République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université de Constantine
Faculté des Sciences de l'Ingénieur

Département d'Informatique

N° d'ordre:
N° Série:

THESE

Pour l'obtention du grade de
DOCTEUR EN SCIENCE

Spécialité

Informatique

Thème

Aide à la Conception des Systèmes Multiprocesseurs Mono-puce à partir de Spécification de haut niveau

Présenté par

Fateh Boutekkouk

Directeur de thèse

Pr. Mohamed Benmohammed

Soutenu le : 27/01/2010

Composition du Jury :

Pr. Mahmoud Boufaida	*Président*	Univ. *Constantine*
Pr. Mouloud Koudil	*Examinateur*	ESI. *Alger*
MC. Mohamed T. Kimour	*Examinateur*	Univ. *Annaba*
MC. Alloua Chaoui	*Examinateur*	Univ. *Constantine*
Pr. Mohamed Benmohammed	*Rapporteur*	Univ. *Constantine*

Remerciements

Je voudrais témoigner à l'ensemble des personnes qui m'ont guidé ou accompagné, ma sincère et profonde reconnaissance :

Monsieur Mohamed Benmohammed, directeur de thèse qui m'a témoigné sa confiance et grâce à lui j'ai pu effectuer cette thèse.

Monsieurs Michel Auguin et Sebastien Bilavarn, pour avoir accepté de m'accueillir au sein du Laboratoire d'Electroniqe, Antennes et Télécommunication (LEAT) de Nice sophia antipolis – France, et pour leurs remarques et critiques.

Les membres du labo LEAT pour leurs contributions et la qualité de discussion que nous avons eu.

Monsieurs les membres du jury pour avoir accepté de juger ce travail.

Les membres de ma famille ; mes parents, mes frères, mes sœurs. Toutefois, j'adresse un remerciement particulier à mon épouse pour son soutient et compréhension envers mes humeurs tout au long de ce travail.

Résumé

La conception des systèmes multiprocesseurs mono-puce (MPSOC) est devenue de plus en plus complexe et très coûteuse. Afin de pouvoir maîtriser la complexité toujours croissante de tels systèmes, les spécialistes du domaine insistent sur la nécessité de lever le niveau d'abstraction et de faire recourir aux technologies maîtrisées du génie logiciel telles que le paradigme visuel objet et les techniques formelles. Dans ce domaine, l'UML est considéré aujourd'hui comme l'un des langages les plus prometteurs dans le développement des systèmes informatiques. En parallèle, il est observé que plus la validation intervient tôt dans le flot de conception d'un système, plus une erreur peut être corrigée rapidement. Afin de réduire les coûts de conception et d'économiser les efforts inutiles, il est important de vérifier la fonctionnalité du système à un niveau d'abstraction élevé. Dans le cadre de cette thèse, on s'est interessé à cinq problèmes. Le premier et le deuxième problèmes sont relatifs à la modélisation UML des MPSOC et l'estimation de performances (temps et consommation) en suivant les principes de la méthode en Y respectivement. Le troisième problème concerne la transformation de modèles UML vers un langage au niveau système standard pour les MPSOC comme le SystemC. Le quatrième problème est lié à l'aspect spécification et vérification formelles de quelques propriétés du système en s'appuyant sur la logique de réécriture et en particulier le langage Maude. Le cinquième problème concerne l'intégration d'UML avec les outils CAD de cosimulation et de synthèse de haut niveau existants et en particulier les outils ciblant les architectures reconfigurables.

Mots clés : MPSOC, Systèmes Embarqués, UML, Logique de réécriture, Maude, Vérification Formelle, SystemC, Simulation, Architectures reconfigurables.

Abstract

Multiprocessor System On Chip (MPSOC) design is becoming more complex and expensive. In order to cope with this complexity, specialists in the field emphasize on raising the level of abstraction and resort to software engineering to borrow from it some well practiced technologies such as object technology and formal techniques. In this domain, the Unified Modeling Language is considered as de facto standard for visual object modeling. On the other side, early validation of system functionality and constraints can reduce the effort of development considerably since verification tools working at a high level of abstraction are available. In the context of this thesis, we are interested in five problems. The first and second problems are related to UML based modeling and high level performance estimation of MPSOC following the Y-Chart approach respectively. The third problem concerns the translation of UML models to a system level language standard such as SystemC. The fourth problem deals with formal specification and verification of some properties using rewriting logic and in particular the Maude language. The last problem copes with the integration of UML with existing CAD tools of high level synthesis and cosimulation targeting reconfigurable architectures.

Key-words : MPSOC, Embedded Systems, UML, SystemC, Rewriting Logic, Maude, Formal verification, Reconfigurable Architectures.

ملخص

تصميم الأنظمة أحادية البطاقة متعددة المعالجات المركزية هي عملية معقدة ومكلفة جدا.
من أجل التحكم العقلاني في التعقيدات الناجمة عن الارتفاع المتزايد في كثافة التكامل المتعلقة بتكنولوجيا نصف النواقل و قيود التصميم, الأخصائيون في المجال يقرون اللجوء الى التقنيات المألوفة في هندسة البرامج و تطبيقها في مجال العتاد.
من بين هذه التقنيات نجد لغة النمذجة الموحدة و التقنيات الوصفية. في هذه المذكرة نقدم مجموعة من المنهجيات والأدوات المطورة التي تساعد المصممون على نمذجة الأنظمة أحادية البطاقة متعددة المعالجات المركزية, تقدير وقت التنفيذ و الطاقة المستهلكة من طرف هاته الأنظمة انطلاقا من نماذج اللغة الموحدة و كذا التأكد من المصداقية بتطبيق التقنيات الوصفية و ربط جسر بين اللغة الموحدة و اللغات المستعملة بكثرة لبرمجة الأنظمة أحادية البطاقة متعددة المعالجات المركزية كلغة SystemC و الأدوات التجارية الموجهة إلى التركيب عالي المستوى و المحاكاة.

كلمات مفتاح : الأنظمة أحادية البطاقة متعددة المعالجات المركزية, لغة النموذجة الموحدة, التقنيات الوصفية .

Table des matières

Table des figures

Liste des tableaux

Introduction générale

Sommaire

1. Systèmes multiprocesseurs Mono puce MPSOC.
2. Conception de haut niveau des MPSOC.
3. Objectifs et contributions.
4. Plan du mémoire.

Les systèmes embarqués (SE) sont de plus en plus présents dans notre quotidien. Cette technologie peut être trouvée aussi bien dans les systèmes multimédia que dans les applications de réseau et de télécommunication. L'utilisation des SE est en train de suivre une courbe exponentielle et on attend même que leurs ventes dépassent en revenus celles des processeurs de PC à usage général [57].

Selon le Dr. Gordon E. Moore, cofondateur d'Intel, la complexité des circuits intégrés suit une loi empirique (la loi de Moore) dans laquelle le nombre et la puissance des transistors doublaient presque tous les deux ans [72]. Ces progrès technologiques ont permis d'intégrer sur une même puce plus de fonctionnalités ce qui a conduit à la définition d'un nouveau paradigme de systèmes embarqués : les systèmes sur puce (SOC : System on Chip).

En effet, Les SOCs sont de nature hétérogène du fait de l'hétérogénéité des applications qu'ils implémentent. Ils combinent typiquement des parties logicielles (des processeurs à usage général GPP, des processeurs à usage spécifique ASIP, des processeurs de traitement du signal DSP) et d'autres matérielles (des modules spécifiques ASIC).

Une implantation logicielle d'une partie de l'application a l'avantage de lui procurer une flexibilité liée à la possibilité de reprogrammation, contrairement à une implantation matérielle figée mais qui a l'avantage de satisfaire plus facilement les contraintes de performances.

Un autre facteur important dans l'évolution de ces systèmes est l'apparition de nouvelles architectures ayant la flexibilité du logiciel et la performance du matériel, basées sur la programmation de circuits matériels tels que les FPGA. Ces propriétés attrayantes et le rythme soutenu des progrès qu'a connu cette technologie (au niveau taille, intégration et performances) ont changé le rôle jusque là joué par ces composants et leur ont permis de passer du rôle de prototypage d'ASIC ou de 'glue logic' au rôle d'unités de calcul alternatives.

On parle de plus en plus de systèmes (ou plateformes) réconfigurables qui intègrent sur un même substrat un ou plusieurs cœurs de processeurs et une matrice programmable (ex: Virtex- 2 Pro [132], Virtex-4 FX [131] de Xilinx.

Par ailleurs, tout un champ technologique émerge actuellement dans le domaine de la reconfiguration dynamique [16]. Cette technologie permet de modifier, en cours d'exécution, partiellement ou complètement la configuration (donc la fonctionnalité) du circuit. Ceci introduit une nouvelle dimension au problème de conception, en élargissant encore l'ensemble des choix d'intégration possibles. Le concepteur se retrouve donc face à des choix

d'implantations logicielles (spécifiques ou génériques) et matérielles (figées ou réconfigurables) pour les différentes parties de l'application.

Les MPSOC permettent alors une réduction importante des coûts de production et une augmentation considérable de la fiabilité des systèmes développés. La réalisation des différents composants d'un système sur puce peut être faite par plusieurs équipes de développement, où chaque équipe utilise son propre environnement de conception pour décrire la fonctionnalité du système étudié. Ce nouveau type de circuit intégré nécessite de nouveaux outils et de nouvelles méthodes de conception, de réalisation et de vérification. La conception des systèmes sur puce est soumise à un ensemble de contraintes fortes pour assurer leur bon fonctionnement. Comme ces systèmes regroupent des descriptions logicielles et matérielles, il est important de prendre en considération l'étude conjointe de ces deux concepts et leurs différentes interactions au plus haut niveau d'abstraction pour faciliter et réduire le cycle de développement. Il est alors nécessaire de maîtriser la conception matériel/logiciel des systèmes sur puce tout en respectant les contraintes de mise sur le marché et les objectifs de qualité.

Toutefois, comme nous venons de l'évoquer, les évolutions technologiques vont, à court terme, permettre de doubler le nombre de portes logiques intégrées sur un circuit de dimension et de coût équivalents. Les concepteurs auront alors la possibilité d'offrir beaucoup plus de fonctionnalités dans leur produit. Malheureusement, si de nouvelles méthodologies et de nouveaux outils de conception ne sont pas disponibles, il faudra alors également doubler le nombre de concepteurs. En effet, comme le montre la Figure 1, si l'on considère qu'un développeur peut produire 200000 portes logiques par an, pour développer un circuit intégrant 30 millions de portes, il faut alors compter 150 hommes par année.

1992	1994	1996	1998	2000	2002	2004	2006	2008	2010
Finesse de gravure									
0,6	0,5	0,35	0,25	0,18	0,13	90	65	45	32
# de portes / mm^2									
1k	5k	15k	30k	45k	80k	150k	300k	600k	1,2M
# de portes / die($50mm^2$)									
50k	250k	750k	1,5M	2,25M	4M	7,5M	15M	30M	60M
# de portes / concepteur / an									
4k	6k	9k	40k	56k	91k	125k	200k	200k	200k
Homme.Année / die($50mm^2$)									
~10	~40	~80	~40	~40	~43	~60	~75	~150	~300

Figure 1. Evolution technologique Vs évolution méthodologique [35].

Bien évidement, cet accroissement exponentiel du nombre de concepteur n'est pas viable à terme pour l'Industrie de la microélectronique, c'est pourquoi de nouvelles méthodologies de conception sont nécessaires. La Recherche en C.A.O. (Conception Assistée par Ordinateur) est ainsi entraînée dans un mouvement perpétuel pour proposer de nouvelles solutions, de nouvelles approches, de nouvelles méthodologies aux concepteurs de circuits. Le but est de maintenir un niveau de productivité par concepteur suffisant pour permettre le développement de circuit à un coût raisonnable et dans un temps limité.

1. Systèmes multiprocesseurs Mono puce MPSOC

Un système multiprocesseur sur puce MPSOC est un système complet intégrant sur une seule pièce de silicium plusieurs composants complexes et hétérogènes tels que des unités de calcul spécifiques programmables et/ou non programmable (CPU, DSP, ASIC, IP, FPGA) des réseaux de communication complexes (Bus hiérarchiques sur puce, réseau sur puce) des composants de mémorisation variés, périphériques E/S, etc. [72]. La Figure 2 représente un modèle générique d'un système MPSOC hétérogène avec des parties matérielles et logicielles structurées en couches pour maîtriser la complexité pour la partie matérielle et logicielle. Le matériel se divise en deux couches:

- La couche basse contient les composants de calcul et de mémorisation utilisés par le système (μP, μC, DSP, matériel dédié, IPs, mémoires),
- La couche matérielle de communication embarquée sur la puce composée de deux sous-couches : média de communication (liens point-à-point, bus hiérarchique, réseau sur puce) et adaptateurs de communication entre le réseau et les composants de la première couche.

Le logiciel embarqué est aussi découpé en couches :

- La couche la plus basse est l'abstraction du matériel (HAL, pour Hardware Abstraction Layer en anglais) permet de faire le lien avec le matériel en implémentant les pilotes E/S des périphériques, des contrôleurs de composants, les routines d'interruption (ISR, pour Interrupt Service Routine).
- La couche système d'exploitation qui permet de porter l'application sur l'architecture (gestion de ressources, communication et synchronisation, ordonnancement).
- La couche application

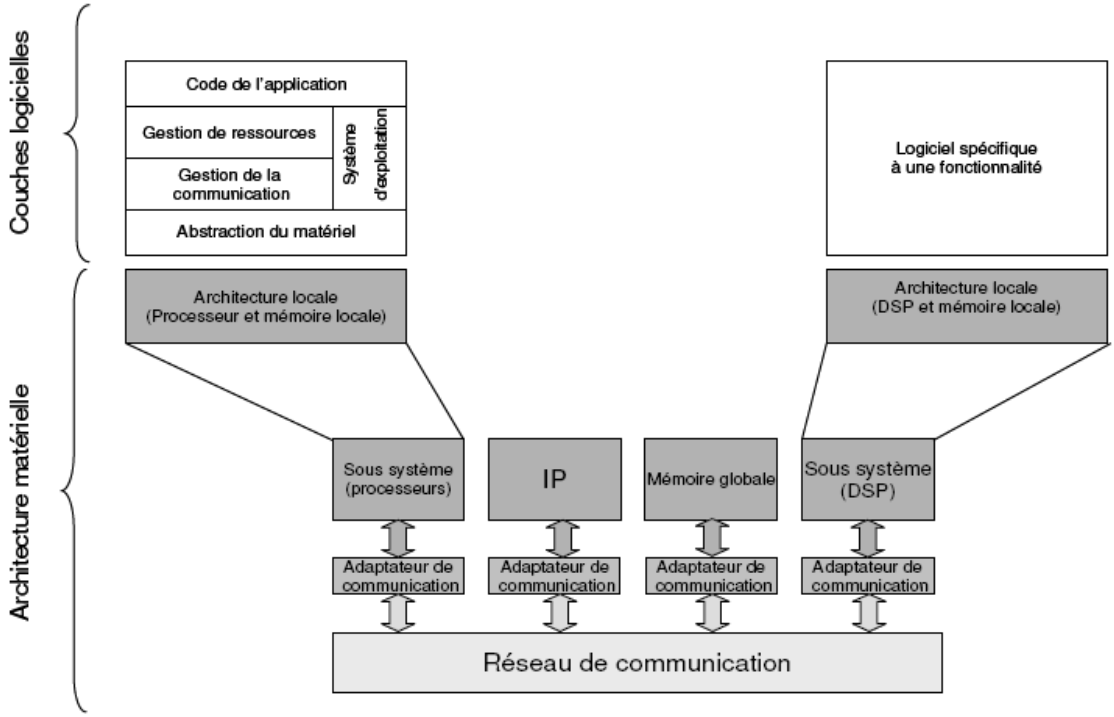


Figure 2. Modèle générique d'un MPSOC [35].

Au niveau de l'architecture globale, les choix des composants de base et du modèle de communication ne peuvent plus être faits sans prendre en compte l'organisation du logiciel. Les compromis logiciel/matériel fixent les grands choix d'architecture ainsi que les limites entre les couches de communications matérielles et les couches basses de communication logicielles. En effet, l'existence de MMU, DMA et autres facilités matérielles pour les entrées/sorties simplifie la couche de communication logicielle.

Enfin, dans le processus de conception, ces systèmes nécessitent la coopération de quatre types de métier :

- Le concepteur de logiciel écrit les différentes couches logicielles,
- Le concepteur de matériel conçoit la ou les parties matérielles,
- L'intégrateur est chargé de faciliter la communication entre le concepteur du logiciel et celui du matériel. C'est généralement lui qui réalise la couche basse du logiciel.

Le fait que les couches de communication soient réalisées par des équipes différentes peut entraîner des surcoûts dus aux précautions et/ou aux sous-utilisations des ressources. Aussi, ce métier doit évoluer et comporter la conception de toutes les couches de communications.

Cette évolution permet la séparation totale entre la conception des couches de communications et les parties matérielles et logicielles.

Une telle séparation est nécessaire pour introduire un peu de flexibilité et de modularité dans l'architecture globale du système et d'abstraire la communication entre les composants doit permettre de mieux finaliser les interfaces entre les différentes couches à l'aide d'interfaces de haut niveau.

On comprend aisément aussi la difficulté à créer des outils automatiques de conception, car il faut des outils d'aide pour chacun des types de métier.

2. Conception de haut niveau des MPSOCs

La conception des MPSOCs à l'aide de méthodes classiques conduit à des coûts et des délais de réalisation inacceptables. En fait, la conception de ces systèmes induit plusieurs points de cassures qui rendent difficile la mise en échelle des méthodes de conception existantes. Les principaux défis sont :

1. La conception des systèmes multiprocesseurs sur puce entraînera inévitablement la réutilisation de composants existants qui n'ont pas été conçus spécialement pour être assemblés sur la même puce. Il sera donc nécessaire d'utiliser des méthodes de composition et d'assemblage de composants hétérogènes;
2. Le concepteur d'un tel système contenant des processeurs doit obligatoirement fournir, en plus du matériel, une couche logicielle permettant la programmation du système intégré. Il sera donc nécessaire d'aider les concepteurs des architectures MPSOC à concevoir les couches logicielles;
3. L'assemblage de composants hétérogènes sur une même puce pour concevoir des architectures dédiées à des applications particulières peut nécessiter des schémas de communication très sophistiqués. La conception de ces interfaces hétérogènes sera vraisemblablement le principal goulet d'étranglement du processus de conception des systèmes multiprocesseurs mono puce;
4. La complexité des systèmes MPSOC est telle qu'il devient quasi impossible de spécifier/valider manuellement à des niveaux bas tels que le niveau transfert de registre (RTL) où il faut donner/valider des détails au niveau cycle d'horloge. Des méthodes de conception/validation basées sur des concepts de plus haut niveau sont donc nécessaires. Comme pour le logiciel, des outils de synthèse de haut niveau seront nécessaire pour le matériel. La vérification des composants synthétisés sera obligatoire afin de pouvoir en assurer la qualité.

Ainsi pour maîtriser cette complexité et répondre aux contraintes de temps de mise sur le marché et de coût, la conception des systèmes multiprocesseurs mono-puce doit se baser sur des approches de conception mixtes (descendante et ascendante). C'est-à-dire aussi bien sur l'assemblage et la réutilisation de composants existants préconçus et pré-validés et d'autre part sur la modélisation système et l'augmentation du niveau d'abstraction.

Face à ces difficultés, il faut proposer des méthodologies, trouver des solutions techniques et développer des outils pour transposer la spécification vers une implémentation sur une architecture MPSOC. La Figure 3 présente un flot générique de conception de haut niveau des MPSOC. Le flot part d'une spécification fonctionnelle de l'application qui permet de fixer les principales contraintes du produit à réaliser. Ce modèle sera aussi utiliser pour une exploration des algorithmes et fixer certains paramètres architecturaux. L'exploration de l'architecture consiste à trouver le meilleur partitionnement logiciel/matériel, l'allocation des processus et le choix des protocoles et des réseaux de communication. L'application est ainsi raffinée à un niveau intermédiaire qui contient les paramètres de l'architecture choisie. A partir de ce niveau, l'implémentation permet de générer l'architecture RTL. Ceci comporte la synthèse de la partie matérielle, la compilation de la partie logicielle pour cibler un processeur bien défini, et le raffinement de la communication qui génère des interfaces matérielles/logicielles connectant les différents composants au réseau de communication.

En parallèle, il est observé que plus la validation intervient tôt dans le flot de conception d'un système, plus une erreur peut être corrigée rapidement. Afin de réduire les coûts de conception et d'économiser les efforts inutiles, il est important de vérifier la fonctionnalité du système à tous les niveaux durant le processus de conception.

Globalement, la complexité des MPSOC peut rendre la modélisation de leur comportement une activité difficile et exposée à des risques d'erreurs. Dans l'industrie, les soucis d'efficacité, de réutilisation, et de programmation sans erreur sont bien connus. D'une part, la première étape dans le développement de tout système informatique se résume en un ensemble d'idées et des besoins visant à définir exhaustivement les spécifications de bases du système à réaliser. Ces besoins sont par nature non totalement formalisables car ils font partie du monde réel, et liés à des habitudes ou des opinions parfois mal conceptualisés. Réciproquement, le concepteur est étranger au monde du client, et doit prendre en considération tous ses besoins. Il s'agit donc de trouver un langage commun entre ces deux mondes différents. D'autre part, un défi supplémentaire est posé par le partage et la réutilisation du travail. Il est donc important de définir des formalismes communs permettant de faciliter l'échange et la réutilisation des différentes parties des systèmes étudiés.

Afin de pouvoir comprendre et agir sur le fonctionnement d’un système, il est nécessaire d’utiliser des techniques de modélisation et des langages communs permettant l’expression, l’échange, et la compréhension des principales fonctionnalités des systèmes étudiés. Dans ce domaine, l’UML est considéré aujourd’hui comme l’une des approches les plus prometteuses dans le développement des systèmes. Cette technique offre un cadre méthodologique et technologique qui permet d’unifier différentes façons de faire dans un processus homogène. Son objectif est de favoriser l’étude séparée des différents aspects du système. Cette séparation permet d’augmenter la réutilisation et aide les membres de l’équipe de conception à partager et s’échanger leurs travaux indépendamment de leur domaine d’expertise.

Dans le cadre du travail présenté dans ce document, nous nous intéressons en particulier à la modélisation au plus haut niveau d’abstraction des systèmes parallèles en utilisant de langages de modélisation communs et standardisés tel que UML pour mieux représenter et spécifier les différentes fonctionnalités du système étudié. Ce langage, largement utilisé dans les domaines industriels et académiques, offre une notation consensuelle permettant de privilégier la réutilisation et l’échange des différentes parties des systèmes étudiés.

Il est intéressant de noter que contrairement aux premières approches, dans lesquelles la recherche en Co-design avait pour objectif la transformation automatique d'une spécification en une implémentation, les chercheurs tendent actuellement à mettre l'accent sur des problèmes plus spécifiques liées au processus de conception, en particulier, la spécification, la modélisation architecturale, le partitionnement, l'estimation des performances, les heuristiques d'exploration, la synthèse de la communication, la validation, et la génération automatique de code exécutable.

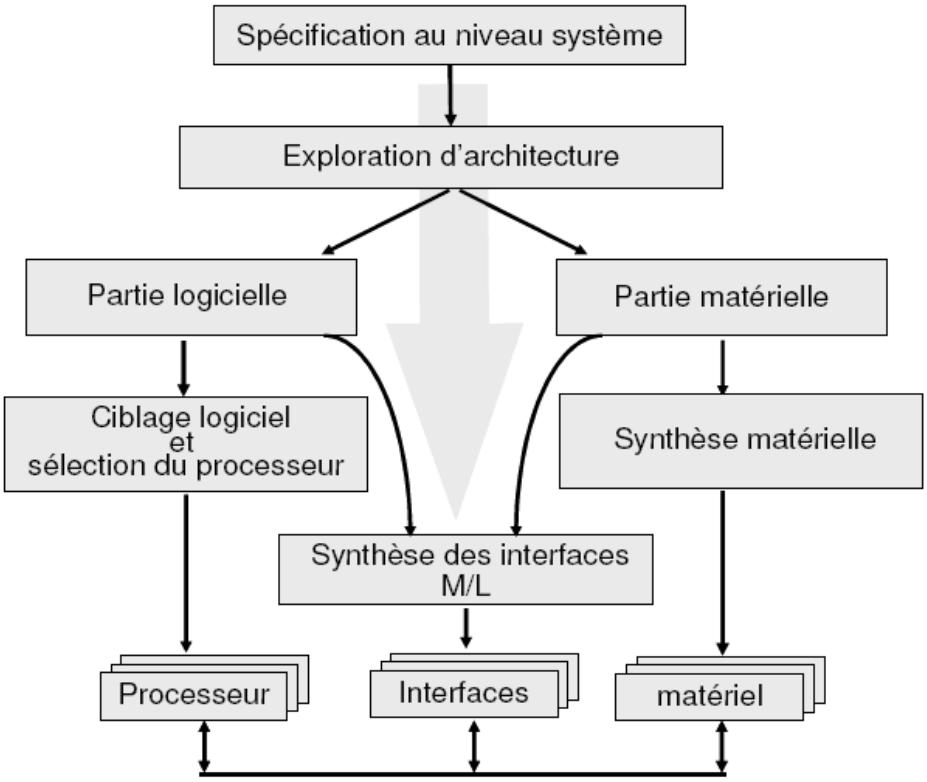


Figure 3. Flot générique de conception des MPSOC [35].

3. Objectifs et Contributions

Le but de cette thèse est d'apporter des solutions au développement des MPSOC à partir des spécifications de haut niveau:

1. Proposition d'un profil UML pour estimer le temps d'exécution au pire de cas (WCET) d'une application sur MPSOC.
2. Modélisation UML des applications mixtes (dominées par le contrôle et les données) et transformation de ces modèles vers le langage Maude qui est fondée sur la logique de réécriture pour effectuer des vérifications formelles sur le système.
3. Etablissement d'un lien entre UML et quelques langages standard au niveau système comme le SystemC.
4. Intégration d'UML avec les outils CAO existants et en particulier les outils visant les architectures réconfigurables.
5. Formalisation du l'ordonnanceur SystemC en s'appuyant sur la logique de réécriture et en particulier le langage Maude.

4. Plan du mémoire

Cette thèse est scindée en deux parties : la première partie est consacrée à l'état de l'art et aux travaux voisins qui sont relatifs à l'application d'UML au domaine des MPSOC. Cette partie comprend deux chapitres : le premier chapitre s'intéresse à la conception conjointe matériel logiciel. Nous détaillerons les différents modèles des MPSOC liés à l'application, à l'architecture ainsi que le partitionnement matériel logiciel.

Dans le deuxième chapitre, nous dresserons un état de l'art sur les profils UML les plus répandus dans le monde des SOC et systèmes embarqués.

La seconde partie de cette thèse se donne pour objectif de présenter nos contributions. Elle se compose de cinq chapitres.

Le troisième chapitre détaillera notre profil UML pour estimer les performances au pire de cas d'une application sur MPSOC. Les différents stéréotypes, les techniques d'estimation de temps et de consommation, ainsi que les heuristiques relatives à la transformation du modèle séquentiel vers un modèle concurrent, d'optimisation et d'association seront définis.

Nous proposerons dans le quatrième chapitre une nouvelle approche intégrant la modélisation UML des systèmes mixtes (orientés données/contrôle), la vérification formelle de quelques propriétés du système en se basant sur la logique de réécriture, et la génération du code SystemC à partir des modèles UML.

Dans le cinquième chapitre, nous présenterons un flot qui permet de faire un lien entre UML et les outils CAO de synthèse de haut niveau et de simulation existants visant une architecture reconfigurable.

Dans le sixième chapitre, nous aborderons la problématique de formalisation de l'ordonnanceur SystemC en s'appuyant sur la logique de réécriture et en particulier le langage Maude.

Enfin, dans le dernier chapitre nous conclurons cette thèse en présentant les apports de nos contributions et en dégageant les perspectives.

Première partie

UML et le monde des MPSOCs et systèmes embarqués

Premier chapitre

La conception conjointe matériel logiciel

Sommaire

1. Introduction

Le terme "Codesign" ou La conception conjointe du logiciel et du matériel est apparu au début des années 1990 pour marquer une nouvelle façon de penser la conception des circuits intégrés et systèmes [7]. Ainsi pour concevoir un système d'un coût raisonnable tout en respectant les performances imposées, les concepteurs s'orientaient vers une approche mixte. Une partie était réalisée avec des composants programmables (la partie logicielle). L'autre partie réalisée avec des composants matériels spécifiques à l'application (la partie matérielle). L'utilisation conjointe de ressources logicielles et matérielles nécessitait de nouvelles méthodes de conception pour trouver le meilleur compromis entre parties logicielles et matérielles et pour permettre leur conception simultanément.

