Assistance au raffinement dans la conception des systèmes embarqués

Hocine Mokrani

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

TELECOM ParisTech

2014-ENST-0029

EDITE - ED 130

Doctorat ParisTech

T H È S E

pour obtenir le grade de docteur délivrée par

TÉLÉCOM ParisTech

Spécialité : Télécommunications et Électronique

présentée et soutenue publiquement par

Hocine MOKRANI

2014

Assistance au Raffinement et à la Vérification formels dans la Conception des Systèmes Embarqués

Jury

M. Philippe COUSSY	HDR - Lab-STICC/CNRS	Rapporteur
M. Dominique MÉRY	Professeur - LORIA & Université de Lorraine	Rapporteur
M. Elie NAJM	Professeur - Télécom-ParisTech	Examinateur
M. François VERDIER	Professeur - LEAT/CNRS	Examinateur
Mme. E. ENCRENAZ-TIPHENE	HDR - Laboratoire d'informatique de Paris VI	Directeur
Mme. R. AMEUR-BOULIFA	Maître de conférences - Télécom ParisTech	Encadrant
M. Thierry LECOMTE	Ingénieur R&D - CLEARSY	Invité

THÈSE

TELECOM ParisTech
école de l'Institut Mines-Télécom

REMERCIEMENTS

J'adresse mes sincères remerciements à celles et ceux qui, par leur aide et leur soutien moral, ont contribué à la réalisation de ce travail.

Tout d'abord, mes remerciements vont à Mme. Emmanuelle Encrenaz-Tiphene, et Mme. Rabèa Ameur-Boulifa pour avoir dirigé l'ensemble de mon travail, pour leurs précieux conseils, et les orientations qu'ils m'ont fournies dans la réalisation de cette recherche. Qu'il trouve ici l'expression de mes sincères remerciements et de mon profond respect.

Je souhaite également remercier vivement Mlle. Sophie Coudert, M. Renaud Pacalet et M. ludovic apvrille pour leur disponibilité , leur accompagnement ainsi que l'intérêt apportés aux travaux.

Je remercie chaleureusement les membres de jury de ma thèse, M. Philippe Coussy, M. Dominique Méry, M. Elie Najm, M. François Verdier et M. Thierry Lecomte, pour l'intérêt qu'ils ont porté à mon travail et pour leur disponibilité. Leurs avis éclairés m'ont été précieux.

Une thèse n'en serait pas sans l'encouragement d'amis qui m'ont apporté le soutien scientifique, bibliographique et logistique d'ont j'avais besoin. Gabriel, Jérome, Jair, Anis, Anes, Said, Hakim, Malik et les autres, recevez ici l'expression de ma gratitude.

Enfin, je souhaite exprimer mes plus chaleureux remerciements à ceux qui sont là en toutes circonstances et qui m'ont encouragé depuis toujours : mon père, ma mère, mes frères, mes sœurs, ma fiancée et tous les membres de ma famille.

Table des matières

Introduction

Dans la société moderne, les technologies embarquées sont devenues importantes, voire indispensables. En effet, elles sont de plus en plus implantées dans divers domaines de la vie courante : personnel, transport, assistance et domaine médical. Les progrès technologiques permettent de concevoir des systèmes de plus en plus complexes avec des contraintes fonctionnelles et non fonctionnelles de plus en plus exigeantes.

La complexité de ces systèmes a explosé à cause de l'intégration de grand nombre de composants sur une même surface, dits "systèmes sur puce". L'accroissement de cette complexité conjugué aux exigences de qualité, de performance et économique demandent que soient repensées les méthodes permettant de concevoir et de valider ces systèmes. Les méthodologies et outils de conception doivent permettre de construire des systèmes de fonctionnalité déterminée, de qualité et garantie, à coût acceptable. Pour répondre à cette problématique de nouvelles approches de conception ont vu le jour, ces approches se basent sur :

- L'identification des décisions de conception tôt dans le flot de conception. Les modifications tardives de conception sont coûteuses en temps, ainsi estimer les performances du système tôt permet de réduire les coûts de conception et d'accroître la productivité.

- La modélisation à des niveaux abstraits visant des simulations rapides et des vérifications en amont sur des modèles simples, raffinés ensuite jusqu'au niveau des portes logiques et/ou la génération d'un code exécutable.

- L'exploration d'architecture par une séparation entre la description du fonctionnement d'un système (dite application) et l'architecture matérielle sur laquelle le système va s'exécuter après une étape de liaison (dite de projection). Cette séparation permet la validation du fonctionnement indépendamment de l'architecture et la réutilisation de la même application sur plusieurs architectures.

- L'introduction des méthodes formelles dans le processus de développement

des systèmes sur puce. Le développement de systèmes critiques requiert la vérification exhaustive de leurs fonctionnements. Les techniques de vérification par "model-checking" qui sont basées sur l'exploration d'états du système sont devenues un classique pour des vérifications en aval dans le processus de développement d'un système. L'avantage de ces techniques est la possibilité d'automatisation du processus de preuve et aussi la possibilité de générer un contre exemple dans le cas de violation de la propriété vérifiée.

- La réutilisation du code existant et des modèles développés dans des projets précédents. La réutilisation des composants préalablement conçus dans un autre contexte peut entraîner une réduction considérable dans le temps et le coût de conception. Généralement, les nouveaux produits sont une évolution des produits existants et rarement une révolution.

La vérification formelle est la phase la plus coûteuse dans la conception des systèmes, cette phase se heurte au problème de la taille des systèmes et au temps nécessaire pour sa réalisation. La conception des systèmes par des niveaux d'abstraction permet de gérer la complexité de ces systèmes et donc réduire le temps nécessaire pour la vérification. Toutefois, le processus d'ajout des détails et le passage entre les différents niveaux de description du système est réalisé par le concepteur, ce qui crée une source potentielle d'introduction d'erreurs de fonctionnement et impose la vérification à posteriori des propriétés. Ainsi, la vérification devient rapidement très coûteuse sur les niveaux les plus bas. Par conséquent, la majorité des circuits sur le marché sont validés presque exclusivement par des tests fonctionnels et ne sont pas vérifiés à 100%, ils peuvent donc renfermer des bugs. Dans nos travaux, nous nous sommes intéressés à l'amélioration de la phase de vérification formelle dans les approches de conception par niveaux.

L'objectif de cette thèse est d'améliorer la conception des systèmes embarqués en proposant une approche de conception par niveaux d'abstraction ; cette approche permet de guider et d'assister les concepteurs dans les étapes de conception spécifiquement de raffinement de composants de communication. La méthode de conception proposée doit permettre aisément de raisonner sur les différents niveaux de description du système lors de sa conception par l'exploitation des techniques de preuve de propriétés par raffinement formel. L'originalité de notre travail réside dans l'intégration des résultats de la théorie de raffinement formel dans une démarche bien définie de transformation de modèle et de rajout de détails liés à la conception des systèmes embarqués. Ce document est décomposé en cinq chapitres :

- **Chapitre 1** donne une vue générale du domaine de conception des systèmes embarqués. Il regroupe un état de l'art sur : (1) la conception des systèmes embarqués et la problématique liée à leur vérification, (2) les techniques et outils utilisés pour la validation et de vérification de ces systèmes, et (3) le rapprochement entre le raffinement dans les systèmes embarqués et la théorie du raffinement. À la fin du chapitre, la description de notre contribution est donnée.

- **Chapitre 2** décrit le cadre méthodologique dans lequel s'inscrit notre approche de construction et de vérification par raffinement. Ce chapitre donne le langage utilisé pour décrire les applications, les caractéristiques de l'architecture, et les supports de communication cibles ainsi que les règles de projection choisies. De plus, ce chapitre décrit les différents niveaux et étapes de raffinement choisis ainsi que les différentes transformations des supports de communication proposées.

- **Chapitre 3** présente les bases théoriques sur lesquelles nous nous appuyons pour la description et la réalisation des étapes de raffinement, et la vérification de la cohérence du comportement entre les différents niveaux de raffinement.

- **Chapitre 4** décrit en détail les règles de transformations proposées pour guider le processus de raffinement de communication dans le flot de conception. Ce chapitre donne également la caractérisation de ces règles de transformation dans un formalisme qui permet de raisonner sur la préservation de comportement.

- **Chapitre 5** illustre la démarche de raffinement proposée par une étude de cas. La validité et l'utilisabilité de cette approche sont montrées sur un exemple de modèle d'appareil photo numérique. Nous montrons comment les modèles peuvent être construits et l'ensemble des propriétés préservées par les transformations proposées.

Première partie

Contexte et État de l'art

Chapitre 1

Problématique et état de l'art

Sommaire

Le travail effectué dans cette thèse s'inscrit dans le cadre de conception sûre des systèmes sur puce. Nous nous sommes intéressés plus particulièrement à l'assistance et la préservation de fonctionnement lors des étapes de modélisation tout au long du processus de conception de ces systèmes. Dans ce chapitre, nous présentons les approches de conception des systèmes embarqués, leurs techniques de validation ainsi que notre proposition pour l'amélioration de ces approches. La section 1.1 présente les différents concepts de système embarqués, les approches de conception de ces systèmes et la problématique de leur vérification. La section 1.2 présente les techniques d'analyse utilisées pour la validation et la vérification formelle, leurs limites et les approches de raffinement visant l'amélioration de la complexité et la vérification de ces systèmes. La section 1.3 présente notre proposition dans ce domaine en discutant les points d'amélioration que nous proposons et les différents choix pris en compte.

1.1 Conception des systèmes embarqués

Informellement, on peut définir les systèmes embarqués comme étant des composants électroniques issus d'une combinaison de composants logiciels et composants matériels qui interagissent continuellement avec leur environnement. Les composants logiciels sont utilisés pour leur flexibilité tandis que les composants matériels sont utilisés pour l'augmentation des performances et la réduction de la consommation d'énergie.

Ces systèmes jouent un rôle important dans de nombreux domaines d'application de plus en plus critiques tels que le domaine de la santé, la conduite assistée, la communication et d'autres domaines dans lesquels leur dysfonctionnement peut générer des pertes économiques ou des conséquences inacceptables pouvant aller jusqu'à des pertes en vies humaines. La conception de tels systèmes doit offrir des garanties de bon fonctionnement et de robustesse en cas de défaillance d'une partie de celui-ci ou lors de la survenue d'un événement non prévu.

De nos jours, de nouvelles approches et environnements de conception se sont développés pour répondre à la complexité croissante des systèmes embarqués, notamment les systèmes sur puce. Pour gérer cette complexité, ces approches se basent sur la description du système à différents niveaux d'abstraction. Ces approches doivent alors : (1) décrire une démarche claire allant du modèle abstrait jusqu'à la génération du code ; (2) faciliter au concepteur la modélisation, l'analyse, la vérification, l'exploration, le raffinement, et la synthèse et (3) permettre la réutilisation et la modification des composants lors de processus de conception.

Dans cette section, nous allons tout d'abord montrer les différents niveaux et vues d'un système, puis définir un flot de conception d'un système embarqué ; nous nous basons sur les flots de conception qui commence par le niveau système. Ensuite, nous allons voir les différents environnements de conception, leurs approches et à quoi ils répondent. Enfin, nous dégageons la problématique de cette thèse.

1.1.1 Niveaux et vues d'un système

Il existe plusieurs niveaux de description d'un système embarqué. Les deux niveaux les plus utilisés sont le niveau RTL (Register Transfer Level) et le niveau système. Dans le niveau ***niveau*** RTL le système est représenté en terme de transfert de données entre les registres, des signaux de contrôles, et d'opérations logiques sur les signaux de contrôle et de données dans des composants de calcul. Dans ce niveau, l'échantillonnage des valeurs des signaux et les transferts sont cadencés par une horloge. C'est à ce niveau que sont construits les composants matériels tels que, les bus, les propriétés intellectuelles (IPs), les interfaces et d'autres composants. Le niveau RTL peut être utilisé pour une description exclusivement matérielle d'un système, mais il ne permet pas la description du fonctionnement implémenté en logiciel. Aussi, ce niveau est très détaillé ce qui rend l'analyse et la simulation très lentes. Pour répondre à ce problème de complexité et d'hétérogénéité, le système est représenté dans un niveau plus abstrait dit ***niveau système***. Dans ce niveau, une représentation d'un système décrit le parallélisme entre les différents processus s'exécutant en matériel et en logiciel ainsi qu'une échelle de temps macroscopique indiquant les temps de calcul et de transfert de données.

En addition aux différents niveaux de description d'un système embarqué, celui-ci peut être modélisé selon trois vues quel que soit le niveau de description choisi. Ces vues correspondent à trois caractéristiques différentes et complémentaires du même système [48, 49] : fonctionnelle, structurelle et physique. La ***vue fonctionnelle*** représente le comportement d'un système avec les échanges des données d'entrées/sorties dans le temps sans donner des détails sur la structure et la géométrie de celui-ci. Dans cette vue, le parallélisme d'un système n'implique pas forcément une concurrence lors de son implémentation sur une plate-forme réelle. Le fonctionnement d'un système est décrit dans un langage spécifique selon le besoin de spécification et le niveau d'abstraction choisi. Par exemple, dans le niveau système le fonctionnement est décrit par des langages de flot de données comme les réseaux de KAHN [71] ou un langage de programmation comme le langage C ou SYSTEMC[89, 9] et SPECC[47] ; Par contre, dans le niveau RTL le fonctionnement d'un système est écrit dans les langages de description matériel HDL comme VHDL et VERILOG. La ***vue structurelle*** d'un système représente la structure des composants et leurs interconnexions en faisant abstraction des détails de calcul et de communication. Dans le niveau système, la structure est décrite par une interconnexion de composants de calcul, de communication, de stockage et d'interfaçage ; Dans le niveau RTL la structure est composée par des blocs de registres, des unités de calcul (UAL) et des bus. Dans la ***vue physique*** d'un système, on rajoute l'information de dimension à la structure du système ce qui permet de donner les informations concernant la taille de chaque composant, sa position et aussi la position de chaque porte dans le système. Cette vue est représentée par des paramètres physiques de la structure des composants. Ces paramètres du système sont liés aux technologies de construction des transistors et sont peu (ou pas) exploitables sur le niveau système.

Il existe plusieurs approches de conception, dites aussi flots de conception, de systèmes embarqués, qui sont basés sur différents niveaux et vues d'un système. Ces approches peuvent être classées selon : les niveaux et les vues utilisés, les relations entre les différents niveaux d'abstraction et entre les vues, le type de plate-forme visée (fixe ou dynamique) et aussi le type de langage de modélisation utilisé. Un des flots pour la conception des systèmes consiste à développer et à étudier chaque composant indépendamment des autres dans un niveau détaillé avant de les assembler pour la construction d'un système global. Cette approche ne permet pas d'avoir une vue globale sur le système avant la construction de ces composants, ce qui cause la détection tardive des erreurs à la construction du système. Pour répondre à ce problème, les environnements actuels de développement des systèmes embarqués, se basent sur le ***flot de conception système*** [48]. Ce flot de conception permet la description d'un système dans sa globalité dans le niveau système et permet ainsi le développement conjoint des composants logiciels/matériels. Il permet la modélisation d'un système à différents niveaux d'abstraction allant du niveau système vers le niveau RTL. Dans la suite de ce chapitre, nous nous intéressons à ce flot de conception système, nous parcourons les différents travaux de recherche visant l'amélioration de ce flot, les différents problèmes rencontrés et nous finissons par la proposition d'une piste de solutions possibles.

1.1.2 Flot de conception système

Il existe plusieurs flots de conception de systèmes qui sont basés sur différentes phases avec différents degrés d'interactivité, d'automatisation et de complexité. Ces flots de conception comportent globalement quatre phases essentielles : phase de modélisation,

phase d'exploration, phase de raffinement, et phase de synthèse. La figure FIG. 1.1 montre un schéma général d'utilisation des différentes phases commun aux différents outils de développement. Ce flot de conception commence par une phase de modélisation de comportement de système, cette phase est suivie par une phase de structuration des différentes fonctionnalités du système, dite phase d'exploration d'architecture qui a pour but la construction de la plate-forme cible. Le choix de la plate-forme est réalisé après un processus d'analyse de performance et de bon fonctionnement. Cette phase peut être aussi réalisée par des étapes de rajout de détails, c'est la phase de raffinement. Enfin, après la spécification de tous les détails architecturaux et de fonctionnement, une phase de synthèse est réalisée dans le but de faire passer le modèle du système du niveau système vers les niveaux détaillés.

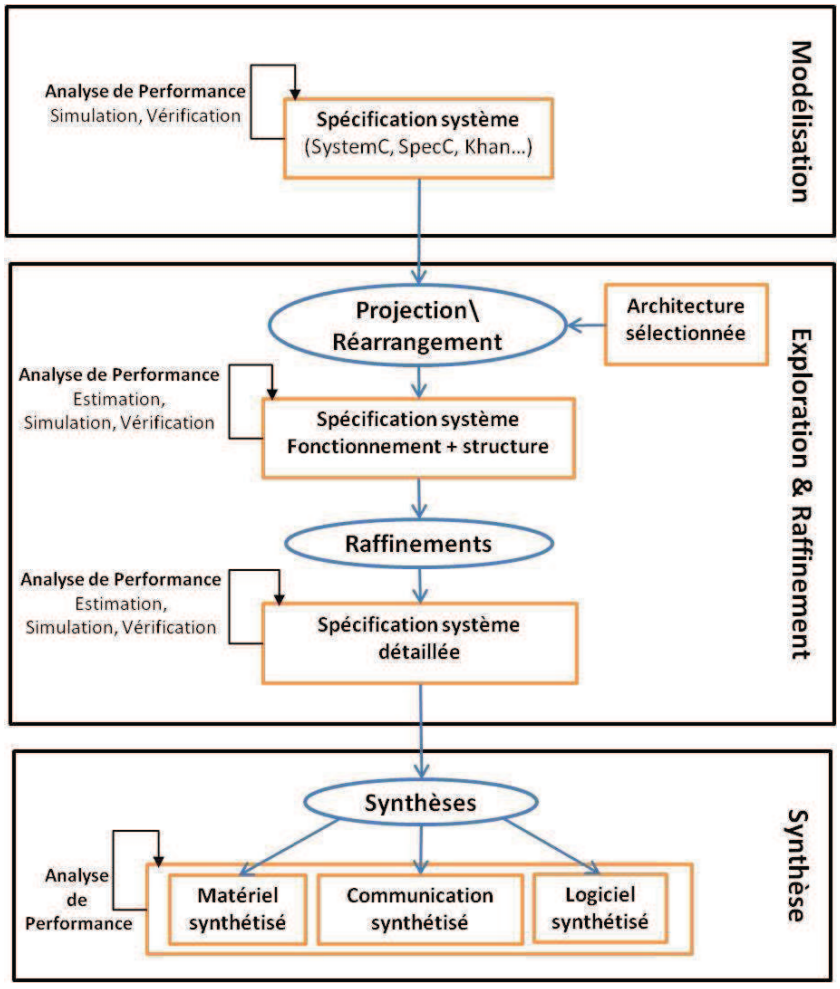


FIG. 1.1 – Schéma général de flot de conception système

Modélisation

La modélisation de système est donc le point de départ du flot de conception. Le choix d'un langage approprié pour la description des systèmes embarqués est important. Idéalement, le langage devrait permettre de décrire le comportement concurrent d'un système, d'exprimer le temps et d'offrir un moyen de décrire différents niveaux d'abstraction. En outre, le langage doit avoir une sémantique bien définie pour l'analyse et la vérification.

De nombreux langages et paradigmes de modélisation sont utilisés pour spécifier les fonctionnalités et la structure d'un système embarqué. On peut citer les langages de conception basés sur C/C++ qui sont familiers à de nombreux concepteurs et ingénieurs, les deux principales initiatives sont SystemC [89, 9] qui est basée sur C++, et SpecC[47] qui est basée sur C. Dans les deux cas, le langage est enrichi avec des constructeurs de concurrence et de temps pour une meilleure modélisation des systèmes embarqués. Les deux langages partagent le même esprit de spécification, de description d'un système dans des niveaux d'abstraction et de processus de validation par simulation. Par contre, les deux langages se basent sur différentes idées et paradigmes d'implémentation [28], comme l'utilisation des événements, la permission de la description hiérarchique des processus. D'autres langages de description sont développés, ils répondent à différents buts, tels que la description détaillée de la structure et le comportement des différents composants d'une architecture. Les deux langages les plus connus pour la description de l'architecture sont SDL[66] et AADL [44]. Certaines approches formelles sont basées sur les langages synchrones tels que ESTEREL [109] et LUSTRE/SCADE [58] qui sont utilisés pour leurs aspects de synchronisme, ce qui les rend particulièrement intéressants pour la conception matérielle des systèmes.

Malgré la panoplie de langages existants, aucun langage parmi ceux cités n'offre l'ensemble des critères attendus. Certains offrent une possibilité de description abstraite de fonctionnement d'un système et permettent la représentation des applications basées sur le flot de données comme les processus de Kahn, d'autres sont utilisés pour la description d'architecture comme AADL, SDL, et d'autres encore sont utilisés pour leur aspect formel ou leur possibilité de décrire les aspects de synchronisation comme ESTEREL ou encore pour leurs environnements de simulation comme SystemC.

Exploration

L'exploration d'architecture est une phase importante dans le flot de conception des systèmes embarqués. Elle a pour but de trouver la structure la plus adaptée aux besoins du comportement à exécuter, incluant : le découpage fonctionnel d'un système avec les performances optimales et le bon fonctionnement, l'ordonnancement des processus, la détermination des protocoles de communication et la topologie du réseau reliant les processeurs et autres constituants (mémoires, DMA, ...). Dans cette phase, le concepteur choisit parmi les différentes structures possibles celle qui répond le mieux aux objectifs et aux exigences fixées. La sélection est réalisée en utilisant diverses techniques d'évaluation, d'estimation et d'analyse de performance. L'espace des solutions possibles peut s'avérer très grand, c'est-à-dire, de nombreuses architectures peuvent être candidates. En outre, chaque structure peut être paramétrée par différentes configurations des composants, ce qui rend la phase d'exploration d'architecture fastidieuse.

Pour rendre cette tâche d'exploration d'architecture plus efficace et plus rapide lors du changement de partitionnement, plusieurs outils et approches de conception ont adopté le concept de la séparation explicite entre le modèle du comportement appelé "application" et le modèle de la structure appelé "architecture", cette séparation permet la conception du modèle du système par l'association des deux modèles dans une opération bien définie dite de projection. POLIS [12] est la première incarnation de cette approche de séparation. Elle a été adoptée ensuite par plusieurs environnements de développement que nous détaillerons dans la section suivante.

L'exploration d'architecture peut être réalisée manuellement ou automatiquement. Dans les approches manuelles le partitionnement de comportement du système sur l'architecture est donné par le concepteur, l'objectif principal de cette approche est l'estimation efficace de partitionnement sélectionné [50]. Les approches automatiques visent la recherche du partitionnement optimal. Généralement ces approches automatiques sont basées sur différents paramètres pour l'optimisation de l'énergie, le coût de fabrication tels que : la taille des caches, le partitionnement des données et le type des supports de communication utilisés ainsi que leur structure [73, 106].

Raffinement

Lorsque le concepteur fixe un partitionnement, il peut transformer et améliorer les performances de son modèle en ajoutant graduellement des détails fonctionnels et architecturaux : c'est le processus de raffinement. Cette phase itérative se termine par la concrétisation du modèle abstrait, c'est-à-dire, la construction de modèles concrets des composants logiciels (en code exécutable) et matériels (en HDL synthétisable) en passant par des niveaux de description du système jusqu'au niveau détaillé. Le raffinement est un processus qui nécessite des informations données par le concepteur à chaque étape de transformation. Le raffinement des systèmes embarqués est généralement décomposé en quatre types [1] : raffinement par décomposition de fonctionnement, raffinement de communication, raffinement de données, et raffinement de calcul.

- ***Le raffinement par décomposition de fonctionnement*** consiste principalement à transformer une spécification séquentielle d'un système en une spécification parallélisée.

- ***Le raffinement de données*** consiste à définir la structure, le type et le domaine des données abstraites par des structures, des types et des domaines implémentables sur les composants d'architecture logiciels /matériels.

- ***Le raffinement de communication*** consiste à transformer (ou à remplacer) des échanges de données et des supports de communication abstraits par des transferts de données et supports de communications concrèts en utilisant les protocoles de communication plus élaborés, des bus et des réseaux sur puce.

- ***Le raffinement de calcul*** consiste à concrétiser des calculs par de nouvelles opérations, nouvelle granularité des données et par découpage des processus complexes en une séquence de processus moins complexes.

Parmi les travaux qui concrétisent les modèles par transformation de raffinement nous citons : les travaux de P. Lieverse et al. dans [74] qui proposent une technique basée sur la transformation de traces pour le raffinement de communication dans la phase d'exploration d'architecture. Cette technique supporte l'implémentation des primitives de communication du niveau abstrait (application) décrit dans un réseau de KAHN par des primitives de communication sur un niveau architectural en introduisant des synchronisations, l'ensemble des traces générées dans ces travaux permettent l'exploration d'espaces de solutions pour l'implémentation optimale d'une application sur une architecture choisie en fonction du temps. J.Brunel et al [27] dans le cadre du projet COSY pour le traitement des vidéos proposent une méthode de conception et d'interfaçage entre le niveau applicatif et le niveau architectural. À partir d'une application donnée, ils décrivent deux niveaux de description d'interface pour les IPs en se basant sur le protocole VCI. Aussi dans cette démarche, le choix de l'architecture optimale est réalisé selon une fonction d'estimation de délais sur les communications. Dans [77] R. Marculescu et al. présentent une approche centrée sur la communication décrite aux différents niveaux d'abstraction où l'application et la plate-forme sont utilisées pour optimiser le système développé à partir d'une analyse des performances par des automates stochastiques. Contrairement aux deux premiers travaux, ce travail n'utilise pas des protocoles de communication prédéfinis, et le processus de raffinement est réalisé totalement par le concepteur. L'application des différents types de raffinements est liée fortement aux langages de spécification utilisés, à l'implémentation ciblée ainsi qu'aux spécifications d'architecture choisies. Lorsque le niveau de description du système est assez détaillé et toutes les contraintes architecturales spécifiées, le processus de raffinement peut être automatisé, on parle dans ce cas de la phase de ***synthèse***.

Synthèse

La synthèse d'un système est un processus de transformation automatisé des spécifications écrites dans des langages de haut niveau sans aucune information sur le temps ou partiellement pris en compte (par exemple, C, C++, SYSTEMC, VHDL) en des spécifications de bas niveau décrites au cycle précis. La synthèse du système est généralement décomposée en trois parties [37] : la synthèse des composants matériels, la synthèse des composants logiciels et la synthèse des composants de communication.

- ***La synthèse des composants matériels*** consiste à générer une description matérielle RTL du système à partir d'une spécification de haut niveau (par exemple SYSTEMC). La description RTL est ensuite synthétisée en des netlists d'interconnexion de transistors.

- ***La synthèse des composants logiciels*** consiste à générer une description exécutable par un processeur cible à partir d'une description haut niveau du système en prenant en compte des contraintes liées à la gestion des ressources ainsi qu'à la politique d'ordonnancement des tâches liée au système d'exploitation choisi.

- ***La synthèse des composants de communication*** consiste à concevoir et à générer des moyens d'interconnexion entre les composants du système. Une fois que les processus sont affectées aux ressources du système, il faut adapter le transfert entre ces processus aux ressources physiques disponibles pour la communication (bus, communication point-à-point, mémoire,. . .etc).

Parmi les travaux de synthèse des systèmes nous citons : les travaux de Hommais et al. dans [61] proposent une approche pour la synthèse des communications des systèmes matériels et logiciels, à partir d'un modèle abstrait de communication décrit par des réseaux de KHAN jusqu'au modèle physique de communication en se basant sur le protocole VCI. Le modèle de communication est bloquant en lecture et en écriture. L'approche proposée se base sur l'implémentation des communications dans des threads POSIX pour la partie logicielle et la proposition d'un module pour la communication des composants matériels. Dans cette démarche, l'analyse du système est réalisée par simulation pour étudier les performances du système. Gupta et al. dans [56] abordent le problème de la synthèse des composants logiciels et matériels en se basant sur des graphes de dépendance de données et de partage des espaces de stockages pour la construction du code C à partir de description HARDWAREC [70], qui est un langage proche du langage C permettant la description du temps et des contraintes de ressources. Les modèles obtenus sont analysés par la simulation par événements, une sélection automatique des interfaces de communication peut être réalisée selon les contraintes de débit de transfert imposées lors de la phase de modélisation. Coussy et al. [38] proposent une approche de synthèse des composants IP décrits par des langages de description de haut niveau (VHDL, SYSTEMC ou SPECC) selon des contraintes de temps d'ordre d'échange de données. L'approche repose sur la modélisation formelle des contraintes d'intégration IP par un graphe décrivant les entrées et sorties en terme de temps, de la structure de données, des modes de transfert et du protocole utilisé. Cette approche permet l'analyse de la faisabilité et la cohérence de l'algorithme à implémenter vis-à-vis ces contraintes. D'autres démarches de synthèse automatiques basées sur des concepts formels existent, Passerone et al.[90] proposent une méthode formelle pour la spécification d'interface entre deux protocoles non-compatibles. Cette méthode repose sur la théorie des jeux pour analyser les situations dans lesquelles les deux protocoles peuvent communiquer. Les deux protocoles sont compatibles dans le cas d'existence d'une solution de communication qui caractérise le comportement d'interface entre les deux protocoles.

Le raffinement et la synthèse sont deux processus qui prennent en compte les contraintes liées à l'architecture cible lors de concrétisation. Contrairement à la synthèse qui est une concrétisation d'une description déjà assez détaillée et automatisée, le raffinement est un processus qui peut commencer par un niveau très abstrait de spécification et permettant le rajout de détails de spécification par étapes dans le but de mieux gérer la complexité du système et aider le concepteur dans le processus de conception. Ce processus permet le changement de granularité de données et la concrétisation des descriptions abstraites de traitement et de transfert.

1.1.3 Environnements de conception

Il existe différents environnements et outils qui intègrent une partie ou l'ensemble des phases du flot de conception dans le niveau système. Ces derniers, issus de travaux académiques ou développés dans l'industrie, offrent différents moyens pour la conception des systèmes embarqués. Densmore et al. [41] ont recensé et comparé une centaine d'environnements et outils différents qui suivent le flot de conception dans le niveau système. Ces outils partagent plusieurs points communs tels que la séparation de spécification fonctionnelle et architecturale, les langages de spécification utilisés, l'intérêt à l'aspect de vérification formelle et la synthèse de haut niveau. Mais, cette comparaison n'a pas discuté la prise

en compte de la phase du raffinement dans ces approches. Dans cette section, nous allons décrire brièvement quelques environnements connus et les positionner par rapport à leur capacité à intégrer des raffinements, ainsi que leur possibilité d'analyse par simulation et/ou vérification formelle.

Environnements de simulation

Cofluent [64] est un outil de conception des systèmes embarqués à un niveau système. Il fournit un format de description UML, la séparation de la vue fonctionnelle et architecturale et permet la génération automatique du code SystemC, ce qui offre la possibilité de valider le système par simulation et par prototypage virtuel. Cet outil est basé sur la conception MCSE[29] (Méthodologie de Conception des Systèmes Electroniques), cette méthodologie couvre le flot de conception de la collection de l'information et la description des exigences jusqu'à l'analyse et le prototypage du système. Par contre, cet outil n'offre pas des règles de transformation explicite ni d'environnement pour la vérification formelle.

SoCLib Tools [53] SoCLib est une plate-forme virtuelle de prototypage des systèmes multi-processeurs. Le cœur de la plate-forme est une bibliothèque de modèles SystemC des composants. Plusieurs outils viennent avec la bibliothèque SoCLib pour faciliter le développement tels que SystemCASS pour la simulation rapide, DSX pour l'exploration d'architecture et la projection basées sur la séparation des vues fonctionnelles et architecturales. Dans cet outil, une application est décrite sous forme de graphe des tâches communicantes en utilisant le langage graphique TCG (pour Task and Communication Graph). L'architecture est décrite aussi par des composants importés de la bibliothèque SoCLib, et GAUT [39] pour la synthèse qui permet la génération des descriptions RTL à partir des descriptions détaillées données en C/C++. Par contre, cet outil commence par un niveau de description détaillé et n'intègre pas dans le flot de conception des outils de raffinement guidé ni de vérification formelle.

Sesame [55] est un outil de modélisation et de simulation permettant l'exploration de l'espace de conception et la reconfiguration des systèmes. Grâce à une modélisation de l'architecture cible à différents niveaux d'abstraction, par des réseaux de Kahn, il est possible d'évaluer un grand nombre de solutions en utilisant une modélisation au niveau le plus haut puis de raffiner en utilisant des modèles d'architecture plus précis. Le changement de niveau est réalisé par une technique basée sur la transformation de traces pour le raffinement de communication [74]. En revanche, cet environnement n'offre aucun outil pour la vérification formelle.

Cierto VCC [101] est un outil industriel de conception conjoint du matériel et du logiciel qui propose un flot de conception allant de la modélisation de spécification fonctionnelle jusqu'à la spécification de plate-forme au niveau système au cycle près. Cet outil permet la séparation entre le fonctionnement et l'architecture et offre la possibilité de simulation et d'estimation d'architecture. Il propose aussi la possibilité de raffinement graduel des communications, mais il n'offre pas une démarche de raffinement guidée. Et aussi, il ne permet pas la vérification formelle.

Environnements offrant la vérification formelle

Ptolemy [72] développé par l'université de Berkeley, est un environnement de simulation et de prototypage des systèmes concurrents, temps réels, embarqués et hétérogènes. Il offre un environnement pour la modélisation hiérarchique par des machines à états connectées et la simulation des applications et des architectures décrites à plusieurs niveaux d'abstraction. Il permet aussi la vérification mathématique des systèmes par les techniques de model-checking [11]. Le processus de raffinement est laissé à la charge du concepteur, il est réalisé par l'expansion des états des modèles par leur comportement détaillé.

Polis [12] est un environnement qui permet la description des systèmes par des réseaux de machines à états dans un haut niveau d'abstraction. Chaque élément de ce réseau peut être projeté sur un nœud matériel ou logiciel. L'approche permet l'ajout des détails au système abstrait, la synthèse des composants logiciels/matériels et offre des environnements pour la simulation et la vérification formelle. Par contre, cette approche n'offre pas des garanties sur les transformations réalisées lors d'ajout des détails. L'approche METROPOLIS [13] issue de POLIS vise des systèmes fortement hétérogènes, en offrant de nombreux modèles et niveaux d'exécution. Cette approche permet la description de l'application, de l'architecture et de la projection dans un même modèle.

SCADE [57] offre un environnement intégré de conception et de développement pour des applications logicielles critiques embarquées. Il offre un cadre couvrant le prototypage, la conception graphique, la simulation, la vérification et la validation, ainsi que la génération de code. SCADE est basé sur le langage LUSTRE qui est un langage de programmation synchrone par flots de données possédant une sémantique formelle. Grâce à son compilateur certifié [76, 59] écrit dans le langage LUSTRE [80], l'environnement SCADE permet l'analyse formelle et la génération de code fiable en langage C et en ADA.

Gaspard2 [40] est un environnement basé sur un profil UML-MARTE [46, 4]. Ce profil développé à l'INRIA propose une notation pour la modélisation des systèmes sur puce temps réel avec la possibilité de représenter de nombreuses notions comme le temps et les ressources. **Gaspard2** propose des règles de transformation de modèles dans le but de l'exploration et de l'analyse d'un système. La transition entre ces différents modèles s'effectue suivant un processus de raffinement aboutissant à la génération des modèles précis au cycle près dans différents langages cibles (SYSTEMC, VHDL...) et même vers des langages synchrones pour la vérification formelle.

Le tableau TAB.1.1 résume les caractéristiques des différents environnements : la modélisation par la séparation des vues fonctionnelle et architecturale (structurelle), l'analyse par simulation et/ou vérification formelle, la conception par raffinement guidé et la synthèse et/ou génération du code. La figure FIG. 1.2 montre les possibilités de ces environnements pour les aspects de simulation, de vérification formelle, de raffinement et de synthèse dans les différents niveaux de conception. En effet, nous avons subdivisé le niveau système en trois différents niveaux de spécification : la spécification fonctionnelle qui décrit la vue fonctionnelle du système, la spécification du système avec les deux vues fonctionnelle et structurelle, et ces mêmes vues avec la représentation du temps. La figure montre que de nombreaux environnements offrent la simulation des spécifications purement fonctionnelle

	Séparation de vues	Analyse par simulation	Analyse par vérification formelle	Raffinement	Synthèse/ Génération du code
Cofluent	Oui	Oui	Non	Non	Oui
SoCLib DXS	Oui	Oui	Non	Non	Oui
Sesame	Oui	Oui	Non	Oui	Non
Cierto VCC	Oui	Oui	Non	Non	Non
Ptolemy	Oui	Oui	Oui	Non	Oui
Polis	Oui	Oui	Oui	Non	Oui
SCADE	Non	Oui	Oui	Non	Oui
Gaspard2	Oui	Oui	Oui	Oui	Oui

Tab. 1.1 – Comparaison entre quelques environnements de conception.

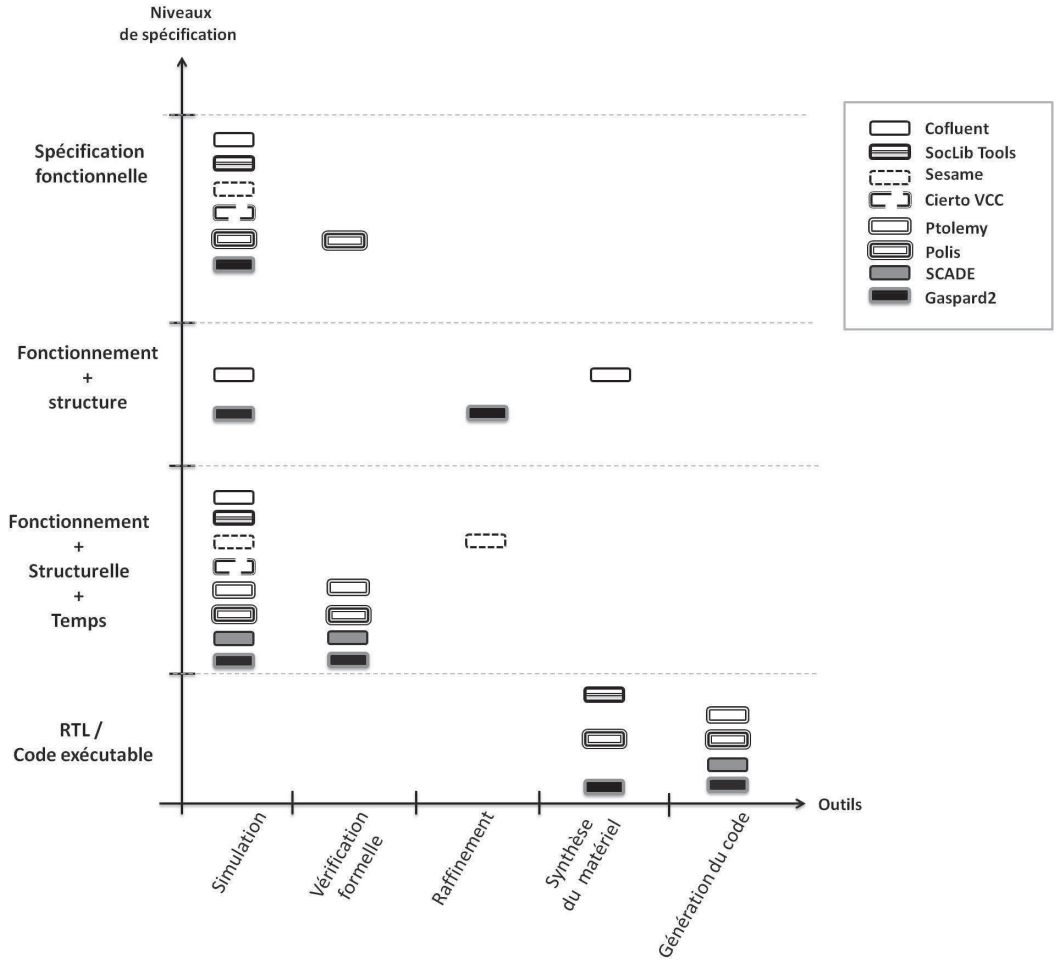


Fig. 1.2 – Zone d'activité des différents environnements

et à la spécification structurelle temporisée. La focalisation sur les aspects temporels est due aux besoins liés à l'estimation des performances des architectures lors de la phase d'exploration et aux informations nécessaires pour la synthèse du logiciel ou du matériel. Certains de ces environnements proposent la génération du code exécutable, comme **Ptolemy** et **SCADE** qui génèrent le code C qui tournera sur le processeur dédié, ou la synthèse du niveau RTL à partir du niveau détaillé de la spécification structurelle temporisée, comme l'environnement **Polis** et l'outil **GAUT** de la suite **SoCLib Tools**. Les possibilités qu'offrent ces environnements pour la vérification formelle et pour le raffinement sont beaucoup moins importantes qu'à la simulation et à la synthèse. Le problème le plus important que rencontre la vérification formelle dans ces environnements est la taille du système à vérifier qui résulte d'un côté à la complexité des systèmes, et d'un autre côté au nombre d'informations nécessaires sur ces spécifications pour l'exploration d'architecture d'une manière proche de la réalité. **Polis**, **Ptolemy** et **SCADE** sont des environnements basés sur des langages formels ce qui leur offre une sémantique claire dans les différents niveaux de spécification et la possibilité de vérification formelle. Contrairement à ces environnements, **Gaspard2** n'offre la vérification formelle qu'après la génération de spécification synchrone à partir d'une spécification structurelle temporisée au niveau système. Concernant le raffinement, la majorité de ces environnements n'offrent pas de procédure de transformation et cette tâche est laissée au concepteur, ce qui crée une source potentielle d'introduction d'erreurs de fonctionnement lors du processus de raffinement de spécification.

1.1.4 Problématique

Comme nous venons de le voir, de nombreux environnements implantant le flot de conception au niveau système existent. Ces environnements intègrent bien la conception des systèmes par niveaux et offre une large possibilité de modélisation, d'exploration, de synthèse et d'utilisation des méthodes de validation des systèmes par simulation et par la vérification de propriétés d'une manière formelle.

Malheureusement, avec toutes les approches de conception des systèmes embarqués qui mettent en œuvre des techniques et outils complexes pour l'exploration d'architecture, la notion de raffinement guidé de comportement avec garantie formelle (preuve mathématique) reste peu accessible. Le processus de raffinement des concepts reste à la charge du concepteur, qui doit alors tirer profit de son expertise. Or, cette tâche qui s'avère cruciale porte un grand potentiel d'introduction de nouveaux comportements non-souhaitables et des erreurs dans la spécification de système.

Les travaux que nous présentons s'inscrivent dans le cadre d'assistance au raffinement et à la vérification formels dans la conception des systèmes embarqués. Nous proposons une nouvelle approche dans le but d'aider le concepteur dans le raffinement du système lors du processus d'exploration d'architecture [85, 83]. À cet effet, nous commençons par un niveau très abstrait de spécification fonctionnelle du système pour gérer au mieux la complexité du système et profiter au maximum du processus de raffinement. Par rapport aux différents environnements de conception existants, la spécification de départ que nous avons utilisée permet l'abstraction des valeurs de données échangées entre les différents composants, on ne garde que l'information de quantité des données, ce qui permet alors l'abstraction des détails de calcul et ainsi la réduction de la complexité du système. Dans cette thèse, nous proposons des étapes de raffinement et des règles de transformations

formelles permettant la vérification de préservation des propriétés de comportement par raffinement formel en arrivant à un niveau de spécification de la structure sans prise en compte de l'aspect temps ni les valeurs des données échangées. Dans la section suivante, nous allons introduire les différentes méthodes de vérification des systèmes embarqués et leurs limites avant d'introduire la modélisation et la vérification par raffinement formel.

1.2 Vérification des systèmes embarqués

La vérification est une des composantes clefs de conception des systèmes. L'objectif de la vérification est de s'assurer que le système construit se comporte selon une spécification de fonctionnement qui est définie préalablement. La vérification des systèmes prend 40% à 70% de l'effort de développement total d'un système [96]. Il existe de nombreuses méthodes de vérification. On peut classer ces méthodes en deux groupes : les méthodes par simulation et les méthodes par vérification formelle [96].

1.2.1 Validation par simulation

La technique de validation par simulation est la technique la plus répandue. Cette technique permet de vérifier le bon fonctionnement d'un système par la vérification d'un ensemble de traces d'exécution. La simulation est basée sur des stimulis et des moniteurs, les stimulis sont un ensemble des séquences des valeurs sur les variables d'entrée du composant à vérifier, les moniteurs sont des composants supplémentaires permettant de comparer pour une séquence d'entrée donnée, la séquence de sortie produite par rapport à la séquence de sortie attendue. Il existe plusieurs techniques de simulation [96] : la simulation par événement, la simulation par cycle, HW/SW co-simulation, l'émulation et le prototypage rapide des systèmes. Bien qu'applicable tout au long du flot de conception, la simulation reste non exhaustive et ne permet pas de garantir à 100% l'absence des erreurs et des bugs de fonctionnement d'un système. De plus, à cause de la complexité des systèmes la simulation requiert parfois une puissance et un temps de calcul considérables. Ce qui donne des limites à cette technique. Des techniques sont développées pour améliorer le temps de simulation des systèmes par l'amélioration des algorithmes de simulation [97] et l'utilisation des avantages des descriptions de haut niveau [69, 67]. Par contre, pour recouvrir l'ensemble des traces exécutions d'un système lors de la vérification, des techniques de vérification formelle doivent être introduites.

Les techniques de vérification formelles utilisent des formalismes mathématiques pour la preuve du bon fonctionnement du système, ce qui rend leur utilisation peu adoptée par les concepteurs non expérimentés dans ce domaine. La vérification formelle compare deux modèles d'un système pour établir une relation entre eux, ou montre qu'un ensemble de propriétés sont satisfaites sur un modèle. Dans les deux cas, les résultats de la vérification formelle sont valides pour tous les scénarios d'exécution du modèle. Les deux techniques de vérification les plus utilisées sont : "model-checking" et "theorem-proving".

1.2.2 Vérification par "Model-Checking"

Le model-checking est une technique de vérification des systèmes basée sur le modèle de machines à états pour vérifier des propriétés comportementales d'un système. Princi-

palement, le model-checking vérifie qu'un modèle de système satisfait ou non un ensemble de propriétés logiques données [96]. Dans les années 80, les premiers travaux sur le model-checking ont vu le jour [33, 95]. Cette technique consiste à créer un modèle du système sous forme de machine à états finis et à vérifier des propriétés sur celui-ci si cette machine est un modèle pour les propriétés logiques à vérifier. La modélisation d'un système peut être réalisée manuellement ou générée à partir d'une description comportementale de celui-ci. Les propriétés atomiques à vérifier sont exprimées comme des expressions booléennes sur les variables du modèle. Les propriétés du système à vérifier sont exprimées par des formules temporelles. Ces formules sont spécifiées en utilisant les variables des systèmes et des opérateurs de temps discret décrivant le changement des valeurs possibles des variables lors d'exécution de ce système. Comme le montre la figure FIG. 1.3, l'outil de vérification (model-checker) confronte une propriété à vérifier au modèle du système étudié. Il produit un résultat vrai si la propriété est vérifiée sur le système, sinon l'outil fournit un contre-exemple qui est la séquence d'états qui invalide la propriété, ce qui constitue la force de cette technique.

