `Snap` est relâché, le système revient directement en mode `FSM`. Le fonctionnement du mode `SPS` se résume donc à la mémorisation de l'image de la vidéo en cours donnée par la valeur de `First_Current_Mode` comme il est expliqué par la figure 9.17.

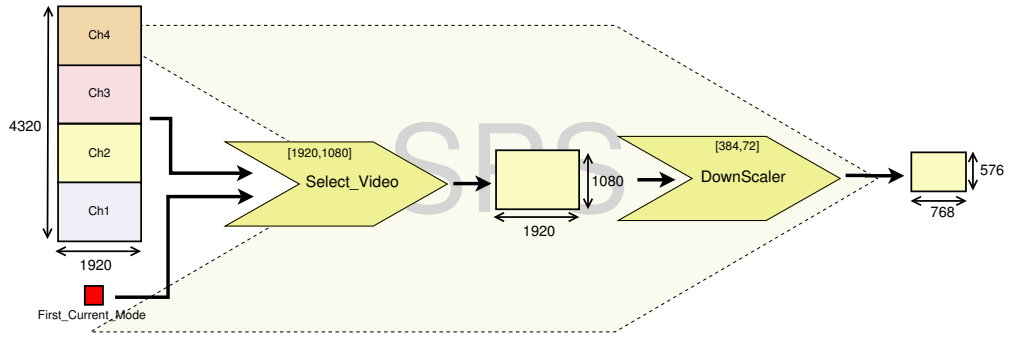


FIG. 9.17 – *Schéma global de fonctionnement du mode* `SPS`

9.2.6 Le mode `DSPS`

Le fonctionnement du mode `DSPS` est lié à celui du mode `SPS`. Le mode `DSPS` permet l'affichage de l'image sauvegardée par le mode `SPS` en plein écran en appuyant sur le bouton `DSnap` comme il est montré par l'exemple de la figure 9.18. Pour des raisons de simplification,

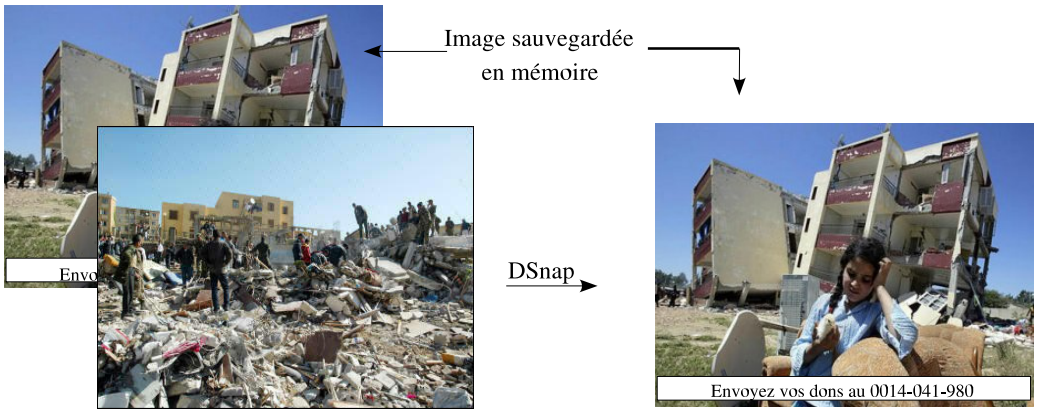


FIG. 9.18 – *Exemple de fonctionnement en mode* `DSPS`

nous considérons que le mode `DSPS` ne peut être activé que si le système est en mode `FSM`. Ainsi, lorsque le système est en mode `DSPS`, le retour en mode `FSM` ne peut être fait qu'en appuyant sur le bouton `Full`. Nous considérons également que si le système passe en mode `DSPS` et aucune image n'est sauvegardée en mémoire, une image par défaut représentant un message d'erreur sera affichée sur l'écran en sortie.

Le comportement du mode de fonctionnement `DSPS` est décrit par le schéma de la figure 9.19, et qui consiste tout simplement à afficher l'image sauvegardée en mémoire sur l'écran en sortie.

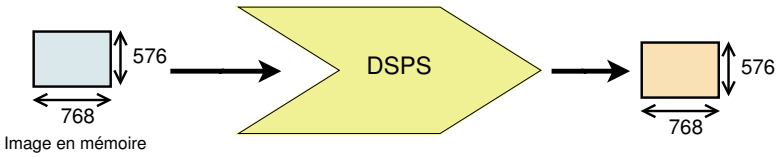


FIG. 9.19 – *Schéma global de fonctionnement du sous-mode* `DSPS`

9.3 Modélisation de l'application MMVP en utilisant le profil GASPARD2

Dans cette section, nous allons présenter la modélisation au plus haut niveau d'abstraction du comportement de l'application MMVP en utilisant le profil GASPARD2 présenté dans le chapitre 8. Comme nous l'avons présenté dans la section précédente, la description de l'application MMVP contient un mélange de traitement de données massivement parallèle et du contrôle. L'étude de cette application permet donc d'illustrer l'utilisation des différents concepts introduits dans le profil GAPSARD2, notamment en ce qui concerne la modélisation de la partie contrôle, les différents modes de fonctionnement, et le lien entre le contrôle et les parties contrôlées. Nous rappelons que le degré de granularité choisi pour la description de l'application MMVP est de la forme $\overrightarrow{d_{DG}} = \binom{768}{576}$, avec $F_{DG} = \begin{pmatrix} 1 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}$ et $P_{DG} = \begin{pmatrix} 0 \\ 0 \\ 576 \end{pmatrix}$. Ce degré de granularité est relatif à la génération d'une seule image en sortie de dimension $[768, 576]$. Ceci nous permettra de créer un flot d'images en sortie pour lequel le changement de modes devient possible uniquement entre l'affichage de deux images successives.

9.3.1 Modélisation de la partie contrôle

Comme nous l'avons mentionné dans la section précédente, la description de l'application MMVP est composée de six modes de fonctionnement. L'activation et le passage entre ces différents modes sont pilotés par un automate de contrôle en fonction de son état courant et des valeurs des événements en entrée (les boutons appuyés).

La modélisation de l'automate de contrôle en respectant la sémantique du profil GASPARD2 consiste à représenter le comportement de cet automate par une fonction de transition et une dépendance inter-répétition sur cette fonction de transition comme il est montré par le modèle de la figure 9.20. Cette modélisation permet d'introduire explicitement une sémantique de flot au modèle de conception sans avoir à modifier la sémantique de base (sémantique de temps banalisé) des modèles dans GASPARD2.

Dans le modèle de la figure 9.20, nous avons introduit le nouveau type énuméré `MMVP_modes` qui permet d'énumérer les différents modes de l'application MMVP sous forme d'éléments de type `ModeLiteral`. Ce type énuméré est utilisé pour la définition du type des données qui peuvent circuler sur les *ports* `initial_mode`, `current_mode` et `mode_MMVP`, et pour la définition des valeurs des conditions d'activation des différents modes de fonctionnement de l'application MMVP comme nous allons l'expliquer dans les sections suivantes. Le composant `MMVP_Control` est un composant de type `RepetitiveTransitionComponent`. Il est utilisé pour la représentation de l'automate de contrôle sous forme d'une répétition sur la *part* de la fonction de transition `MMVP_Transition_Function`. Le composant `MMVP_Control` prend en entrée une infinité de valeurs booléennes relatives aux boutons de la télécommande `Full`, `PiP`, `Split`, `Multi`, `Snap` et `DSnap`, et retourne en sortie une infinité de valeurs de mode à activer en fonction de son mode actuel. La dimension infini ($*$) des *ports* en entrées et celui en sortie est relative à la dimension temporelle. Cette dernière permet d'exprimer un flot de valeurs pour les événements de contrôle et le mode en sortie grâce à la description du lien de type `interRepetitionLinkTopology` défini sur les différentes répétitions de la *part* `MMVP_Tr`.

Le comportement de l'automate de contrôle est donc représenté par la fonction de transition `MMVP_Transition_Function` et une dépendance inter-répétition d'ordre 1 sur cette fonc-

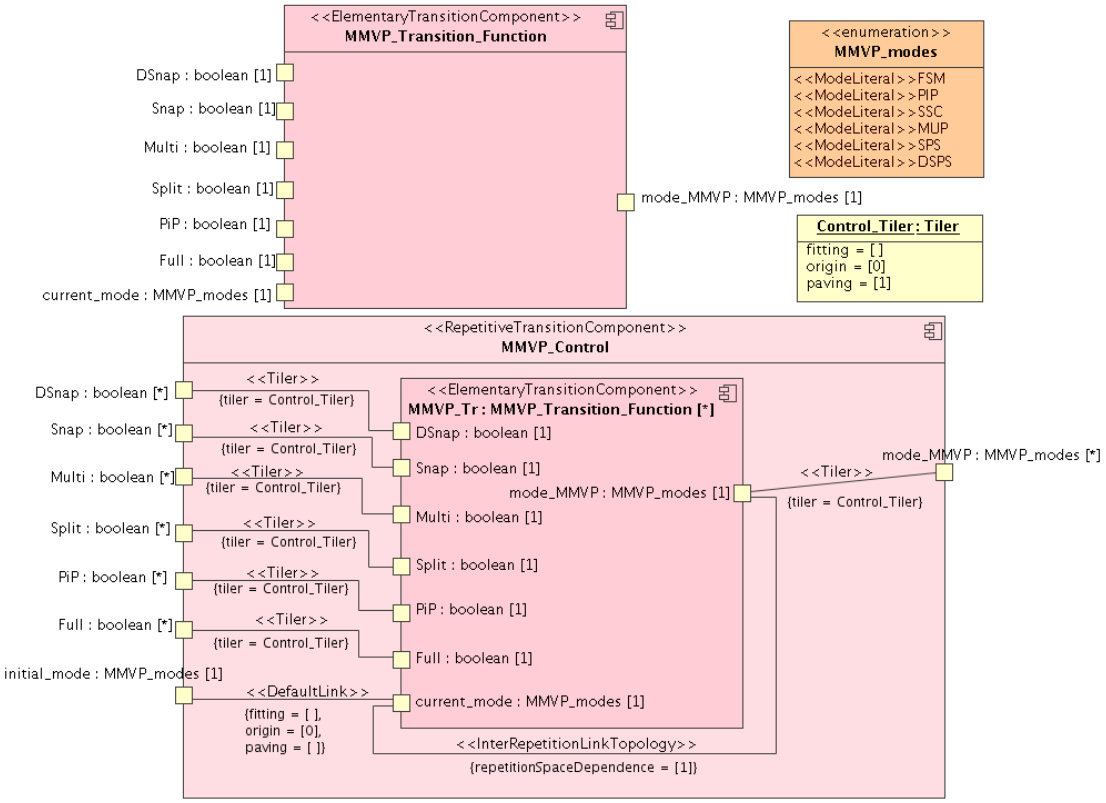


FIG. 9.20 – *Modélisation de l'automate de contrôle pour l'application* MMVP

tion de transition pour permettre la mémorisation de l'état courant de cet automate. Pour des raisons de simplification, nous avons choisi de décrire la fonction de transition de l'automate par un composant de type `ElementaryTransistionComponent`. Dans ce cas, le composant de cette fonction de transition est vu comme une boite noire dont le comportement sera par la suite décrit dans un langage source. La répétition de la fonction de transition est exprimée via le composant répétitif `MMVP_Control`, et elle est basée sur les concepts du modèle ARRAY-OL. Ces concepts sont exprimés en utilisant des connecteurs de type `Tiler`, et les informations de la répétition sont introduites via l'instance de classe `Control_Tiler`. Cette dernière permet d'éviter la réécriture des informations d'origine, de pavage et d'ajustage puisqu'elles sont identiques pour tous les `Tilers` de la répétition de l'automate de contrôle.

9.3.2 Modélisation des processus communs aux différents modes de fonctionnement

De façon générale, le comportement des différents modes de fonctionnement de l'application MMVP consiste à sélectionner l'image (ou les images) de la vidéo (ou des vidéos) à afficher, effectuer une mise à l'échelle, et afficher l'image (ou les images) sur l'écran en sortie. Dans ce qui suit, nous allons présenter le principe de base et la modélisation des processus communs aux différents modes de fonctionnements. Ces processus sont principalement : `Select_Video` qui permet de sélectionner l'image de la vidéo à afficher, et `DownScaler` qui permet d'effectuer une mise à l'échelle des images en entrée pour pouvoir les afficher sur

l'écran en sortie.

Le processus `Select_Video`

Le processus `Select_Video` permet de sélectionner l'image de la vidéo à afficher selon le choix de l'utilisateur, et via les boutons de la télécommande `Left` et `Right`. Le comportement de ce processus peut être modélisé sous forme d'une application contrôlée qui, en fonction de la valeur des événements de contrôle (les boutons appuyés), peut choisir l'image de la vidéo à afficher parmi celles des quatre vidéos disponibles.

Puisque nous considérons que le degré de granularité pour l'application globale est relatif à la génération d'une seule image en sortie, nous allons modéliser le contrôle du processus `Select_Video` pour une seule fonction de transition permettant la génération d'une seule image en sortie. La répétition de cette fonction de transition sera par la suite décrite via la répétition sur l'application globale. La modélisation de la fonction de transition pour cette application, en utilisant les concepts du profil GASPARD2, est donnée par la figure 9.21. Cette

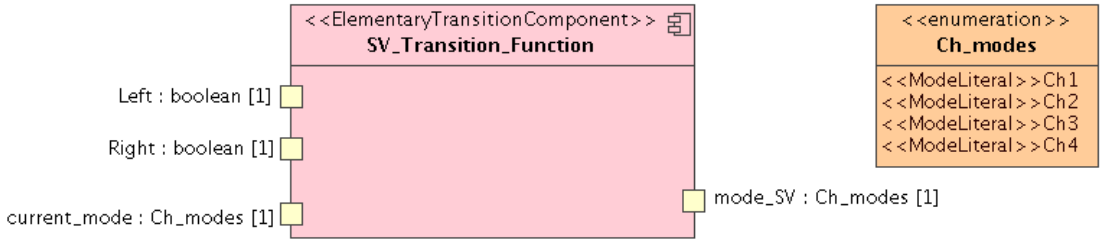


FIG. 9.21 – *Modélisation de l'automate de contrôle pour l'application* `Select_Video`

fonction de transition permet de piloter l'activation et le passage entre les quatre sous-modes de fonctionnement `Ch1_mode`, `Ch2_mode`, `Ch3_mode` et `Ch4_mode` relatifs aux différentes vidéos manipulées par le système. Le principe de modélisation de cette partie de contrôle est similaire à celui utilisé dans la section 9.3.1 pour la modélisation de l'automate de contrôle pour l'application globale. La seule différence est que la modélisation de l'automate de contrôle par une dépendance inter-répétition sur la fonction de transition sera pris en compte à un niveau plus haut de la hiérarchie.