Dans les années 1990, de nombreuses équipes de recherches, notamment aux Etats-Unis et en France (laboratoire TIMA, I3S, …) se sont penchées sur le problème du partitionnement logiciel/matériel. Des techniques et algorithmes ont permis de résoudre le problème dans des cas particuliers (systèmes orientés flot de données ou flot de contrôle), mais en se limitant généralement à des architectures très simples (monoprocesseurs). Néanmoins, plusieurs outils spécifiques et liés à une plateforme de simulation (ou d'émulation) ont montré l'intérêt d'aider le concepteur dans cette étape de conception. L'automatisation permettrait de guider le concepteur pour décider du partitionnement.

Malgré ces efforts, on peut constater que le problème n'est pas résolu et reste identique à ce jour. D'une part, le problème est extrêmement difficile et dépend de paramètres technologiques (vitesse, consommation, …), de l'application, de l'architecture et de paramètres économiques (coût de conception et fabrication). Ces derniers sont difficiles à formuler mais aussi difficilement quantifiables et mesurables, contrairement aux autres paramètres. De plus, ils évoluent, ce qui remettent en cause les solutions trouvées il y a une dizaine d'années. Des outils automatiques existent déjà et sont (plus ou moins) performants pour les niveaux d'abstraction assez bas où la séparation est claire entre le matériel et le logiciel. Pour des niveaux plus hauts où la distinction entre ces deux parties ne peut et ne doit pas être faite encore, ces outils manquent, principalement ceux qui offrent un flot complet à partir de la spécification au niveau système de l'application jusqu'à son raffinement vers les outils de niveau RTL. Cette façon de voir un flot de conception séparé pour logiciel et matériel a rapidement fait apparaître deux nouvelles difficultés : la nécessité de concevoir des interfaces spécialisées et optimisées pour faire communiquer logiciel et matériel, et l'intégration et la validation des composants après leur conception.

La tendance des nouveaux environnements de conception conjointe consiste à adopter un modèle de construction basé sur une séparation nette entre les deux vues : fonctionnelle (algorithmes de l'application), architecturale (architecture matérielle d'implémentation) dès les premières étapes du cycle conception- implémentation. Ce modèle de construction particulier (appelé "Y-chart approach" [78]) repose sur un environnent permettant de différencier la spécification des algorithmes décrivant le comportement de l'application, de la spécification de l'architecture supportant leur implémentation et de la spécification de l'étape d'implémentation proprement dite de l'application sur une architecture particulière.

Cette séparation des modèles autorise aussi bien le portage des algorithmes et architectures que leur réutilisation : une même application peut être réutilisée sur une nouvelle architecture, ou être transformée et ré-implémentée sur une architecture existante. Ceci favorise une exploration efficace de l'espace des implémentations possibles de l'algorithme sur l'architecture et par là même une réduction considérable du délai de la recherche de la meilleure implémentation (l'adéquation algorithme/architecture) ainsi qu'une arrivée plus rapide du produit final sur le marché.

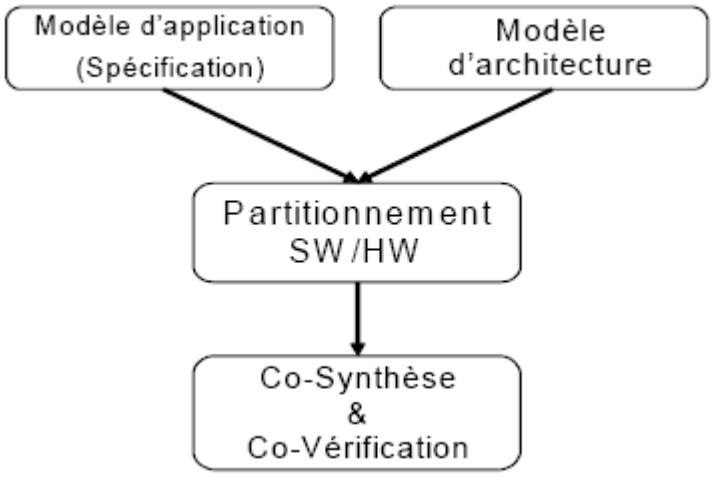


Figure 1.1. Méthode en Y pour les systèmes embarqués [78].

2. Modèles d'application

Le partitionnement opère sur un modèle de l'application (ou une spécification). Le choix du formalisme pour décrire l'application a donc un fort impact sur la qualité des résultats. Ces modèles de description ont en général été développés pour spécifier un comportement ou une structure et/ou vérifier des propriétés. D. Gajski et F. Vahid les définissent dans [55] comme étant un moyen clair permettant de saisir la fonctionnalité désirée d'un système. Etant donné

une séquence de valeurs en entrée, une spécification permet de déterminer ce que devrait être la réponse du système.

Les modèles d'application offrent différents niveaux d'abstraction. La spécification peut être abstraite ou exécutable. Les modèles abstraits définissent un ensemble d'assertions relatives aux propriétés du système comme sa fonctionnalité, sa charge de travail ou ses dimensions physiques, sans exécuter la spécification. Alors que les modèles exécutables représentent l'application elle même ou un modèle de performance à événements discrets de l'application. Ces modèles sont appelés parfois simulations (terme surtout approprié quand le modèle exécutable est clairement distinct du système qu'il modélise).

Les modèles exécutables sont construits à partir d'un modèle de calcul (MOC), qui représente l'ensemble des lois qui régissent l'interaction des composants dans le modèle. Plus généralement, c'est un ensemble de règles abstraites (sémantique) qui représente l'ossature permettant au concepteur de créer des modèles. Les modèles de calcul les plus intéressants pour modéliser des systèmes embarqués doivent être en mesure de gérer la concurrence, la réactivité et le temps. En effet, les systèmes embarqués consistent typiquement en un ensemble de composants opérant simultanément et ayant de multiples sources de stimuli simultanés. De plus, ils opèrent dans un environnement réel où le temps de réponse à leurs stimuli est aussi important que l'exactitude de la réponse.

Le choix du modèle de calcul dépend fortement du type de modèle à construire. Par exemple, pour un système purement de calcul qui transforme un bloc fini de données en un autre bloc fini, la sémantique impérative commune des langages de programmation comme C, C++, Java et Matlab serait adéquate.

Pour modéliser un système mécanique, la sémantique doit être capable de gérer la concurrence et la continuité temporelle, dans ce cas un modèle de calcul à temps continu comme celui trouvé dans Simulink, Saber, ADS et VHDL-AMS est plus approprié.

Une différence essentielle entre les différents modèles de calcul est donc leur représentation du temps : temps continu, temps discret, temps à ordre partiel, etc. Plusieurs autres manières de les distinguer existent selon par exemple, le type de l'application cible.

Nous considérons dans la suite deux grandes familles de modèles de calcul, une orientée contrôle et une autre orientée données. Nous entendons par contrôle tout ce qui est régi par le temps en terme d'évènements discrets, et par traitement tout ce qui est régi par les données elles mêmes. Nous exposons quelques uns de leurs avantages et inconvénients pour présenter ensuite quelques modèles hybrides construits à partir des modèles classiques et finir par citer quelques environnements de modélisation multi paradigmes.

2.1. Modèles orientés données

2.1.1. Modèles à base de langages impératifs

Le langage C est le langage le plus utilisé dans la pratique pour exprimer les comportements des systèmes embarqués. D'autres langages impératifs gagnent du terrain comme C++ ou Java du fait de la réutilisation potentielle apportée par le concept d'objet. Ces langages ne permettent pas de révéler un parallélisme potentiel aisément exploitable par les méthodes de conception du fait de leur nature séquentielle. Il existe des compilateurs permettant d'extraire le parallélisme au niveau instruction (ILP), donc ayant une granularité très fine. Ces compilateurs ne sont cependant pas capables d'exploiter le parallélisme à gros grain qui est celui considéré par le concepteur dans une phase initiale de partitionnement ou de conception système. Un langage de type C (qui peut être étendu pour représenter par exemple des processus concurrents) est ainsi utilisé dans les approches [48, 65, 75, 93, 125]. D'autres approches comme [79] et [41] utilisent l'environnement Simulink de Matlab comme langage de spécification.

2.1.2. Modèles à base de graphes de flots

Dans ces modèles, ce sont les données qui fixent l'exécution.

- ***Graphes de flot de données***

Les graphes de flot de données (DFG) ont été initialement utilisés pour la synthèse de haut niveau et la compilation logicielle [58]. Ils sont formés de nœuds, qui représentent les opérations, reliés par des arcs orientés qui représentent les dépendances (ou chemins) de données. On n'exécute un nœud que lorsque toutes ses données en entrée sont disponibles. Les nœuds peuvent avoir des granularités différentes qui varient de la granularité de la simple instruction à celle de la fonction.

- ***Graphes de flots mixtes de données et de contrôles***

Les graphes de flots de données et de contrôles (CDFG) sont très appropriés pour décrire les manipulations de données, les communications ainsi que de simples contrôles locaux. Le contrôle est enfoui dans le flot de données au moyen de nœuds de branchement et de fusion et souvent au moyen de structures de boucles spéciales. L'expression du flot de contrôle est souvent limitée et n'est pas en mesure d'exprimer aisément les synchronisations (hors synchronisation par échange de données), ou les interruptions.

On parle aussi de graphes de flots de données et de contrôle hiérarchiques HCDFG [90], où le niveau de la hiérarchie le plus bas est le DFG avec des nœuds élémentaires de traitement

et des liens de communication. A un niveau plus haut on retrouve les CDFG avec des nœuds test (*if, case, loop*...) et des appels à des DFG. Au niveau le plus haut, le HCDFG peut contenir des nœuds test, d'autres HCDFG et des CDFG.

2.1.3. Processus communicants concurrents

Les modèles basés sur ces processus permettent de décrire le parallélisme d'une application tout en étant indépendant d'une implémentation logicielle ou matérielle. Ces modèles sont bien adaptés à la description de systèmes de télécommunication.

- ***Le modèle basé sur les réseaux de processus de Kahn***

C'est un modèle de processus concurrents qui s'échangent des données au travers des liens de communication point à point bufférisés (FIFO de taille infinie).

Dans ce modèle [73], les arcs représentent des séquences de données (*tokens*), et les entités représentent les fonctions qui transforment les séquences d'entrée en séquences de sortie. Pour assurer le déterminisme, quelques restrictions sur ces fonctions ont été introduites comme les lectures bloquantes (condition de mono-tonicité). Ces réseaux de processus sont faiblement couplés et donc relativement aisés à paralléliser ou à distribuer. Ils peuvent être efficacement implémentés aussi bien en logiciel qu'en matériel et donc les options d'implémentation restent ouvertes.

Ce modèle est assez souple mais il a l'inconvénient d'être difficile à ordonnancer (ordonnancement dynamique en général). Une autre limitation de ce modèle est dans la spécification de la logique de contrôle et de la réactivité (l'interaction avec un utilisateur par exemple).

- ***Le modèle SDF***

Le modèle SDF est un sous ensemble restreint des réseaux de processus de Kahn, où l'ordonnancement des exécutions, séquentielles ou parallèles, est calculé statiquement. Cette propriété fait de SDF un formalisme de spécification extrêmement utile pour le matériel et pour le logiciel temps-réel embarqué.

2.1.4. Processus communicants séquentiels

Ces processus communiquent par des actions atomiques et instantanées appelés *rendez-vous* (ou parfois, passage synchrone de message). Comme exemples de modèles avec *rendez-vous,* on peut citer le modèle CSP de Hoare [69].

Ce modèle de calcul a été utilisé dans de nombreux langages de programmation comme Lotos et Occam. Les modèles de calcul basés sur les processus communicants concurrents et

les processus séquentiels communicants possèdent un modèle de temps abstrait dit à ordre partiel. Cette notion de temps à ordre partiel est une sorte de contrainte imposée par la causalité.

2.2. Modèles orientés contrôle

2.2.1. Les réseaux de Petri avec notion de temps

Pour bénéficier de la puissance des réseaux de Petri dans le domaine des applications temps-réel, différentes extensions ont été introduites aux réseaux de Petri classiques [103] en particulier pour prendre en compte la notion de temps. Ces modèles restent identiques en forme, mais diffèrent par la sémantique des opérations ou par leur manière d'annoter le temps.

Un réseau de Petri ordinaire décrit une relation de causalité entre les évènements, ce qui les ordonne dans le temps. Le temps est pris en compte de manière qualitative. Plusieurs modèles fondés sur les réseaux de Petri permettent de prendre en compte le temps de manière quantitative en associant des durées (des temporisations) aux transitions ou aux places comme les réseaux de Petri temporisés [114], les réseaux de Petri temporels et les réseaux de Petri stochastiques [46, 116].

Les réseaux de Petri temporisés ainsi que les réseaux de Petri stochastiques sont plutôt utilisés dans la communauté analyse de performances pour construire des modèles réalistes afin d'évaluer, par simulation, des systèmes à évènements discrets. Les réseaux temporels, eux, sont utilisés pour la construction de graphes de couverture des états accessibles du système et la vérification formelle.

Une des faiblesses de ce modèle est la quasi-absence de commodités pour décrire l'aspect communication de données. Quelques modèles fondés sur les réseaux de Petri ont permis d'obtenir une description compacte des parties traitements et données. Ces modèles, dits de haut niveau, attachent une partie des données aux jetons. Parmi ces modèles on cite les réseaux de Petri colorés [126], les réseaux Prédicat-Transitions et les réseaux à objet.

2.2.2. Les machines à états finis

Les entités de ce modèle représentent les états, et les connexions représentent les transitions entre états. L'exécution est une succession strictement ordonnée d'états et de transitions. Les machines à états finis (FSM) sont des modèles bien adaptés pour exprimer la logique de contrôle et pour construire des modèles de modes (systèmes avec des modes d'opération distincts, où le comportement est différent pour chaque mode : intéressant pour

exprimer différent modes de reconfiguration dans un système reconfigurable dynamiquement).

Les modèles à base de FSM se prêtent bien à une analyse formelle et peuvent, de ce fait, être utilisés pour vérifier l'absence de comportements inattendus du système.
L'une des faiblesses de ce modèle réside dans le fait que le nombre d'états peut devenir rapidement excessivement grand en fonction de la complexité des systèmes.

2.2.3. Modèles réactifs synchrones

Le modèle Réactif Synchrone (RS) a été très utilisé pour la co-conception logicielle/matérielle. Parmi les points forts des langages basés sur le modèle RS, on cite leur syntaxe simple à utiliser et leur sémantique succincte et formelle (ce qui a permis de développer des outils de simulation et de vérification efficaces). Dans le modèle RS, les entités représentent des relations entre les valeurs en entrées et celles en sorties à chaque *réaction* du système.

Comme exemple de langages basés sur le modèle de calcul RS, on cite Esterel [14], Signal [13], Lustre [33] et Argos [96].

Les modèles RS se prêtent parfaitement pour décrire des applications comprenant de la logique de contrôle complexe et concurrente. Grâce à l'hypothèse de forte synchronisation qu'offrent ces modèles, ils peuvent cibler des applications temps-réel avec contrainte de sûreté. Cependant, à cause de cette forte synchronisation, quelques applications peuvent être sur-spécifiées, même avec un langage RS multi-horloges comme Signal, limitant ainsi les alternatives d'implémentation. Par ailleurs, disposer d'une horloge unique et totalement synchronisée (Esterel ou Lustre) couvrant la totalité des nœuds d'un système temps-réel distribué peut être extrêmement coûteux ou même simplement infaisable.

2.2.4. Graphes de tâches

Pour tenir compte des évolutions des SOC qui intègrent généralement plusieurs processeurs et au moins un système d'exploitation, différentes approches considèrent un modèle de type graphe de tâches avec un ordonnancement statique en ligne préemptif, par exemple [43, 81]. Les tâches sont généralement de forte granularité et traitées comme des boites noires avec des conditions de précédence. C'est le système d'exploitation qui gère l'exécution des tâches en utilisant ses propres structures de contrôle.

2.3. Modèles hybrides

Une approche de modélisation unifiée cherche à définir un modèle de calcul concurrent qui servirait à modéliser toutes les composantes d'un système. Ceci pourrait être possible en combinant tous les modèles précédents, mais une telle combinaison serait extrêmement complexe, difficile à définir et à utiliser (notons par exemple l'expérience VCC de Cadence), et les outils de synthèse et de simulation seraient aussi difficiles à concevoir.

Plusieurs modèles hybrides ont vu le jour, essayant de profiter des avantages de certains modèles de calcul pour masquer les inconvénients des autres. Parmi ces modèles on peut citer:

2.3.1. Les StateCharts et les CFSM

Les StateCharts [9] comme les CFSM [7] peuvent être vu comme la combinaison des machines à états finis avec le modèle réactif synchrone. Les CFSM ont l'avantage d'être localement synchrones, globalement asynchrones ce qui simplifie grandement le partitionnement entre modules logiciels et matériels. L'outil POLIS [7] de co-conception logicielle/matérielle élaboré à l'Université de Berkeley est basé sur le modèle CFSM. Cet outil utilise *Esterel* pour décrire chaque CFSM, puis les interconnecte pour former un réseau hiérarchique de machines d'états finis réactives. Cet outil à été la base d'un autre outil, commercial cette fois-ci, de co-conception logicielle/matérielle appelé VCC qui a été commercialisé par Cadence.

2.3.2. Le modèle SDL

Le modèle SDL peut être vu comme la combinaison des machines à états finis avec les réseaux de processus communicants. C'est un formalisme graphique, basé sur les processus communicants qui permet de décrire le parallélisme d'une application tout en étant indépendant d'une implémentation logicielle ou matérielle. Chaque processus est une machine à états finis étendue par des variables. Le formalisme SDL est adapté à la description des protocoles de télécommunication. Parmi les approches de conception utilisant le modèle SDL, on peut citer [41,133].

2.3.3. Le modèle PSM

Le modèle PSM associe les machines d'états finis et les processus séquentiels communicants (CSP). Ce modèle s'exprime dans le langage SpecCharts, qui consiste en une extension de VHDL. L'approche SpecSyn détaillée dans [58] propose un environnement de conception au niveau système utilisant le modèle PSM. Un autre langage issu du modèle PSM

est le langage SpecC [56]. Ce langage se base sur un réseau hiérarchique de comportements et de canaux. C'est surtout une extension du C-ANSI pour la conception matérielle en intégrant les concepts de temps, de concurrence, de synchronisation.

2.3.4. Modèles au niveau transactionnel

- ***SystemC***

SystemC est une librairie de classes construite à base du standard C++ et d'un noyau de simulation par événements [117]. Cette librairie de classes permet de modéliser des systèmes logiciels/matériels avec la possibilité de décrire la concurrence, une notion du temps et quelques types de données matériels spécifiques. Plusieurs outils commerciaux utilisent SystemC dans leurs flots de conception, parmi lesquels on trouve Cocentric de Synopsys et ConvergenSC de CoWare.

- ***SystemVerilog***

SystemVerilog est un nouveau langage considéré comme une extension du langage de description matériel Verilog afin de supporter de plus hauts niveaux d'abstraction pour la modélisation et la vérification [119].

2.3.5. Modèles orientés objet

UML est le langage de modélisation orienté objet émergeant actuellement et utilisé principalement pour le développement de systèmes logiciels [17]. Il consiste en un ensemble de blocs de base, de règles qui dictent l'utilisation et l'arrangement de ces blocs et de mécanismes qui améliorent la qualité des modèles UML. Ce langage est en train de gagner l'attention de la communauté s'intéressant au logiciel embarqué qui le considère comme étant une solution possible pour travailler à un niveau d'abstraction plus élevé.

Plusieurs travaux ont été réalisés ces dernières années pour étudier la possibilité d'utiliser UML comme langage de base pour la spécification des systèmes embarqués [63, 97], ou plus particulièrement des plateformes embarquées [36].

L'approche détaillée dans [44] combine les modèles UML et SDL. La spécification au plus haut niveau est réalisée en UML et le comportement de chaque module est spécifié en SDL.

2.4. Les environnements de modélisation multi-paradigmes

2.4.1. Ptolemy

Une des initiatives les plus remarquables sur la définition d'un environnement d'accueil de différents paradigmes est le système Ptolemy [107] qui définit des sémantiques précises pour interfacer les différents modèles (domaines). Ptolemy est avant tout un environnement de modélisation et de simulation. Parmi les améliorations apportées dans Ptolemy II [106], on cite la notion de polymorphisme entre domaines (où les composants peuvent être conçus de manière à opérer dans des domaines différents), la notion de modèles modaux (où les FSM sont combinés hiérarchiquement avec d'autres modèles de calcul) et la notion de domaine à temps continu (qui, une fois combinée avec la modélisation modale, permet la modélisation de systèmes hybrides).

2.4.2. Metropolis

Cet environnement est construit à partir d'un méta-modèle basé sur un modèle de calcul de type CSP. Dans ce méta-modèle on peut englober un ensemble de modèles de calcul comme étant des cas particuliers qui en dérivent. Metropolis [8] propose un environnement de conception de systèmes hétérogènes qui intègrent des outils de simulation, de synthèse, d'analyse et de vérification.

2.4.3. Rosetta

Il s'agit d'un langage développé dans le cadre de l'initiative SLDL. Rosetta [2] permet d'intégrer plusieurs théories de domaines en une structure à sémantique commune et d'organiser et de décomposer les contraintes au niveau système sur les différents domaines de sémantique différente que peut comporter le système. Notons qu'une initiative de standardisation du langage *Rosetta* supportée par Accellera est en cours.

3. Modèles d'architecture

En fonction de l'adéquation à une spécification particulière et de la disponibilité des options de personnalisation, les plateformes peuvent être classées en quatre catégories [72]. Le tableau 1.1 résume quatre types de plateformes.

3.1. Plateformes pour applications entières

Ces plateformes permettent aux concepteurs de développer des applications complètes sur des architectures matériel/logiciel. Pour faciliter la conception de produits dérivés (derivative-product design), les plateformes pour applications entières contiennent généralement une bibliothèque de modules matériels où chaque module possède plusieurs schémas du design.

Les concepteurs peuvent choisir à partir de cette bibliothèque des modules pour construire des systèmes plus complexes.

3.2. Plateformes centrées au tour du processeur

Ces plateformes se concentrent beaucoup plus sur des cœurs du processeur spécifiques et se concentrent aussi sur le logiciel permettant l'accès aux processeurs. Les concepteurs ont souvent besoin d'autres blocs matériels spécifiques à l'application et dans certains cas, un système d'exploitation temps réel (RTOS) différent afin de réaliser des applications complètes. ImprovJazz et ST StarCore illustrent bien ce type de plateformes.

3.3. Plateformes centrées au tour de communication

Cette approche offre aux consommateurs une plateforme de communication optimisée, et personnalisée, pour une application spécifique. Là, encore le principe de conception des produits dérivés est requis pour inclure de nouveaux composants afin d'obtenir une application complète. Les architectures Sonics et PalmChip sont deux exemples de ce type de plateformes.

3.4. Plateformes entièrement programmables

Ces plateformes sont similaires aux deux premières catégories, sauf que ces plateformes incluent également une logique embarquée reconfigurable. L'ajout de cette logique programmable permet aux concepteurs de personnaliser la plateforme avec le matériel et le logiciel. Des exemples incluent les plateformes Triscend, Excalibur Altera et Xilinx P-FPGA.

Type de la plateforme	Exemple
SOC pour applications entières	- Nexperia: Philips Semiconductors - Open Multimedia Applications Platform (OMAP): TI
SOC centrés au tour du processeur	- Micropack: ARM
SOC centrés au tour de communication	- uNetwork: Sonics - AMBA bus architecture: ARM
SOC entièrement programmables	- Virtex-II Pro: Xilinx

Tableau 1.1. Les différents types des plateformes.

4. Partitionnement logiciel/matériel

Le partitionnement d'une spécification est un problème complexe (il s'agit d'un problème NP-difficile) du fait du grand nombre de paramètres à considérer.

L'une des principales étapes de conception est celle de partitionnement. Dans le cas général, cette étape se subdivise en trois parties principales:

- Une partie qui effectue l'allocation en choisissant le type et le nombre de ressources matérielles et logicielles nécessaires.
- Une partie qui effectue le partitionnement spatial en prenant des décisions sur l'affectation (assignation) des tâches sur le matériel et le logiciel.
- Une partie qui effectue l'ordonnancement des exécutions des tâches et des communications.

Cependant, dans le cas des architectures réconfigurables, une nouvelle partie s'intercale entre partitionnement et ordonnancement:

- Une partie qui effectue le partitionnement temporel en agençant les tâches matérielles dans des segments temporels (ou contextes) différents.

Du fait de la complexité du processus de partitionnement, plusieurs approches ont préféré laisser au concepteur le soin de déterminer une architecture (étape d'allocation) et de choisir une répartition des fonctionnalités de l'application sur celle-ci (étape de partitionnement spatial). Dans ce cas, les travaux visent à apporter une aide à la conception par l'intermédiaire d'un environnement de Co-simulation qui permet d'animer le modèle de l'application en fonction des paramètres de l'architecture choisie. Ce type d'approche est utilisé dans le projet POLIS [7] qui a servi de base au développement de l'outil VCC de Cadence, et dans

l'environnement CoWare. Dans la suite, nous considérons les approches de partitionnement automatique en détaillant les différentes étapes listées plus haut.

4.1. Etape d'allocation

En entend par allocation l'étape qui consiste à trouver le meilleur ensemble de composants pour implémenter les fonctionnalités d'un système. Cependant, il faudrait choisir ces composants parmi des centaines de composants. A une extrême, on trouve des composants matériels très rapides, spécialisés, mais très coûteux comme les ASICs. A l'autre extrême, on a des composants logiciels flexibles, moins chers mais moins rapides comme les processeurs à usage général. Entre ces deux extrêmes, existent d'innombrables composants (dont les composants réconfigurables) qui offrent différents compromis en termes de coût, performance, flexibilité, consommation, taille et fiabilité par exemple.

L'étape d'allocation est l'une des étapes les plus déterminantes dans le processus de partitionnement. En effet, elle fixe et dimensionne très tôt dans le processus de conception l'architecture du système et en particulier ses performances maximum alors que des dysfonctionnements (non-respect des contraintes ou des débits) ne sont souvent repérés que très tard et après réalisation du système complet. C'est pour cela que dans l'industrie, ou le mot d'ordre est '*First Silicon Right*' (juste et du premier coup), les concepteurs, même chevronnés, tendent à sur-dimensionner l'architecture dès l'étape d'allocation en choisissant des processeurs plus rapides, des bus avec des débits plus importants ce qui fait qu'en général, on remet en cause très rarement les décisions prises à ce niveau de la conception.

4.2. Etape de partitionnement Spatial

Le problème du partitionnement se limitait souvent au partitionnement spatial ou à l'affectation des différents composants fonctionnels du modèle de l'application sur les composants de l'architecture allouée. Une formulation intéressante de ce problème est développée dans [57] et peut être résumée comme étant un problème de partitionnement de trois types d'objets de spécification sur les composants de l'architecture. Ces objets sont :

- Les variables: Ces variables contiennent les valeurs des données et doivent être assignées à des composants mémoire.
- Les comportements: Ces objets transforment les valeurs des données et doivent être assignées à des unités de traitement.
- Les canaux: Ces objets transfèrent les données d'un comportement à un autre et doivent être assignées à des bus ou à des réseaux d'interconnexion plus évolués.

La plupart des études concentrent leurs efforts sur le partitionnement des comportements sur les unités de traitement de l'architecture en sous-entendant que le partitionnement des autres objets en découle naturellement.

Ce partitionnement nécessite d'abord de définir la granularité de ces comportements. Elle dépend de la granularité utilisée lors de la spécification et définit, comme on l'a vu dans le premier chapitre, le plus petit objet fonctionnel indivisible utilisé durant le partitionnement, comme les tâches, les processus, les boucles, les blocs de traitement, les opérations arithmétiques, les expressions booléennes. Cependant, vu la complexité croissante des applications traitées, on tend naturellement à considérer des granularités assez importantes afin de limiter le nombre d'objets ainsi que leurs interactions.

La plupart des approches actuelles partent de spécifications sous forme de graphe de tâches de granularité importante. Ensuite, le partitionnement impose de déterminer la fonction de coût qui permet de mesurer la qualité d'une solution donnée et guider l'algorithme de partitionnement vers une meilleure solution.

Et enfin, il faut choisir un algorithme de partitionnement efficace pour explorer l'immense espace de solutions et en retenir une ou plusieurs qui vérifient les contraintes imposées au départ et optimise la fonction de coût. Ce problème d'affectation (comme le problème d'allocation et d'ordonnancement dans le cas multiprocesseurs hétérogènes) est NP-difficile. Aussi, les approches optimales basées sur la programmation linéaire en nombres entiers ou l'exploration exhaustive [93] ne peuvent être appliquées qu'à des problèmes assez réduits. Cependant, pour garantir des temps de calcul raisonnables, il est nécessaire de sacrifier la garantie d'optimalité et dans ce cas, des algorithmes d'améliorations itératives comme les algorithmes génétiques ou le recuit simulé ont été utilisés avec succès pour résoudre le problème d'affectation.

4.3. Etape de partitionnement temporel

C'est une étape nouvelle dans le processus de partitionnement afin de tenir compte de la capacité de reconfiguration dynamique de certaines architectures.