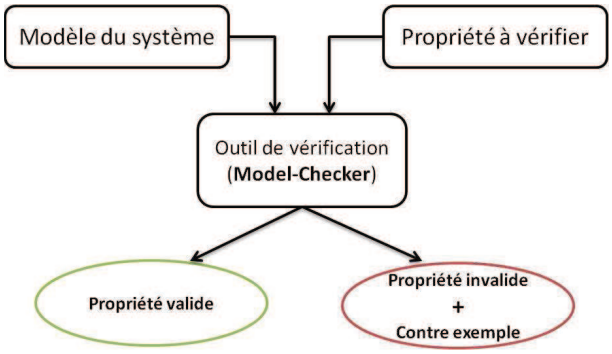


FIG. 1.3 – Processus de preuve par "Model-Checking"

Pour montrer que le système vérifie une propriété, l'outil de model-checking (model-checker) procède par le parcours d'espace d'états du système ; cette technique se heurte rapidement au trop grand nombre d'états du système pour être analysables efficacement. Pour contrer ce problème, de nombreuses techniques d'amélioration du model-checking classique ont vu le jour : les techniques d'abstraction, le model-checking borné, le model-checking dirigé, le model-checking symbolique et l'équivalence checking. Les techniques d'abstraction [34, 25] permettent de proposer un modèle abstrait du modèle initial cachant des détails dans le but de maitriser l'explosion du nombre d'états lors de vérification des propriétés. La vérification sur le système abstrait peut introduire des chemins invalidant la propriété qui n'ont pas de réalité sur le modèle concret ce qui est appelé "fausse alerte". Dans ce cas, il faut raffiner l'abstraction afin de s'assurer de l'exactitude de l'alerte. Le model-checking borné [21] permet de vérifier exhaustivement une partie de l'automate en limitant la longueur des séquences analysées. Le model-checking dirigé permet de sélectionner les chemins à parcourir et d'orienter l'exécution à vérifier. Le model-checking symbolique utilise des structures de données adaptées pour la représentation et la manipulation efficaces d'ensembles d'états pour repousser le problème d'explosion combinatoire [20]. L'équivalence-checking utilise un système de référence et le système à vérifier. Comme les deux systèmes doivent avoir le même comportement, il suffit de vérifier que pour les mêmes séquences d'entrées les deux systèmes produisent les mêmes séquences de sorties.

Plusieurs outils de vérification par model-checking existent, parmi ces outils nous citons. CADP [52] une boîte à outils qui propose plusieurs fonctionnalités : la réduction d'automates, la vérification des propriétés et la vérification d'équivalence de deux spécifications par plusieurs sémantiques d'équivalence telles que la sémantique des traces et les bisimulations faible et forte. Cette boite à outils accepte plusieurs langages d'entrée tels que LOTOS [65] et FC2 [22]. UPPAAL[68] est un autre outil de model-checking. La description des systèmes en UPAAL se base sur les automates temporisés, il utilise un sous-ensemble de la logique temporelle arborescente CTL pour la description des propriétés. SPIN [60] est un outil basé sur le langage PROMELA pour la spécification et permet la vérification des propriétés LTL. VIS [26] est un système de vérification et de synthèse, les spécifications sont écrites dans un langage proche du VERILOG, il permet la vérification des propriétés décrites en CTL et en LTL. RuleBase [15] est un outil industriel développé par IBM offrant la vérification des spécifications par model-checking borné basé sur l'outil SMV [79]. Le mode d'utilisation la plus courante de cet outil et la vérification des spécifications au niveau RTL décrites en VHDL ou Verilog [14]. Cadence SMV est une autre extension de l'outil de model-checking symbolique SMV. Il a un mode plus expressif et accepte en addition le langage Verilog synthétisable. Cet outil adapte une variété de techniques pour la vérification compositionnelle permettant ainsi son utilisation dans des spécifications plus larges. Les outils de model-checking sont utilisés aussi comme des plugins de vérification par des environnements de conception des systèmes embarqués, comme c'est le cas de l'outil TTool [102] qui génère des descriptions LOTOS à partir des descriptions UML-DIPLODOCUS pour une vérification des propriétés temporelles en utilisant la boite à outils CADP. Aussi, l'environnement TOPCASED [93] permet la vérification formelle des descriptions écrites en AADL. Les modèles AADL du système sont transformés en une description formelle intermédiaire écrite en Fiacre[18]. Ce formalisme permet la représentation du comportement et les aspects temporels du système qui peuvent être transformés ensuite en des formalismes LOTOS ou des réseaux de Petri temporisés pour la vérification des propriétés par model-checing.

1.2.3 Vérification par "Theorem-Proving"

Une approche alternative pour la vérification formelle est la technique de vérification par raisonnement déductif [48] ou la preuve des théorèmes. La preuve de théorèmes se base sur un ensemble d'axiomes et des règles de déduction ou d'inférence. La vérification dans ce cas consiste à démontrer une proposition à partir des axiomes par application des règles d'inférence. Dans le cas où la preuve d'une proposition n'est pas trouvée alors on ne peut pas affirmer que la proposition est un théorème. La preuve de théorèmes n'est pas totalement automatique, elle nécessite une intervention humaine experte.

Comme la preuve peut s'avérer complexe, des étapes de décomposition et de simplification des obligations de preuve sont nécessaires. Cette étape peut être réitérée plusieurs fois dans le cas d'axiomes complexes. La figure Fig. 1.4 montre comment fonctionne l'environnement de preuve d'un assistant de preuve. L'assistant de preuve tente la vérification des obligations de preuve automatiquement ou d'une manière interactive en utilisant les hypothèses des bases théoriques de la logique utilisée ou des règles d'inférence.

Par opposition au model-checking, cette technique ne souffre pas des problèmes d'explosion d'états. Mais, un des grands problèmes de cette technique est qu'elle nécessite une

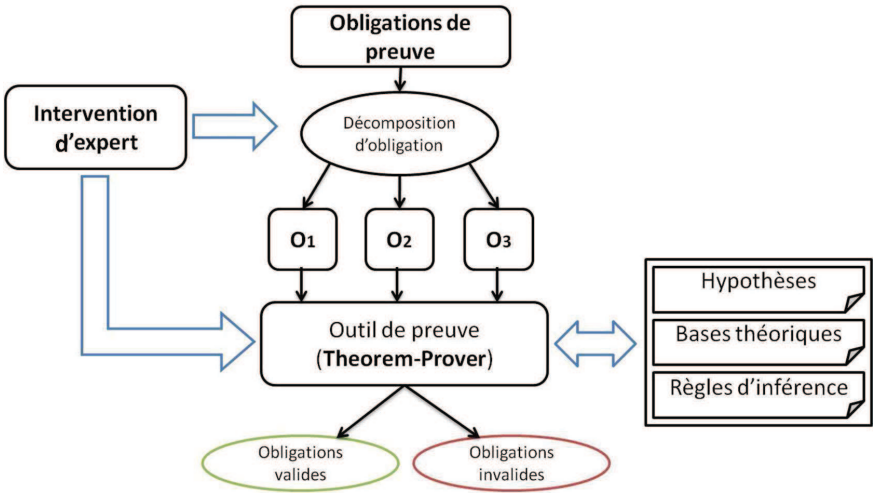


FIG. 1.4 – Processus de preuve dans les outils de “Theorem-Proving”.

expertise et un effort considérable du concepteur autant au niveau de la spécification de chaque composant qu'au moment de guider la preuve ce qui rend cette méthode moins largement utilisée dans l'industrie. Il existe plusieurs outils de preuve, dit assistant de preuve. Ils ont chacun leurs propres caractéristiques. Citons : COQ [19] qui est un système de construction de preuve formelle basée sur le calcul de construction développé par l'*INRIA*. Il fournit une théorie des types d'ordre supérieur avec un environnement pour le développement interactif des preuves pour vérifier des modèles des systèmes. ISABELLE/HOL [87] est un autre outil d'assistance de preuve de la même famille que COQ. Ces outils sont utilisés comme une base pour la génération correcte du code des modèles et la vérification des propriétés de bon fonctionnement et de sécurité. Parmi les travaux de vérification par la preuve de théorème citons : les travaux d'Andronick et al. [5] qui ont proposé une démarche de vérification des propriétés de sécurité du code embarqué dans les cartes embarquées. La démarche permet l'extraction de spécification formelle à partir des annotations décrites sur le code C sur lequel des propriétés de sécurité sont prouvées par l'assistant de preuve COQ. Les travaux de Pell [91] présentent un environnement de génération des composants matériels dans des circuits FPGA. La preuve de correction du modèle généré est réalisée en utilisant l'outil ISABELLE.

À cause des limites des méthodes de vérification formelle et la nécessité d'une expertise mathématique considérable par l'ingénieur ces techniques sont peu adoptées par l'industrie. Le ”model-checking” souffre du problème de l'explosion combinatoire et le ”theorem-proving” est assez compliqué pour les concepteurs de systèmes peu expérimentés. De nouvelles approches visent l'amélioration et la facilité d'utilisation de ces dernières, citons : l'approche compositionnelle qui décompose un système en sous-système et permet de raisonner sur le système global à partir des résultats partiels, et l'approche par raffinement qui consiste à construire le système graduellement et à garantir par construction des propriétés sur le système final. ***Le raffinement formel permet de réduire considérablement l'effort de conception et la vérification par la réduction de la complexité du système à étudier. Notre travail vise l'introduction d'une telle démarche dans la conception des systèmes sur puce.***

1.2.4 Raffinement formel

Le raffinement consiste à détailler une spécification décrite initialement à un haut niveau d'abstraction avec plus de précisions sur les opérations et d'étendre le domaine des données. Une telle démarche aide d'une manière efficace la gestion de la complexité. Mais la vérification des propriétés du système à chaque niveau de raffinement ne suffit pas pour améliorer les problèmes de temps de vérification et l'explosion d'espace des états des niveaux détaillés. Pour contrer ce problème, on cherche à raffiner le système tout en gardant les propriétés vérifiées dans les niveaux abstraits, c'est-à-dire, préserver la validité de la propriété lors de l'application des transformations de raffinement. Cette démarche est appelée "raffinement formel".

Cette notion de raffinement formel a été introduite dans les années 1970 par Dijkstra [42], puis formalisée par Back [102] dans les années 1980 et depuis elle n'a cessé de susciter de l'intérêt [2]. L'intérêt majeur d'utilisation du raffinement formel est la garantie des propriétés d'un système, dans une démarche de construction progressive, tout en maîtrisant la complexité dûe à la grandeur du système cible. La démarche de raffinement est décomposée en quatre phases : la spécification du système dans un langage formel, le raffinement des spécifications, la preuve des propriétés sur les niveaux abstraits et la preuve de préservation des propriétés dans le flot de conception.

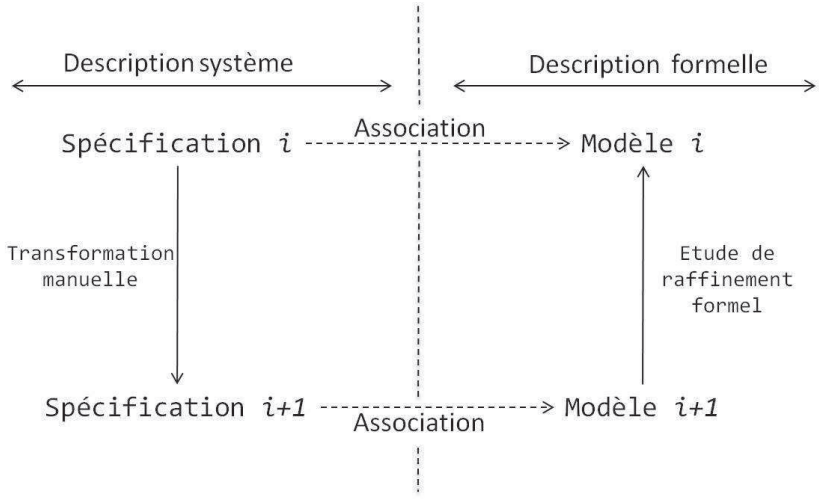


FIG. 1.5 – Schéma d'intégration du raffinement formel

La figure FIG. 1.5 montre un schéma d'utilisation du raffinement formel dans la conception d'un système. Au départ, le concepteur écrit la spécification d'un système dans un niveau d'abstraction (Spécification i) et procède par des rajouts de détails pour obtenir une nouvelle spécification plus détaillée (Spécification $i+1$). Ayant les deux spécifications, le concepteur associe à chaque niveau de spécification un modèle formel. Le concepteur peut analyser la préservation des propriétés. Le raffinement formel garantit que les deux modèles sont cohérents vis-à-vis de certaines propriétés. En d'autres termes, le raffinement garantit que ce qui était vrai en amont est toujours vrai en aval (la flèche qui part du modèle M_{i+1} vers le modèle M_i). Le cas où le modèle M_{i+1} est un raffinement du modèle M_i est noté par $M_i \sqsubseteq M_{i+1}$.

La relation $\sqsubseteq$ doit vérifier trois propriétés importantes : ***la réflexivité***, un système est le raffinement de lui-même, ***la transitivité***, un système peut être raffiné par étapes et ***la monotonie***, c'est-à-dire, l'ajout de nouveaux détails par raffinement ne remet pas en cause les conclusions tirées dans le système initial.

Pour illustrer la notion du raffinement, considérons l'exemple de modélisation d'un canal de communication, cet exemple est tiré de la thèse de [86]. Le canal de communication a pour rôle l'envoi des données par paquet en utilisant un buffer. Dans ce modèle, l'envoi des paquets de données est réalisé par le canal quand le buffer est vide. Le canal renvoie des nouvelles données sur le buffer dans le cas où les données sont totalement traitées (buffer vide) ou s'il reçoit un signal de réinitialisation.

Au départ, le concepteur peut choisir de modéliser le canal de communication à un niveau abstrait tout en cachant l'information sur le buffer. La figure FIG. 1.6 montre une représentation abstraite possible du canal de communication dans le langage B événementiel (event-B)[3], qui est un langage connu pour la modélisation, la vérification et le raffinement des systèmes. Le canal est décrit dans la clause **machine** nommé canal-communication, la description des données est donnée par la clause **variables**, la description des propriétés sur les variables est réalisée par la clause **invariant**, la définition d'état initial du système est donnée par la clause **initialisation** et les événements effectués par le canal sont donnés par la clause **events**. Dans cette spécification abstraite du canal, trois événements sont considérés : l'action d'envoyer *Envoyer* qui émet un paquet de messages d'une taille donnée *TailleEnvoi*, le traitement des données spécifié par l'action *Traiter* qui reçoit et traite les messages un par un, l'abandon du traitement est décrit par l'action *Reset*. Notons que dans ce modèle, toutes les informations sur la valeur des paquets de données, et sur le mode de gestion et de traitement sont abstraites, nous n'avons pris en compte que le nombre de paquets à envoyer *TailleEnvoi*.

machine *Canal_Communication* /*Nom de la machine*/
variables *TailleEnvoi* /*Variables de la machine*/
invariant $TailleEnvoi \in \mathbb{N}$ /*Spécification des variables*/
initialisation $TailleEnvoi := 0$ /*Initialisation des variables*/
events /*Liste des événements*/

Envoyer $\triangleq$ **select** $TailleEnvoi = 0$ **then** $TailleEnvoi :\in \mathbb{N}^+$ **end** ; /* *TailleEnvoi* $:\in \mathbb{N}^+$ signifié que */ /* la valeur de la variable */ /* devient un entier non vide*/

Traiter $\triangleq$ **select** $TailleEnvoi > 0$ **then** $TailleEnvoi := TailleEnvoi - 1$ **end** ;

Reset $\triangleq$ **select** $TailleEnvoi > 0$ **then** $TailleEnvoi := 0$ **end** ;

end

FIG. 1.6 – Exemple de spécification d'un canal de communication abstrait en event-B

Plusieurs raffinements peuvent être réalisés sur cette machine : un raffinement de valeurs de données, une introduction d'autres paramètres "variables" par ajout de détails, une structuration des données du modèle et enfin un changement de granularité des événements traités dans le niveau abstrait, c'est-à-dire, de rendre observables de nouveaux événements qui détaillent le comportement du système.

Reprenons l'exemple de la figure FIG. 1.6, considérons le raffinement de ce modèle par rajout d'information sur la taille de buffer du canal de communication. Cette taille limite le nombre des paquets en attente et permet le raffinement de la gestion d'envoi. Sur ce raffinement, nous supposons que l'envoi et le traitement des données depuis et vers le buffer se fait un par un. Un modèle possible de la machine raffinée est montré sur la figure FIG. 1.7. Dans cet exemple, des détails de manipulation des données sont rajoutés, la variable *TBuffer* représente la taille du buffer, la variable *DBuffer* représente le nombre de données qui existent dans le buffer et la variable *AEnvoyer* représente le nombre de données à envoyer vers le buffer. La taille d'envoi *TailleEnvoi* de la machine abstraite est alors spécifiée dans ce raffinement par la somme des deux variables *DBuffer* et *AEnvoyer*. L'événement abstrait *Envoyer* est alors décomposé en deux événements : *Envoyer* qui permet d'initier le nombre de paquets à envoyer par l'insertion d'une valeur à la variable *AEnvoyer*, et *Envsuite* qui décrit le transfert des paquets un par un du canal vers le buffer. Cet événement est codé par la décrémentation de la variable *AEnvoyer* et l'incrémentation de la variable *DBuffer*. Enfin, les deux événements abstraits *Traiter* et *Reset* sont réécrits pour la manipulation des variables du modèle concret pour l'événement de traitement des données sur le buffer et la réinitialisation du canal.

machine *Canal_Communication_R*
refines *Canal_Communication*
constants *TBuffer*
properties $TBuffer \in \mathbb{N}^+$
variables *DBuffer*, *AEnvoyer*
invariant $AEnvoyer \in \mathbb{N} \wedge DBuffer \in 0..TBuffer \wedge DBuffer + AEnvoyer = TailleEnvoi$
variant *AEnvoyer*
initialisation $DBuffer := 0$, $AEnvoyer := 0$
events
Envoyer $\hat{=}$ **select** $AEnvoyer = 0 \wedge DBuffer = 0$
then $AEnvoyer :\in \mathbb{N}^+$ **end** ;

Traiter $\hat{=}$ **select** $DBuffer > 0$
then *DBuffer :=Dbuffer-1* **end** ;

Reset $\hat{=}$ **select** $DBuffer > 0 \wedge AEnvoyer > 0$
then $AEnvoyer := 0 \,||\, DBuffer := 0$ **end** ;

EnvSuite $\hat{=}$ **select** $DBuffer < TBuffer \wedge AEnvoyer > 0$
then $AEnvoyer := AEnvoyer - 1 \,||\, DBuffer := DBuffer + 1$ **end** ;
end

FIG. 1.7 – Exemple d'un canal de communication raffiné en event-B

Le processus de raffinement peut être poursuivi par d'autres transformations telles que, la définition de la structure de données à envoyer, la définition des domaines des valeurs des paquets, le raffinement du comportement du traitement des données, le raffinement du comportement qui mène à la réinitialisation d'envoi ....

Pour vérifier la préservation de comportement entre deux niveaux, une liaison entre les deux niveaux doit être définie et le raffinement doit être établi selon des propriétés exprimées sur cette liaison. Par exemple, en B cette liaison est définie sur des variables et le raffinement est établi après la vérification des obligations de preuve qui sont bien définies. Dans l'exemple du canal, la liaison entre les deux modèles est réalisée par l'égalité entre la variable de taille d'envoi *TailleEnvoi* du modèle abstrait et la somme des deux variables *DBuffer* et *AEnvoyer* du modèle raffiné. Le raffinement est établi car l'égalité est vérifiée pour toutes les exécutions possibles du système.

En fait, l'étude de raffinement peut être réalisée selon plusieurs sémantiques. Le choix de la sémantique est fortement lié à la nature des propriétés à vérifier et les caractéristiques du modèle formel. Il existe plusieurs sémantiques de raffinement [102], les plus connues sont la sémantique de transformation de prédicats et la sémantique opérationnelle. La variété des sémantiques créée un ensemble de langages et outils de raffinement formel. ***La méthode*** B est probablement la plus connue, cette méthode a été introduite au milieu des années 80 par J.-R. Abrial [2]. Elle se base sur la théorie des ensembles et sur la logique des prédicats pour décrire le raffinement et la preuve formelle des systèmes. Cette méthode permet d'établir une chaîne de production qui va de la spécification du programme au code source associé ; en partant d'un modèle abstrait (une machine abstraite), puis en réduisant le niveau d'abstraction, étape par étape (par des raffinements), jusqu'à l'obtention d'un modèle concret, proche du code source (l'implémentation). Aussi, une variante de la méthode B est développée pour les systèmes réactifs event-B [3], elle se base sur la description des ensembles d'événements déclenchables selon un ensemble de conditions appelées gardes, ces derniers déterminent l'ensemble d'exécutions possibles d'un système. Plusieurs outils sont issus de cette méthode : atelier B [35] et B-Toolkit [10]. ***La méthode*** Z [113] est un des premiers langages qui propose des règles de raffinement. Elle est standardisée depuis 2002. Cette méthode utilise une sémantique relationnelle [102], le raffinement s'exprime au niveau des types de données pour garantir la préservation de propriété entre les différents niveaux de raffinement. Issu de Z, l'OBJECT-Z [108] est un langage qui supporte les notions d'objets et de classes. Il existe plusieurs outils intégrant la spécification et la preuve en Z tel que, HOL-Z [112] un environnement de preuve de spécification Z utilisé comme un plug-in dans l'outil de preuve générique ISABELLE/HOL. ***L'outil*** VDM[75] originairement développé par les laboratoires IBM en 1970, VDM est un langage qui se base sur une sémantique opérationnelle pour la description d'un programme. L'utilisation du VDM commence par un modèle abstrait de la spécification suivi par un processus de concrétisation jusqu'à une implémentation. Chaque étape requiert le raffinement des données et la décomposition des opérations. Cette méthode utilise aussi une sémantique relationnelle pour la preuve du raffinement des données et une sémantique de prédicats pour la preuve du raffinement du comportement. Cette méthode est outillée, VDMTOOL est un des outils les plus connus, il permet la génération automatique du code C++ et JAVA. ***L'outil*** FDR [45, 99] est un outil de model-checking basé sur l'algèbre de processus CSP. Cet outil permet également d'analyser et d'étudier la relation de raffinement entre différents composants (machines à états) en utilisant différentes sémantiques de raffinement d'algèbre de processus : raffinement de traces, raffinement traces-divergences.

1.2.5 Raffinement formel dans les systèmes embarqués

Il existe plusieurs travaux qui tentent l'intégration du raffinement formel dans la conception des systèmes sur puce. La majorité des travaux de raffinement utilisent la méthode B comme support et outil pour la preuve. Parmi ces travaux nous citons les travaux de thèse de Colley [36] ; l'auteur s'est basé sur une démarche de raffinement des composants des systèmes sur puce, pour étudier le raffinement des machines pipeline et les latences de mémoires en utilisant le langage Event-B. Dans [23] Boulusset propose une démarche de liaison entre la spécification d'architecture logicielle et le monde formel en utilisant des patrons de réécriture des modèles qui sont écrites en π-SPACE, langage de description d'architecture logicielle basé sur les opérateurs du langage formel π-CALCULS. Ces patrons permettent l'obtention des descriptions en langage B, ce qui permet d'étudier le raffinement entre les niveaux de spécification. Dans [31] Méry et al. proposent une démarche de vérification incrémentale des systèmes embarqués basée sur le raffinement en Event-B. L'approche se base sur la spécification du modèle d'ordonnanceur du système décrit dans le langage SystemC, le modèle du système est alors décrit comme des instanciations décrivant une étape de raffinement sur le modèle formel de l'ordonnanceur. Ces travaux proposent une description formelle des systèmes sur puce dans le but d'appliquer les concepts de vérification et de raffinement formels. Malheureusement, ces travaux supposent une expertise dans le domaine formel des concepteurs de systèmes, ce qui n'est pas toujours le cas.

D'autres travaux existent dans le cadre de preuve par raffinement des spécifications architecturales, des protocoles ou des composants de traitement dédiés. Dans [30] les auteurs proposent une preuve par raffinement de la propriété producteur/consommateur pour le protocole PCI, une donnée envoyée par le producteur doit être lue par le consommateur avant son écrasement. Ce travail a mené à proposer des corrections sur le modèle du protocole du multi-Bus PCI 2.1 car la spécification initiale des normes PCI ne respecte pas cette propriété. Dans [103] les auteurs proposent une formalisation du protocole de bus PI en utilisant le langage CSP pour sa spécification, deux implémentations de la spécification ont été réalisées et la preuve de raffinement est réalisée par l'outil FDR. D'autres travaux visent les aspects de sécurité sur les différents protocoles. Les auteurs de [17] proposent des patrons de conception basée sur Event-B pour des protocoles de cryptographies. La spécification du protocole passe par trois modèles de description. Le premier modèle représente la spécification abstraite du protocole et les propriétés de sécurité qu'elle doit vérifier, le second modèle représente l'implémentation de la spécification, et dans le troisième modèle, un modèle d'attaques introduit sur le modèle précédent, et la vérification est réalisée sur les obligations de preuve générées. Les auteurs de [100] construisent une preuve sur la sécurité du mécanisme de transaction des données dans les EEPROM des cartes à puce. La spécification formelle et la preuve de la sécurité des transactions sont réalisées en utilisant la méthode B. L'ensemble de ces travaux ciblent une spécification particulière permettant de valider ces spécifications et de détecter les erreurs de conception. Par contre, ces travaux ne proposent pas une démarche d'aide à la conception par des étapes de raffinement cadrées et valides.

Abdi [62, 32, 105] s'est intéressé à la préservation de comportement dans la conception et le développement des systèmes sur puce. Afin de cadrer le raffinement et de vérifier la préservation du comportement des spécifications dans un flot de conceptions, Abdi a défini un formalisme de modélisation des spécifications au niveau système. Le modèle

proposé est un modèle algébrique (Model Algebra) qui représente le système par un bloc de calcul avec des entrées et sorties symbolisant les requêtes et les réponses, ainsi que des variables dans lesquelles peuvent être stockées les données. La conception des blocs peut être hiérarchique. La notion de raffinement est introduite dans le but de distribuer le comportement des spécifications dans des composants architecturaux. Le raffinement d'un système dans la démarche d'Abdi est réalisé par des transformations sur le modèle initial pour trouver le partitionnement d'une architecture souhaitée ; les transformations sont de deux types : des réarrangements et des remplacements de blocs. Un exemple de transformation est le remplacement de plusieurs canaux de communication par un seul canal partagé. Pour vérifier la validité des modèles définis à chaque étape du raffinement, Abdi vérifie l'équivalence observationnelle entre les variables d'entrées et de sorties des modèles de différents niveaux. Cette démarche ne commence pas à un niveau de description assez abstrait et ne va pas jusqu'au cycle près, le système est toujours représenté dans un niveau système. Les transformations ne recouvrent pas tous les aspects de raffinement : la décomposition des données et la décomposition du transfert ne sont pas traités. Elle repose sur des phases de construction de modèles : une phase de partitionnement qui représente une phase de réarrangement du modèle, une phase de sérialisation pour la séquentialisation du comportement, une phase de changement d'ordonnancement de communication, et une phase de cheminement de transfert dans laquelle les composants de transfert sont introduits pour la réalisation de la communication.

1.3 Notre proposition

Comme évoqués dans ce chapitre, les méthodologies de conception des systèmes sur puce existantes n'offrent pas la possibilité de valider la préservation des propriétés à chaque étape de raffinement et les méthodes de vérification classiques sont devenues impuissantes devant la dimension et la complexité des systèmes. De plus, les démarches de raffinement sont peu exploitables dans les démarches de conception des systèmes sur puce à cause de la difficulté de cette tâche et du manque d'expertise des concepteurs. Actuellement, la majorité des travaux sur le raffinement formel dans la conception des systèmes sur puce ciblent le raffinement de spécification et de protocole des composants matériels.

Dans notre travail, nous cherchons à offrir une aide aux concepteurs des systèmes sur puce [81, 82, 85] au travers d'une démarche par raffinement. Cette démarche permet, d'une part, l'assistance du concepteur dans le processus de rajout de détails lors de la transformation des descriptions abstraites du système dans la phase d'exploration d'architecture (particulièrement le raffinement d'élément de communication). D'autre part, elle permet de raisonner sur les différents niveaux d'abstraction induits par les raffinements successifs par l'exploitation des techniques de preuve de préservation de propriétés. Nous souhaitons fournir au concepteur des outils formels permettant d'introduire progressivement les détails architecturaux et de vérifier la cohérence des modèles issus des différentes étapes de raffinement. ***Dans le cadre de cette thèse nous nous sommes intéressés au raffinement des médiums de communication***.

Notre approche [83, 84], contrairement à celle d'Abdi, démarre à un niveau de modélisation fonctionnelle très abstrait permettant l'abstraction des valeurs des données de calcul et la parallélisation maximale entre des processus. Les transformations d'Abdi visent essentiellement le rangement des blocs pour la description d'une architecture, tandis que

dans notre travail, nous effectuons des rajouts de détails dirigés par les contraintes d'architecture : raffinement des actions par décomposition de données, rajout de synchronisation et partage de ressources. Les modèles mathématiques sur lesquels se base notre démarche sont différents de ceux utilisés dans [62]. Notre approche vise à assurer le bon fonctionnement des systèmes par la préservation du comportement décrit comme des suites d'actions exécutées par le système, et donc la vérification et la préservation des propriétés de traces d'actions. Abdi assure la préservation des valeurs des variables même si le comportement change, c'est-à-dire, équivalence observationnelle sur les variables, ce qui n'est pas dans nos objectifs. La figure FIG. 1.8 montre notre démarche pour étendre le flot de conception système. Cette démarche se base sur un niveau très abstrait de modélisation du fonctionnement du système et offre un processus de raffinement du modèle fonctionnel par trois étapes de transformation permettant l'intégration des contraintes architecturales. La spécification d'un système au NIVEAU_1 représente le modèle de l'application après le raffinement de la granularité de données abstraite vers la granularité de données d'implémentation, la spécification du système à ce niveau est contraint par des paramètres des composants de calcul et de stockages de l'architecture cible. La spécification du système au NIVEAU_2 explicite les signaux de synchronisation requis pour l'accès aux données des canaux. La spécification d'un système au NIVEAU_3 rajoute l'information de cheminement des données entre les déférents composant de l'architecture à travers un support de communication partagé. En référent à la zone d'activité des différents environnements (la figure FIG. 1.2), notre approche offre la vérification formelle et le raffinement guidé entre la spécification fonctionnelle du système à très haut niveau d'abstraction (le NIVEAU_0) et la spécification abstraite de la structure du système sans prise en compte de l'aspect temps (le NIVEAU_3).

1.3.1 Cadre méthodologique

Dans le cadre de notre travail, nous nous basons sur la démarche de conception suivant le schéma général en Y [48, 43]. Cette méthodologie commence par la description de l'application à un haut niveau d'abstraction sans détails d'architecture et par une étape de projection l'application est associée à une architecture. Ce processus de projection permet à cette méthode de conception d'intégrer parfaitement le flot de conception par raffinement : dans un processus itératif, des informations et des détails peuvent être ajoutés au modèle initial jusqu'à l'obtention du code source des modules logiciels et du modèle synthétisé des modules matériels.

1.3.2 Niveau abstrait de modélisation

L'un des problèmes de la phase d'exploration d'architecture est le temps nécessaire pour trouver une solution acceptable après le processus d'analyse et d'estimation notamment pour la vérification formelle. La vérification est confrontée aux limitations dues à la complexité des modèles à vérifier car même si ces modèles permettent de décrire le système en un haut niveau d'abstraction, la description du calcul et des données avec leurs structures reste très détaillée. Dans notre travail, nous cherchons à augmenter le niveau d'abstraction du système pour réduire la complexité du modèle et profiter au maximum du processus de raffinement : réduire le temps de vérification et prendre des décisions plus tôt lors de la conception.

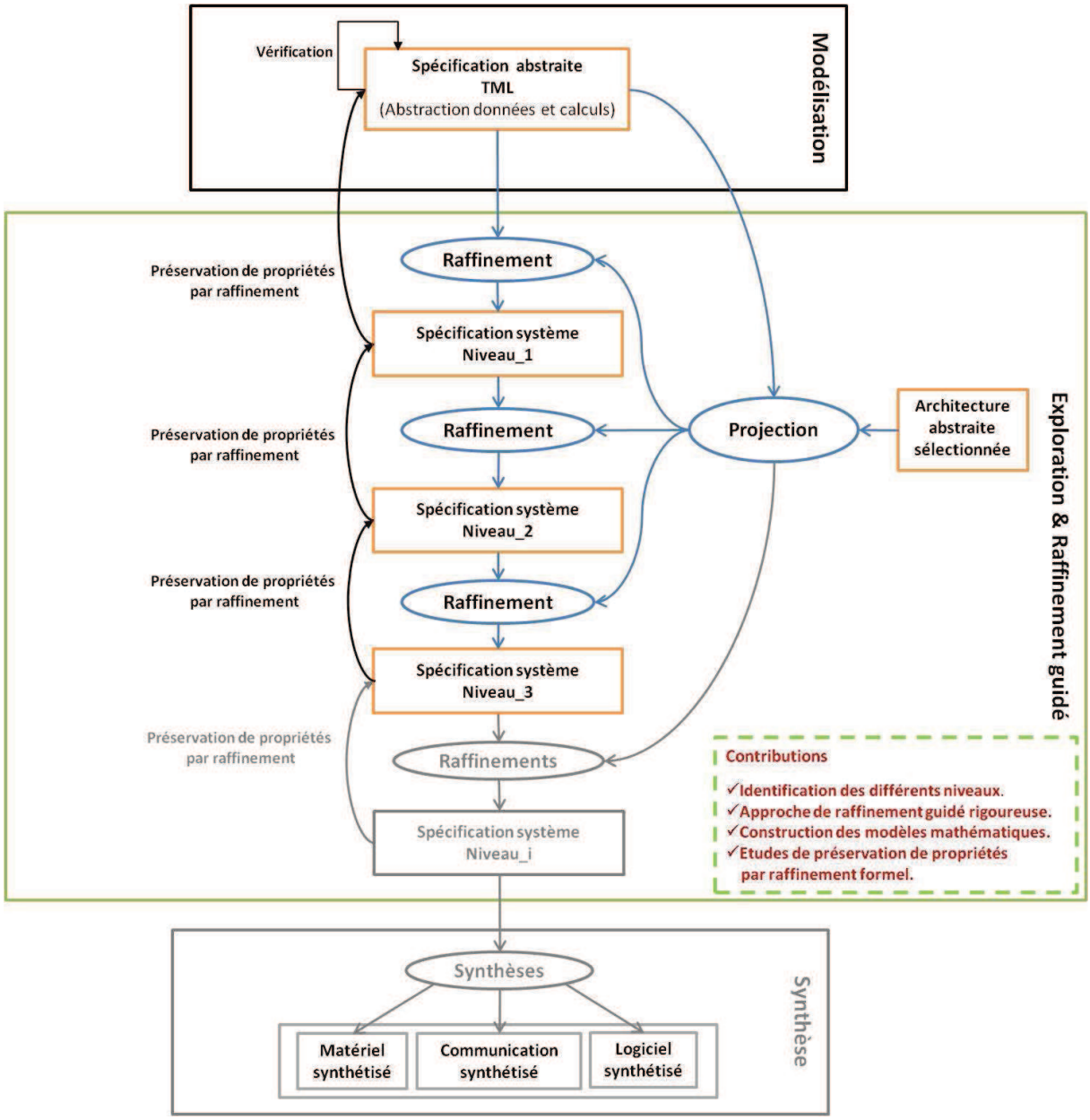


FIG. 1.8 – Notre proposition dans de flot de conception système

À cet effet, nous avons choisi d'utiliser un langage abstrait, appelé TML (Task Modeling Language) pour décrire les systèmes [6]. Ce langage permet de modéliser le comportement fonctionnel d'un système à haut niveau d'abstraction sans les détails d'architecture et offre un mécanisme d'abstraction des valeurs des données ainsi que les traitements. En TML un système est composé essentiellement d'un ensemble de processus appelés tâches décrivant le comportement du système. Les tâches communiquent entre elles via des supports de communication appelés canaux. En TML, il n'y a aucune différence entre les tâches logicielles (software) exécutées sur un processeur et les tâches matérielles implémentées sur un coprocesseur matériel (hardware).

Un modèle TML est proche du formalisme des réseaux de processus de Kahn [74] très utilisés dans les approches d'exploration d'architecture. En effet, le modèle de TML est un assemblage de processus, communicants de manière asynchrone par des files de types FIFO. Contrairement au processus de Kahn qui définit un seul type de canal FIFO unidirectionnel point à point infini, TML définit trois types de canaux de communication : des canaux de transfert des données unidirectionnel point à point, des canaux d'événements représentant les contrôles point à point et des canaux de communication des requêtes unidirectionnels représentant les contrôles point à multipoint. De plus, en TML les canaux de données ne transportent pas la valeur des données, juste la quantité des données est spécifiée. Enfin, les tâches TML contiennent une instruction de choix indéterministe, ce qui sort du modèle de KAHN.

1.3.3 Raffinement de communication guidé

Comme nous concrétisons le modèle d'application, le mécanisme de communication matériel doit être pris en compte car le comportement des tâches dépend du comportement du médium de communication. Plus précisément, le comportement et les contraintes imposées par la plate-forme doivent être intégrés dans le modèle de départ. Les composants d'une architecture communiquent par l'intermédiaire de supports de communication partagés appelés "bus" ou d'un "réseau sur puce" (NoC). Nous avons choisi de nous focaliser sur le raffinement des médiums de communication abstraits vers les bus hiérarchiques, vu leur large utilisation, en garantissant la correction du comportement de modèle du système résultant vis-à-vis son modèle initial.

Puisque nous commençons par un haut niveau d'abstraction de modélisation du système, plusieurs détails et paramètres du système doivent être raffinés sur le niveau système avant la synthèse du niveau RTL. Le processus de raffinement que nous proposons ne comportent pas tous les aspects de communication existants dans les systèmes sur puce, ils représentent néanmoins un noyau caractéristique. Dans notre travail actuel, nous sommes intéressés à la caractérisation, la formalisation et la validation des étapes de raffinement que nous avons proposées. Ces étapes de raffinement sont appliquées manuellement mais le processus de vérification de préservation des propriétés est automatisé en utilisant les outils de raffinement formel. Toutefois, les étapes de raffinement peuvent être automatisées car les transformations sont bien définies et caractérisées sur des relations d'ordre entre les différentes actions.

1.3.4 Contributions

Nous résumons notre contribution par les points suivants :

1. ***Identifier des niveaux de description du système et définir des étapes de transformation entre ces niveaux***. Dans la pratique courante, l'étape de raffinement est plus ou moins explicitée dans le raffinement des systèmes sur puce. Dans un premier temps, nous avons identifié des phases et des niveaux qui intègrent différents détails de communication. Nous avons alors proposé un ensemble de transformations qui modifient la structure et illustrent le changement de niveau de l'architecture. Bien que ces transformations et niveaux ne comportent pas tous les aspects de communication existants dans les systèmes embarqués, ils représentent néanmoins un noyau caractéristique de processus du raffinement de ces systèmes.

2. ***Proposer une nouvelle approche de transformation rigoureuse basée sur la notion d'ordre en utilisant le formalisme*** POMSET [94]. Le choix de formalisme décrivant l'ordre offre plusieurs avantages : il permet de faciliter la gestion des événements d'un système lors de la transformation. Il facilite l'association et la description des modèles formels aux différents niveaux descriptifs, ainsi que l'analyse et la preuve de préservation des propriétés.

3. ***Étudier la préservation des propriétés linéaires entre les modèles des niveaux successifs***. Pour étudier la préservation des propriétés nous transformons naturellement des spécifications POMSET des composants vers les automates finis étiquetés (LTS pour Labeled Transition System). La construction du modèle global des automates se fait par produit synchronisé des automates des composants du système. Cela permet d'utiliser la puissance des sémantiques de raffinement et les outils développés sur les algèbres de processus [111].

   Les propriétés analysées sur un système sont de deux types : propriétés linéaires qui décrivent des traces d'exécution et des propriétés d'arborescences qui décrivent la structure des branchements d'un système. Dans le flot de raffinement, par ajout de contraintes architecturales, il est fort probable de perdre des branchements du système décrit dans le modèle abstrait. Nous avons donc choisi d'analyser la préservation des propriétés linéaires qui ont plus de chance d'être préservées. De plus, avec l'utilisation large des propriétés linéaires, avoir une démarche de vérification des propriétés linéaire par raffinement est plus intéressant et permet de gagner un temps considérable dans la phase de vérification des systèmes complexes. La vérification de préservation des propriétés linéaires est réalisée par l'analyse de la non-introduction de nouveaux états de blocage et de nouveaux chemins d'exécution.

4. ***Montrer la faisabilité de notre démarche par une étude de cas.*** Nous avons choisi d'analyser la transformation d'une étude de cas réel d'une application d'un appareil photo numérique issue de [110]. Même si l'application des transformations et le passage d'un niveau à un autre sont actuellement réalisés manuellement, des résultats prometteurs de raffinement de la spécification sont mis en évidence. Dans cette phase du travail, nous avons spécifié notre système avec le langage FIACRE et réalisé la preuve de préservation des propriétés par raffinement avec l'outil CADP [51].

1.4 Conclusion

Dans ce chapitre, nous avons présenté une vue générale sur les différents flots et environnements de conception depuis le niveau système, et particulièrement celles qui intègrent la vérification formelle dans le processus de développement. Aussi, nous avons parcouru les différentes méthodes de vérification des systèmes, leurs intérêts et leurs limites. Une voie d'amélioration de l'aspect de vérification des systèmes est l'intégration de la notion de raffinement formel. Cette technique est restée peu utilisée à cause du manque d'expertise des concepteurs sur les notions d'abstraction, de raffinement et de preuve.

Nous proposons dans cette thèse une nouvelle approche d'aide aux concepteurs dans une démarche de conception des systèmes sur puce par raffinement. Notre approche démarre à un niveau de modélisation fonctionnelle très abstrait, permettant l'abstraction des valeurs des données de calcul et la parallélisation maximale entre des processus qui permet de gérer la complexité du système et de vérifier rapidement les propriétés associées. Elle vise aussi la systématisation et l'extension de conception incrémentale dans les systèmes sur puce en formalisant les différentes transformations et en raffinant les systèmes sur puce de telle sorte que le système obtenu préserve le comportement du système d'entrée.

Nous avons choisi de suivre une méthodologie de conception en Y vu son adaptation à l'approche de raffinement et de conception de système par niveau et son aspect de séparation entre les différentes vues de système (application et architecture). Cette séparation permet de contrôler parfaitement l'introduction des détails architecturaux par étape ainsi que la gestion de la complexité de ces systèmes.

Dans le chapitre suivant nous décrivons notre démarche de raffinement de communication en détaillant les différents concepts : application, architecture, projection et le processus de raffinement de communication que nous proposons avec les différents niveaux de construction que nous définissons.

Deuxième partie

Contributions

Chapitre 2

Cadre méthodologique

Sommaire

Le chapitre précédent a montré les différents environnements de conception basés sur le flot de conception système, les méthodes de vérification utilisées et leurs avantages et inconvénients. Une idée centrale d'amélioration de la vérification dans le flot de conception système est l'intégration du raffinement formel tout au long du processus de conception. Dans ce travail de thèse, nous proposons un flot de conception guidé permettant le passage entre différents niveaux de spécification de système en vérifiant la préservation des propriétés fonctionnelles par des approches de raffinement formel.

Dans ce chapitre, nous présentons le cadre méthodologique dans lequel se place notre approche de vérification par raffinement. Nous allons décrire les modèles d'application et d'architecture choisis, la famille de support de communication ciblée, les règles de projection et de partitionnement prises en compte. Et enfin, nous décrivons les étapes de raffinement de communication proposées, les différents niveaux de spécification et les différentes contraintes d'architecture considérées sur chaque étape de raffinement.

2.1 Méthodologie en Y

La méthodologie adoptée s'appuie sur le schéma général en Y (voir FIG. 2.1). Un système est construit à partir d'un modèle d'une application et d'un modèle d'une architecture fournis séparément. Le concepteur propose une projection et un partitionnement initiaux qui permettent l'exécution de l'application sur l'architecture pour obtenir la première plate-forme. L'évaluation de cette dernière révèle si les choix d'architecture et de partitionnement satisfont ou non les exigences définies préalablement.

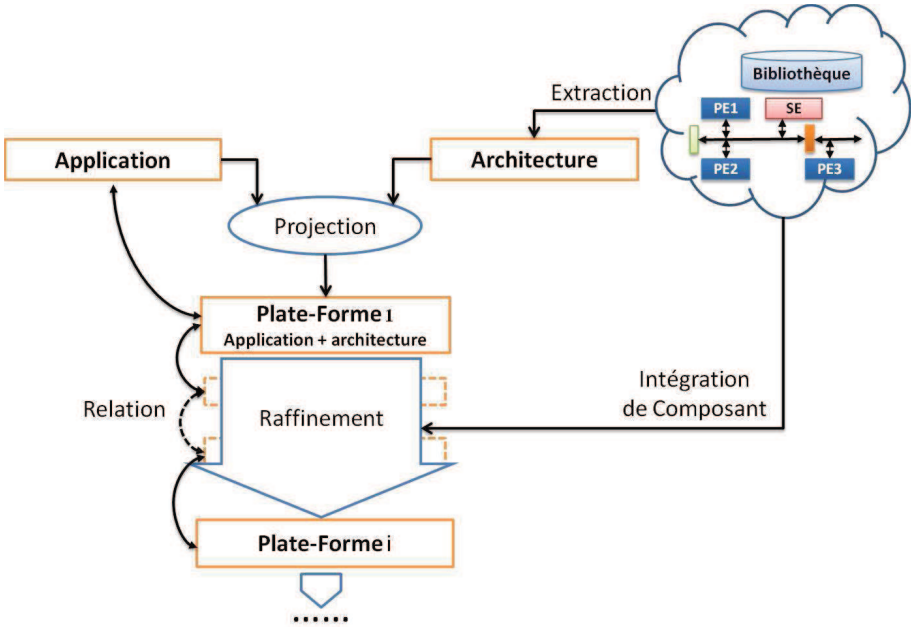


FIG. 2.1 – Notre méthodologie de conception basée le schéma Y-chart

La séparation du modèle de l'application et du modèle de l'architecture permet au concepteur d'utiliser un modèle d'application unique afin de tester diverses architectures et partitionnements. En effet, l'application peut être projetée sur une gamme de modèles d'architecture pouvant représenter différentes architectures ou différentes instances d'une même architecture.

2.1.1 Modèle d'application

Le modèle d'application décrit dans un langage spécifique un comportement et une fonctionnalité d'un système sans prendre en compte des détails architecturaux. Pour modéliser une application à un haut niveau d'abstraction, l'équipe LABSOC propose un langage très simple appelé TML (Task Modeling Language) [6]. Une application TML est composée essentiellement d'un ensemble de processus appelés tâches décrivant un comportement séquentiel. Au niveau des modèles TML, il n'y a aucune différence entre les tâches logicielles (software) et les tâches matérielles implémentées sur un coprocesseur matériel (hardware). De plus, TML ne permet pas la hiérarchisation de comportement, c'est-à-dire, il ne permet pas le parallélisme dans une tâche. Les données manipulées par une tâche sont typées, il

est possible de définir des types (entier, booléen, image, pixel, ...) qui sont de tailles différentes. Les types se mesurent par une unité de base que nous appellerons *unité du type* (que nous notons UNIT). Les tâches TML communiquent entre elles via des supports de communication de trois sortes : des canaux de communication de données de traitement ne transportant que les quantités des données, des canaux de communication par événements et des canaux de communication par requêtes représentant les valeurs des signaux de contrôles.

Tâche (***Task***) : est l'unité de base d'un code TML. Une tâche décrit un processus séquentiel de calcul par des instructions de traitement, de contrôle ou de transfert. Les valeurs des données manipulées ne sont pas représentées ; les transferts et les traitements des données portent sur des échantillons décrivant la quantité de données manipulées ou transférées. Une tâche s'exécute d'une manière infinie, elle peut être bloquée dans le cas d'attente de données de calcul, d'événements ou de requêtes de contrôle.