Les quatre sous-modes de fonctionnement `Ch1_mode`, `Ch2_mode`, `Ch3_mode` et `Ch4_mode` permettent respectivement de sélectionner une image de la vidéo `Ch1`, `Ch2`, `Ch3` et `Ch4`. Le fonctionnement de ces quatre sous-modes est très similaire. La seule différence est dans la valeur de l'origine pour la sélection de l'image de la vidéo en entrée. Il est à noter que la modélisation du comportement de cette application peut être plus simple et plus facile à réaliser dans la possibilité de description des changements des informations des TILERs en entrée et en sortie. Puisque le changement des TILERs n'est pas encore étudié, nous allons nous limiter à la description du comportement de cette application en utilisant différentes instances d'un composant qui permet la sélection des éléments d'une image en entrée, mais avec des valeurs d'origine différentes comme il est montré par le modèle de la figure 9.22. Dans ce modèle, chaque sous-mode prend quatre images en entrée de dimension $[1920, 4320]$ relatives aux quatre vidéos manipulées par le système, et retourne en sortie une image de dimension $[1920, 1080]$ liée à la vidéo traitée par le sous-mode en question.

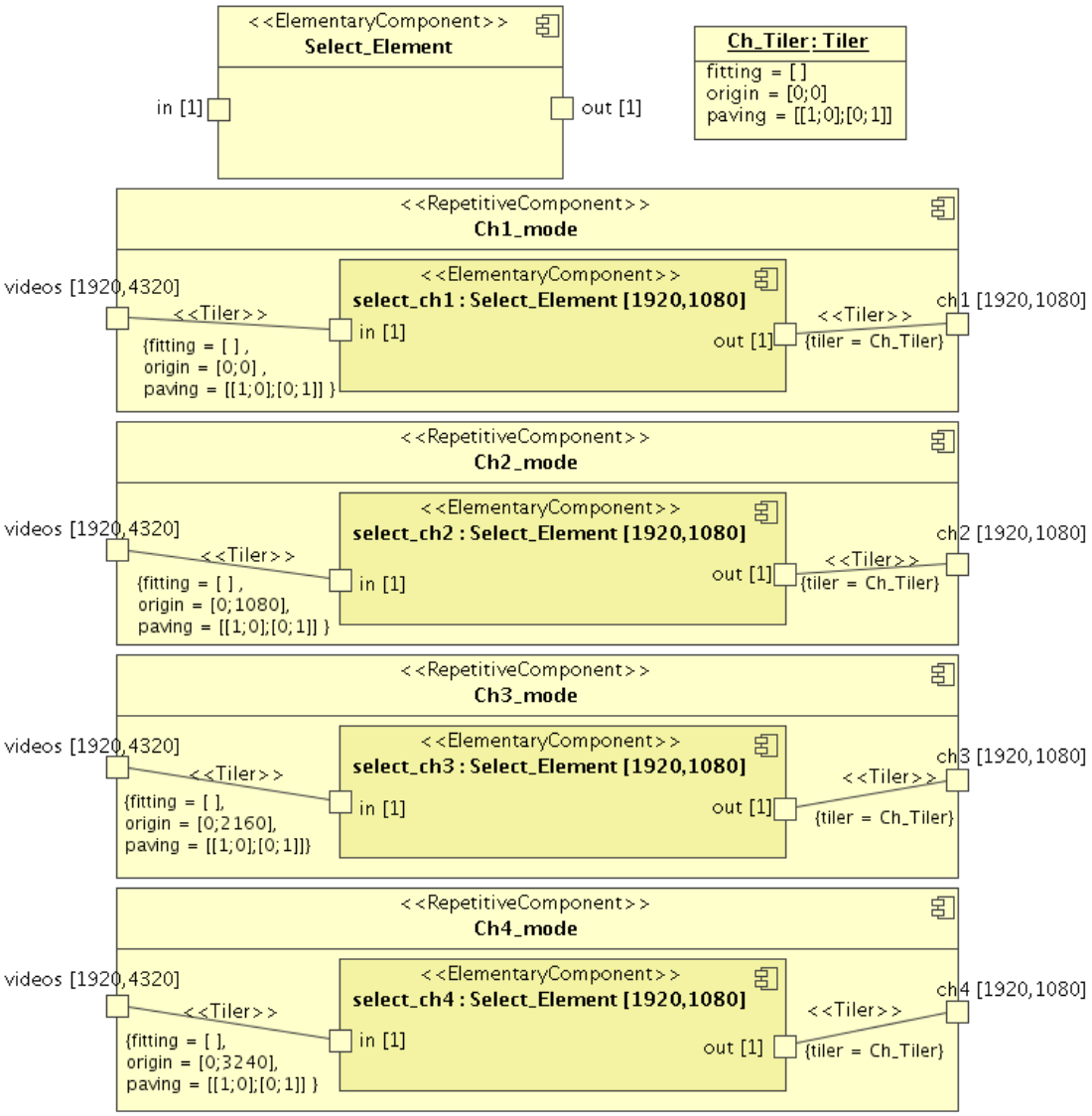


FIG. 9.22 – *Modélisation des différents sous-modes de l'application* `Select_Video`

La modélisation de la partie contrôlée correspondant au *switch* entre les quatre sous-modes de fonctionnement du processus `Select_Video` est représentée par le modèle de la figure 9.23. Dans ce modèle, chaque sous-mode est représenté par une *part* de type `AlternativeComponentPart` permettant de modéliser le caractère alternatif des différents sous-modes de fonctionnement de l'application `Select_Video`. Pour ce faire, nous avons défini pour chaque sous-mode de fonctionnement une valeur différente de type `Ch_modes` pour la condition d'activation via la *tagged value* `activationCondition`. Le composant `Switch_Select_Video` est un composant de type `ControlledComponent` dont le rôle est de regrouper les différentes *parts* représentant les quatre sous-modes de fonctionnement, et choisir celle à activer en fonction de la valeur de mode disponible sur son *port* d'entrée `mode`.

La figure 9.24 représente le modèle global relatif au processus `Select_Video`. Comme le lecteur peut le remarquer, nous avons modélisé le processus `Select_Video` pour une seule

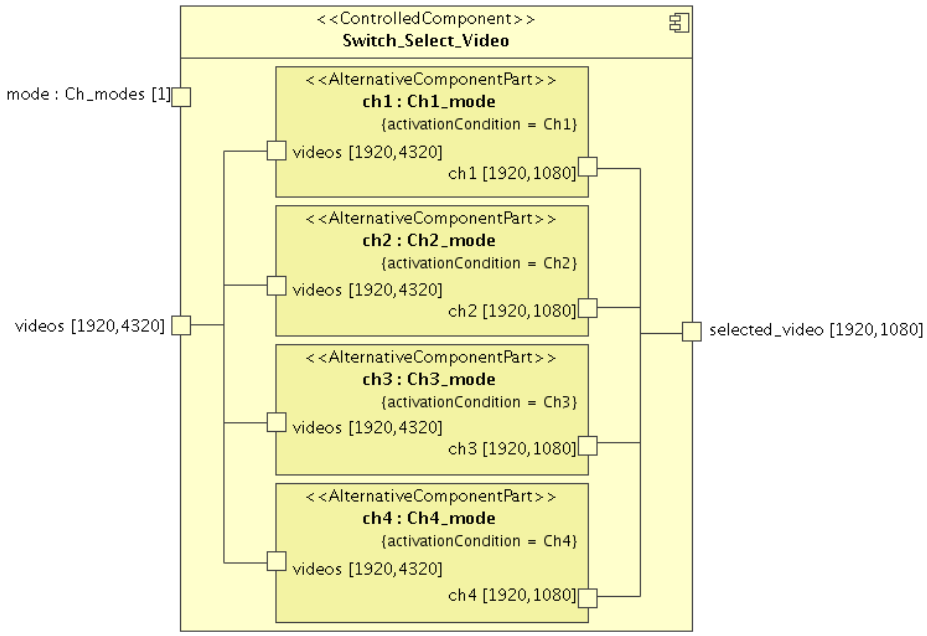


FIG. 9.23 – *Modélisation du switch entre les différents sous-modes de l'application* `Select_Video`

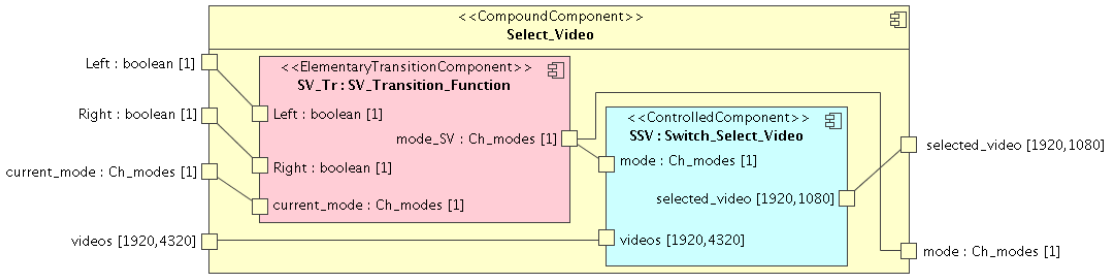


FIG. 9.24 – *Modélisation du comportement global de l'application* `Select_Video` *pour la génération d'une seule image en sortie*

répétition permettant la génération d'une seule image en sortie de dimension $[1920, 1080]$. Le but est de définir la répétition sur le traitement des différentes images en sortie de façon globale pour toute l'application MMVP puisque le degré de granularité choisi pour cette application est relatif à la génération d'une seule image en sortie. En d'autres termes, nous allons regrouper les différentes étapes de traitement utilisées pour la génération d'une seule image en sortie dans un seul composant piloté par un automate de contrôle, et qui sera répété pour le traitement de toutes les images en sortie comme nous allons l'expliquer dans les sections suivantes. Puisque le processus de sélection d'une image de la vidéo représente une des étapes de traitement de l'image en sortie, nous avons modélisé ce processus par le composant `Select_Video` de type `CompoundComponent` qui permet la génération d'une image de dimension $[1920, 1080]$ à partir des images relatives aux quatre vidéos en entrée de dimension $[1920, 4320]$.

Le composant `Select_Video` est constitué d'une instance de la fonction de transition `SV_Transition_Function` qui permet de déterminer le sous-mode à activer en fonction du

choix de l'utilisateur via les boutons `Left` et `Right`, et d'une instance du composant contrôlé `Switch_Select_Video` qui, en fonction de la valeur de mode fournie par la fonction de transition, permet de sélectionner l'image correspondante à la vidéo sélectionnée. Ce composant fournit également en sortie la valeur de mode pour permettre la définition de la dépendance inter-répétition sur les différentes répétitions de ce composant.

Le processus `DownScaler`

Le processus `DownScaler` est utilisé par tous les modes de fonctionnement de l'application `MMVP` pour la mise à l'échelle des images en entrée et les afficher sur l'écran en sortie. Comme nous l'avons décrit précédemment, le comportement global du processus `DownScaler` est composé d'un filtre vertical et d'un filtre horizontal. La seule différence entre les processus de mise à l'échelle des différents modes de fonctionnement est dans la valeur des facteurs d'échelle horizontal et vertical.

Pour des raisons de simplification, nous allons uniquement décrire la modélisation du processus `DownScaler` pour la génération d'une image en sortie de dimension $[768, 576]$ à partir d'une image en entrée de dimension $[1920, 1080]$. Le comportement des autres processus de mise à l'échelle est très similaire avec comme seule différence la valeur des facteurs de mise à l'échelle, et par conséquent, la valeur des informations pour les TILERs en entrée et ceux en sortie. Dans le cas choisi, le facteur d'échelle horizontal est de $5/2$ (11 pixels en entrée pour la génération de 2 pixels en sortie), tandis que le facteur d'échelle vertical est de $15/8$ (21 pixels en entrée pour la génération de 8 pixels en sortie).

Puisque le processus `DownScaler` représente une des étapes de traitement d'une image en sortie pour l'application globale MMVP, nous allons modéliser ce processus uniquement pour la génération d'une seule image en sortie. La répétition sur ce processus pour la génération de toutes les images en sortie sera par la suite décrite via la répétition sur l'application globale. La figure 9.25 représente la modélisation du comportement du filtre horizontal pour le processus `DownScaler`. Dans ce modèle, le composant `RepHF` est de type `RepetitiveComponent` qui permet d'exprimer le parallélisme potentiel, à la ARRAY-OL, de l'exécution des différentes *parts* du composant élémentaire `HF`. Le composant `RepHF` est utilisé pour le calcul des filtres horizontaux permettant la génération d'une seule image en sortie de dimension $[768, 1080]$.

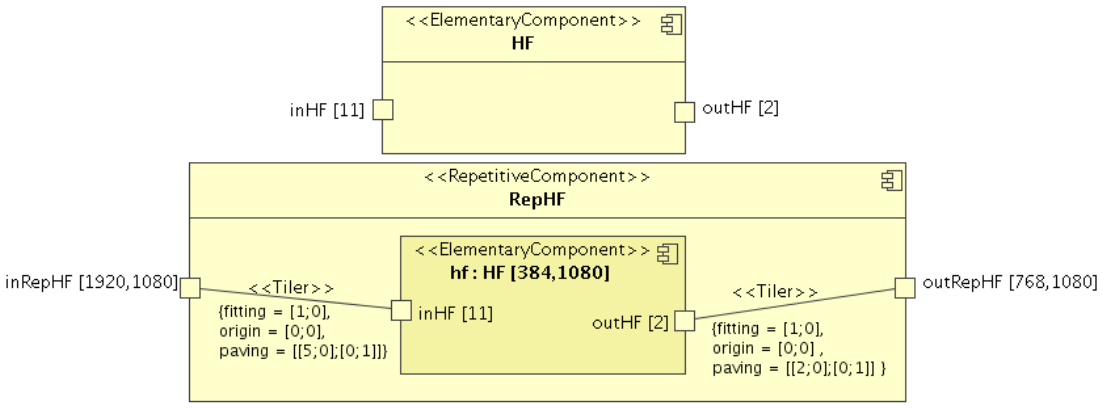


FIG. 9.25 – *Modélisation du comportement d'un filtre horizontal pour la génération d'une seule image en sortie*

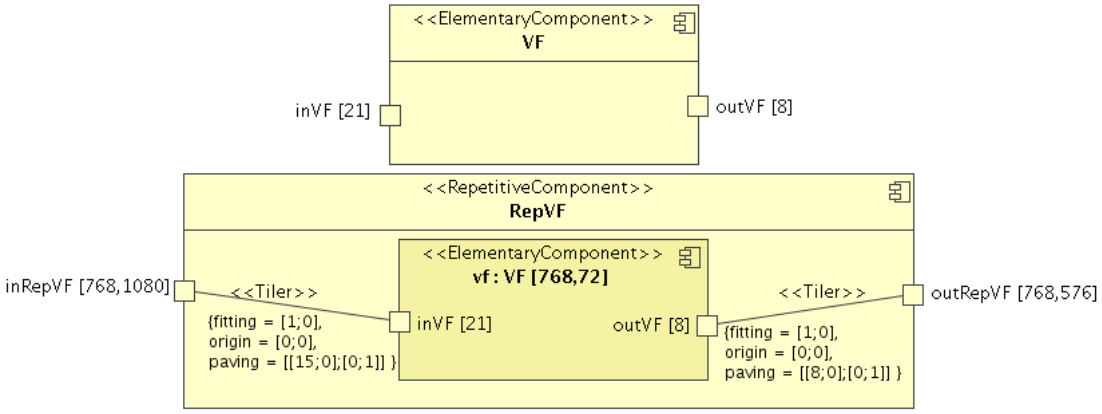


FIG. 9.26 – *Modélisation du comportement d'un filtre vertical pour la génération d'une seule image en sortie*