Comme la taille du composant matériel est souvent beaucoup plus petite que la somme des ressources nécessaires pour toutes les configurations, au lieu d'essayer de redimensionner le matériel ou de diminuer les tailles des configurations, on charge (et décharge) les configurations dans le composant lorsqu'on en a besoin. Cette technique permet le partage en temps de la même ressource matérielle.

Cette étape consiste donc à grouper les tâches (ou fonctions de forte granularité), déjà assignées à une implémentation matérielle sur le composant reconfigurable, au sein de contextes de configuration. Ces contextes vont être chargés puis déchargés de façon séquentielle sur le reconfigurable selon un ordre bien défini qui sera déterminé lors de la prochaine étape d'ordonnancement.

Généralement, dans le cas d'architectures à reconfiguration partielle (Virtex, AT40K, AT6000...), le temps de reconfiguration de ces contextes dépend linéairement de leur taille.

4.4. Etape d'ordonnancement

Cette étape a pour but d'ordonnancer les exécutions des tâches, les communications et les reconfigurations du FPGA et de vérifier que les choix de partitionnement et d'allocation respectent les contraintes imposées. Le problème d'ordonnancement multiprocesseurs hétérogènes, même statique et hors ligne, est un problème NP-complet dont la complexité augmente encore en considérant des composants réconfigurables dynamiquement et extrêmement parallèles.

5. Conclusion

Dans ce chapitre, nous avons présenté les différents modèles d'application et d'architectures des MPSOC ainsi que le partitionnement matériel logiciel. On peut constater qu'il y a une diversité des modèles et d'algorithmes. Le choix d'un tel modèle(s) ou algorithme(s) dépend de plusieurs facteurs comme le type d'application à modéliser, l'objectif primaire du concepteur (analyse, synthèse, etc.…), les contraintes de conception, et la disponibilité des plateformes.

Par conséquent, l'expérience et la familiarité du concepteur avec un tel langage, modèle et architecture jouent des rôles primordiaux pour minimiser l'effort de conception.

En résumé, on peut dire que la conception des MPSOC nécessite une synergie de plusieurs domaines d'ingénieries dans un cadre unifié qui permet au premier temps une modélisation abstraite de tout les aspects liés aux MPSOC puis à travers des raffinements successifs, des implémentations matérielle et logiciel auront lieu.

Deuxième chapitre

UML pour la conception des SOCs

Sommaire

1. UML
2. Profils UML2 pour SOCs et systèmes embarqués
3. Discussion
4. Conclusion

1. UML

Pour répondre aux besoins de documentation et de spécification de logiciels, de nombreux langages graphiques ont vu le jour, en particulier au sein de la communauté des objets.

Afin de privilégier la réutilisation de composants, d'architectures, ou de favoriser l'interconnexion entre des systèmes modélisés avec des notations différentes, il apparaît évident qu'une notation consensuelle est nécessaire. De ce constat est née la notation UML [17].

UML (Unified Modeling Language) est un langage de modélisation visuelle utilisé pour documenter, spécifier, et visualiser graphiquement les aspects d'un système logiciel. Comme son nom l'indique, ce langage est le résultat d'une longue maturation. Il est né de la fusion de plusieurs langages de modélisation qui ont le plus influencé la modélisation objet au milieu des années 90, notamment les méthodes OMT et OOSE.

L'ambition d'UML est de rassembler en une seule notation les meilleures caractéristiques des différents langages de modélisation à objets. Cette unification a aussi pour effet de donner une masse critique à UML.

En tant que standard de l'OMG, et en tant que successeur de notations déjà bien implémentées, UML jouit d'une popularité à la fois dans l'industrie du logiciel et dans le monde de la recherche scientifique. L'approche UML ne représente pas une méthode à proprement parler, mais plutôt un langage de représentation de processus.

En d'autres termes, UML n'impose pas une démarche pour l'enchaînement des activités d'une organisation, mais essentiellement un support de communication, qui facilite la représentation et la compréhension de ces activités. Sa notation graphique permet d'exprimer visuellement une solution, et facilite la comparaison et l'évaluation des différentes solutions proposées.

Le langage UML facilite la communication entre clients et concepteurs, et entre équipes de concepteurs. Sa syntaxe étant bien définie sous forme de diagrammes UML, cela accélère le développement des outils, et permet le passage d'une spécification UML de haut niveau vers la génération d'un code exécutable. Principalement, UML définit deux catégories de diagrammes : structurels et comportementaux. L'aspect structurel regroupe les notions de base permettant la représentation de la structure statique du système. Il spécifie l'organisation interne des différents éléments du système, ainsi que leurs interactions possibles. Parmi les diagrammes structurels les plus utilisés dans UML nous citons les diagrammes de classe et les diagrammes de composant. L'aspect comportemental permet quant à lui de modéliser la

dynamique du système en spécifiant comment les valeurs ou les états des objets changent dans le temps. Dans ce contexte, UML permet la spécification du comportement d'un objet individuel, ou d'un ensemble d'objets coopérant. Le premier type de comportement est généralement représenté par des Statecharts ou des diagrammes d'activités, tandis que le deuxième est représenté par des diagrammes de séquence ou des diagrammes de temps.

L'aspect formel des notations UML permet généralement de limiter les ambiguïtés et les incompréhensions des modèles. La véritable force de la modélisation UML est due au fait qu'elle repose sur un méta-modèle qui normalise la sémantique des différents concepts proposés. Cependant, la modélisation graphique proposée par le langage UML n'est généralement pas suffisante pour donner une spécification bien précise et non ambiguë du système étudié. Il y a toujours un besoin pour spécifier des contraintes supplémentaires sur les différents objets du modèle et leurs modes d'interaction. De telles contraintes ont été souvent décrites dans un langage naturel, mais cette description peut conduire à des confusions et des incompréhensions du modèle. Elles peuvent être également décrites dans des langages formels, mais l'inconvénient de ces langages est qu'ils sont utilisables par les personnes ayant une formation mathématique et formelle, mais difficile à manipuler par les industriels ou les concepteurs et développeurs des systèmes. Pour résoudre ce problème, un langage appelé OCL (Object Constraint Language) est développé pour définir un compromis entre le langage naturel et les descriptions formelles. L'introduction du langage OCL a permis d'enrichir la modélisation UML en spécifiant des contraintes supplémentaires sur le comportement du système. C'est un langage déclaratif, textuel et formel qui joue un rôle important dans la phase d'analyse dans le cycle de développement du système.

Depuis sa standardisation par l'OMG en 1997, plusieurs versions d'UML ont été proposées. La dernière version est UML2.0 standardisée en juin 2003 qui fournit des éléments de modélisation et une sémantique plus étendue et systématique que les versions précédentes. Pour la spécification des contraintes sur les objets UML2.0, une version plus adaptée du langage OCL (OCL2.0) est aussi développée en 2004 pour assurer la conformité des deux langages. Les contraintes de bon fonctionnement des SOC, ont imposé une démarche méthodique pour le développement de ces systèmes. Dans ce contexte, les processus itératifs et les méthodes orientés objets ont remplacé les processus en cascade et les méthodes procédurales. Il s'avère que si quelques modèles peuvent être directement étudiés par le langage UML, d'autres nécessitent une spécialisation liée à leur classe d'application. Cette nécessité a conduit vers l'introduction des extensions au langage UML qui sont reconnues par la communauté UML et rendues possibles via la notion du profil.

Un profil correspond au regroupement d'extensions et de spécialisation du langage UML du point de vue notation et sémantique. Ceci est principalement réalisé via le concept de stéréotype qui représente la définition d'une propriété supplémentaire appliquée sur un élément standard d'UML (classe, association, attribut, etc.).

Dans le domaine de la Co-conception des SOC, plusieurs profils ont été proposés pour permettre la prise en compte des caractéristiques de base de ces systèmes dans un modèle UML, aussi bien sur le plan modélisation et analyse que sur le plan validation et génération du code exécutable. Dans ce qui suit, on présente les profils UML pour les systèmes embarqués et les SOC les plus répandus.

2. Profils UML2 pour SOCs et systèmes embarqués

2.1. SysML

SysML (System Modeling Language) est le résultat d'une initiative conjointe de l'OMG et l'INCOSE (International Council on Systems Engineering) [118]. Il réutilise un sous-ensemble d'UML 2.0 et fournit les extensions nécessaires pour l'ingénierie des systèmes. SysML supporte la spécification, l'analyse, la conception, la vérification et la validation d'une vaste gamme de systèmes hétérogènes complexes, qui ne sont pas forcément orientés logiciel. Il est destiné à unifier les divers langages de modélisation des systèmes actuellement utilisés par les ingénieurs. Comme le montre la figure 2.1, l'ensemble de méta-classes UML à être réutilisés sont fusionnés en un seul méta paquetage appelé UML4SysML. Le profil SysML peut être appliqué soit "strictement", où, seules les méta-classes UML référencés par SysML sont disponibles à l'utilisateur de ce modèle ou "non-strictement" où les méta-classes UML supplémentaires qui ne sont pas explicitement référencées mais sont disponibles.

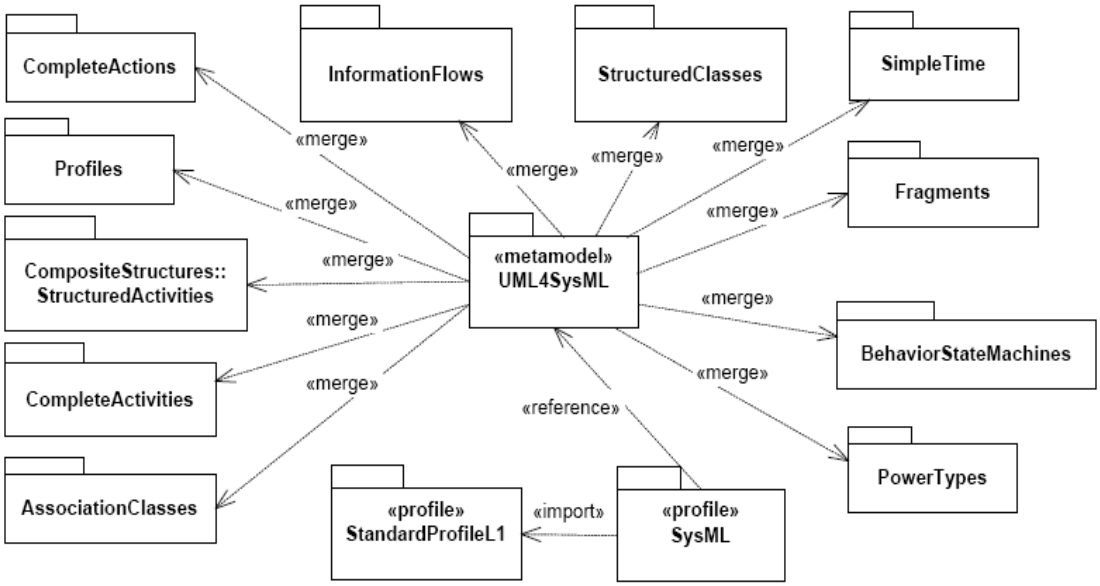


Figure 2.1. Architecture du SysML [118].

SysML introduit deux nouveaux diagrammes (Figure 2.2): le diagramme des exigences et le diagramme paramétrique. Le diagramme des exigences permet à l'ingénieur système de modéliser les exigences et les relier à d'autres éléments du modèle qui les satisfont ou vérifient (figure 2.3). Le diagramme paramétrique est utilisé pour modéliser les paramètres des systèmes de et les relier les uns aux autres. La définition de bloc, le bloc interne, et les diagrammes d'activité sont similaires aux diagrammes de classes, diagramme de structure composite, et diagramme d'activité d'UML2, respectivement, avec quelques extensions. Nous notons en particulier les concepts de l'assemblage et le port de flot (FlowPort) pour les diagrammes composites, et le mécanisme de contrôle d'exécution des actions pour les diagrammes d'activité (par exemple, les actions en cours d'exécution peuvent être désactivées). SysML n'utilise pas le diagramme objets d'UML, le diagramme de communication, le diagramme d'aperçu interactionnel, le diagramme temporel, et le diagramme de déploiement. Dans le cas des diagrammes de déploiement, le déploiement du logiciel sur le matériel peut être représenté dans le diagramme de bloc de SysML interne en utilisant le concept de l'allocation qui est une forme plus abstraite que le déploiement fourni par UML.

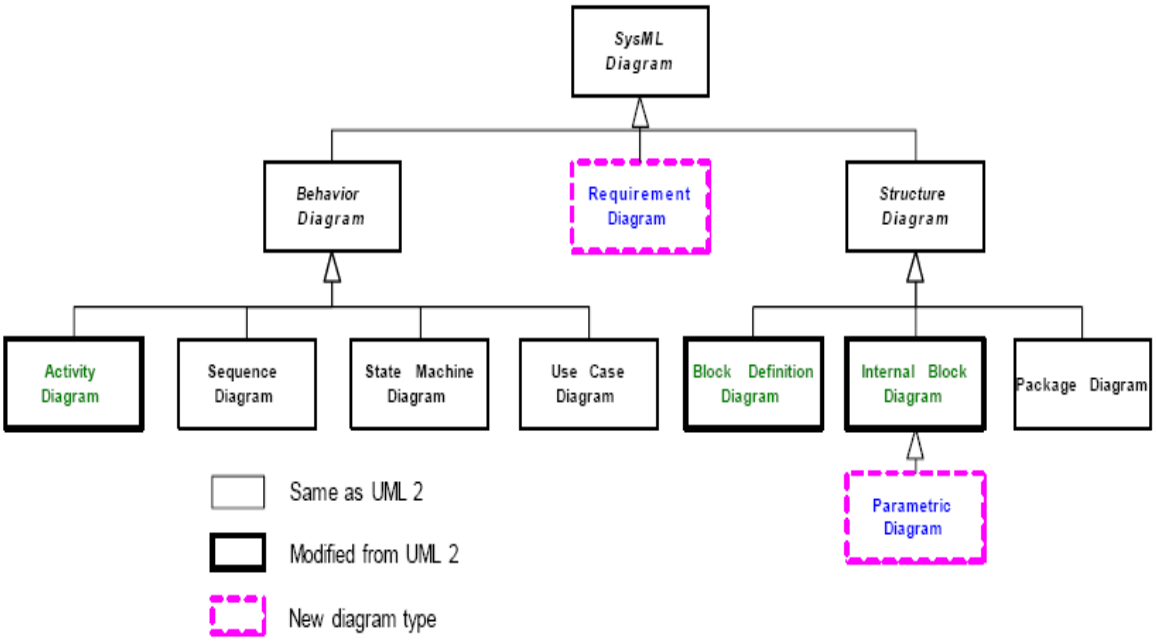


Figure 2.2. Taxonomie de diagrammes SysML [118].

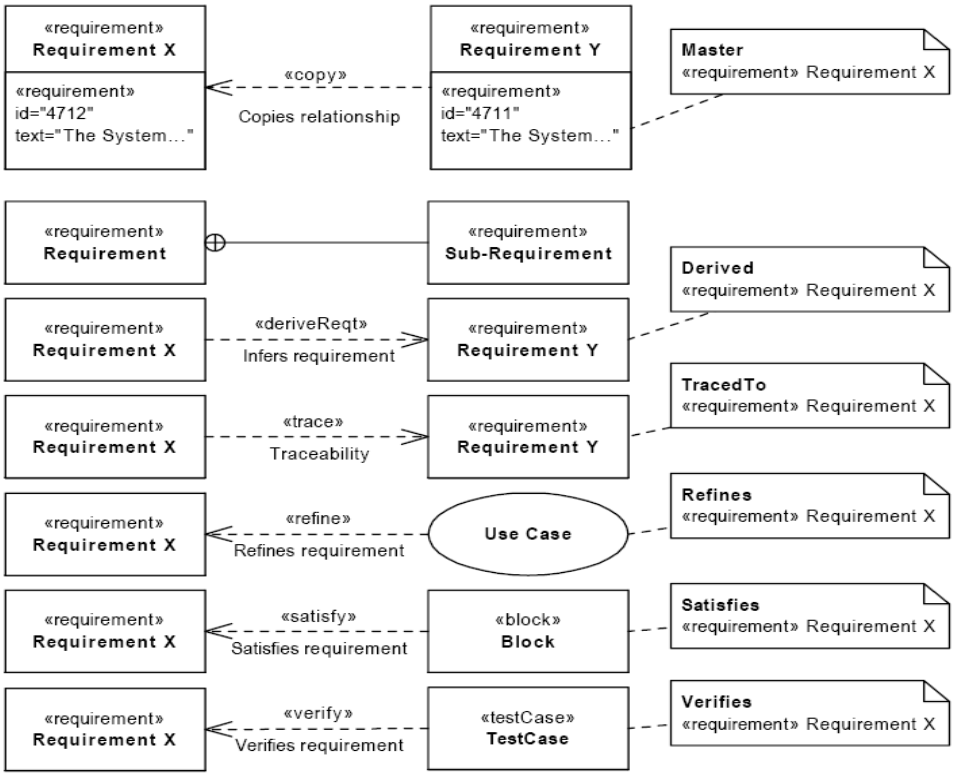


Figure 2.3. Les exigences en SysML [118].

2.2. MARTE (Modeling and Analysis of Real Time Embedded Systems)

Il est défini par le Groupe de travail ProMARTE et est voté à l'OMG pour le développement et l'analyse dirigée par les modèles des systèmes embarqués temps réel. MARTE vise à remplacer le profil UML SPT [123]. Il est basé sur le méta-modèle UML2.0, OCL2 et MOF 2.0 QVT. Comme illustrée en figure 2.4, l'architecture MARTE est fondée sur quatre paquetages: le paquetage de base, le modèle de conception, le modèle d'analyse, et les annexes. Le paquetage de base comprend le profil des NFPs pour la modélisation des propriétés non fonctionnelles. Une NFP (voir figure 2.5), peut être soit élémentaire ou complexe, qualitative ou quantitative. Une valeur NFP peut être spécifiée comme une valeur constante, une variable ou une expression; le profil TIME pour la modélisation du temps logique et physique ; le profil GRM pour la modélisation générique de ressources. Le GRM est détaillé par le biais de DRM pour la modélisation détaillée de ressources ; le profil GCM pour la modélisation de composants génériques et le profil ALLOC pour modéliser l'association matériel/logiciel. Le paquetage de modèle de conception représente le profil de base, il englobe le profil RTEMOCC pour les modèles de calcul et de communication temps réels. Ce dernier est basé sur le concept Runit qui combine entre les paradigmes objet et processus ; le profil SRM pour la modélisation de ressources logiciel et le profil HRM pour la modélisation des ressources matériel. Le paquetage d'analyse de MARTE introduit les éléments communs qui peuvent être utilisés pour contribuer à des nombreux types d'analyse quantitative. Trois types d'analyse sont pris en compte, La SAM pour l'analyse d'ordonnançabilité, la PAM pour l'analyse des performances et le WCETAM pour l'analyse de temps d'exécution au pire des cas (WCET). Le paquetage d'annexes inclut en particulier le sous-profil VSL (Value Specification Language) qui est un langage d'expression, utilisé pour spécifier les valeurs non-fonctionnelles. En fin, le sous-profil RSM pour modéliser les structures répétitives.

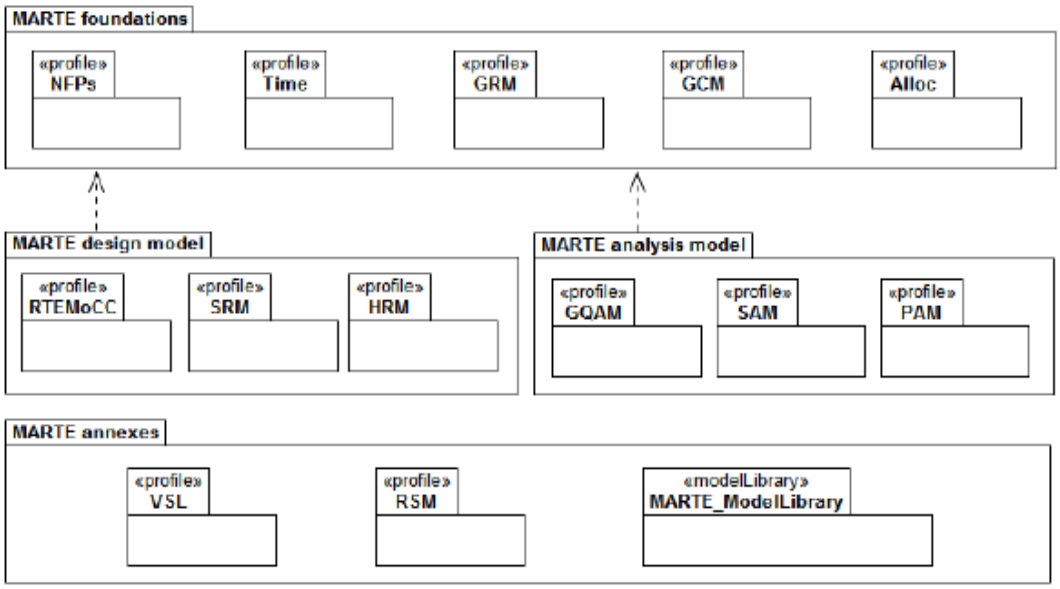


Figure 2.4. Architecture du profil MARTE [123].

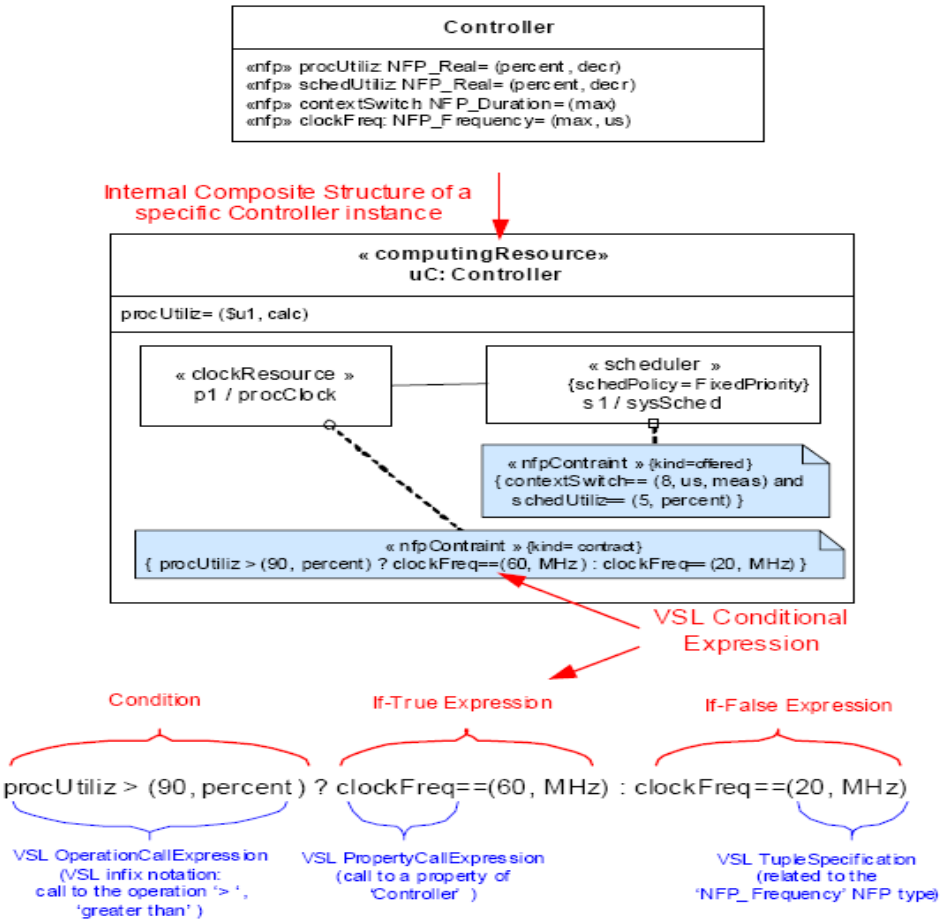


Figure 2.5. Exemple d'un model utilisateur avec NFPs et VSL [123].

MARTE apporte de nombreux avantages car il fournit un support pour la spécification, l'analyse, la conception, la vérification, et la validation, fournit un moyen de modélisation pour les aspects matériels et logiciels de systèmes embarqués temps réel en vue d'améliorer la communication entre les développeurs, et favorise la construction de modèles qui peuvent être utilisés pour faire des prédictions quantitatives.

2.3. UML-SOC

Il est développé par les laboratoires Fujitsu Limited et Fujitsu [122]. Une soumission OMG a été préparée par un consortium constitué de Fujitsu Limited, IBM Corporation, CANON INC, CATS Co., Ltd métaboliques, RICOH COMPANY LTD., et Toshiba Corporation. Ce profil vise à décrire les informations spécifiques aux SOCs en utilisant UML. Il intègre les concepts de SOC et permet la génération automatique de code pour le matériel (par exemple, SystemC), couvrant les niveaux d'abstraction à partir de la modélisation de niveau transactionnel (TLM) au niveau transfert de registre (RTL). UML-SOC est appuyé sur le diagramme de structure d'UML2.0. Il propose les stéréotypes qui permettent la modélisation structurelle, modélisation de la communication, et la modélisation des opérations et des propriétés. Le tableau 2.1 donne la correspondance entre certains stéréotypes de SOCs et les constructeurs UML. La motivation de tel profil est qu'UML définit plusieurs types de diagrammes, mais il ne décrit pas comment les utiliser. Les décisions concernant la partie de spécification à modéliser et les diagrammes à utiliser, ainsi que la façon de modéliser la spécification avec les différents diagrammes doivent être faites. Dans cette approche, UML est utilisé comme un modèle formel pour la spécification de SOCs afin de permettre la validation de la cohérence et l'exhaustivité de spécification (voir figure 2.6). La mise en œuvre conséquente de SOCs est validée par une dérivation automatique des scénarios de test à partir des modèles UML. UML est intégré dans le processus de la vérification, sans modification de style de conception actuel. Seuls les diagrammes des cas d'utilisation, les diagrammes de séquence et les diagrammes de classes sont utilisés pour modéliser les fonctions, les types de données et les comportements de la spécification. Les interfaces dans les SOCs ne peuvent pas être simplement modélisées par des opérations et des méthodes. En revanche, un langage propriétaire (CWL : Component Wrapper Language), en tant qu'un langage de spécification d'interface est utilisé pour modéliser la spécification des changements d'un signal au niveau des ports d'entrée / sortie [122].

SoC Model element	Stereotype	UML metaclass
Module	SoCModule	Class
Process	SoCProcess	Operation
Data	Data	Class
Controller	Controller	Class
Protocol Interface	SoCInterface	Interface
Channel	SoCChannel	Class
Protocol	SoCProtocol	Collaboration
Port	SoCPort	Port/Class
Module Part	SoCModuleProperty	Property
Channel Part	SoCChannelProperty	Property
Connector	SoCConnector	Connector

Tableau 2.1. Un sous-ensemble des stéréotypes de profil UML-SOC [122].

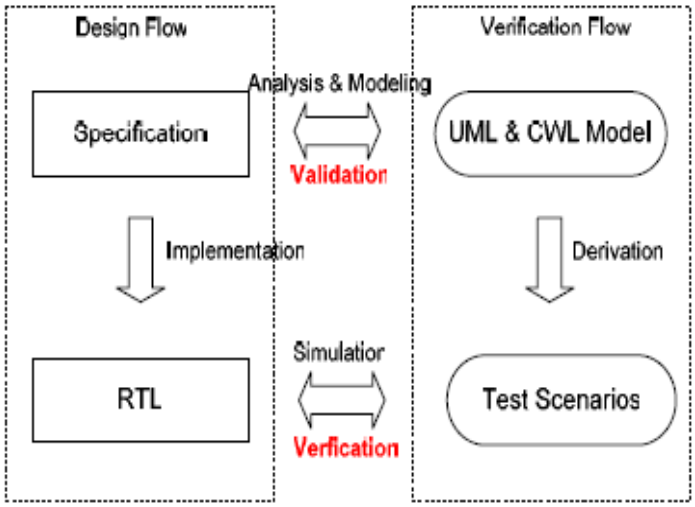


Figure 2.6. Le flot de profil UML- SOC [122].

2.4. UML-SystemC

Ce profil est élaboré par l'université de Catane et STMicroelectronics [109]. Il tire profit des avantages de deux standards : UML2.0 et SystemC en suivant les principes du MDA. SystemC est bien adapté pour l'implémentation des modèles UML, car il supporte le paradigme orienté-objet et peut présenter de manière uniforme le matériel et le logiciel dans un seul langage. En outre, SystemC est entrain de devenir le langage standard pour la

conception des SOCs au niveau système. Selon [109], UML peut améliorer le flot de conception de SOCs de trois façons:

1. UML peut être adopté au niveau système en tant qu'un langage fonctionnel exécutable pour décrire la spécification.
2. Le profil UML pour SystemC peut être utilisé pour la description du matériel aux niveaux abstraits au delà de niveau RTL.
3. Les profils UML adaptés pour les langages de programmation comme C / C + +, Java, etc., peuvent être utilisés pour la partie logiciel.