Canal (***Channel***) : est un élément de communication unidirectionnel point-à-point permettant le transfert des données de calcul entre les différentes tâches. Du fait de l'abstraction des valeurs des données, elle est représentée sur le canal uniquement par la quantité (sans connaître la valeur) des données véhiculées. TML offre trois types de canaux de communication des données : les FIFOs finies bloquantes en lecture et en écriture notées (BR-BW), les FIFOs infinies qui sont bloquantes en lecture seulement notées (BR-NBW), et des canaux qui sont non bloquants en lecture et en écriture notés (NBR-NBW).

Événement (***Event***) : Les tâches peuvent communiquer entre elles par des événements. La communication par événements est unidirectionnelle point-à-point et permet le transfert des données de contrôle. Contrairement au canal de données, un canal du type *Event* contient un espace de stockage de la valeur de la donnée à transmettre. La politique de gestion de l'espace est réalisée en FIFO qui peut être fini ou infini.

Requête (***Request***) : Les tâches peuvent communiquer entre elles aussi par des requêtes. Le canal de type *Request* est un canal de transfert de données de contrôle. La seule différence avec les événements est que le canal *Request* est un canal infini unidirectionnel point-à-multipoint.

Dans la suite, nous n'utilisons que les constructeurs basés sur le transfert et le traitement des données abstraites, c'est-à-dire, nous ne traitons pas les constructeurs de contrôles (Event et Request). La partie du langage considérée décrit bien les applications de flot de données. Ce choix n'est pas une restriction forte car le contrôle peut être décrit par les données abstraites, sauf qu'avec une telle abstraction les conditions des choix deviennent toutes indéterministes. En outre, pour les communications point-à-multipoint, nous pouvons les remplacer dans certains cas par un ensemble de canaux de données simulant le fonctionnement point-multipoint souhaité. Le tableau TAB. 2.1 présente la grammaire des tâches et canaux du langage TML. Les tâches portent un nom, elles sont construites par des instructions de base de programmation impérative, déclarations des variables locales, les boucles,

l'alternative et les opérations. Nous trouvons aussi des opérations de communications et de synchronisation. Les canaux portent un nom, ils sont typés, ⟨*name_type*⟩ spécifie le nombre de données manipulées est exprimé par une valeur entière ⟨*width*⟩, la taille du canal est calculée par la multiplication du type de données et le nombre de données manipulées. Pour les canaux infinis le paramètre ⟨*width*⟩ est vide.

⟨*task*⟩	:=	***task*** ⟨*name_task*⟩ {⟨*variables*⟩$^+$; ⟨*inst_bloc*⟩}
⟨*inst_bloc*⟩	:=	⟨*inst*⟩ | ⟨*inst*⟩; ⟨*inst_bloc*⟩
⟨*inst*⟩	:=	⟨*inst_atomic*⟩ | ⟨*inst_cond*⟩ | ⟨*inst_loop*⟩
⟨*inst_atomic*⟩	:=	***read*** ⟨*num*⟩ ⟨*name_ch*⟩
	|	***write*** ⟨*num*⟩ ⟨*name_ch*⟩
	|	***exec*** ⟨*num_iterations*⟩
⟨*inst_loop*⟩	:=	***repeat*** ⟨*num_iterations*⟩ ***times*** ⟨*inst_bloc*⟩ ***endrepeat***
⟨*inst_cond*⟩	:=	***if*** ⟨*condition*⟩ ***then*** ⟨*inst_bloc*⟩ ***endif***
	|	***if*** ⟨*condition*⟩ ***then*** ⟨*inst_bloc*⟩ ***else*** ⟨*inst_bloc*⟩ ***endif***
⟨*variables*⟩	:=	***variable*** ⟨*name*⟩ ⟨*type*⟩
⟨*condition*⟩	:=	⟨*condition*⟩⟨*operator*⟩⟨*condition*⟩
	|	(⟨*condtion*⟩)
	|	⟨*name_var*⟩
	|	⟨*integer*⟩
	|	⟨*boolean*⟩
⟨*channel*⟩	:=	***channel*** ⟨*name*⟩⟨*name_type*⟩⟨*ch_type*⟩⟨*width*⟩⟨*param*⟩
⟨*param*⟩	:=	⟨*name*⟩ ⟨*name*⟩
⟨*ch_type*⟩	:=	*BR-NBW*
	|	*NBR-NBW*
	|	*BR-BW*
⟨*type*⟩	:=	***type*** ⟨*name_type*⟩⟨*num*⟩
⟨*num_iterations*⟩	:=	⟨*num*⟩
⟨*operator*⟩	:=	== |! = | > | < | <= | >= |*or*|*and*
⟨*name_type*⟩	:=	$[a...z, A...Z][a...z, A...Z, O...9, -]$
⟨*name_task*⟩	:=	$[a...z, A...Z][a...z, A...Z, O...9, -]$
⟨*name_var*⟩	:=	$[a...z, A...Z][a...z, A...Z, O...9, -]$
⟨*name_ch*⟩	:=	$[a...z, A...Z][a...z, A...Z, O...9, -]$
⟨*width*⟩	:=	$[1...256]$
⟨*num*⟩	:=	$[1...256]$
⟨*boolean*⟩	:=	***true*** |***false***
⟨*integer*⟩	:=	$[+\|-]$⟨*nat*⟩

TAB. 2.1 – Grammaire d'un noyau du langage TML

Pour une utilisation ultérieure, nous donnons une représentation graphique d'un exemple d'une application TML (voir figure FIG. 2.2) : les tâches sont décrites par des rectangles ; les canaux par des ellipses et des liens entre les tâches qu'ils relient. Dans cette figure, l'application est composée de quatre tâches T1, T2, T3 et T4 reliées entre elles par trois médiums de différents types : un canal de données C1, un canal de type Event E1 et canal de type Request R1. La tâche T2 échange des données avec la tâche T3 via C1, et communique par des événements avec la tâche T4 via E1.

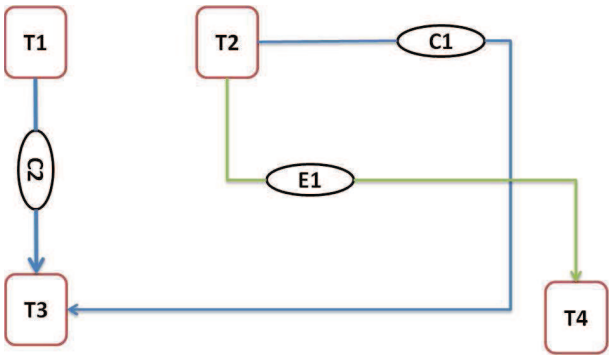


FIG. 2.2 – Un exemple d'un schéma graphique d'application TML.

Nous caractérisons formellement une application par la définition suivante.

Définition 1 (Application, Tâches et Canaux) *Une* application $\mathcal{T}$ *est un 2-tuplet d'ensemble des composants tel que :*

$$\mathcal{T} = \langle \mathcal{T}asks, \mathcal{C}hannels \rangle$$

- $\mathcal{T}asks$ *est un ensemble fini de* tâches. *Pour chaque tâche* $t \in \mathcal{T}asks$ *est associée :*
  - $\mathcal{A}cts(t)$ *: un ensemble d'*actions *qui représente les instructions de lecture, d'écriture et d'exécution de la tâche* t.
  - $\mathcal{B}ehav(t)$ *: une fonction qui décrit le* fonctionnement *séquentiel de la tâche* t. *Ce fonctionnement est donné par une machine à états extraite du code TML, la transition entre les états de la tâche est réalisée par l'exécution des actions de la tâche.*

- $\mathcal{C}hannels$ *: un ensemble fini de* canaux. *Pour chaque* $c \in \mathcal{C}hannels$ *est associé la taille du canal* $size(c)$, *le type de données manipulées par le canal exprimé en unité de type* UNIT, *le type de gestion de données du canal* $type(c)$ *et un couple* $(src(c), tgt(c)) \in Tasks^2$ *telles que* $src(c)$ *est la tâche d'entrée et* $tgt(c)$ *est la tâche de sortie.*

2.1.2 Modèle d'architecture

Une architecture est un ensemble de composants matériels assemblés et reliés entre eux. L'architecture générale d'un système sur puce est constituée de composants dédiés, des composants de mémorisation, des périphériques d'entrée/sortie et de communication (appelés composants matériels) et des composants processeurs programmables standards disposant de tous les composants logiciels nécessaires à l'exécution des tâches tels que le système d'exploitation, les pilotes des périphériques (appelés composants logiciels). Un modèle d'architecture décrit les ressources architecturales et leurs structures. Le modèle peut ne comporter que des composants matériels ou bien des composants matériels/logiciels. Durant le processus de conception d'un système embarqué le choix des composants d'architecture ainsi que leurs interconnexions peut être prédéfini, dans le cas d'une architecture fixe, ou construit à partir des composants qui peuvent être sélectionnés par le concepteur à partir d'une bibliothèque.

Les différents composants d'une architecture et leurs paramètres peuvent être représentés dans plusieurs niveaux de description selon les choix d'abstraction. Un grand nombre de langages et de modèles de description d'architecture existe, ces langages sont utilisés essentiellement pour la description détaillée de la structure, les connecteurs et le comportement des différents composants d'une architecture. Les langages les plus connus pour la description de l'architecture sont SDL [66], AADL [44], SYSTEMC [89] et aussi UML qui est souvent utilisé dans des phases de conception en amont pour la spécification et l'analyse du système, mais rarement durant les opérations de spécifications détaillées de conception car il n'est pas assez précis pour décrire de manière détaillée les architectures. Dans notre démarche de conception, puisque nous commençons par un niveau de description très abstrait du système, le modèle ne représente pas les valeurs de données ni les détails de calcul. Nous nous focalisons sur un ensemble de paramètres liés en particulier aux communications sans prise en compte de la notion de temps. Pour cela, une représentation formelle de la structure de l'architecture avec les différents paramètres et les interconnexions suffit. Par contre, cette représentation peut bien être raffinée dans le futur vers des langages de description d'architecture comme AADL ou des bibliothèques SOCLIB[53] pour finaliser les étapes de raffinement. Nous classons ces composants selon leur fonctionnalité en quatre familles : éléments de calcul, éléments de stockage, éléments de communication et éléments d'interfaçage.

Éléments de calcul (noté PE pour Processing Elements). Ce sont des composants permettant l'exécution des instructions de calcul et des algorithmes transcrits dans un langage de programmation. On peut distinguer les processeurs et les coprocesseurs. Les processeurs exécutent des tâches diverses. Les composants coprocesseurs fournissent des fonctionnalités très spécifiques telles que par exemple la transformée de Fourier. Ils ont généralement des performances supérieures à celles d'un processeur et peuvent être reconfigurables. Un PE a plusieurs paramètres tels que le nombre et type d'interfaces, le nombre de cœurs d'exécution, la mémoire cache, la mémoire locale et le système d'exploitation utilisé. Nous nous intéressons dans notre travail aux PEs dotés de mémoire locale interne et pouvant avoir plusieurs cœurs pour l'exécution des instructions. Les paramètres considérés sont : la *taille de mémoire locale en* UNIT, le *nombre de cœurs d'exécution* et le *nombre d'interface de communication*.

- *Nombre d'interfaces du composant :* Une interface est la fenêtre d'échange et de communication d'un composant avec le monde extérieur. Un composant peut comporter une ou plusieurs interfaces. Chaque interface est définie par un nombre de signaux et un mode de communication avec les autres types de composant, comme le mode synchrone ou asynchrone.

- *Nombre de cœurs d'exécution :* Un cœur est l'unité de calcul d'un composant. Ce paramètre permet d'identifier le nombre de processus qui peuvent être exécutés en parallèle. Notons qu'un composant de calcul multi-cœurs peut partager des ressources comme la mémoire cache. Par exemple, dans les processeurs double-cœurs d'Intel, chaque cœur à son propre contrôleur mémoire et sa propre mémoire cache. Tandis que dans les processeurs double-cœurs d'AMD, cette partie est commune aux deux cœurs.

- *Mémoire locale :* Les mémoires locales sont des éléments de stockage d'information internes au composant. La taille, la vitesse d'accès à une mémoire locale influencent de manière importante le fonctionnement, les performances et le temps de calcul d'un composant.

Éléments de Stockage (noté SE pour Storage Elements). Il s'agit typiquement les mémoires volatiles. La mémoire est accédée par deux opérations : la lecture (read) et l'écriture (write). Les données sont accessibles par leur adresse ou alors selon l'ordre dans lequel elles ont été placés dans la mémoire (FIFO ou LIFO). Nous considérons les mémoires partagées comme les seuls éléments de stockage. Les paramètres pris en compte pour les mémoires sont : La *capacité de stockage* qui représente le volume global d'informations que la mémoire peut stocker et le *nombre interface de communication* qui représente le nombre de fenêtre de communication avec les autres composants. De plus, nous décomposons l'espace de stockage d'une mémoire partagée ou locale en zones de stockage disjointes pour faciliter l'identification des espaces mémoires allouées au canal lors de processus de projection, chaque zone est caractérisée par sa taille.

Éléments de Communication (noté CE pour Communication Elements). Les composants d'une architecture communiquent par l'intermédiaire des supports de communication partagés appelés "bus" ou "réseau sur puce" (NoC). Les caractéristiques essentielles des composants de communication sont : la largeur de bus, l'intervalle d'adressage, le protocole de communication, le débit de transfert et le taux d'erreurs. Dans notre travail, nous nous intéressons aux supports de transfert partagés "bus" dotés d'une politique de gestion d'accès centralisée géré par un arbitre. La section 2.1.3 de ce chapitre détaille ce type de support.

Éléments d'interfaçage (noté IF pour Interface Elements). Les éléments d'interfaçage étendent les capacités d'un système en apportant des fonctionnalités permettant la communication entre les différents composants ou l'amélioration des performances de l'architecture cible (accès à la mémoire, gestion d'interruptions, gestion de synchronisation,...etc). Ces composants sont caractérisés par : la taille de la mémoire interne, les protocoles de communication supportés ainsi que les modes de fonctionnement associé à celui-ci. Nous nous intéressons à l'introduction des interfaces de communication qui permettent l'adaptation des protocoles des composants au protocole du bus et la composition des différents bus. Les paramètres d'une interface pris en compte sont : *la politique de communication qui est décrite par une machine d'états finis* et la *taille de l'espace de stockage interne*.

Un exemple d'architecture est donné par la figure Fig. 2.3. Elle présente une architecture matérielle composée de : trois éléments de calculs PE1, PE2 et PE3 ainsi qu'une mémoire SE1. Ces éléments sont interconnectés grâce à deux éléments de communication Bus1 et Bus2 qui à leur tour sont reliés entre eux par un bridge Bridge qui représente un élément d'interfaçage. La communication sur le Bus1 est administrée par un arbitre Arbitre, le Bus2 n'a pas besoin d'un arbitre car il n'y a qu'un seul maitre de bus PE4. Nous définissions une architecture par :

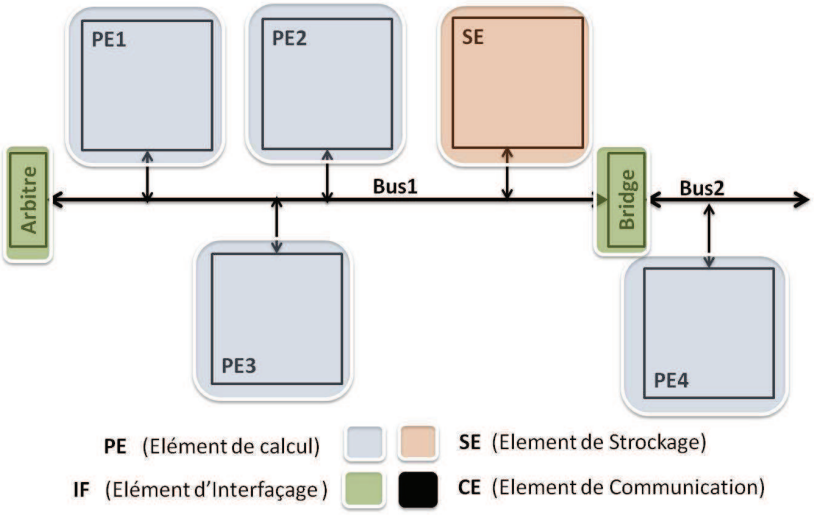


FIG. 2.3 – Un exemple de modèle d'architecture.

Définition 2 (Architecture et Chemins) *Une* architecture $\mathcal{A}$ *est un 4-tuplet tel que :*

$$\mathcal{A} = \langle \mathcal{P}, \mathcal{C}, \mathcal{S}, \mathcal{I}, \mathcal{C}hemins \rangle$$

- $\mathcal{I}$ *: un ensemble fini d'*éléments d'interfaçage*. Chaque* $I \in \mathcal{I}$ *est caractérisé par :*
  - $Cnt(I)$ *: un ensemble de* connecteurs *de l'interface* I*.*
  - $Fnt(I)$ *: une fonction qui décrit le* fonctionnement *de l'interface* I*. Ce fonctionnement est donné sous-forme d'une machine à états, la transition entre les états est réalisée selon les informations envoyées/reçues sur les connecteurs de l'interface.*

- $\mathcal{C}$ *: Un ensemble fini de* support de communication (Bus)*. Chaque* $b \in \mathcal{C}$ *est caractérisé par un ensemble de* connexion $Cnt(b)$*. Le support de communication peut être un bus hiérarchique, constitué de bus connectés entre eux à travers des interfaces bridges.*

- $\mathcal{P}$ *:Un ensemble fini* d'élémentd de calcul*. Pour chaque* $p \in \mathcal{P}$ *est associé un ensemble* $Cnt(p)$ *de* connexions *et une mémoire* m_p*.*

- $\mathcal{S}$ *: Un ensemble fini de* mémoires partagées*, chaque* $m \in \mathcal{S}$ *est caractérisée par sa taille de stockage* $size(m)$ *et un ensemble* $Cnt(m)$ *de* connexions*. Chaque mémoire est décomposée en un ensemble d'espaces disjoints, noté* $Zone(m)$*, telle que :*

$$\sum_{z \in Zone(m)} size(z) \quad = \quad size(m)$$

.

- $Chemins$ *: Un ensemble des chemins dans* $\mathcal{A}$*, est défini par :*

$$\begin{aligned} Chemins = \{(p, b, p', b', m) \in (\mathcal{P} \times \mathcal{C})^2 \times \mathcal{S} \mid\ & Cnt(p) \cap Cnt(b) \neq \emptyset \wedge \\ & Cnt(m) \cap Cnt(b) \neq \emptyset \ \wedge\ Cnt(p') \cap Cnt(b') \neq \emptyset \wedge \\ & Cnt(m) \cap Cnt(b') \neq \emptyset \ \} \end{aligned}$$

Dans cette définition, chaque composant est représenté par ses paramètres et interfaces d'échange de données, un composant est connecté au bus si l'interconnexion de l'ensemble des connexions n'est pas vide. Notons que la définition du chemin permet la communication entre deux PEs par deux différents éléments de communication b et b' reliés entre eux par une mémoire partagée, c'est-à-dire, la mémoire partagée est entre les deux éléments de communication. La spécification d'un chemin par un cinq-uplet est une conséquence du choix de cheminement de communication lors de projection, dans lequel les PEs communiquent entre eux uniquement à travers une mémoire partagée.

2.1.3 Famille de bus

Afin de présenter notre démarche de raffinement sur les médiums de communication, nous allons donner dans cette section les familles de support de communication bus ciblées. Nous détaillons les différents modes d'arbitrage et de transfert utilisés par les bus partagés.

Modes d'arbitrage

Pour gérer l'accès au bus dans le cas où plusieurs maîtres y accèdent, une politique d'accès au bus doit être définie. Il existe plusieurs modes d'arbitrage statique/dynamique, centralisé/décentralisé ou bien visible/caché.

Dans ce cas d'arbitrage **centralisé**, l'arbitrage est effectué par un seul composant, dit "Arbitre", dédié à la tâche de gestion d'accès au bus. Les communications se font sur des lignes spécialisées pour les requêtes d'accès et de libération du bus. Lors d'une demande d'accès au bus par les composants maîtres, l'arbitre doit choisir un maître selon une politique de gestion de ces composants. Tous les composants de traitement raccordés au bus possèdent un indice de priorité. La priorité des composants peut être **fixe** ou **dynamique**. Dans le premier cas, lors de cycles d'acquisition de bus, le composant maître ayant la priorité la plus élevée accède à celui-ci. Ce type d'arbitrage se caractérise par sa simplicité de mise en œuvre mais peut causer une famine. Pour répondre au problème de la famine, le second type de priorité est utilisé. Au départ, tous les composants de traitement ont une priorité fixe, un composant reçoit une priorité plus faible que la priorité actuelle à chaque fois qu'il accède au bus. Pour augmenter la performance en temps d'exécution d'une application, la politique d'**arbitrage hiérarchique** peut être utilisée, l'idée est de rassembler les maitres dans des groupes, un groupe peut appartenir à un autre groupe et ainsi de suite ; ce regroupement permet de gérer les composants et les groupes avec des politiques différentes.

Contrairement à l'arbitrage centralisé, **l'arbitrage décentralisé** est effectué par l'ensemble les composants maîtres. Dans ce type d'arbitrage des lignes de priorités sont utilisées entre les maîtres du bus. Dans le cas où un maître veut accéder au bus, il compare son niveau de priorité à celui des requêtes en cours des autres maîtres, il pourra contrôler le bus si son niveau de priorité est le plus élevé.

Le cycle de gestion d'accès au bus peut être réalisé en deux temps, avant ou pendant l'occupation du bus pour la réalisation d'un transfert. Dans le premier cas, dit **arbitrage visible**, l'arbitrage se fait dans un cycle exclusivement réservé au choix du composant maître, ce qui cause le ralentissement de débit de transfert. Dans le deuxième cas, dit

arbitrage caché, la gestion d'accès dans le cycle suivant se fait au cours des transferts des données. Alors, le chevauchement du cycle d'arbitrage avec le/les cycle(s) de transfert permet une augmentation du débit.

Modes de transfert

Le transfert des données peut être **synchrone** ou **asynchrone**. Dans le mode **synchrone**, les échanges sont cadencés par une horloge commune aux composants, ce type de mode de transfert implique très peu de logiques et il peut s'exécuter très vite. Un inconvénient de ce mode de transfert est que chaque périphérique sur le bus doit fonctionner à la même fréquence d'horloge. Contrairement au mode de transfert synchrone, le mode de transfert **asynchrone** ne requiert pas d'horloge commune entre les composants liés au bus et accepte des composants avec des vitesses variables. Il nécessite un protocole pour identifier le début et la fin des transmissions.

Comme le bus de données n'est pas utilisé dans tous les cycles de transfert, le mode de transfert en **pipeline** est mis en œuvre pour optimiser le temps de transfert. Alors qu'une donnée est en cours de transfert, l'envoi de la donnée suivante est en préparation. Autrement dit, on profite du cycle de transfert de donnée en cours, pour donner la main à un autre maître quand cela est possible et pour passer une seconde adresse. Ce schéma ne peut s'effectuer que si on utilise deux nappes de fils distincts pour véhiculer les adresses et les données.

Pour augmenter le débit d'un bus, diverses techniques de transfert peuvent être utilisées. Une des techniques est la définition de différentes tailles de transfert de données. La technique de transfert la plus utilisée est le transfert de données en **rafale (burst)**. Un mode de transfert en burst permet au bus de réaliser des transferts par paquet de données les uns après les autres. Le maître initiateur de transfert réalise le transfert des paquets de données sans interruption d'accès au bus jusqu'à la fin du transfert. Ce type de transfert est réalisé dans la plupart des cas avec l'utilisation des DMAs. De plus, un transfert de données peut être réalisé par étapes successives. On parle alors de transfert avec reprise (**split-transaction**). Un transfert de bus en mode de reprise est découpé en deux transactions. Le transfert commence normalement par la phase d'adressage, l'esclave peut répondre immédiatement, dans le cas où il est prêt. Dans le cas contraire, il note le nom du maître demandeur et libère le bus pour une allocation à un autre maître par l'arbitre. Dès que l'esclave est prêt pour une réponse, il signale ce fait à l'arbitre en activant un signal. Ce mécanisme peut donner une priorité supérieure au maître en attente, dans le cas où l'arbitre gère cette situation. Enfin, le transfert est repris par le maître adéquat.

Dans notre démarche de raffinement, nous avons choisi une spécification abstraite du bus partagé avec une politique de gestion centralisée, un composant ayant le rôle de gestion d'accès au bus "*Arbitre*" est alors spécifié. Le mode d'arbitrage réalisé sur cette abstraction est visible. Puisque la plate-forme cible peut contenir des composants ayant des interfaces non forcément compatibles avec le bus, une introduction des composants d'interfaçage "*Interface*" pour l'adaptation est réalisée. De plus, la spécification abstraite d'interface peut bien manipuler les transferts de tailles multiples.

Le choix d'avoir une représentation de bus abstrait avec un arbitre centralisé est justifié par l'avantage d'avoir un large choix de politique de gestion et la vérification facile des protocoles d'accès en transformant le fonctionnement du composant arbitre. Aussi, ce choix

d'architecture permet d'avoir une représentation abstraite commune à un large nombre de bus existants tels que, la famille des bus AMBA ou CoreConnect. Le tableau suivant montre plusieurs exemples de bus concrets ciblés par l'abstraction proposée.

	PI-Bus	PCI	AMBA ASB	AMBA AHB	CoreConnect PLB
Arbitrage centralisé	Oui	Oui	Oui	Oui	Oui
Arbitrage caché	Oui	Oui	Oui	Oui	Oui
Transfert synchrone	Oui	Oui	Oui	Oui	Oui
Transfert en pipeline	Oui	Non	Oui	Oui	Oui
Transfert en burst	Oui	Oui	Oui	Oui	Oui
Transfert avec reprise	Non	Non	Non	Oui	Oui
Transfert multiple taille	Oui	Non	Oui	Oui	Oui

FIG. 2.4 – Les caractéristiques de différents bus

Le bus PI [107] est conçu dans le cadre du programme de la Communauté européenne "ESPRIT". Le bus PI offre une large gamme de fonctions, envoi en mode burst et transferts avec multiple taille. Le bus PCI [104] créé par Intel et géré actuellement par un large groupe d'industries nommé (PCI-SIG). Le bus PCI peut fonctionner de manière synchrone ou asynchrone et permet le contrôle de flux. Ce bus utilise un seul canal de transfert d'adresses et de données. La famille des bus AMBA [7] est un standard conçu par ARM, il est largement utilisé dans les systèmes embarqués. AMBA définit une architecture de bus segmenté, où deux segments de bus sont connectés par un bridge, un segment de bus de haute performance (AHB et ASB) et un bus de basse performance (APB) acceptant un seul maître (le composant Bridge). CoreConnect est un bus développé par IBM [63]. La spécification du bus CoreConnect propose trois types de bus PLB, OPB et DCP pour l'interconnexion entre les composants. Cette architecture partage plusieurs critères de haute performance avec les bus de la famille AMBA et permet de plus d'avoir deux segments de bus qui peuvent tous les deux avoir plusieurs maîtres.

2.1.4 Projection/ Partitionnement

Une fois le modèle d'application et le modèle d'architecture définis, le processus de projection peut être réalisé pour obtenir la première plate-forme. Cette projection contraint le comportement de l'application par les paramètres de l'architecture, ce qui peut causer la non-préservation de l'ensemble des propriétés comportementales du système initial. Dans notre démarche de raffinement, nous visons le raffinement des canaux applicatifs par l'introduction progressive du support de communication partagé après processus de projection, tout en préservant les propriétés linéaires de comportement du système. Pour cela, nous avons défini quatres règles de base spécifiant le processus de projection d'une application sur une architecture :

Règle 1.

Chaque tâche est projetée sur un élément de calcul (PE), sans possibilité de migration de tâches. Si plusieurs tâches sont projetées sur un même élément de calcul (PE), celui-ci doit être doté d'un ordonnanceur (gestionnaire de priorité).

Nous choisissons de projeter une seule tâche par PE. Cela permet d'abstraire l'information d'ordonnancement sur les PEs et de se focaliser sur le raffinement de communication. Pour les canaux de communication, nous devons spécifier l'emplacement des espaces de stockage des données ainsi que le chemin d'accès à ces données par la tâche émettrice et la tâche réceptrice. Nous considérons que la communication entre deux tâches projetées sur deux différents PEs ne se fait qu'à travers une mémoire partagée, ce qui introduit la règle de projection suivante :

Règle 2.

L'espace de stockage d'un canal de données est projeté sur une zone de stockage d'une mémoire partagée.

Nous considérons également que les espaces de stockage des mémoires doivent être disjointement alloués aux différents canaux, d'où la règle suivante :

Règle 3.

Une zone de stockage de la mémoire est associée à un et un seul canal de communication.

De plus, nous imposons que les données requises (resp. produites) par l'exécution des actions de calcul soient lues (resp. écrites) sur les mémoires locales. Donc, les actions de lectures et d'écritures des tâches depuis et vers les canaux sont transformées en des actions de transferts de données vers et depuis l'espace mémoire locale de la tâche et l'espace de stockage de la mémoire partagée associée au canal à travers un support de communication.

Règle 4 (Spécification de cheminement des données)**.**

Le transfert de quantité de données entre deux tâches à travers un canal de données est transformé en un transfert de quantité de données entre la mémoire locale du l'élément de calcul PE associé à la tâche et la mémoire partagée SE associée au canal à travers un chemin constitué d'une combinaison d'éléments de communication (CE) et d'élément d'interfaçage (IF) reliant le PE à la mémoire SE.

L'existence de la mémoire locale spécifique à un PE permet de masquer tous les transferts réalisés lors du calcul, ce qui nous permet de nous focaliser exclusivement sur le raffinement des transferts dûs aux écritures et aux lectures du modèle d'application. Le masquage des transferts requis à l'exécution d'une tâche permet d'alléger le modèle du système raffiné utilisé pour la vérification de préservation de propriétés de comportement. En effet, il est toujours possible de raffiner ce modèle par le déplacement et l'intégration de l'espace mémoire locale dans une mémoire partagée ce qui fait apparaître des actions de transfert de données de calcul sur le support partagé.

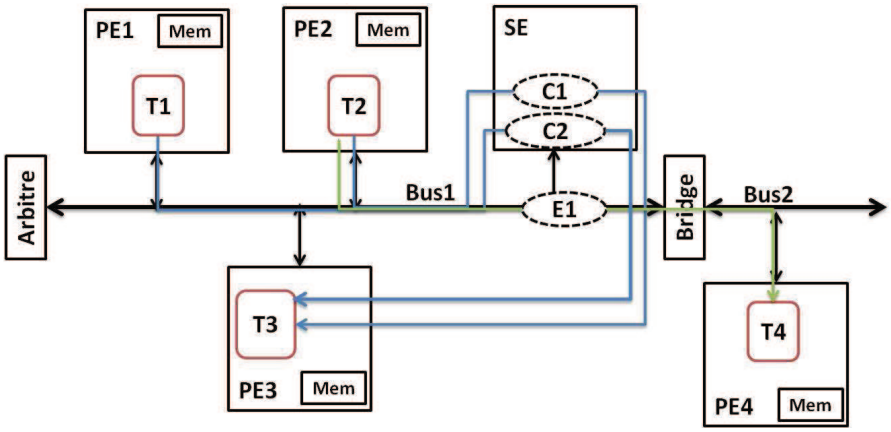


FIG. 2.5 – Projection d'application sur l'architecture.

La figure FIG. 2.5 présente un exemple de projection de l'application donnée par la figure FIG. 2.2 sur l'architecture de l'exemple donnée par la figure FIG. 2.3. Les tâches T1, T2, T3 et T4 sont projetées respectivement sur les éléments de calculs PE1, PE2, PE3, et PE4. Quant aux canaux de communication, les canaux de données C1, C2 sont projetés sur la mémoire SE1 via le BUS1, le canal d'événement E1 est projeté sur les deux bus BUS1 et BUS2 interconnectés par le BRIDGE. Nous définissons la projection par :

Définition 3 (Projection) *Soient $\mathcal{T}$ une application, $\mathcal{A}$ une architecture et soit $\mathcal{Z}$ l'ensemble des zones de stockage des mémoires partagées et des mémoires internes des éléments de calcul de l'architecture $\mathcal{A}$. Une* projection *$\mathcal{J}$ de l'application $\mathcal{T}$ sur l'architecture $\mathcal{A}$ est un 4-tuplet, tel que :*

$$\mathcal{J} = \langle \mathcal{T}, \mathcal{A}, \mathcal{J}^T, \mathcal{J}^C \rangle$$

- $\mathcal{J}^T : \mathcal{T}asks_{\mathcal{T}} \rightarrow \mathcal{P}_{\mathcal{A}}$ *est une fonction injective de projection des* tâches *sur des* éléments de calcul.

- $\mathcal{J}^C : \mathcal{C}hannels_{\mathcal{T}} \rightarrow \mathcal{C}hemins_{\mathcal{A}} \times \mathcal{Z}^3$ *est une fonction de projection des canaux sur les éléments de* communication *et de* stockage*, telle que :*

$$\forall\, c \in \mathcal{C}hannels_{\mathcal{T}}.\ \mathcal{J}^C(c) = ((p, b, p', b', m), (z, z', z'')) \Rightarrow p = \mathcal{J}^T(src(c)) \wedge p' = \mathcal{J}^T(tgt(c)) \wedge z \in Zone(m_p) \ \wedge\ z' \in Zone(m_{p'}) \wedge z'' \in Zone(m).$$

La projection d'une seule tâche sur un élément de calcul est caractérisée par la fonction $\mathcal{J}^T$. Concernant les canaux, pour tout canal est associé un chemin sur l'architecture (p, b, p', b', m) et un triplet de zones de stockage (z, z', z'') tel que, la première zone de stockage (z) est allouée dans l'espace mémoire d'élément de calcul (p) sur lequel est projeté la tâche source du canal c $(src(c))$, donc $z \in Zone(m_p)$. La seconde zone de stockage (z') est allouée dans l'espace mémoire d'élément de calcul (p') sur lequel est projetée la tâche destination du canal c $(tgt(c))$, d'où $z \in Zone(m'_p)$. Et la dernière zone de stockage (z'') est allouée dans l'espace de stockage de la mémoire partagée m, d'où $z'' \in Zone(m)$.

2.2 Raffinement de communication

Le but de notre travail est de raffiner les médiums de communication en garantissant la correction du comportement de la plate-forme résultante vis-à-vis de la plate-forme initiale. Nous considérons le modèle TML d'une l'application dans sa forme abstraite comme la plate-forme initiale sur laquelle le processus de raffinement est réalisé : chaque tâche représente un PE sans contrainte et chaque canal représente un élément de communication du même comportement. Le processus de raffinement que nous proposons commence par la phase de projection. Lors de la projection sur une architecture, le comportement des tâches est modifié en raison des contraintes architecturales des éléments de calcul et les canaux sont contraints par les protocoles des supports de communication sur lesquels seront acheminées les données. Dans notre démarche de raffinement nous utilisons principalement deux types de raffinement : le raffinement de données et le raffinement de communication. Ce choix est justifié vis-à-vis les différents types raffinements décrit dans la section 1.1.2 du chapitre 1 par :

La décomposition de fonctionnement : qui consiste principalement à créer, à partir d'une description d'un système, sa description la plus parallélisée possible. ***Or dans notre cas, nous considérons le parallélisme de l'application exprimé sur l'application*** TML***. Dans le cas où le concepteur juge que l'application peut être encore plus parallélisée, alors le modèle de l'application doit être révisé.***

Le raffinement de données : les données de traitement et de communication d'un niveau abstrait peuvent être plus grossières que celles du niveau le plus bas d'implémentation. ***Dans le cas de l'application*** TML***, nous démarrons par des granularités de données de gros grains. Pour modéliser les composants de l'application projetée, nous proposons un mécanisme de transformation des actions des données abstraites (de gros grains) en des données manipulées par l'architecture (de grains fins)***.

Le raffinement de communication : ce raffinement procède au partage des ressources physiques, bus et mémoires, par les différents canaux de données. ***Lors de la projections, les canaux*** TML ***doivent partager le même support de communication, ce partage crée le besoin d'ajout de nouvelles synchronisations pour la gestion d'accès aux ressources partagées afin de préserver le comportement du système et de gérer les accès***.

Le raffinement algorithmique : ce raffinement consiste en la génération du code, comment seront implémentées les différentes actions d'une application par l'introduction des détails de calcul et l'implémentation des différentes fonctions de synchronisation. ***En l'état actuel, notre démarche ne considère pas ce type de raffinement***.

Pour réaliser le raffinement de données et le raffinement de communication, nous utilisons les contraintes fournies par le processus de projection d'une manière progressive. Pour cela, nous décomposons le processus de projection en deux phases. La première phase de projection consiste à associer les tâches aux éléments de calculs et les canaux aux éléments de stockage sans prendre en considération l'information de cheminement des données sur le support de communication partagé. La deuxième phase de projection permet de compléter le modèle de la plate-forme en rajoutant le partage de chemin de communication par l'introduction du support de communication. En appliquant la première phase de projection, l'application est en fait raffinée en deux étapes : Raffinement de granularité de données et mise en œuvre de gestion des communications détaillées. Et en appliquant la deuxième phase de projection, l'application est encore raffinée par introduction de gestion d'un bus abstrait sur le modèle obtenu par les deux premières étapes de raffinement. Chaque étape de raffinement modifie l'application et la fait passer d'un niveau à un autre en se basant sur les détails de projection. En fait, chaque étape de raffinement est elle-même décomposée en un ensemble d'opérations de transformation des différents composants d'un niveau donné.

La figure FIG. 2.6 montre le schéma de raffinement que nous proposons et sa position dans le processus de projection. Les contraintes de la première phase de projection (projection1) permettent la réalisation du raffinement de granularité de données et de mise en œuvre de la gestion du canal pour la construction d'un niveau de description plus détaillé (NIVEAU_2) des tâches et canaux correspondant à la (plate-forme0.1). Les contraintes du support de communication de la seconde phase de projection permettent le raffinement du modèle d'application par introduction de modèle de bus abstrait et ainsi la construction de la plate-forme du niveau (NIVEAU_3) correspondant à la plate-forme (plate-forme1).

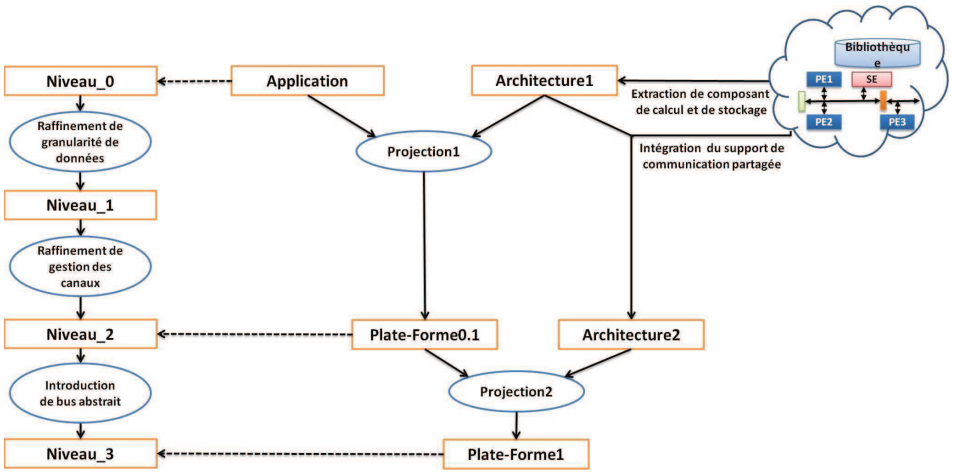


FIG. 2.6 – Schéma de raffinement de communication dans le flot de conception.

Dans les sections suivantes, nous détaillons les différentes étapes de raffinement que nous proposons et les différentes contraintes d'architecture et de projection prises en compte sur chaque étape.

2.2.1 Premier raffinement : Changement de granularité de données

Cette étape consiste à modifier la taille de l'unité de données manipulée par l'application. Nous convertissons les granularités de données du modèle abstrait en des données d'implémentation, donc le raffinement de granularité de données du gros-grain en une granularité de grain fin. Lors de la projection, la granularité de données doit être transformée en des granularités de données d'implémentation selon l'algorithme ciblé (de l'image vers des pixels par exemple) ou encore transformée en des granularités de données manipulées par le bus. Dans cette première étape de raffinement, puisque les informations concernant le support de communication est abstraites, nous nous intéressons au changement de granularité de données par ***décomposition de données*** de transfert et de calcul provenant du choix d'implémentation réalisé par le concepteur. Les actions (écriture, lecture et exécution) sur des données abstraites sont transformées en un ensemble ordonné d'actions manipulant des données de granularité plus fines, ce qui fait passer le modèle d'application (NIVEAU_0) au premier niveau (NIVEAU_1). L'ordre entre les actions construit par le raffinement de granularité de données est régi par deux contraintes principales : la dépendance de données entre les actions de la nouvelle granularité et la contrainte de partage de ressources extraite des paramètres de projection. Ces contraintes sont le partage de mémoire, le nombre des cœurs d'exécution, et le nombre d'interface de communication. Aussi, dans cette étape les canaux doivent être transformés pour gérer les nouvelles granularités. De plus, la taille des canaux est limitée selon l'espace mémoire qui leur a été alloué.

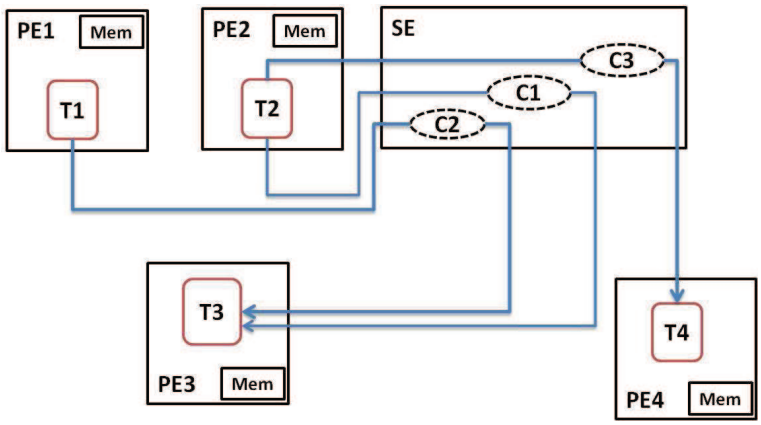


FIG. 2.7 – Schéma d'un exemple de projection.

La figure FIG. 2.7 montre un schéma d'un exemple de la première phase de projection d'une application sur une architecture. Les tâches T1, T2, T3 et T4 sont projetées respectivement sur les éléments de calculs PE1, PE2, PE3, et PE4. Les canaux de communication C1, C2, C3, sont projetés sur la mémoire SE1. Les lectures et écritures vers et depuis un canal sont réalisées entre l'espace de stockage de la mémoire SE et les mémoires locales des composants de calcul *Mem* associées aux tâches d'entrée et de sortie du canal. Dans cette phase, nous avons considéré uniquement la projection des tâches sur les PEs (**Règle1**) et les espaces de stockage des canaux sur les espaces mémoires (**Règle 2**) et aucune information sur le support de communication partagé (bus) n'est introduite.

Contrainte de dépendance de données

Une dépendance de données [88] existe entre deux actions a_1 et a_2, si a_2 utilise les données produites par l'action a_1. Dans le cas de partage de mémoire, deux actions a_1 et a_2 sont en relation de dépendance directe si a_1 écrit sur une cellule-mémoire qui est plus tard lue par a_2 et il n'y a pas une autre instruction a_3 qui affecte cette cellule-mémoire après a_1 et avant a_2.

Lors du raffinement de granularité de données, nous nous basons sur la relation de dépendance de données entre les actions pour la construction d'ordre entre les nouvelles actions des tâches manipulant les nouvelles granularités des données. En TML la notion de dépendance de données n'est pas explicite, la relation entre les actions existe par un ordre entre les différentes actions données par le code séquentiel des tâches. Dans notre travail, l'information de dépendance de données est extraite essentiellement de l'algorithme d'implémentation visé ou fournie par le concepteur. La spécification de la dépendance de données permet de déterminer le moment d'utilisation des données et la disponibilité de l'espace mémoire lors de l'exécution d'une tâche (appelé la persistance de données).

Contrainte de persistance et disponibilité des données

La ***persistance*** d'une donnée [88] est caractérisée par la durée de présence de la donnée dans un espace mémoire. Elle est déterminée par l'opération d'écriture de la donnée sur l'espace mémoire et de son effacement de cet espace par une réécriture sur le même espace mémoire. Dans cette période de persistance la donnée est toujours **disponible**. Nous utilisons la notion de persistance pour étudier la possibilité d'exécution d'une tâche selon les ressources disponibles de la mémoire locale. Nous construisons l'ordre entre les actions qui respecte cette notion de telle manière que l'espace occupé par une donnée produite par une action ne soit libérée qu'après que cette donnée ait été utilisée par toutes les actions qui en dépendent. Dans le cas où aucun ordre d'exécution de comportement d'une tâche n'est possible, le processus de projection et de partitionnement des ressources doit être réétudié afin de proposer un nouveau partitionnement ou dimensionnement des ressources qui permet l'exécution des tâches de l'application.

Contrainte de structure interne d'un élément de calcul

La structure interne des composants de calcul entraîne des contraintes supplémentaires sur l'ordre entre les actions des tâches. Avec l'abstraction de la structure interne des PEs, le nombre d'actions permises en parallèle est au maximum égal au nombre de ressources disponibles soit pour le calcul (cœurs) ou pour le transfert (interfaces). Dans le cas d'un mono-cœur par exemple, les actions d'une tâche doivent être totalement ordonnées.

2.2.2 Second raffinement : Gestion des canaux

Dans cette étape de raffinement, aucune nouvelle contrainte de projection n'est introduite. Le but de cette étape est de s'assurer que l'espace mémoire alloué pour un canal est contrôlé en respectant fidèlement son comportement initial. Pour cela, nous proposons

la transformation d'un canal par décomposition de son comportement abstrait en deux types de communication : des accès aux espaces de données et des synchronisations pour l'accès à cet espace. Nous avons principalement transformé le comportement des canaux et tâches pour rendre explicite la politique de gestion des canaux de type BR-BW. Une action d'écriture/lecture sur une donnée d'un canal BR-BW dans la mémoire se transforme en des actions de transfert et des actions de synchronisation, de vérification de l'existence de données ou d'espace libre plus de l'acquittement de la disponibilité de données ou d'espace. Après la transformation de chaque action, une étape d'ordonnancement des nouvelles actions dans les tâches est réalisée selon les contraintes architecturales de la projection (*projection 1* de la figure FIG. 2.7).

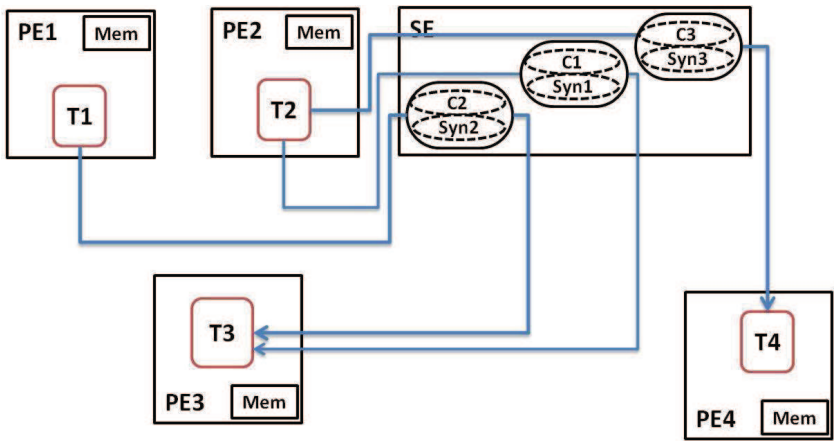


FIG. 2.8 – Schéma d'un exemple de la plate-forme 0.1 .

La figure FIG. 2.8 présente une plate-forme obtenue par l'application de la seconde étape de raffinement sur l'exemple de la figure FIG. 2.7. Notons dans cette plate-forme la non-modification de l'architecture, aucune information sur le support de communication n'est ajoutée.