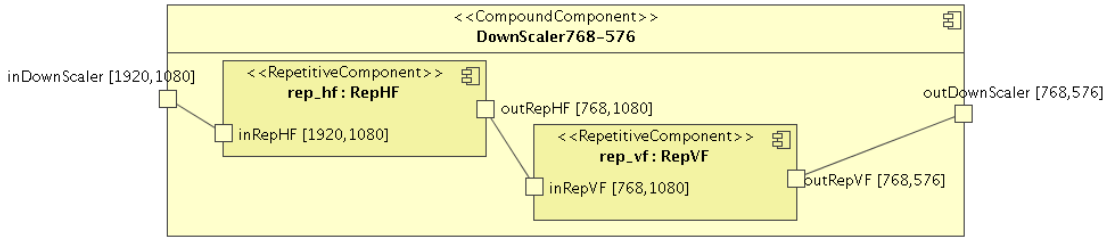


FIG. 9.27 – *Modélisation du comportement du processus* `DownScaler` *pour la génération d'une seule image en sortie*

Le processus du filtre horizontal est suivi par celui d'un filtre vertical. La figure 9.26 représente la modélisation du comportement du filtre vertical pour le processus `DownScaler`. Dans ce modèle, le composant `RepVF` est utilisé pour le calcul des filtres verticaux permettant la génération d'une seule image en sortie de dimension $[768, 576]$. C'est un composant de type `RepetitiveComponent` qui permet d'exprimer le parallélisme potentiel, à la ARRAY-OL, des différentes *parts* du composant élémentaire `VF`.

Le comportement global du processus `DownScaler` pour la génération d'une seule image en sortie est donné par le modèle de la figure 9.27. Dans ce modèle, le composant `DownScaler768-576` est de type `CompoundComponent` qui regroupe deux *parts* relatives aux composants `HF` et `VF`. Son rôle est donc d'appliquer sur l'image en entrée de dimension $[1920, 1080]$ un filtre horizontal suivi par un filtre vertical pour la génération d'une image en sortie de dimension $[768, 576]$.

9.3.3 Modélisation du mode `FSM`

Le fonctionnement du mode `FSM` consiste à sélectionner une image des vidéos en entrée et effectuer une mise à l'échelle sur cette image pour l'afficher sur l'écran en sortie. Ce comportement est donc principalement composé des deux processus `Select_Video` et `DownScaler` dont nous avons décrit la structure dans la section précédente.

Comme nous l'avons décrit dans la section 9.2.1, le comportement du mode de fonctionnement `FSM` est composé d'un premier composant de `Select_Video`, suivi par un pro-

cessus de DownScaler comme il est montré par le modèle de la figure 9.28. Ce modèle

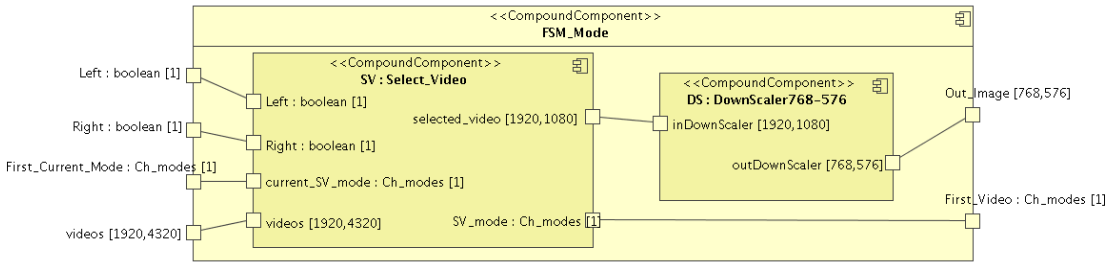


FIG. 9.28 – *Modélisation du comportement du mode de fonctionnement* `FSM` *pour la génération d'une seule image en sortie*

permet l'affichage d'une seule image en sortie. La répétition de cette application pour la génération de toutes les images en sortie sera par la suite décrite via la répétition sur l'application globale. Le composant `FSM_Mode` prend en entrée l'ensemble des quatre images de dimension $[1920, 4320]$ relatives aux différentes vidéos manipulées par le système, et fournit en sortie une image de dimension $[768, 576]$ relative à la vidéo à afficher sur l'écran de la télévision en fonction du choix de l'utilisateur. Ainsi, ce composant est de type `CompoundComponent` puisqu'il est constitué de deux *parts* relatives aux composants `Select_Video` et `DownScaler768-576`. Il fournit également en sortie une information sur la vidéo en cours de traitement qui sera prise en compte pour les prochaines répétitions.

9.3.4 Modélisation du mode `PIP`

Comme nous l'avons décrit dans la section 9.2.2, le comportement du mode de fonctionnement `PIP` consiste à afficher deux images sur l'écran en sortie qui sont généralement relatives à deux vidéos différentes. Le modèle décrivant ce comportement est donné par la figure 9.29. Ce modèle prend en considération le traitement d'une seule image en sortie

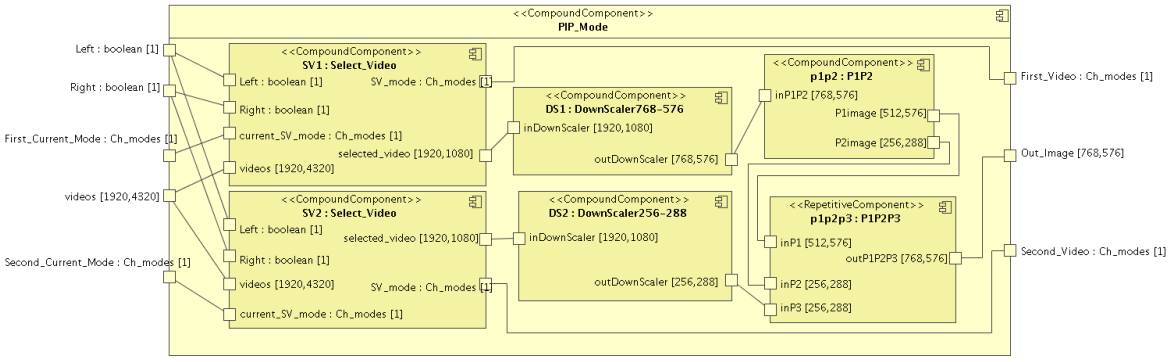


FIG. 9.29 – *Modélisation du comportement du mode de fonctionnement* `PIP` *pour la génération d'une seule image en sortie*

puisque la génération de toutes les images en sortie sera par la suite pris en compte via la description de la répétition pour l'application MMVP globale.

Le composant `PIP_Mode` est de type `CompoundComponent`. Ce composant est constitué de deux *parts* du composant `Select_Video` permettant de sélectionner les deux images relatives

aux vidéos à afficher. La sélection de ces deux vidéos est réalisée en fonction du choix de l'utilisateur et selon l'état courant du système donné par les valeurs de `First_Current_Mode` et `Second_Current_Mode`. Après avoir sélectionner les deux images à afficher, un processus de mise à l'échelle est effectué pour ces deux images avec des facteurs d'échelle différents. Le comportement de ces deux processus de mise à l'échelle est modélisé via les *parts* des composants `DownScaler768-576` et `DownScaler256-288`. La génération de l'image en sortie est réalisée par la *part* du composant `P1P2P3`. Ce dernier permet la construction des trois parties de l'image en sortie, comme nous l'avons expliqué par la figure 9.9 (page 208), tout en respectant l'hypothèse d'assignation unique pour les différents points de l'image en sortie. La structure du composant `P1P2P3` est donné par le modèle de la figure 9.30. Ce compo-

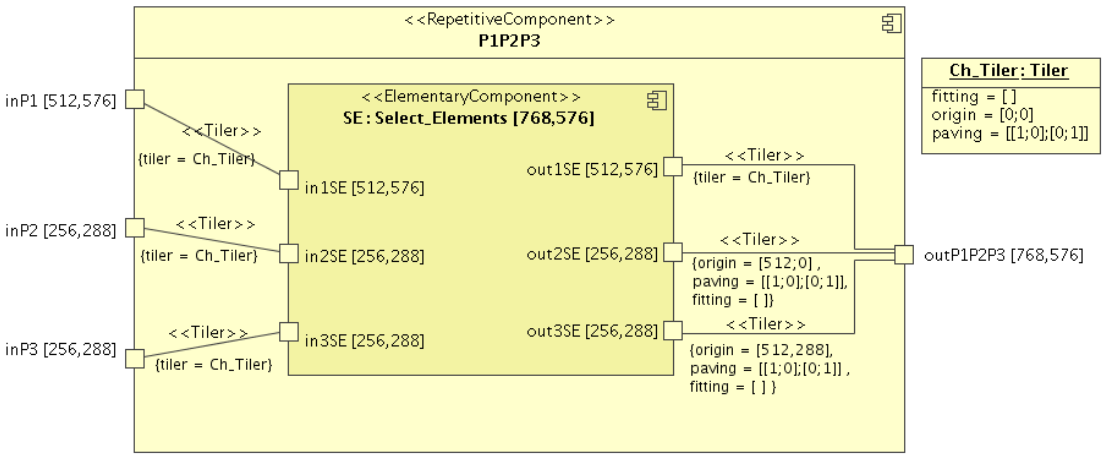


FIG. 9.30 – *Modélisation du comportement du composant `P1P2P3`*

sant prend en entrée les trois parties utilisées pour la construction de l'image en sortie. Les deux premières parties sont générées à partir de la *part* du composant `P1P2` représenté par le modèle de la figure 9.31, tandis que la troisième partie est générée par la *part* du composant `DownScaler256-288`, et elle est relative à l'image de la vidéo affichée sur la fenêtre de dimension $[256, 288]$ située en haut à droite de l'écran.

9.3.5 Modélisation du mode `SSC`

Comme nous l'avons décrit dans la section 9.2.3, le comportement du mode de fonctionnement `SSC` consiste à afficher sur l'écran en sortie deux images en les divisant de façon horizontal ou vertical. Ces images sont relatives aux deux vidéos sélectionnées par l'utilisateur parmi les quatre vidéos disponibles en utilisant les boutons de la télécommande `Left` et `Right`. La description du comportement de ce mode est donc composée de deux sous-modes de fonctionnement `SSCH_Mode` et `SSCV_Mode` qui permettent respectivement l'affichage horizontal et vertical des deux images sélectionnées. Le comportement de ces deux sous-modes est très similaire. Les seules différences sont dans la valeur des facteurs de mise à l'échelle pour les deux images en entrée, et dans les informations des TILERs en entrée et ceux en sortie utilisés pour la construction de l'image finale qui sera affichée sur l'écran en sortie. Pour cette raison, nous allons uniquement présenter le modèle relatif au sous-mode de fonctionnement `SSCH_Mode`.

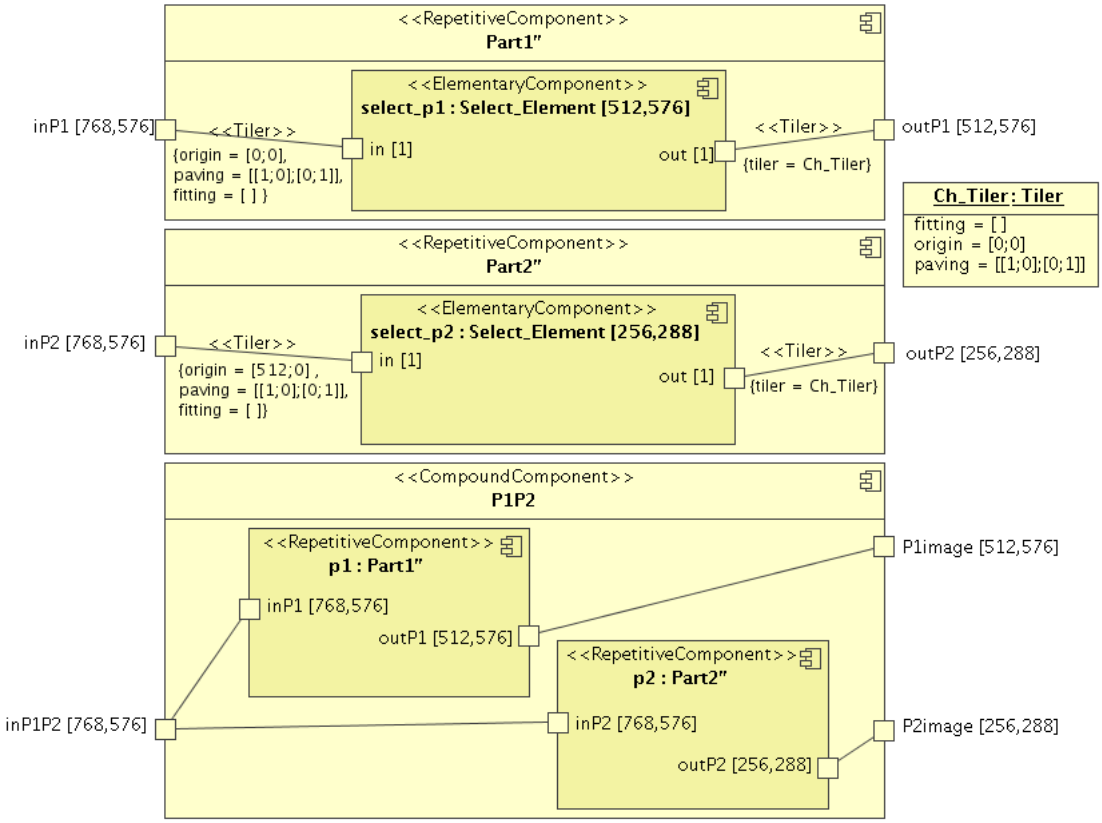


FIG. 9.31 – *Modélisation du comportement du composant* `P1P2`

La figure 9.32 représente le modèle de conception décrivant le comportement du sous-mode de fonctionnement `SSCH` en utilisant les concepts du profil GASPARD2. Dans ce modèle, le composant `SSCH_Mode` est de type `CompoundComponent` qui permet la génération d'une seule image en sortie puisque la génération de toutes les images en sortie sera par la suite décrite via la répétition sur l'application globale. Le composant `SSCH_Mode` est constitué de deux *parts* du composant `Select_Video` permettant la sélection des deux images à afficher, et de deux *parts* du composant `DownScaler384-576` permettant d'effectuer une mise à l'échelle de ces deux images pour pouvoir les afficher sur l'écran en sortie. La construction de l'image en sortie est réalisée par la *part* du composant `P1P2'` dont le principe de fonctionnement est similaire à celui donné par la figure 9.30. Le comportement du composant `P1P2'` est représenté par le modèle de la figure 9.33. Ce composant permet la génération d'une image en sortie de dimension $[768, 576]$ à partir des deux images en entrée de dimension $[384, 576]$.