Le profil UML-SystemC capture à la fois les caractéristiques structurelles et comportementales du langage SystemC et permet la modélisation de haut niveau de SOCs avec une simple translation vers le code SystemC. Il est basé sur deux diagrammes: les diagrammes de classes et les statecharts. La figure 2.7 montre la correspondance entre les concepts de SystemC et UML. Le profil proposé est censé de favoriser la portabilité, et la réutilisation des IPs.

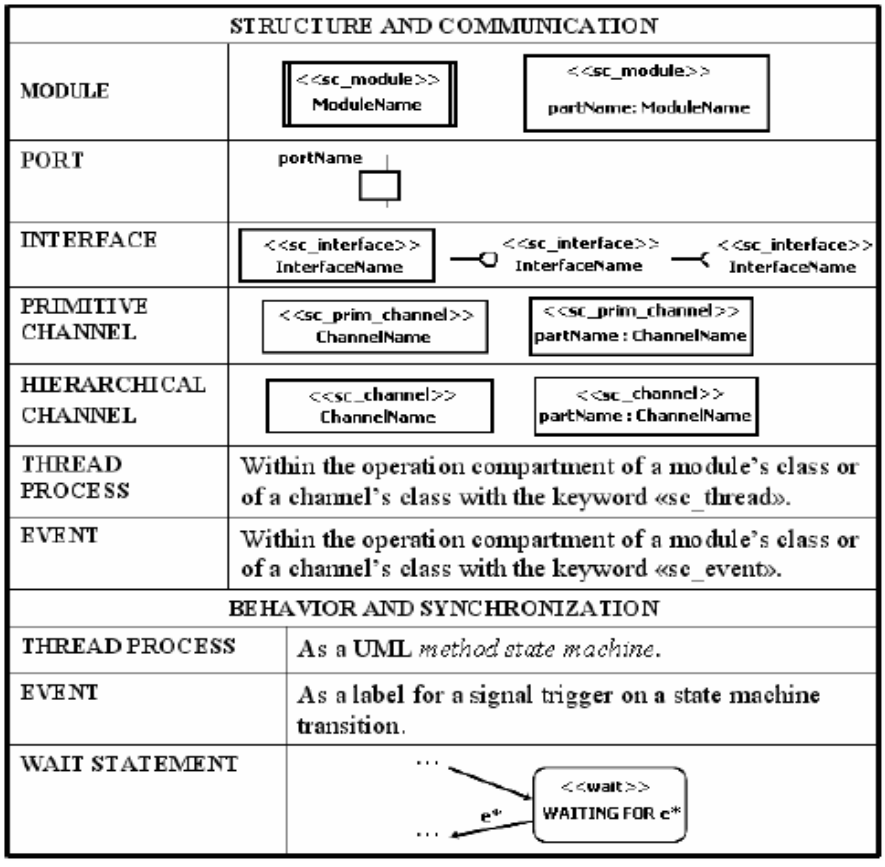


Figure 2.7. Notation UML pour les concepts SystemC [109].

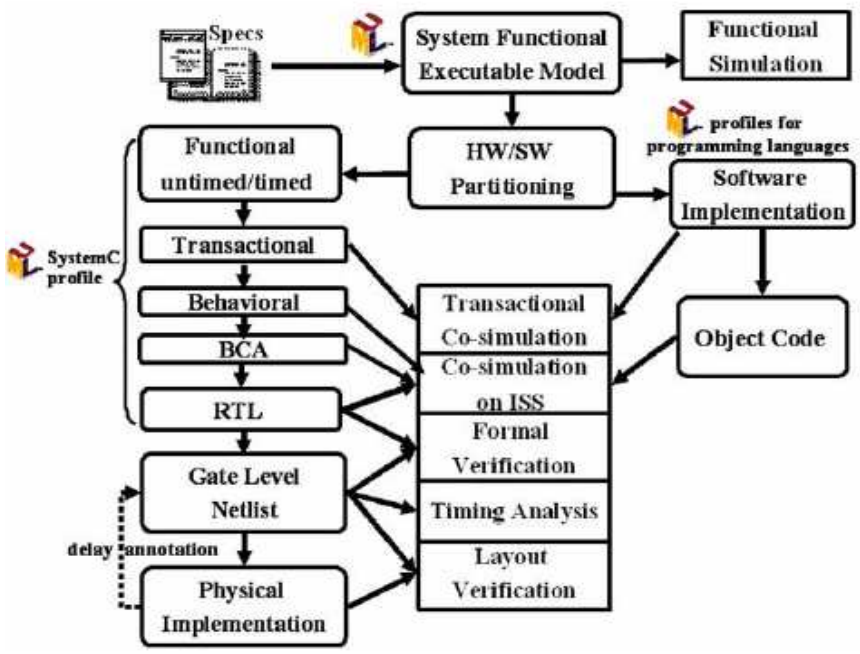


Figure 2.8. Flot UML –SystemC pour la conception de SOCs [109].

2.5. TUT

Développé à l'institut de conception de systèmes numériques à l'université de technologie de Tampere (TUT), TUT est un profil UML2.0 pour la conception de MPSOCs [82]. Il classe les différents composants de l'application et de la plateforme en définissant une diversité de stéréotypes et des règles strictes concernant la façon de les utiliser. L'objectif est de renforcer le support d'outils externes d'analyse automatique, de profilage, et de modification de la description UML2.0 d'un système embarqué. La classification attribue également des paramètres aux composants. En utilisant ce profil, l'application est modélisée comme un réseau de processus selon la sémantique de modèle de calcul KPN (Kahn Process Network). Chaque comportement de processus est modélisé par l'intermédiaire d'un Statechart. La description de la plateforme n'est plus utilisée pour la synthèse du matériel mais elle représente une abstraction d'une bibliothèque disponible de composants RTL paramétrées. L'association matériel/logiciel est défini par le stéréotype "PlatformMapping". Il est appliqué pour décrire la manière dont un groupe de processus est associé à un composant de la plateforme. La méthodologie est réalisée dans un flot de conception appelé Koski (figure 2.9). L'association de la plateforme peut être effectuée explicitement par le concepteur, ou avec l'aide d'outils. Dans ce dernier cas, un outil UML de profilage qui combine la description UML2.0 et les statistiques de simulation est développé. Sur la base des résultats de profilage,

l’application peut être modifiée afin de satisfaire les contraintes temps-réel. Lorsque la vérification se termine, un exécutable de l’application sur la plateforme est généré automatiquement à partir de la description UML2.0.

Le profil TUT fournit un chemin automatisé à partir d’UML vers un prototypage basé FPGA, y compris la vérification fonctionnelle et l'exploration automatique des architectures en s’appuyant sur le profilage et l’annotation en arrière (back annotation) automatiques des valeurs de performance.

Stereotype name (***extended Metaclass***)	**Description**
Application (*Class*)	Top-level application class
ApplicationComponent (*Class*)	Functional application component (active class, has behavior)
ApplicationProcess (*Structural feature*)	Instance of a functional application component
ProcessGroup (*Structural feature*)	Group of application processes
ProcessGrouping (*Dependency*)	Dependency between an application process and a process group
Platform (*Class*)	Top-level platform class
PlatformComponent (*Class*)	Defines features of a platform component
PlatformComponentInstance (*Structural feature*)	Instantiated platform component
CommunicationWrapper (*Dependency*)	Defines wrapper parameters of a communication agent
CommunicationSegment (*Structural feature*)	Interconnection structure of communicating agents
PlatformMapping (*Dependency*)	Dependency between a process group and a platform component instance

Tableau 2.2. Résumé de stéréotypes du profil TUT [82].

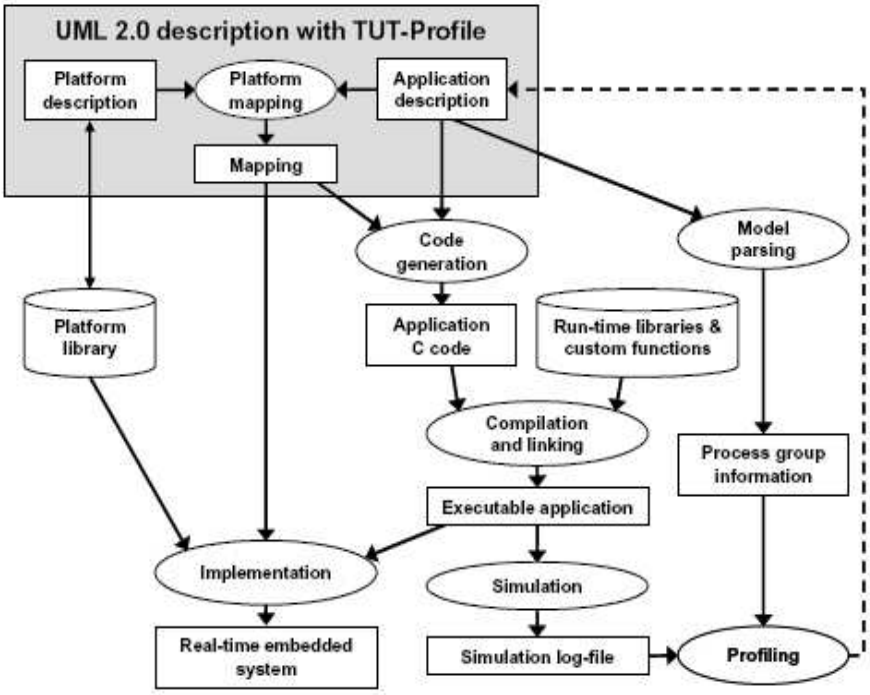


Figure 2.9. Le flot de conception Koski [82].

2.6. Gaspard2

Développé par le DART (Dataparallelism pour Real-Time) de l'équipe LIFL (Laboratoire d'Informatique Fondamentale de Lille-France), Gaspard2.0 est un profil UML2.0 qui vise le domaine de traitement de signal intensif [12]. Il applique les principes du MDA au niveau système et met l'accent sur la Co-modélisation et la concurrence, la séparation des préoccupations (communication vs calcul, données vs contrôle, application vs architecture), la simulation, le raffinement de modèles, la génération automatique du code (par exemple, SystemC, VHDL, JAVA) et l'intégration des IPs.

Le profil Gaspard2.0 étend la sémantique d'UML2 pour permettre à l'utilisateur de décrire un SOC à différents niveaux d'abstractions en trois étapes: l'application, l'architecture matérielle, et l'association de l'application à l'architecture matérielle. Le profil gaspard 2.0 comprend six paquetages majeurs (voir figure 2.10), qui sont le composant, la factorisation, l'architecture matérielle, l'application, le contrôle, et l'association. Dans Gaspard, l'application est modélisée au moyen de trois modèles de calcul qui sont: KPNs (Kahn Process Network) pour modéliser les tâches de calcul en utilisant les stéréotypes *GaspardComponent* et *GaspardPort* (paquetage de composants). Un composant Gaspard peut être primaire, hiérarchique ou répétitive ; Array-OL pour exprimer d'une manière compacte les topologies

de relations et de dépendances entre les tableaux multidimensionnels des éléments connectable (paquetage factorisation), et la programmation réactive synchrone (Esterel, Lustre) pour modéliser les aspects liés à la réactivité et au contrôle par des automates (paquetage de contrôle). Les principes de méta-modèle d'application sont basés sur le profil ISP UML qui permet l'expression de parallélisme de tâches et de données. L'architecture matérielle décrit les ressources matérielles et leurs topologies à un niveau fort de granularité. Le but de l'association est de fournir des outils qui lient une application à une architecture matérielle. Ils consistent principalement en l'association des tâches à des composants actifs et des données à la mémoire, en prenant en compte l'hiérarchie et les répétitions.

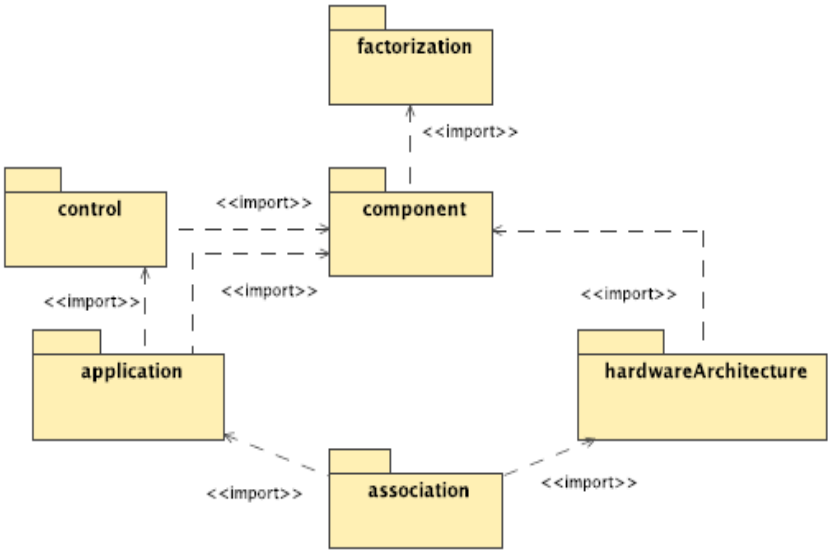


Figure 2.10. Paquetages du Gaspard [12].

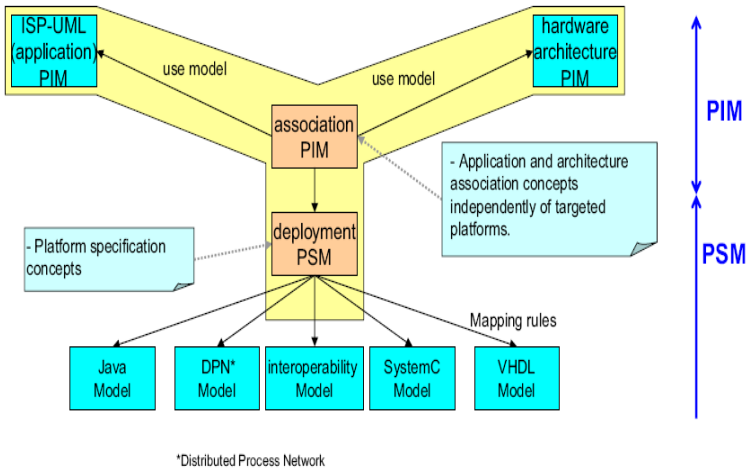


Figure 2.11. L'approche MDA/Y-Chart adoptée par Gaspard2 [12].

2.7. DIPLODOCUS (Design Space Exploration based on Formal Description techniques, UML and SystemC)

Est un profil UML2.0 s'appuyant sur le profil UML TURTLE qui cible le domaine des SOCs [5]. Il est axé sur quatre aspects:

1. Modélisation abstraite de l'application en utilisant deux types de diagrammes UML: les diagrammes de classes Diplodocus modélisant les tâches et les diagrammes d'activité pour modéliser les comportements internes de ces tâches. Les tâches communiquent au moyen de trois paradigmes: le canal, l'événement, et la requête. Une simulation ou une analyse statique peut être effectuée à partir de ces diagrammes.
2. Modélisation de l'architecture comme une composition des instances de cinq composants génériques: CPU, bus, mémoire, l'accélérateur du matériel et les périphériques d'entrée / sortie. Ces composants sont abstraits et paramétrés par le biais d'un ensemble de paramètres simples.
3. Association de chaque tâche d'exécution à un nœud d'architecture.
4. Raffinement de l'application vers une implémentation finale.

Le profil UML Diplodocus se concentre sur l'exploration de l'espace de conception. Sa force repose sur les règles de transformation qui permettent de transformer automatiquement les modèles Diplodocus soit en SystemC en vue de la simulation, ou une spécification LOTOS. Avant que la simulation soit effectuée, chaque comportement de tâche, qui est modélisé par le biais d'un diagramme d'activité se transforme à un comportement équivalent exprimé dans un langage simple appelé TML (Task Modeling Language) [4]. Ce langage abstrait l'échange de données, le traitement de données et l'échange de contrôle en utilisant des instructions de forte granularité. Il n'existe pas de détail de traitement de données à l'intérieur de tâches. Elles sont uniquement axées sur le contrôle, sans aucune notion de temps physique. Mais les opérations à l'intérieur de tâches, sont totalement ordonnées et parmi un ensemble de tâches, elles sont partiellement ordonnées. La simulation fonctionnelle est basée sur le code SystemC obtenu à partir des instructions TML.

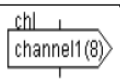
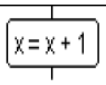
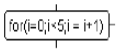
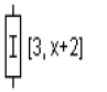
Icon	Description	SystemC	LOTOS
	Sending 8 samples in *channel1*	task.WR(8, channel1);	write_channel1!8
	Waiting for *evt1* with 2 params	task.WAIT(x, y, channel1);	wait_evt1?x?y
	Action state; here, increment x.	x = x + 1;	P[gates](.,x+1,.)
	Loop structure (loop on i)	for(i=0;i<5;i=i+1) { /* loop */} /* after loop*/;	process P[gates](.,0,.) $[i < 5] - >$ /* loop */ P[gates](.,i+1,.) $[not(i < 5)] - >$ /* after loop */
	Time interval between 3 and x+2	t = TML_tasks::myrand(3, x+2); task_T2.EXECI(t);	ignored (timing information are necessary for simulation only)

Figure 2.12. Les différentes Sémantiques des opérateurs des diagrammes D'activités de Diplodocus [5].

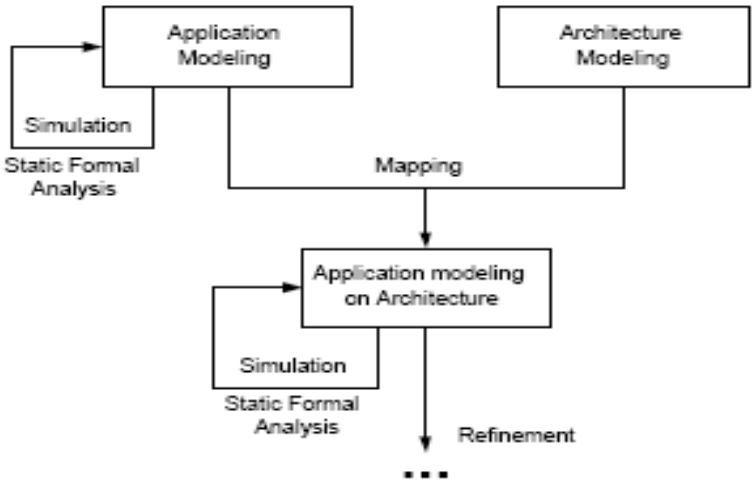


Figure 2.13. La Méthodologie adoptée par DIPLODOCUS [5].

2.8. UML-Platform

Développé par l'Université de Californie à Berkeley, le profil UML-Platform vise le domaine de protocoles sans fil [36]. Il est basé sur UML2.0, et UML RT (Real Time). Depuis que le profil UML-Platform suit les principes de la conception basée plateforme (platform-based design), il définit un ensemble de stéréotypes pour l'application, la plate-forme, l'association, et le raffinement. Dans ce profil, l'application est modélisée comme un réseau de processus à l'aide des modèles de calcul (MOC) standard, telles que KPN, Dataflow, etc. et

des briques (blocs) élémentaires, tels que les tampons et les protocoles qui peuvent être utilisés pour spécifier un modèle de calcul. Le comportement des différents composants est spécifié en utilisant la machine à états finis, les diagrammes d'activité, ou une notation textuelle. Le modèle de plateforme comprend de nombreux types de composants stéréotypés tels que les ressources physiques et logiques, les services offerts par les ressources, les contraintes sur les qualités de service (QOS), les relations entre les ressources, et les services utilisateurs. La sémantique de ce profil est définie en terme de méta-modèle Metropolis par l'établissement d'une correspondance directe entre les éléments de modélisation du profil et les éléments de méta-modèle Metropolis.

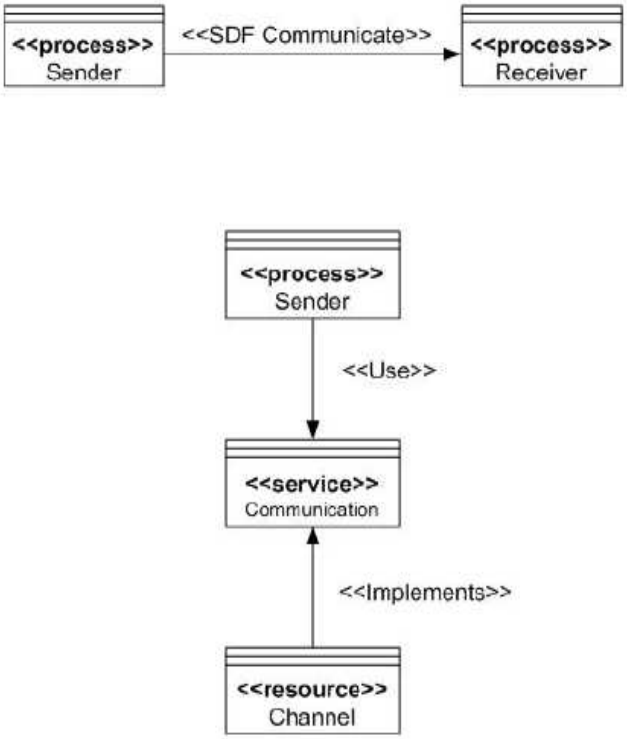


Figure 2.14. Relations entre Stéréotypes [36].

La méthodologie de conception adoptée par ce profil est montrée dans la Figure 2.15. Dans la première étape, le problème de conception est formulé en utilisant les diagrammes de cas d'utilisation et les contraintes annotées au modèle, puis, la fonctionnalité se capture et se décompose à l'aide de stéréotypes définis dans ce profil. Les contraintes sont propagées aux composants fonctionnels. Dans une prochaine étape, la spécification de profil est compilée en une spécification de Metropolis. Ensuite, le raffinement des communications et l'association auront lieu. L'analyse de performance et la validation prennent place dans l'environnement de simulation Metropolis.

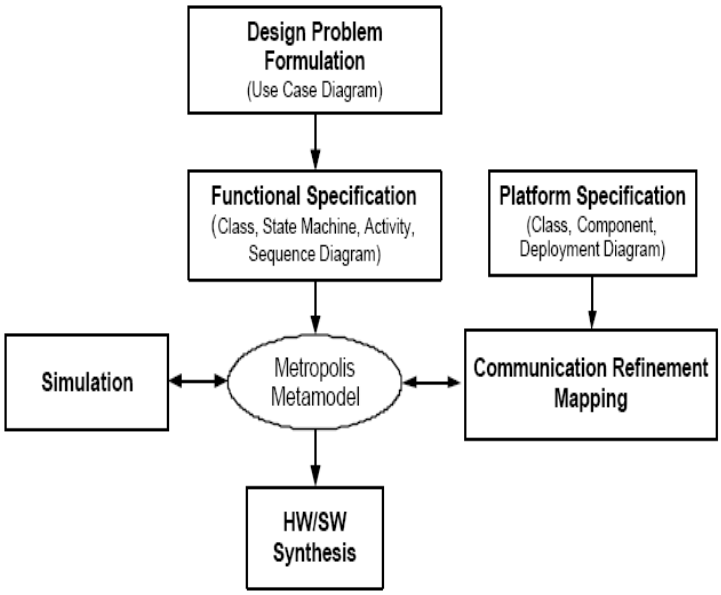


Figure 2.15. Flot de conception adopté par le profil UML-Platform [36].

3. Discussion

Dans ce paragraphe, nous essayons de mettre en évidence les limites de chaque profil [25]. Une comparaison entre les différents profils est montrée dans le tableau 2.3.

3.1. SysML

Vu les particularités des systèmes embarqués et des SOCs, il existe des fortes similitudes entre les méthodes utilisées dans le domaine de l'ingénierie des systèmes et la conception de SOCs complexes, telles que la gestion précise des exigences, la spécification de systèmes hétérogènes, la simulation, la vérification et la validation. L'une des principales contributions de SysML dans le domaine de SOC est le support pour modéliser les exigences. Les principales limites de SysML sont surtout au niveau du raffinement du système vers des implémentations logiciels/matériel. D'autre part, SysML ne résout pas le problème de l'absence de sémantique dans UML2.0 et ne dicte aucun processus de développement à être utilisé. Afin d'être en mesure d'intégrer les modèles d’exigences SysML dans les flots de conception des SOC, la formalisation de telles annotations informelles est indispensable.

3.2. MARTE

MARTE vise principalement les systèmes embarqués temps réel. Ce profil offre une facilité de modélisation et d'analyse des applications temps réel, toutefois, dans le contexte de

Co-conception, où les développements du matériel et de logiciel ont souvent lieu simultanément, le profil est moins utile: les problèmes liés au matériel, comme l'exploration de l'espace de conception, la synthèse, la génération d'interfaces matériel-logiciel ne sont pas suffisamment adressés. Il souffre également d'un manque de modélisation des exigences, de l'analyse et une discussion plus profonde à propos de l'abstraction et de la hiérarchie de l'application et de la plate-forme matérielle serait nécessaire.

3.3. UML-SOC

Le profil UML-SOC peut être considéré comme une extension de processus conventionnel de conception de SOC. Il ne traite que des aspects limités de développement de systèmes intégrés à savoir la formalisation des spécifications, et la dérivation de scénarios de test. Vu les aspects particuliers de systèmes embarqués complexes et les SOC, les principales limites de ce profil sont:

- Les aspects liés aux propriétés non fonctionnelles, ne sont pas abordés dans la spécification UML au niveau système.
- Le raffinement des interfaces est basé sur un langage propriétaire CWL entraînant un manque d'interopérabilité entre outils.
- La mise en œuvre est décrite au niveau RTL séparément, mais la vérification fonctionnelle utilise les mêmes scénarios de tests qu'au niveau UML.
- Certaines sémantiques des stéréotypes sont définies informellement (par exemple, le stéréotype protocole), d'autres nécessitent encore quelques précisions (sémantique de Synchronicité).

3.4. UML-SystemC

Ce profil cible des aspects liés au matériel tels que la modélisation niveau système, la synthèse, la simulation et la réutilisation des IP en exploitant les capacités des deux standard UML2.0 et SystemC. Toutefois, dans le cadre de Co-conception, il montre des limites, qui sont:

- Il ne traite, ni la capture des besoins, ni les propriétés non-fonctionnelles.
- Il ne prend pas en considération ni la partie logiciel, ni la génération des interfaces hardware / software.
- Le manque d'une sémantique claire conduisant à la traduction de la totalité du code SystemC à UML, donc plusieurs pages sont nécessaires pour capturer une simple fonction.

- Le profil est restrictif car il ne considère que les diagrammes d'états: La méthodologie basée sur le diagramme d'état est trop détaillée. Les diagrammes d'activité sont également importants pour la modélisation des systèmes dominés par les données.
- Le profil comprend des relations inhabituelles pour les outils UML, comme les associations entre les pseudo-états.

3.5. TUT

Le but principal de ce profil est l'automatisation de l'exploration d'architecture ciblant le prototypage à base de FPGA. Selon nos connaissances, le profil TUT est le premier profil montrant le profilage et les annotations en arrière automatiques de modèles UML. Les limites de ce profil sont:

- Il est restrictif, parce que, d'une part, il prend en compte un seul modèle de calcul (KPN), d'autre part, il modélise le comportement de chaque processus avec un Statechart: les diagrammes d'activité sont également importants.
- Manque de support formel pour la validation et de vérification.

3.6. GASPARD2

Ce profil vise le domaine de traitement de signal, il met l'accent sur les principes de MDA et l'approche en Y à différents niveaux d'abstractions. Le premier objectif de ce profil est la Co-modélisation de SOC en utilisant une variété de modèles de calcul, la Co-simulation, le raffinement des modèles et la génération automatique du code. Cependant, il manque un support pour la modélisation des propriétés non fonctionnelles, l'analyse formelle, la capture des besoins et la synthèse automatique des interfaces matériel / logiciel.

3.7. DIPLODOCUS

Ce profil est basé sur le profil TURTLE, dont le premier objectif est l'analyse formelle. Ce nouveau profil a tendance de renforcer le profil TURTLE afin de supporter la Co-conception et les aspects liés à l'exploration d'espace de conception, l'association et la Co-simulation. Les principales limitations sont:

- Depuis que la même spécification abstraite sert d'entrée à la fois à l'analyse formelle et à la simulation, il n'est pas clair si l'abstraction (à la fois des données et de comportements internes des tâches), qui est l'un des principes de base de Diplodocus conflits ou non avec la sémantique formelle de LOTOS.
- Le modèle architectural est fortement dépendant de la sémantique de TML.

- Il n'y a pas de support pour la hiérarchie.
- L'exploration de l'espace de conception ne concerne que l'architecture, mais pas l'application. Dans certains cas, nous devons, par exemple, diviser une tâche de calcul intensif en sous-tâches parallèles ou de fusionner deux tâches avec une forte charge de communication dans une seule tâche.
- La méthode est encore en cours d'expérimentation, et doit prouver son efficacité pour des architectures plus complexes et plus réalistes.

3.8. UML Platform

Ce profil rajoute une couche superficielle sur le méta-modèle Metropolis. Tous les aspects de Co-conception sont liés directement au Metropolis. Depuis que ce profil est fortement lié à l'approche de Metropolis, par conséquent, il manque l'interopérabilité avec les autres profils et outils.