2.2.3 Troisième raffinement : Introduction de bus abstrait

Dans cette dernière étape de raffinement, le processus de projection est totalement effectué. Les contraintes de projection liées au support de communication partagé sont introduites, telles que : la structure hiérarchique de bus, les tailles des bus de données et les modes de transfert pris en compte. Dans cette étape, nous décrivons d'abord un protocole abstrait du bus par la définition du comportement abstrait des éléments d'interfaçage (arbitre, interface et bridge). Ensuite, nous introduisons des protocoles de synchronisation pour résoudre les problèmes d'accès au support partagé, pour bloquer et débloquer des tâches suivant certaines conditions. En parallèle à l'introduction des synchronisations, nous réalisons le *changement de granularité de données de transfert pour se ramener à la taille du bus de données choisi*. En appliquant ces transformations, nous obtenons le NIVEAU_3 correspondant à la *plate-forme1* de la figure FIG. 2.6.

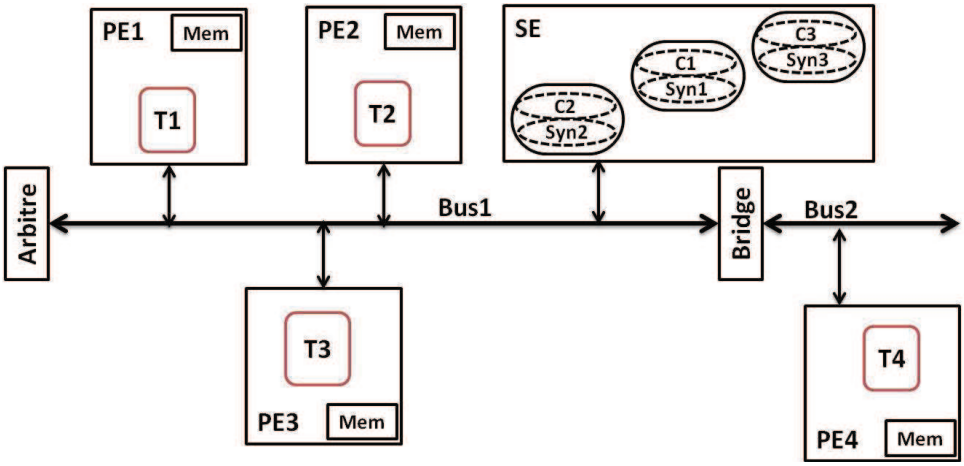


FIG. 2.9 – Schéma d'un exemple de la plate-forme 1.

Une plate-forme possible obtenue après la seconde phase de projection (*Projection2* de la figure FIG. 2.7) est représentée par la figure FIG. 2.9. Notons l'introduction d'un support de communication partagé composé de deux bus *Bus1* et *Bus2* connectés entre eux par le *Bridge*. La gestion d'accès au bus (BUS1) est assurée par le composant *Arbitre*. Les différents composants doivent des interfaces de communication compatibles au protocole de bus. Dans le cas contraire, des interfaces peuvent être spécifiées pour l'adaptation de protocoles des composants au protocole de bus.

Ce niveau peut bien être raffiné pour atteindre des niveaux d'entrée des outils de synthèse. Pour cela, le processus de raffinement peut être poursuivi par : l'introduction des nouvelles contraintes d'architecture telles que la structure et des paramètres des éléments de calcul, le choix des paramètres liés aux modes de transfert, la concrétisation des valeurs de données et des détails de calcul. À ce stade, nous avons proposé de réaliser le processus de projection par des règles de transformation bien définies permettant le passage du niveau le plus abstrait NIVEAU_0 au niveau NIVEAU_3, le but de ce travail de thèse est de vérifier que les propriétés linéaires vraies au NIVEAU_0 sont également vraies sur la plate-forme du NIVEAU_3.

2.2.4 Résumé des étapes de raffinement

Chaque étape de raffinement est elle-même décomposée en un ensemble d'opérations de transformation des différents composants d'un niveau donné. Le choix des transformations et le choix de l'ordre de leur application sont réalisés afin de gérer la complexité du système, d'optimiser le processus de raffinement et d'établir une séquence d'étapes qui assure la préservation des propriétés.

Naturellement, raffiner la granularité des données avant l'introduction des synchronisations sur les canaux et l'accès au support de communication est plus simple. Après l'étape du raffinement de granularité de données et la mise en œuvre de la gestion de communication, toutes les actions de communication liées au modèle du système sont définies. Dans ce cas, l'étape de cheminement peut être spécifiée pour l'ensemble des actions avec la gra-

nularité manipulée. Notons que dans le cas où la largeur du bus de données est inférieure à la granularité de donnée manipulée, une décomposition d'un transfert de données en plusieurs transferts doit être réalisée. Dans le cas contraire, c'est-à-dire, si le raffinement de granularité de données est fait après le processus de projection et d'introduction de support de communication partagée, le choix d'implémentation peut ne pas être optimal, à cause d'un côté des granularités de gros grains de données qui restreint les choix de projection possibles et d'un autre côté de la complexité du modèle lors de processus de décomposition des données abstraites en des données de grains plus fines.

2.3 Conclusion

Dans ce chapitre, nous avons présenté l'approche proposée pour le raffinement d'application, basée sur l'introduction de détails architecturaux lors de la projection et plus précisément sur les médiums de communication. Nous avons également introduit les différentes notions utilisées dans la méthodologie de conception : le modèle TML, la notion d'architecture, de projection et les contraintes de projection prises en compte.

Notre démarche de raffinement se base sur une décomposition de processus de projection en deux phases. Une phase de projection des tâches qui introduit le raffinement de granularité des données manipulées et la mise en œuvre de la gestion des communications dû au comportement des éléments de stockage et une phase d'introduction de support de communication partagé qui permet d'introduire une description abstraite de bus. Le résultat de ces transformations est un modèle abstrait de la plate-forme obtenue après la projection.

Cette démarche permet de décomposer le processus de projection en des étapes de raffinement bien définies et permet ainsi de cadrer et de systématiser les transformations du système réalisées par le concepteur. L'apport est la formalisation de cette démarche dans le but de prouver la préservation des propriétés comportementales du niveau abstrait au niveau concert. Dans notre travail, nous avons réalisé ce processus de raffinement dans un cadre mathématique en manipulant des relations d'ordre entre les actions exécutées par les applications.

Le chapitre suivant décrit les modèles mathématiques utilisés pour la mise en œuvre des transformations de raffinement pour établir la preuve de raffinement et de préservation de propriétés. Le chapitre décrit aussi la sémantique de raffinement sur laquelle nous nous basons pour établir les preuves.

Chapitre 3

Formalisme

Sommaire

Ce chapitre présente les bases théoriques sur lesquelles nous nous appuyons pour la description et la réalisation des étapes de raffinement, et pour établir la vérification de la préservation de comportement (préservation des propriétés fonctionnelles) entre les différents niveaux de raffinement.

Tout au long de processus de conception, nous étudions l'ordre d'exécution des différentes actions de chaque tâche et de l'application globale. Cette étude permet de construire un modèle de l'application projetée sur une architecture conforme aux contraintes liées à l'implémentation et aux paramètres de l'architecture. Pour décrire et manipuler les modèles des applications et des architectures, nous utilisons le formalisme d'ensemble ordonné.

Afin d'étudier et de vérifier la préservation des propriétés lors du raffinement, nous nous basons sur les sémantiques de raffinement décrites sur les algèbres de processus et les machines à états finis. Les résultats théoriques sur ces formalismes permettent de prouver algorithmiquement le raffinement par un parcours du graphe d'accessibilité des modèles correspondant aux différents niveaux de représentation d'un système.

3.1 Pomset

Un ordre partiel étiqueté, noté ***lpo*** (de l'anglais : labelled partial order) [54] est un ensemble fini d'événements ordonnés qui sont étiquetés par des lettres d'un alphabet. Cette notion formalise la notion intuitive d'arrangement entre les éléments d'un ensemble.

Définition 4 (Ordre partiel étiqueté)
Un ordre partiel étiqueté est un 4-tuple $\langle V, A, \preceq, \mu \rangle$ *avec*

- V *un ensemble fini d'événements.*
- A *un ensemble fini d'actions.*
- $\preceq \subseteq V \times V$ *une relation d'ordre sur l'ensemble* V*.*
- $\mu : V \rightarrow A$ *une fonction d'étiquetage surjectif attribuant une action à un événement.*

Un ordre est une relation binaire réflexive, transitive et antisymétrique. La restriction de la relation d'ordre aux couples d'éléments distincts est dite relation d'***ordre strict***, c'est le cas d'une relation irréflexive. Dans la suite nous utilisons essentiellement la relation d'ordre strict entre les événements notée $\prec$. Deux ***lpo***s sont dits isomorphes s'il existe une bijection entre les événements des deux ***lpo***s qui préserve l'ordre et l'étiquetage des événements.

Définition 5 (Isomorphisme des *lpo*s)
Deux ordres partiels étiquetés $p_1 = \langle V_1, A_1, \prec_1, \mu_1 \rangle$ *et* $p_2 = \langle V_2, A_2, \prec_2, \mu_2 \rangle$ *sont isomorphes s'il existe une bijection* $\mathcal{B} : V_1 \rightarrow V_2$ *préservant :*

- *l'ordre :* $(v \prec_1 v') \Leftrightarrow (\mathcal{B}(v) \prec_2 \mathcal{B}(v'))$.
- *l'étiquetage :* $\mu_1(v) = \mu_2(\mathcal{B}(v))$.

La classe isomorphe d'un ordre partiel étiqueté $p = \langle V, A, \prec, \mu \rangle$ est appelée un ***pomset*** (de l'anglais : partially ordered multiset) [94], notée par $[V, A, \prec, \mu]$. Dans la suite nous utilisons la notion de pomset pour désigner indifféremment n'importe quel ***lpo*** de sa classe isomorphe. Nous appelons un pomset ***singleton*** un pomset contenant un seul événement $([\{v\}, \{a\}, \emptyset, (v, a)])$ et nous notons par ε un pomset vide $[\emptyset, \emptyset, \emptyset, \emptyset]$. Les pomsets sont quelques fois représentés par des graphes. Par exemple, le pomset suivant :

$$p = [\{v_1, v_2, v_3, v_4\}, \{a, b, c, d\}, \{(v_1, v_2), (v_1, v_3), (v_1, v_4), (v_2, v_3), (v_2, v_4)\}, \{(v_1, a), (v_2, b), (v_3, c), (v_4, d)\}]$$

est représenté graphiquement par :

$$\begin{array}{ccccc} a_1 & \longrightarrow & b_2 & \longrightarrow & c_3 \\ & & \downarrow & & \\ & & d_4 & & \end{array}$$

Les nœuds du graphe sont les événements du pomset représentées directement par les actions de l'alphabet et indexées par l'événement correspondant (données par la fonction μ) et les flèches représentent la relation d'ordre entre les événements. Par exemple, $(v_1, v_2) \in \prec$ avec $\mu(v_1) = a$ et $\mu(v_2) = b$ est noté $a_1 \longrightarrow b_2$. Par souci de lisibilité, la relation de transitivité entre les nœuds n'est pas représentée et par la suite quand il n'y a pas d'ambiguïté, nous omettons les indices des événements.

3.1.1 Algèbre de Pomset

La théorie des pomsets est dotée d'une algèbre qui permet de construire des pomsets complexes à partir des pomsets de base. L'algèbre est décrite par des opérations. Nous donnons quelques opérateurs et définitions techniques, qui nous seront utiles par la suite.

Définition 6 (Concurrence)
Soient $p = [V, A, \prec, \mu], p' = [V', A', \prec', \mu']$ deux pomsets. La concurrence des deux pomsets, notée $p \parallel p'$, est définie par $p \parallel p' = [V \uplus V', A \cup A', \prec \cup \prec', \mu \cup \mu']$.

La concurrence de deux pomsets est un pomset dont l'ensemble des événements est l'union disjointe des ensembles des événements des opérateurs, c'est à dire si l'intersection des ensembles des événements des opérateurs est vide, l'ensemble des événements sera l'union des deux ensembles ; sinon des copies différentes des éléments de l'intersection sont créées. Pour l'alphabet, la relation d'ordre et la fonction d'étiquetage le résultat est l'union des ceux des deux opérandes. La composition de deux pomsets par concurrence laisse libres les deux pomsets opérandes.

Exemple 1 *Reprenons le pomset p de l'exemple précédent. Considérons le pomset p' suivant :*

$$p' \;=\; \begin{array}{ccc} e & \longrightarrow & f \\ & \searrow & \\ & & g \end{array}$$

La concurrence des deux pomsets p et p' donnera :

$$\begin{array}{ccccccc} a \longrightarrow b \longrightarrow c & \parallel & e \longrightarrow f & = & a \longrightarrow b \longrightarrow c & & e \longrightarrow f \\ \quad\searrow & & \quad\searrow & & \quad\searrow & & \quad\searrow \\ \quad d & & \quad g & & \quad d & & \quad g \end{array}$$

Définition 7 (Concaténation)
Soient $p = [V, A, \prec, \mu], p' = [V', A', \prec', \mu']$ deux pomsets. La concaténation des deux pomsets, notée $p; p'$, est définie par $p; p' = [V \uplus V', A \cup A', \prec \cup \prec' \cup (V \times V'), \mu \cup \mu']$.

La concaténation de deux pomsets est un pomset construit par l'extension du premier opérande par les événements et les ordres du second opérande.

Exemple 2 *Reprenons les pomsets p et p' précédents, l'opération de concaténation donnera :*

$$\begin{array}{ccccccc} a \longrightarrow b \longrightarrow c & ; & e \longrightarrow f & = & a \longrightarrow b \longrightarrow c \longrightarrow e \longrightarrow f \\ \quad\searrow & & \quad\searrow & & \quad\searrow \quad\nearrow \quad\searrow \\ \quad d & & \quad g & & \quad d \qquad\quad g \end{array}$$

Les opérations de concurrence et de concaténation des pomsets sont associatives. Le pomset vide ε est l'élément neutre pour ces opérations. La concaténation est une opération prioritaire sur l'opération de concurrence. Ces opérations peuvent être itérées sur un pomset :

- ***Répétition de la concurrence***
  $p^{|n}$ exprime une concurrence de n répétitions de copies de p :
  $p^{|0} = \varepsilon$ et $p^{|n} = p \parallel p^{|n-1}$
- ***Répétition de la concaténation***
  p^n exprime une concaténation de n répétitions de copies de p :
  $p^0 = \varepsilon$ et $p^n = p; p^{n-1}$

Une autre opération utile sur les pomsets est la substitution. La substitution d'un pomset permet de remplacer chaque événement du pomset par un pomset donné.

Définition 8 (Substitution des événements)
Soit $p = [V, A, \prec, \mu]$ un pomset. La substitution des événements est une fonction f qui associe chaque événement $v \in V$ à un pomset noté $[V_{f(v)}, A_{f(v)}, \prec_{f(v)}, \mu_{f(v)}]$.
Le pomset obtenu par substitution des événements est le pomset, noté $p[f]$, défini par $p[f] = [V_f, A_f, \prec_f, \mu_f]$, tel que :

- $V_f = \uplus_{v \in V}\ V_{f(v)}$
- $\mu_f = \uplus_{v \in V}\ \mu_{f(v)}$
- $A_f = \uplus_{v \in V}\ A_{f(v)}$
- $\prec_f = (\bigcup_{v \in V}\ \prec_{f(v)})\ \cup\ (\bigcup_{v_1, v_2 \in V}\ V_{f(v_1)} \times V_{f(v_2)})$ *avec* $(v_1, v_2)\ \in \prec$

La substition de deux événements disjoints $v_0, v_1 \in V$, donne deux ensembles $V_{f(v_0)}$ et $V_{f(v_1)}$ qui sont également disjoints. Un exemple de substitution d'événement de pomset est donné ci-après :

Exemple 3 *Reprenons les pomsets p et p'. Considérons f la fonction qui substitue un événement v étiqueté par l'action a du pomset p par le pomset p' défini par* $f(v) = \begin{cases} p' & si \quad \mu(v) = a \\ v & si \quad \mu(v) \neq a \end{cases}$
Le pomset obtenu par cette opération est le pomset suivant :

$$p[f] = e \longrightarrow f \longrightarrow b \longrightarrow c \quad \begin{matrix} \searrow & \nearrow & \searrow \\ & g & & d \end{matrix}$$

Puisque l'événement étiqueté a et le prédécesseur de tous les événements de pomset. Après substitution, tous les événements du pomset p' sont des prédécesseurs des événements de pomset p.

Une autre notion importante sur les pomsets est la notion de ***processus*** qui permet de représenter l'ensemble de comportement possible d'un système représenté par un pomset.

Définition 9 (Processus) *Un **processus** est un ensemble de pomsets.*

La plupart des définitions relatives aux pomsets peuvent être étendues aux processus. Puisqu'un processus est un ensemble de pomsets, les opérations ensemblistes peuvent être utilisées pour composer les processus. Si P et Q sont deux processus, $P \cup Q$ donne l'union (le choix) des comportements de P et de Q, $P \cap Q$ représente les comportements communs des deux processus, et $P \setminus Q$ représente les comportements propres à P qui ne sont pas dans Q. Les notations d'itérations des pomsets sont aussi étendues par des ensembles infinis d'itérations finies de pomset en utilisant cette notion de processus. Soit p un pomset :

- ***Répétition non bornée de la concurrence***
  $p^{\dagger}$ exprime une concurrence de plusieurs répétitions finies de copies de p, tel que : $p^{\dagger} = \{p^{|n} \mid n \in \mathbb{N}\}$
- ***Répétition non bornée de la concaténation***
  p^{*} exprime une concaténation de plusieurs répétitions finies de copies de p, tel que : $p^{*} = \{p^{n} \mid n \in \mathbb{N}\}$

3.1.2 Extension sur l'algèbre des pomsets

Pour utiliser le formalisme *Pomset* dans notre démarche de raffinement, nous avons proposé des nouvelles définitions sur la substitution des pomset, le renforcement d'ordre sur les pomsets et des définitions pour le parcours d'un pomset. Nous avons aussi défini des notations d'imbrication de pomsets pour pouvoir décrire le comportement des canaux de communication d'une manière simple et compacte.

Définissons d'abord deux ensembles d'événements particuliers d'un pomset. Les événements d'ordre supérieur sont les derniers événements dans la séquence d'un ordre. Notons $Sup_p(V)$ l'ensemble des événements ***supérieurs*** d'un pomset $p = [V, A, \prec, \mu]$ défini par :

$$Sup_p(V) = \{v \in V \mid \nexists v' \in V,\, v \prec v'\}$$

De la même manière nous définissons l'ensemble des événements ***inférieurs*** $Inf_p(V)$, l'ensemble des événements qui n'ont aucun prédécesseur sur le pomset p, défini par :

$$Inf_p(V) = \{v \in V \mid \nexists v' \in V,\, v' \prec v\}$$

Nous introduisons une nouvelle substitution qui ne force pas l'ordre entre tous les événements des pomsets opérandes mais qui ordonne seulement les événements d'ordre supérieur des deux pomsets. L'idée est qu'un événement est associé à un pomset, l'ordre des événements du pomset de départ ne représente que l'ordre entre les derniers événements des pomsets de la substitution.

Définition 10 (Substitution par des événements supérieurs)
Soit $p = [V, A, \prec, \mu]$ un pomset. La substitution par concaténation des événements supérieurs est une fonction f qui associe chaque événement $v \in V$ de p d'un pomset, noté $f(v) = [V_{f(v)}, A_{f(v)}, \prec_{f(v)}, \mu_{f(v)}]$. Le nouveau pomset obtenu par substitution des événements est le pomset $p^{[f]} = [V_f, A_f, \prec_f, \mu_f]$, tel que :

- $V_f = \uplus_{v \in V}\ V_{f(v)}$
- $\mu_f = \uplus_{v \in V}\ \mu_{f(v)}$
- $A_f = \uplus_{v \in V}\ A_{f(v)}$
- $\prec_f = (\bigcup_{(v \in V)}\ \prec_{f(v)}) \cup (\bigcup_{(v_1, v_2 \in V)} (Sup_{f(v_1)}(V_{f(v81)}) \times (Sup_{f(v_2)}(V_{f(v_2)})))$ *avec* $(v_1, v_2) \in \prec$

Notons que cette définition diffère principalement de la définition 3.1.2 par la relation d'ordre. Celle-ci est reportée sur les événements supérieurs des pomsets associés.

Exemple 4 *Considérons le pomset p des exemples précédents et le pomset p'' suivant :*

$$p'' \;=\; b_1 \longrightarrow b_2 \longrightarrow b_3$$

Considérons f' la fonction de substitution des événements supérieurs qui substitue un événement v étiqueté par l'action b du pomset p par le pomset p'' définie par $f'(v)=\begin{cases} p'' & si \ \ \mu(v) = b \\ v & si \ \ \mu(v) \neq b \end{cases}$
Le pomset obtenu par cette opération est le pomset suivant :

$$p[f'] = b_1 \longrightarrow b_2 \longrightarrow \begin{array}{c} a \\ \downarrow \\ b_3 \\ \downarrow \\ d \end{array} \longrightarrow c$$

L'opération de substitution peut être également définie sur les actions. Il s'agit alors d'adapter la substitution d'événements à la substitution des actions associées aux événements.

Définition 11 (Substitution d'actions par événements supérieurs)
Soit $p = [V, A, \prec, \mu]$ un pomset. La substitution d'action est une fonction f' qui associe à chaque action $a \in A$ de p un pomset $f'(a) = [V_{f'(a)}, A_{f'(a)}, \prec_{f'(a)}, \mu_{f'(a)}]$. Le pomset obtenu par substitution d'actions est le pomset $p^{[f' \circ \mu]}$.

En effet, on a la fonction d'étiquetage $\mu : V \rightarrow A$ et $f' : A \rightarrow \mathbb{P}$ une fonction de transformation d'actions par un ensemble de pomsets, noté $\mathbb{P}$. La composée des deux fonctions est une fonction définie pour chaque événement.

$$\forall v \in V.\ f' \circ \mu(v) = f'(\mu(v))$$

On retrouve donc la substitution d'événements. Une telle substitution relie les événements étiquetés par la même action à un même pomset. Elle permet la substitution directe des actions du pomset p par des nouvelles actions en projetant l'ordre des évènements liés aux actions du pomset p sur un ordre entre les derniers événements étiquetés des pomsets de la substitution. La substitution d'action est aussi définie sur la substitution des événements de la définition par la composition $p[f' \circ \mu]$.

Afin de pouvoir ajouter des contraintes d'ordre à un pomset, par exemple pour restreindre un ordre, une notion de renforcement d'ordre est introduite. Notons par R^+ la *clôture transitive* d'une relation R, celle-ci est définie comme étant est la plus petite relation transitive contenant cette relation, c'est à dire $R \subseteq R^+$ et R^+ est transitive. Nous définissons la fonction du renforcement d'un pomset comme suit :

Définition 12 (Renforcement d'un pomset)
Soient $p = [V, A, \prec, \mu]$ un pomset et $R \subseteq V \times V$ une relation binaire. Un renforcement d'un pomset p par la relation R, noté $p \lhd R$, est défini si la clôture transitive $(\prec \cup R)^+$ est un ordre partiel strict par $[V, A, (\prec \cup R)^+, \mu]$.

L'opération de renforcement introduit explicitement des entrelacements entre les événements des pomsets par la relation R. Cette opération n'est possible que si la fermeture transitive de R et de l'ordre du pomset forment un ordre partiel.

En plus des opérateurs introduits jusqu'ici, nous proposons une notation d'écriture des configurations des pomsets et des processus d'une manière imbriquée.

Définition 13 (Imbrications concaténée)
Soient p, q deux pomsets et $n, m \in \mathbb{N}$. Nous définissons par récursion un opérateur d'imbrication sur la concaténation noté $\overset{n;m}{\bowtie}$ par :

$$(p \overset{n;m}{\bowtie} q) = \begin{cases} \varepsilon & si \quad n = 0 \ \vee \ m = 0 \\ \underbrace{(p; (p; \ldots; (p; q)^m; \ldots; q)^m; q)^m}_{n \ fois} & si \quad n \neq 0 \ \wedge \ m \neq 0 \end{cases}$$

Nous utilisons cet opérateur pour décrire les files d'attente des canaux TML. Les deux pomsets p et q représentent les accès à une file d'attente, sa taille est spécifiée par le paramètre de profondeur de l'imbrication n et le nombre de répétition d'opérations d'accès à celle-ci est représenté par le paramètre m. Cette définition peut être également étendue sur des opérations de concurrence et les processus pour décrire les exécutions infinies.

Nous définissons également la notion d'*idéal* d'un pomset dans le but de parcourir celui-ci. Un idéal est un sous-ensemble d'événements dont tous les prédécesseurs de chaque événement de cet ensemble sont aussi dans cet ensemble.

Définition 14 (Idéal) *Soit $p = [V, A, \prec, \mu]$ un pomset. Un idéal de p est un ensemble $I \subseteq V$, tel que : $\prec \cap (I \times I) = \prec \cap (V \times I)$.*

Cette définition permet de décrire un état de parcours d'un pomset à un instant donné. Elle permet de définir l'ensemble d'événements franchissables. Les événements franchissables d'un pomset sont les événements successeurs directs de tous les événements d'un idéal de ce pomset. Nous notons $\mathcal{I}_p$ l'ensemble des idéaux d'un pomset p.

Définition 15 (Événements franchissables)
Soit $p = [V, A, \prec, \mu]$ un pomset. L'ensemble des événements franchissables à partir d'un idéal I du pomset p noté $Access_p(I)$ est défini par :

$$Access_p(I) = Inf_p(V - I)$$

En effet, cette définition permet de définir l'ensemble des événements qui peuvent être exécutés dans le prochain instant d'exécution. Ces événements franchissables sans des événements qui s'exécutent en parallèle. La figure FIG. 3.1 montre cette notion d'événements franchissables à partir d'un idéal donné I sur un ensemble d'événements d'un pomset p.

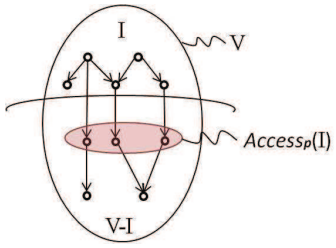


FIG. 3.1 – Exemple d'ensemble franchissable

En fait, l'union d'un idéal I du pomset p avec chaque sous-ensemble de l'ensemble d'événements franchissable est aussi un idéal de ce pomset.

$$\forall\ I \in \mathcal{I}_p.\ I \cup \mathcal{P}(Acces_p(I))\ \in\ \mathcal{I}_p$$

Cette propriété est une conséquence directe de la définition d'un idéal car les prédécesseurs de chaque événement de l'ensemble des événements franchissables sont déjà dans I.

3.2 Sémantique TML

À présent, nous donnons la sémantique d'un noyau de TML dans le langage des pomsets. Cette sémantique nous permet de générer automatiquement des pomsets à partir d'un code TML. Le noyau du langage TML considéré (voir table TAB. 3.1) est constitué d'opérations de base pour écrire une application TML. Il est composé d'opérations de communication et des opérations d'exécution utilisées pour coder les applications. Les communications entre les tâches se font par des canaux de transfert de données (channel).

⟨*task*⟩	:=	***task*** ⟨*name_task*⟩ {⟨*variables*⟩$^+$; ⟨*inst_bloc*⟩}
⟨*inst_bloc*⟩	:=	⟨*inst*⟩ | ⟨*inst*⟩; ⟨*inst_bloc*⟩
⟨*inst*⟩	:=	⟨*inst_atomic*⟩ | ⟨*inst_cond*⟩ | ⟨*inst_loop*⟩
⟨*inst_atomic*⟩	:=	***read*** ⟨*num*⟩ ⟨*name_ch*⟩
	|	***write*** ⟨*num*⟩ ⟨*name_ch*⟩
	|	***exec*** ⟨*number_of_iterations*⟩
⟨*inst_loop*⟩	:=	***repeat*** ⟨*number_of_iterations*⟩ ***times*** ⟨*inst_bloc*⟩ ***endrepeat***
⟨*inst_cond*⟩	:=	***if*** ⟨*condition*⟩ ***then*** ⟨*inst_bloc*⟩ ***endif***
	|	***if*** ⟨*condition*⟩ ***then*** ⟨*inst_bloc*⟩ ***else*** ⟨*inst_bloc*⟩ ***endif***
⟨*channel*⟩	:=	***channel*** ⟨*name*⟩⟨*name_type*⟩⟨*ch_type*⟩⟨*width*⟩⟨*param*⟩
⟨*param*⟩	:=	⟨*name*⟩ ⟨*name*⟩
⟨*ch_type*⟩	:=	*BR-NBW*
	|	*NBR-NBW*
	|	*BR-BW*

TAB. 3.1 – Noyau TML considéré

Chaque composant de l'application(tâche et canal) est traduit en un processus pomset. Cette traduction est donnée par les fonctions sémantiques $[\![.]\!]_{task}$, $[\![.]\!]_{inst}$, $[\![.]\!]_{inst\_bloc}$, $[\![.]\!]_{inst\_loop}$, $[\![.]\!]_{inst\_cond}$, $[\![.]\!]_{inst\_atom}$, $[\![.]\!]_{ch\_type}$, et $[\![.]\!]_{channel}$ définies sur la syntaxe du code TML et qui sont données dans les tableaux (TAB. 3.2 et TAB. 3.3).

3.2.1 Sémantique d'une tâche

Le code d'une tâche comporte un nombre fini d'instructions de lecture, écriture et d'exécution qui sont exécutées infiniment. Le modèle d'une tâche TML est un ensemble infini de pomset représentant le bloc d'instructions d'une tâche. Chaque instruction de la tâche

est associée à un pomset singleton. La séquence d'exécution des instructions dans le code est traduite par la concaténation des instructions. Une boucle de n itérations est représentée par une répétition concaténée de taille n du pomset du bloc d'instructions contenu dans la boucle. Les blocs d'instruction sous une condition sont interprétés par l'union de deux pomsets représentant les deux blocs de choix. Si les instructions de condition ne contiennent pas la deuxième alternative du choix (*else*), alors cette branche est interprétée par le pomset vide ε.

$$
\begin{array}{lcll}
[\![\alpha]\!]_{task} & = & [\![\langle inst\_bloc\rangle]\!]^{*}_{inst\_bloc} & \text{si } \alpha = \langle task\rangle \\
[\![\alpha]\!]_{inst} & = & \left\{ \begin{array}{l} [\![\alpha]\!]_{inst\_atom} \\ [\![\alpha]\!]_{inst\_cond} \\ [\![\alpha]\!]_{inst\_loop} \end{array} \right. & \begin{array}{l} \text{si } \alpha = \langle int\_atomic\rangle \\ \text{si } \alpha = \langle int\_cond\rangle \\ \text{si } \alpha = \langle int\_loop\rangle \end{array} \\
[\![\alpha]\!]_{inst\_bloc} & = & \left\{ \begin{array}{l} [\![\alpha]\!]_{inst} \\ [\![inst]\!]_{inst}; [\![\alpha\, inst\_bloc]\!]_{inst\_bloc} \end{array} \right. & \begin{array}{l} \text{si } \alpha = \langle inst\rangle \\ \text{si } \alpha = \langle inst\rangle; \langle inst\_bloc\rangle \end{array} \\
[\![\alpha]\!]_{int\_loop} & = & [\![\langle inst\_bloc\rangle]\!]^{n}_{inst\_bloc} & \begin{array}{l} \text{si } \alpha = \boldsymbol{repeat}\ n\ \boldsymbol{times} \\ \langle inst\_bloc\rangle\ \boldsymbol{endrepeat} \end{array} \\
[\![\alpha]\!]_{int\_cond} & = & \left\{ \begin{array}{l} [\![\langle inst\_bloc\rangle]\!]_{inst\_bloc} \cup \varepsilon \\ [\![\langle inst\_bloc\rangle_1]\!]_{inst\_bloc} \cup [\![\langle inst\_bloc\rangle_2]\!]_{inst\_bloc} \end{array} \right. & \begin{array}{l} \text{si } \alpha = \ \boldsymbol{if}\ \langle condition\rangle \\ \quad \boldsymbol{then}\ \langle inst\_bloc\rangle\ \boldsymbol{endif} \\ \text{si } \alpha = \ \boldsymbol{if}\ \langle condition\rangle \\ \quad \boldsymbol{then}\ \langle inst\_bloc\rangle_1 \\ \quad \boldsymbol{else}\ \langle inst\_bloc\rangle_2\ \boldsymbol{endif} \end{array} \\
[\![\alpha]\!]_{int\_atom} & = & \left\{ \begin{array}{l} [\{v\}, \{\text{read}\}, \emptyset, \{v \mapsto \text{read}\}] \\ [\{v\}, \{\text{write}\}, \emptyset, \{v \mapsto \text{write}\}] \\ [\{v\}, \{\text{exec}\}, \emptyset, \{v \mapsto \text{exec}\}] \end{array} \right. & \begin{array}{l} \text{si } \alpha = \ \boldsymbol{read} \\ \text{si } \alpha = \ \boldsymbol{write} \\ \text{si } \alpha = \ \boldsymbol{exec} \end{array}
\end{array}
$$

TAB. 3.2 – Sémantique des tâches TML.

3.2.2 Sémantique d'un canal

Nous construisons d'une manière automatique le modèle d'un canal à partir de son type et de sa taille (voir tableau TAB .3.3). Un canal TML du type *NBR-NBW* permet les lectures et les écritures d'une manière non bornée quelle que soit sa taille n. Nous traduisons un tel comportement sur les pomsets par un choix entre le pomset singleton de l'action d'écriture et celui de l'action de lecture. Puisque chaque action peut être exécutée indéfiniment, les deux pomsets singletons d'écriture et de lecture sont étendus par une répétition de la concaténation infinie ($*$).

Un canal bloquant en lecture et en écriture *BR-BW* est une file d'attente qui est contrôlée par sa taille n. Nous traduisons ces types de canaux en utilisant l'opérateur d'imbrication concaténée entre le pomset singleton représentant l'action d'écriture et le pomset singleton représentant l'action de lecture. Par exemple :

- La traduction d'un canal *BR-BW* de taille une, $n = 1$, est représenté par le processus : $p \overset{1;*}{\bowtie} p' = (p; p')^{*}$ avec $p = [\{v\}, \{\text{write}\}, \emptyset, \{v \mapsto \text{write}\}] \wedge p' = [\{v\}, \{\text{read}\}, \emptyset, \{v \mapsto \text{read}\}]$

- La traduction d'un canal ***BR-BW*** de taille deux, $n = 2$, est représenté par le processus :

$$p \overset{2;*}{\bowtie} p' = (p; (p; p')^*; p')^*$$

Notons que le comportement du canal de taille un est imbriqué dans le comportement du canal de taille deux. Nous nous basons sur cette propriété d'imbrication pour la construction du modèle pomset des canaux de taille plus importante. Dans le cas d'un canal infini ***BR-NBW*** le modèle du canal est construit avec $(n = *)$.

$$[\![\alpha]\!]_{\mathit{channel}} = [\![\langle \mathit{ch\_type}\rangle\ \langle \mathit{width}\rangle]\!]_{\mathit{ch\_type}} \quad \text{si } \alpha = \langle \mathit{channel}\rangle$$

$$[\![\alpha]\!]_{\mathit{ch\_type}} = \begin{cases} [\{v\}, \{\mathsf{write}\}, \emptyset, \{v \mapsto \mathsf{write}\}]^* \cup [\{v\}, \{\mathsf{read}\}, \emptyset, \{v \mapsto \mathsf{read}\}]^* & \text{si } \alpha = \mathit{NBR\text{-}NBW}\ n \\ [\{v\}, \{\mathsf{write}\}, \emptyset, \{v \mapsto \mathsf{write}\}] \overset{n;*}{\bowtie} [\{v\}, \{\mathsf{read}\}, \emptyset, \{v \mapsto \mathsf{read}\}] & \text{si } \alpha = \mathit{BR\text{-}BW}\ \ n \\ [\{v\}, \{\mathsf{write}\}, \emptyset, \{v \mapsto \mathsf{write}\}] \overset{*;*}{\bowtie} [\{v\}, \{\mathsf{read}\}, \emptyset, \{v \mapsto \mathsf{read}\}] & \text{si } \alpha = \mathit{BR\text{-}NBW} \end{cases}$$

TAB. 3.3 – Sémantique des canaux TML.

3.3 Les systèmes de transitions finis

Pour étudier la préservation des propriétés, nous utilisons les outils de model-checking. Ces derniers sont basés sur les machines à états, (automates). Les automates sont des outils puissants pour modéliser le comportement des systèmes, notamment des systèmes concurrents. Parmi la famille des machines à états nous trouvons les ***systèmes de transitions étiquetés*** notés LTS (de l'anglais Labelled Transition Systems) [8]. Un LTS est constitué d'un ensemble d'états reliés entre eux par des transitions. Chaque transition est étiquetée. L'ensemble des étiquettes d'un LTS est appelé l'alphabet du LTS. Dans la suite, nous nous limitons à des LTS finis.

Définition 16 (Labelled tranition systems LTS)
Un système de transition étiqueté LTS *est un tuple* $\langle A, S, S_0, \rightarrow\rangle$ *avec : A l'*alphabet *est l'ensemble fini de toutes les actions ; S l'ensemble d'états fini du système ;* s_0 *est un état initial du système ; et* $\rightarrow \subseteq S \times A \cup \{\tau\} \times S$ *est la* relation de transition

Nous distinguons entre les actions observables appartenant à l'alphabet d'actions et les actions non-observables que nous notons τ. Nous notons par $s \xrightarrow{e} s'$ tout triplet $(s, e, s') \in \rightarrow$. La relation de transition s'étend à des séquences d'actions, $\rightarrow \subseteq S \times A^* \times S$ comme suit :

- $s \xrightarrow{\varepsilon} s' \Leftrightarrow s = s'$ avec ε représentant le mot vide.
- $s \xrightarrow{et} s'' \Leftrightarrow \exists s' \in S.\ s \xrightarrow{e} s' \wedge s' \xrightarrow{t} s''$ avec $e \in A\ \wedge\ t \in A^*$.

Une trace est une séquence d'actions dans lesquels le système peut s'engager depuis l'état initial. Avant de donner la définition formelle des traces, nous définissons par $next_{\mathsf{LTS}}(s)$ les actions dans lesquelles un état s peut s'engager.

Définition 17 (Actions franchissables (next))
Soit LTS $= \langle A, S, S_0, \rightarrow \rangle$ *un système de transitions, la fonction* $next_{\mathsf{LTS}}(s) : S \longrightarrow \mathcal{P}(A)$ *est définie par :*

$$next_{LTS}(s) \stackrel{def}{=} \{e \in A \mid \exists s' \in S.\ s \xrightarrow{e} s'\}$$

Nous distinguons les différentes variantes de la définition de trace :

Définition 18 Traces

1. $t \in A^*$ *est une* **trace**, *s'il existe un état* s_i *tel que* $s_0 \xrightarrow{t} s_i$. *L'ensemble des traces d'un* LTS *notée* $Trace(\mathsf{LTS})$ *est définie par :*

$$Trace(\mathsf{LTS}) \stackrel{def}{=} \{t \in A^* \mid \exists s \in S_0, s \xrightarrow{t} s'\}.$$

2. $t \in A^*$ *est une* **trace complète**, *s'il existe un état* s_i *tel que* $s_0 \xrightarrow{t} s_i$ *et* $s_i \nrightarrow$, *c'est-à-dire, à partir de* s_i *on ne peut plus s'engager dans d'autres actions. L'ensemble des traces complètes d'un* LTS, *notée* $CTrace(\mathsf{LTS})$, *est définie par :*

$$CTrace(\mathsf{LTS}) \stackrel{def}{=} \{t \in A^* \mid \exists s \in S_0, s \xrightarrow{t} s' \wedge next_p(s') = \emptyset\}.$$

3. $t \in A^\infty$ *est une* **trace infinie**, *avec* $t = ab....$ *L'ensemble des traces infinies d'un* LTS, *notée* $Trace^\infty(LTS)$ *est définie par :*

$$Trace^\infty(\mathsf{LTS}) \stackrel{def}{=} \{t \in A^\infty \wedge t = ab... \mid \exists s \in S_0, s \xrightarrow{a} s' \xrightarrow{b} s''...\}.$$

À partir de cette définition, nous établissons la notion d'un préfixe d'une trace. Un préfixe d'une trace, trace complète et trace infinie d'un système est considéré comme une trace de celui-ci. La trace préfixe d'une trace n'est pas complète ni finie car il existe ou moins une action à suivre.

Nous pouvons construire des systèmes de transitions à partir d'autres systèmes de transitions. Le produit ***synchronisé*** permet d'obtenir le système de transition d'un système composé de processus communiquants à partir des systèmes de transitions de chaque composant. Les actions synchronisées peuvent seulement être exécutées quand tous les LTSs franchissent l'action de la synchronisation au même moment (Rendez-Vous).

Définition 19 (Produit synchronisé)
Soient deux LTSs $\langle S_1, s_{0_1}, A_1, \rightarrow_1 \rangle$ *et* $\langle S_2, s_{0_2}, A_2, \rightarrow_2 \rangle$. *Soit* $Syn \subseteq A_1 \cup A_2 \cup A_1 \times A_2$ *une contrainte de synchronisation. Le produit synchronisé des deux* LTSs *est le* LTS $\langle S, s_0, A, \rightarrow \rangle$ *tel que :* $S = S_1 \times S_2$ *;* $s_0 = (s_{0_1}, s_{0_2})$ *;* $A \subseteq Syn,$ *; et* $\rightarrow \subseteq S \times A \times S$ *définie par les règles suivantes :*

- $(s_1, s_2) \xrightarrow{a_1} (s'_1, s_2)$ *ssi* $s_1 \xrightarrow{a_1} s'_1 \in \rightarrow_1 \wedge \quad a_1 \in Syn,$
- $(s_1, s_2) \xrightarrow{a_2} (s_1, s'_2)$ *ssi* $s_2 \xrightarrow{a_2} s'_2 \in \rightarrow_2 \wedge \quad a_2 \in Syn,$
- $(s_1, s_2) \xrightarrow{(a_1,a_2)} (s'_1, s'_2)$ *ssi* $s_1 \xrightarrow{a_1} s'_1 \in \rightarrow_1 \wedge\ s_2 \xrightarrow{a_2} s'_2 \in \rightarrow_2 \wedge (a_1, a_2) \in Syn.$

La contrainte de synchronisation contient le tuple des actions qui doivent être exécutées par le rendez-vous. Si $Syn \subseteq A_1 \cup A_2$ alors les deux systèmes s'exécutent en parallèle (de façon asynchrone), ce produit est appelé ***produit parallèle***.

3.3.1 Sémantiques de raffinement des systèmes de transition

Dans les sémantiques des algèbres de processus et des logiques temporelles, on distingue la notion de trace d'exécution (le temps linéaire) et la notion d'arbre d'exécution (le temps arborescent). Dans la première, les exécutions possibles d'un processus sont déterminées par ses traces (chemins). Dans la seconde, la structure interne de branchement du processus est prise en compte. Définir une relation de raffinement ou d'équivalence entre les LTSs dépend de la sémantique considérée : sémantique de temps linéaire ou sémantique de temps arborescent. Van Glabbeek [111] a étudié les sémantiques dans le cadre de l'algèbre de processus et il propose un treillis pour une dizaine de sémantiques connues, définies en terme de relations ordonnées, de la sémantique la plus faible qui permet la vérification d'équivalence de comportement de deux processus, à la sémantique la plus forte qui définit l'identité de la structure entre deux processus. Dans sa classification, on trouve comme sémantique la plus grossière (la plus faible) la sémantique de traces, et la plus fine (la plus forte), la sémantique de bisimulation. L'ordre du classement caractérise un ordre sur la préservation des propriétés : toutes les relations (de raffinement ou d'équivalence) de la sémantique plus fine impliquent les relations de la sémantique plus grossière.

Dans notre travail, nous cherchons à construire le système concret en s'assurant que nous n'avons pas introduit des nouvelles traces d'exécution et des nouveaux états de blocages. Pour cela, nous choisissons la vérification du raffinement par la sémantique de traces complètes infinies données par [16] et définie comme suit.

Définition 20 (Relation de raffinement)
Soient $\mathsf{LTS}_1 = \langle S_1, s_{0_1}, A_1, \rightarrow_1 \rangle$ *et* $\mathsf{LTS}_2 = \langle S_2, s_{0_2}, A_2, \rightarrow_2 \rangle$ *deux* LTS*s et soit une action* $a \in A_2$*. La relation de raffinement* $\rho \subseteq S_1 \times S_2$ *est la plus petite relation satisfaisant les conditions suivantes pour que* S_2 *raffine* S_1 *:*

1. *Raffinement strict :*
   $\forall s_2, s_2' \in S_2.\ \exists s_1, s_1' \in S_1.\ (s_2\, \rho\, s_1 \wedge s_2 \xrightarrow{a} s_2' \in \rightarrow_2) \Rightarrow (s_1 \xrightarrow{a} s_1' \wedge s_2'\, \rho\, s_1')$
2. *Raffinement par begaiement :*
   $\forall s_2, s_2' \in S_2.\ \exists s_1 \in S_1.\ (s_2\, \rho\, s_1 \wedge\ s_2 \xrightarrow{\tau} s_2' \in \rightarrow_2) \Rightarrow s_2'\, \rho\, s_1$
3. *Non introduction de deadlocks :*
   $\forall s_2 \in S_2.\ \exists s_1 \in S_1.\ (s_2\, \rho\, s_1 \wedge s_2 \nrightarrow) \Rightarrow s_1 \nrightarrow$
4. *Non* τ*-divergence :*
   $\forall s_2 \in S_2.\ \exists s_1 \in S_1.\ s_2\, \rho\, s_1 \Rightarrow \neg(s_2 \xrightarrow{\tau^+} s_2)$

Nous dirons que LTS_1 est raffiné par LTS_2 ou LTS_2 raffine LTS_1 selon la relation ρ notée $\mathsf{LTS}_1 \sqsubseteq_\rho \mathsf{LTS}_2$, quand $\forall s_2.(s_2 \in S_2 \Rightarrow \exists s_1.(s_1 \in S_1 \wedge\ s_1\, \rho\, s_2))$. Cette relation de raffinement satisfait les propriétés suivantes :

- Les clauses 1 et 2 signifient que les chemins du système concret LTS_2 doivent exister dans le système abstrait LTS_1 et ne sont qu'une dilatation des chemins de système abstrait décrit par des τ-transitions.

- La clause 4 signifie que les transitions non-observables ne doivent jamais prendre indéfiniment la main. Dans des LTSs finis, un tel chemin est décrit par un τ-cycle.

- La clause 3 signifie que les nouvelles transitions ne doivent pas introduire de nouveaux états de blocage. Donc, pour chaque état de blocage s_2 du système LTS_2 il doit exister au moins un état de blocage s_1 dans le système LTS_1 telle qu'on arrive aux deux états par la même trace.

Dans [16] il a été montré que ρ peut être calculée pour des systèmes de transition finis. Cela permet de prouver algorithmiquement le raffinement par un parcours du graphe d'accessibilité de l'automate abstrait LTS_1 et l'automate concret LTS_2. Cette relation de raffinement garantit la préservation d'un sous-ensemble des traces complètes et infinies du modèle abstrait. Comme ces traces vérifient déjà les propriétés qui sont analysées vraies sur le modèle abstrait LTS_1, alors les propriétés restent vraies sur le modèle concret LTS_2. Donc, vérifier le raffinement entre deux modèles d'un système avec cette relation permet de vérifier la préservation des propriétés linéaires de sûreté et de vivacité du système abstrait.

Cette ρ-simulation est plus forte que la sémantique de traces décrit en bas du treillis de Van Glabbeek [111]. La sémantique de raffinement de traces est définie par les deux clauses 1 et 2 de la définition 20, ce qui signifie qu'un préfixe d'une trace est considéré aussi comme une trace acceptable sur le système concret. Cette définition donne une possibilité d'introduction des nouveaux états de blocages sur le système concret ce qui peut causer la non-préservation des propriétés de vivacité comme l'inévitabilité d'exécution d'une action. Nous illustrons la différence entre ces deux sémantiques par les deux exemples suivants :

Exemples *Considérons deux systèmes* LTS_1 *abstrait et* LTS_2 *concret (voir figure* FIG. *3.2). Le système* LTS_2 *est obtenu après une opération de réduction de la trace constituée de la boucle infinie de l'action* a *sur* LTS_1 *par une trace d'une seule transition étiqueté par* a.