Il est également à noter que le composant `SSCH_Mode` fournit en sortie des informations sur les deux vidéos sélectionnés via les *ports* `First_Video` et `Second_Video` pour pouvoir les utilisées dans la sélection des images des vidéos pour les répétitions suivantes.

La représentation du comportement global du mode de fonctionnement `SSC` est donnée par le modèle de la figure 9.34. Dans ce modèle, le composant `SSC_Mode` est constitué d'une *part* du composant `SSC_Transition_Function` qui permet de sélectionner le sous-mode de fonctionnement du mode `SSC` à activer, et d'une *part* du composant `Switch_SSC` qui représente le *switch* entre les deux sous-modes de fonctionnement `SSCH` et `SSCV` comme il est mon-

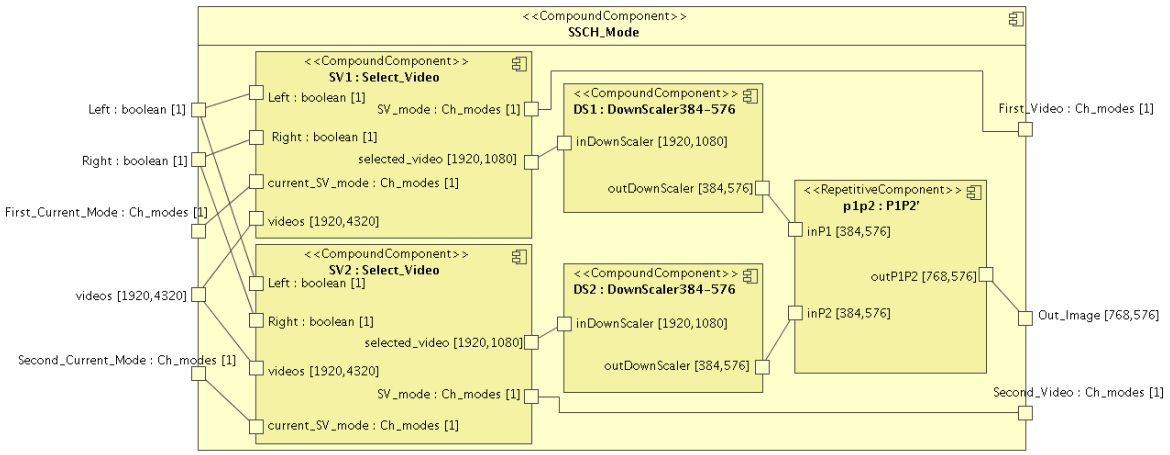


FIG. 9.32 – *Modélisation du comportement du sous-mode de fonctionnement* `SSCH` *pour la génération d'une seule image en sortie*

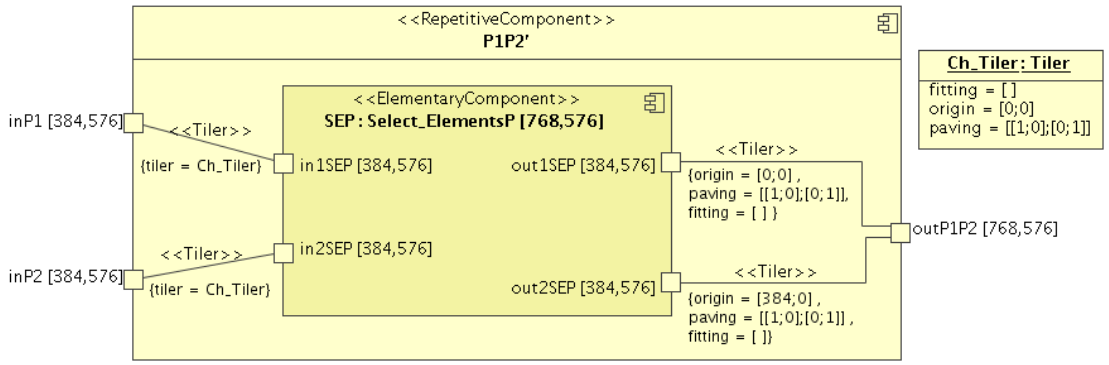


FIG. 9.33 – *Modélisation du comportement du composant* `P1P2'`

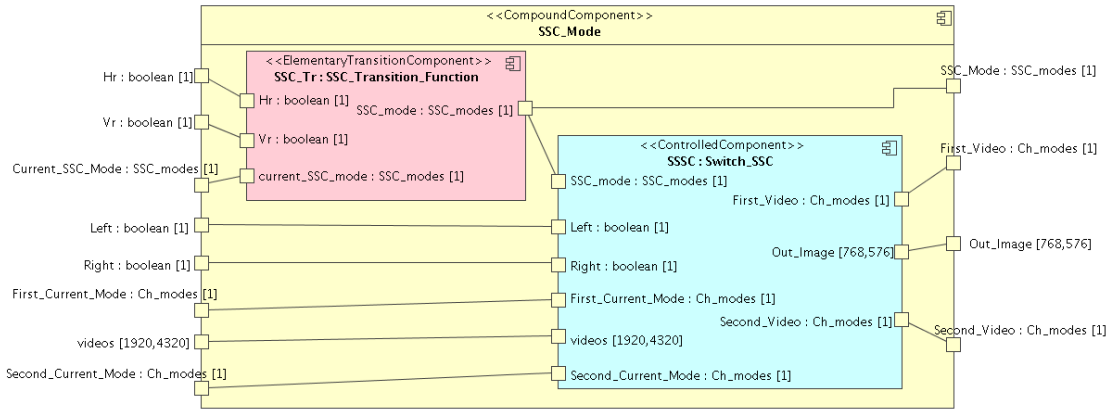


FIG. 9.34 – *Modélisation du comportement du mode de fonctionnement* `SSC` *pour la génération d'une seule image en sortie*

tré par le modèle de a figure 9.35. Dans ce modèle, le composant `Switch_SSC` est constitué

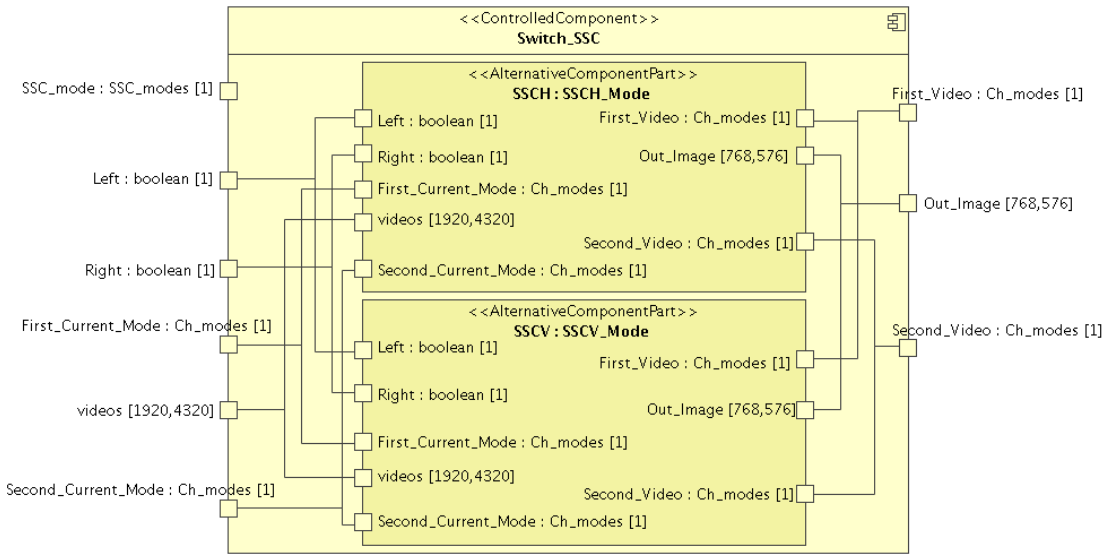


FIG. 9.35 – *Modélisation du comportement du composant `Switch_SSC`*

de deux *parts* de type `AlternativeComponentPart` permettant de décrire le comportement alternatif entre les deux sous-modes de fonctionnement `SSCH` et `SSCV`.

9.3.6 Modélisation du mode `MUP`

Le mode de fonctionnement `MUP` permet l'affichage des images des quatre vidéos en mosaïque comme nous l'avons expliqué dans la section 9.2.4. Le comportement de ce mode consiste à effectuer un processus de mise à l'échelle pour les quatre images en entrée pour la génération de l'image en sortie comme il montré par le modèle de la figure 9.36. Dans ce mo-

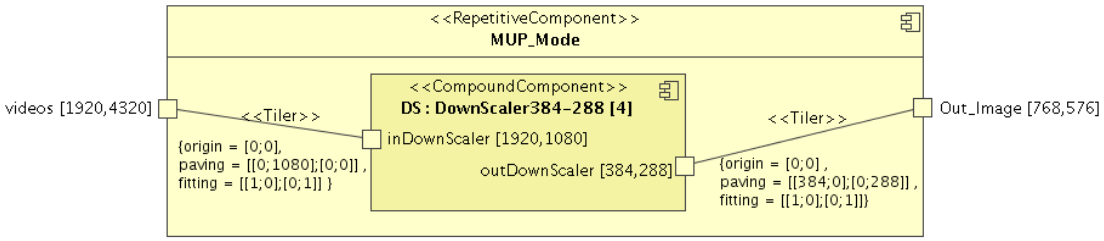


FIG. 9.36 – *Modélisation du comportement du mode de fonctionnement `MUP` pour la génération d'une seule image en sortie*

dèle, le composant `MUP_Mode` est de type `RepetitiveComponent` qui permet la définition du parallélisme sur la répétition des *parts* du composant `DownScaler384-288` pour la génération de l'image finale qui sera affichée sur l'écran en sortie. Ce composant permet la génération d'une seule image en sotie puisque nous considérons que la génération de toutes les images en sortie sera spécifiée via la répétition sur l'application globale.

9.3.7 Modélisation du mode SPS

Comme nous l'avons expliqué dans la section 9.2.5, le comportement du mode SPS consiste à sauvegarder l'image courante en mode FSM pour pouvoir l'afficher par la suite en mode DSPS.

La description du comportement du mode SPS consiste à sélectionner l'image courante selon la valeur de First_Current_Mode, effectuer un processus de mise à l'échelle pour cette image, et la sauvegarder en mémoire. La figure 9.37 représente le modèle relatif au fonctionnement du mode SPS. Dans ce modèle, le composant SPS_Mode est de type CompoundCompo-

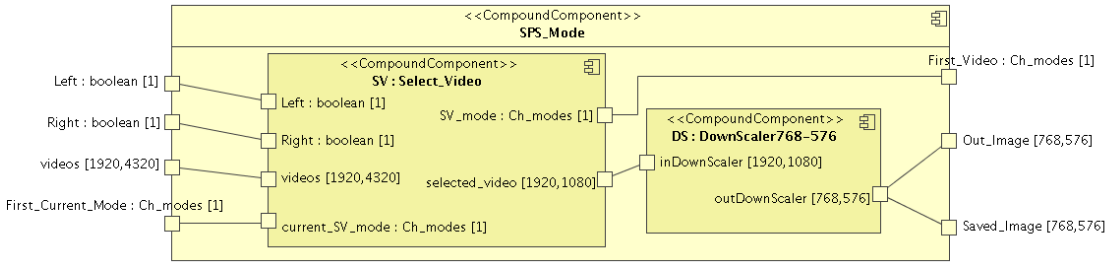


FIG. 9.37 – *Modélisation du comportement du mode de fonctionnement SPS pour la génération d'une seule image en sortie*

nent qui permet de sélectionner l'image à afficher sur l'écran en sortie et la sauvegarder en mémoire.

9.3.8 Modélisation du mode DSPS

Le mode DSPS permet l'affichage en plein écran de l'image sauvegardée en mémoire comme nous l'avons décrit dans la section 9.2.6. Le comportement de ce mode de fonctionnement est représenté par le modèle de la figure 9.38. Ce modèle est principalement défini

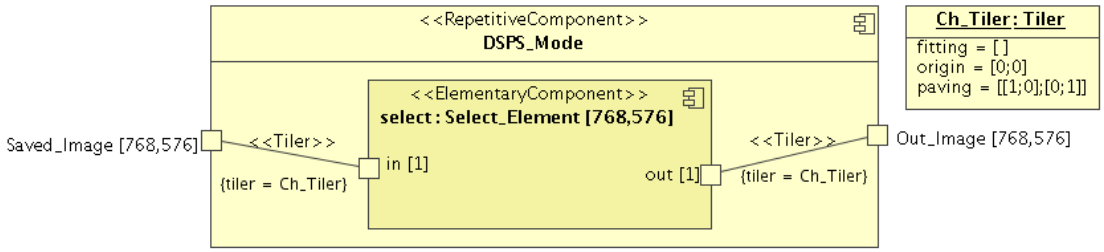


FIG. 9.38 – *Modélisation du comportement du mode de fonctionnement DSPS pour la génération d'une seule image en sortie*

par le composant DSPS_Mode qui est de type RepetitiveComponent permettant l'affichage de l'image sauvegardée en mémoire. Il est à noter que le comportement de ce mode peut être plus facilement décrit par une connexion de type Reshape entre les éléments de l'image sauvegardée et ceux de l'image qui sera affichée sur l'écran. Cependant, la description du comportement alternatif n'est pas possible sur les connecteurs, et il est uniquement défini pour les *parts* des composants. Pour cette raison, nous avons modélisé le comportement du mode DSPS par un composant répétitif.

9.3.9 Modélisation de l'application globale MMVP

Dans les sections précédentes, nous avons présenté la modélisation de l'automate de contrôle et les différents modes de fonctionnement de l'application MMVP. La représentation du comportement de l'application globale consiste à définir le lien entre l'automate de contrôle, qui permet la sélection du mode de fonctionnement, et le *switch* entre les différents mode de fonctionnement de l'application.

Comme nous l'avons discuté dans les chapitres précédents, les différents modes de fonctionnement d'une application contrôlée doivent avoir la même interface pour faciliter leur manipulation et réutilisation. Pour cette raison, nous allons définir un niveau de hiérarchie supplémentaire pour chaque mode de fonctionnement de l'application MMVP permettant d'assurer la même interface pour les six modes de cette application comme il est expliqué par le modèle de la figure 9.39. La définition de l'ensemble des entrées du composant hiérarchique englobant est relative à l'union des entrées des différents modes de fonctionnement et l'ensemble des valeurs par défaut pour les sorties des modes éventuellement non définies, tandis que la définition de l'ensemble des entrées de ce composant est relative à l'union des entrées des différents modes de fonctionnement (voir section 5.3.2, page 92).