En fin, nous pouvons constater que la Co-conception des MPSOC est un travail de recherche récent et qui est toujours en cours d'évolution. Dans ce domaine, plusieurs autres travaux existent déjà dans la littérature, tandis que d'autres débutent leur développement. Par exemple, P. Green et al. ont développé une méthode appelée HASOC (Hardware and Software Objects on Chip) pour le développement des systèmes sur puce [61]. C'est une méthode orientée objet basée sur une extension d'UML, et sur la décomposition du système en sous-systèmes matériel/logiciel pouvant être placés sur une plateforme réelle. Les auteurs ont également proposé la modélisation d'une plateforme basée sur l'utilisation d'UML et SystemC dont le but est d'enrichir UML pour mieux supporter la conception des systèmes embarqués jusqu'à la génération du code. Une autre plateforme de modèle d'exécution générique, appelée MEP (Model Execution Platform), est introduite pour la conception des systèmes matériel/logiciel [10]. Cette plateforme est également basée sur l'utilisation d'UML pour faciliter le passage de la spécification du système vers son implémentation en définissant une sémantique d'exécution bien précise. Plusieurs autres travaux ont été proposés dans le même domaine définissant ainsi un nouveau thème de recherche sur l'utilisation d'UML pour la co-conception des systèmes sur puce (UML for SOC Design) [18]. C'est un domaine qui bénéficie d'un grand intérêt académique et industriel, et regroupe plusieurs communautés scientifiques allant de la modélisation UML jusqu'au développement des circuits électroniques.

	SysML	UML SOC	UML SystemC	TUT	MARTE	GASPARD2	DIPLODOCUS	UML PLATFORM
NC	N	N	N	O	O	O	O	O
RC	O	N	N	O	N	N	O	N
PA	N	N	N	O	O	O	O	O
HS	N	O	O	O	N	O	O	O
HSI	N	N	N	O	N	N	N	Y
IPR	N	O	O	O	N	O	N	N
FA	N	N	N	N	N	N	O	O
PAR	COM	PROC	PROC	PROC	COM PROC	COM PROC SR Array-OL	PROC	PROC
TD	SYS	SOC	SOC	SOC	ERS	ISP SOC	SOC	WCP ES
AF	?	MDA	MDA	KOSKI	?	MDA Y-Chart	Y-Chart	PBD

NC: Capture de besoins non fonctionnels. **RC**: Capture de besoins fonctionnels. **PA**: Analyse de Performance. **HS**: Synthèse Matériel. **HSI**: Synthèse Interface Matériel/Logiciel. **IPR**: Réutilisation et Intégration des IPs. **FA**: Analyse Formelle. **PAR**: Paradigme. **TD**: Domaine cible. **AF**: Flot associé (Méthodologie). **COM**: Composant. **PROC**: Processus. **SR**: Reactive Synchrone. **ISP**: Traitement du signal intensif. **SOC**: Système mono puce. **ERS**: Systèmes embarqués Temps Réel. **ES** : Systèmes Embarqués. **WCP** : Protocoles de communication sans fil. **PBD**: Conception basée plateforme. **MDA**: Architecture dirigée par les modèles. **MDD**: Développement dirigé par les modèles.

Tableau 2.3. Profils UML pour SOCS et systèmes embarqués

4. Conclusion

Dans ce chapitre, nous avons présenté les profils UML les plus répandus dans le domaine de SOC et de systèmes embarqués. D'après l'analyse que nous avons fait, nous pouvons tirer les points suivants:

- Depuis, que la plupart des profils UML s'appuient sur le paradigme processus (tâche), il leur manque les capacités du paradigme orienté objet. Bien que le paradigme processus soit plus approprié pour la synthèse, le partitionnement matériel/logiciel et l'analyse de performance, il manque des possibilités de réutilisation et de l'abstraction. Selon nos connaissances, peu de travaux visent la synthèse de matériel à partir de spécifications purement orienté-objet. Les travaux de [52], ciblent la génération des architectures réconfigurables à partir de spécifications objets en exploitant les principes de paradigme objet comme le polymorphisme, l'encapsulation et l'héritage.
- La plupart des profils manquent de support formel pour l'analyse, le raffinement et la validation.

Deuxième partie : Contributions

Troisième chapitre

Profil UML pour estimer le temps d’exécution au pire de cas (WCET) d’une application sur MPSOC

Sommaire

1. Introduction

Notre objectif est de développer un profil UML pour la modélisation et l'estimation des performances d'une application multimédia (AMM) sur MPSOC en suivant les principes de la méthode en Y. Toutes les AMM sont centrées autour d'un flot de données de type image, sons ou vidéo. Elles fonctionnent sur une large (ou quasi-infini) séquence de données provenant de l'extérieur. Généralement les AMM se caractérisent par un flot de données stable, mais parfois une modification de structure de flot peut se produire en raison de certains événements externes.

La méthodologie que nous avons proposé porte sur deux niveaux d'abstraction [23,28]: le niveau système où l'application est décrite comme un réseau de composants de type boîte noire, et le niveau comportemental, où les comportements des composants sont définis.

En effet, notre méthode commence à un stade précoce de développement (la phase d'analyse) au cours duquel le concepteur modélise les besoins fonctionnels et non fonctionnels de l'application en utilisant les diagrammes de séquences d'UML annotés par des contraintes temporelles. A ce stade, l'application est modélisée comme une collaboration entre les objets.

Les comportements internes des objets ne sont pas encore connus, seulement les interactions séquentielles (éventuellement hiérarchiques) avec les informations de contrôle comme les conditions, les boucles et les temps d'exécution au pire de cas WCETs (Worst Case Execution Time) des méthodes d'objets. Par conséquent, la concurrence n'est pas modélisée et tous les messages sont considérés synchrones et exécutés de façon séquentielle. En d'autre terme, notre modèle initial est un paradigme objet purement séquentiel. Nous pensons que ce modèle offre de nombreux avantages. D'abord, parce que, un paradigme orienté objet est préférable en terme d'abstraction et de réutilisation. Deuxièmement, nous pensons que le modèle séquentiel facilite la tâche de modélisation relevant le concepteur de la charge de la modélisation de la concurrence. La deuxième étape de notre approche consiste à la transformation du modèle d'analyse (diagramme de séquence) au modèle de conception comprenant un ensemble de tâches communicantes par le bais des canaux virtuels. Ce nouveau modèle expose le parallélisme de tâches, le parallélisme de données, le pipelining, et la hiérarchie explicitement.

Le passage du modèle d'analyse au modèle de conception se fait via un ensemble des heuristiques (guidelines). Afin de supporter l'exploration rapide de l'espace de conception, le modèle d'architecture doit également être abstrait en fonction du niveau d'abstraction de l'application. Dans notre cas, l'architecture du SOC est un ensemble de composants abstraits

et génériques. Sans abstraire le système, il est très difficile d'effectuer une exploration rapide de l'espace de conception. Il peut être largement dit que l'architecture du SOC est composée de trois types de ressources: les Ressources de calcul (CR) comme les processeurs, FPGA, et IP ; Ressources de communication, tels que les bus et les ressources de mémorisation comme les RAMs. Chaque élément d'architecture est modélisé comme une abstraction de son modèle à granularité fine et ils sont génériques de sorte que cette architecture pourrait être utilisée pour modéliser tous les types des SOC.

Afin d'estimer les performances (temps, énergie, occupation mémoire, et le coût), l'application est associée au SOC. L'association est réalisée à travers une série des guidelines. Ces derniers ne garantissent pas la meilleure association, mais, ils aident le concepteur à trouver une bonne première solution. Selon les résultats d'estimation, le concepteur peut modifier la structure d'application ou de modifier les paramètres des composants matériels.

Dans l'étape suivante, le concepteur décrit les comportements internes des tâches en utilisant les diagrammes d'activité d'UML avec des actions à forte granularité. A partir de diagrammes d'activité d'UML, une spécification Maude est générée. Cette dernière est utilisée pour effectuer des estimations de temps et de consommation d'énergie plus précises et de vérifier formellement certaines propriétés telles que le deadlock (en raison d'une mauvaise association).

2. Modélisation d'application

2.1. Modélisation de calcul

2.1.1. Comportement Feuil

Un comportement feuil (CF) représente un calcul ordonnançable. Nous définissons un nouveau stéréotype nommé "behavior" avec les paramètres (tagged values) suivants:

- Le WCET relatif, exprimé en termes de nombre de cycles.
- Le Nombre maximum d'itérations.
- Le nombre maximum d'accès en lecture / écriture à une donnée partagée.

Les entrées et les sorties d'un CF sont jointes à ses ports en tan que paramètres. La taille de données transmises ou reçues est exprimée en termes de nombre de jetons. L'une de nouveautés de notre approche est l'abstraction de données. Donc, au lieu de spécifier la taille ou le type de données, il est préférable d'exprimer les données en termes des jetons abstraits. Un jeton abstrait est la donnée élémentaire communiquée ou traitée par les CFs. L'avantage

d'une telle abstraction est la grande opportunité pour appliquer l'analyse statique et des simulations rapides peuvent être facilement effectuées.

2.1.2. Comportements en séquence

Afin de modéliser les comportements qui s'exécutent en séquence, nous introduisons un nouveau stéréotype appelé "sequence". En figure 3.1, les comportements B1 et B2 s'exécutent en séquence.

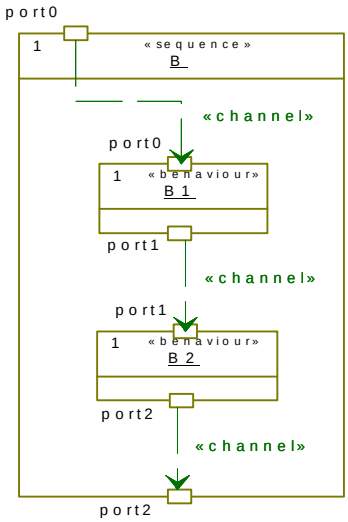


Figure 3.1. Comportements en séquence

2.1.3. Comportements en pipeline

Nous introduisons un nouveau stéréotype appelé "pipeline". Ce stéréotype compose les comportements dans l'ordre, avec la sortie de l'un relie l'entrée de la suivante. Le stéréotype «pipeline» contient deux paramètres qui sont le "PipeDepth", et le nombre maximum d'itérations. Le premier indique le nombre des étages de pipeline. Chaque comportement inclus dans le stéréotype "pipeline" est attribué à un processeur différent. L'exécution en pipeline implique une exécution itérative de fils. Dans l'exemple illustré à la figure 3.2, les comportements fils B1, B2 et B3 forment un pipeline à trois étages. Lorsque le pipeline est lancé, uniquement B1 est exécuté. Quand B1 se termine, la seconde itération commence et B1 et B2 s'exécutent en parallèle. Enfin, dans la troisième et les itérations suivantes, les trois comportements s'exécutent en parallèle. Le stéréotype " pipeline" prend également en charge la modélisation de mémoire tampon de communication entre les étapes de pipeline. L'exemple montré à la figure 3.2 comporte trois variables de communication entre les étapes du pipeline (V1, V2 et V3). Chaque variable est stéréotypé par un nouveau stéréotype appelé "pipe". Une variable stéréotypé par "pipe" peut être considérée comme une variable avec deux zones de

stockage. Un accès en écriture à une telle variable écrit toujours dans la première zone. Un accès en lecture, en revanche, lit toujours de la deuxième zone. En outre, le contenu de la première zone est déplacé vers la deuxième zone une fois que le pipeline commence une nouvelle itération. En d'autre terme, les données produites par B1 seront accessibles par B2 que dans la deuxième itération, et par B3 seulement dans la troisième itération.

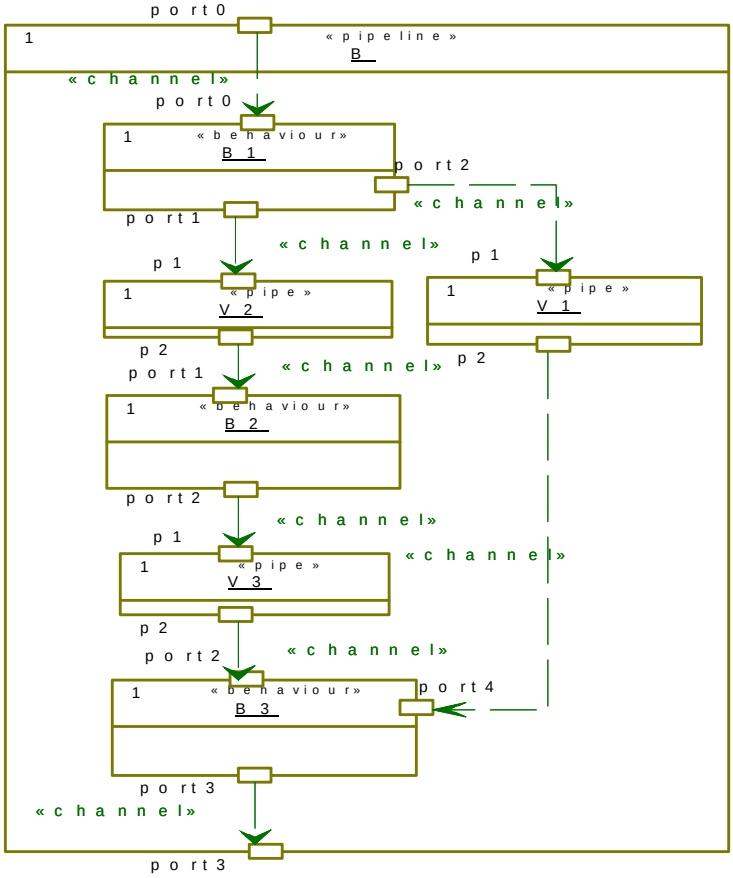


Figure 3.2. Comportements en Pipeline

2.1.4. Parallélisme de données

Pour répondre aux besoins des AMM, nous introduisons un nouveau stéréotype appelé "datapartition". Ce stéréotype distribue les données à un ensemble des flux en parallèle, qui sont ensuite réunis. Chaque flux de données est exécuté par le même code et nécessite un tampon mémoire. Ainsi, ce stéréotype duplique un comportement en un ensemble des comportements synchronisés. Le stéréotype "datapartition" contient trois paramètres: PartitionsNumber pour préciser le nombre de partitions de données, PartitioningMecanism pour spécifier comment les données sont dispersées et CollectionMechanism qui spécifie le mécanisme de collecte des données. Chaque partition de données est stéréotypée par

"partition" avec un seul paramètre qui spécifie la taille de la partition en termes de nombre de jetons: PartitionSize. Un comportement stéréotypé par "datapartition" joue le rôle d'un contrôleur (maître). Il est responsable de création, de synchronisation entre les comportements esclaves, de diffusion et de collecte de données. Le contrôleur exécute concurremment avec ses esclaves. La figure 3.3 montre un comportement stéréotypé par "datapartition". Dans cet exemple, nous supposons que le flux de données est réparti en deux partitions. Pour chaque partition, un comportement est créé. Par conséquent, nous avons trois comportements feuilles en parallèle: B (maître), B1 et B2 (esclaves) et deux partitions: data1 et data2.

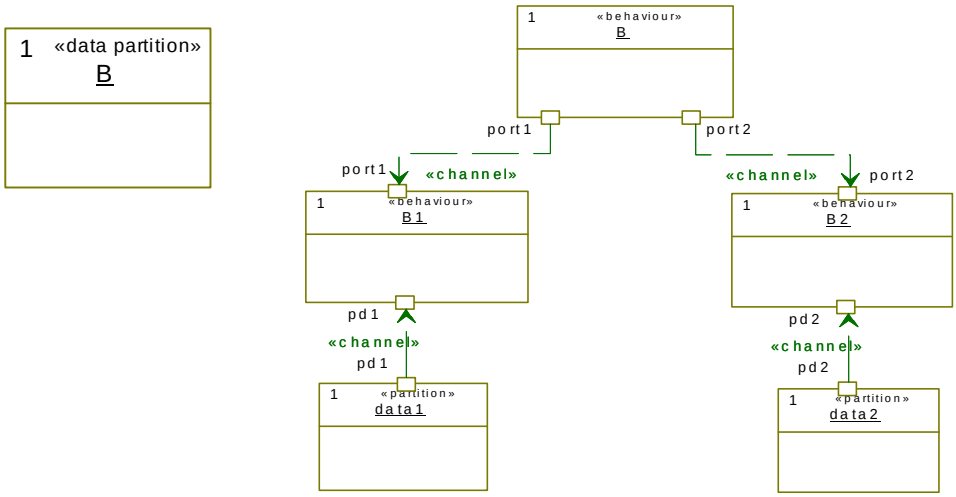


Figure 3.3. Comportement Partitionnement de données.

2.1.5. L'hiérarchie

Nous introduisons un nouveau stéréotype appelé "structure". Ce stéréotype définit un comportement hiérarchique. Nous utilisons ce stéréotype pour gérer la complexité et de permettre une description hiérarchique. La figure 3.4 montre un exemple d'un comportement hiérarchique appelé B. Celui-ci contient quatre comportements B1 (séquence), B2 (pipeline), B3 (datapartition) et B4 (comportement feuille). B1, B2, B3 et B4 s'exécutent concurremment.

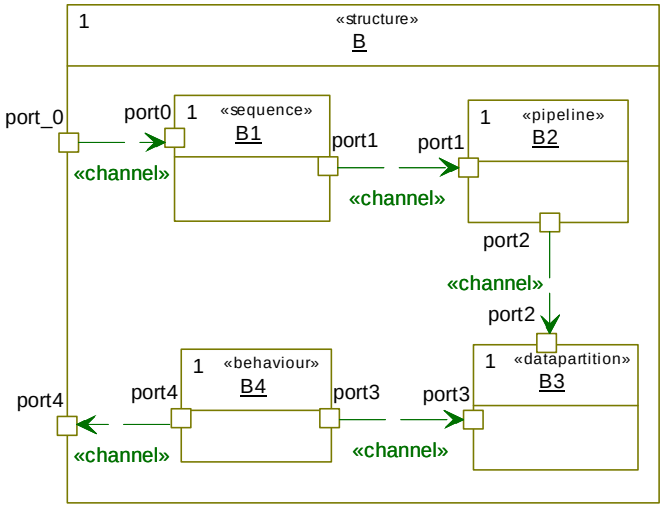


Figure 3.4. Comportement hiérarchique.

2.1.6. Comportements exclusifs

Dans certains cas, il est préférable d'exposer les comportements dont l'exécution dépend d'une certaine condition (un seul comportement s'exécute à la fois). Pour ce faire, nous définissons un nouveau stéréotype appelé "exclusive". Dans la figure 3.5, B est un comportement exclusif, il contrôle les deux comportements B1 et B2. Un comportement stéréotypé par «exclusif» joue le rôle d'un contrôleur. Selon la valeur de condition, il déclenche le comportement fils approprié. Par conséquent, le comportement du contrôleur est modélisé comme une machine à états finis.

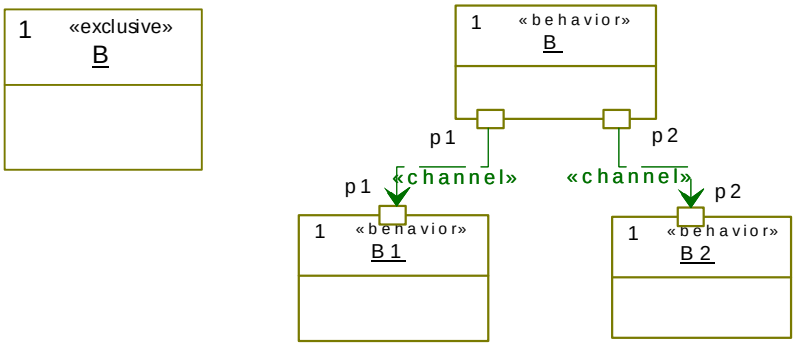


Figure 3.5. Comportement exclusif.

2.2. Modélisation de communication

Deux modèles de communication sont pris en compte, l'envoi de message, et la mémoire partagée.

2.2.1. Envoi de message

Dans ce cas, les comportements se communiquent via des canaux abstraits. Chaque canal est connecté à deux ports. Nous définissons un stéréotype appelé "channal". Ce stéréotype est appliqué au flot SysML. Il possède deux paramètres: *SampleMax* pour préciser la taille maximale de FIFO attachée au canal en terme de nombre de jetons et le style de communication qui peut-être lecture bloquante/écriture bloquante, (BRBW), lecture bloquante/écriture non bloquante (BRNW), ou lecture non bloquante/écriture non bloquante (NRNW).

2.2.2. Mémoire partagée

Dans ce cas, les comportements se communiquent via des données partagées. Ici, deux styles sont également possibles, synchrone et asynchrone. En mode synchrone, un verrou est associé à chaque bloc de mémoire partagée et un seul comportement peut accéder à la mémoire à un moment précis.

Le mode asynchrone n'a pas de verrouillage associé à la mémoire, et donc des multiples accès peuvent se produire. Nous définissons un stéréotype appelé "shared data" avec deux paramètres: SizeData spécifiant la taille de données partagées et le mode de communication qui peut être synchrone ou asynchrone.

3. Modélisation d'architecture et d'association

3.1. Modélisation d'architecture

3.1.1. Le modèle CPU

Ce modèle concerne les processeurs à usage général (GPP) et les processeurs à application spécifique ASIP (par exemple, DSP). CPU peut exécuter un ou plusieurs comportements. Dans ce dernier cas, il a besoin d'un ordonnanceur. CPU est paramétré par cinq paramètres qui sont le coût, le facteur de vitesse (SF), consommation d'énergie dynamique (DP) par cycle (mode de fonctionnement), consommation d'énergie statique (SP) par cycle (mode inactif), et la taille de la mémoire de données. Le facteur de vitesse est un nombre indiquant la vitesse

relative de CPU. Pour un GPP, SF = 1. Pour une ressource de calcul plus rapide que GPP (par exemple, FPGA, IP), SF <1.

3.1.2. Le modèle IP

Ce modèle concerne les blocs matériels pré-caractérisés. IP est paramétré par son coût, SF, et DP.

3.1.3. Le modèle FPGA

Il est paramétré par son coût, SF, DP, DS, la taille de la mémoire locale, et le temps de reconfiguration (RT) en termes de nombre de cycles.

3.1.4. Le modèle bus

A ce niveau d'abstraction, nous supposons que le bus communique directement avec les autres composants sans utiliser les interfaces. Le bus est paramétré par quatre paramètres qui sont: le coût, le taux de transfert (TRB) en terme de nombre de jetons transférés par cycle, DP, SP, et le type de bus (partagé ou dédié). Dans le cas de bus partagé, il faut préciser le mécanisme d'arbitrage pour résoudre le problème des transferts concurrents. Si deux composants matériels ont besoin d'un lien rapide entre eux, le concepteur peut configurer ce lien comme étant un bus dédié.

3.1.5. Le modèle mémoire

Il est caractérisé par son coût, le taux de transfert (TRR) en termes de nombre de jetons lus par cycle, taux de transfert (TRW), en termes de nombre de jetons écrit par cycle, le nombre de lectures simultanées (SRN), le nombre d’écritures simultanés (SWN), DP et SP.

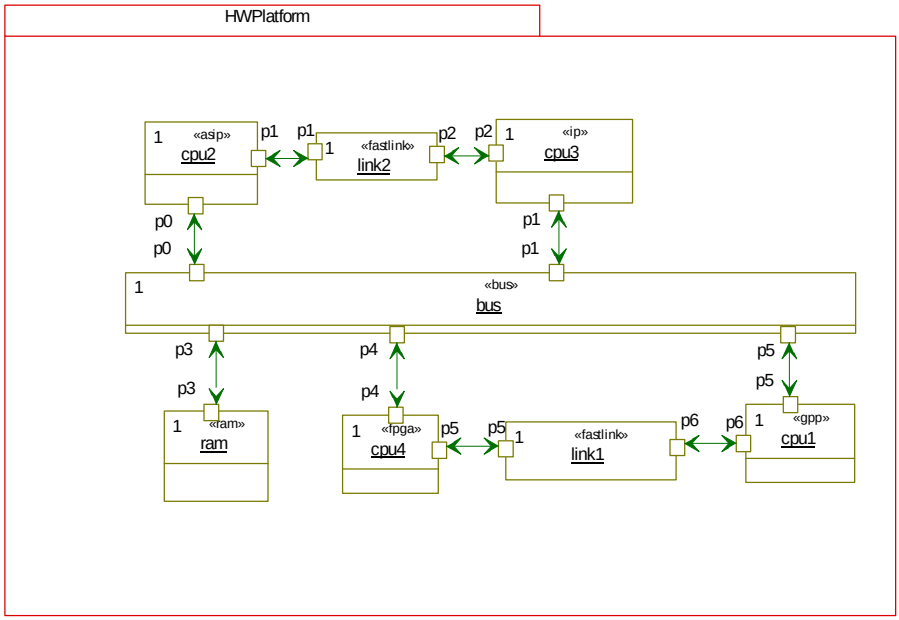


Figure 3.6. Architecture matérielle abstraite.

3.2. Modélisation d’association

L’association (Mapping) consiste en l'assignation et l’ordonnancement des composants de l'application aux composants de l'architecture de sorte que les comportements sont associés aux ressources de calcul (CPU, IP, FPGA), les canaux de communication aux bus, et les données aux mémoires. Nous définissons un nouveau stéréotype appelé *"AllocatedTo"*. Ce stéréotype est appliqué sur la contrainte UML; il a deux paramètres : le premier spécifie la ressource matérielle à laquelle la composante logique sera assignée et le second désigne un nombre qui détermine l'ordre d’ordonnancement du comportement (le transfert). L’allocation ne concerne que les comportements feuils. La figure 3.7 montre un exemple d’association en utilisant la contrainte d’UML. Dans cet exemple, le comportement B est affecté à une ressource physique appelé FPGA existant dans le paquetage HWPlatform. On note que les données peuvent être attribuées soit à la mémoire partagée soit aux mémoires locales de CPU et FPGA.

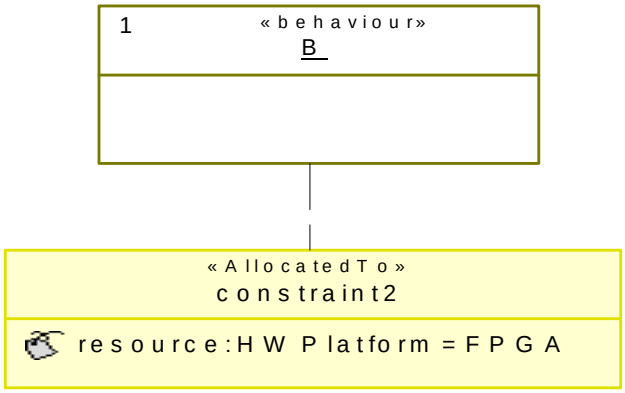


Figure 3.7. Modélisation d'association.

4. Guidelines pour l'extraction du parallélisme à partir de diagramme de séquence

Dans cette section, nous présentons notre technique d'extraction de la concurrence, de pipeline, la hiérarchie et les comportements mutuellement exclusifs à partir de diagrammes de séquence d'UML2.0.

Le point de départ de notre flot est l'établissement d'un modèle séquentiel de l'application. Pour ce faire, nous utilisons le diagramme de séquence (SD). Ce dernier est un bon choix pour modéliser les interactions séquentielles (éventuellement hiérarchique) entre les objets. En outre, il expose les données de contrôle et de dépendances, les boucles, et les conditions explicitement. Nous enrichissons le SD avec les contraintes temporelles (WCETs). La figure 3.8 montre un exemple d'un diagramme de séquence hiérarchique. Il existe cinq objets nommés MAIN, O1, O2, O3 et O4. Ces objets interagissent via l'envoi de message. Tous les objets sont considérés passifs et tous les messages sont synchrones. *La première étape ; identification de comportements concurrents, en séquence, en pipeline, et exclusifs :*

Les scénarios bout en bout peuvent être des comportements concurrents. Un scénario bout en bout est une séquence de méthodes déclenchée par un appel à une méthode ou un événement externe. Dans l'exemple de la figure 3.8, nous pouvons identifier quatre comportements concurrents nommés B1, B2, B3 et B4. B1 = (M1, M2, Get_attrib, M3, M4, Set_attrib), B2 = (M21, M4, Set_attrib, M31), B3 = (M11, M22, M23, M24), B4 = (SD1). La communication entre ces comportements est à la base de mémoire partagée. Dans cet exemple, B1 et B2 accèdent à l'attribut "attribut" de l'objet O3 via les méthodes Get_attrib

(lire), et Set_attrib (écrire). Donc, nous pouvons créer un nouvel objet de données appelé "attrib" stéréotypé par "shared data".

Nous pouvons toujours transformer le schéma de communication à mémoire partagée à un schéma de communication à envoi de message équivalent. Dans ce cas, nous ne créons pas les objets de données "shared data", plutôt que nous considérons les données partagées comme des données d'entrées / sorties de comportements. Tous les accès aux données partagées sont remplacés par des communications point-à-point à travers les canaux.