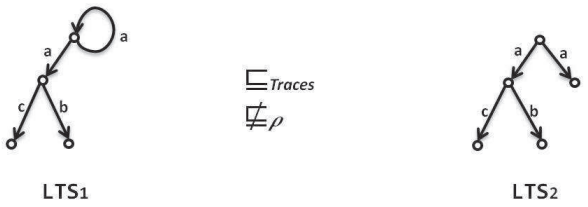


FIG. 3.2 – Exemple illustratif de raffinement de traces mais pas traces complètes infinies.

L'ensemble des traces des deux systèmes :

- $Trace(\mathsf{LTS}_1) = \{a,\ ab,\ ac,\ aa,\ aaa, \ldots\}$ et $Trace(\mathsf{LTS}_2) = \{a,\ ab,\ ac\}$.
- $CTrace(\mathsf{LTS}_1) = \{ab,\ ac\}$ et $CTrace(\mathsf{LTS}_2) = \{a,\ ab,\ ac\}$.
- $Trace^{\infty}(\mathsf{LTS}_1) = \{aaaaaa.....\}$ et $Trace^{\infty}(\mathsf{LTS}_2) = \emptyset$.

Dans ce cas, le système concret est un raffinement de traces selon les clauses 1 et 2 de la ρ-simulation, mais n'est pas un raffinement de trace complètes infinies car le raffinement ne vérifie pas la clause 3 de cette définition. Ce qui est vérifié par la non inclusion des traces complètes, $CTrace(\mathsf{LTS}_2) \nsubseteq CTrace(\mathsf{LTS}_1)$. *En effet, la nouvelle trace complète introduite* a *n'existe pas dans* LTS_1, *donc un nouveau état de blocage est construit.*

La figure FIG. *3.3 montre un autre exemple de modèle* LTS_1 *abstrait et de modèle* LTS_2 *concret. Le modèle concret est obtenu par l'élimination de la boucle infinie étiquetée par l'action* a.

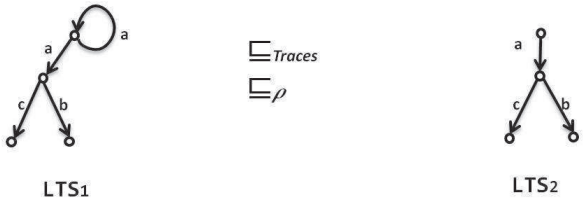


FIG. 3.3 – Exemple illustratif de raffinement de traces et de traces complètes infinies.

L'ensemble des traces du système LTS_2 *dans ce cas :*

- $Trace(\mathsf{LTS}_2) = \{a,\ ab,\ ac\}$.
- $CTrace(\mathsf{LTS}_2) = \{ab,\ ac\}$.
- $Trace^{\infty}(\mathsf{LTS}_2) = \emptyset$.

Le système concret est un raffinement de traces complètes et infinies selon la définition 20, car nous avons éliminé la trace infinie et nous avons gardé toutes les traces finies sans introduction des nouveaux traces, des nouveaux états de blocage ni des τ*-cycles. On a bien :*

$$CTrace(\mathsf{LTS}_2) \cup Trace^{\infty}(\mathsf{LTS}_2) \subseteq CTrace(\mathsf{LTS}_1) \cup Trace^{\infty}(\mathsf{LTS}_1)$$

Dans notre démarche de raffinement, nous introduisons de nouvelles actions sur le modèle concret du système vis-à-vis du modèle abstrait de celui-ci. Afin de comparer le comportement des deux modèles, ces derniers doivent être définis sur le même alphabet, pour cela il faut définir un renommage des actions du système du système concret vis-à-vis du système abstrait. On utilise la notion d'actions observables et non-observables, les actions non-observables sont masquées, et les actions observables sont associées à des actions observables du LTS du système abstrait. Nous définissons le processus de masquage par la fonction *Masq*(LTS, A_2) qui renomme toutes les actions de A_1 qui ne sont pas dans A_2 par l'action non-observable τ.

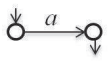
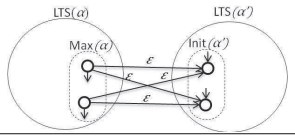
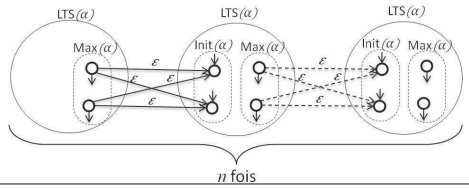
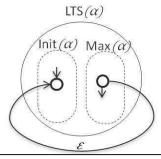
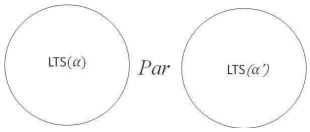
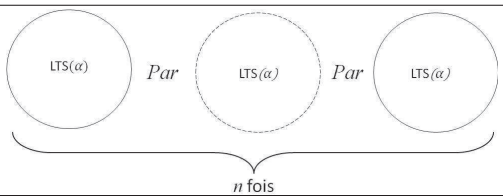
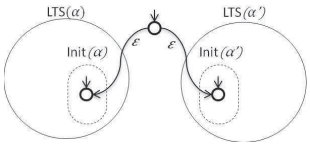
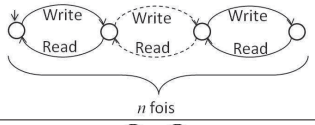
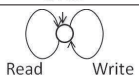
Définition 21 (Masquage d'actions par τ)
Soit $\mathsf{LTS} = \langle S_1, s_{0_1}, A_1, \rightarrow_1 \rangle$ *un automate et soit* A_2 *un sous-ensemble de* A_1 *(*$A_2 \subseteq A_1$*).* *Masq*$(\mathsf{LTS}, A_2) = \langle S_2, s_{0_2}, A_2 \cup \{\tau\}, \rightarrow_2 \rangle$ *avec :* $S_2 = S_1$, $s_{0_2} = s_{0_1}$ *et* $\rightarrow_2$ *est définit pour chaque* $a \in A_1$ *et* $s, s' \in S_1$ *par :*

- $s \xrightarrow{a} s' \ \in \rightarrow_2 \quad ssi \quad s \xrightarrow{a} s' \in \rightarrow_1 \wedge\ a \in A_2$,
- $s \xrightarrow{\tau} s' \ \in \rightarrow_2 \quad ssi \quad s \xrightarrow{a} s' \in \rightarrow_1 \wedge\ a \in A_1 \setminus A_2$.

L'automate résultant de cette fonction de masquage est un automate qui contient le même nombre d'états et de transitions que l'automate d'entrée, les transitions étiquetées par les actions concrètes sont rendues non-observables en les renommant par l'action τ.

3.3.2 Traduction des pomsets aux LTSs

Nos transformations de raffinement sont décrites sur les pomsets et l'étude de la préservation de comportement et propriétés est réalisée sur des LTS, ce qui nécessite la traduction des modèles pomsets des tâches et canaux en des modèles LTSs. Le tableau TAB 3.4 montre un squelette de traduction de modèle pomset en LTS. Pour les tâches, nous traduisons chaque pomset singleton ou événement d'un pomset par un automate contenant deux états, un état initial et un état final, reliés entre eux avec une transition. La transition part de l'état initial vers l'état final et désigne l'action associée à l'événement du pomset.

	Pomset	Représentation LTS
(1)	$[\{v\}, \{\mathsf{a}\}, \emptyset, \{v \mapsto \mathsf{a}\}]$	
(2)	$\alpha; \alpha^{'}$	
(3)	α^{n}	
(4)	α^{*}	
(5)	$\alpha \parallel \alpha^{'}$	
(6)	$\alpha^{\mid n}$	
(7)	$\alpha \cup \alpha'$	
(8)	$[\{v\}, \{\mathsf{write}\}, \emptyset, \{v \mapsto \mathsf{write}\}] \overset{*,n}{\bowtie} [\{v\}, \{\mathsf{read}\}, \emptyset, \{v \mapsto \mathsf{read}\}]$	
(9)	$[\{v\}, \{\mathsf{write}\}, \emptyset, \{v \mapsto \mathsf{write}\}]^{*} \cup [\{v\}, \{\mathsf{read}\}, \emptyset, \{v \mapsto \mathsf{read}\}]^{*}$	

TAB. 3.4 – Traduction des modèles POMSETS vers des modèles LTSs

La concaténation de deux pomsets $\alpha; \alpha'$ est traduite par une concaténation de LTSs par la fusion des états finaux du premier LTS, notés $\mathrm{Max}(\alpha)$, aux états initiaux de deuxième LTS, noté $\mathrm{Init}(\alpha')$. La fusion de deux états d'un LTS est caractérisée par une transition étiquetée par le mot vide ε. La répétition infinie d'un pomset est traduite par la fusion des états finaux du pomset aux états initiaux de celui-ci. Deux pomsets concurrents sont traduits par un produit parallèle (Par) des deux LTSs correspondant. L'union entre deux pomset est traduite par le choix indéterministe entre les LTSs correspondant aux deux pomsets. Les lignes de 1 à 7 du tableau TAB 3.4 montre la traduction des pomset des

tâches. Pour les canaux, la traduction vers les LTSs est réalisée selon le type de canal. Pour les canaux de type BR-BW qui sont décrits par l'opérateur $\overset{*,n}{\bowtie}$ avec $n \in \mathbb{N}$, nous construisons un LTS représentant le comportement de file d'attente de n, la valeur de n représente le nombre d'écritures successives permises avant une lecture. Pour les canaux de type BR-NBW nous affectons à n la valeur maximale des entiers codés sur le langage utilisé à la description des LTSs. Pour les canaux de type NBR-NBW qui sont décrits par un processus pomset avec l'opérateur d'union, nous les traduisons par un automate d'un seul état avec deux transitions, une pour l'action d'écriture et l'autre pour l'action de lecture. Les lignes 8 et 9 du tableau TAB 3.4 montrent la traduction des pomsets des canaux en LTS.

3.4 Conclusion

L'objectif de ce chapitre est la description des formalismes mathématiques sur lesquels nous nous basons pour la réalisation des étapes de transformation et l'analyse du raffinement. Nous avons présenté au premier lieu le formalisme *Pomset* et nous avons proposé une extension de ce formalisme pour l'adapter à nos besoins. Nous avons également présenté les systèmes de transitions finis LTSs qui possèdent des outils permettant le raisonnement sur la préservation de propriétés.

Les deux formalismes sont connus pour la modélisation des systèmes concurrents. Pour gérer le parallélisme entre les différentes actions d'une application lors du processus de projection de celle-ci sur une architecture. Nous avons utilisé le formalisme pomset qui permet la distinction d'ordre des actions par une association de ces derniers aux événements disjoints. Avec cette propriété de distinction des actions et la possibilité de gestion des séquencements représentés par une relation d'ordre entre les différents événements, les étapes de transformation et de rajout des contraintes d'ordre sont bien cadrées mathématiquement.

Pour effectuer l'analyse du raffinement, nous transformons le modèle pomset des différents composants en un système de transition étiqueté. Le passage du formalisme *Pomset* aux machines d'états permet d'utiliser les algorithmes de calcul et de vérification de relation de simulation existant sur les machines à états pour l'analyse du raffinement. Ce passage permet de valider notre démarche de raffinement algorithmiquement. Dans notre démarche, nous étudions la préservation des propriétés linéaires par raffinement en se basant sur la relation de simulation préservant les traces complètes infinies.

Dans le chapitre suivant nous décrivons en détail les étapes de raffinement et les différentes transformations que nous proposons pour chaque étape de la démarche de raffinement proposée.

Chapitre 4

Raffinement des canaux de communication

Sommaire

Dans ce chapitre, nous donnons les différentes étapes proposées pour encadrer le raffinement des canaux de communication. Les étapes de transformation commencent par le raffinement de granularité de données manipulées par l'application, cette étape est suivie par le raffinement de la gestion de canaux en introduisant un protocole explicitant le comportement du canal abstrait. Enfin, le raffinement se termine par une introduction d'un bus abstrait représentant une famille de bus partagé.

Par raffinement des modèles des tâches et des canaux, les actions du niveau abstrait sont transformées en des séquences d'actions du niveau concret avec possibilité de création d'entrelacement. Pour décrire l'ordre entre les actions, nous nous basons sur le formalisme *Pomset*. L'ordre entre les actions concrètes est établi en prenant en compte des détails et des paramètres de l'architecture cible. Le modèle concret construit par ce processus représente l'ensemble des traces d'exécution de l'application "acceptées" par l'architecture. L'analyse de la préservation du comportement fonctionnel se fait par l'analyse de préservation des traces des modèles LTSs associés aux *Pomsets*. En fait, par application des règles proposées dans un ordre donné nous garantissons la non-création de nouvelles traces ou d' états de blocage qui n'existaient pas dans le modèle initial.

4.1 Premier raffinement : Changement de granularité de données

Le changement de granularité de données consiste à changer la granularité de données du gros-grain en une granularité de grains plus fins ou inversement. Nous considérons dans la suite le cas du raffinement, c'est-à-dire, le passage de granularité de données du gros-grain en une granularité de grain plus fin. Le choix de la granularité de données est lié à l'implémentation ciblée et au choix du concepteur. Ce changement de granularité de données implique le raffinement des actions des modèles des tâches et des canaux en un ensemble d'actions manipulant les données de granularité plus fines (micro-actions) ce qui fait passer le modèle d'un système décrit dans le NIVEAU_0 vers un modèle de système décrit dans le NIVEAU_1. Ce modèle raffiné est régi par les contraintes d'architecture (la taille de mémoire allouée à chaque canal et la contrainte de partage des ressources).

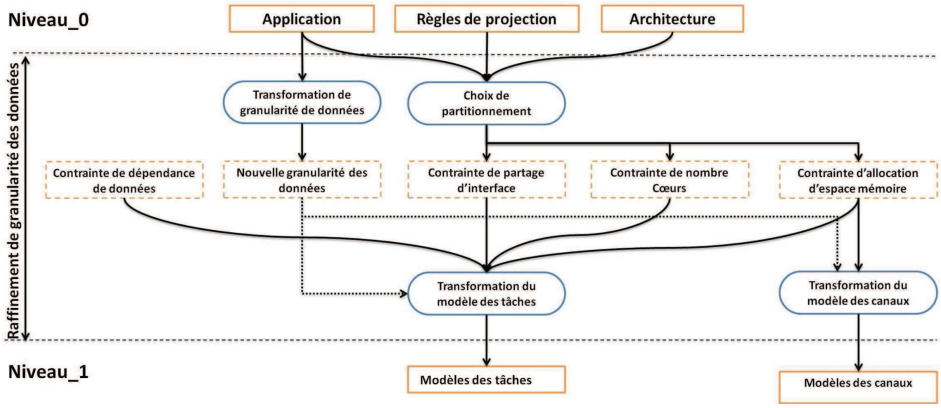


FIG. 4.1 – Contraintes prises en compte lors du raffinement de granularité de données

Comme le montre le schéma de la figure FIG. 4.1, le raffinement des canaux est guidé principalement par la contrainte de la taille de l'espace mémoire alloué au canal après la projection. Le raffinement des tâches est guidé quant à lui par les contraintes de dépendance de données, d'allocation d'espace mémoire partagé et local, ainsi que le nombre d'interfaces et de cœurs d'exécution sur un PE. Le raffinement d'un canal est réalisé par une étape de transformation ; en revanche le raffinement d'une tâche se décompose en plusieurs étapes de transformations. À chaque étape de transformation, une contrainte est prise en compte. Pour illustrer ce premier raffinement donnons un exemple qui servira tout au long des étapes de transformation de ce chapitre.

Exemple 5 *Considérons l'exemple du producteur-consommateur décrit en* TML *: deux tâches qui se synchronisent sur une file d'attente à une case mémoire (voir le code de la figure* FIG. *4.2(a)). La tâche* TASK1 *envoie des données à travers le canal* C1 *avant de récupérer d'autres données sur le canal* C2*, la tâche* TASK2 *envoie des données à travers le canal* C2 *après le traitement des données du canal* C1*. Le canal* C1 *(resp.* C2*) permet le transfert des données de type* IMAGE1 *(resp.* IMAGE2*) entre les tâches. Nous considérons dans cet exemple que le type* IMAGE1 *(resp.* IMAGE2*) est de taille 6 unité architecturale* UNIT *(resp. 4* UNIT*).*

Supposons la projection de cette application sur une architecture composée de deux PE*s interconnectés par un bus et une mémoire partagée (voir la configuration dans la figure* FIG. *4.2(b)). La tâche* TASK1 *est projetée sur l'élément* PE1*, la tâche* TASK2 *est projetée sur l'élément* PE2 *et les canaux* C1 *et* C2 *sur le bus et la mémoire.*

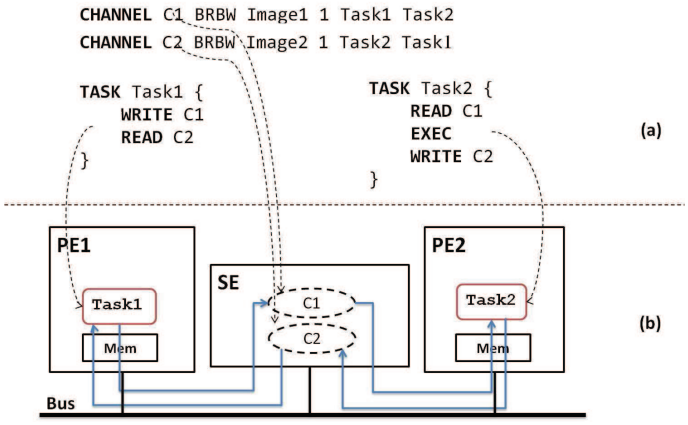


FIG. 4.2 – Description et projection de l'application TML (producteur/consommateur)

Supposons aussi le changement de granularité de données de type IMAGE1 *et* IMAGE2 *en une granularité plus fine de type* PIXEL*. De plus, nous supposons que le type* PIXEL *correspond à une unité architecturale* UNIT*.*

4.1.1 Transformation du modèle des canaux

Le modèle des canaux du NIVEAU_1 est obtenu par la transformation du modèle des canaux du NIVEAU_0 (des canaux TML). Cette transformation de modèle de canaux est réalisée selon les critères suivants :

- L'existence ou non du parallélisme entre les micro-actions lors d'opérations d'écriture ou de lecture sur le canal.
- La taille du canal. La taille d'un canal C, notée $size(C)$, représente le nombre de pas de transfert successifs permis en écriture (resp. lecture) avant de pouvoir exécuter une action de lecture (resp. d'écriture).
- La taille de la mémoire physique allouée au canal. En effet, le nombre maximal de données stockées sur un canal ne doit pas dépasser la taille maximale de la mémoire allouée à celui-ci.

Les deux premiers critères sont des paramètres qui seront fournis par le concepteur, le troisième critère est déterminé à partir du choix de projection. Les canaux TML n'ont qu'un seul point d'accès, nous choisissons de ne pas paralléliser les micro-actions de transfert lors de raffinement afin de préserver cette propriété de non-simultanéité des lectures et écritures sur les canaux abstraits. La transformation d'un canal est décrite par la transformation 1, notée $\mathcal{T}_1$.

Transformation 1 (Transformation du canal)**.**

Un canal **c** manipulant des données de type t_1 est transformé en un canal **c'** manipulant des données de type t_2, avec :
$type(c') =$ BR-BW si $type(c) =$ BR-BW ou BR-NBW,
$type(c') =$ NBR-NBW si $type(c) =$ NBR-NBW.
De plus, la taille d'un canal raffiné **c'**, projeté sur une zone mémoire de taille m, doit être inférieure ou égale à l'espace de la zone allouée et à la taille du canal abstrait :

$$size(c') \times t_2 \leq min(m, size(c) \times t_1)$$

Comme nous utilisons le formalisme *Pomset* pour décrire l'ordre partiel étiqueté des exécutions des tâches et des canaux, la transformation 1, se traduit sur ce formalisme comme suit :

Formalisation. Étant donné un canal c projeté sur une zone mémoire z de taille m, la transformation $\mathcal{T}_1$ est définie par :

$$\mathcal{T}_1(c,z) = \begin{cases} [\{v\},\{\text{write}\},\emptyset,\{v \mapsto \text{write}\}]^* \cup [\{v\},\{\text{read}\},\emptyset,\{v \mapsto \text{read}\}]^* \\ \qquad \text{si } type(c) = \textit{NBR-NBW} \\ [\{v\},\{\text{write}\},\emptyset,\{v \mapsto \text{write}\}] \overset{*,m}{\bowtie} [\{v\},\{\text{read}\},\emptyset,\{v \mapsto \text{read}\}] \\ \qquad \text{si } type(c) = \textit{BR-BW} \vee type(c) = \textit{BR-NBW} \wedge size(z) = m \end{cases}$$

Un canal projeté sur un espace de taille finie ne peut être qu'un canal fini. Les canaux FIFO infini de type BR-NBW sont donc transformés en des canaux FIFO finis de type BR-BW. Dans le cas où la taille du canal raffiné est inférieure à la taille du canal abstrait ($size(C') \times t_2 \leq t_1 \times size(C)$), une réduction d'alternatives d'exécution, donc de traces, sur le comportement du système raffiné par rapport au système abstrait peut être identifiée. Dans le cas inverse, c'est-à-dire, la taille du canal raffiné est supérieure à la taille du canal abstrait ($size(C') \times t_2 \geq t_1 \times size(C)$), il y aura création de nouvelles alternatives d'exécution dans le système raffiné par rapport à sa spécification abstraite, c'est exactement ce que nous souhaitons éviter, car nous considérons le modèle d'application comme une base d'exécution souhaitable du système. Dans le cas où le concepteur choisit d'utiliser plus d'espace sur son modèle des canaux, la solution que nous proposons est d'augmenter la taille du canal initial.

Exemple 6 *Reprenons notre exemple de producteur-consommateur. Considérons la projection du canal* C1 *(resp.* C2*) de type* BR-BW *sur une zone mémoire de taille 6* UNIT *(resp. de taille 4* UNIT*) allouée sur l'élément de stockage* SE*.*

Selon la transformation 1, le canal C1 *(resp.* C2*) de taille de 1 case, manipulant des données de type* IMAGE1 *(resp.* IMAGE2*), peut être transformé en un canal* C'1 *(resp.* C'2*) de type* BR-BW*, manipulant des données de type* PIXEL *et qui ne dépasse pas une taille de six cases de stockage (resp. quatre cases de stockage).*

4.1.2 Transformation des modèles des tâches

Les modèles des tâches sont obtenus par la transformation des modèles des tâches générés au NIVEAU _0. En effet, le changement de granularité de données résultant du choix d'implémentation introduit un changement sur la nature des données manipulées par les actions des tâches, ce qui introduit également une transformation sur les actions de traitement et de transfert. Les actions de transfert des données des tâches seront décomposées en des micro-actions de transfert de données de taille plus petite. Les exécutions peuvent être également subdivisées en des micro-actions d'exécution. Par contre, le paramètre de décomposition des actions d'exécution peut être différent de celui des actions de transfert ; il est soit fourni par le concepteur ou il peut être extrait du choix d'implémentation cible.

À cette étape, chaque action d'une tâche sera transformée en un groupe de micro-actions ; les micro-actions appartenant à un même groupe portent sur des données de même granularité, ce qui n'est pas toujours le cas entre les actions appartenant à deux groupes différents. Le modèle d'une tâche sera calculé par substitution de chaque action de la tâche par le groupe de micro-actions associé et par la construction de relation d'ordre entre les micro-actions intra-groupe et inter-groupe de façon à ce que le modèle résultat préserve le comportement du modèle initial. Nous caractérisons la transformation des tâches de ce niveau par quatre étapes successives : substitution d'actions du modèle des tâches, introduction de la dépendance de données, construction d'ordre en respectant la contrainte de la persistance des données, et introduction de les contraintes du nombre d'interfaces et du nombre de cœurs dans un élément de calcul.

1. Substitution d'actions dans un modèle de tâche

La conséquence du changement de granularité de données est le remplacement d'une action par un groupe de micro-actions, nous appelons cette transformation "expansion d'action". Par exemple, une lecture d'une donnée de type image se raffine en n lectures de données de type pixel.

READ

$$\underbrace{\textit{Read}\quad \textit{Read}\quad \ldots\ \textit{Read}}_{n}$$

Étant donné que les canaux de communication considérés n'ont qu'un seul point d'accès, les actions de transfert ne peuvent se faire qu'en séquence. Par conséquence, un ordre linéaire est imposé aux actions de transfert raffinées. Et étant donné l'absence de l'identité et de la valeur des données en TML, cet ordre sera arbitraire. La substitution précédente devient :

READ

$$\underbrace{\textit{Read} \rightarrow \textit{Read} \rightarrow \cdots \rightarrow \textit{Read}}_{n}$$

Concernant le calcul, le nombre des micro-actions doit être spécifié par le concepteur à partir de l'implémentation cible et l'ordre entre les actions doit être calculé à partir du

nombre de cœurs d'exécution du PE sur lequel la tâche est projetée. La transformation d'expansion d'actions est décrite par la transformation 2, notée $\mathcal{T}_2$.

Transformation 2 (Expansion d'actions 1)**.**

Considérons le modèle d'une tâche. Étant donné un ensemble d'actions d'une tâche, un paramètre associé au changement de la granularité de données et un paramètre de parallélisme maximal sur l'élément de calcul sur lequel la tâche est projetée. La transformation par expansion d'actions consiste à substituer chaque action de la tâche par un ensemble d'actions ordonnées paramétré par le paramètre de changement de granularité et le paramètre de parallélisme maximal.

Formalisation. Soit $p = [V, A, \prec, \mu]$ le pomset correspondant à une tâche obtenu par la fonction $[\![.]\!]_{Task}$, $n \in \mathbb{N}$ un paramètre de changement de granularité et $q \in \mathbb{N}$ un paramètre de parallélisme maximal. La transformation $\mathcal{T}_2$ est définie pour chaque action $a \in A$ par :

$$\forall a \in A,\ \mathcal{T}_2(a, n, q) = [\{v_1, v_2, ...v_n\}, \{a\}, \prec, \{a \mapsto v \mid v \in \{v_1, v_2, ..., v_n\}]$$

Cette transformation associe un pomset à chaque action du pomset de la tâche d'entrée obtenu par la fonction $[\![.]\!]_{Task}$. La relation d'ordre $\prec$ sur les pomsets associés aux actions est définie selon le paramètre q. Pour les actions de transfert $a \in \{\text{WRITE}, \text{READ}\}$, le paramètre $q = 1$, c'est-à-dire qu'un seul transfert n'est possible à la fois et donc la relation d'ordre est totale, on aura alors :

$$\mathcal{T}_2(\text{READ}, n, 1) = [\{v_1, v_2, ..., v_n\}, \{\text{READ}\}, (v_i, v_j)_{1 \le i \le j \le n} \in \prec, v \mapsto \text{READ}.\ \forall v \in \{v_1, v_2, ..., v_n\}]$$

$$\mathcal{T}_2(\text{WRITE}, n, 1) = [\{v_1, v_2, ..., v_n\}, \{\text{WRITE}\}, (v_i, v_j)_{1 \le i \le j \le n} \in \prec, v \mapsto \text{WRITE}.\ \forall v \in \{v_1, v_2, ..., v_n\}]$$

Par contre, la valeur du paramètre q pour l'action EXEC est définie selon le nombre de cœurs dont dispose l'élément de calcul PE sur lequel s'exécute la tâche.

Afin de construire l'ordre du modèle concret de la tâche avec la nouvelle granularité de données et préserver le comportement initial de l'application, il faut reporter sur le modèle concret les ordres entre les actions du modèle abstrait. Ainsi, une fois l'expansion d'actions réalisée, la construction du modèle se poursuit en étendant l'ordre entre les actions des différents groupes. Cette transformation 3, notée $\mathcal{T}_3$, traduit les ordres du modèle abstrait sur le modèle concret.

Transformation 3 (Reconstruction d'ordre des actions)**.**

Considérons le modèle d'une tâche. Étant donné une transformation par expansion d'actions sur l'ensemble de ses actions, la reconstruction d'ordre sur les actions du modèle de la tâche consiste à trouver les actions maximales des différents groupes d'actions et de construire un ordre entre ces actions d'ordre maximal en respectant l'ordre du modèle abstrait.

Formalisation. Soit $p = [V, A, \prec, \mu]$ le pomset correspondant à une tâche. Et soit une transformation $\mathcal{T}_2$ sur l'alphabet A. La transformation $\mathcal{T}_3$ est définie par :

$$\mathcal{T}_3(p) = p^{[\mathcal{T}_2 \circ \mu]}$$

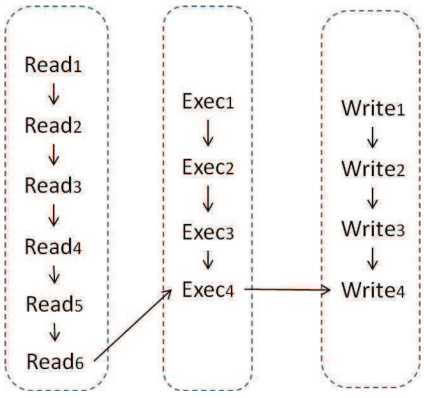
La transformation $\mathcal{T}_3$ est la substitution par les événements supérieurs du résultat de la transformation $\mathcal{T}_2$ sur les actions du pomset d'entrée p. Remarquons que cette transformation est définie sur un pomset, elle se généralise au processus du modèle de la tâche par l'application de l'opérateur de répétition infinie sur le pomset résultat. Notons que nous avons utilisé la forme currifiée de la transformation $\mathcal{T}_2$, c'est-à-dire, elle est écrite sous une forme qui ne considère que le paramètre $a \in A$, les paramètres n et q sont implicites.

Exemple 7 *Reprenons notre exemple, le pomset de la tâche* Task2 *obtenu par l'application de la fonction de traduction* $[\![$Task2$]\!]_{\text{Task}}$ *est représenté par le graphe suivant :*

$$(\text{READ} \longrightarrow \text{EXEC} \longrightarrow \text{WRITE})^*$$

*Notons que l'exécution infinie de la tâche est représentée par l'opérateur de répétition infinie du pomset, notée *. Nous omettrons cet opérateur dans les modèles qui suivent, nous ne montrons que la transformation du pomset.*

Selon notre hypothèse de transformation de granularité de données de type IMAGE *en type* PIXEL*, une lecture d'image depuis le canal* C1 *(resp. écriture vers le canal* C2*) est transformée en 6 lectures successives de pixels (resp. 4 écritures successives de pixels). L'exécution sur une image est transformée en 4 exécutions sur des pixels. Et d'après l'hypothèse, l'élément de calcul* PE2 *ne contient qu'un seul cœur d'exécution. L'application de la transformation* $\mathcal{T}_2$ *sur les actions* READ, WRITE *et* EXEC *sera :* $\mathcal{T}_2(\text{READ}, 6, 1)$, $\mathcal{T}_2(\text{EXEC}, 4, 1)$, $\mathcal{T}_2(\text{WRITE}, 4, 1)$*. En appliquant la transformation* $\mathcal{T}_3$ *sur le résultat de ces trois transformations au pomset d'entrée de la tâche* Task2 *on obtient le pomset suivant :*



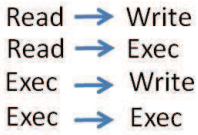
Notons que cette substitution établit un ordre entre les événements supérieurs uniquement. En effet, elle impose que la dernière action de lecture READ_6 *soit avant les dernières actions d'exécution* EXEC_4 *et d'écriture* WRITE_4 *; de même la dernière action d'exécution* EXEC_4 *soit avant la dernière action d'écriture* WRITE_4*.*

À ce niveau, on a effectué la substitution d'actions abstraites des tâches par des actions raffinées. De plus, on a rétabli un ordre sur les actions selon l'ordre du modèle abstrait.

Dans les étapes suivantes, cet ordre sera réévalué par la prise en compte de contraintes supplémentaires.

2. Introduction de dépendance de données

Afin de gérer au mieux l'espace mémoire de l'architecture sur laquelle est projetée l'application, il est nécessaire d'introduire sur les modèles des tâches la dépendance de données entre ces actions. La dépendance existe principalement entre les actions de lecture-écriture, lecture-exécution, exécution-écriture et exécution-exécution. Elle est explicitement exprimée par une relation de précédence représentée entre les actions :



La flèche bleue signifie que l'exécution de l'action à droite requiert les données produites par l'action à gauche. Une conséquence directe de l'abstraction des valeurs des données en TML est la perte de cette information sur la dépendance entre les données et entre les actions. Cette information de dépendance de données peut être fournie par le concepteur ou extraite de l'implémentation cible si celle-ci est connue.

La réintroduction de cette information dans le modèle pomset peut casser l'ordre entre les actions décrit dans l'étape de transformation précédente. Afin de préserver l'ordre initial, il faut que l'union de la relation d'ordre existante et la relation de dépendance de données fournie n'engendre pas un cycle d'exécution. Dans le cas contraire, la relation de dépendance doit être révisée. La transformation 4 décrit cette opération de réintroduction de la dépendance de données.

Transformation 4 (Introduction de dépendance de données)**.**

Considérons le modèle d'une tâche. Étant donné une relation de dépendance de données entre ses actions, la transformation par introduction de la dépendance de données consiste à introduire dans le modèle de la tâche l'ordre entre les actions fourni par la relation de dépendance de données de façon à s'assurer que le modèle obtenu construit bien un ordre entre les actions de la tâche (absence de cycle d'exécution).

Mathématiquement, une *dépendance de données* est une relation *acyclique* et *non-transitive*. Rappelons qu'une relation acyclique est une relation antisymétrique, c'est-à-dire, ne permet pas un chemin de retour entre deux éléments qui sont en relation. La transformation 4, notée $\mathcal{T}_4$, est alors formalisée comme suit :

Formalisation. Soient un pomset $p = [V, A, \prec, \mu]$ et $D \subseteq V \times V$ une relation de dépendance de données entre les événements du pomset p.

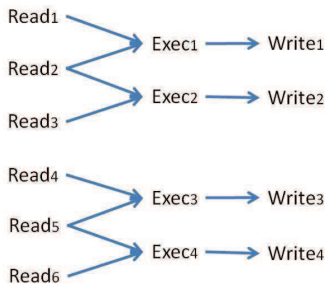
$$\mathcal{T}_4(p, D) = p \lhd D$$

La transformation $\mathcal{T}_4$ renforce l'ordre du pomset p par la relation de dépendance de données D. Elle augmente donc l'ordre du pomset p avec la fermeture transitive de la relation D.

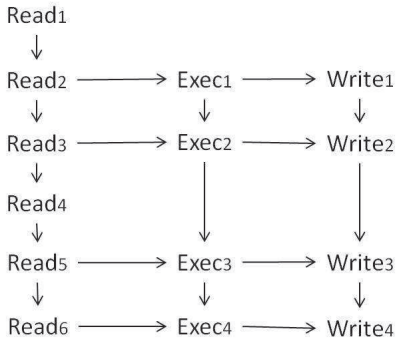
Exemple 8 *Revenons à notre exemple. Supposons donnée une relation de dépendance de données D sur les actions de la tâche* Task2 *:*

$$\begin{aligned} D \quad = \quad & \{(\text{READ}_1,\text{EXEC}_1)\,;(\text{READ}_2,\text{EXEC}_1)\,;(\text{READ}_2,\text{EXEC}_2)\,;(\text{READ}_3,\text{EXEC}_2)\,; \\ & (\text{READ}_4,\text{EXEC}_3)\,;(\text{READ}_5,\text{EXEC}_3)\,;\,(\text{READ}_5,\text{EXEC}_4)\,;(\text{READ}_6,\text{EXEC}_4)\,; \\ & (\text{EXEC}_1,\text{WRITE}_1)\,;\,(\text{EXEC}_2,\text{WRITE}_2)\,;(\text{EXEC}_3,\text{WRITE}_3)\,;(\text{EXEC}_4,\text{WRITE}_4)\} \end{aligned}$$

La représentation graphique de cette dépendance est donnée par la figure ci-dessous.



L'application de la transformation $\mathcal{T}_4$ sur le modèle pomset de notre exemple, c'est-à-dire, le pomset obtenu après application de la transformation $\mathcal{T}_3$ (exemple 7 page 85), et la relation de dépendance de donnée D ci dessous donnera le pomset suivant :



Notons que la relation de dépendance de données rajoute de nouvelles relations d'ordre entre les groupes d'actions. Par exemple les relation : $\text{READ}_2 \longrightarrow \text{EXEC}_1$ *et* $\text{READ}_3 \longrightarrow \text{EXEC}_2$. *Par contre, d'autres actions peuvent demeurer parallèles, par exemple,* EXEC_1 *et* READ_3.

Ayant à ce stade un modèle d'une tâche partiellement transformée, nous cherchons à construire dans la suite une implémentation de la tâche conforme à la contrainte de disponibilité de l'espace mémoire (persistance de données) et l'exclusion mutuelle sur les accès aux ressources partagées.

3. Persistance de données

Une notion liée à la dépendance de données est la persistance de données. L'espace occupé par une donnée produite par une action ne doit être libéré qu'après que la donnée produite soit utilisée par toutes les actions qui en dépendent. Une donnée produite peut être écrasée quand toutes les actions requérantes sont exécutées.

À chaque tâche est allouée un espace mémoire local pour la manipulation des données des canaux. Afin de respecter la contrainte de persistance de données, le modèle de chaque tâche doit d'une part être exécutable sur l'espace utilisé, et d'autre part, assurer que les données produites ne soient pas écrasées avant que toutes actions requérantes soient exécutées. La transformation 5, notée $\mathcal{T}_5$, transforme le modèle de façon à garantir le respect de cette contrainte.

Transformation 5 (Persistance de données)**.**

Considérons le modèle d'une tâche. Étant donné une relation de dépendance de données et une taille de mémoire locale, cette transformation consiste à contraindre l'ordre du modèle de la tâche de façon à ce qu'il ne puisse pas y avoir de production successive d'une quantité de données dont la taille totale est supérieure à la taille de cet espace.

Partant de l'information sur l'espace mémoire alloué aux canaux de données manipulées par une tâche fournie lors de la projection, et l'information de dépendance de données, nous calculons un nouvel ordre entre les événements en fonction de l'impact de l'ensemble d'actions sur la taille de l'espace mémoire alloué. Plusieurs ordres peuvent être possibles pour l'exécution d'une tâche. Le modèle de la tâche s'exécutant sur une architecture doit être construit de façon à ce que toutes les exécutions possibles vérifient le non-dépassement de la taille mémoire allouée à la tâche.

Pour définir cette propriété, nous introduisons la notion d'espaces libérables sur un ensemble d'actions exécutées. Pour cela, nous associons d'abord à chaque action a deux fonctions :

- La fonction $Zone$ qui retourne la zone mémoire locale dans laquelle peut opérer l'action a. Les zones mémoires sont disjointes. Une zone est identifiée par un identifiant id. On note sa taille par $\|id\|$.
- La fonction Imp qui retourne la quantité d'espace mémoire libérée ou occupée après l'exécution d'une action a. La quantité est signée (+) dans le cas d'une libération et signée (-) dans le cas d'une allocation.

En effet, les zones mémoires allouées aux actions d'une tâche appartiennent à l'élément de calcul sur lequel la tâche est projetée. Pour une action donnée a, la valeur absolue de l'impact de l'action ne doit jamais dépasser la taille de la zone mémoire allouée à celle-ci, c'est-à-dire, $|Imp(a)| \leq \|Zone(a)\|$. Nous définissons aussi l'ensemble des événements acquittés selon la dépendance de données à un instant donné dans le processus d'exécution de la tâche.

Définition 22 (Acquittement d'événements selon une relation)
Soient $p = [V, A, \prec, \mu]$ un pomset, $I \subseteq \mathcal{I}_p$ un idéal sur p et D une relation de dépendance de données sur V. L'ensemble d'événements acquittés selon l'idéal I et la dépendance de données D, noté $\mathcal{A}_p$, est défini par :

$$\mathcal{A}_p(D, I) = \{v \in I \mid \forall v' \in V.\ (v, v') \in D \Rightarrow v' \in I\}$$

Cet ensemble définit l'ensemble d'événement exécutés et dont les événements qui en dépendent sont aussi exécutés. En utilisant la notion d'événements acquittés selon une dépendance de données nous pouvons définir la quantité de mémoires utilisée à chaque pas d'exécution d'un pomset.

Définition 23 (Impact d'exécution sur une zone mémoire)
Soient $p = [V, A, \prec, \mu]$ un pomset, $I \subseteq \mathcal{I}_p$ un idéal de p et D une relation de dépendance de données sur V. Un impact d'exécution sur une zone mémoire z est une fonction, notée G_z, définie par :

$$G_z(D, I) = \sum_{(v \in I) \wedge (Zone(\mu(v)) = z)} Imp(\mu(v)) \quad - \sum_{(v' \in \mathcal{A}_p(D,I)) \wedge (Zone(\mu(v')) = z)} Imp(\mu(v'))$$

Cette fonction définit l'espace occupé sur une zone mémoire à un instant donnée d'exécution d'un pomset. Le calcul est réalisé par le décompte d'impacts des événements acquittés sur cette zone de l'impact des événements exécutés sur la même zone. En fait, cette définition est donnée pour une zone mémoire. Puisque nous considérons que les zones mémoires sont disjointes, cette définition se généralise à une région ou l'ensemble de la mémoire par la somme d'impact des zones. Nous définissons alors la propriété d'exécutabilité d'un pomset en prenant en compte la taille de la mémoire.

Propriété 1 (Pomset exécutable selon la taille de la mémoire allouée)
Soient $p = [V, A, \prec, \mu]$ un pomset, $\mathcal{I}_p$ l'ensemble des idéaux du pomset p, D une relation de dépendance de données sur V, et $\mathcal{Z} = \bigcup_{a \in A} Zone(a)$ l'ensemble des zones mémoires allouées pour les actions du pomset p. Le pomset p est exécutable selon la taille de la mémoire allouée si et seulement si :

$$\forall\ I \subseteq \mathcal{I}_p.\ \ \forall\ z \in \mathcal{Z}.\ \ G_z(D, I)\ \leq\ \parallel z \parallel \qquad \textit{(P1)}$$

Cette propriété permet de vérifier le non-écrasement des données lors d'exécution de pomset d'une tâche. Le pomset vérifie cette propriété si pour tous les idéaux du pomset, l'impact d'exécution des événements sur chaque zone mémoire ne dépasse pas la taille de celle-ci.

Mathématiquement, la transformation $\mathcal{T}_5$ décrit une augmentation d'ordre sur le pomset d'entrée de façon que le pomset résultat satisfait la propriété de pomset exécutable selon la taille de la mémoire dont dispose le PE sur lequel s'exécute la tâche. Elle se caractérise par :

Formalisation. Soient $p = [V, A, \prec, \mu]$ un pomset, D une relation de dépendance de données sur les événements de V. Soient $Zone$, Imp deux fonctions sur l'alphabet A.

$$\mathcal{T}_5(p) = p'$$

tel que p' satisfait la propriété ***P1*** selon la relation D et les fonctions $Zone$ et Imp.

Notons que $\mathcal{T}_5$ peut retourner p comme résultat dans le cas où ce dernier satisfait la propriété *P1*. Dans le cas contraire, p' sera le résultat d'une restriction d'ordre sur p. Si aucun pomset satisfaisant la propriété *P1* ne peut être construit par cette transformation, alors la tâche associée au pomset n'est pas exécutable sur le PE associé. Dans ce cas, les contraintes de projection doivent être révisées.

Les transformations $\mathcal{T}_1$, $\mathcal{T}_2$, $\mathcal{T}_3$ et $\mathcal{T}_4$ sont des transformations directes de pomsets. Pour la transformation $\mathcal{T}_5$ l'opération est plus complexe, elle peut être donnée sous forme d'un algorithme de recherche de solution qui satisfait la propriété *P1*.

Algorithme de $\mathcal{T}_5$

Soient $p = [V, A, \prec, \mu]$ un pomset, D une relation de dépendance de données sur les événements de V, et $Zone$, Imp deux fontion sur A. Un algorithme possible de construction de pomset exécutable selon la contrainte de la taille mémoire est le suivant :

1. *Calculer l'idéal initial, noté* $I = Inf_p(V)$*, sur le pomset* p*.* (Définition. 14)
2. *Calculer l'ensemble des événements franchissables, noté* $Access_p(I)$*.* (Définition. 15)
3. *Construire un ensemble de sous-ensembles d'événements franchissables, noté* $\mathcal{E}$*, tel que pour chaque ensemble* $E \in \mathcal{E}$ *le nouveau idéal construit,* $I \cup E$*, satisfait la propriété P1.*

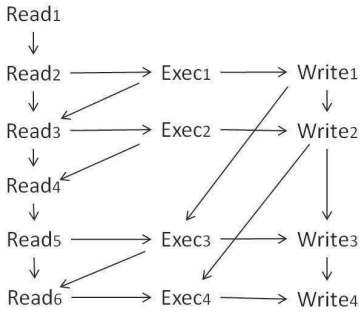
$$\mathcal{E} = \{E \mid I \cup E \in \mathcal{I}_p \wedge I \cup E \ \ satisfait \ \ P1\}$$

4. Si $\mathcal{E}$ est non vide, $\mathcal{E} \neq \emptyset$, alors :
   (a) *Choisir un élément* $E \in \mathcal{E}$ *et construire un nouveau idéal,* $I' = I \cup E$*.*
   (b) *Construire un ordre entre les événements du nouveau idéal* I' *et le reste d'événement* $V \setminus I'$*. On obtient donc le pomset :* $p' = [V, A, \prec \cup (I' \times (V \setminus I')), \mu]$*.*
   (c) *Si* $I' \neq V$*, alors répéter les étapes 2, 3 et 4, en prenant comme paramètres le nouveau pomset* p' *et le nouveau idéal* I'*.*
   (d) *Sinon, l'idéal construit est égal à l'ensemble d'événements du pomset et le pomset construit est le pomset de sortie.*
5. *Sinon, si l'ensemble d'événements franchissables est vide,* $E = \emptyset$*, et nous n'avons pas parcouru tous les événements du pomset,* $I \neq V$*, alors restaurer le contexte précédent et chercher une nouvelle solution d'ordre à partir d'un nouveau choix d'idéal, faire un retour sur le point 4.(a) et choisir un nouveau ensemble franchissable. Par ailleurs, si pour tous les idéaux de p, il n'existe pas un sous-ensemble d'événements franchissables qui satisfait la propriété P1 alors le choix de la projection de la tâche doit être révisé.*

Cet algorithme est récursif. Il recherche un ordre d'exécution des actions de la tâche qui satisfait le non-écrasement des données décrit par la propriété *P1*. Il parcourt en profondeur

l'ensemble des exécutions possibles du pomset. L'algorithme s'arrête à la première solution trouvée. Si sur un état d'exécution quelconque il n'existe pas d'ensemble d'exécutions futures satisfaisant la propriété *P1* , l'algorithme backtrack pour choisir un nouveau chemin d'exécution. Dans le cas au aucune solution n'est trouvée, les paramètres de la projection doivent être révisés.