Le modèle de la figure 9.40 représente le composant contrôlé relatif au *switch* entre les différents modes de fonctionnement de l'application MMVP. Les différentes *parts* définies à l'intérieur du composant contrôlé `Switch_MMVP` sont de type `AlternativeComponentPart`. Ces *parts* permettent la représentation du comportement alternatif des différents modes de fonctionnement de l'application MMVP. Le choix du mode de fonctionnement à activer est réaliser en fonction de la valeur de mode disponible sur le *port* `Mode_MMVP`, et en fonction de la valeur de la condition d'activation définie via la *tagged value* `activationCondition`.

Le composant contrôlé `Switch_MMVP` permet la génération d'une seule image en sortie de dimension $[768, 576]$. La description de la répétition sur ce composant pour la génération de toutes les images en sortie est donnée par le modèle de la figure 9.41. Dans ce modèle, le composant `Rep_Switch_MMVP` représente la répétition sur le composant `Switch_MMVP` pour la génération de toutes les images en sortie. Ce composant prend en entrée une infinité d'images de dimension $[1920, 4320]$, et génère en sortie une infinité d'images de dimension $[768, 576]$ relatives aux vidéos affichées sur l'écran en sortie. Les liens de type `interRepetitionLinkTopology` définis sur le composant `Switch_MMVP` permettent de garder des informations sur les vidéos traitées entre les différentes images affichées.

Nous rappelons que le degré de granularité choisi pour l'application MMVP est relatif à une seule image en sortie. Dans ce contexte, la *part* du composant `Switch_MMVP` prend en entrée une seule valeur de mode pour le traitement de chaque image en sortie de dimension $[768, 576]$. Ce comportement permet l'introduction d'une sémantique de flot pour le modèle global de l'application MMVP, et associe un comportement réactif à la description de cette application.

La modélisation du comportement global de l'application MMVP est donnée par le modèle de la figure 9.42. Dans ce modèle, le composant `MMVP_Application` est de type `CompoundComponent` qui regroupe la partie contrôle relative à l'automate de contrôle de l'application globale présenté dans la section 9.3.1, et la partie contrôlée relative à l'ensemble des modes de fonctionnement de l'application. Ce composant reçoit en entrée une infinité d'images relatives aux quatre vidéos manipulées, ainsi qu'une infinité de valeurs relatives aux différents boutons de la télécommande. En sortie, ce composant retourne une infinité d'images des vidéos à afficher sur l'écran de la télévision selon le choix de l'utilisateur.

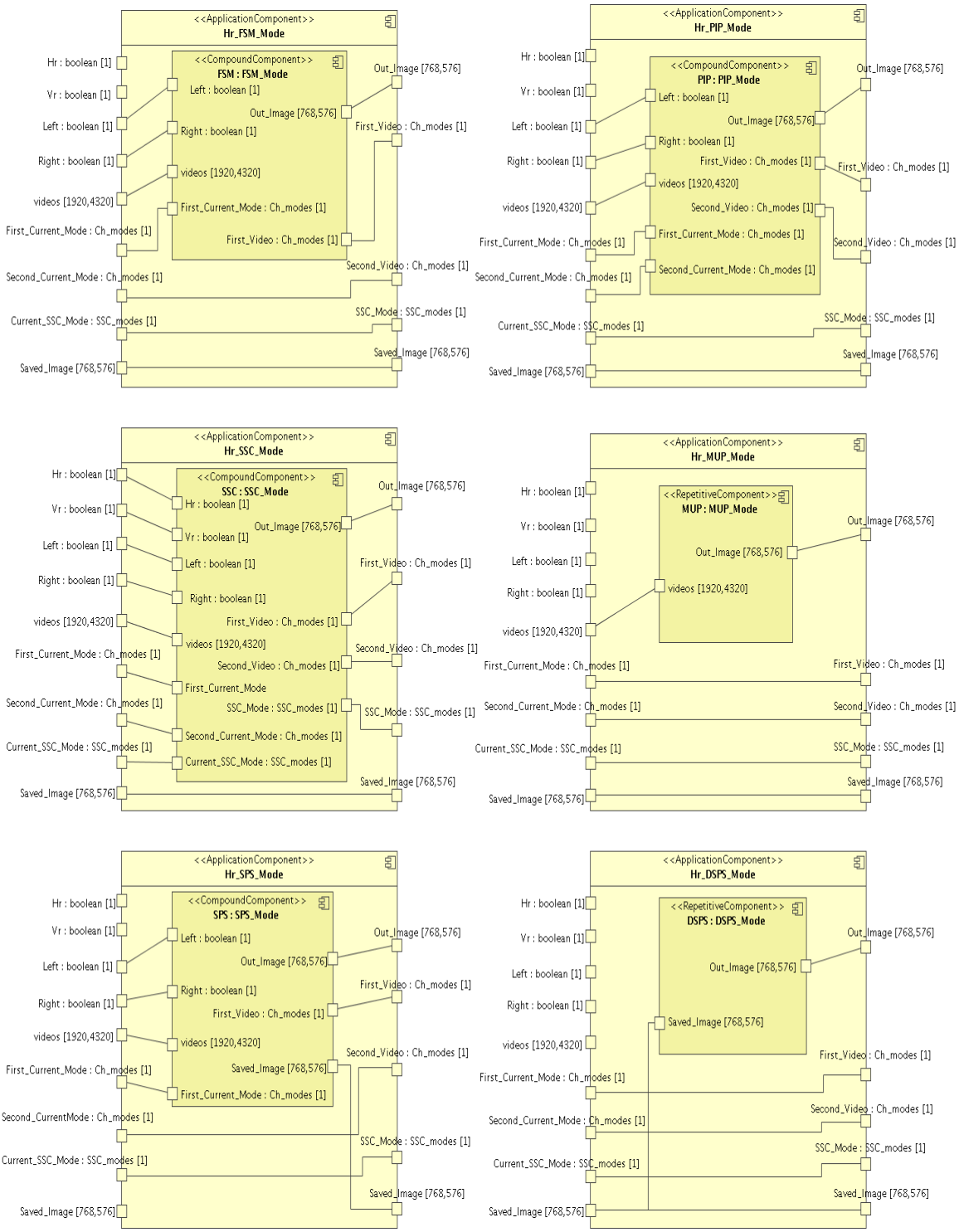


FIG. 9.39 – *Introduction d'un niveau de hiérarchie pour assurer la même interface aux différents modes de fonctionnement de l'application* MMVP

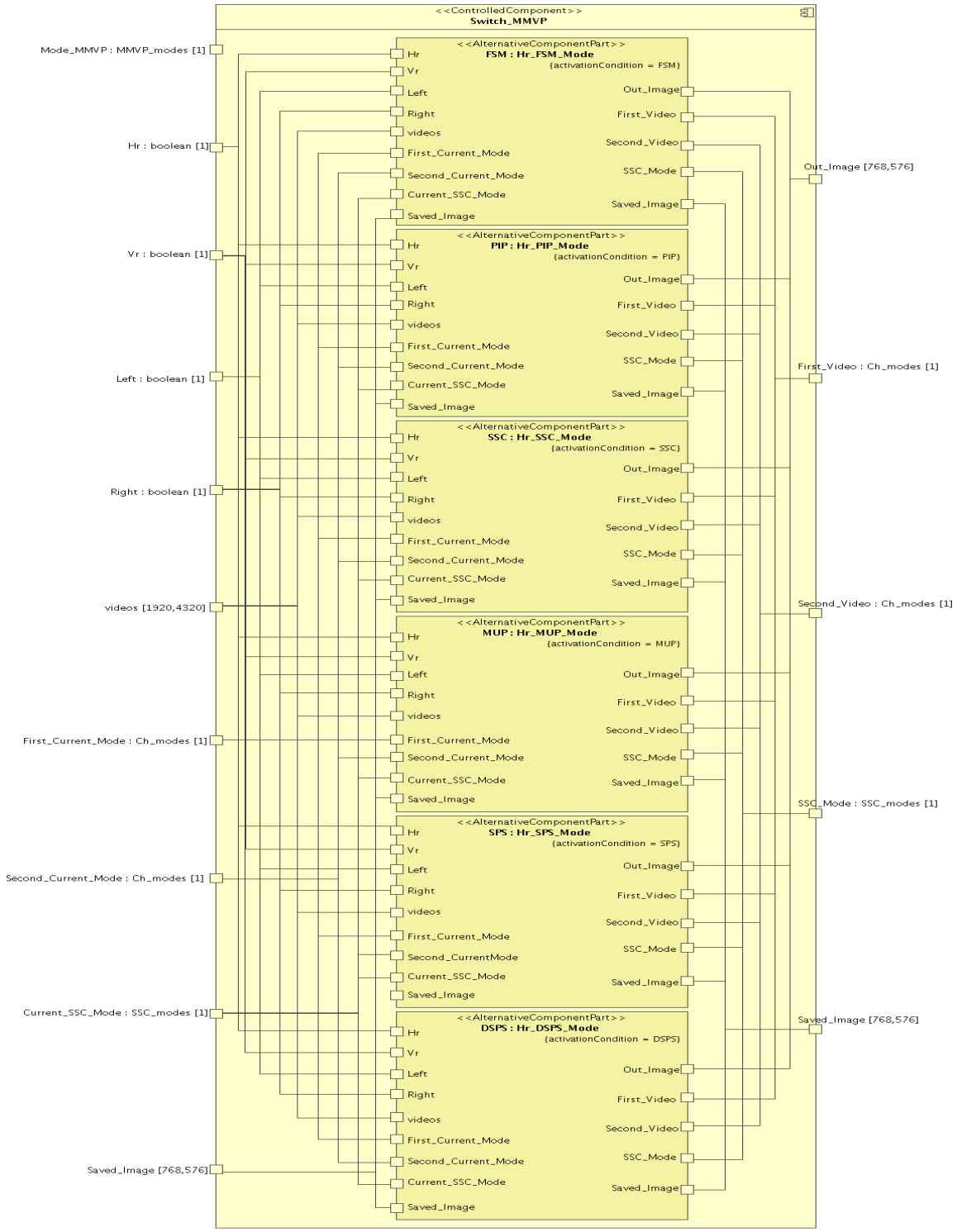


FIG. 9.40 – *Modélisation du switch entre les modes de fonctionnement de l'application* MMVP *pour la génération d'une seule image en sortie*

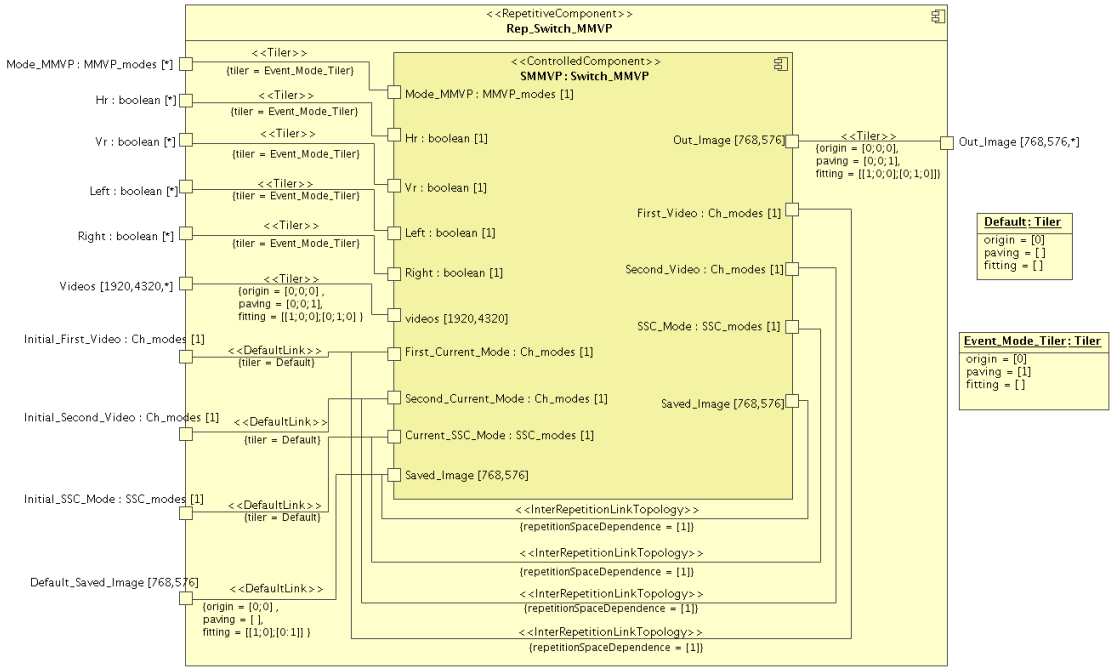


FIG. 9.41 – *Modélisation de la répétition sur le composant* `Switch_MMVP` *pour la génération de toutes les images en sortie*

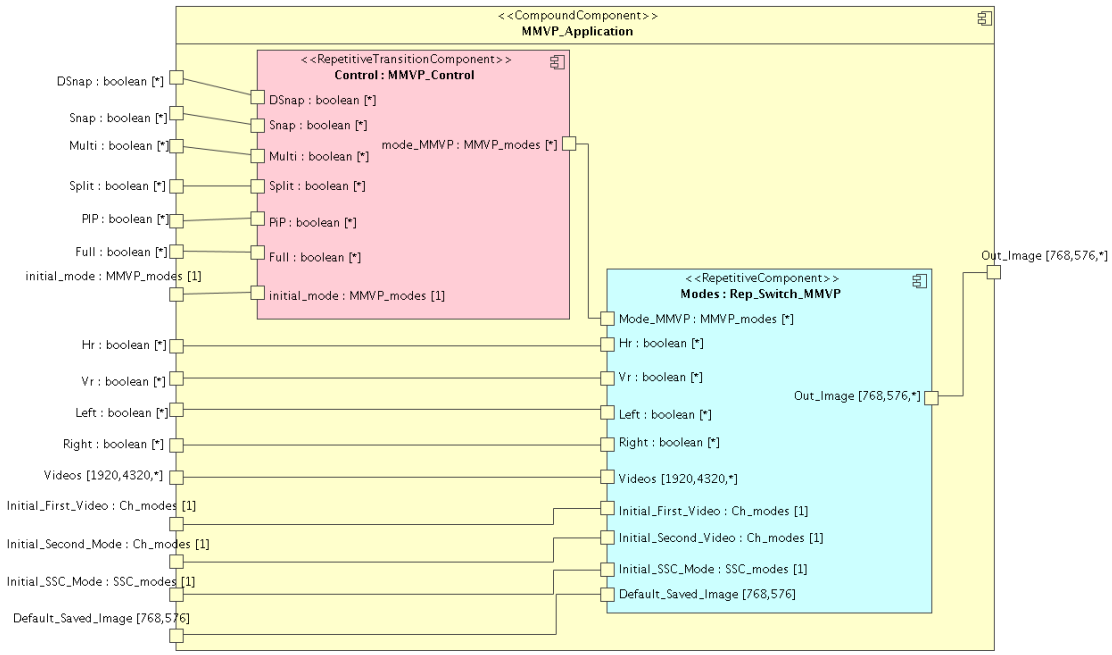


FIG. 9.42 – *Modélisation du comportement global de l'application* MMVP

9.4 Synthèse et conclusion

Nous avons présenté dans ce chapitre une étude de cas d'une application de traitement de vidéo à multi-modes de fonctionnement. La description de cette application contient un mélange de traitement de données massivement parallèle et du contrôle. Il est donc intéressant de modéliser son comportement en utilisant le profil GASPARD2 principalement conçu pour ce type d'applications, et en particulier la partie contrôle de ce profile permettant de prendre en considération le comportement réactif des applications parallèles.