Une autre forme de parallélisme que l'on peut extraire de SD est le pipeline. Dans la figure 3.8, les méthodes M22, M23 et M24 de comportement B3 peut être exécuté dans un pipeline. Dans le même exemple, on remarque que la valeur retournée de B2 (e) sert d'entrée pour B3, alors B2 et B3 sont en séquence. Étant donné que les messages sont synchrones, l'appelant doit attendre (bloqué) les valeurs renvoyées. Pas toutes les méthodes recevaient ou retournent des données. Dans ce cas, nous introduisons deux événements de contrôle à zéro délai: *Request* et *Ret*.

A ce stade, on peut également extraire les comportements qui s'excluent mutuellement. Par exemple, B1 et B2 appartiennent à des différentes branches de l'opérateur "alt". Donc nous pouvons les mettre dans un seul comportement stéréotypé par "exclusive". Enfin B4 est un comportement hiérarchique. La figure 3.8 montre le résultat de l'application des guidelines où "main" est le contrôleur et il exécute concurremment avec les autres comportements. En utilisant le diagramme de séquence, on peut extraire pour chaque comportement, un ensemble de paramètres temporels. Par exemple pour B1 : WCET = 40 + 10 + (5 + 20) * 40 = 1050 cycles. Ici, la boucle <1,40> spécifie le nombre maximum d'itérations. Pour B1 le nombre d'itérations = 1, le nombre Maximum des accès en lecture à la donnée partagée = 1 (Get_attrib) ; le nombre Maximum des accès en écriture = 40. La figure 3.9 montre le résultat de l'application des guidelines sur l'exemple de la figure 3.8.

La deuxième étape ; identification des comportements "partition de données": Ce type de parallélisme ne peut pas être identifié à partir de diagramme de séquence. Il requiert des connaissances sur les structures de données internes des méthodes et de la façon les méthodes manipulent ces structures. En règle générale, les méthodes ayant un grand temps d'exécution sont des bonnes candidates pour les scinder en comportements concurrents moins intensifs.

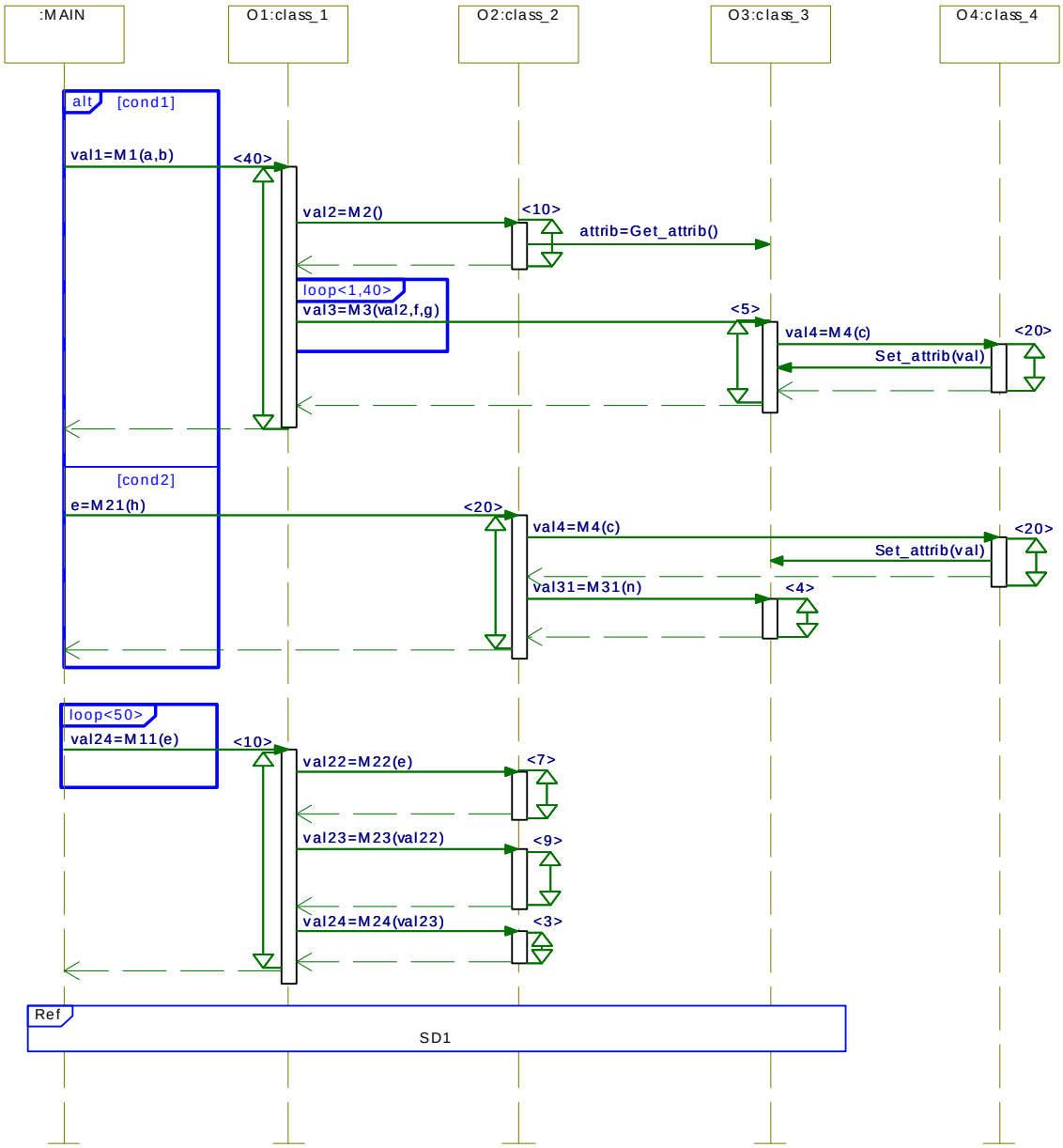


Figure 3.8. Diagramme de séquence hiérarchique.

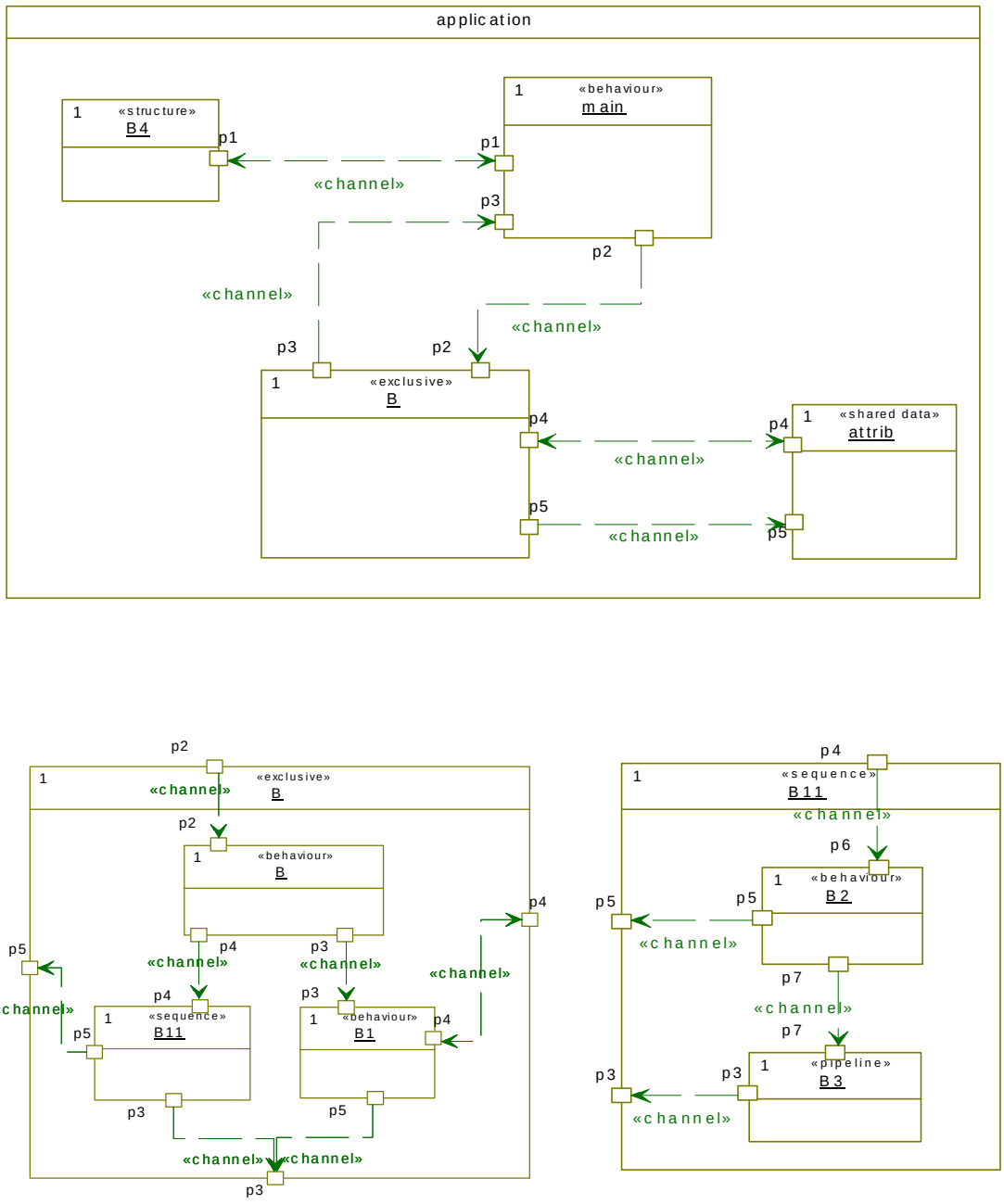


Figure 3.9. Modèle concurrent hiérarchique.

5. Guidelines d’association et d’optimisation

5.1. Guidelines d’association

Comme mentionné ci-dessus, ces guidelines ne garantissent pas l’association optimale, plutôt qu’ils permettent au concepteur de trouver une bonne première solution. Ces directives traitent deux objectifs contradictoires: la minimisation de temps de communication et de temps d'exécution des comportements.

1. En général, les comportements avec calcul intensif, sont mis en oeuvre dans le matériel (comme IP ou FPGA). Mais, en raison du manque de ressources matérielles, des ressources doivent être partagées. Pour surmonter ce problème, nous pouvons associer les comportements séquentiels et mutuellement exclusifs à la même ressource.
2. Si un comportement est dominé par le contrôle, il ne sera pas assigné à un FPGA, plutôt qu’il est préférable de l’assigner à un GPP.
3. Si un comportement est dominé par des données avec une très forte charge de calcul, alors il ne sera pas assigné à un GPP, à moins qu'il n'y ait pas de ressources matérielles disponibles.
4. Si le premier objectif est la minimisation de temps de communication, alors assigner les comportements ayant une forte charge de communication à la même ressource de calcul, même s'ils sont concurrents.
5. Pour réduire la congestion de bus, assigner les canaux à forte circulation aux bus dédies.

5.2. Guidelines d’optimisation

En effet, il existe de nombreux facteurs qui influent sur les performances globales du système. Les plus importants sont:

1. Granularité de comportements.
2. Granularité à laquelle les données sont communiquées.
3. La quantité de données traitées par chaque comportement dans le cas de partition de données.
4. La gestion de données partagées et le schéma de communication.
5. Les mécanismes de diffusion et de collecte de données.
6. La profondeur de pipelines.

Ces paramètres doivent être choisis soigneusement. Le temps supplémentaire introduit par la synchronisation peut contrecarrer les avantages de parallélisme : comportements à forte granularité (ce qui signifie petit nombre de comportements) sont meilleurs que les comportements à granularité fine en terme de temps de communication. Toutefois, ces comportements peuvent attendre plus longtemps. Enfin, en fonction de l'application, le partitionnement des données peut entraîner un temps de communication supplémentaire pour les dépendances des données entre les partitions. Selon les résultats d'estimation, le concepteur peut:

1. Fusionner les comportements avec une grande charge de communication en un seul comportement.
2. Dans la figure 3.8, on remarque que la méthode M4 est dupliquée en B1 et B2. Depuis que M4 ait un grand temps d'exécution, il serait préférable de la mettre à part.
3. Pour chaque comportement "goulet d'étranglement" raffiner le en suivant les guidelines de la section 4.
4. Afin de minimiser le temps de synchronisation qui due au partitionnement de données, il est préférable que chaque partition de données traite des blocs de données plus larges [1].
5. Le temps supplémentaire de la reconfiguration des FPGA peut être minimisé si nous mettons les comportements en séquence dans la même séquence de bits (bitstream).

6. Estimation de performances

Nous présentons notre technique d'estimation concernant le temps d'exécution, la consommation d'énergie, l'occupation mémoire, et le coût.

En ce qui concerne l'estimation des performances, la plupart des travaux s'appuient sur le profilage d'application où le temps et l'énergie sont calculés sur la base du code exécutable. Comme cette technique nécessite le code complet, elle ne peut donc pas être appliquée aux premiers stades de développement. Cependant, une bonne estimation peut être obtenue avant le codage, même dès les premiers stades de conception, basée sur l'expérience antérieure et les modèles existants. Donc, notre modèle d'estimation est basé sur une formule analytique. Puisque nous considérons un niveau d'abstraction élevé, l'analyse analytique semble plus appropriée. Bien sûr, notre formule est estimative, mais elle sert une bonne première tentative pour modéliser les aspects liés au temps et à la consommation d'énergie à un niveau d'abstraction élevé.

6.1. Estimation de temps d'exécution

Nous nous somme intéressés par l'estimation de WCET, qui est exprimée en termes de nombre de cycles. Mais avant de présenter notre technique d'estimation, nous faisons trois hypothèses:

1. La Communication via les données partagées est asynchrone.
2. Le mode de communication sur les canaux est supposé NRNW (FIFO de taille infinie).
3. Nombre de lectures (écritures) simultanées de la mémoire ne dépassent jamais SRN (SWN).

À ce niveau d'abstraction, le temps de blocage qui est du au partage de ressources, ne peut être déterminé analytiquement (à moins que, le concepteur a une estimation exacte selon son expérience). Pour cette raison nous prenons uniquement en compte le cumul de temps résultant de séquencement des comportements et des transferts. Ces hypothèses facilitent la tâche de l'estimation.

Soit t le WCET de comportement B et *SF* le facteur de vitesse de ressource de calcul CR. Si B est affecté au CPU, alors le temps estimée Et pour B est :

*Et = t * SF*.

Si B est associé à un FPGA, on inclut le temps de reconfiguration:

Et = Et + Treconfig.

Si il y a d'autres comportements concurrents qui sont affectés au même CPU, alors

Et = Et + Tcpu.

Où *Tcpu* est le temps supplémentaire qui du au séquençement des comportements. B accède des données partagées, alors

Et = Et + Tdata.

Où *Tdata* est le temps supplémentaire qui du à l'accès aux données partagées (dans ce cas *Tdata* est introduite par le concepteur, sinon il est égal à zéro, conformément à la deuxième hypothèse).

Tread = Tdata + Twrite.

Si B est le seul comportement qui accède aux données partagées alors

*Tread = NBread * (Ttrans + Trmem).*

Où *Ttrans* est le temps du transfert sur le bus, *Trmem* est le temps de lecture effectif et *NBread* est le nombre des accès en lecture. (Nous supposons que l'accès aux données partagées se fait via un bus).

Ttrans = DataSize / TRB,

Trmem = DataSize / TRR,

*Twrite = NBwrite * (Ttrans + Twmem).*

Où *Twmem* est le temps de l'écriture effectif, et *NBwrite* est le nombre des accès en écriture. *Twmem = DataSize / TRW.*

S'il y a plusieurs comportements qui accèdent aux mêmes données partagées simultanément, alors :

Ttrans = Ttrans + Tbus.

Où *Tbus* est le temps supplémentaire qui est du au séquencement des transferts. En outre, si B s'exécute itérativement, alors :

*TEt = Et * Iter.*

Où *Iter* est le nombre maximum d'itérations et *TEt* est le temps d'exécution total de B.

Soient t1, t2 les WCETs de comportements B1, B2, respectivement.

- **B1 et B2 sont en séquence**

  Si B1 et B2 sont associés à la même CR, alors :

T = TEt1 + TEt2).

Si *CR = FPGA, Treconfig* est ajouté une seule fois.

SI B1 et B2 sont associés à deux CRs distincts liés par un bus dédié, alors

T = TEt1 + TEt2 + TCOM,

Avec *TCOM = Iter * DataSize / TRB* ;

Le temps de communication entre B1 et B2, où *DataSize* est la taille de données transférées entre B1 et B2, et *Iter* est le nombre d'exécutions de B1. Si le lien entre les CRs est un bus partagé alors :

TCOM = TCOM + Tbus.

- **B1 et B2 sont mutuellement exclusifs**

  T = MAX (TEt1, TEt2).

- **B1 et B2 sont concurrents**

  Si B1 et B2 sont associés à deux CRs distincts, alors :

T = MAX (TEt1, TEt2).

Si B1 et B2 sont associés à la même CR, alors on calcule T comme si B1 et B2 sont en séquence.

- **B1 et B2 sont en pipeline**

Soit n le nombre d'itérations de pipeline. B1 est associé à CR1 avec SF1 et B2 est associé à CR2. Seul le comportement B1 sera exécuté à la première itération. Dans la deuxième itération, B1 et B2 seront exécutés simultanément. Dans la troisième et toutes les itérations suivantes, les deux comportements sont exécutés en parallèle. Après l'itération n, seulement B2 sera exécuté (dans l'itération n +1).

*T = TEt1 + (n-1) * MAX (TEt1, TEt2) + TEt2.*

Plus généralement, si la profondeur du pipeline est m et le nombre d'itérations est n, on peut estimer T par la formule:

*T = (nm-1) * MAX (TEt1, TEtm ...) + + TEt1 + MAX (TEt1, TEt2) + ... + MAX (TEt1, TEt2, ... TEtm-1) + + TEtm + MAX (TEtm, TEtm-1) + MAX (TEtm, TEtm-1, TEtm-2) + ... + MAX (TEtm, ..., TEt2).*

- **B est une partition de données**

Soit n le nombre de partitions de données et soient S1, S2, ..., Sn leurs tailles respectivement. Un comportement "datapartition" B joue le rôle d'un contrôleur. Il permet de créer n comportements B1, B2, ... Bn s'exécutant simultanément. B diffuse et collecte aussi les données. Supposant que le temps d'exécution est proportionnel à la quantité de données traitées, pour chaque comportement Bi, on peut estimer son WCET par la formule :

*ti = t * Si / S.*

Où t est le WCET de B, Si est la taille de partition de données de comportement Bi, et S est la taille de la somme de toutes les partitions de données. Lorsque Bi est alloué au CRi, alors :

*Eti = SFi * t * Si / S.*

Avant que les esclaves commencent leurs exécutions, le contrôleur doit transmettre les données des partitions à ses esclaves. On peut estimer le temps de transmission par la formule *Tpart = MAX(S1/Ttrans, S2/Ttrans, ..., Sn / Ttrans).*

Le contrôleur lui-même s'exécute concurremment avec ses esclaves et prend de temps pour la diffusion et la collecte de données.

T = Tpart + MAX (TEt1 + Tcol1, TEt2 + Tcol2 ..., + TEtn + Tcoln).

Où *Tcoli* est le temps supplémentaire qui du à la collecte des données.

*Tcoli = Iter * DataSizei / Ttrans.*

Où *Iter* est le nombre d'exécutions de Bi, et *DataSizei* est la taille de données de sortie de Bi. Des temps supplémentaires peuvent être rajoutés, si le concepteur a des informations précises sur la fonctionnalité du contrôleur.

6.2. Estimation de consommation

En général, la consommation est proportionnelle au temps d'exécution. La connaissance de la quantité d'énergie dynamique consommée par cycle par un processeur conduit à l'estimation de la consommation d'énergie pour un comportement feuil B par la formule:

$P = DP*t * Iter * SF$.

Pour chaque transfert de données,

$P = DP * DataSize / TRB$.

Pour chaque lecture mémoire

$P = DP * DataSize / TRR$ et

$P = DP * DataSize / TRW$ pour chaque écriture mémoire.

L'énergie totale est la somme des énergies consommées par tous les processeurs, IP, RAMs, et les bus. Nous introduisons *SP* qui spécifie la quantité d'énergie statique consommée (par cycle), par le CPU quand il est en état de repos. Cela conduit à une estimation plus précise. Supposons que C est un comportement associé au même processeur que B et C s'exécute le premier en t1 cycles avant B. On estime la consommation d'énergie de B par la formule:

$P = t1 * SP + P$ [53].

6.3. Estimation d'occupation mémoire

Dans l'ensemble, l'occupation de mémoire externe est donnée par la formule:

$SM = SPP + SSD + SDP$.

Où *SPP*, *SDP* et *SSD* sont les tailles des variables en pipeline, des partitions de données, et des variables partagées respectivement.

6.4. Estimation de coût

Le coût global de la plate-forme matérielle est donné par la formule:

$C = CostCR + CostBus + CostMem$.

Où *CostCR*, *CostBus* et *CostMem* sont les coûts de ressources de calcul (CPU, IP, FPGA), des bus, et des mémoires respectivement.

7. Implémentation et étude de cas

Nous avons mis en place un outil basé Rhapsody pour la modélisation et l'estimation du temps d'exécution au pire de cas (WCET) d'application sur SOC. Pour accomplir notre objectif, nous avons utilisé l'interpréteur VB intégré dans Rhapsody 7.2 [108]. Avec l'API

VB du Rhapsody, nous pouvons traiter les diagrammes UML et programmer dans l'environnement Rhapsody.

L'estimation se fait en suivant une approche ascendante. Nous ajoutons un nouveau stéréotype appelé "application" pour modéliser l'objet racine. L'architecture matérielle est modélisée comme un paquetage. Il comprend l'ensemble des composants matériels stéréotypés. La Figure 3.10 montre une capture d'écran d'outil que nous avons développé.

Nous avons testé notre méthode sur deux exemples : le décodeur H264 de compression vidéo et le décodeur MP3. Il s'agit de deux applications multimédia.

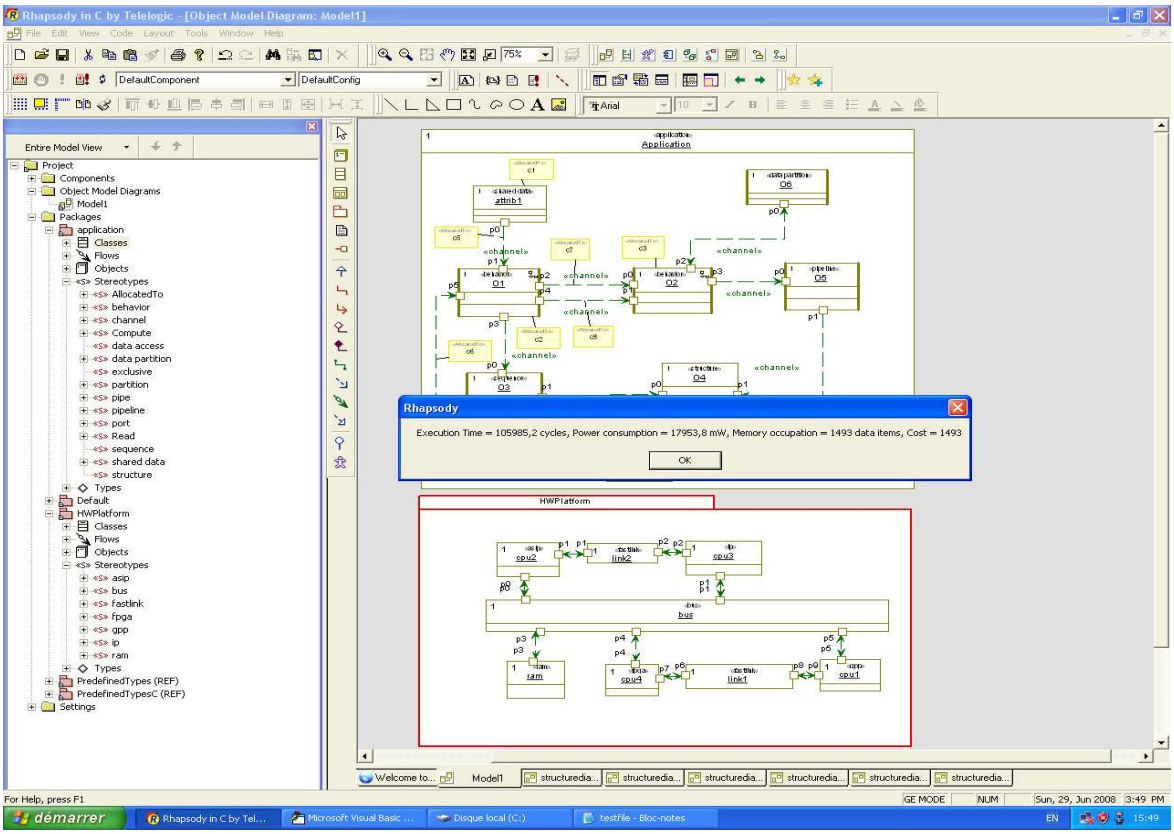


Figure 3.10. Capture Ecran de notre outil.

7.1. Le décodeur H264

La figure 3.11 montre le diagramme de blocs fonctionnels du décodeur H264. Nous avons exploité le code C du décodeur H264 fourni par Chemnitz [23] (avec une modification mineure). Nous avons récupéré l'ensemble des classes H264 en utilisant l'outil de reverse engineering intégré dans Rhapsody. Après cela, nous avons établi le diagramme de séquence (Figure 3.12) à partir de quel nous avons extrait un diagramme stéréotypé hiérarchique des tâches (Figure 3.13) suivant les guidelines de la section 4. Les WCETs des Méthodes sont obtenus par profilage et représentent des pourcentages (valeurs relatives) de l'ensemble du temps d'exécution de l'application. Dans cette étude cas, nous nous somme intéressés seulement par le passage du modèle séquentiel vars un modèle concurrent hiérarchique.

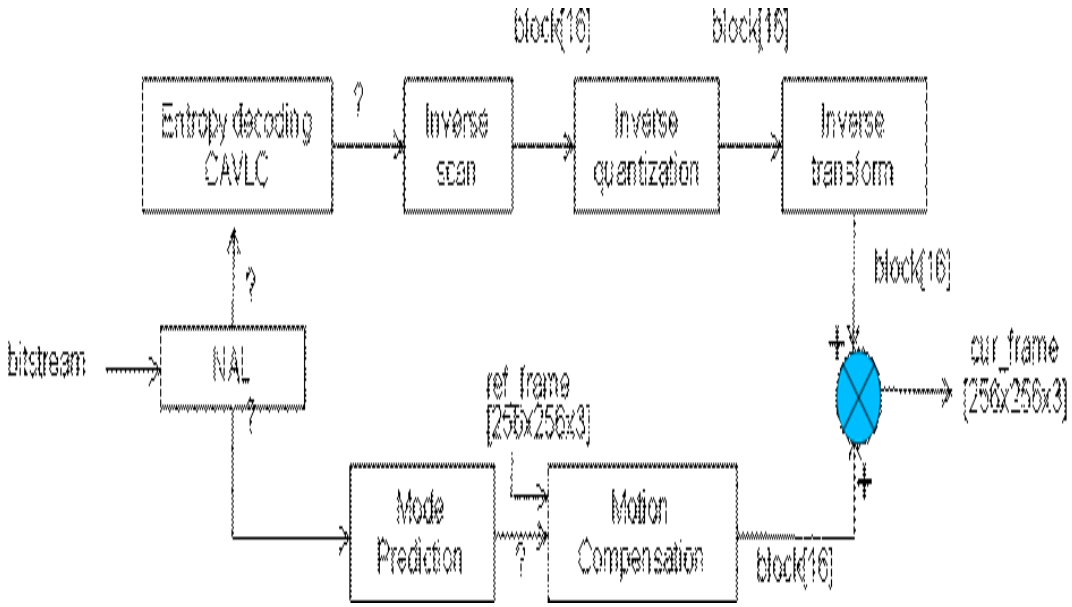


Figure 3.11. Diagramme de blocs fonctionnels de décodeur H264 [59].