Exemple 9 *Soient le modèle de la tâche* *Task2* *obtenu après la transformation* $\mathcal{T}_4$ *et la relation de dépendance de données D donnée dans l'exemple 8. Considérons que lors de la projection de la tâche* *Task2* *sur l'élément* PE2*, le concepteur alloue un espace mémoire locale de quatre unités architecturales* 4 UNIT*. Cet espace est utilisé pour la manipulation des données échangées avec les canaux. Supposons que la fonction Zone associée à l'action de lecture la zone mémoire* z_1*,* $Zone(\text{READ}) = z_1$*, et aux actions d'exécution et d'écriture la zone mémoire* z_2*,* $Zone(\text{EXEC}) = z_1$ *et* $Zone(\text{WRITE}) = z_1$*. Supposons également qu'un* PIXEL *est égal à une unité architecturale* UNIT*. Une action de lecture consomme dans ce cas une* UNIT *de l'espace mémoire de la zone* z_1*,* $Imp(\text{READ}) = -1$*, une action d'exécution consomme une* UNIT *de l'espace mémoire de la zone* z_2*,* $Imp(\text{EXEC}) = -1$*, et une action d'écriture libère une* UNIT *de l'espace mémoire de la zone* z_2*,* $Imp(\text{WRTE}) = +1$*. Une solution possible après application de l'algorithme de la transformation* $\mathcal{T}_5$ *sur le résultat de la transformation* $\mathcal{T}_4$ *est le pomset suivant :*



Selon la fonction d'association d'actions aux espaces mémoires, la tâche peut exécuter au plus deux lectures consécutives avant d'effectuer un calcul et au plus deux calculs consécutifs avant d'effectuer une écriture. Selon la contrainte de dépendance de données D, nous pouvons déduire que l'action de calcul EXEC_1 *(resp.* EXEC_3*) s'affranchit de la dépendance de l'action de lecture* READ_1 *(resp.* READ_3*) ce qui libère l'espace de la donnée associé à la lecture, et l'action de calcul* EXEC_2 *(resp.* EXEC_4*) s'affranchit de la dépendance des actions* READ_2 *et* READ_3 *(resp.* READ_5*,* READ_6*) ce qui libère deux espaces mémoires de données associés aux lectures. De plus, chaque action d'écriture vers le canal, c'est-à-dire* $\text{WRITE}_1, \text{WRITE}_2, \text{WRITE}_3$ *et* WRITE_4*, libère un espace de données produit par les actions d'exécution.*

4. Contrainte de nombre d'interface et de cœurs

Un autre type de contraintes matérielles prises en compte est le nombre d'interfaces et le nombre de cœurs dont dispose un élément de calcul. De la même manière que le partage d'espace mémoire, ces contraintes engendrent une transformation des modèles des tâches afin de construire un modèle des tâches qui réalise l'exclusion mutuelle lors d'accès aux ressources.

Partant de l'hypothèse de projection, une tâche est projetée sur un élément de calcul. Nous spécifions à chaque action d'une tâche un cœur d'exécution et une interface qu'elle utilise sur cet élément de calcul. Pour cela, nous introduisons les deux fonctions suivantes :

- La fonction $Core$ qui fournit l'identité du cœur sur lequel une action s'exécute.
- La fonction Inf qui fournit l'identité de l'interface sur laquelle une action de communication réalise le transfert.

Les actions de transfert sont associées aux interfaces pour le transfert de données depuis et vers le canal projeté sur une mémoire partagée. Les actions d'exécution, quant à elle, ne sont pas associées à une interface car elles ne requièrent que la mémoire locale. À cette étape de transformation, le modèle de la tâche doit respecter le partage des ressources (cœurs et interfaces). Cette contrainte se caractérise par la propriété suivante :

Propriété 2 (Pomset exécutable selon la contrainte de cœurs et d'interfaces) *Soit $p = [V, A, \prec, \mu]$ un pomset et soient $Core$, Inf deux fonctions sur A. Le pomset p de la tâche est exécutable selon la contrainte des cœurs et interface si :*

$$\forall a, a' \in A.\ Core(a) = Core(a') \vee Inf(a) = Inf(a') \Rightarrow (a, a') \in \prec \vee (a', a) \in \prec \qquad \textit{(P2)}$$

Cette propriété exprime la nécessité d'existence d'un ordre entre deux actions qui partagent la même ressource. La transformation de la tâche selon la contrainte de partage de cœurs et d'interface est spécifiée par la transformation 6, notée $\mathcal{T}_6$.

Transformation 6 (Partage d'interface et de cœurs)**.**

Considérons le modèle d'une tâche. Étant donné le nombre d'interfaces et de cœurs d'un élément de calcul sur lequel la tâche est exécutée, cette transformation consiste à contraindre l'ordre entre les actions qui partagent la même ressource pour garantir l'exclusion mutuelle pour l'accès aux ressources partagées.

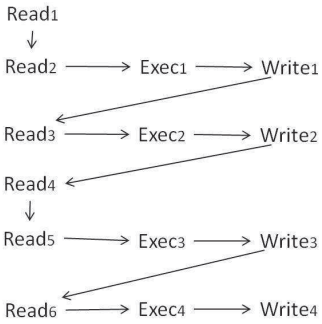
Formalisation. Soit $p = [V, A, \prec, \mu]$ un pomset, et soient définies les deux fonctions $Core$, Inf sur l'alphabet A.

$$\mathcal{T}_6(p) = p'$$

tel que p' satisfait la propriété *P2* selon les fonctions $Core$ et Inf .

À l'instar de la transformation $\mathcal{T}_5$ cette transformation peut être réalisée algorithmiquement. Le principe de l'algorithme est d'ajouter un ordre sur les événements étiquetés par les actions qui partagent la même ressource. Dans le cas d'un processeur mono-cœur et mono-interface il s'agit d'exprimer l'exclusion mutuelle entre toutes les actions de la tâche accédants à ces ressources.

Exemple 10 *Revenons à notre exemple. Considérons que l'élément* PE2 *sur lequel est projetée la tâche* Task2 *ne contient qu'un seul cœur. Dans ce cas, le modèle de cette tâche obtenu par la transformation 6 constitue un ordre total entre les actions de la tâche. Un choix possible de transformation d'ordre qui respecte la propriété P2 est le modèle suivant :*



Remarquons dans ce choix qu'un ordre est imposé entre les actions WRITE$_1$⟶READ$_3$, WRITE$_2$⟶READ$_4$ *et* WRITE$_3$⟶READ$_6$. *Le modèle de la tâche est une exécution linéaire de toutes ses actions.*

En fait, la construction du modèle de l'application est donnée composant par composant, c'est-à-dire, tâche et canaux séparés. Aussi, la construction d'une tâche est donnée indépendamment des autres tâches. Le modèle global de l'application est obtenu par composition des modèles de composants la constituant. Le problème est que l'ordre choisi séparément sur les tâches peut créer un cycle d'exécution sur le modèle global. Par exemple, dans le cas où une tâche lit une donnée qu'elle a déja écrit dans le passé et qui a été traité par d'autres tâches. Il faut donc vérifier la non création de cycle à chaque application d'une transformation. Une manière de vérifier la non-création de ce type de cycle sur le système global consiste à s'assurer à chaque étape de transformation pour chaque ordre direct entre une lecture et une écriture sur une tâche, il n'existe pas un ordre inverse indirect construit sur le modèle global.

À ce stade de raffinement, le protocole de communication avec lequel sera gérée la communication entre les tâches n'est pas décrit. La seconde étape est dédiée à cette fin, nous décrivons donc un protocole abstrait qui nous permettra de gérer la communication sur les canaux et d'aller vers une implémentation de l'application. Typiquement, le protocole de communication devra représenter le comportement des canaux abstraits.

4.2 Second raffinement : Gestion des canaux

À l'issue de la première étape de raffinement, les canaux abstraits sont raffinés en prenant en compte des informations de l'architecture sur laquelle s'exécute cette application. À présent, nous donnons plus de détails sur le protocole de communication. Le protocole abstrait que nous proposons est inspiré des travaux [92, 43]. Ce protocole décompose les accès de transfert vers et depuis le canal en des accès de gestion du canal (synchronisation) et des accès de transferts depuis et vers celui-ci. Les instructions de communication entre deux tâches (READ/WRITE) sont transformées en une séquence d'actions d'accès au canal plus des synchronisations : les actions de transfert de données sont les actions (STORE-DATA/LOAD-DATA) et les synchronisations sont les actions (CHECK-ROOM/CHECK-DATA et SIGNAL-ROOM/SIGNAL-DATA).

En fait, une action d'écriture (WRITE) est transformée en une séquence d'actions de vérification de disponibilité de l'espace dans le canal (CHECK-ROOM), suivie par une action

de d'écriture (STORE-DATA) de données avant de signaler l'action d'écriture (SIGNAL-DATA). De la même manière, une action de lecture (READ) est transformée en une séquence d'actions de vérification de disponibilité d'une donnée dans le canal (CHECK-DATA), suivie par une action de lecture de la donnée (LOAD-DATA) avant de signaler l'action la libération de l'espace (SIGNAL-ROOM). L'introduction de ce protocole impose une transformation des modèles des tâches et des canaux.

4.2.1 Transformation du modèle des canaux

Le protocole proposé permet de raffiner les canaux de type BR-BW. Les canaux de type NBR-NBW ne sont pas impactés par ce protocole. En effet, comme l'accès est libre à ces derniers, il n'y a aucun besoin de gestion des synchronisations supplémentaires sur ces canaux. Et les canaux de type BR-NBW n'ont plus d'existence à ce niveau puisqu'ils sont déjà transformés en des canaux BR-BW.

Le raffinement d'un canal BR-BW est réalisé par la transformation du canal lui-même et la transformation des actions de transfert des tâches communicant à travers ce canal. Nous illustrons cette transformation sur un exemple donné sous forme graphique (voir figure FIG. 4.3). Le canal C1 de type BR-BW est de taille deux cases, il est accédé par les tâches *TasK1* et *Task2* respectivement en écriture et lecture, l'automate de C1 (voir figure FIG. 4.3(b)) décrit la gestion FIFO de taille deux : La tâche TASK2 ne peut pas lire si la tâche TASK1 n'a pas écrit sur le canal et la tâche TASK1 ne peut plus écrire si le nombre de données produites sur le canal et non lus est égal à 2.

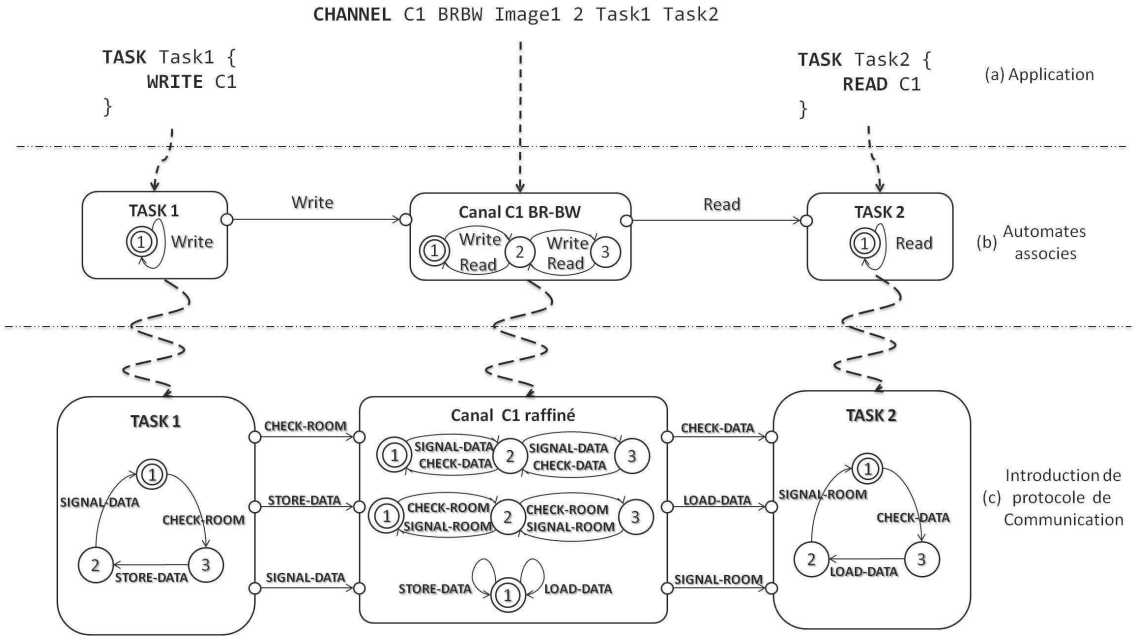


FIG. 4.3 – Exemple de raffinement d'un canal BR-BW.

Une fois raffiné, l'automate du canal C1 est transformé en un produit parallèle de trois automates (voir la figure FIG. 4.3(c)). Ce modèle du canal raffiné ne représente pas à lui seul la gestion FIFO d'un canal. En fait, la gestion FIFO du canal raffiné est

réalisée par la composition du comportement du canal avec le comportement des tâches. Les actions de lecture et d'écriture des tâches sont aussi transformées en des séquences de synchronisations et de lecture/écriture. Une écriture est réalisée par une vérification de la vacuité du canal (CHECK-ROOM) puis d'écriture effective (STORE-DATA) et enfin un déblocage de lecture en signalant la disponibilité de la donnée (signal-data). Une lecture est réalisée par une vérification de l'existence de la donnée sur le canal (CHECK-DATA) puis de lecture effective (LOAD-DATA) et enfin un déblocage de l'écriture en signalant la disponibilité de l'espace (SIGNAL-ROOM). Cette transformation d'un canal par le protocole proposé est réalisée par la transformation 7, notée $\mathcal{T}_7$.

Transformation 7 (Transformation du canal 2)**.**

Considérons un canal *c* de type BR-BW, le canal est transformé en un canal *c'* avec un protocole de synchronisation donné gérant explicitement les accès à l'espace mémoire.

Formalisation. Étant donné un canal c de $type(c) = \text{BR-BW}$ et de taille $size(c) = n$.

$$\begin{aligned}\mathcal{T}_7(c) \quad = \quad & [\{v\}, \{\text{SIGNAL-DATA}\}, \emptyset, \{v \mapsto \text{SIGNAL-DATA}\}] \overset{*;n}{\bowtie} [\{v\}, \{\text{CHECK-DATA}\}, \emptyset, \{v \mapsto \text{CHECK-DATA}\}] \quad \| \\ & [\{v\}, \{\text{CHECK-ROOM}\}, \emptyset, \{v \mapsto \text{CHECK-ROOM}\}] \overset{*;n}{\bowtie} [\{v\}, \{\text{SIGNAL-ROOM}\}, \emptyset, \{v \mapsto \text{SIGNAL-ROOM}\}] \quad \| \\ & [\{v\}, \{\text{STORE-DATA}\}, \emptyset, \{v \mapsto \text{STORE-DATA}\}]^* \cup [\{v\}, \{\text{LOAD-DATA}\}, \emptyset, \{v \mapsto \text{LOAD-DATA}\}]^*\end{aligned}$$

On voit bien dans cette formalisation les trois parties d'un canal raffiné composées par l'opérateur de concurrence ($\|$). Le premier et le second opérandes codés par l'opérateur d'imbrication sur les actions de synchronisations, SIGNAL-DATA et CHECK-DATA pour le premier opérande et SIGNAL-ROOM et CHECK-ROOM pour le second opérande. Le troisième opérande codé par l'union des deux processus infinis d'actions d'accès au canal LOAD-DATA et STORE-DATA.

4.2.2 Transformation du modèle des tâches

La transformation de ces actions par suite du raffinement du canal est donnée par la transformation 8, notée $\mathcal{T}_8$.

Transformation 8 (Expansion d'actions 2)**.**

Considérons le modèle d'une tâche. Étant donné le protocole de gestion des canaux choisi, les actions communicantes vers un canal BR-BW sont transformées par expansion en des actions respectant le protocole. Les actions de calcul sont gardées inchangées :

$$\begin{aligned}\text{READ} \quad &\equiv \quad \text{CHECK-ROOM} \rightarrow \text{STORE-DATA} \rightarrow \text{SIGNAL-DATA} \\ \text{WRITE} \quad &\equiv \quad \text{CHECK-DATA} \rightarrow \text{LOAD-DATA} \rightarrow \text{SIGNAL-ROOM} \\ \text{EXEC} \quad &\equiv \quad \text{EXEC}\end{aligned}$$

Formalisation. Étant donné un alphabet d'une tâche $A = \{\text{READ, WRITE, EXEC}\}$

$$\mathcal{T}_8(a) = \begin{cases} [\{v, v', v''\}, \{\text{CHECK-ROOM, STORE-DATA, SIGNAL-DATA}\}, \{(v, v'), (v, v''), (v', v'')\}, \\ \{v \mapsto \text{CHECK-ROOM}, v' \mapsto \text{STORE-DATA}, v'' \mapsto \text{SIGNAL-DATA}\}] \\ \qquad \text{si } a = \text{READ} \ \wedge \ \textit{name\_ch}(a) = c \ \wedge \ type(c) = \text{BR-BW} \\ [\{v, v', v''\}, \{\text{CHECK-DATA, LOAD-DATA, SIGNAL-ROOM}\}, \{(v, v'), (v, v''), (v', v'')\}, \\ \{v \mapsto \text{CHECK-DATA}, v' \mapsto \text{LOAD-DATA}, v'' \mapsto \text{SIGNAL-ROOM}\}] \\ \qquad \text{si } a = \text{WRITE} \ \wedge \ \textit{name\_ch}(a) = c \ \wedge \ type(c) = \text{BR-BW} \\ [\{v\}, \{\text{EXEC}\}, \emptyset, v \mapsto \text{EXEC}] \\ \qquad \text{si } a = \text{EXEC} \\ [\{v\}, \{\text{LOAD-DATA}\}, \emptyset, v \mapsto \text{LOAD-DATA}] \\ \qquad \text{si } a = \text{READ} \ \wedge \ \textit{name\_ch}(a) = c \ \wedge \ type(c) = \text{NBR-NBW} \\ [\{v\}, \{\text{STORE-DATA}\}, \emptyset, v \mapsto \text{STORE-DATA}] \\ \qquad \text{si } a = \text{Write} \ \wedge \ \textit{name\_ch}(a) = c \ \wedge \ type(c) = \text{NBR-NBW} \end{cases}$$

Avec *name_ch* une fonction qui retourne le nom du canal sur lequel est réalisé l'action de communication.

L'application de la transformation $\mathcal{T}_8$ sur le modèle de la tâche ne sera pas une substitution directe d'actions de la tâche. En effet, certaines actions de synchronisation peuvent être parallélisées pour permettre une exécution optimale. Par exemple, une séquence de lecture suivie par une exécution (READ→EXEC) sur le modèle abstrait ne sera pas forcément substituée par une suite linéaire de transferts suivie par une exécution :

$$\text{CHECK-DATA} \rightarrow \text{LOAD-DATA} \rightarrow \text{SIGNAL-ROOM} \rightarrow \text{EXEC}$$

En effet, l'action d'exécution (EXEC) pouvant être aussi réalisée juste après le chargement de la donnée (LOAD-DATA). Donc, l'ordre optimale serait :

$$\begin{array}{c} \text{CHECK-DATA}\rightarrow\text{LOAD-DATA}\rightarrow\text{SIGNAL-ROOM} \\ \downarrow \\ \text{EXEC} \end{array}$$

Par contre, l'action d'exécution (EXEC) ne pourra jamais s'effectuer avant l'action de chargement (LOAD-DATA). Si tel était le cas, la transformation ne préserverait pas la sémantique du modèle abstrait, la lecture avant l'exécution (l'action LOAD-DATA étant la lecture effective). Pour construire le modèle de la tâche représentant le protocole choisi et qui préserve la sémantique du modèle abstrait on applique la transformation 9, notée $\mathcal{T}_9$.

Transformation 9 (Introduction de protocole)**.**

Considérons le modèle d'une tâche. Étant donné un le protocole de gestion des canaux choisi, la transformation par introduction du protocole consiste à reconstituer des ordres sur le modèle de la tâche entre les séquences d'actions selon les règles données dans le tableau TAB. 4.1.

(1)	READ→ READ	CHECK-DATA→LOAD-DATA→SIGNAL-ROOM ↓ ↓ ↓ CHECK-DATA→LOAD-DATA→SIGNAL-ROOM
(2)	WRITE→ WRITE	CHECK-ROOM→STORE-DATA→SIGNAL-DATA ↓ ↓ ↓ CHECK-ROOM→STORE-DATA→SIGNAL-DATA
(3)	WRITE→ READ	CHECK-ROOM→STORE-DATA→ SIGNAL-DATA ↓ ↓ ↓ CHECK-DATA → LOAD-DATA →SIGNAL-ROOM
(4)	READ→ WRITE	CHECK-DATA → LOAD-DATA →SIGNAL-ROOM ↓ ↓ ↓ CHECK-ROOM→STORE-DATA→ SIGNAL-DATA
(5)	READ→ EXEC	CHECK-DATA→LOAD-DATA→SIGNAL-ROOM ↓ EXEC
(6)	WRITE→ EXEC	CHECK-ROOM→STORE-DATA→SIGNAL-DATA ↓ EXEC
(7)	EXEC→ READ	EXEC ↓ CHECK-DATA→LOAD-DATA→SIGNAL-ROOM
(8)	EXEC→ WRITE	EXEC ↓ CHECK-ROOM→STORE-DATA→SIGNAL-DATA
(9)	EXEC→EXEC	EXEC→EXEC

TAB. 4.1 – Transformation d'ordre d'actions selon le protocole de gestion de canal.

Dans ce tableau, chaque ligne présente un patron de transformation : à gauche l'ordre sur les actions du NIVEAU_1, à droite leur transformé par la transformation $\mathcal{T}_9$ qui donnent l'ordre sur les actions du NIVEAU_2.

Formalisation. Soit $p = [V, A, \prec, \mu]$ un pomset, et soit $\mathcal{T}_8$ une fonction qui associe à chaque action $a \in A$ un pomset noté $[V_{\mathcal{T}_8(a)}, A_{\mathcal{T}_8(a)}, \prec_{\mathcal{T}_8(a)}, \mu_{\mathcal{T}_8(a)}]$. Le pomset $\mathcal{T}_9(p) = [V_{\mathcal{T}_9}, A_{\mathcal{T}_9}, \prec_{\mathcal{T}_9}, \mu_{\mathcal{T}_9}]$ est le pomset obtenu par substitution des actions du pomset, tels que :

- $V_{\mathcal{T}_9} = \uplus_{v \in V}\ V_{\mathcal{T}_8(\mu(v))}$
- $\mu_{\mathcal{T}_9} = \uplus_{v \in V}\ \mu_{\mathcal{T}_8(\mu(v))}$
- $A_{\mathcal{T}_9} = \bigcup_{v \in V}\ A_{\mathcal{T}_8(\mu(v))}$
- $\prec_{\mathcal{T}_9} = (\bigcup_{v \in V}\ \prec_{\mathcal{T}_8(\mu(v))}) \cup\ \prec_T$, avec $\quad \prec_T \subseteq\ V_{\mathcal{T}_9} \times V_{\mathcal{T}_9}$.

La relation $\prec_T$ est la relation définie dans le tableau TAB. 4.1 pour chaque $(v_1, v_2)\ \in \prec$.

De la même manière que l'étape de transformation 6, après application de la transformation $\mathcal{T}_8$ une étape de restriction d'ordre selon les contraintes de nombre d'interfaces et de cœurs dont dispose un PE sur lequel s'exécute une tâche est nécessaire. Le résultat de l'application des transformations $\mathcal{T}_6$ après $\mathcal{T}_8$ est le modèle du système du NIVEAU_2.

À ce niveau de description, la gestion des canaux de communication et les transferts de données sont décrits sans que l'information sur le cheminement des données et le protocole de communication entre les différents composants d'architecture ne soit décrite. L'étape suivante introduit un support de communication abstrait permettant la mise en œuvre des politiques de gestion de bus partagée.

4.3 Troisième raffinement : Introduction de bus abstrait

À cette étape de raffinement, nous introduisons l'information de cheminement de données à travers un support de communication partagé, le support est un bus partagé dont la description est donnée dans la figure FIG. 4.4. Le choix d'avoir une représentation de bus abstrait avec un arbitre centralisé est justifié par l'avantage d'avoir un large choix de politique de gestion et la possibilité d'adapter facilement le fonctionnement du composant arbitre. Aussi, ce choix d'architecture permet d'avoir une représentation abstraite commune à un nombre étendu de bus existants tels que, la famille de bus AMBA, CoreConnect et d'autres. Nous avons choisi une description abstraite centralisée qui permet de regrouper un nombre de possibilités de bus au lieu d'une abstraction dédiée qui restreint le choix de raffinement ou impose une réécriture du modèle lors de changement de politique d'arbitrage. Ce modèle de bus abstrait ne prend pas en compte certains aspects notamment le mode verrouillé des bus ainsi que le transfert multiple sur différents canaux de bus. Aussi, le cycle d'arbitrage est réalisé d'une manière visible, et aucune notion de pipeline n'est spécifiée.

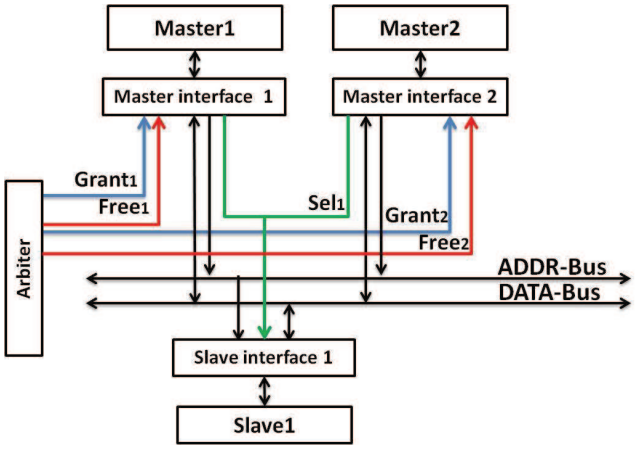


FIG. 4.4 – Modèle du bus abstrait

Concrètement, lorsqu'un maître veut accéder au bus, il envoie une requête à l'arbitre, le maître ne transfère les données à travers le bus qu'après la réception du signal d'autorisation d'accès envoyé par l'arbitre. Dans ce niveau abstrait de description, nous avons fusionné les deux signaux de demande et d'autorisation d'accès, i.e, nous ne représentons pas les signaux de demande d'accès au bus réalisés par un maître, l'arbitre propose directement le bus au maître. Ce choix simplifié le modèle de bus et permet l'exploration rapide des différentes solutions garantissant la préservation des propriétés fonctionnelles. Par contre, ce choix ne permet pas de données des détails sur des protocoles de gestion d'accès au bus basés sur l'ordre de réception des requêtes ou d'autres informations venant du maître.

Dans ce niveau de description, un maître n'accède pas au bus avant que l'arbitre ne lui ait donné la main par l'action $Grant_x$ avec x l'identité du maître. Le maître accède à un esclave par l'activation du signal Sel_y avec y l'identité de l'esclave choisi. Le maître libère le bus après réception du signal $Free_x$. Les données et les adresses sont transférées respectivement sur les bus *DATA-Bus* et *ADDR-Bus*. Les maîtres et les esclaves accédants

au bus sont dotés d'interfaces de communication, ce qui permet l'adaptation de protocoles des composants au protocole du bus. Dans la suite, nous décrivons en détail les différents composants de bus dans ce niveau.

4.3.1 Arbitre

Le composant arbitre a un comportement abstrait ; il choisit le maître à qui il attribue le bus et décide également de la libération de celui-ci ; ceci est réalisé par les signaux $Grant_X$ et $Free_X$ avec x l'identité du maître auquel est destinée le signal. L'arbitre peut gérer l'accès au bus par différentes politiques. La figure FIG. 4.5 montre deux exemples de politiques possibles d'accès au bus réalisées par un arbitre dans une architecture composée de deux maîtres.

FIG. 4.5 – Exemples de politique d'arbitrage de bus

À gauche, un exemple de politique basée sur le protocole "round-robin" où n'existe aucune priorité entre les deux maîtres. À droite, un autre exemple de politique d'arbitrage dans lequel le premier maître a la priorité sur le second, un GRANT2 est forcément suivi par un GRANT1, et un GRANT1 peut être suivi par un GRANT1 ou par GRANT2 sans priorité.

4.3.2 Interfaces de communication

Une interface est dotée des modules de mémorisation interne pour stocker les données échangées depuis et vers le bus. Nous choisissons de gérer ces modules de stockage comme des FIFOs d'entrée et de sortie. Une interface maître reçoit des commandes de lecture et/ou d'écriture du maître auquel elle est rattachée et transfère les données par le protocole du bus dédié. De l'autre côté, l'interface esclave initie la communication vers le composant cible à travers le bus de communication après avoir reçu des commandes envoyées par le maître.

Interface maître

Dans cette description, nous utilisons un protocole de communication permettant l'envoi de plusieurs données à la fois entre un composant maître et son interface : le maître accède à un espace d'adressage pour effectuer des écritures ou des lectures et indique le nombre de données à transférer, le maître sera bloqué jusqu'à la fin du transfert. De l'autre côté, l'interface doit fournir un protocole adaptable au protocole de bus pour réaliser les transferts.

La figure FIG. 4.6 montre un schéma de l'interface maître abstraite que nous proposons. Elle utilise deux FIFOs comme tampon pour la communication dans les deux modes écriture/lecture. À droite de la figure, les différents signaux échangés entre l'interface et le maître. Un composant maître initie le transfert par une requête (REQUEST) pour l'envoi ou pour la réception de données avec le nombre d'échantillons (LENGTH) ainsi

que le mode de transfert en lecture ou écriture (MODE), l'interface répond à la fin du transfert par l'action (DONE). À gauche de la figure, les différents signaux de l'interface échangés avec le bus, une interface d'un composant maître reçoit des signaux d'accès au bus (GRANT) et de libération du bus (FREE) et envoie des informations concernant l'esclave sélectionné (SEL[x]), la notation [x] représente une famille de 1 à x des composants esclaves recevant le signal. Le transfert d'adresses est réalisé sur le bus d'adresse (ADDR-BUS), et le transfert de données sur le bus de transfert de données (WDATA-BUS/ RDATA-BUS) en mode (écriture/lecture).

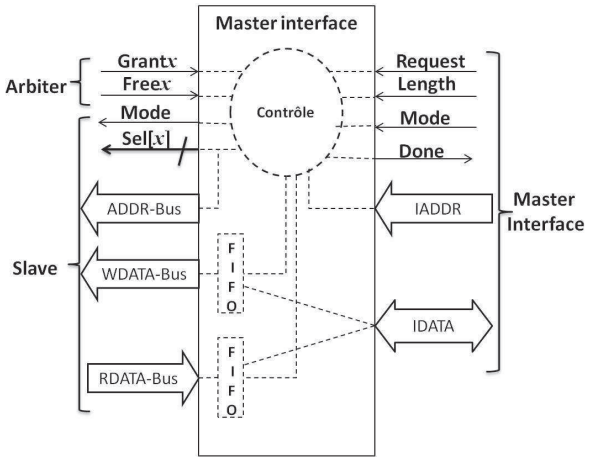


FIG. 4.6 – Interface maître abstraite

La figure FIG. 4.7 montre le modèle de fonctionnement d'un module interface maître en mode d'écriture (à gauche) et en mode lecture (à droite) dont les FIFOs ont une seule case mémoire chacune. En mode d'écriture, un maître initie le transfert avec le signal REQUEST. Dès que l'interface reçoit la requête il attend d'avoir un accès au bus (réception de GRANT) et la récupération de la donnée de la part du maître (réception des données sur IDATA), ce qui se traduit par un entrelacement entre les actions. Lorsque l'interface accède au bus et récupère la donnée, il effectue un transfert de données sur le bus WDATA-BUS et enfin l'interface libère le bus (envoi de FREE) et signale la fin du transfert au maître (envoi de DONE). En mode lecture, une interface maître cherche à accéder au bus après une requête de demande de lecture de données envoyé par le maître. Lorsqu'une interface récupère la donnée, l'interface libère le bus et procède à l'envoi de la donnée au maître.

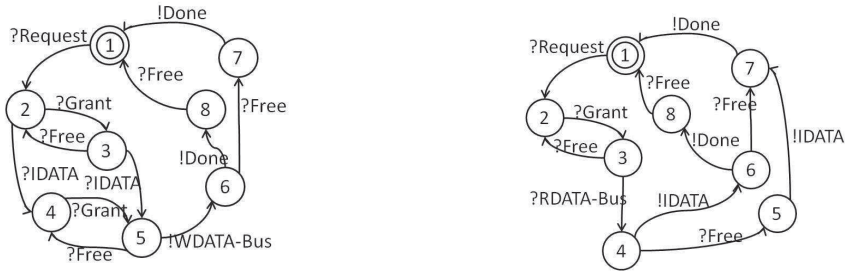


FIG. 4.7 – Exemple de fonctionnement d'interface maître, à gauche en mode écriture, et à droite en mode lecture.

Les deux modèles d'interfaces maîtres sont des versions simplifiées du modèle d'interface. Les informations de mode de transfert MODE, le transfert de plusieurs données LENGTH sont codés comme des paramètres du signal REQUEST.

Interface esclave

Une interface esclave est une interface symétrique à celle du maître, à partir de l'état initial, une interface esclave reçoit un signal d'activation (SEL) ainsi que le mode de transfert (MODE). L'interface esclave doit répondre aux requêtes d'envois et/ou de récupération de données envoyées par l'interface maître. Après la réception des commandes, l'interface esclave initie le transfert par une requête (REQUEST) pour l'envoi ou pour la réception de données avec le nombre d'échantillons (LENGTH) et le mode de transfert en lecture ou écriture (MODE) au composant dédié, ce dernier répond à la fin du transfert par l'action (DONE). La Figure FIG. 4.8 montre un schéma d'interface de communication d'un composant esclave.

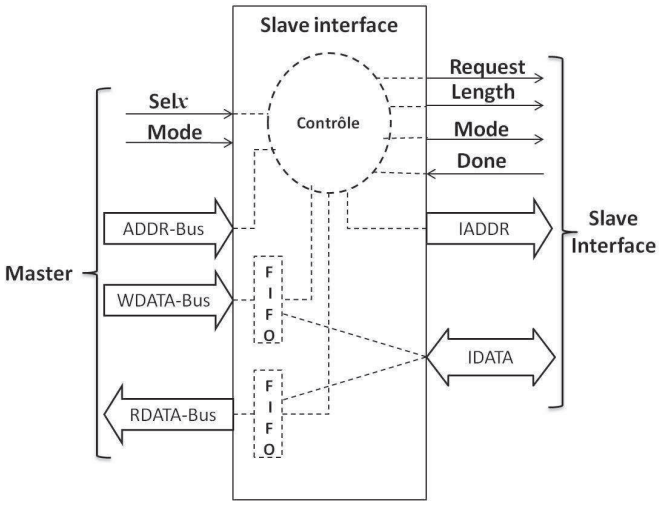


FIG. 4.8 – Interface de communication abstrait d'un esclave

La figure FIG. 4.9 montre un exemple de modèle de comportement d'une interface esclave en mode de lecture (à gauche) et écriture (à droite) dont les FIFOs ont également une seule case mémoire chacune. En mode écriture, après qu'une interface esclave soit sélectionnée, elle lit la donnée envoyée par le maître, après la réception des données, elle initie une demande vers le composant esclave. L'interface maître désactive son signal de sélection lors de la perte du bus et/ou à la fin de la transmission. Après la réception de la demande en mode lecture, l'interface esclave initie une demande au composant esclave. À cet instant, le bus peut être donné à un autre maître, ce qui cause la désactivation du signal de sélection de l'esclave, noté NSEL. L'interface esclave récupère la donnée demandée, et attend sa réactivation pour la transmettre au maître demandeur. Dans le modèle d'interface esclave proposé, nous avons choisi de ne pas interrompre le processus de récupération de donnée demandée, c'est-à-dire, l'interface esclave ne répond pas à un autre maître tant que la fin de la transmission avec le maître qui vient de perdre l'accès au bus n'est pas achevée.

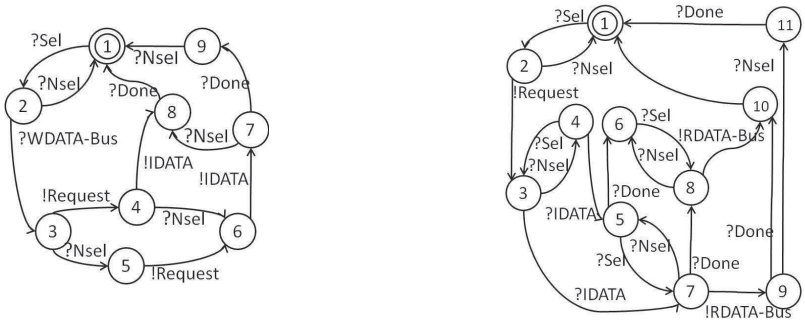


FIG. 4.9 – Exemple de fonctionnement d'interface esclave, à droite en mode écriture, à gauche en mode lecture

Nous simplifions l'interface de communication pour les composants de stockage SE (voir FIG. 4.10) dans le but de minimiser le modèle automate destiné à la vérification. Pour cela, nous intégrons l'interface de communication au composant de stockage. Cela permet d'écrire directement les données récupérées à partir du bus sur les espaces mémoires, ce qui permet d'éliminer les tampons FIFOs ainsi que l'interface de communication vers l'élément de stockage. Le protocole de communication de l'interface vers et depuis l'élément de stockage est désormais réalisé par un accès direct du contrôleur de l'interface aux données.

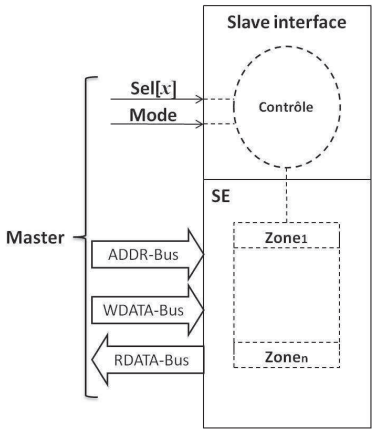


FIG. 4.10 – Interface de communication abstrait d'un élément de stockage

4.3.3 Transformation du modèle des tâches

Pour construire le modèle de l'application du NIVEAU_3, nous adaptons les modèles des tâches au protocole du bus partagé choisi. Pour cela, nous appliquons les transformations 10 et 11.

Transformation 10 (Intégration de bus)**.**

Considérons le modèle modèle d'une tâche, la transformation d'intégration de protocole de bus consiste à étendre toutes les actions de communication sous forme de requêtes d'initiation des communications (REQUEST) et d'attente de l'acquittement de fin de la communication (DONE) ; de façon à toujours préserver l'ordre des actions.

En fait, il s'agit de transformer chaque action STORE-DATA, LOAD-DATA, CHECK-ROOM, SIGNAL-ROOM, CHECK-DATA, SIGNAL-DATA en une séquence d'actions de requête vers l'interface (REQUEST), suivie par l'action de transfert en question avant de recevoir le signal de réalisation du transfert (DONE). Par exemple, une action STORE-DATA sera transformée comme suit :

$$\text{STORE-DATA} \equiv \text{REQUEST} \rightarrow \text{STORE-DATA} \rightarrow \text{DONE}$$

À cette étape la contrainte de la taille du bus de données IDATA sera prise en compte de façon à assurer le transfert intégral de données sur le bus. Nous décomposons un transfert en un ensemble de transferts selon la taille de bus IDATA par la transformation 11.

Transformation 11 (Décomposition de transfert)**.**

Considérons le modèle d'une tâche obtenu par transformation d'intégration de bus. Étant donné le la taille de bus de données IDATA, cette transformation décompose les actions de communication de données en une suite linéaire d'actions de transfert de données de granularité inférieure ou égale à la taille du bus de données.

Le nombre d'envois de données vers l'interface est spécifié par le paramètre LENGHT porté par le signal de requête RESQUEST. Par exemple, dans le cas où la taille de donnée transférée par STORE-DATA est égale au double de la taille du bus IDATA, le paramètre LENGHT sera égal à 2 et l'action de transfert STORE-DATA sera décomposée en deux actions de transfert. Les deux transformations 10 et 11 transformeront alors l'action STORE-DATA comme suit :

$$\text{STORE-DATA} \equiv \text{RESQUEST} \rightarrow \text{STORE-DATA}_1 \rightarrow \text{STORE-DATA}_2 \rightarrow \text{DONE}$$

Nous caractérisons les transformations 10 et 11 par la même transformation notée $\mathcal{T}_{11}$.

Formalisation. Soient $p = [V, A, \prec, \mu]$ un pomset et $n \in N$ un entier. $\mathcal{T}_{11}$ une fonction qui associe à chaque action $a \in A$ un pomset tels que :

$$\forall a \in A.\ \mathcal{T}_{11}(a, n) = [\{v\}, \{\text{REQUEST}\}, \emptyset, \{v \mapsto \text{REQUEST}\}]\,;[\{v\}, \{\text{a}\}, \emptyset, \{v \mapsto \text{a}\}]^n\,;[\{v\}, \{\text{DONE}\}, \emptyset, \{v \mapsto \text{DONE}\}]$$

Le pomset obtenu par cette transformation est la concaténation de pomset singleton caractérisant l'action REQUEST, la répétition concaténée du pomset singleton caractérisant l'action de transfert avec le paramètre n représentant le nombre de transfert requis pour qu'une donnée soit complètement transférée sur le bus de donnée IDATA, et le pomset singleton caractérisant l'action de fin de transfert *Done*.

Une fois l'intégration d'un protocole de bus et la décomposition de transfert réalisées, la construction d'ordre dans le modèle des tâches peut être réalisée entre les différentes séquences d'actions du protocole de transfert par la transformation 12, notée $\mathcal{T}_{12}$.

Transformation 12 (Substitution d'actions de transfert dans une tâche)**.**

Considérons le modèle d'une tâche. Étant données une transformation des actions par intégration de protocole de bus et la décomposition du transfert de celle-ci, la substitution d'actions du modèle de la tâche consiste à construire un ordre entre les micro-actions des groupes de transfert selon l'ordre entre les actions du modèle abstrait.

Formalisation. Soient $p = [V, A, \prec, \mu]$ un modèle pomset d'une tâche, et la transformation $\mathcal{T}_{11}$ sur l'alphabet A.

$$\mathcal{T}_{12}(p) = p[\mathcal{T}_{11} \circ \mu]$$

Cette transformation substitue chaque événement du pomset p par le résultat de l'application de la transformation $\mathcal{T}_{11}$ appliquée à l'ensemble de ses actions, c'est-à-dire, $\forall v \in V$, $\mathcal{T}_{11}(\mu(v))$.

Après cette transformation, le modèle du système au NIVEAU_3 est construit. Ce niveau peut bien être encore raffiné par concrétisation du protocole de bus, en introduisant des informations sur le transfert en mode pipeline et aussi des cycles d'arbitrage cachés.

4.4 Schéma de notre démarche de raffinement et de vérification des niveaux 1, 2 et 3

À chaque étape de raffinement, des règles de transformation sont appliquées en respectant des contraintes d'architecture et de projection afin de construire un modèle d'application exécutable sur l'architecture choisie. La figure FIG.4.11 récapitule les règles de transformations que nous proposons ainsi que les contraintes d'architecture et de la projection considérés pour chaque règle de transformation. Notons que les contraintes liées au support de communication bus ne sont prises en compte qu'au troisième raffinement. Notons aussi que la granularité de données choisie et la dépendance de données sur ces données doivent être compatibles avec la granularité de données et les dépendances des données de l'algorithme cible. La contrainte de dépendance de données peut être extraite de l'algorithme cible et adaptée à la granularité de données du raffinement. Le calcul de la dépendance de données de la nouvelle granularité est réalisé à partir de la dépendance de données initiale et la nouvelle granularité de données. Ce calcul n'est pas trivial, il consiste à construire à partir de la relation de dépendance de données initiale et de la correspondance entre les données de la granularité de l'algorithme et les données de la granularité du raffinement une nouvelle relation de dépendance caractérisant les données du raffinement.

Notre démarche de raffinement se décompose en trois phases : La première phase consiste à modéliser le système étudié en pomset et à appliquer les règles de transformations sur celui-ci dans les différentes étapes de raffinements. La deuxième phase consiste à générer les modèles LTSs des différents niveaux pour analyser le raffinement. La dernière phase consiste à vérifier la préservation des propriétés linéaires sur les différents niveaux du système par l'application de sémantique de raffinement des traces infinies.

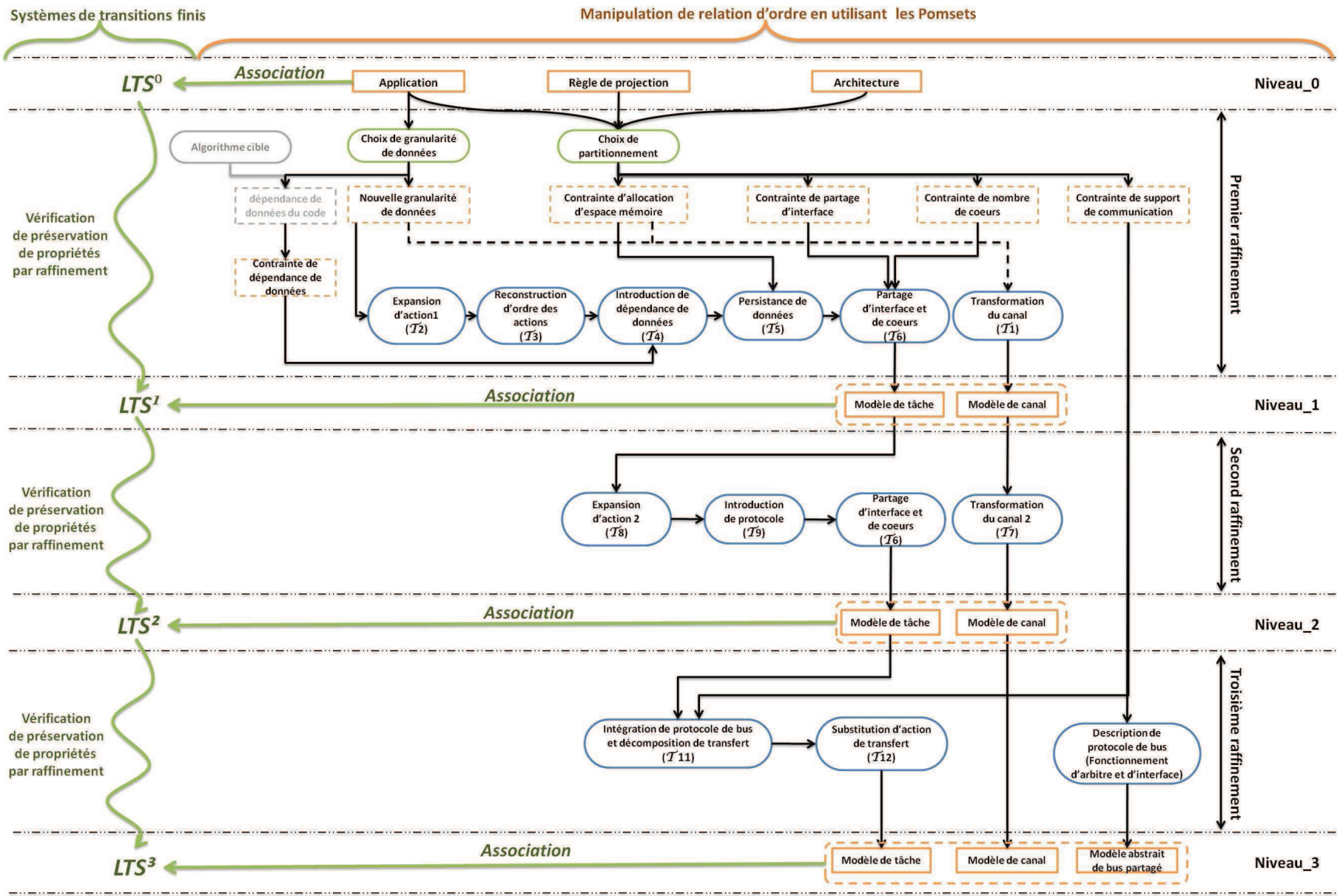


FIG. 4.11 – Schéma de démarche de raffinement et de vérification.