L'étude de l'application MMVP a montré que les concepts introduits dans le profil GASPARD2 sont suffisant pour la modélisation et l'étude de ce genre de systèmes mixant des traitements de données parallèle et du contrôle. Cette étude représente donc un exemple de référence pour l'utilisation des concepts de contrôle introduit dans le profil GASPARD2 pour la représentation du comportement réactif tout en respectant sa sémantique de base.

Conclusion Générale et Perspectives

Chapitre 10

Conclusion et perspectives

10.1 Bilan

Les travaux présentés dans ce document s'inscrivent dans le cadre des recherches menées sur la modélisation et la conception des systèmes sur puce à hautes performances pour les applications de traitement systématique à parallélisme massif. Dans cette thèse, nous nous sommes intéressés à la modélisation au plus haut niveau d'abstraction de ces applications, et en particulier, à l'introduction des comportements de contrôle dans leur description. Notre objectif était de répondre à la question suivante : *comment modéliser des comportements de contrôle dans des applications de traitement systématique à parallélisme massif ?*. Cette question est formulée à la base de trois mots clés qui touchent trois domaines de recherches différents : la modélisation des systèmes informatiques, l'étude des comportements de contrôle, et les applications de traitement systématique à parallélisme massif. Pour pouvoir répondre à cette question, il était nécessaire d'étudier les travaux réalisés et le lien entre ces différents domaines de recherche.

Nous avons présenté les différents modèles de calcul existant pour la spécification des applications de traitement systématique à parallélisme massif, et nous avons conclu que le modèle ARRAY-OL semblait bien adapté pour la description de l'aspect parallèle et multidimensionnel des données. Cependant, ce modèle ne permet pas l'expression des comportements de contrôle qui sont généralement indispensables dans la description de certaines applications de traitement du signal largement utilisées dans notre quotidien. Le but de notre travail était donc de proposer un modèle de spécification introduisant la notion de contrôle dans le modèle ARRAY-OL pour permettre la prise en compte d'une plus grande catégorie d'applications, mixant des traitements de données parallèles et du contrôle.

Afin d'associer la notion de contrôle aux applications parallèles décrites en ARRAY-OL, nous avons étudié les travaux réalisés autour des systèmes réactifs synchrones, et en particulier celles permettant la description des systèmes hybrides. Cette étude nous a permis de constater que les approches existantes pour la spécification des systèmes hybrides n'imposent aucune méthodologie de conception permettant de séparer clairement entre la partie contrôle et les différentes parties de traitement. Ce mélange peut rendre difficile l'étude, la maintenance et la réutilisation des différentes parties de l'application étudiée. Pour remplir cette lacune, nous avons proposé une méthodologie de conception permettant de séparer clairement le contrôle et le calcul, et donc de favoriser leur étude séparée. Nous avons également montré via une étude de cas les avantages de cette méthodologie de séparation, notamment en ce qui concerne l'application des processus de vérification formelle.

Notre méthodologie de séparation contrôle/données nous a facilité la tâche pour l'introduction du contrôle dans les applications décrites en ARRAY-OL dans le sens où nous n'avions pas besoin de modifier la sémantique de base du modèle ARRAY-OL pour pouvoir prendre en considération l'aspect réactif. Cependant, la différence sémantique entre le modèle de contrôle, basé sur une sémantique de flot, et le modèle ARRAY-OL, basé sur une sémantique de dépendance de donnée, rendait difficile le couplage de ces deux mondes différents. Pour résoudre ce problème, nous avons proposé une approche basée sur le concept de degré de granularité permettant de définir les différents moments de prise en compte des valeurs de contrôle, et d'associer une sémantique de flot au modèle ARRAY-OL. Notre objectif était d'étudier la synchronisation entre la partie contrôle et les différentes parties de calcul tout en assurant les fonctionnalités attendues de l'application. Nous avons également étudié la possibilité d'étendre notre approche aux cas plus complexes, via le concept de multi-degrés de granularité, permettant ainsi de prendre en considération plusieurs niveaux

de contrôle et selon différents rythmes de fonctionnement.

Puisque notre étude était restreinte à la modélisation au plus haut niveau d'abstraction des applications mixant des traitements de données parallèles et du contrôle, nous avons étudié l'approche IDM et la modélisation UML pour les appliquer dans notre contexte de travail. Nous nous somme consacré en particulier à l'étude de la modélisation des concepts de contrôle dans les applications décrites en ARRAY-OL. Pour ce faire, nous avons proposé une codification de l'automate de contrôle sous forme d'une fonction de transition et une dépendance explicite entre ses différentes répétitions. Cette représentation permet la modélisation d'un automate de contrôle, et d'introduire une notion de flot dans le modèle ARRAY-OL sans avoir à modifier sa sémantique de base. Nous avons également proposé des concepts pour la représentation des différents modes de fonctionnement, et le lien entre l'automate de contrôle et la partie contrôlée. Sachant que notre travail s'inscrit dans le cadre du projet de développement de l'environnement GASPARD2, nous avons contribué à ce projet en intégrant nos résultats dans la description de la nouvelle version du profil GASPARD2 permettant ainsi d'offrir un support plus complet pour la modélisation à haut niveau d'abstraction des applications mixant des traitements de données parallèles et du contrôle.

10.2 Perspectives

Le travail présenté dans ce document synthétise les travaux réalisés dans le cadre d'une contribution à l'étude du comportement réactif des systèmes sur puce à hautes performances. Ce travail peut être continué dans plusieurs directions, et ses perspectives d'évolution concernent principalement :

- la création de liens entre les modèles de GASPARD2 et la technologie synchrone,
- l'étude de l'introduction de la notion d'horloge dans le modèle ARRAY-OL,
- la mise en relation du mécanisme de multi-degrés de granularité avec les processus de calcul d'horloge et de fusion,
- l'intégration de la modélisation du contrôle dans la chaîne de transformation de GASPARD2,
- l'étude de la possibilité des changements de TILERs, et
- l'introduction du contrôle dans les parties architecture et association du profil GASPARD2.

GASPARD2 et la technologie synchrone

La problématique d'associer un comportement réactif aux applications de traitement parallèle et intensif décrites selon le modèle de spécification ARRAY-OL peut être également abordée en proposant un modèle synchrone pour ces applications. Ceci peut être réalisé via la transformation des applications à parallélisme de données vers des équations synchrones. Il s'agit donc de transformer, selon l'approche IDM, les modèles exprimés selon les métamodèles GASPARD2 vers un système d'équations synchrones pouvant être traduit vers un langage de spécification synchrone tels que LUSTRE, LUCID SYNCHRONE et SIGNAL. Les modèles synchrones résultants permettent de répondre à plusieurs questions liées à la validation formelle des fonctionnalités de l'application étudiée, et de tirer profit des différents outils et techniques formelles offerts par la technologie synchrone. Ils permettent également la description des aspects non fonctionnels dans des applications de traitement de données

intensives. Ainsi, il peut être intéressant de comparer les résultats d'analyse et les codes générés directement dans la chaîne de GASPARD2 avec ceux obtenus à partir du modèle synchrone correspondant. Ce travail est au coeur d'un sujet de thèse[65] lancé au sein de l'équipe West.

Introduction de la notion d'horloge

La difficulté principale pour la description du comportement réactif dans une application de traitement parallèle décrite en ARRAY-OL est la description des différents instants de prise en compte des valeurs de contrôle. Ceci est dû au fait que le modèle ARRAY-OL ne donne aucun moyen pour la définition de la notion d'horloge et se base sur l'hypothèse du temps banalisé. Il s'avère donc important d'étudier l'introduction de la notion d'horloge dans le modèle ARRAY-OL et son impact sur sa sémantique de base pour permettre, d'une part, de faciliter l'étude des comportements réactifs, et d'autre part, de garder la puissance d'expression du comportement parallèle du modèle ARRAY-OL. De point de vue modélisation, l'introduction de la notion d'horloge au modèle ARRAY-OL va faciliter la représentation des composants de contrôle sous forme d'automates, et par conséquent, permettre l'utilisation de la structure des machines à états définis en UML pour la représentation de la partie contrôle dans un modèle GASPARD2.

Multi-degrés de granularité et processus de calcul d'horloge et de fusion

A la fin du chapitre 7, nous avons discuté de la spécification des scénarios complexes d'applications de traitement parallèle contrôlées par plusieurs parties de contrôle et selon différents rythmes de fonctionnement. Nous avons montré que l'étude de ces applications nécessite la synchronisation du fonctionnement des différentes parties du contrôle avec celles de traitement parallèle afin d'assurer les fonctionnalités attendus. C'est un problème complexe qui doit faire appel aux processus de calcul d'horloge et de fusion pour déterminer le comportement global de l'application étudiée qui doit rester déterministe, parallèle et compositionnel. Cette problématique fait récemment partie des sujets de recherche abordés au sein de l'équipe West[66].

Introduction dans la chaîne de conception GASPARD2

Le travail présenté dans ce document s'inscrit dans le cadre du projet de développement de l'environnement GASPARD2. La version actuelle de cet environnement est disponible sous forme d'un plug-in Eclipse permettant d'offrir un outil complet pour la modélisation des systèmes sur puce à hautes performances. Actuellement, l'environnement GASPARD2 est basé sur la version précédente du profil qui ne prend pas en considération la notion du contrôle. Il est envisagé à court terme d'intégrer la nouvelle version du profil, permettant la description des comportements du contrôle, dans la chaîne de transformation de GASPARD2

[65] Thèse de Huafeng Yu commencée en octobre 2005.
[66] Thèse de Calin Glitia commencée en septembre 2006.

Changement de modes pour les TILERs

Nous avons étudié l'introduction du contrôle pour les applications parallèles décrites en ARRAY-OL en se limitant à la description des changements de mode pour les tâches de calcul. Cependant, il est également possible d'avoir besoin de changer une ou plusieurs information des TILER en entrée et/ou en sortie en fonction des valeurs d'événement de contrôle. Ceci permet la réutilisation des mêmes tâches de calcul dans des contextes variés où les tableaux en entrée et en sortie peuvent être parcourus différemment. L'étude des changements de TILERs représente une problématique complexe qui doit assurer la génération de tous les points des tableaux en sortie tout en respectant les contraintes de base du modèle ARRAY-OL.

Introduction du contrôle dans les parties architecture et association

Notre contribution sur l'introduction du contrôle pour la conception des systèmes sur puce à hautes performances se situe dans la partie application du modèle *Y* de GASPARD2. Il est également intéressant d'étudier l'introduction du contrôle pour les parties architecture et association de ce modèle. L'introduction du contrôle dans la partie architecture permet la description des architectures reconfigurables qui offre la possibilité d'utiliser les ressources matérielles nécessaires en fonction de l'application à réaliser, tandis que l'introduction du contrôle dans la partie association permet une meilleure utilisation des algorithmes de placement et d'ordonnancement qui peuvent être adaptés à l'application et l'architecture utilisées.

Bibliographie

[ABD05] Abdelkader Amar, Pierre Boulet, and Philippe Dumont. Projection of the ARRAY-OL specification language onto the kahn process network computation model. In *International Symposium on Parallel Architectures, Algorithms, and Networks*, Las Vegas, Nevada, USA, décembre 2005.

[ACdT06] Charles André, Arnaud Cuccuru, Robert de Simone, and Jean-Pierre Talpin. Modeling with logical time in UML for real-time embedded system design. Rapport de recherche RR-5895, INRIA, avril 2006.

[Ade01] Emmanuel Ademovië. Théorie & traitement du signal, 2001. `http://rsa.esigetel.fr/Doc/Supports_A12/Tns/Supports/Signal.pdf`.

[Ala05] D. Alazard. Introduction au filtre de kalman, janvier 2005. `http://personnel.supaero.fr/alazard-daniel/Pdf/cours_Kalman.pdf`.

[And96] Charles André. Representation and analysis of reactive behaviors : A synchronous approach. *CESA'96, IEEE-SMC*, juillet 1996. `http://www.i3s.unice.fr/sports/SyncCharts/TR96-28.pdf`.

[And03] Charles André. Semantics of S.S.M. (Safe State Machine). Rapport de recherche RR200324, Université Sophia Antipolis, Nice, avril 2003.

[APR01] C. André, M-A Peraldi-Frati, and J-P. Rigault. Scenario and property checking of real-time systems using a synchronous approach. In *Fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2001)*, Magdeburg, Germany, mai 2001.

[APR02] C. André, M-A. Peraldi-Frati, and J-P. Rigault. Integrating the synchronous paradigm into UML : Application to control-dominated systems. In *UML '02 : Proceedings of the 5th International Conference on The Unified Modeling Language*, pages 163–178, London, UK, octobre 2002. Springer-Verlag.

[Aut98] *Automatica, Special Issue on Hybrid Systems*, 35(3), mars 1998.

[BAMP83] Mordechai Ben-Ari, Zohar Manna, and Amir Pnueli. The temporal logic of branching time. *Acta Informatica*, 20 :207–226, 1983.

[BB91] Albert Benveniste and Gérard Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9) :1270–1282, septembre 1991.

[BBDS03] Luciano Baresi, Francesco Bruschi, Elisabetta Di Nitto, and Donatella Sciuto. *System Specification and Design Languages*, chapter SystemC Code Generation from UML Models, pages 129–141. The ChDL series. Kluwer Academic Publishers, 2003.

[BBG+99] Loïc Besnard, Patricia Bournai, Thierry Gautier, Nicolas Halbwachs, Simin Nadjm-Tehrani, and Annie Ressouche. Design of a multi-formalism application and distribution in a dataflow context : An example. In World Scientific, editor, *Proceedings of the 12th international Symposium on Languages for Intentional programming*, Athens, juin 1999.

[BBGG85] Albert Benveniste, Patricia Bournai, Thierry Gautier, and Paul Le Guernic. SIGNAL : a data flow oriented language for signal processing. Technical Report RR-0378, INRIA, centre de Rennes IRISA, mars 1985.

[BCE+03] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1), janvier 2003.