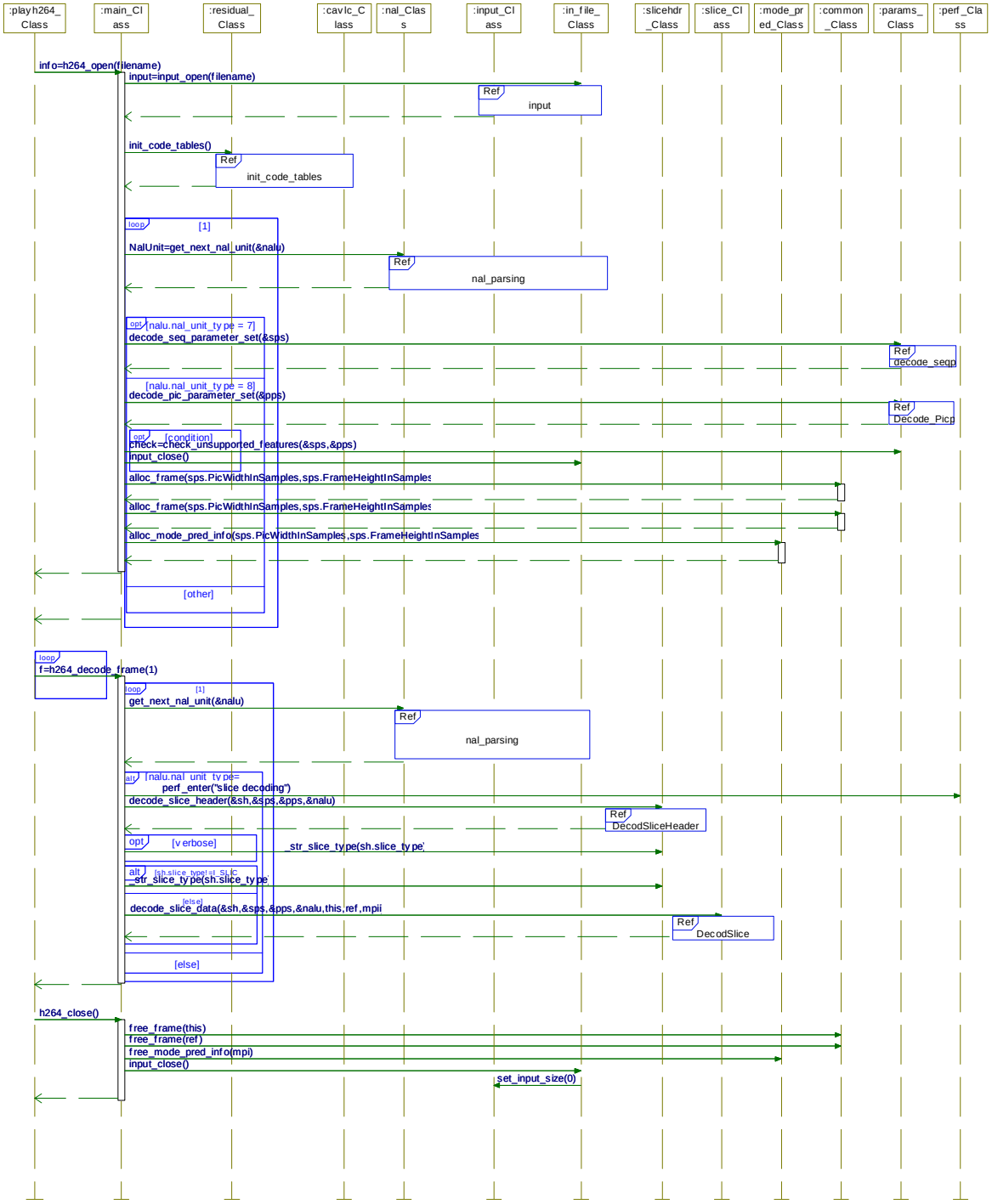


Figure 3.12. Diagramme de séquence principal de décodeur H264.

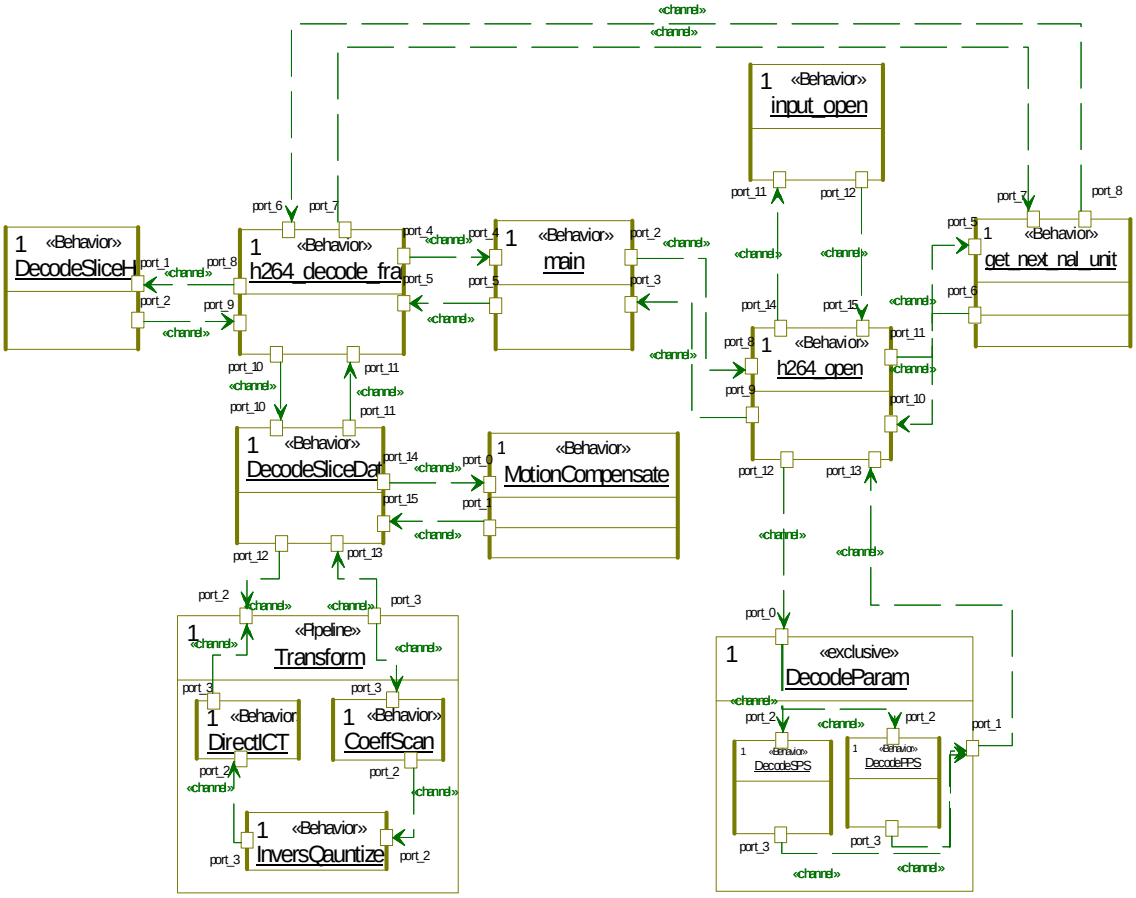


Figure 3.13. Modèle concurrent hiérarchique de décodeur H264.

7.2. Le décodeur MP3

Le diagramme de blocs fonctionnels du décodeur MP3 est présenté en figure 3.14 [115]. Toutes les tâches sont exécutées périodiquement, consomment et produisent de blocks de données de 576. Un tel bloc d'échantillon est appelé frame. Chaque frame peut être modélisé avec 576 jetons. La tâche 'Huffman decoder' opère sur deux frames en même temps, tant que les autres tâches traitent un seul frame à la fois. Le tableau 3.2 illustre les WCET des tâches sur un ARM7 ainsi que le temps total d'exécution et l'énergie consommée. Les tâches 'Antialias', 'Hybrid Synth', 'Freq. Inv', et 'Subb. Inv' peuvent s'exécuter en pipeline. Dans ce cas, elles doivent être allouées aux CPUs distincts. Les tâches 'Req.' et 'Reorder' peuvent s'exécuter en séquence. Les figurent 3.15 et 3.16 montrent la modélisation UML d'application et l'architecture respectivement. L'allocation des comportements feuils (tâches) aux différents composants d'architecture est montrée dans le tableau 3.2. Nous notons que l'allocation se fait manuellement à travers les contraintes UML.

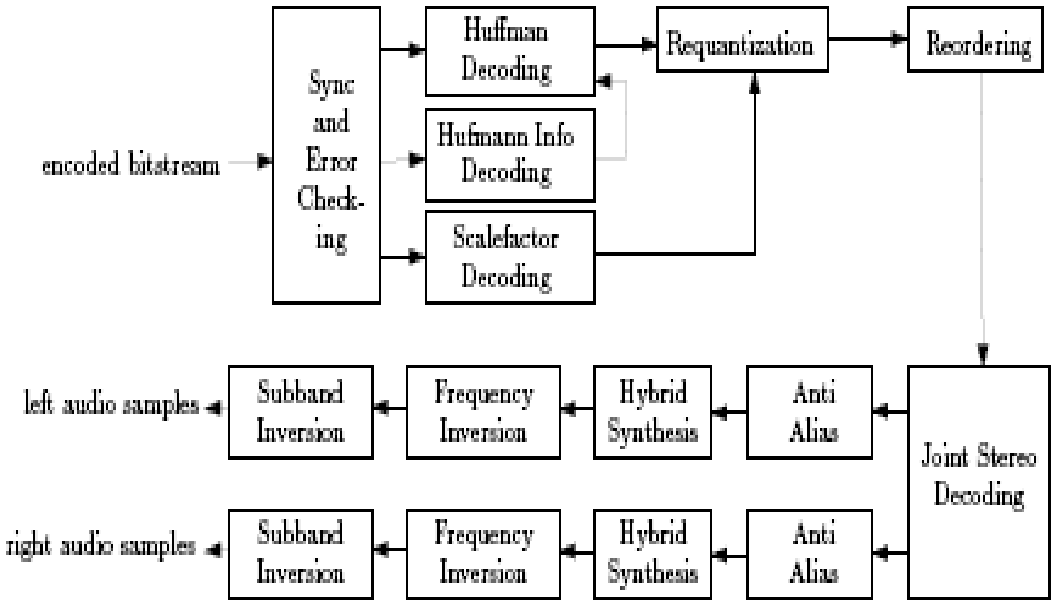


Figure 3.14. Diagramme de blocs fonctionnels de décodeur MP3 [115].

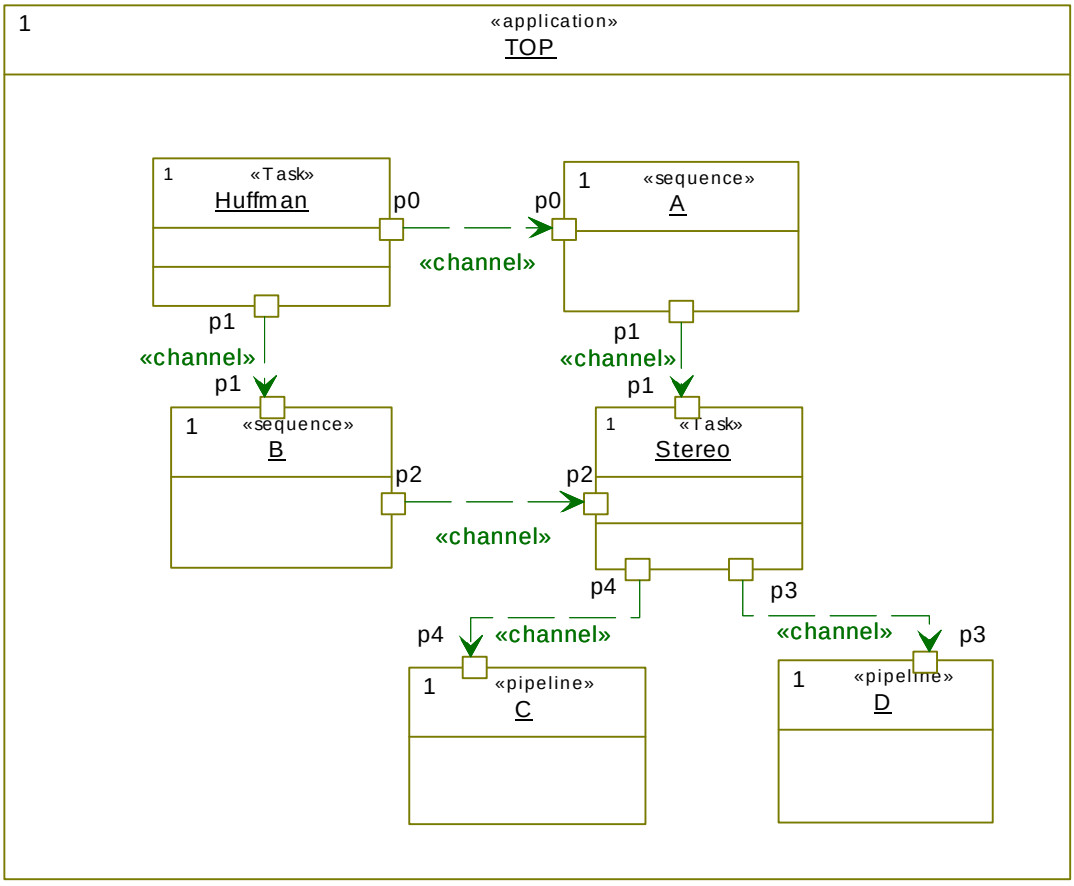


Figure 3.15. Modélisation UML de décodeur MP3.

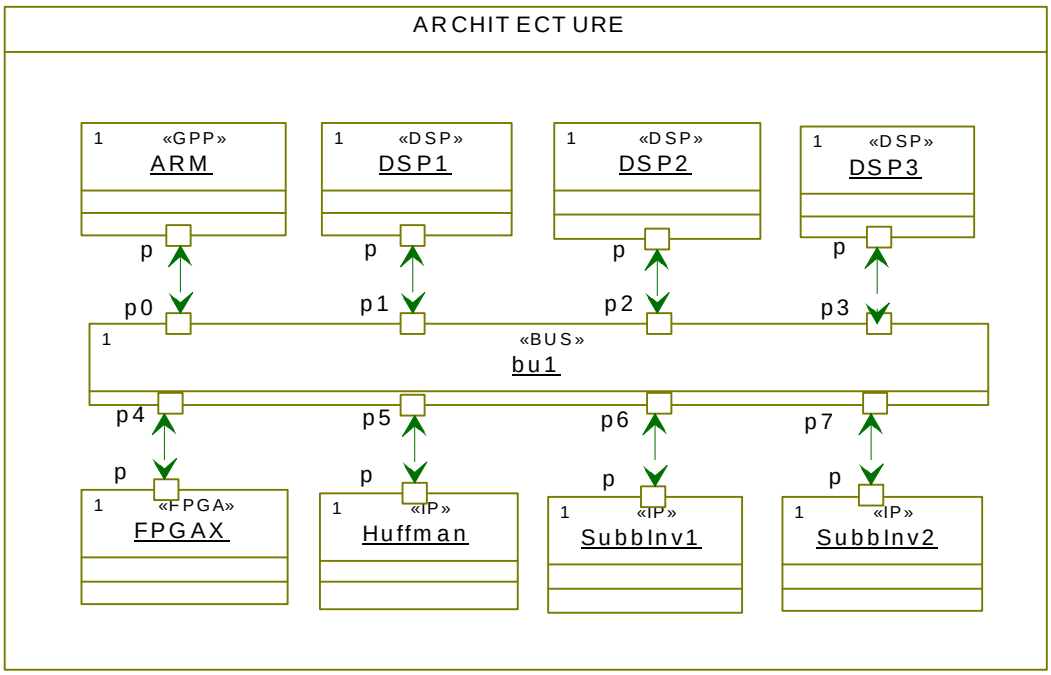


Figure 3.16. Modélisation UML d'architecture.

CPU/Bus	coût	Facteur de Vitesse	DP	SP	Tconfig (cycles)
ARM	500	1.0	1.0	0.2	
DSP1	700	0.6	1.2	0.3	
DSP2	700	0.5	1.3	0.4	
DSP3	700	0.5	1.3	0.4	
IP1	900	0.2	0.7	0	
IP2	800	0.3	0.7	0	
FPGA	800	0.7	0.8	0.3	3
Bus	200	10	0.5	0.1	

Tableau 3.1. Les différents paramètres d'architecture.

Tâche	WCET (cycles)	CPU	Temps Total (cycles)	Energie (unités)
Huffman	151977	IP2	1099000.33	974000.33
Req.	72695	DSP1		
Reorder	34684	DSP2		
Stereo.	53602	FPGA		
Antialias	409	ARM		
Hybrid Synth.	7414	DSP1		
Freq. Inv.	4912	ARM		
Subb. Inv.	1865001	IP1		

Tableau 3.2. Les WCETs des différentes tâches du MP3.

8. Conclusion

Dans ce chapitre, nous avons détaillé notre profil UML pour estimer les performances au pire de cas d'une application exécutant sur un MPSOC. Nous avons aussi présenté la méthodologie associée à ce profil.

Le point de départ de notre flot est un modèle purement séquentiel qui s'appuie sur le diagramme de séquence d'UML. A partir de ce modèle, un modèle concurrent hiérarchique est extrait. Ce dernier comporte tous les types du parallélisme qu'on peut trouver dans une application embarquée typique comme le parallélisme de tâches et de données. L'estimation de performances est basée essentiellement sur un modèle analytique. L'expérience du concepteur joue un rôle primordial dans l'exactitude des résultats obtenus. En fait, les valeurs d'estimation ne sont pas des valeurs absolues plutôt qu'elles sont des valeurs relatives que nous utilisons pour comparer entre les solutions possibles.

Quatrième chapitre

Modélisation UML de systèmes mixtes, vérification formelle, et simulation

Sommaire

1. Introduction
2. Logique de réécriture et le langage Maude.
3. Modélisation d’application
4. Modélisation d’architecture
5. Modélisation d’association
6. Transformation de modèles UML vers Maude.
7. Spécification de propriétés et vérification
8. Exemple
9. Transformation de modèles UML vers SystemC.
10. Conclusion.

1. Introduction

Dans le troisième chapitre, nous avons présenté notre méthode qui porte sur le premier niveau d'abstraction. Dans ce chapitre, nous introduisons notre approche qui correspond au deuxième niveau d'abstraction. Elle est fondée sur les principes suivants:

- Utilisation d'UML pour modéliser l'application et l'architecture.
- Séparation entre application et architecture.
- Séparation entre les tâches dominées par les données et les tâches dominées par le contrôle.
- Séparation entre calcul et communication.
- Abstraction de données.
- Utilisation du système logique Maude pour estimer la consommation d'énergie, et la vérification.
- Génération de code SystemC à partir de modèles UML.

2. Logique de Réécriture et le langage Maude

2.1. Logique de réécriture

La logique de réécriture a été introduite par Meseguer [101]. Cette logique ayant une sémantique complète unifie tous les modèles formels qui expriment la concurrence. Dans la logique de réécriture, les formules logiques sont appelées règles de réécriture. Ils ont la forme suivante: ***R: [t] -> [t'] si C***. la règle R indique que le terme t se transforme en t' si une certaine condition C est vérifiée. Un terme représente un état partiel d'un état global S du système. La modification de l'état global du système S à un autre état S' s'effectué à travers la réécriture parallèle d'un ou de plusieurs termes qui expriment les états partiaux du système.

2.2. Maude

Maude est un langage de spécification et de programmation basé sur la logique de réécriture [38]. Deux niveaux de spécifications sont définis en Maude. Le premier niveau concerne la spécification du système, tandis que la seconde porte sur la spécification de propriétés.

La spécification du système est fournie par la théorie de réécriture. Elle est principalement spécifiée par les modules système. Pour une bonne description modulaire, trois types de modules sont définis dans Maude.

Les modules fonctionnels permettent de définir les types de données et leurs fonctions par l'intermédiaire de la théorie des équations.

Les modules système définissent le comportement dynamique du système. Ce type de modules étend les modules fonctionnels en introduisant les règles de réécriture. Un maximum de degré de concurrence est offert par ce type de module.

Enfin, il y a les modules orientés objet qui peuvent être réduits aux modules système. Toutefois, les modules orientés objet offrent une syntaxe plus appropriée pour décrire la syntaxe des entités de base du paradigme objet, entre autres: les objets, les messages et la configuration.

Une seule règle de réécriture permet l'expression de la consommation de certains messages, l'envoi de nouveaux messages, la destruction d'objets, la création de nouveaux objets, le changement d'état de certains objets, etc. La figure 4.1 illustre l'utilisation d'un module système *BANK-ACCOUNT* qui permet de définir un objet compte bancaire *A* et deux opérations capables d'influer sur le contenu de compte *credit* et *debit* lors de l'exécution des règles de réécriture définies dans ce module. Note que, après l'exécution de la règle inconditionnelle [credit], le message *credit (A, M)* est consommé et le contenu de compte est augmenté. De la même manière, l'exécution de la règle *[debit]* exige que la condition (N> = M) soit vérifiée. L'exécution de telle règle consomme le message debit (A, M) et la réduit le contenu de compte.

```
mod BANK-ACCOUNT is
protecting INT .
including CONFIGURATION .
op Account : -> Cid.
op bal :_ : Int -> Attribute .
ops credit debit : Oid Nat -> Msg .
var A : Oid . vars M N :Int
rl [credit]: < A : Account | bal : N > credit(A, M) =>
< A : Account | bal:N + M >
crl [debit] : < A : Account | bal : N > debit(A, M) =>
< A : Account | bal : N - M > If  N >= M .
endm
```

Figure 4.1. Exemple d'un module système de Maude [38].

Le niveau spécification de propriété définit les propriétés du système qui doivent être vérifiées. Le système est décrit en utilisant un module système. En évaluant l'ensemble des états accessibles à partir d'un état initial, le vérificateur de modèles (model-checker) permet de vérifier une propriété donnée dans un état ou un ensemble d'états. Cette propriété est exprimée dans la logique temporelle LTL (Linear Temporal Logic) ou BTL (Branching Temporal

Logic). Le vérificateur de modèles supporté par la plate-forme Maude utilise essentiellement la logique LTL pour sa simplicité et les procédures de décision qu'il offre.

Dans un module LTL prédéfini, nous trouvons la définition des opérateurs pour la construction de formule (la propriété) dans LTL. Dans la figure 4.2, en masquant certains détails d'implémentation, on trouve une partie d'opérateurs LTL exprimée en Maude.

```
fmod LTL is
…
*** defined LTL operators
 op _->_ : Formula Formula -> Formula . *** implication
 op _<->_ : Formula Formula -> Formula . *** equivalence
 op <>_ : Formula -> Formula .      *** eventually
 op []_ : Formula -> Formula .      *** always
 op _W_ : Formula Formula -> Formula .  *** unless
 op _|->_ : Formula Formula -> Formula . *** leads-to
 op _=>_ : Formula Formula -> Formula . *** strong implication
 op _<=>_ : Formula Formula -> Formula . *** strong equivalence
…
endfm
```

Figure 4.2. Un module en Maude implémentant les opérateurs LTL [38].

Les opérateurs LTL sont représentés en Maude à l'aide d'une forme syntaxique similaire à leur forme originale. Par exemple, l'opération [] est définie en Maude par l'opérateur (always). Ce dernier est appliqué à une formule pour donner une nouvelle formule. En outre, nous avons besoin d'un opérateur indiquant si une formule est vraie ou fausse dans un certain état. Nous trouvons un tel opérateur (| =) dans un module prédéfini appelé SATISFACTION (figure 4.3).

```
fmod SATISFACTION is
 protecting LTL .
 sort State .
 op _|=_ : State Formula ~> Bool .
endfm
```

Figure 4.3. Un module en Maude implémentant l'opérateur de satisfaction d'une formule dans un état [38].

L'état *State* est générique. Après spécification du comportement du système en utilisant le module système du Maude, l'utilisateur peut spécifier plusieurs prédicats exprimant des propriétés liées au système. Ces prédicats sont décrits dans un nouveau module qui importe à son tour, deux modules: le premier qui décrit l'aspect dynamique du système, où le second est

Le module SATISFACTION. Soit, par exemple, M-PREDS (figure 4.4) le nom du module décrivant les prédicats sur les états du système. M est le nom du module décrivant le comportement du système. L'utilisateur doit spécifier que l'état choisi (la configuration choisie dans cet exemple), de son propre système est un sous-type (sub-type) de type State. En fin, on trouve le module MODEL-CHECKER (figure 4.5) qui offre la fonction model-Check. L'utilisateur peut appeler cette fonction tout en spécifiant un état initial et une formule. Le model-checker du Maude vérifie si cette formule est valable (en fonction de la nature de la formule et la procédure du model-checker du système Maude) dans cet état ou dans l'ensemble de tous les états atteignables à partir de l'état initial. Si la formule n'est pas valide, un contre exemple (counter example) est affiché. Le contre exemple concerne l'état dans lequel la formule n'est pas valable.

```
mod M-PREDS is
  protecting M .
  including SATISFACTION .
  subsort Configuration < State .
  …

endm
```

Figure 4.4. Un module Maude décrivant les prédicats [38].

```
fmod MODEL-CHECKER is
including SATISFACTION .
…
op counterexample : TransitionList TransitionList -> ModelCheckResult
[ctor] .

op modelCheck : State Formula ~> ModelCheckResult .
…
endfm
```

Figure 4.5. Un module Maude contenant les services offerts à l'utilisateur par le Model-checker du Maude [38].

3. Modélisation d'application

Comme mentionné auparavant, le modèle d'application est un réseau de tâches concurrentes qui communiquent via des canaux abstraits. Afin de supporter l'hiérarchie, nous introduisons un stéréotype appelé "Module". Ce stéréotype est appliqué sur les objets composites UML et désigne une tâche hiérarchique.

En Utilisant UML, chaque tâche est modélisée comme étant un objet (instance de classe) stéréotypé par le stéréotype "Dtask" pour les tâches dominées par les données et "Ctask" pour les tâches dominées par le contrôle. Chaque objet tâche possède un ou plusieurs ports.

Chaque port est caractérisé par son nom et la taille des données transmises en termes de nombre de jetons. Les canaux se caractérisent par une communication point-à-point entre les tâches. Ces tâches peuvent être exécutées sur la même ressource matérielle, ainsi que sur des ressources matérielles différentes. Dans notre cas, nous distinguons deux types de canaux: les canaux de données transportant les données proprement dites et les canaux de contrôle qui transportent les données de contrôle. Dans ce travail, nous nous sommes intéressés au modèle de calcul KPN qu'on l'adopte pour les tâches dominées par les données. Dans le paradigme de Kahn, les tâches communiquent via des FIFOs de taille infinie. La lecture des FIFOs est bloquante, tandis que l'écriture est non bloquante. Pour modéliser les canaux, nous utilisons les flots SysML stéréotypés par "Dchannel" pour les canaux de données et "Cchannel" pour les canaux de contrôle. Chaque canal de données a un paramètre qui est le nombre de jetons disponible dans la FIFO. Un canal de contrôle a un paramètre déterminant le nombre d'événements disponibles dans la FIFO. La figure 4.6 montre un exemple de modélisation structurelle d'une application qui se compose de deux modules nommés "Module1" et "Module2". "Module1" comprend une tâche dominée par le contrôle (T1) et deux tâches dominées par les données (T2 et T3). Module2 englobe deux tâches (T4 et T5) dominées par les données. Pour le module de plus haut niveau (c'est-à-dire l'application), nous introduisons un nouveau stéréotype appelé "Top". Nous constatons que lorsqu'un enfant possède le même port que le port de son père, nous n'avons pas besoin de modéliser le canal (par exemple les ports, in1, et po2 dans la figure 4.6).