4.4.1 Application des transformations

La construction des modèles des canaux et tâches à chaque étape de raffinement peut être réalisée en parallèle. Pour une tâche (resp. canal) les règles de transformations sont appliquées l'une après l'autre. Nous avons choisi de spécifier des règles de transformation par des étapes simples et précises à appliquer dans un ordre déterminé. Ainsi, toutes les informations utilisées par les transformations sont soit obtenues de l'étape précédente ou intégrées comme paramètre dans l'étape courante. Lorsqu'il n'est pas possible de construire un modèle du système acceptant des contraintes de l'architecture et de la projection choisies, une révision du choix de projection ou le changement de l'architecture est nécessaire. Concrétement, nous construisons les modèles pomsets des différents niveaux comme suit :

Notons PST_t^0 (resp. PST_c^0) le pomset d'une tâche t (resp. d'un canal c) de l'application du NIVEAU_0 obtenus par la fonction $[\![.]\!]_{Task}$ (resp. la fonction $[\![.]\!]_{channel}$). En passant la première étape raffinement, les canaux de données du modèle du NIVEAU_0 sont transformés en des canaux de données du NIVEAU_1 par l'application de la transformation 1. Pour simplifier, nous considérons dans cette section la forme simplifiées des transformations en considérant que le paramètre de pomset comme entrée des fonctions.

$$\forall c \in Channel : \mathsf{PST}_c^1 = \mathcal{T}_1(\mathsf{PST}_c^0)$$

De la même manière, les modèles des tâches du NIVEAU_0 sont aussi transformés en des modèles des tâches du NIVEAU_1 par application successive des transformations $\mathcal{T}_1$, $\mathcal{T}_2$, $\mathcal{T}_3$, $\mathcal{T}_4$, $\mathcal{T}_5$ et $\mathcal{T}_6$.

$$\forall t \in Task : \mathsf{PST}_t^1 = \mathcal{T}_6(\mathcal{T}_5(\mathcal{T}_4(\mathcal{T}_3(\mathcal{T}_2(\mathsf{PST}_t^0)))))$$

Remarque : Toutes les transformations sont purement locales aux tâches et aux canaux. Le problème est que par application des transformations, on risque de créer des nouvelles dépendances entre les différentes tâches dans le modèle global du système notamment la création d'un blocage : la source du blocage dans ce cas est due à l'interdépendance d'écriture et de lecture des données entre les différentes tâches. Par exemple, deux tâches communicantes attendent respectivement la réception d'une donnée de l'un vers l'autre. Précisément, cela peut arriver après l'application des transformations $\mathcal{T}_2$ $\mathcal{T}_3$, $\mathcal{T}_4$, $\mathcal{T}_5$ et $\mathcal{T}_6$. Pour éviter ce problème, nous nous assurons à l'application de chacune de ces transformations, qu'à chaque étape de transformation pour chaque relation d'ordre entre une lecture et une écriture sur une tâche, il n'existe pas un ordre inverse indirect construit sur le modèle global.

Dans la seconde étape raffinement, nous explicitons la gestion de communication par le remplacement des canaux abstraits par le protocole de communication choisi. Les canaux de données du modèle obtenus au NIVEAU_1 sont transformés en des canaux du NIVEAU_2 par la transformation $\mathcal{T}_7$.

$$\forall c \in Channel : \mathsf{PST}_c^2 = \mathcal{T}_7(\mathsf{PST}_c^1)$$

Les modèles des tâches du NIVEAU_1 sont également transformés en des tâches du NIVEAU_2 par application des transformations $\mathcal{T}_6$, $\mathcal{T}_8$ et $\mathcal{T}_9$.

$$\forall t \in Task : \mathsf{PST}_t^2 = \mathcal{T}_6(\mathcal{T}_9(\mathcal{T}_8(\mathsf{PST}_t^1)))$$

Dans la troisième et dernière étape de raffinement, nous ne transformons que les modèles des tâches du NIVEAU_2 par l'application des transformations $\mathcal{T}_{11}$ et $\mathcal{T}_{12}$.

$$\forall t \in Task : \ \mathsf{PST}_t^3 = \mathcal{T}_{12}(\mathcal{T}_{11}(\mathsf{PST}_t^2))$$

Le schéma de notre démarche de raffinement est repris dans la figure FIG.4.11. Il reprend tous les niveaux : les modèles et transformations et les paramètres pris en compte à chaque niveau. Le modèle obtenu par la troisième étape de raffinement correspond à la plate-forme construite par le processus de projection. À partir de cette plate-forme, une concrétisation de bus partagé est possible par le raffinement de comportement des interfaces et arbitre de bus jusqu'à la spécification de protocole concret de celui-ci. De plus, d'autres raffinements peuvent être réalisés, tels que la concrétisation des données échangées, l'explicitation des opérations de traitement et la génération du code exécutable des tâches et la synthèse matérielle.

4.4.2 Génération des modèles LTS

Les outils d'analyse de raffinement que nous utilisons portent sur les automates. Pour analyser nos modèles, nous générons donc à partir des modèles pomsets des tâches et des canaux leurs LTSs (automate étiqueté) correspondants par l'application de l'algorithme dont le schéma est décrit dans le tableau TAB. 3.4.

Le modèle automate du système du NIVEAU_0, NIVEAU_1 et NIVEAU_2, noté LTS^i avec $i \in \{0, 1, 2\}$, est obtenu par produit synchronisé (désigné par $||$) des modèles automates correspondants au modèles des tâches et des canaux, notés LTS_t^i pour une tâche t et LTS_c^i pour un canal c. Nous construisons le modèle global du système comme suit :

$$\forall i \in \{0, 1, 2\}.\ \mathsf{LTS}^i = ((||_{t \in Task}\ \mathsf{LTS}_t^i) \ || \ (||_{c \in Channel}\ \mathsf{LTS}_c^i))$$

Et le modèle automate du système dans le NIVEAU_3, noté LTS^3, est obtenu par produit synchronisé des modèles automates des tâches LTS_t^3, des canaux LTS_t^2 du NIVEAU_2, des modèles des arbitres $\mathsf{LTS}_{\mathcal{A}}$, ainsi que les modèles d'interfaces $\mathsf{LTS}_{\mathcal{I}}$ pour chaque interface. Après l'étape de transformation des pomsets des tâches en automates et la spécification des modèles automates des interfaces et arbitres, nous construisons le modèle global du système comme suit :

$$\mathsf{LTS}^3 = ((||_{t \in Task}\ \mathsf{LTS}_t^3) \ || \ (||_{c \in Channel}\ \mathsf{LTS}_c^2) \ || \ (||_{\mathcal{I} \in Interface}\ \mathsf{LTS}_{\mathcal{I}}) \ || \ (||_{\mathcal{A} \in Arbiter}\ \mathsf{LTS}_{\mathcal{A}}))$$

Notons que dans notre modèle, il y a possibilité d'avoir plusieurs arbitres ; en effet, ce cas peut correspondre à une situation où un bus peut être constitué par une hiérarchie de bus dont chacun possède son propre arbitre.

4.4.3 Étude de raffinement

La phase de vérification de préservation de propriétés peut être réalisée à chaque étape de raffinement après génération de modèle LTS, ou encore, après la phase d'application des trois étapes de raffinement et la génération des modèles LTSs des différents niveaux. La preuve est décomposée en deux parties.Premièrement, les propriétés linéaires sont vérifiées

sur le LTS^0 en utilisant les techniques de model-cheching. La deuxième étape consiste à l'étude de préservation de propriétés en utilisant l'analyse de raffinement par la sémantique de trace infinie, noté $\sqsubseteq$, entre chaque deux niveaux consécutifs.

$$\mathsf{LTS}^0 \sqsubseteq \mathsf{LTS}^1 \sqsubseteq \mathsf{LTS}^2 \sqsubseteq \mathsf{LTS}^3$$

Puisque l'alphabet des modèles automates des niveaux n'est pas le même, nous procédons à une étape de masquage des actions de chaque niveau i selon le niveau précédent $i-1$, ce qui permet de comparer des modèles ayant le même alphabet et la vérification d'inclusion des traces complète infinie.

4.5 Conclusion

Ce chapitre décrit notre démarche de raffinement de communication dans les systèmes embarqués. Il décrit les différentes transformations à utiliser pour passer d'une étape à une autre dans la démarche proposée. Cette démarche décompose le raffinement de communication en trois étapes de raffinement successives :

- Raffinement de granularité de données ;
- Raffinement de gestion des canaux ;
- Et raffinement par introduction de bus abstrait.

Chaque étape de raffinement est caractérisée par l'application successive d'une série de transformations introduites dans un ordre déterminé. Ces transformations à appliquer sur le modèle de l'application introduisent graduellement les contraintes d'architecture et de projection. Une caractérisation formalisée de ces derniers est donnée sous forme de modèles pomsets, ainsi que sous forme d'automates correspondants afin d'analyser la préservation de propriétés de comportement.

Pour une validation de notre démarche, nous l'appliquons à une étude de cas d'un appareil photo numérique. Le chapitre suivant décrit l'utilisation de notre approche sur cette étude de cas en reprenant tous les niveaux présentés dans ce chapitre et les outils et techniques utilisés pour prouver le raffinement dans les différents niveaux générés.

Troisième partie

Étude de cas et conclusion

Chapitre 5

Étude de cas

Sommaire

Pour montrer la faisabilité de notre démarche de raffinement, nous l'avons appliquée sur une étude de cas issue d'un exemple réel. Nous avons choisi de modéliser le fonctionnement d'un appareil photo numérique exécutant l'algorithme de compression JPEG et le stockage final d'images. L'application choisie est basée sur l'exemple décrit et implémenté en SYSTEMC par Vahid et Givargis [110].

Dans ce chapitre, nous allons donc appliquer sur cette étude de cas les différentes transformations définies dans le chapitre précédent depuis le modèle du NIVEAU_0 de l'application décrite en TML jusqu'au NIVEAU_3. À chaque niveau, nous construisons les modèles des composants du système et nous vérifions la relation de raffinement du modèle produit (concret) et du modèle niveau précédent (abstrait). La relation de raffinement utilisée est la relation d'inclusion de traces complètes infinies. Nous montrons également le gain de temps en terme de vérification de propriétés fonctionnelles.

5.1 Exemple d'application : Appareil photo numérique

La fonctionnalité de l'appareil photo numérique consiste en la capture d'une image et son transfert vers un ordinateur personnel ou une carte externe. Tout d'abord, l'appareil photo numérique doit capturer, traiter et stocker des images dans une mémoire interne. Cette tâche est lancée lorsque l'utilisateur appuie sur le déclencheur de l'appareil pour prendre une photo. Après que l'image ait été capturée dans une forme numérique, la correction de l'image est effectuée afin de rectifier les erreurs liées à la capture. L'image corrigée est ensuite compressée ; ce processus se compose de deux étapes : l'application de la transformation de cosinus discrète (DCT), la quantification et la compression Huffman. Enfin, l'image est stockée dans la mémoire interne de l'appareil. Ce dernier doit être en mesure de transférer par la suite les images stockées vers une carte externe.

5.1.1 Les principaux modules de l'application

Dans leur code SystemC [110], Vahid et Givargis ont décomposé l'application en cinq modules, chaque module exécute un fonctionnement spécifique.

1. **Module CCD.** Module générateur d'image, il est responsable de la capture d'une image. Cela est accompli par deux fonctions, la capture de l'image et l'envoi de celle-ci. Lorsque le module CCD reçoit une demande pour une capture d'image, il lit la valeur de chaque pixel et le charge dans son espace mémoire. Enfin, lorsque la capture est achevée la procédure d'envoi de l'image au composant CCDPP est réalisée.
2. **Module CCDPP.** Module responsable de l'ajustement de la luminosité de chaque pixel. Après que l'image soit reçue depuis le module CCD, elle est transférée au module CNTRL après le traitement.
3. **Module CODEC.** Module appliquant la première partie de l'algorithme de compression en traitant des blocs de pixels. Le module CODEC est constitué de trois fonctions : réception des blocs des pixels à partir du CNTRL, exécution d'une partie de l'algorithme de compression (transformation DCT) et enfin l'envoi de bloc de pixel au CNTRL.
4. **Module CNTRL.** Module effectuant l'envoi d'image au module CODEC pour une première étape de compression. Après la réception des données du module CODEC, ce module effectue la deuxième partie de compression (la quantification et la compression Huffman). Enfin, le contrôleur envoi l'image au module TRANS.
5. **Module TRANS.** Module responsable du transfert des données (images) au périphérique externe.

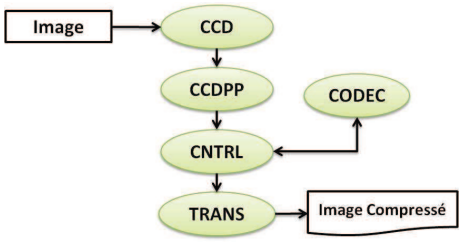


Fig. 5.1 – Schéma de relation entre les différents modules

La figure FIG. 5.1 montre les cinq modules constituant l'application et le flux de données échangées entre eux. Dans le codage SYSTEMC de [110], les auteurs ont abstrait les codes des traitements du codage Huffman ainsi que la transformation de l'image en des flux binaires JPEG et les sorties de tous les modules restent des flux de pixels, ce qui permet de se focaliser sur la manipulation des pixels tout au long du processus de traitement et de transfert.

5.1.2 Description TML de l'application

Nous proposons un modèle TML de cette application. Chaque module est codé naturellement par une tâche TML, chaque liaison entre modules est codée par un canal de données reliant les tâches correspondant aux modules. Nous abstrayons la granularité de données manipulées par les tâches, nous choisissons la manipulation des images et de blocs d'image. Les canaux de données doivent respecter le sens du flux de transfert de données échangées entre les différents modules, leurs types et tailles devrait respecter le non-écrasement d'information décrit par l'application initial. Nous avons donc choisi pour cette application des canaux de données BR-BW (bloquants en lecture et en écriture), d'un espace d'une case de type de l'objet stocké (image ou bloc). Les canaux de données de même type avec des tailles plus grandes peuvent être aussi choisis de même que les canaux infinis (de type BR-NBW). Par contre, les canaux de type NBR-NBW donneraient lieu à un écrasement des données.

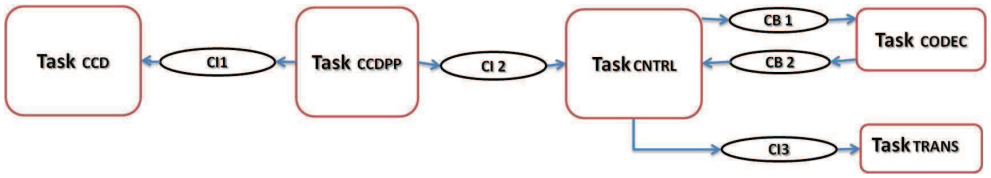


FIG. 5.2 – Représentation graphique de l'application en TML.

Une représentation graphique du modèle TML est donnée par la figure FIG. 5.2 ; les tâches portent les mêmes noms que les modules correspondants. Les liens entre les tâches sont représentés par des canaux. Par exemple, le lien entre la tâche CODEC et la tâche CNTRL est représenté par deux canaux différents CB1 et CB2 car l'échange entre les deux modules est bidirectionnel et les canaux sont unidirectionnels. La figure FIG. 5.3 montre le code de l'application, notamment le code des tâches qui sont des processus qui s'exécutent d'une manière cyclique.

Le module CCD capture des images et réalise le transfert (WRITE CI1) vers la tâche CCDPP. Le module CCDPP récupère les données produites (READ CI1) sur le canal CI1 et exécute l'ajustement sur l'image récupérée (EXEC) avant de la transférer (WRITE CI2) sur le canal CI2 vers le contrôleur CNTRL. Le module CODEC applique l'algorithme de codage des données récupérées depuis le contrôleur. Le module CNTRL récupère les données envoyées depuis CCDPP et réalise des échanges vers et depuis CODEC par blocs de données avant que celui-ci exécute la quantification. Le module TRANS modélise le processus de transfert en envoyant une image vers la mémoire externe. La variable *N*, dans

```
CHANNEL  CI1  Image1 BRBW  1  CCD   CCDPP
CHANNEL  CI2  Image2 BRBW  1  CCDPP  CNTRL
CHANNEL  CI3  Image2 BRBW  1  CNTRL  TRANS
CHANNEL  CB1  Bloc   BRBW  1  CNTRL  CODEC
CHANNEL  CB2  Bloc   BRBW  1  CODEC  CNTRL

TASK CCDPP{
   read CI1
   exec
   write CI2
   }
TASK CCD {
   write CI1
   }
TASK TRANS{
   read CI3
   }

TASK CNTRL{
   read CI2
   Repeat N times
        write CB1
        read CB2
        exec
   Endrepeat
   write CI3
   }

TASK CODEC
   {
   Repeat N times
        read CB1
        exec
        write CB2
   Endrepeat
   }
```

FIG. 5.3 – Code de l'application en TML.

le code de la tâche CNTRL et la tâche CODEC, est un paramètre de boucles qui représente le nombre de blocs contenu dans une image : une image est transférée et traitée entre la tâche CNTRL et CODEC bloc par bloc, ce paramètre vaudra 2 pour une image contenant deux blocs. La généralisation de l'algorithme sur des images de taille plus importante est possible par la modification des types de données et de ce paramètre *N*. Justement, pour les différentes expérimentations que nous avons réalisées, nous avons fait varier les types de données, le paramètre *N* ainsi que les paramètres de l'architecture choisie.

5.1.3 Architecture et partitionnement

Pour construire le système à partir de la description TML, il est nécessaire de déterminer le partitionnement des tâches et des canaux sur les éléments de l'architecture. Nous avons choisi une architecture composée de cinq éléments de calculs PE1, PE2,..., PE5, deux éléments de stockage SE1, SE2. Les éléments PE1, PE2 et SE2 sont interconnectés entre eux via l'élément de communication CE2, les éléments PE2, PE3, PE4, PE5 et SE1 sont interconnectés entre eux via le bus CE1 (voir la figure FIG. 5.4).

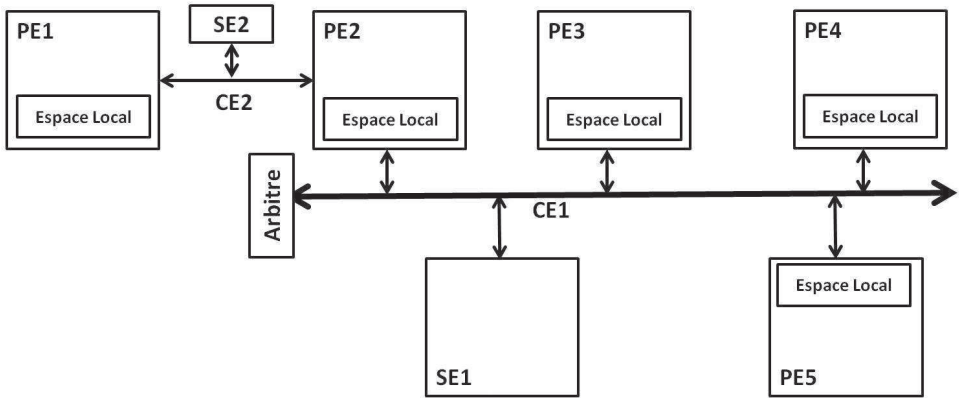


FIG. 5.4 – Architecture proposée pour l'exécution de l'application.

La figure FIG. 5.5 présente le choix retenu de partitionnement de l'application sur cette architecture. Nous avons choisi la projection des tâches CCD, CCDPP, CNTRL, CODEC et TRANS respectivement sur les unités de calcul PE1, PE2, PE3, PE4 et PE5.

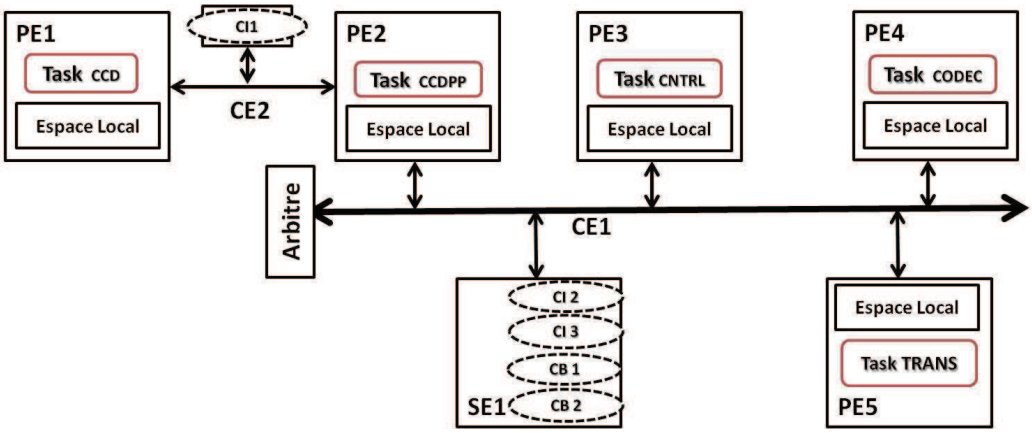


FIG. 5.5 – Plate-forme = Application + Architecture.

Le canal CI1 est projeté sur une zone mémoire de l'élément de stockage SE2, le cheminement des données depuis la tâche CCD vers le canal CI1 jusqu'à la tâche CCDPP est réalisé au travers le bus CE2, les deux éléments de calcul PE1 et PE2, sur les quels les deux tâches CCD et CDDPP sont projetées, sont les deux maîtres du bus. Les canaux CI2, CI3, CB1 et CB2 sont projetés sur des zones mémoires disjointes de l'élément de stockage SE1, le cheminement des données entre les différents PEs des tâches est réalisé au travers du bus CE1. Le but de notre expérimentation est de construire un modèle concret de l'application dans lequel les propriétés fonctionnelles de l'application sont préservées. Ce modèle est construit par notre démarche de raffinement introduite dans le chapitre 5.

Pour la validation de notre démarche, nous l'avons appliquée sur le modèle de l'étude de cas avec différentes tailles d'images et de bloc d'images. L'explication de la démarche se fera dans la suite sur des paramètres de taille réduite, mais l'expérimentation se fera aussi sur des tailles plus grandes. Le tableau TAB. 5.1 résume les différentes tailles d'images, des blocs d'image ainsi que les paramètres d'architecture choisis et utilisés pour les expérimentations. Nous avons spécifié pour un élément de calcul le nombre de cœurs, d'interfaces et la taille de mémoire interne. Concernant les éléments de stockage, nous avons choisi de spécifier des tailles suffisamment grandes pour la projection des canaux de données raffinés. Pour la première expérimentation, nous avons choisi de modéliser des types de données de petites tailles, les données récupérées par CCDPP seront de taille 2×3 UNIT, les données produites par cette tâche seront de 2×2 UNIT, un bloc de données échangé entre la tâche CNTRL et la tâche CODEC est de taille de 2 UNIT. Nous avons de plus spécifié pour chaque élément de calcul un cœur d'exécution et un nombre de mémoire interne spécifique pour la gestion des données transférées depuis et vers les canaux.

5.1.4 Analyse de propriétés

Tout au long du processus de raffinement de l'application, nous avons cherché à vérifier la préservation des propriétés linéaires de sûreté et de vivacité, ces propriétés portant

	TML					PE1			PE2			PE3			PE4			PE5			SE1	SE2
	Image1	Image2	Image3	Bloc1	Bloc2	Coeur	Interface	Memoire	Coeur	Interface	Memoire	Coeur	Interface	Memoire	Coeur	Interface	Memoire	Coeur	Interface	Memoire	Espace	Espace
Cas 1	2×3	2×2	2×2	2	2	1	1	6	1	2	4	1	1	2	1	1	2	1	1	2	$Size(CI1)$	$\sum_{c \in \{CI2, CI3, CB1, CB2\}} Size(c)$
Cas 2	16×18	16×16	16×16	64	64	1	1	16×18	1	2	18	1	1	16×16	1	1	64	1	1	16×16	$Size(CI1)$	$\sum_{c \in \{CI2, CI3, CB1, CB2\}} Size(c)$
Cas 3	32×34	32×32	32×32	64	64	1	1	32×34	1	2	34	1	1	32×32	1	1	64	1	1	32×32	$Size(CI1)$	$\sum_{c \in \{CI2, CI3, CB1, CB2\}} Size(c)$

TAB. 5.1 – Tableau récapulatif des différentes types et tailles en UNIT de données échangées ainsi que les paramètres des composants architecturaux choisis

sur des traces d'exécution qui décrivent le comportement du système. Spécifiquement, nous avons cherché à vérifier des propriétés garabtissant : le non-blocage du système, l'ordonnancement des actions des tâches et la non-violation de la taille maximale des canaux.

Nous avons décrit ces propriétés en utilisant le langage MCL[78] (pour Model Checking Language) qui est un langage d'entrée pour l'outil d'évaluation des propriétés de la boite à outils CADP[52]. Ce langage permet de décrire des propriétés linéaires et arborescentes par des prédicats, des expressions régulières et des opérateurs de point fixe portant sur des actions paramétrées supportant même les données. Des exemples de propriétés de sûreté que nous avons analysées sont :

- L'absence des états de blocage dans le système (deadlock freedom).

$$P0 : \ [true^*] < true > \ true$$

- On ne peut pas lire dans le canal avant d'y avoir écrit au moins une donnée. Sur le canal CB1 cette propriété est spécifiée par :

$$P1 : \ [true^* .(\neg \ write(CB1))^* \ . \ read(CB1)] \ false$$

- Le nombre d'écritures sur un canal ne dépasse jamais la taille de celui-ci. Lorsqu'un canal est chargé le système n'effectue pas des nouvelles écritures sur celui-ci jusqu'à l'exécution d'au moins une lecture sur celui-ci. Cette propriété est exprimée sur le canal CB1 de taille deux par :

$$P2 : \ [true^* \ . \ write(CB1) \ . \ (\neg \ read(CB1))^* \ . \ write(CB1) \ . \ (\neg \ read(CB1))^* \ . \ write(CB1)] \ false$$

(pas trois écritures successives non entrelacées avec au moins une lecture).

Aussi, nous avons spécifié et analysé des propriétés de vivacité :

- L'absence des cycles infinis d'exécution des nouvelles actions (livelock freedom).

$$P3 : \ [true^*] \ \mu \ X. \ [\tau] \ X$$

- Le système revient à l'état initial quelque soit l'état du système. Concrètement, le système doit inévitablement exécuter dans le futur l'action WRITE sur la tâche CCD :

$$P4 : \ [true^*] \ \mu \ X. \ (< true > true \ \wedge \ [\neg \ write(CI1)] \ X)$$

- Une écriture sur un canal est forcément suivie dans le futur par une lecture. Cette propriété est exprimée sur le canal CI1 par :

$$P5 : \ [true^* \ . \ write(CI1)] \ \mu \ X. \ (< true > true \ \wedge \ [\neg \ read(CI1)] \ X)$$

- Une tâche est infiniment souvent exécutée. Cette propriété est exprimée sur la tâche CCDPP par :

$$P6 : \ [true^* \ . \ read(CI1) \ . \ true^* \ . \ exec1 \ . \ true^* \ . \ write(CI2)] \ \mu \ X.(< true > \ true \wedge [\neg \ read(CI1)] \ X)$$

- Une capture d'une image finit toujours par un transfert vers un support extérieur. Cette propriété est exprimée par :

$$P7 : \ [true^* \ . \ write(CI1)] \ \mu \ X \ .(< true > true \ \wedge \ [\neg \ read(CI3)] \ X)$$

Une formule sous la forme [R]F exprime la nécessité, avec R une expression régulière et F une formule. Cette formule est satisfaite par un état de l'automate si et seulement si chaque séquence de transition de sortie de cet état vérifiant l'expression régulière R, elle doit conduire à un état satisfaisant la formule F. Une formule sous la forme $< R > F$ exprime la possibilité : Elle est satisfaite par un état de l'automate si et seulement s'il existe une certaine séquence de transition de sortir de cet état qui répond à la formule régulière R et conduit à un état satisfaisant la formule F. L'opérateur μ permet la description des propriétés par un point fixe.

5.2 Démarche de raffinement des canaux de communication

Pour montrer la validité de l'approche proposée, nous avons expérimenté d'abord avec les paramètres de premier cas du tableau TAB. 5.1 et nous avons suivi les différentes étapes de raffinement décrites dans le chapitre précédent : le raffinement de granularité de données, le raffinement de la gestion des canaux et l'introduction de bus abstrait. Nous avons utilisé le modèle POMSET pour la transformation des tâches et des canaux de l'application vers des niveaux d'abstraction plus bas et le modèle LTS correspondant pour l'étude de propriétés fonctionnelles et leur préservation.

Nous commençons par construire le modèle POMSET de l'application, de chaque tâche et de chaque canal, et nous générons leurs LTSs respectifs. Les transformations sur les modèles sont effectuées manuellement en suivant les règles définies dans le chapitre 5, la construction des modèles globaux et la vérification de raffinement des modèles sont réalisées automatiquement, en utilisant l'outil CADP. La figure FIG. 5.6 décrit le modèle POMSET obtenu par l'application de la fonction $[\![.]\!]_{Task}$ sur le code de la tâche CCDPP et le modèle LTS correspondant obtenu par l'application de l'algorithme de traduction des pomsets vers les LTSs décrites dans le tableau TAB. 3.4. Le modèle LTS décrit l'exécution cyclique de la tâche, à partir de l'état initial (état 0) la tâche lit sur le canal CI1 (?READ(CI1)), puis fait un calcul local (EXEC) et enfin écrit le résultat sur le canal CI2 (!WRITE(CI2)).

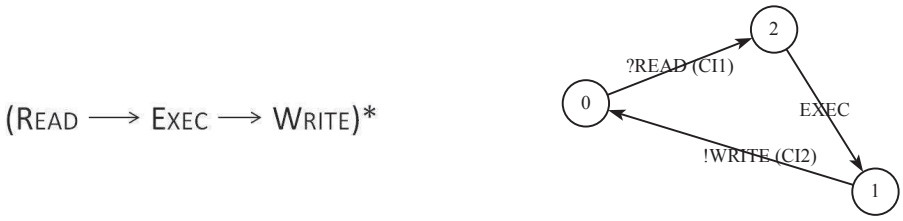


FIG. 5.6 – Modèles POMSET et LTS correspondant de la tâche CCDPP.

Nous construisons le LTS global par produit synchronisé des LTSs des composants tâches et canaux. Le modèle global LTS de l'application du NIVEAU_0 est donné par la figure FIG. 5.9, les tâches et les canaux TML sont représentés par des boîtes contenant les LTSs représentant leur comportement respectif. Les points sur les boîtes représentent les actions visibles du composant. Les flèches entre les boîtes représentent la synchronisation entre les LTSs sur les actions de communication. Par exemple, la tâche CCD se synchronise

avec le canal CI1 par l'envoi d'écriture sur le canal (!WRITE(CI1)) et la réception de l'écriture par le canal CI1 (?WRITE(CI1)), cette synchronisation est nommée par l'action WRITE1. Le modèle du NIVEAU_0 de l'application, noté M^0, comporte 336 états et 872 transitions (tableau TAB. 5.3 page 126).

5.2.1 Premier raffinement : Changement de granularité de données

Pour cette première étape de raffinement, nous avons choisi la transformation des unités de type IMAGE et de type BLOC de l'application en des blocs de données d'unité de type UNIT. La correspondance entre les différentes granularités des données du premier cas de raffinement est donnée par le tableau TAB. 5.2, la granularité des données de l'application, notée UT_{TML}, est transformée en données d'unités raffinées, notée UT_{DR}.

UT_{TML}	UT_{DR}
IMAGE1	6 UNIT
IMAGE2	4 UNIT
IMAGE3	4 UNIT
BLOC1	2 UNIT
BLOC2	2 UNIT

TAB. 5.2 – Transformations de données choisies.

Les canaux de données du modèle du NIVEAU_0 sont transformés en des canaux de données de granularité fine par la transformation 1. Par exemple, le canal CI1 est transformé en un canal de taille de six cases. La figure FIG. 5.7 montre le modèle LTS de ce canal raffiné.

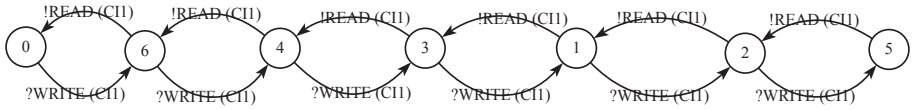


FIG. 5.7 – Modèle du canal CI1 du NIVEAU_1

Les modèles de tâches sont aussi transformés. Les actions des tâches sont transformées en des actions manipulant la nouvelle granularité de données. Les modèles des tâches du NIVEAU_1 sont construits par application successive des étapes de transformations $\mathcal{T}_1$, $\mathcal{T}_2$, $\mathcal{T}_3$, $\mathcal{T}_4$, $\mathcal{T}_5$ et $\mathcal{T}_6$. Nous donnons dans la figure FIG. 5.8 le LTS correspondant à la tâche CCDPP obtenue au NIVEAU_1. À gauche, le modèle généré directement, et à droite, le modèle obtenu après minimisation.

La construction du modèle de cette tâche se base sur les mêmes hypothèses que l'exemple du chapitre précédent de la tâche TASK2, à savoir, elle utilise la même dépendance, elle est dirigée par les mêmes contraintes d'espace mémoire locale, et du nombre de cœurs d'exécution. On obtient donc le même POMSET de la tâche TASK2 pour la tâche CCDPP. L'automate global du NIVEAU_1, noté M^1, comporte 12530 états et 43340 transitions.

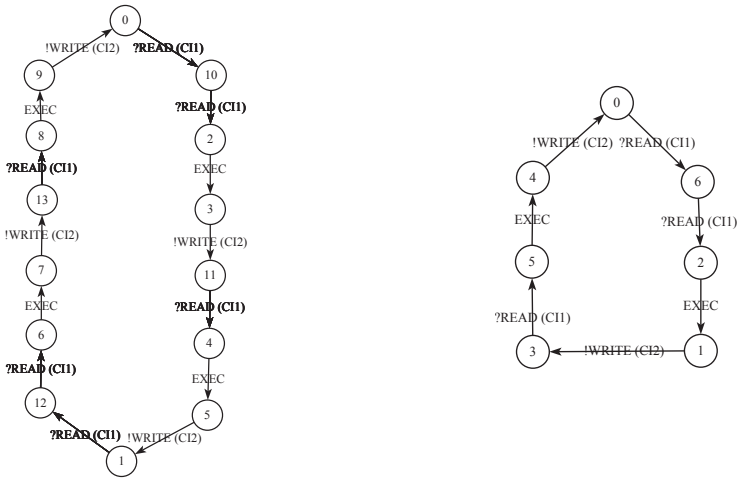


FIG. 5.8 – LTS généré de la tâche CCDPP (à gauche), LTS minimisé (à droite).

5.2.2 Second raffinement : Gestion des canaux

Dans cette étape, nous avons explicité la gestion de communication par le remplacement des protocoles des canaux abstraits par le protocole décrit dans la section. 4.2. Les canaux de données du modèle obtenu au NIVEAU_1 sont transformés en des canaux de données avec une gestion explicite d'accès aux données par l'application la transformation $\mathcal{T}_7$. La figure FIG. 5.10 montre la transformation du modèle de canal CI1. Remarquons que le LTS du canal est régulier, il a six niveaux qui représentent la taille du canal, il modélise aussi l'entrelacement des signaux de synchronisation et les signaux d'accès aux données des cases mémoire du canal.

Les modèles des tâches sont aussi transformés de façon à respecter le protocole choisi par l'application des transformations $\mathcal{T}_8$ et $\mathcal{T}_9$. Par exemple, pour la transformation de la séquence de lecture suivie d'exécution (READ→EXEC), nous appliquons la règle 5 donnée par le tableau Tab. 4.1 :

$$\begin{array}{c} \text{CHECK-DATA} \rightarrow \text{LOAD-DATA} \rightarrow \text{SIGNAL-ROOM} \\ \downarrow \\ \text{EXEC} \end{array}$$

Mais puisque les PEs considérés dans cet exemple sont mono-cœur, le modèle de la tâche se réduit à une trace d'actions totalement ordonnées. Nous avons contraint l'ordre précédent comme suit :

$$\text{CHECK-DATA} \rightarrow \text{LOAD-DATA} \rightarrow \text{EXEC} \rightarrow \text{SIGNAL-ROOM}$$

Ce choix d'ordre restreint l'accès à l'espace du canal en ne permettant la libération des ressources (SIGNAL-ROOM) qu'après la réalisation du calcul (EXEC), ce qui engendre la réduction des traces d'exécution de l'application. Nous donnons dans la figure FIG. 5.11 le modèle de la tâche CCDPP obtenu dans ce niveau. Le modèle global du NIVEAU_2, noté M^2, comporte 437782 états et 1646712 transitions.

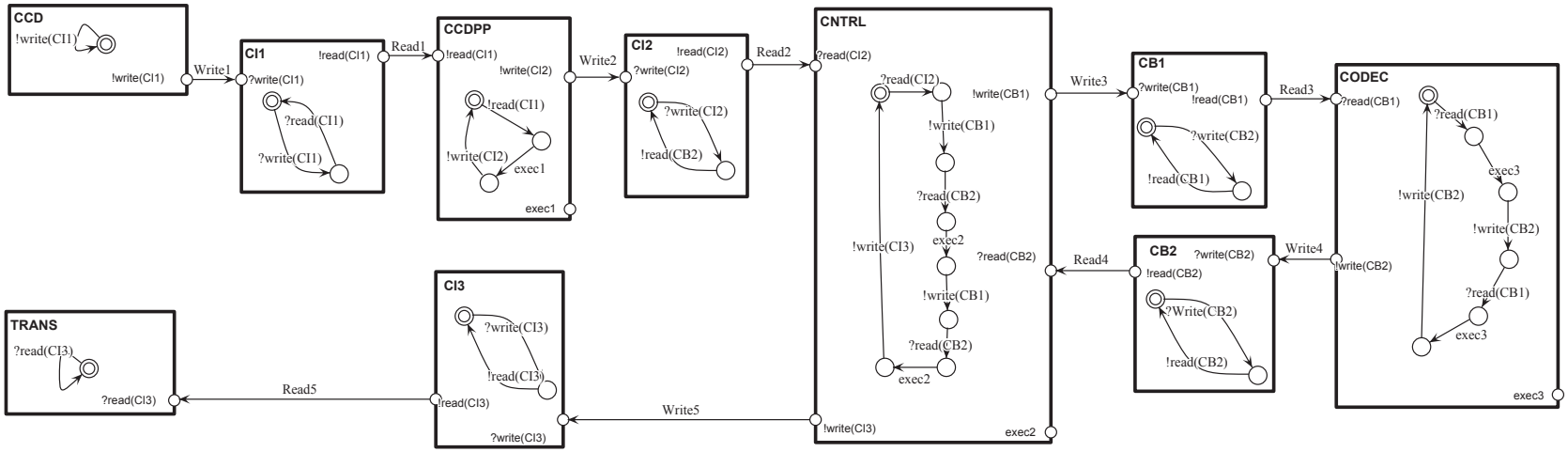


FIG. 5.9 – Modèle LTS de l'application dans le NIVEAU_0.

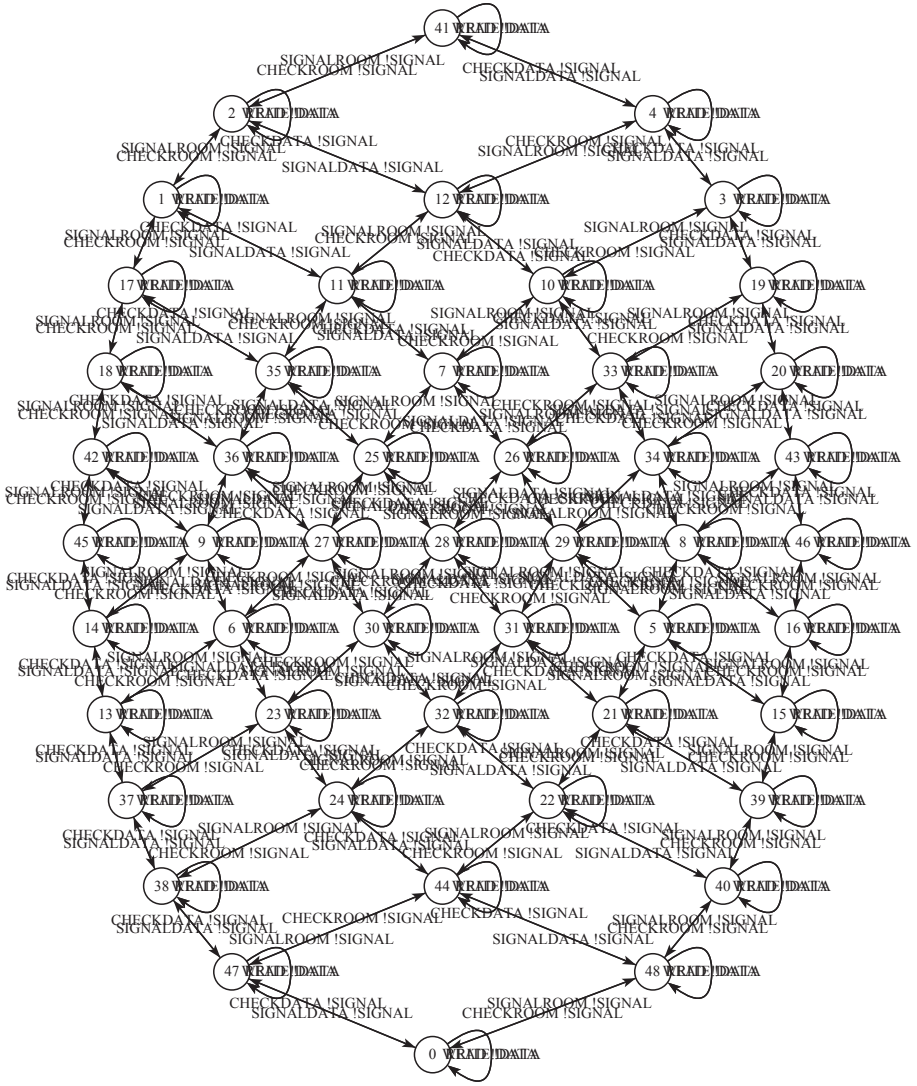


FIG. 5.10 – Modèle LTS du canal CI1 du NIVEAU_2.

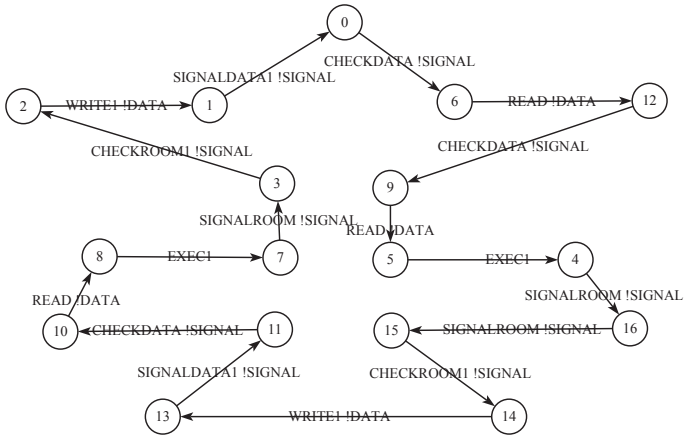


FIG. 5.11 – Modèle LTS de la tâche CCDPP du NIVEAU_2.

5.2.3 Troisième raffinement : Introduction de bus abstrait

Dans cette étape, nous avons concrétisé le protocole de communication entre les différents PEs par l'introduction du protocole de bus abstrait pour obtenir une plate-forme de NIVEAU_3. Nous avons associé à chaque élément de calcul une interface de communication pour l'adaptation de communication à travers le bus. Le bus de données de l'interface de communication peut échanger une donnée de deux UNITs. Et donc, nous choisissons aux interfaces des buffers de stockage des données échangées FIFOs de taille deux. Nous choisissons une politique d'arbitrage abstrait simple, elle donne une priorité égale à tous les maîtres de bus. Cette abstraction peut bien être raffinée vers le protocole "Round Robin". Le LTS représentant ce protocole est donné dans la figure FIG. 5.12.

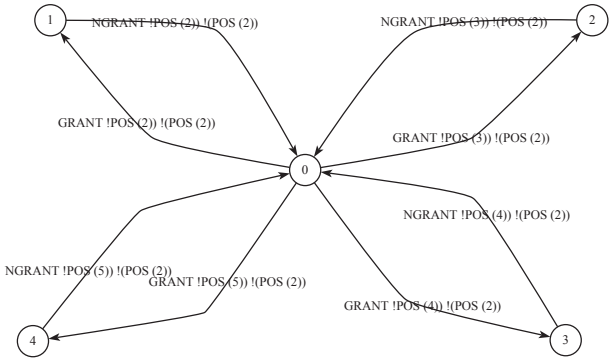


FIG. 5.12 – Modèle LTS de l'arbitre de bus CE2.

À partir de l'état initial l'arbitre donne la main par un signal GRANT soit au PE2, PE3, PE4 ou PE5. Une fois que l'arbitre a décidé de libérer le bus, il envoie un signal FREE. Le modèle de la plate-forme à ce niveau intégrera de nouveaux comportements qui sont les modèles des interfaces et des arbitres. La figure FIG. 5.13 montre les différents composants de la plate-forme, le point d'accès aux interfaces des éléments de stockage représente le point de partage entre les différents maîtres. Le modèle global de ce niveau, noté M^3, comporte 173546709 états et 472413097 transitions.

5.3 Analyse de préservation de propriétés

Dans le but d'analyser la préservation de propriétés du système, nous avons utilisé la boite à outils CADP [51, 52]. CADP permet principalement la vérification par model-checking des propriétés fonctionnelles et aussi la vérification d'équivalence et de raffinement entre les systèmes en utilisant plusieurs sémantiques d'équivalence et de raffinement telles que, la bisimulation forte et faible, la simulation arborescente et la simulation des traces. Ces outils acceptent en entrée plusieurs formats de description d'automates tels que, LOTOS, FC2 et FIACRE.

Nous avons décrit les différents composants dans les différents niveaux d'abstraction en utilisant le langage FIACRE [18]. Ce langage permet la description concurrente des systèmes par composition, il permet aussi l'intégration des contraintes temporelles et des contraintes de priorités sur les actions.

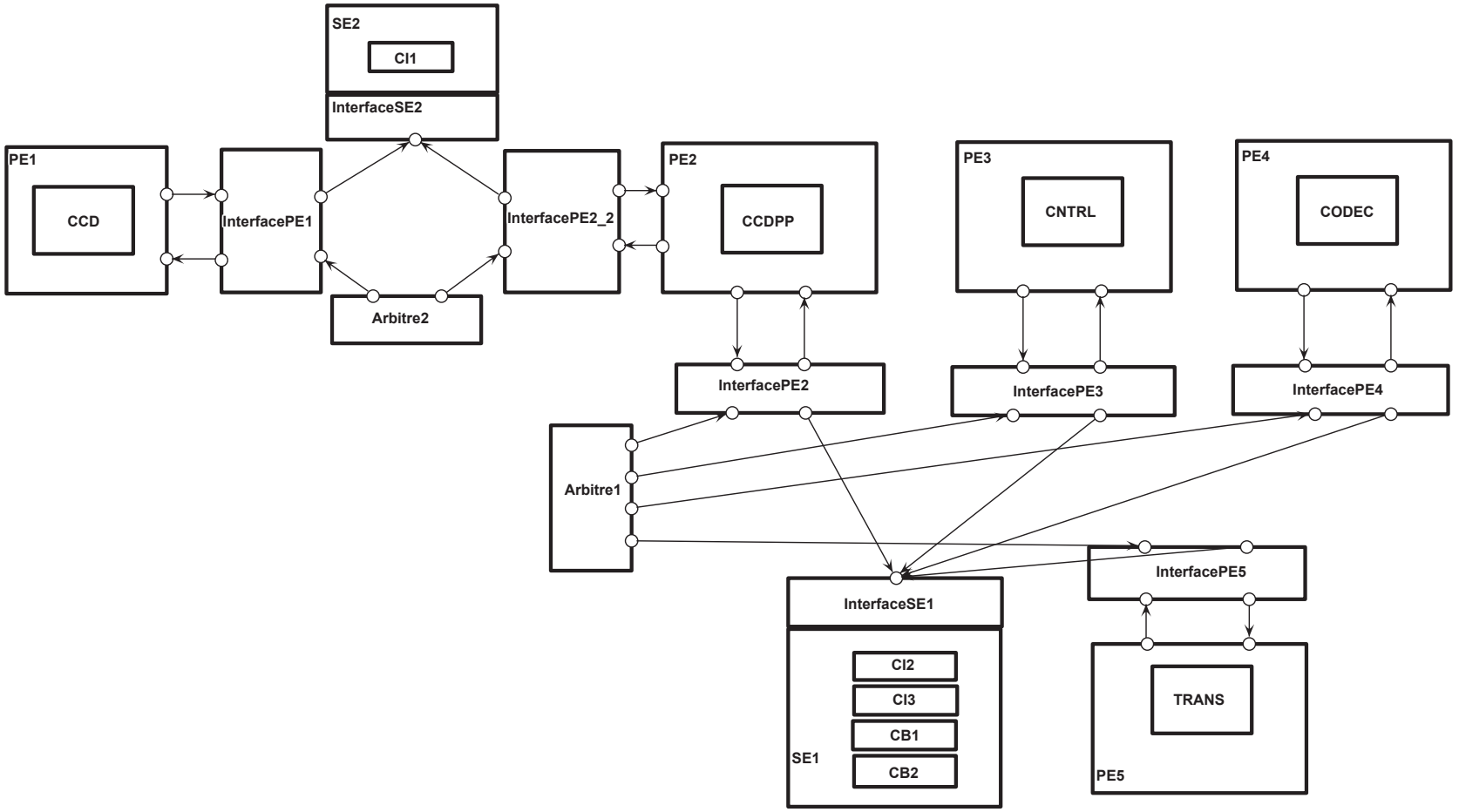


FIG. 5.13 – Modèle LTS du NIVEAU_3.