[BCM+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking : 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[Bey04] Veerle Beyst. PROSPER : Project for research on speed adaptation policies on european roads, final report on stakeholder analysis, version 1.4. Rapport technique, 2004.

[BG92] Gerard Berry and Georges Gonthier. The esterel synchronous programming language : Design, semantics, implementation. *Science of Computer Programming*, 19(2) :87–152, 1992.

[BK97] Andrew D. Bagdanov and Junichi Kanai. Projection profile based skew estimation algorithm for JBIG compressed images. In *ICDAR '97 : Proceedings of the 4th International Conference on Document Analysis and Recognition*, pages 401–406, Washington, DC, USA, 1997. IEEE Computer Society.

[BNLD00] Yann Le Bianic, Eric Nassor, Emmanuel Ledinot, and Sylvan Dissoubray. UML object specification for real-time software. *RTS : Real-Time Systems*, mars 2000.

[Boc03] Conrad Bock. UML 2 activity and action models. *Journal of Object Technology*, 2(4) :43–53, 2003. `http://www.jot.fm/issues/issue_2003_07/column3.pdf`.

[Bon95] Frédéric Boniol. Synchronous communicating reactive processes. *Models and Proofs, 2nd AMAST Workshop on Real-Time Systems, Bordeaux, France*, juin 1995.

[Bon97] Frédéric Boniol. Une approche synchrone multi-langages pour la conception de systèmes embarqués. *9èmes Rencontres sur le Parallélisme*, mai 1997.

[Bon98] Frédéric Boniol. Une approche synchrone multi-formalismes pour la conception de systèmes temps-réel distribués. *Technique et science informatiques, Hermès*, 17(9) :1099–1128, 1998.

[Boo91] Grady Booch. *Object-Oriented Design with Application*. 1991.

[Bou02] Pierre Boulet. Contributions aux environnements de programmation pour le calcul intensif. Habilitation à diriger des recherches, Univeité des Sciences et Technologies de Lille, décembre 2002.

[BRG+01] J-R Beauvais, E. Rutten, T. Gautier, R. Houdebine, and Y.M. Tang. Modelling statecharts and activitycharts as signal equations. In *ACM Transactions on Software Engineering and Methodology*, volume 10, pages 397–451, octobre 2001.

[BV02] Torbjörn Biding and Vägverket. Intelligent speed adaptation (isa), results of large-scale trials in borlänge, lidköping, lund and umeå during the period 1999-2002. Rapport technique, 2002.

[CCM+03] Paul Caspi, Adrian Curic, Aude Maignan, Christos Sofronis, Stavros Tripakis, and Peter Niebert. From simulink to scade/lustre to tta : a layered approach for distributed embedded applications. In *LCTES'03 : Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 153–162, New York, NY, USA, 2003. ACM Press.

[CDMB05] Arnaud Cuccuru, Jean-Luc Dekeyser, Philippe Marquet, and Pierre Boulet. Towards uml 2 extensions for compact modeling of regular complex topologies - a partial answer to the marte rfp. In *In MoDELS/UML 2005, ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems*, pages 445–459, Montego Bay, Jamaica, octobre 2005.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Proceedings of the IBM Workshop on Logic of Programs*, pages 52–71, London, UK, 1981. Springer-Verlag.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications : Apractical approach. In *ACM Transaction on Programming Languages and Systems*, volume 8, pages 244–263, 1986.

[Cha93] C. C. Chan. An overview of electric vehicle technology. In *Proceedings of the IEEE*, volume 81, pages 1202–1213, septembre 1993.

[Cho05] Thomas Choquet. Analyse et développement d'un logiciel d'aide à la vision. Mémoire de DESS, Université Vincennes Saint-Denis Paris 8, septembre 2005.

[CHP06] Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet. Mixing signals and modes in synchronous data-flow systems. Submitted to publication, mai 2006. `http://www.lri.fr/$\sim$pouzet/bib/bib.html`.

[Coo95] Jeremy R. Cooperstock. Making the user interface disappear : the reactive room. In *CASCON'95 : Proceedings of the 1995 conference of the Centre for Advanced Studies on Collaborative research*, Toronto, Ontario, Canada, 1995. IBM Press.

[CP03] Jean-Louis Colaço and Bruno Pagano. SSM♯ : a qualificable and LUSTRE compliant subset of ESTEREL's safe state machines. Esterel Technoogies paper, 2003.

[CPHP87] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE : a declarative language for real time programming. pages 178–188, Munich, West Germany, janvier 1987.

[CPP05] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. A conservative extension of synchronous data-flow with state machines. In *EMSOFT'05 : Proceedings of the 5th ACM international conference on Embedded software*, pages 173–182, New York, NY, USA, 2005. ACM Press.

[Cra06] Michelle L. Crane. On the syntax and semantics of state machines. submitted to the School of Computing Queen's University, février 2006. `http://www.cs.queensu.ca/$\sim$crane/stuff/publications/depthfinal~hyperlinked.pdf`.

[Cuc05] Arnaud Cuccuru. *Modélisation unifiée des aspects répétitifs dans la conception conjointe logicielle/matérielle des systèmes sur puce à haute performances*. Thèse de doctorat, Laboratoire d'Informatique Fondamentale de Lille, Univeité des Sciences et Technologies de Lille, septembre 2005.

[dA03] Robert de Simone and Charles André. Towards a "synchronous reactive" UML subprofile ? *Workshop on Specification and Validation of UML Models for Real Time and Embedded Systems (SVERTS)*, octobre 2003.

[DaSHJJB05] Jun-Hyeong Do, Hyoyoung Jang an Sung Hoon Jung, Jinwoo Jung, and Zeungnam Bien. Soft remote control system in the intelligent sweet home. In *Intelligent Robots and Systems (IROS)*, pages 3984–3989. IEEE/RSJ International Conference, août 2005. ISBN : 0-7803-8912-3.

[DB05] Philippe Dumont and Pierre Boulet. Another multidimensional synchronous dataflow : Simulating ARRAY-OL in PTOLEMY II. Rapport de recherche RR-5516, INRIA, mars 2005.

[DDMS95] A. S. Debelack, J. D. Dehn, L. L. Muchinsky, and D. M. Smith. Next generation air traffic control automation. *IBM Systems Journal*, 34(1) :63–77, 1995.

[de 97] Florant Dupont de Dinechin. *Systèmes structurés d'équations récurrentes : mise en oeuvre dans le langage* ALPHA *et applications*. Thèse de doctorat, Université de Rennes 1, 1997.

[Dio04] Bernard Dion. Correct-by-construction methods for the development of safety-critical applications. *Conferences SAE World Congress*, (04AE-129), 2004.

[DLB+95] Alain Demeure, Anne Lafarge, Emmanuel Boutillon, Didier Rozzonelli, Jean-Claude Dufourd, and Jean-Louis Marro. ARRAY-OL : Proposition d'un formalisme tableau pourle traitement de signal multi-dimensionnel. In *Gretsi*, Juan-Les-Pins, France, septembre 1995.

[Dou04] Bruce Powel Douglass. *Real Time* UML *Third Edition, Advances in the* UML *for Real-Time Systems*. Object Technology Series. Douglass, 2004.

[dQRR99] F. Dupont de Dinechin, P. Quinton, S. Rajopadhye, and T. Risset. First steps in alpha. Rapport de recherche 1244, Irisa, Rennes, juin 1999.

[Dum05] Philippe Dumont. *Spécification multidimentionnelle pour le traitement du signal systématique*. PhD thesis, Laboratoire d'Informatique Fondamentale de Lille, Université des Sciences et Technologies de Lille, décembre 2005.

[EG03] M.D. Edwards and P.N. Green. UML for hardware and software object modeling. In *UML for real : design of embedded real-time systems*, pages 127–147. Kluwer Academic Publishers, 2003.

[FB91] Robert de Simone Frédéric Boussinot. The esterel language. *Proceedings of the IEEE*, 79(9) :1293–1304, septembre 1991.

[FW95] William T. Freeman and Craig D. Weissman. Television control by hand gestures. *IEEE Intl. Wkshp. on Automatic Face and Gesture Recognition*, juin 1995.

[GE02] P. Green and M. Edwards. Platform modelling with UML and SystemC. *FDL'02 : Forum on specification and Design Language*, septembre 2002.

[GE04] Peter Green and Salah Essa. Integrating the synchronous dataflow model with UML. In *Proceedings of the Design, Automation and Test in Europ Conference and Exhibition (DATE'04)*, Washington, DC, USA, 2004. IEEE Computer Society.

[GEE02] P.N. Green, M.D. Edwards, and S. Essa. HASOC : Towards a new method for system-on-a-chip development. In *Design Automation for Embedded Systems*, volume 6, pages 333–353. Springer, juillet 2002.

[GGBM91] Paul Le Guernic, Thierry Gautier, Michel Le Borgne, and Claude Le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9) :1321–1336, septembre 1991.

[GK83] Daniel Gajski and Robert H. Kuhn. Improving data locality with loop transformations. *IEEE Computer*, 16(12) :11–14, 1983.

[GLL99] Alain Girault, Bilung Lee, and Edward A. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions On Computer-Aided Design Of Integrated Circuits And Systems*, 18(6) :742–760, juin 1999.

[Gre05] Peter Green. *UML for SoC Design*, chapter UML as a Framework for Combining Different Models of Computation, pages 37–62. Springer, 2005.

[Gro04] Object Management Group. Unified modeling language : Supetructure, version 2.0, juillet 2004. `http://www.omg.org/cgi-bin/doc?formal/05-07-04`.

[Gro05a] Object Management Group. Mof qvt final adopted specification. OMG document formal/05-11-01, novembre 2005. `http://www.omg.org/docs/ptc/05-11-01.pdf`.

[Gro05b] Object Management Group. Ocl 2.0 specification, veion 2.0, juin 2005. `http://www.omg.org/docs/ptc/05-06-06.pdf`.

[Gro05c] Object Management Group. UML extension profile for SoC RFC. OMG document realtime 2005-03-01, mars 2005. `http://www.omg.org/cgi-bin/doc?realtime/2005-03-01`.

[Gro05d] Object Management Group. UML^{TM} profile for schedulability, performance, and time specification. OMG document formal/05-01-02, janvier 2005. `http://www.omg.org/cgi-bin/doc?formal/2005-01-02`.

[Gro05e] ProMARTE Working Group. MARTE initial submission. OMG Document realtime/05-11-01, novembre 2005. `http://www.omg.org/cgi-bin/doc?realtime/2005-11-01`.

[GRY+06] Abdoulaye Gamatié, Eric Rutten, Huafeng Yu, Pierre Boulet, and Jean-Luc Dekeyser. Synchronous modeling of data intensive applications. Rapport de recherche RR-5876, INRIA, avril 2006.

[Gun01] Martin Gunnarsson. Parameter estimation for fault diagnosis of an automotive engine using extended kalman filters. Master's thesis, Vehicular Systems Dept. of Electrical Engineering at Linköpings Universitet, novembre 2001.

[Gup93] Vineet Gupta. Concurrent kripke structures. In *Proceedings of the North American Process Algebra Workshop, Cornell CS-TR-93-1369*, août 1993.

[Hal99] Nicolas Halbwachs. Tableaux en LUSTRE-V6. Verimag Paper, juin 1999.

[Har87] David Harel. Statecharts : A visual formalism for complex systems. *Science of Computer Programming*, 8(3) :231–274, juin 1987.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9) :1305–1320, septembre 1991.

[HLL+03] Christopher Hylands, Edward A. Lee, Jie Liu, Xiaojun Liu, Steve Neuendorffer, Yuhong Xiong, and Haiyang Zheng. PTOLEMY II heterogeneous concurrent modeling and design in JAVA, volume 1 : Introduction to PTOLEMY II. Technical Report UCB/ERL M03/27, EECS Department, University of California, Berkeley, août 2003. `http://www.enee.umd.edu/DSPCAD/papers/bhat2002x3.pdf`.

[HLR93] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In *AMAST'93 : Proceedings of the Third International Conference on Algebraic Methodology and Software Technology*. Springer Verlag, juin 1993.

[HP85] David Harel and Amir Pnueli. On the development of reactive systems. In Springer-Verlag NATO ASI Series, editor, *Logics and Models of Concurrent Systems*, volume 13, pages 477–498, New York, 1985.

[HP88] D.J. Hatley and I.A. Pirbhai. *Strategies for Real-Time Specification*. Dorset House Publishing, 1988.

[HP98] Leszek Holenderski and Axel Poigné. The multi-paradigm synchronous programming language LEA. *Formal Techniques for Hardware and Hardware-like Systems (FTH'98)*, 1998.

[IEE98] *IEEE Transaction on Automatic Control, Special Issue on Hybrid Systems*, 43(4), avril 1998.

[IEE00] *IEEE Transaction on Automatic Control, Special Issue on Hybrid Systems*, 88(7), juillet 2000.

[Ike02] Kentaro Ikemoto. Adaptive coding schemes using finite state machine for software defined radio. *European Wireless*, pages 884–888, 2002.

[JBR97] Ivar Jacobson, Grady Booch, and James Rumbaugh. *Unified Modeling Language Reference Manual*. 1997.

[JCJO92] Ivar Jacobson, Magnus Christeon, Patrik Jonsson, and Gunnar Overgaard. *Object-Oriented Software Engineering - A Use Case Driven Approach*. 1992.

[JLMR94] M. Jourdan, F. Lagnier, F. Maraninchi, and P. Raymond. A multiparadigm language for reactive systems. In *IEEE International Conference on Computer Languages (ICCL)*, Toulouse, France, mai 1994.

[JLPR04] C. S. Jensen, H. Lahrmann, S. Pakalnis, and J. Runge. The INFATI data. Rapport technique, TimeCenter, 2004.

[Jou94] Muriel Jourdan. *Etude d'un environnement de programmation et de vérification des systèmes réactifs, multi-langages et multi-outils*. Thèse de doctorat, Université Joseph-Fourier, Grenoble I, 1994.

[Ken02] Stuart Kent. Model driven engineering. In *IFM*, pages 286–298, 2002.

[KMW67] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *J. ACM*, 14(3) :563–590, juillet 1967.

[LD06] Alain Le Guennec and Bernard Dion. Bridging UML and safety-critical software development environments. *3rd European Congress ERTS : Embedded Real TimeSoftware*, janvier 2006. `http://www.esterel-technologies.com/files/ERTS2006-paper-SCADE-UML.pdf`.

[LDB05] Ouassila Labbani, Jean-Luc Dekeyser, and Pierre Boulet. Mode-automata based methodology for SCADE. In *Hybrid Systems : Computation and control, 8th international workshop*, LNCS series 3414, pages 386–401, Zurich, Switzerland, mars 2005. Springer.

[LDBR05] Ouassila Labbani, Jean-Luc Dekeyser, Pierre Boulet, and Éric Rutten. Introducing control in the gaspard2 data-parallel metamodel : Synchronous approach. In *International Workshop MARTES : Modeling and Analysis of Real-Time and Embedded Systems (in conjunction with 8th International Conference on Model Driven Engineering Languages and Systems, MoDELS/UML 2005)*, Montego Bay, Jamaica, octobre 2005. `http://www.lifl.fr/west/publi/LDBR05.pdf`.

[LDBR06] Ouassila Labbani, Jean-Luc Dekeyser, Pierre Boulet, and Éric Rutten. Uml2 profile for modeling controlled data parallel applications. In *FDL'06 : Forum on specification and Design Languages*, Darmstadt, Allemagne, septembre 2006.

[LDR06] Ouassila Labbani, Jean-Luc Dekeyser, and Éric Rutten. Separating control and data flow : Methodology and automotive system case study. Rapport de recherche, INRIA, février 2006.

[LLLK00] Honam Lee, Bonggeun Lee, Youngho Lee, and Bongsoon Kang. Optimized VLSI design for enhanced image downscaler. In *Proceedings of the Second IEEE Asia Pacific Conference*, pages 139–142, Cheju, South Korea, 2000.

[LM87a] Edward A. Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. on Computers*, 36(1) :24–35, janvier 1987.

[LM87b] Edward A. Lee and David G. Messerschmitt. Synchronous Data Flow. In *Proceedings of the IEEE*, volume 75, pages 1235–1245, septembre 1987.

[LMLD06] Sébastien Le Beux, Philippe Marquet, Ouassila Labbani, and Jean-Luc Dekeyser. FPGA implementation of embedded cruise control and anti-collision radar. In *DSD'06, 9th Euromicro conference on Digital System Design : Architectures, Methods and Tools*, Dubrovnik, Croatia, août 2006.

[LRD06] Ouassila Labbani, Éric Rutten, and Jean-Luc Dekeyser. Safe design methodology for an intelligent cruise control system with GPS. In *64th IEEE Vehicular Technology conference*, Montréal, Québec, Canada, septembre 2006.

[LS92] C. Lavarenne and Y. Sorel. Specification, performance optimization and executive generation for real-time embedded multiprocessor applications with SynDEx. In *Proceedings of Real-Time Embedded Processing for Space Applications*, Les Saintes Maries de la Mer, France, novembre 1992. CNES International Symposium.

[LSSS91] C. Lavarenne, O. Seghrouchni, Y. Sorel, and M. Sorine. The SynDEx software environment for real-time distributed systems, design and implementation. In *Proceedings of European Control Conference, ECC'91*, Grenoble, France, juillet 1991.

[Mar92] Florence Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In W.R. Cleaveland, editor, *3rd International Conference on Concurrency Theory, CONCUR'92*, volume 630 of *LNCS*, pages 550–564, Stony Brook, USA, août 1992. Springer-Verlag.

[Mar97] Florence Maraninchi. Modélisation et validation des systèmes réactifs : un langage synchrone à base d'automates. Habilitation à diriger des recherches, Univeité Joseph Fourier - Grenoble I, mai 1997.

[Mau89] Christophe Mauras. *Alpha : un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones*. PhD thesis, Université de Rennes I, décembre 1989.

[ME05] Mrinal Mandal and Ehab Elmallah. Guest editorial : special issue on multimedia computing and communications. In *Canadian Journal of Electrical and Computer Engineering*, volume 30, pages 07–09. IEEE Canada, 2005.

[MES03] Michel Marchi, Jacques Ehrlich, and Laurent Salesse. LAVIA : the french isa project, main issues and first results on technical tests. In *Proceedings of the 10th ITS Congress*, Madrid, Spain, 2003.

[MGH02] David R. Morse, Henrik S. Gedenryd, and Simon Holland. A simple, technology-neutral lingua franca for location systems, applied to combined indoor-outdoor navigation. Rapport technique, Department of Computing, The Open University, 2002.

[MH96] F. Maraninchi and N. Halbwachs. Compiling Argos into boolean equations. In *Formal Techniques for Real-Time and Fault-Tolerance (FTRTFT)*, Uppsala, Sweden, septembre 1996. Springer Verlag, LNCS.

[ML95] P.K. Murthy and Edward A. Lee. A generalization of multidimensional synchronous dataflow to arbitrary sampling lattices. Technical Report UCB/ERL M95/59, EECS Department, University of California, Berkeley, mars 1995.

[ML96] P. K. Murthy and Edward A. Lee. An extension of multidimensional synchronous dataflow to handle arbitrary sampling lattices. In *Int. Conf. Acoustics, Speech, and Signal Processing*, pages 3306–3309, mai 1996.

[ML02] Praveen K. Murthy and Edward A. Lee. Multidimensional synchronous dataflow. *IEEE Transactions on Signal Processing*, juillet 2002.

[MM03] Joaquin Miller and Jishnu Mukerji. MDA guide (draft version 0.2), janvier 2003. `http://www.omg.org/docs/ab/03-01-03.pdf`.

[MM05] Grant Martin and Wolfgang Müller, editors. *UML for SoC Design*. Springer, 2005.

[Moo56] Edward F. Moore. Gedanken-experiments on sequential machines. *Automata Studies, Annals of Mathematical Studies*, pages 129–153, 1956.

[Mor02] Lionel Morel. Efficient compilation of array iterators for lustre. In *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier, 2002.

[MR98] Florence Maraninchi and Yann Rémond. Mode-automata : About modes and states for reactive systems. In *European Symposium On Programming*, Lisbon, Portugal, mars 1998. Springer verlag.

[MR00] Florence Maraninchi and Yann Rémond. Applying formal methods to industrial cases : The language approach (the production-cell and mode-automata). In *5th International Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, Berlin, avril 2000. `http://www.inrialpes.fr/vasy/fmics`.

[MR01] Florence Maraninchi and Yann Rémond. Argos : An automaton-based synchronous language. *Computer Languages*, 27(1-3) :61–92, octobre 2001.

[MRR00] F. Maraninchi, Y. Rémond, and Y. Raoul. MATOU : An implementation of mode-automata into DC. In *Compiler Construction*, Berlin (Germany), mars 2000. Springer verlag.

[Mur96] Praveen Kumar Murthy. *Scheduling Techniques for Synchronous and Multidimensional Synchronous Dataflow*. Thèse de doctorat, University of California, Berkeley, CA, 1996.

[NS73] I. Nassi and B. Shneiderman. Flowchart techniques for structured programming. *ACM SIGPLAN Notices*, 8(8) :12–26, 1973.

[Owe04] Philippe Owezarski. Métrologie des réseaux de l'internet : principales actions et impact sur les évolutions technologiques. In *Colloque Mesures de l'internet*, avril 2004. `http://www.laas.fr/$\sim$owe/PUBLIS/owe_intro_metro_et_AS.pdf`.

[Pag04] Jean-Manuel Page. A final technical report on the belgian intelligent speed adaptation (ISA) trial. Technical report, Institut Belge pour la Sécurité Routière, 2004.

[Pal03] Grégory Pallone. *Dilatation et Transposition sous Contraintes Perceptives des Signaux Audio : Application au Transfert Cinema-Video*. Thèse de doctorat, Université d'Aix-Marseille II, juin 2003.

[PAR02] M-A. Péraldi-Frati, C. André, and J-P Rigault. UML et le paradigme synchrone : Application à la conception de contrôleurs embarqués. Rapport de recherche I3S/RR-2002-02-FR, Laboratoire I3S, Projet SPORTS, février 2002. `http://www.i3s.unice.fr/$\sim$mh/RR/2002/RR-02.02-C.ANDRE.pdf`.

[PMM+98] Axel Poigné, Matthew Morley, Olivier Maffeïs, Leszek Holendeki, and Reinhard Budde. The synchronous approach to designing reactive systems. *Formal Methods in System Design*, 12(2) :163–187, 1998.

[Poo01] John D. Poole. Model-driven architecture : Vision, standards and emerging technologies. In *Workshop on Adaptive Object-Models and Metamodeling, ECOOP 2001*, avril 2001.

[Pou06] Marc Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual*. Université Paris-Sud, LRI, avril 2006. `http://www.lri.fr/$\sim$pouzet/lucid-synchrone/`.

[PS03] Nicolas Pernet and Yves Sorel. From specification to optimized implementation of distributed real-time embedded systems mixing control and data processing. In *Proceedings of ISCA 16th International Conference : Computer Applications in Industry and Engineering, CAINE'03*, Las Vegas Nv, USA, novembre 2003.

[PVS+03] Marc Pauwels, Yves Vanderperren, Geert Sonck, Paul van Oostende, Wim Dehaene, and Trevor Moore. *System Specification and Design Languages*, chapter A Design Methodology for the Development of a Complex System-on-Chip using UML and Executable System Models, pages 129–141. The ChDL series. Kluwer Academic Publishers, 2003.

[QS82] J.P. Queille and J. Sifakis. Specification and verification of concurrent system in CESAR. In *Proceedings of the 5th International Symposium in Programming, Lecture Notes in Computer Science*, volume 137, pages 337–371, Berlin, New York, 1982.

[Rau02] Antoine Rauzy. Mode automata and their compilation into fault trees. *Reliability Engineering and System Safety*, 78 :1–12, 2002.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hal, New Jeey, 1991.

[RH91] F. Rocheteau and N. Halbwachs. Pollux, a lustre-based hardware design environment. In P. Quinton and Y. Robert, editors, *Conference on Algorithms and Parallel VLSI Architectures II*, Chateau de Bonas, juin 1991.

[RRFL99] Javad Razavilar, Farrohk Rashid-Farrokhi, and K. J. Ray Liu. Software radio architecture with smart antennas : a tutorial on algorithms and complexity. In *IEEE Journal on Selected Areas in Communications*, volume 17, pages 662–676, 1999.

[RSRB05] Elvinia Riccobene, Patrizia Scandurra, Alberto Rosti, and Sara Bocchio. A UML 2.0 profile for SystemC : toward high-level SoC design. In *ACM International Conference on Embedded Software : EMSOFT*, pages 138–141, 2005. `http://doi.acm.org/10.1145/1086228.1086254`.

[Rém01] Yann Rémond. *Un support langage pour les modes de fonctionnement des systèmes temps-réel : extension de LUSTRE par des automates de modes*. PhD thesis, Université Grenoble I - Joseph Fourier, octobre 2001.

[Sel96] Bran V. Selic. Real-time object-oriented modeling (ROOM). *RTAS : Real-Time Technology and Applications Symposium*, pages 214–217, juin 1996.

[Sel04] Bran V. Selic. On the semantic foundations of standard UML 2.0. In *SFM*, pages 181–199, 2004.

[SGG99] Irina M. Smarandache, Thierry Gautier, and Paul Le Guernic. Validation of mixed SIGNAL-ALPHA real-time systems through affine calculus on clock synchronisation constraints. In *World Congress on Formal Methods (2)*, pages 1364–1383, 1999.

[SJSL00] Slobodan N. Simic, Karl Henrik Johansson, Shankar Sastry, and John Lygeros. Towards a geometric theory of hybrid systems. In *Hybrid Systems : Computation and Control HSCC'00*, pages 421–436, Pittsburgh, PA, mars 2000.

[SL03] Djamal-Eddine Saidouni and Ouassila Labbani. Maximality-based symbolic model checking. In *ACS/IEEE International Conference on Computer Systems and Applications*, pages 98–107, juillet 2003.

[Sma98] Irina Madalina Smarandache. *Transformations affines d'hologes : application au codesign de systèmes temps-réel en utilisant les langeges* SIGNAL *et* ALPHA. Thèse de doctorat, Université de Rennes 1, octobre 1998.

[SMR05] Tim Schattkowsky, Wolfgang Mueller, and Achim Rettberg. *UML for SoC Design*, chapter A Generic Model Execution Platform for the Design of Hardware and Software, pages 63–88. Springer, 2005.

[Sou01] Julien Soula. *Principe de compilation d'un langage de traitement de signal*. PhD thesis, Laboratoire d'Informatique Fondamentale de Lille, Université des Sciences et Technologies de Lille, décembre 2001.

[SR98] Bran Selic and Jim Rumbaugh. Using UML for modeling complex real time systems. Technical report, Object Time Limited and Rational Software Corporation, mars 1998. `http://www-128.ibm.com/developerworks/rational/library/139.html`.

[SWT+01] Markus Schu, Dirk Wendel, Christian Tuschen, Marko Hahn, and Ulrich Langenkamp. System-on-silicon solution for high quality consumer video processing-the next generation. In *IEEE Transactions on Consumer Electronics CE*, volume 47, pages 412–419, 2001.

[TBC+98] J-P. Talpin, A. Benveniste, B. Caillaud, C. Jard, Z. Bouziane, and H. Canon. BDL, a language of distributed reactive objects. In *ISORC '98 : Proceedings of the The 1st IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 196–205, Washington, DC, USA, 1998. IEEE Computer Society.

[Tea06] SysML Merge Team. SysML specification v. 1.0 (draft). OMG document ad/06-03-01, avril 2006. `http://www.omg.org/cgi-bin/doc?ad/06-03-01`.

[Tec03a] Esterel Technologies. Efficient development of airborn software with SCADE suite^{TM}, 2003.

[Tec03b] Esterel Technologies. SCADE training kit, safe state machine course, 2003.

[Tiw02] A. Tiwari. Formal semantics and analysis methods for Simulink Stateflow models. Rapport technique, SRI International, 2002. `http://www.csl.sri.com/users/tiwari/stateflow.html`.

[TNP+02] Roman Trobec, Roman Novak, Sreco Plevel, Ivan Zupan, and Mira Zupancic. Single-board wavelet video compression/decompression unit. In *Video/Image Processing and Multimedia Communications 4th EURASIP-IEEE Region 8 International Symposium on VIPromCom*, pages 99–104, Zadar, Croatia, juin 2002. IEEE Computer Society. ISBN : 953-7044-01-7.

[TNTBS00] Stéphane Tudoret, Simin Nadjm-Tehrani, Albert Beneviste, and Jan-Erik Strömberg. Co-simulation of hybrid systems : Signal-simulink. In *FTRTFT*, pages 134–151, 2000.

[tTSSot02] ITU-T Telecommunication Standardization Sector of ITU. *Specification and Description Language (SDL)*, août 2002. `http://www.itu.int/ITU-T/studygroups/com17/languages/Z100.pdf`.

[Wan01] Yunming Wang. *UML et technologie synchrone pour les systèmes réactifs distribués*. Thèse de doctorat, Université de Rennes 1, décembre 2001.

[WTBG00] Yunming Wang, Jean-Pierre Talpin, Albert Benveniste, and Paul Le Guernic. A semantics of UML state-machines using synchronous pre-order transition systems. In *International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'2000)*, pages 96–103, Los Alamitos, CA, USA, mars 2000. IEEE Computer Society.

[YGR+06] Huafeng Yu, Abdoulaye Gamatié, Éric Rutten, Pierre Boulet, and Jean-Luc Dekeyser. Vers des transformations d'applications à parallélisme de données en équations synchrones. In *SympA'2006 : SYMPosium en Architecture nouvelles de machines*, Perpignan, France, octobre 2006.