Pour construire le modèle global du réseau des automates communicants nous avons utilisé les fichiers de description des vecteurs de synchronisation écrits dans le langage EXP 2.0 [24]. Ce langage utilise des opérateurs de parallélisme, de composition, de masquage d'action (hiding operator), de renommage et de priorité.

Puisque les systèmes sur puces sont des systèmes qui ne doivent pas bloquer et qui doivent tourner infiniment, ces systèmes sont construits par des états finis reliés avec des chemins infinis, ce qui peut être décrit par des traces infinies d'exécution. L'étude de raffinement de traces infinies entre deux niveaux successifs nous permet de vérifier la préservation des propriétés de sûreté et de vivacité.

Comme l'alphabet des modèles des différents niveaux n'est pas le même, dans le niveau $i+1$ apparaissent de nouvelles actions et de nouvelles synchronisations n'appartenant pas à l'alphabet du modèle du niveau i, l'alphabet de chaque niveau i est inclus dans l'alphabet du niveau successeur $i+1$. Pour pouvoir vérifier l'inclusion des traces infinies, nous avons procédé à un masquage des actions non-observables sur le niveau i par un renommage par l'action τ sur le niveau $i+1$ par la fonction $Masq(M, B)$. Cette fonction remplace par τ, les actions de l'automate M qui ne sont pas dans l'alphabet B (voir définition 21).

Concrètement, nous cherchons à établir la relation de raffinement des traces infinies en utilisant la relation de préordre de trace de l'outil BISIMULATOR de CADP. Cette relation de préordre vérifie la non-introduction de nouvelles traces, pour vérifier l'inclusion de traces infinies, nous vérifions aussi la non-introduction de nouvel état de blocage (deadlock freedom) et la non-introduction de τ-cycles (livelock freedom) sur chaque niveau. En fait, le modèle abstrait du bus de NIVEAU_3 crée des livelock (τ-cycles), cela résulte du choix qu'un maître peut acquérir et libérer le bus sans réaliser un transfert. Le masquage des actions GRANT et FREE sur le modèle générera donc des τ-cycles ce qui cause la non-préservation des propriétés de vivacité. Ce problème de livelock est dû à l'abstraction des signaux de reprise et d'acquittement, ce qui peut être résolu par raffinement qui réintroduit ces derniers. Dans ce niveau de description de système, nous faisons la différence entre les τ-cycles qui possèdent une sortie d'action observable et les τ-cycles qui ne possède pas des sorties vers des actions observables. Nous n'acceptons que le premier cas. Concrètement, la résolution de ce problème se fait par l'élimination des τ-cycles par l'outil REDUCTOR de CADP (en utilisant la réduction *taucompression*). Notons que dans le second cas où les τ-cycles ne possède pas une sortie avec des transitions observables, alors l'élimination des τ-cycles construit un état de blocage qui est détecté par la propriété P0 de non-introduction d'état de blocage.

Dans la suite, nous discutons les différentes stratégies et résultats de preuve obtenus lors de l'application de notre démarche de raffinement sur les trois expérimentations.

5.3.1 Première stratégie de validation

La première stratégie d'étude de raffinement adoptée est de comparer le modèle du niveau $i+1$ avec celui du niveau i en masquant suivant l'alphabet du niveau précédent i. Soient $M^0 = \langle A^0, S^0, S^0_0, \rightarrow^0 \rangle$, $M^1 = \langle A^1, S^1, S^1_0, \rightarrow^1 \rangle$, $M^2 = \langle A^2, S^2, S^2_0, \rightarrow^2 \rangle$ et $M^3 = \langle A^3, S^3, S^3_0, \rightarrow^3 \rangle$ les différents LTSs obtenus respectivement au niveau 0, 1, 2 et 3 avec $A^0 \subseteq A^1 \subseteq A^2 \subseteq A^3$. Nous avons vérifié la satisfaction des trois relations :

1. $M^0 \sqsubseteq Masq(M^1, A^0)$
2. $M^1 \sqsubseteq Masq(M^2, A^1)$
3. $M^2 \sqsubseteq Masq(M^3, A^2)$

La figure FIG. 5.14 montre le schéma complet de génération des modèles et des preuves réalisées par cette stratégie. Le processus d'analyse de raffinement peut commencer après la génération de LTS global de chaque niveau à partir des descriptions POMSET des composants et le processus de masquage des étiquettes sur chaque niveau i par rapport au niveau $i-1$.

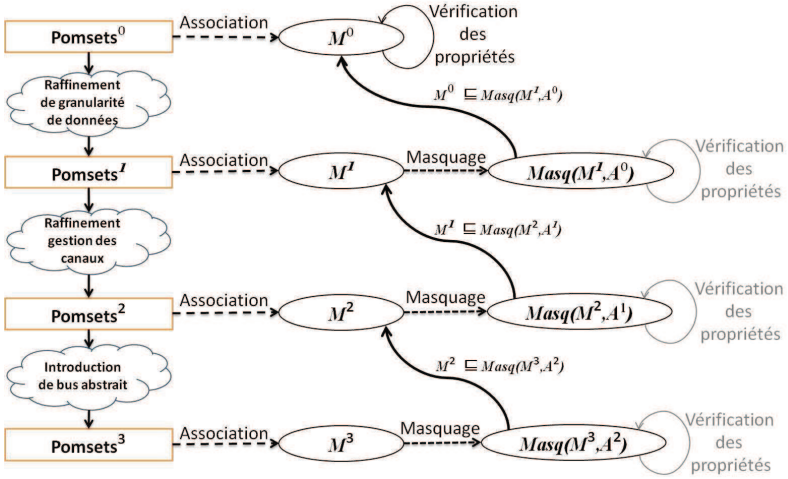


FIG. 5.14 – Schéma de génération et vérification des LTSs (cas1).

Les résultats de l'expérimentation avec cette stratégie sont résumés dans le tableau TAB. 5.3. Le tableau montre que le système concret a plus d'états et de transitions que le système abstrait. Cela résulte de l'ajouts de nouvelles actions et de nouvelles synchronisations. Cependant, nous avons bien une inclusion de trace entre les différents niveaux : $M^0 \sqsubseteq M^1 \sqsubseteq M^2 \sqsubseteq M^3$. Le tableau donne aussi la durée mise par l'outil pour vérifier le raffinement. On remarque que plus le système est grand plus le temps de vérification est plus important. Grâce à la vérification du raffinement plus le diagnostic de CADP qu'à chaque niveau, il n'y a pas de deadlock ; nous pouvons déduire l'inclusion de traces infinies.

		NIVEAU_0	NIVEAU_1	NIVEAU_2	NIVEAU_3
♯État M^i		336	123.088	437.782	173.546.709
♯Transition M^i		872	431.622	1.646.712	472.413.097
Sûreté	Verification P0	$0,8s$	$13\ s$	$9\ mn$	$26\ mn$
	Verification P1	$0,5\ s$	$8\ \ s$	$5\ \ mn$	$16\ mn$
	Verification P2	$0,7\ s$	$9\ \ s$	$5\ \ mn$	$15\ mn$
Vivacité	Verification P3	$0,6\ s$	$16\ s$	$10\ mn$	$26\ mn$
	Verification P4	$0,9\ s$	$21\ s$	$13\ mn$	$41\ mn$
	Verification P5	$0,8\ s$	$14\ s$	$11\ mn$	$39\ mn$
	Verification P6	$1,2\ s$	$35\ s$	$17\ mn$	$55\ mn$
	Verification P7	$0,9\ s$	$21\ s$	$12\ mn$	$41\ mn$
Raffinement		$\times$	$8\ \ s$	$3\ \ mn$	$13\ mn$

TAB. 5.3 – Analyse du raffinement des niveaux de l'application (Cas 1).

Le tableau montre aussi les résultats de la vérification de l'ensemble des propriétés sélectionnées. Ces propriétés sont satisfaites sur les modèles à tous les niveaux, il n'y a pas de mystère car nous avons bien un raffinement entre les différents niveaux. Par contre, nous notons la différence considérable de temps mis par EVALUATOR pour la vérification de chaque propriété dans les différents niveaux, ce qui montre le bénéfice de notre démarche. Une fois notre méthodologie prouvée, le concepteur n'aura qu'à vérifier les propriétés dans le niveau_0, leurs vérification est garantie aux niveaux raffinés par construction.

Nous avons aussi appliqué notre démarche de raffinement sur une décomposition d'image en sous blocs de pixels de taille 16×16 UNIT et 32×32 UNIT (les cas 2 et 3 du tableau TAB. 5.1). Le passage à des images plus grandes a engendré un problème d'explosion combinatoire lors de la génération des LTSs globaux (en particulier pour les niveaux NIVEAU_2 et NIVEAU_3). Pour remédier à ce problème, nous avons appliqué une seconde stratégie de validation.

5.3.2 Seconde stratégie de validation

Nous avons construit le LTS global de chaque niveau par synchronisation avec masquage des actions qui n'existent pas sur l'alphabet du NIVEAU_0, ce qui permet d'avoir moins de transitions observables et donc la possibilité de générer les modèles globaux des niveaux complexes grâce à la minimisation du système. Nous avons vérifié la satisfaction des trois relations :

1. $M^0 \sqsubseteq Masq(M^1, A^0)$
2. $Masq(M^1, A^0) \sqsubseteq Masq(M^2, A^0)$
3. $Masq(M^2, A^0) \sqsubseteq Masq(M^3, A^0)$

La figure FIG. 5.15 présente le schéma de la seconde stratégie de preuve d'inclusion des traces entre les différents niveaux. Nous avons généré directement les modèles abstraits des niveaux en gardant que les actions du NIVEAU_0 observables dans le but de répondre au problème d'explosion combinatoire.

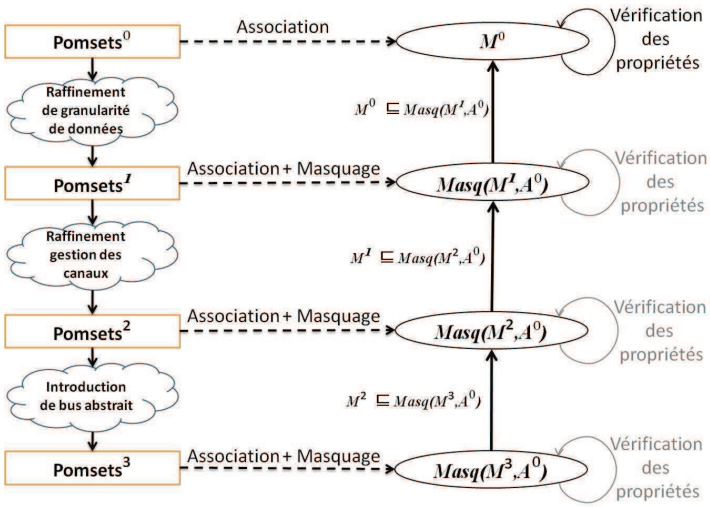


FIG. 5.15 – Schéma de génération et vérification des LTSs (cas2 et Cas3).

Contrairement à la première stratégie de preuve, cette stratégie de preuve permet uniquement de s'assurer de la préservation des propriétés du NIVEAU_0 à travers le raffinement, c'est-à-dire, elle ne permet pas de s'assurer que les propriétés sur des actions du niveau $i-1$ sont préservées dans le niveau i. Comme la première stratégie, la vérification d'inclusion des traces est réalisée sur chaque niveau $i+1$ selon le niveau précédent i et pas directement avec le modèle initial, cela permet de bien suivre la réduction des traces toutes au long de processus de raffinement et de s'assurer qu'un niveau $i+1$ contient moins de traces qu'un niveau i.

La vérification de préservation de propriétés linéaires est établie pour les deux expérimentations. Les résultats de vérification des propriétés par raffinement pour les deux expérimentations sont résumés dans les deux tableaux TAB. 5.4 et TAB. 5.5, le temps de vérification de l'inclusion des traces entre deux modèles successifs et le temps de vérification des propriétés pour chaque niveau. Ces résultats renforcent les résultats précédents sur les avantages de la stratégie de vérification par raffinement (comparer la somme des temps de colonne du NIVEAU_0 plus la somme des temps de ligne "raffinement" plus la somme des lignes de P0 avec la somme du temps de la colonne du NIVEAU_3 plus la somme du temps de vérification des propriétés du NIVEAU_0).

En comparant les temps des deux tableaux avec le tableau TAB. 5.3, on remarque le gain concernant la taille des LTSs globaux ainsi que le temps de vérification même si les modèles des deux dernières expérimentations sont plus importants. Cela est dû d'un côté au masquage des actions lors de la génération des LTSs et aussi au choix des tailles des canaux de communication lors de transformations. En fait, nous avons gardé la même taille des canaux spécifiée par la première expérimentation dans les deux nouvelles expérimentations ; sur la première expérimentation, la taille des canaux raffinés est égale à la taille de l'image, sur les deux autres d'études la taille des canaux ne représente qu'une petite partie de taille de l'image, ce qui réduit considérablement les parallélismes entre les tâches sur les deux dernières expérimentations et ainsi réduire les chemins d'exécution possible du système.

		NIVEAU_0	NIVEAU_1	NIVEAU_2	NIVEAU_3
♯État M^i		814	25.382	72.494	593.744
♯Transition M^i		2.604	71.862	145.089	1.651.889
Sûreté	Verification P0	$1,1\ s$	$25s$	$1\ mn$	$10\ mn$
	Verification P1	$0,8\ s$	$16\ s$	$46\ s$	$6\ mn$
	Verification P2	$0,8\ s$	$16\ s$	$45\ s$	$6\ mn$
Vivacité	Verification P3	$0,8\ s$	$31\ s$	$2\ mn$	$13\ mn$
	Verification P4	$1,6\ s$	$40\ s$	$2\ mn$	$16\ mn$
	Verification P5	$1,2\ s$	$16\ s$	$45\ s$	$6\ mn$
	Verification P6	$2,1\ s$	$71\ s$	$4\ mn$	$28\ mn$
	Verification P7	$1,5\ s$	$30\ s$	$2\ mn$	$16\ mn$
Raffinement		$\times$	$18\ s$	$46\ s$	$19\ mn$

TAB. 5.4 – Analyse du raffinement des niveaux de l'application (Cas 2).

		NIVEAU_0	NIVEAU_1	NIVEAU_2	NIVEAU_3
♯État M^i		2.542	141.302	516.326	2.369.408
♯Transition M^i		7.212	419.758	1.038.037	6.593.033
Sûreté	Verification P0	$3\ s$	$3\ mn$	$11\ mn$	$40\ mn$
	Verification P1	$2\ s$	$2\ mn$	$5\ mn$	$25\ mn$
	Verification P2	$2\ s$	$2\ mn$	$6\ mn$	$25\ mn$
Vivacité	Verification P3	$2\ s$	$3\ mn$	$11\ mn$	$50\ mn$
	Verification P4	$5\ s$	$4\ mn$	$16\ mn$	$66\ mn$
	Verification P5	$4\ s$	$2\ mn$	$6\ mn$	$25\ mn$
	Verification P6	$7\ s$	$7\ mn$	$27\ mn$	$118\ mn$
	Verification P7	$5\ s$	$3\ mn$	$16\ mn$	$64\ mn$
Raffinement		$\times$	$3\ mn$	$25\ mn$	$75\ mn$

TAB. 5.5 – Analyse du raffinement des niveaux de l'application (Cas 3).

5.4 Conclusion

L'approche proposée dans cette thèse est appliquée dans ce chapitre sur un exemple d'application réel. Nous avons appliqué sur cet exemple toutes les transformations définies dans le chapitre 5. Leur mise en œuvre sur les modèles s'est faite manuellement et les modèles correspondants aux différents niveaux sont codés en FIACRE. L'établissement du raffinement est réalisé automatiquement en utilisant les outils EVALUATOR et BISIMULATOR de la boite à outils CADP. La vérification des propriétés illustre bien l'intérêt de notre approche. En effet, le temps de vérification des propriétés dans le modèle du NIVEAU_3 est plus grand que celui du NIVEAU_0 (croit exponentiellement avec le nombre d'états/transitions). Si on peut prouver que les compositions de transformations que nous proposons sont des raffinements pour des classes d'applications, le concepteur n'aura qu'à vérifier les propriétés dans le NIVEAU_0. Même avec la démarche d'analyse de raffinement actuel, nous rapportons une amélioration du temps de vérification.

Le résultat de l'expérimentation valide notre approche. Cette étude montre que par certains choix de transformation sur les canaux de communication, on peut garantir la préservation de propriétés. La démarche de transformation doit encore être automatisée (génération automatique des modèles FIACRE des LTSs des différents niveaux). Et de plus, la préservation de propriétés fonctionnelles reste à être prouvée mathématiquement pour toutes les applications d'entrées.

Le chapitre suivant résume les contributions et les perspectives futures de cette thèse.

Conclusion

La complexité des nouveaux systèmes embarqués force le concepteur à développer des modèles abstraits qui sont vérifiables facilement et permettent de prendre des décisions tôt dans le flot de conception. Ces modèles sont concrétisés en introduisant des détails graduellement jusqu'à l'implémentation. En revanche, les propriétés fonctionnelles établies sur les modèles abstraits ne sont par toujours garanties sur les modèles concrets.

Cette thèse propose une approche pour la conception de systèmes embarqués. Cette approche permet à la fois la construction et l'analyse incrémentales des systèmes développés. Elle permet de construire un système à partir d'un modèle d'application et d'un modèle d'architecture abstraits, puis par des étapes de projection et de raffinement, elle permet d'enrichir graduellement le système par des détails d'architecture et des choix de concepteur.

Les applications que nous étudions sont décrites dans un langage de haut niveau, le langage TML. Ce langage permet la modélisation parallèle de l'application et l'abstraction des valeurs des données de traitement en ne gardant que l'information de l'existence et ou l'absence des données. Dans ce langage, aucune information sur l'architecture n'est décrite. Les architectures sont des composants matériels interconnectés entre eux via des supports de communication ; elles sont aussi abstraites. Nous nous focalisons sur les paramètres des supports de communications partagées ainsi que des informations sur la taille des mémoires et les cours d'exécutions des différents éléments de calcul. Le processus de projection permet d'associer le modèle TML de l'application au modèle de la tâche dans le but d'introduire des détails architecturaux sur la spécification du comportement du système par un processus de raffinement.

Nous nous intéressons au raffinement des canaux de communication. Nous avons proposé des étapes de raffinement sous forme des règles qui transforment les canaux point-à-point depuis la spécification abstraite "application" vers les bus concrets décrits par l'architecture. Ces étapes correspondent à des niveaux de description de systèmes. De plus, ces transformations appliquées dans un certain ordre garantissent la préservation de certaines exécutions.

Pour décrire les étapes de raffinement, nous avons proposé une sémantique au langage de description TML, cette sémantique d'une application est décrite en une suite de traces qu'elle peut exécuter.

Nous avons illustré et validé notre approche sur un exemple réel que nous avons modélisé en TML et sur lequel nous avons appliqué les transformations proposées. La validation est réalisée par analyse de raffinement sur les différents niveaux obtenus par les transformations en utilisant des outils de model-checking.

Résumé de la contribution.

La contribution principale de la thèse est ***"d'offrir un cadre à l'assistance au raffinement dans la conception des systèmes embarqués en utilisant des méthodes formelles"***. La démarche proposée présente une solution pour rendre disponible l'utilisation du raffinement formel dans le processus de conception et de vérification des systèmes. Ainsi, elle permet de réduire le temps de conception par des étapes guidée et de vérification de préservation de propriété par raffinement. Les autres contributions de cette thèse peuvent se résumer par les points suivants :

- ***Donner et définir des règles claires pour transformer des médiums de communication.*** Ces règles sont formalisées comme étant des règles de transformation de la sémantique POMSET de l'application TML, qui prend en compte les contraintes architecturales, les contraintes d'implémentation, et le choix du concepteur. Ce qui permet la systématisation de processus de raffinement.

- ***Proposer une sémantique pour le langage*** TML**.** Nous avons proposé une sémantique formelle à ce langage, cette sémantique est donnée dans un langage POMSET qui décrit l'ensemble des traces qu'une application peut s'exécuter par des ordres entre les différentes actions de celle-ci. Ce formalisme est réputé pour la modélisation des systèmes concurrents et permet de manipuler les arrangements des actions lors du raffinement et de bien identifier et caractériser les règles de transformations.

- ***Établir un schéma de preuve de la préservation de propriétés linéaires après application des règles de transformation***.

- ***Montrer la faisabilité de notre démarche en l'appliquant sur une étude de cas***.

Perspectives

Cette thèse ouvre différentes directions pour la suite de travaux de recherche sur l'assistance au raffinement prouvé dans les systèmes embarqués. Les deux perspectives immédiates de notre travail sont "la preuve formelle et la garantie de préservation des propriétés" et "l'extension des étapes de raffinement aux autres concepts". Ces deux perspectives permettent une "automatisation de processus de raffinement" de bout en bout.

- ***La preuve formelle et la garantie de préservation des propriétés.*** Il s'agit de démontrer la correction du raffinement engendré par les transformations quel que soit le modèle d'entrée ou pour une classe de modèles d'entrées bien identifiée. Ce qui permettra, par la suite, la conception des systèmes corrects par construction. Un schéma de preuve peut être le suivant : D'abord, les propriétés sur les composants et le système sur chaque niveau doivent être caractérisées et classifiées. À partir de ces propriétés, une étude de la correction des règles de la sémantique de raffinement doit être établie pour chaque classe de propriété.

  Pour la preuve du raffinement entre l'application et le premier niveau de raffinement, nous proposons de nous baser sur les travaux de [98], qui établissent des conditions

permettant de prouver l'absence d'état de blocage d'un système global ; notamment la non-divergence des traces des composants ainsi que des propriétés sur le composant et son entourage de communication.

La propriété de non-divergence et l'intégration des traces infinies des composants du premier niveau de raffinement peuvent être prouvées selon le schéma suivant : Puisqu'une action abstraite est considérée effectuée si la dernière action de sa transformée est effectuée, ces deux propriétés peuvent être prouvées sur les modèles des tâches par un raisonnement sur la transformation de substitution d'événement supérieur de pomset $\mathcal{T}_2$, cette transformation permet la preuve de non-divergence, car la dernière action de la même séquence est toujours observable, de plus, la préservation d'ordre sur les dernières actions des séquences permet la preuve de reconstruction de trace de modèle de la tâche abstraite. Puisque les autres règles sont des restrictions de la relation d'ordre d'une tâche sur ce niveau, les traces construites ne peuvent représenter qu'un sous-ensemble du comportement initial. De la même manière, une preuve de la préservation des traces infinies des canaux peut être réalisée à partir de la règle de transformation $\mathcal{T}_2$, cette règle construit un canal raffiné qui possède un sous-ensemble de comportement des canaux abstraits. Ces propriétés de préservation des traces infinies et la non-divergence des traces non-observables des tâches et canaux sont une étape vers la preuve de non introduction des états de blocage. Le même processus de preuve peut être appliqué aux niveaux suivants.

- ***Étendre les étapes de raffinement.*** Il s'agit d'intégrer des nouvelles contraintes liées à la structure interne des éléments de calcul, l'ordonnancement des tâches, la génération du code, la synthèse du matériel, le raffinement par des réseaux de communications ainsi que la proposition d'une bibliothèque de spécifications abstraites et concrètes des différents composants. La structure interne des processeurs ainsi que l'information sur l'ordonnancement permettent de construire un modèle des tâches qui peut être généré dans un langage compilé ou synthétisé pour une implémentation matérielle. En fait, de la même manière que le raffinement de communication, ce processus doit proposer des transformations conformes aux contraintes de processus, des propriétés de construction des modèles selon des contraintes de partage de ressource ainsi que l'introduction des comportements propres aux contrôleurs et aux politiques d'ordonnancement. De plus, une démarche de raffinement de calcul, d'introduction des données de contrôle ainsi que la proposition d'intégration de l'information de l'adressage et la valeur des données doivent être étudiées.

- ***Automatisation des transformations.*** Il s'agit de générer automatiquement les différents modèles des différents niveaux ainsi que l'automatisation des étapes de raffinement. Le but de l'automatisation est de construire un outil de conception permettant de proposer des raffinements (semi)-automatiques à partir d'un modèle d'application et un modèle d'architecture fournis par le concepteur. Cet outil doit permettre de vérifier des obligations de preuve de préservation des propriétés de fonctionnement et des propriétés de l'exécution du système sur l'architecture lors des étapes de raffinement. Actuellement, notre démarche de raffinement peut être intégrée dans l'environnement DIPLODOCUS [102], cet environnement basé sur UML offre une description basée sur le langage TML. Les étapes de raffinement de communication que nous proposons permettront bien l'extension de l'environnement actuel par une démarche d'assistance au raffinement guidé avec la vérification formelle des propriétés par raffinement.

Bibliographie

[1] F. Abbes, E. Casseau, and M. Abid. Spécification et conception de systemes sur puce avec SystemC, Etude de cas : le turbo-codage. *RenPar'15/CFSE'3/ SympAAA'2003*, 2003.

[2] J.-R. Abrial. *The B-book : assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996.

[3] J.-R. Abrial. *Modeling in Event-B : System and Software Engineering*. Cambridge University Press, New York, NY, USA, 1st edition, 2010.

[4] C. André. Le temps dans le profil UML MARTE. Technical Report RR-2007-19, I3S, Sophia-Antipolis, (France), 2007.

[5] J. Andronick, B. Chetali, and C. Paulin-Mohring. Formal Verification of Security Properties of Smart Card Embedded Source Code. In J. Fitzgerald, I. J. Hayes, and A. Tarlecki, editors, *FM*, Lecture Notes in Computer Science, pages 302–317. Springer, 2005.

[6] L. Apvrille, W. Muhammad, R. Ameur-Boulifa, S. Coudert, and R. Pacalet. Abstract application modeling for system design space exploration. *Euromicro conference on Digital System Design (DSD 06)*, 2006.

[7] ARM. AMBA Specification (Rev 2.0). Technical report, ARM, 1999.

[8] A. Arnold. Sémantique des Processus Communicants. *ITA*, 15(2) :103–139, 1981.

[9] Automation Design and Committee Standards. IEEE Standard SystemC Language Reference Manual. *IEEE Computer Society*, 2002 :1666–2005, March 2006.

[10] B-Core. B-Core's website. Technical report, B-Core (UK) Limited.

[11] K. Bae, P. C. Ölveczky, T. H. Feng, E. A. Lee, and S. Tripakis. Verifying hierarchical Ptolemy II discrete-event models using Real-Time Maude. *Sci. Comput. Program.*, 77(12) :1235–1271, 2012.

[12] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. *Hardware-software co-design of embedded systems : the POLIS approach*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[13] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis : an integrated electronic system design environment. *Computer*, 36(4) :45–52, 2003.

[14] J. Baumgartner and T. Heyman. An overview and application of model reduction techniques in formal verification. In *Performance, Computing and Communications, 1998. IPCCC'98., IEEE International*, pages 165–171, Feb 1998.

[15] I. Beer, S. Ben-David, C. Eisner, D. Geist, L. Gluhovsky, T. Heyman, A. Landver, P. Paanah, Y. Rodeh, G. Ronin, and Y. Wolfsthal. RuleBase : Model checking at IBM. In O. Grumberg, editor, *Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 480–483. Springer Berlin Heidelberg, 1997.

[16] F. Bellegarde, J. Julliand, and O. Kouchnarenko. Ready-Simulation Is Not Ready to Express a Modular Refinement Relation. In *the Third Internationsl Conference on Fundamental Approaches to Software Engineering*, FASE'00, pages 266–283, 2000.

[17] N. Benaissa and D. Méry. Proof-Based Design of Security Protocols. In E. W. Mayr, editor, *5th International Computer Science Symposium in Russia, CSR 2010*, volume 6072 of *Lecture Notes in Computer Science*, pages 25–36. Farid Ablayev, Springer, 2010.

[18] B. Berthomieu, J.-P. Bodeveix, S. Dal Zilio, P. Dissaux, M. Filali, P. Gaufillet, S. Heim, and F. Vernadat. Formal Verification of AADL models with Fiacre and Tina. In *Embedded Real-Time Software and Systems, ERTSS 2010*, pages 1–9, TOULOUSE (31000), France, 2010. 9 pages DGE Topcased.

[19] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development.* Springer, June 2004.

[20] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic Model Checking Using SAT Procedures instead of BDDs. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, DAC'99, pages 317–320, 1999.

[21] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in computers*, 58 :117–148, 2003.

[22] A. Bouali, A. Ressouche, V. Roy, and R. De Simone. The Fc2tools set. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology*, volume 1101 of *Lecture Notes in Computer Science*, pages 595–598. Springer Berlin / Heidelberg, 1996.

[23] T. Boulusset. *β-SPACE : Raffinement de descriptions architecturales en machines abstraites de la méthode formelle* B. PhD thesis, Université de Savoie, 1992.

[24] M. Bozga, J.-C. Fernandez, L. Mounier, F. Lang, and H. Garavel. Manual page EXP 2.0 language in http ://cadp.inria.fr/man/exp.open.html. CADP.

[25] C. Braunstein and E. Encrenaz. Using CTL formulae as component abstraction in a design and verification flow. In *Seventh International Conference on Application of Concurrency to System Design, ACSD 2007*, pages 80–89, 2007.

[26] R. Brayton, G. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. Ranjan, S. Sarwary, T. Staple, G. Swamy, and T. Villa. VIS : A system for verification and synthesis. In R. Alur and T. Henzinger, editors, *Computer Aided Verification*,

volume 1102 of *Lecture Notes in Computer Science*, pages 428–432. Springer Berlin Heidelberg, 1996.

[27] J.-Y. Brunel, E. A. Kock, W. M. Kruijtzer, H. J. Kenter, and W. J. Smits. Communication refinement in video systems on chip. In *Proceedings of the seventh international workshop on Hardware/software codesign*, CODES'99, pages 142–146. ACM, 1999.

[28] L. Cai, S. Verma, and D. Gajski. Comparison of SpecC and SystemC Languages for System Design. *Technical Report, CECS-TR-03-11, UCI*, May 2003.

[29] J. P. Calvez. Embedded real-time systems. A specification and design methodology. John Wiley and Sons, 1993.

[30] D. Cansell, G. Gopalakrishnan, M. Jones, D. Méry, and A. Weinzoepflen. Incremental Proof of the Producer/Consumer Property for the PCI Protocol. In *2nd International Conference of B and Z Users, ZB 2002*, volume 2272 of *Lectures Notes in Computer Science*, pages 22–41. Springer, 2002.

[31] D. Cansell, D. Méry, and C. Proch. System-on-chip design by proof-based refinement. *International Journal on Software Tools for Technology Transfer*, 11(3) :217–238, 2009.

[32] H. Cho, S. Abdi, and D. Gajski. Interface synthesis for heterogeneous multi-core systems from transaction level models. In *Proceedings of the 2007 ACM SIGPLAN/-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 140–142, 2007.

[33] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent system using temporal logic specifications : A practical approach. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 117–126. ACM, 1983.

[34] E. M. Clarke, A. Gupta, and O. Strichman. SAT-based counterexample-guided abstraction refinement. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 23(7) :1113–1123, 2004.

[35] CLEARSY. Computer aided software environment, atelier de génie logiciel. Technical report, CLEARSY.

[36] J. L. Colley. *Guarded Atomic Actions and Refinement in a System-on-Chip Development Flow : Bridging the Specification Gap with Event-B.* PhD thesis, University of Southamption, 2010.

[37] P. Coussy. *Synthèse d'interface de communication pour les composants virtuels.* PhD thesis, Université de Bretagne sud, 2003.

[38] P. Coussy, E. Casseau, P. Bomel, A. Baganne, and E. Martin. A formal method for hardware IP design and integration under I/O and timing constraints. *ACM Trans. Embedded Comput. Syst.*, 5(1) :29–53, 2006.

[39] P. Coussy, C. Chavet, P. Bomel, D. Heller, E. Senn, and E. Martin. GAUT : A High-Level Synthesis Tool for DSP Applications. In P. Coussy and A. Morawiec, editors, *High-Level Synthesis*, chapter 9, pages 147–169. Springer Netherlands, 2008.

[40] J.-L. Dekeyser, A. Gamatié, S. Meftali, and I. R. Quadri. Models for Co-Design of Heterogeneous Dynamically Reconfigurable SoCs. In Ian O Connor, editor, *Heterogeneous Embedded Systems - Design Theory and Practice*, page 26. Springer, 2011.

[41] D. Densmore and R. Passerone. A Platform-Based Taxonomy for ESL Design. *Design Test of Computers, IEEE*, 23(5) :359–374, 2006.

[42] E. W. Dijkstra. *A Discipline of Programming*. Prince-Hall, 1976.

[43] C. Erbas. *System-Level Modeling and Design Space Exploration for Multiprocessor Embedded System-on-Chip Architectures*. PhD thesis, Amsterdam University Press, 2006.

[44] P. H. Feiler, D. P. Gluch, and J. J. Hudak. The Architecture Analysis & Design Language (AADL) : An Introduction. Technical Report CMU/SEI-2006-TN-011, Software Engineering Institute, Carnegie Mellon University, 2006.

[45] Formal Systems (Europe) Ltd. Failures-Divergence Refinement, FDR2 User Manual. Technical report, Oxford University Computing Laboratory, 2010.

[46] O. M. G. UML Profile for MARTE : Modeling and Analysis of Real-Time Embedded Systems. Technical report, Object Management Groupe (OMG), 2011.

[47] D. Gajski. *SpecC : Specification Language and Methodology*. Kluwer Academic Publishers, 2000.

[48] D. Gajski, S. Abdi, A. Gerstlauer, and G. Schirner. *Embedded System Desing : Modeling, Synthesis and Verification*. Springer Dordrecht Heidlberg london New York, 2009.

[49] D. Gajski and R. H. Kuhn. New VLSI Tools. *IEEE Computer*, 16(12) :11–14, 1983.

[50] D. Gajski, F. Vahid, S. Narayan, and J. Gong. System-level Exploration with SpecSyn. In *Proceedings of the 35th Annual Design Automation Conference*, DAC'98, pages 812–817. ACM, 1998.

[51] H. Garavel, F. Lang, R. Mateescu, and W. Serve. CADP 2010 : A Toolbox for the Construction and Analysis of Distributed Processes. In P. A. Abdulla, K. Rustan, and M. Leino, editors, *TACAS'11*, volume 6605 of *LNCS*, Saarbrücken, Germany, 2011. Springer, Heidelberg.

[52] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2011 : A Toolbox for the Construction and Analysis of Distributed Processes. *International Journal on Software Tools for Technology Transfer*, 2012.

[53] D. Genius, E. Faure, and N. Pouillon. Mapping a telecommunication application on a multiprocessor System-on-Chip. In G. Gogniat, D. Milojevic, A. Morawiec, and

A. Erdogan, editors, *Algorithm-Architecture Matching for Signal and Image Processing*, volume 73 of *Lecture Notes in Electrical Engineering*, pages 53–77. Springer Netherlands, 2011.

[54] S. Grumbach and T. Milo. An algebra for pomsets. In G. Gottlob and M. Vardi, editors, *Database Theory ICDT'95*, volume 893 of *Lecture Notes in Computer Science*, pages 191–207. Springer Berlin Heidelberg, 1995.

[55] N. Guelfi and B. Ries. SESAME : A Model-Driven Test Selection Process for Safety-Critical Embedded Systems. *ERCIM News*, 2008.

[56] R. K. Gupta, C. N. Coelho, and G. D. Micheli. Synthesis and simulation of digital systems containing interacting hardware and software components. In *Proceedings of the 29th ACM/IEEE Design Automation Conference*, DAC'92, pages 225–230. IEEE Computer Society Press, 1992.

[57] N. Halbwachs. A synchronous language at work : the story of Lustre. In *Formal Methods and Models for Co-Design, 2005. MEMOCODE'05. Proceedings. Third ACM and IEEE International Conference on*, pages 3–11, 2005.

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

[59] G. Hamon and M. Pouzert. Modular resetting of synchronous data-flow programs. In *Proceedings of the 2nd ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP'00, pages 289–300. ACM, 2000.

[60] G. J. Holzmann. The Model Checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5) :279–295, 1997.

[61] D. Hommais, F. Petrot, and I. Auge. A practical tool box for system level communication synthesis. In *CODES'01 : Proceedings of the ninth international symposium on Hardware/software codesign*, pages 48–53, 2001.

[62] Y. Hwang, G. Schirner, and S. Abdi. Automatic Generation of Cycle-Approximate TLMs with Timed RTOS Model Support. In *Analysis, Architectures and Modelling of Embedded Systems, Third IFIP TC 10 International Embedded Systems Symposium, IESS 2009*, pages 66–76, 2009.

[63] IBM. CoreConnect Bus Architecture. Technical report, IBM, 1999.

[64] L. Isenegger, L. Jérôme, and W. O. Cesário. Verifying a CoFluent SystemC IP Model from a SystemVerilog UVM Testbench in Mentor Graphics Questa. Montor Graphics, 2010.

[65] ISO-LOTOS. A Formal Description Technique Based On the Temporal Ordering of Observational Behavior. *In draft international Standard 8807, International Organisation For Standarisation- information processing systems-Open systems Interconnection*, 1987.

[66] Iulian Ober and Ileana Ober, editor. *SDL 2011 : Integrating System and Software Modeling*, volume 7083 of *Lecture Notes in Computer Science*. Springer, 2012.

[67] C. Jaber, A. Kanstein, L. Apvrille, A. Baghdadi, P. L. Moenner, and R. Pacalet. High-Level System Modeling for Rapid HW/SW Architecture Exploration. In *IEEE International Workshop on Rapid System Prototyping*, pages 88–94, 2009.

[68] B. Johan and Y. Wang. Timed Automata : Semantics, Algorithms and Tools. *Lectures on Concurrency and Petri Nets*, pages 87–124, 2004.

[69] D. Knorreck, L. Apvrille, and R. Pacalet. Fast Simulation Techniques for Design Space Exploration. In *Objects, Components, Models and Patterns*, volume 33 of *47th International Conference, TOOLS EUROPE-09*, pages 308–327. Lecture Notes in Business Information Processing, 2009.

[70] D. Ku and G. D. Micheli. HardwareC - A Language for Hardware Design Version 2.0. *Technical Report No. CSL-TR-90-419*, April 1990.

[71] E. Lee and T. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5) :773–801, 1995.

[72] E. A. Lee. Finite State Machines and Modal Models in Ptolemy II. Technical Report UCB/EECS-2009-151, EECS Department, University of California, Berkeley, 2009.

[73] H. G. Lee, N. Chang, U. Y. Ogras, and R. Marculescu. On-chip Communication Architecture Exploration : A Quantitative Evaluation of Point-to-point, Bus, and Network-on-chip Approaches. *ACM Trans. Des. Autom. Electron. Syst.*, 12(3), 2007.

[74] P. Lieverse, P. V. Wolf, and E. D. Deprettere. A trace transformation technique for communication refinement. In *CODES'01 : Proceedings of the ninth international symposium on Hardware/software codesign*, 2001.

[75] S. Maharaj and J. Bicarregui. On the Verification of VDM Specification and Refinement with PVS. In *Proof in VDM : Case Studies, FACIT (Formal Approaches to Computing and Information Technology)*, pages 157–190. Springer-Verlag, 1997.

[76] F. Maraninchi and Y. Rémond. Mode-Automata : a new domain-specific construct for the development of safe critical systems. *Sci. Comput. Program.*, 46 :219–254, 2003.

[77] R. Marculescu, Ü. Y. Ogras, and N. H. Zamora. Computation and communication refinement for multiprocessor SoC design : A system-level perspective. *ACM Trans. Design Autom. Electr. Syst.*, 11(3) :564–592, 2006.

[78] R. Mateescu and D. Thivolle. A Model Checking Language for Concurrent Value-Passing Systems. In J. Cuellar, T. Maibaum, and K. Sere, editors, *FM 2008 : Formal Methods*, volume 5014 of *Lecture Notes in Computer Science*, pages 148–164. Springer Berlin Heidelberg, 2008.

[79] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[80] J. Mikác and P. Caspi. Flush : an example of development by refinements in SCADE/Lustre. *International Journal on Software Tools for Technology Transfer (STTT)*, 11(5) :409–418, 2009.

[81] H. Mokrani and R. Ameur-Boulifa. A Refinement Approach to Design and Verification of On-Chip Communication Protocols. *Journée du Groupe SAFA (Sophia-Antipolis Formal Analysis Group), informal proceeding*, 2011.

[82] H. Mokrani, R. Ameur-Boulifa, S. Coudert, and E. Encrenaz-Tiphène. Communication Refinement for SOC Design. *Journée du Groupe SAFA (Sophia-Antipolis Formal Analysis Group), informal proceeding*, 2010.

[83] H. Mokrani, R. Ameur-Boulifa, and E. Encrenaz-Tiphene. Assisting Refinement In System-on-Chip Design. In *Forum on Specification Design Languages, FDL'13*, pages 1–6, 2013.

[84] H. Mokrani, R. Ameur-Boulifa, and E. Encrenaz-Tiphene. Assisting Refinement In System-on-Chip Design. In M. M. Louërat and T. Maehne, editors, *chapter : Models, Methods and Tools for Complex Chip Design, selected contributions from FDL'13, Lecture Notes in Electrical Engineering (LNEE), Springer*, 2014 (À paraitre).

[85] H. Mokrani, R. Ameur-Boulifa, E. Encrenaz-Tiphène, and S. Coudert. Approche pour l'intégration du raffinement formel dans le processus de conception des SOC. In *Journal Européen des Systèmes Automatisés (JESA), Special Issue MSR'11*, volume 45, pages 221–236, 2011.

[86] S. Nicolas. *Systèmes de transitions symboliques et hiérarchiques pour la conception et la validation de modèles B raffinés.* PhD thesis, Institut National Polytechnique de Grenoble - INPG, 2007.

[87] T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science, LNCS*. Springer, 2002.

[88] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Commun. ACM*, 29(12) :1184–1201, 1986.

[89] P. R. Panda. SystemC : a modeling platform supporting multiple design abstractions. In *Proceedings of the 14th International Symposium on Systems Synthesis*, ISSS'01, pages 75–80, New York, NY, USA, 2001. ACM.

[90] R. Passerone, L. de Alfaro, T. A. Henzinger, and A. L. Sangiovanni-Vincentelli. Convertibility verification and converter synthesis : two faces of the same coin. In L. T. Pileggi and A. Kuehlmann, editors, *In International Conference on Computer Aided Design ICCAD*, pages 132–139. ACM, 2002.

[91] O. Pell. Verification of FPGA Layout Generators in Higher-Order Logic. *Journal of Automated Reasoning*, 37(1-2) :117–152, 2006.

[92] A. Pimentel, L. Hertzbetger, P. Lieverse, V. D. Wolf, and E. Deprettere. Exploring embedded-systems architectures with Artemis. *Computer*, 34(11) :57–63, 2001.

[93] N. Pontisso and D. Chemouil. TOPCASED Combining Formal Methods with Model-Driven Engineering. In *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*, pages 359–360, 2006.

[94] V. R. Pratt. The Pomset Model of Parallel Processes : Unifying the Temporal and the Spatial. In *Seminar on Concurrency*, pages 180–196, 1984.

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

[96] P. Rashinkar, P. Paterson, and L. Singh. *System-on-a-chip Verification : Methodology and Techniques*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[97] E. Rigoni, C. Kavka, A. Turco, G. Palermo, C. Silvano, V. Zaccaria, and G. Mariani. Optimization Algorithms for Design Space Exploration of Embedded Systems. In C. Silvano, W. Fornaciari, and E. Villar, editors, *Multi-objective Design Space Exploration of Multiprocessor SoC Architectures*, pages 51–74. Springer New York, 2011.

[98] A. W. Roscoe and N. Dathi. The Pursuit of Deadlock freedom. *Information and Computation*, 75(3) :289–327, 1987.

[99] A. W. Roscoe, C. A. R. Hoare, and R. Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[100] D. Sabatier and P. Lartigue. The Use of the B Formal Method for the Design and the Validation of the Transaction Mechanism for Smart Card Applications. In J. Wing, J. Woodcock, and J. Davies, editors, *Formal Methods (FM'99)*, volume 1708 of *Lecture Notes in Computer Science*, pages 348–368. Springer Berlin Heidelberg, 1999.

[101] F. Schirrmeister and A. Sangiovanni-Vincentelli. Virtual component co-design-applying function architecture co-design to automotive applications. In *Vehicle Electronics Conference, 2001. IVEC 2001. Proceedings of the IEEE International*, pages 221–226, 2001.

[102] F. B. Schneider and D. Gries, editors. *Refinement Calculus : A Systematic Introduction*. Springer-Verlag New York, Inc., 1998.

[103] K. Seidel. Case study : Specification and refinement of the PI-Bus. In M. Naftalin, T. Denvir, and M. Bertran, editors, *FME'94 : Industrial Benefit of Formal Methods*, volume 873 of *Lecture Notes in Computer Science*, pages 532–546. Springer Berlin Heidelberg, 1994.

[104] T. Shanley and D. Anderson. *PCI System Architecture*. Addison Wesley, 4th edition, 2006.

[105] D. Shin, S. Abdi, and D. Gajski. Automatic generation of bus functional models from transaction level models. In *Design Automation : Electronic Design and Solution Fair 2004*, pages 756–758, 2004.

[106] W. T. Shiue and C. Chakrabarti. Memory exploration for low power, embedded systems. In *Design Automation Conference*, pages 140–145, 1999.

[107] Siemens AG. PI-Bus (Rev. 0.3d) : Draft standard OMI 324. Technical report, Siemens AG, 1994.

[108] G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[109] O. Tardieu. A Deterministic Logical Semantics for Pure Esterel. *ACM Trans. Program. Lang. Syst.*, 29(2), 2007.

[110] F. Vahid and T. Givargis. *Embedded System Design : A Unified Hardware/Software Introduction*. Wiley, international student edition, 2001.

[111] R. van Glabbeek. The Linear Time – Branching Time Spectrum I ; The Semantics of Concrete, Sequential Processes. In J. Bergstra, A. Ponse, and S. Smolka, editors, *Handbook of Process Algebra*, pages 3–99. Elsevier, 2001.

[112] K. S. Wolff, T. Santen, and B. Wolff. A Structure Preserving Encoding of Z in Isabelle/HOL. In *Theorem Proving in Higher-Order Logics, LNCS 1125*, pages 283–298. Springer Verlag, 1996.

[113] J. Woodcock and J. Davies. *Using Z specification, refinement and proof*. Prentice Hall, 1996.