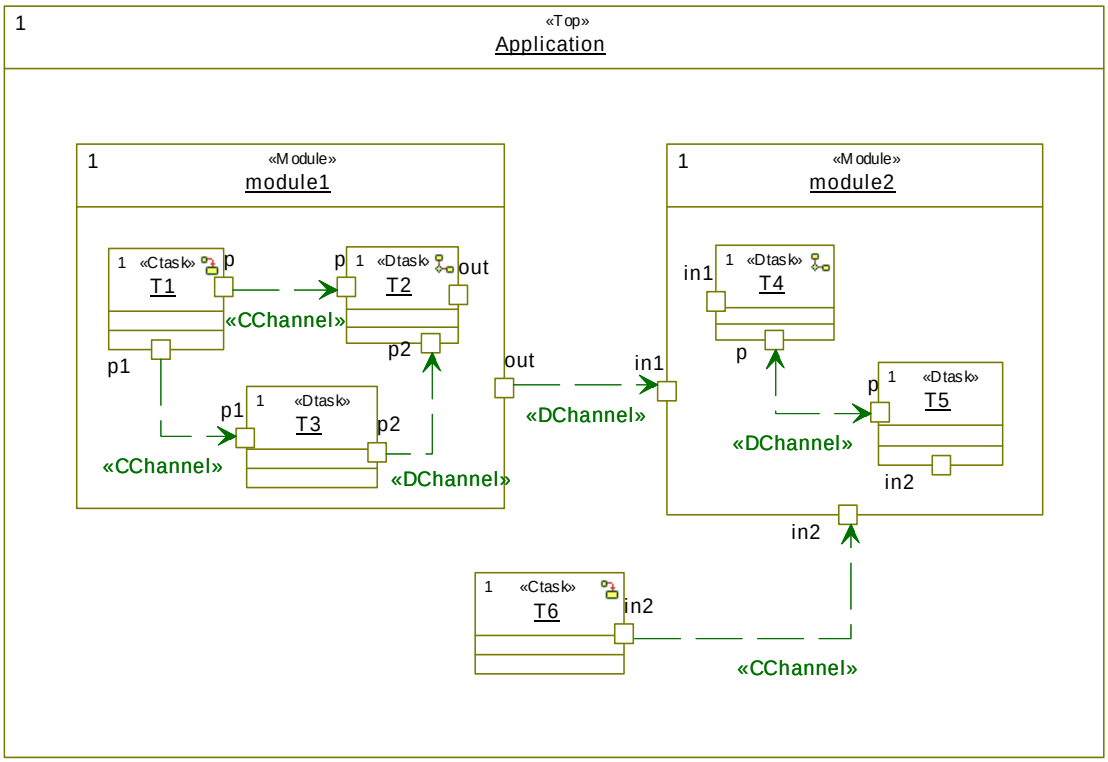
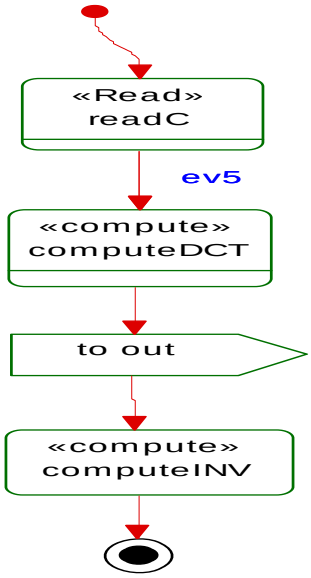
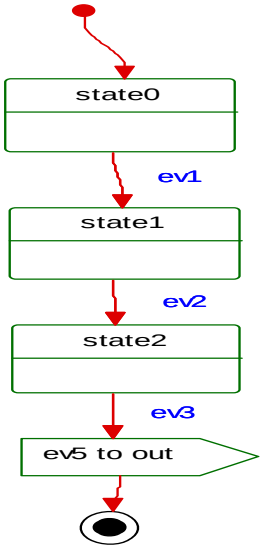


Figure 4.6. Modélisation UML d'application

Pour modéliser les comportements internes des tâches dominées par les données, nous utilisons les diagrammes d'activité d'UML2.0 avec opérations à forte granularité (CGA). Chaque CGA appartient à l'un des trois types génériques: Actions de Calcul (CA), Actions de Lecture (RA), ou Actions d'écriture (WA). Pour ce faire, nous définissons un nouveau stéréotype appelé "*compute*" avec deux paramètres précisant le nombre d'instructions élémentaires dans un calcul, et le type de l'opération élémentaire (c'est-à-dire nombre entier ou flottant). Chaque WA ou RA a deux arguments: le nom du port et la taille des données transmises. L'action d'écriture est modélisée à l'aide de "send action" d'UML2.0. Figure 4.7 (a) montre un diagramme d'activité avec deux CA: *computeDCT* et *computeINV*; une RA: *readC*, et une WA sur le port *p0*. Les tâches dominées par le contrôle sont représentées comme des machines à états finis (FSM) à l'aide des statecharts d'UML. Les Statecharts sont plus adaptés aux systèmes réactifs. Dans notre modèle, nous supposons que les opérations associées aux transitions et aux états s'exécutent en un temps nul. Dans l'exemple de la figure 4.7 (b), il y a trois événements d'entrée: EV1, EV2 et EV3, et un événement de sortie ev5 sur le port *out*.



(a)



(b)

Figure 4.7. Diagramme d'activité avec actions à forte granularité et diagramme d'états transitions avec zéro délai

4. Modélisation d'architecture

Afin d'estimer les performances de l'application, l'architecture matérielle doit également être modélisée. Ainsi, les composants de la plateforme sont modélisés comme une abstraction de leurs modèles à granularité fine et ils sont suffisamment génériques pour que l'ensemble de l'architecture puisse être utilisée pour la modélisation de tous les types de plateformes matérielles. Ici, nous identifions quatre composants génériques qui sont CPU, bus, pont, et RAM. Le modèle CPU est caractérisé par les paramètres suivants:

- **Temps de commutation de contexte de tâches:**

Chaque fois qu'une tâche est bloquée sur une opération de lecture, le processeur commute vers une autre tâche qui est prête à l'exécution. Il ya des cycles perdues au cours de cette commutation. Toutefois, ce nombre de cycles peut être déterminé en se basant sur l'expérience de concepteur.

- **Temps pour passer à l'état d'inactivité:**

Quand il n'y a pas de tâche prête à l'exécution, le CPU passe à l'état 'inactif'. Toutefois, avant de transiter vers cet état, le CPU consomme des cycles pour effectuer un test de faisabilité. Ces cycles peuvent être spécifiés par le concepteur.

- Nombre de cycles pour effectuer une opération élémentaire.
- La quantité moyenne de l'énergie consommée par cycle en mode de fonctionnement.

- La quantité moyenne de l’énergie consommée par cycle en mode inactif.
- Algorithme d’ordonnancement:

Dans notre cas, l'algorithme d'ordonnancement que nous adoptons est Round Robin sans préemption de tâches, et sans priorité. En vertu de Round Robin, lorsqu'une opération de lecture est effectuée, à chaque cycle, le CPU vérifie l'état de la FIFO associé au canal. Si la FIFO contient un échantillon de données pour la lecture, le CPU continue son travail dans le cycle actuel et la FIFO sera vérifiée à nouveau dans le prochain cycle. D'autre part, s'il n'y a pas d'échantillon disponible pour la lecture, le CPU arrête la tâche, et exécute la tâche suivante, qui est prête. Le CPU arrête une tâche, quand elle est bloquée ou elle se termine normalement après exécution de toutes ses instructions. En utilisant UML, nous définissons un stéréotype appelé "CPU" avec les paramètres mentionnés ci-dessus comme paramètres.

Un accès au bus n'est requis que pour effectuer une opération d'écriture. L’opération de lecture ne nécessite pas d'accès au bus car le CPU lit les données à partir de la FIFO dans son interface locale. Toutefois, si la communication entre les tâches se fait par l'intermédiaire de RAM, l’opération de lecture nécessite l'accès au bus. Le modèle de Bus se caractérise par le type de bus qui peut être partagé ou dédié, le taux de transfert exprimé en termes de nombre de jetons par cycle, la quantité d'énergie consommée par cycle en mode de fonctionnement, et la politique d'arbitrage dans le cas d'un bus partagé. Nous définissons un stéréotype appelé "BUS" avec les paramètres mentionnés ci-dessus.

La RAM est caractérisée par son taux de lecture, taux d'écriture, et la quantité d'énergie consommée par cycle en mode de fonctionnement. Nous définissons un stéréotype "RAM" avec les paramètres mentionnés ci-dessus.

Nous utilisons le pont pour relier plusieurs bus. Le pont est caractérisé par son taux de transfert, la quantité d'énergie consommée par cycle en mode de fonctionnement. Nous définissons un nouveau stéréotype "BRIDGE". La figure 4.8 montre un exemple d'architecture modélisée en utilisant Rhapsody.

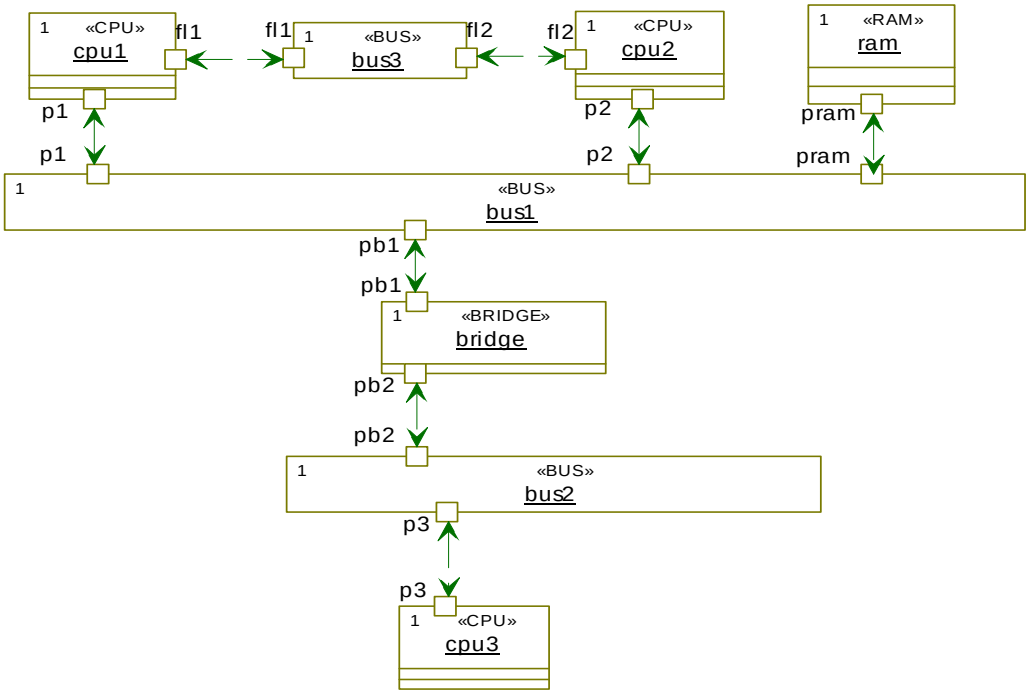


Figure 4.8. Modélisation UML d'architecture.

5. Modélisation d'association

L'association est modélisée par le stéréotype "*AllocateTo*" défini précédemment. On note que, s'il n'y a qu'un seul chemin entre deux composants matériels, alors on n'a pas besoin de préciser l'association de transfert et le chemin (c'est-à-dire le bus) est déterminé à partir de l'architecture. Par défaut, tous les transferts sont associés aux bus, à moins que, le concepteur introduit l'association de transfert via une RAM ou un bus dédié explicitement.

6. Transformation de modèles UML vers Maude

6.1. Transformation de l'aspect structurel

En utilisant Maude, chaque tâche dominée par les données est définie comme:

<A: Dtask | hwname: cpu, State: statut, Action: act>.

Où A est une instance de la classe Dtask; *hwname* est le nom de CPU auquel la tâche A est associée; State est l'état actuel de la tâche (*ready*, *run*, *wait*, *idle*); *Action* désigne l'opération effectuée par la tâche. Ici, nous avons trois cas selon le type d'action: si *Action* est de type calcul, alors on y ajoute les attributs *Nombre* et *Type*. Nombre désigne le nombre d'opérations de calcul élémentaires impliquées dans une action. *Type* est utilisé pour spécifier le type des opérations élémentaires (c'est-à-dire *Entier* ou *float*). Si l'action est une opération d'entrée /

sortie, alors on ajoute l'attribut *Token*. Ce dernier détermine le nombre de jetons de données transmises sur le canal. Enfin, l'action peut être une action d'attente (c'est-à-dire l'attente de l'occurrence d'un évènement). La tâche dominée par le contrôle est déclarée comme:

<A: Ctask | hwname: cpu, State: sta, FSMS: fsms >.

Où *FSMS* peut désigner le nom de l'état actuel de la FSM ou un état d'attente (c'est-à-dire l'attente d'occurrence d'un évènement). De même, chaque canal abstrait est déclaré comme une classe nommée Dchannel ou Cchannel:

<ch: Dchannel | hwname: hw, source: A, target: B, available: x> ou

<ch: Cchannel | hwname: hw, source: A , target: B, size: x>.

Où *ch* est le nom de canal abstrait; *Hwname* désigne le nom du bus ou de RAM à laquelle le canal est mappé; *source* et *target* désignent les tâches source et cible liées par *ch. available* désigne le nombre de jetons de données disponibles dans la FIFO du canal de données ou dans la RAM, et *size* désigne le nombre d'événements disponibles dans la FIFO d'événements attachée au canal de contrôle.

En utilisant Maude, nous déclarons chaque CPU comme une classe avec un ensemble d'attributs:

<cpu: CPU | LinkTo: bus, ContextSwitch: cont, GoIdle: idl, iop: iop, Fop: fop, Power: pw, PowIdle: pwd, TPower : tp> .

Où *cpu* est le nom du processeur; *LinkTo* désigne le nom du bus auquel le processeur est connecté. Dans le cas où *cpu* est connecté à plus d'un bus, il faut rajouter plusieurs attributs *LinkTo*. *ContextSwitch* désigne le temps de commutation de contexte; *GoIdle* désigne le nombre de cycles requis pour le passage à l'état de repos; *iop*, et *Fop* désignent les nombres des cycles requis pour l'exécution d'une opération élémentaire entière et flottante respectivement; *Power*, et *PowIdle* désignent les quantités d'énergie moyennes consommée par cycle en mode de fonctionnement et de repos, respectivement, et enfin *TPower* est l'énergie totale consommée par le CPU. Nous déclarons le bus comme une classe nommée BUS: *<bus: BUS | Speed: sp, Power: pb, TPower: tpb, free: true>*.

Où *bus* est le nom du bus ; *Speed* désigne le taux de transfert du bus, et *free* est une variable booléenne. Quand elle est vraie, *free* signifie que l'accès est accordé par une tâche, sinon elle est fausse. L'attribut *free* est utilisé uniquement avec un bus partagé. Lorsque plus d'une tâche demande l'accès à un bus au même temps, une politique d'ordonnancement doit être définie. Dans notre cas, nous adoptons une politique FIFO.

Nous déclarons la RAM comme une classe nommée RAM :

<ram: RAM | LinkTo: bus, Rrate: rr, Wrate: wr, Power: pb, TPower: TPR>.

Où *ram* est le nom de l'instance de RAM; *Rrate* et *Wrate* désignent les taux de lecture et d'écriture respectivement.

Bridge est déclaré comme une classe nommée BRIDGE :

<bridge: BRIDGE | LinkTo:bus1, LinkTo: bus2, Speed : sp, Power: pb, TPower: TPB>.

Où bus1 et bus2 désignent les deux bus auxquels le pont est connecté.

UML	**Maude**
Classe	Classe
Stéréotype	Attribut
Flot SysML (channel)	Classe (channel)
Tagged value	Attribut
Diagramme d'activité action (CGA)	Attribut
Etat FSM	Attribut
transition diagramme d'activité	Règle de réécriture
transition FSM	Règle de réécriture

Tableau 4.1. Correspondance entre UML et Maude

6.2. Transformation de l'aspect comportemental

En utilisant Maude, le comportement de tâche dominée par les données est défini comme une séquence de règles de réécriture. Chaque règle de réécriture correspond à une transition entre deux états qui déclenche l'exécution d'un CGA. Une règle de réécriture peut être conditionnée par la survenance de certains événements et / ou des conditions éventuellement. Ci-dessous, nous donnons des règles de réécriture pour l'exemple de la figure 4.7 (a). Supposons que la tâche A est associée au processeur *cpu*.

*rl [start] : ***1*
start(A)
< A : Dtask | hwname : cpu, state : ready, > =>
< A : Dtask | hwname : cpu1, state : run, action : readC, token : 5 > .

*crl [readCwait] : ***2*
< A : Dtask | hwname : cpu, state : run, action : readC, token : n >
< cpu : CPU | LinkTo : bus, ContextSwitch: cont, GoIdle : idl, Iop : iop, Fop : fop, Power : pw, PowIdle : pwd, TPower : tp >
< ch1 : Dchannel | hwname : bus1, source : A, target : B, available : x > < C : Dtask | hwname : cpu, state : s > =>
< A : task | hwname : cpu, state : wait, action : readC, token : n – x >

*< cpu : CPU | LinkTo : bus, ContextSwitch : cont, GoIdle : idl,Iop : iop, Fop : fop, Power : pw, PowIdle : pwd, TPower : tp + (float(cont) * pw) > < ch1 : Dchannel | hwname : bus1, source : A, target : B, available : 0 > < C : Dtask | hwname : cpu, state : s > wakeup(C)*
if (x < n) and (s == ready) .

*crl [readCreadEV5] : ***3*
< A : Dtask | hwname : cpu, state : run, action : readC, token : n >
< ch1 : Dchannel | hwname : bus1, source : A, target : B, available : x >
=>
< A : Dtask | hwname : cpu, state : run, action : waitEV5 >
< ch1 : Dchannel | hwname : bus1, source : A, target : B, available : x - n >
if n < x or n == x .

*crl [waitEV5] : ***4*
< A : Dtask | hwname : cpu, state : run, action : waitEV5 >
< cpu : CPU | LinkTo : bus, ContextSwitch : cont, GoIdle : idl, Iop : iop, Fop : fop, Power : pw, PowIdle : pwd, TPower : tp >
< ch : Cchannel | hwname : bus1, source : A, target : B, size : sz >
< C : Dtask | hwname : cpu, state : s, action : act, token : n1 >
=>
< A : Dtask | hwname : cpu, state : wait, action : waitEV5 >
*< cpu : CPU | LinkTo : bus, ContextSwitch : cont, GoIdle : idl,Iop : iop, Fop : fop, Power : pw, PowIdle : pwd, TPower : tp + (float(cont) * pw) >*
< ch : Cchannel | hwname : bus1, source : A, target : B, size: sz >
< C : Dtask | hwname : cpu, state : s, action : act, token : n > wakeup(C)
if (sz == 0) and (s == wait) and (act == writeC) .

*crl [EV5occurence] : ***5*
< A : Dtask | hwname : cpu, state : run, action : waitEV5 >
< ch : Cchannel | hwname : bus1, source : A, target : B, size : sz >
=>
< A : Dtask | hwname : cpu, state : run, action : computeDCT, Nombre : 1000, Type : integer > < ch : Cchannel | hwname : bus1, source : A, target : B, size : sz – 1 > if sz > 0 .

*rl [computeDCT] : ***6*
< A : Dtask | hwname : cpu, state : run, action : computeDCT, Nombre : 1000, Type : integer > < cpu : CPU | LinkTo : bus ,ContextSwitch : cont, GoIdle : idl, Iop : iop, Fop : fop, Power : pw, PowIdle : pwd, TPower : tp >
=>
< A : Dtask | hwname : cpu, state : run, action : writeA, token : 5 >
*< cpu : CPU | LinkTo : bus, ContextSwitch : cont, GoIdle : idl, Iop : iop, Fop : fop, Power : pw, PowIdle : pwd, TPower : tp + (float(1000 * iop)* pw) > .*

*rl [AccessBus] :***7*
< A: Dtask | hwname : cpu, state : run, action : writeA , token : n >
< bus: BUS | Speed : sp, Power : pb, TPower: tpb, free : f >
=> if f == true then < A : Dtask | hwname : cpu, state : run, action : writeA , token : n> <bus : BUS | Speed : sp, Power : pb, TPower :tpb, free : false > AccessBus(bus) else
< A : Dtask | hwname : cpu, state : wait, action : writeA , token : n >
< bus : BUS | Speed : sp, Power : pb, TPower : tpb, free : f > fi .

*rl [writeA] : ***8*
AccessBus(bus)
< A : Dtask | hwname : cpu, state : run, action : writeA , token : n >
< ch2 : Dchannel | hwname : bus, source : A, target : D, available : x >
< bus : BUS | Speed : sp, Power : pb, TPower : tpb, free : f >
=>
< A : Dtask | hwname : cpu, state : run, action : computeINV, Nombre : 200, Type : float >
< ch2 : Dchannel | hwname : bus, source : A, target : D, available : x + n >
*< bus : BUS | Speed : sp, Power : pb, TPower : tpb + (pb * (float(n) / sp)) , free : true > .*

*crl [ReaccessBus] : ***9*
< A: Dtask | hwname : cpu, state : wait, action : writeA , token : n >
< bus: BUS | Speed : sp, Power : pb, TPower: tpb, free : f >
=>
< A: Dtask | hwname : cpu, state : run, action : writeA , token : n > < bus: BUS | Speed : sp, Power : pb, TPower: tpb, free : f > if f == true .

*rl [computeINV] : ***10*
< A : Dtask | hwname : cpu, state : run, action : computeINV, Nombre : 200, Type : float >
< cpu : CPU | LinkTo : bus ,ContextSwitch : cont, GoIdle : idl, Iop : iop, Fop : fop, Power : pw, PowIdle : pwd, TPower : tp >
=>
< A : Dtask | hwname : cpu, state : run, action : end >
*< cpu : CPU | LinkTo : bus ,ContextSwitch : cont, GoIdle : idl, Iop : iop, Fop : fop, Power : pw, PowIdle : pwd, TPower : tp + float(200 * fop) * pw > .*

*rl [endA] : ***11*
< A : Dtask | hwname : cpu, state : run, action : end >
< cpu : CPU | LinkTo : bus ,ContextSwitch : cont, GoIdle : idl, Iop : iop, Fop : fop, Power : pw, PowIdle : pwd, TPower : tp >
=>
< A : Dtask | hwname : cpu, state : idle >
*< cpu : CPU | LinkTo : bus ,ContextSwitch : cont, GoIdle : idl, Iop : iop, Fop : fop, Power : pw, PowIdle : pwd, TPower : tp + float(idle * pwd) > .*

*rl [wakeupC] : ***12*
wakeup(C)
< C : task | hwname : cpu, state : ready, >
=>
< C : task | hwname : cpu, state : run, action : Compute1 , Number : 1000, Type : float > .

La première règle force la tâche *A* à changer son état de *ready* à exécuter la première instruction à savoir *readC* (lire cinq jetons). Nous utilisons la méthode *start* pour spécifier la première tâche à exécuter par *cpu*.

La deuxième règle force la tâche A à attendre sur l'opération de lecture, si le nombre de jetons sur le canal d'entrée est inférieur au nombre de jetons lus par la tâche. Dans ce cas, *cpu* s'arrête et réveille la tâche suivante nommée C qui est prête en utilisant le message *wakeup*.

Au cours de ce changement de contexte, nous mettons à jour *TPower*. En général, quand une tâche est bloquée, le CPU réveille la prochaine tâche qui est prête à l'exécution (l'ordre d'exécution est extrait à partir d'association UML).

La troisième règle spécifie la lecture succèsante de *readC*. Ensuite la tâche A attend la survenance de l'événement *ev5*. Dans cet exemple, nous supposons que la lecture ne nécessite pas d'avoir un accès au bus. Toutefois, si le canal est associé à la RAM, alors nous devons spécifier l'accès au bus et la mise à jour de l'énergie consommée qui due à la lecture de RAM, les transferts sur le bus et le pont.

La règle 4 force *cpu* à bloquer A, commuter le contexte et réveiller la tâche C.

La règle 5 spécifie la survenance de l'événement *ev5* qui permet à la tâche d'exécuter *computeDCT* qui inclut 1000 opérations entières. La règle 6 spécifie l'exécution de l'action *computeDCT* et de la mise à jour de *TPower*.

La règle 7 permet à la tâche A d'accorder l'accès de bus pour écrire des données: si le bus est libre (*free == true*), alors l'accès est accordé, par ailleurs, la tâche A sera bloquée.

La règle 8 spécifie l'opération de l'écriture et la mise à jour correspondante de l'énergie. Depuis que les écritures ne sont pas bloquantes, nous n'avons aucune condition sur la taille maximale des FIFOs de données. La tâche libère le bus quand elle termine l'écriture. La règle 9 permet à une tâche bloquée à ré-accéder au bus partagé.

La règle 10 spécifie un calcul qui s'appelle *computeINV* incluant 200 opérations flottantes et la mise à jour d'énergie consommée.

La règle 11 spécifie l'achèvement de l'exécution de la tâche. Ici, la tâche A change son état en *idle* et la valeur de *TPower* est mise à jour.

La règle 12 spécifie le réveil de la tâche C.

Pour spécifier les comportements des tâches dominées par le contrôle, nous appliquons les mêmes principes que pour les tâches dominées par les données. Toutefois, au lieu de modéliser les actions de calcul et les données entrées / sorties, nous spécifions les états de la FSM et les événements d'entrée / sortie associés. Ci-dessous, nous donnons trois règles de réécriture pour l'exemple de la figure 4.7 (b).

rl [start] :
< A: Ctask | hwname: cpu, state: ready, FSMS: BEGIN >
=>
< A: Ctask | hwname: cpu, state: run, FSMS: STATE0 > **.**

rl [state0] :
< A: Ctask | hwname: cpu, state: run, FSMS: STATE0 >
=> < A: Ctask | hwname: cpu, state: run, FSMS: waitEV1 > **.**

Crl [waitonEV1] :
< A : Ctask | hwname : cpu, state : run, FSMS : waitEV1 >
< cpu : CPU | LinkTo : bus, ContextSwitch : cont, GoIdle : idl, Iop : iop, Fop : fop, Power : pw, PowIdle : pwd, TPower : tp >
< ch3 : Cchannel | hwname : bus, source : A, target : B, size : sz >
< D : Dtask | hwname : cpu, state : s, action : act >
=>
< A : Ctask | hwname : cpu, state : wait, FSMS : waitEV5 >
*< cpu : CPU | LinkTo : bus, ContextSwitch : cont, GoIdle : idl,Iop : iop, Fop : fop, Power : pw, PowIdle : pwd, TPower : tp + (float(cont) * pw) >*
< ch3 : Cchannel | hwname : bus, source : A, target : B, size: sz >
< D : Dtask | hwname : cpu, state : s, action : act > wakeup(D)
if (sz == 0) and (s == wait) and (act == waitEV5) .

On note que l'estimation de la consommation d'énergie est basée sur l'hypothèse que l'énergie est proportionnelle au temps (nombre de cycles). Ici, nous nous somme intéressés à la consommation dynamique depuis que la consommation statique est généralement constante. Nous observons également que les tâches dominées par les données consomment plus d'énergie que celles de tâches dominées par le contrôle. Un autre facteur qui influe sur la consommation d'énergie est la politique d'ordonnancement (nombre de commutations de contexte).

7. Spécification de propriétés et vérification

A ce niveau d'abstraction, on peut vérifier certaines propriétés souhaitables (non-souhaitables) comme le deadlock. Ce dernier peut se produire en raison d'un ordonnancement inapproprié. En utilisant la commande de Maude *"search in application: initial =>! X:conf such that TaskEnd(X:conf) == true."*, on peut facilement vérifier si toutes les tâches atteignent l'état de fin d'exécution (ce qui signifie qu'il n'ya pas du deadlock). Notons que la commande *"search S =>! S', where =>!"* signifie que nous sommes entrain de chercher les états terminaux. *TaskEnd* est une fonction définie comme suit:

sort conf .
subsort conf < Configuration .
op sta : conf -> states .
op TaskEnd : conf -> Bool .
eq sta (< T1 : Dtask | hwname : cpu, state : st >) = st .
eq sta (< T2 : Ctask | hwname : cpu, state : st >) = st .
eq TaskEnd (T) = if sta(T) == idle then true else false fi .

De même façon, on peut vérifier si le bus est toujours occupé, qui est une propriété non-souhaitable. Pour cette raison nous utilisons la commande *"search in application : initial =>! X:conf such that BusBusy(X:conf) == true."* où *BusBusy* est une fonction définie comme suit:

op BusBusy : conf -> Bool .

eq BusBusy (< bus : BUS | Speed : sp, Power : pb, TPower : tpb, free : bool >) = if bool == false then true else false fi .

Une autre propriété importante que nous pouvons vérifier est le fait que le nombre de jetons de données stockées en FIFO des canaux de données ne dépasse pas un certain seuil dans chaque état du système. Pour cette raison, on peut définir une propriété *FIFOsize* comme:

var cf : configuration .
op FIFOsize : Configuration -> Prop .
ops ch A B bus : -> Oid .
vars x TS : Nat .
ceq < ch : Dchannel | hwname : bus, source : A, target : B, available : x, threshold : TS > cf |= FIFOsize (< ch : Dchannel | hwname : bus, source : A, target : B, available : x, threshold : TS > cf) = true if x < TS .

En utilisant la commande: *"red modelCheck(initial, []FIFOsize(initial)) ."*, nous pouvons facilement vérifier la propriété FIFOsize. *initial* est le nom de la configuration initiale.

Le symbole [] signifie globalement: la propriété doit être vérifiée pour chaque état de chaque chemin de calcul. Si la propriété n'est pas vraie un contre-exemple est indiqué.

8. Exemple

Un exemple est présenté avec objectif de démontrer le niveau d'abstraction de l'application, et la vérification formelle. Supposons qu'il y a trois tâches dominées par les données nommées A, B, C, et deux tâches dominées par le contrôle nommées Controller1 et Controller2 qui se communiquent par l'intermédiaire de deux canaux de données ch1 (A et B), et, ch2 (B et C) et quatre canaux de contrôle ch3 (Controller1 et A), ch4 (Controller1 et Controller2), Ch5 (C et Controller2), et Ch6 (Controller2 et Controller1). Les tâches A et C sont associées à CP2, (A s'exécute en premier). B est associée à CP3, Controller1 et Controller2 sont associés à CP1. *ch1* est associé à *ram*. Depuis que Controller1 et Controller2 sont assignés au même CPU (cpu1), ch4 et Ch6 sont associés à cpu1. ch2, ch3, et ch5 sont assignés aux buses (voir figure 4.12). La tâche A exécute *computeA1* incluant 300 opérations entières et écrit 10 jetons de données sur ch1. Ensuite, elle attend l'occurrence de l'événement *request1* provenant de Controller1 (via ch3) pour effectuer *computeA2* incluant 120

opérations flottantes. La tâche B effectue *computeB1* de 100 opérations entière et tente de lire 5 jetons de données à partir du ch1, puis elle effectue *computeB2* de 50 opérations flottantes et écrit 8 jetons de données sur ch2. La tâche C tente de lire 5 jetons de données de ch2, effectue *computeC* qui inclut 1000 opérations entières, puis elle envoie un événement *ok1* à Controller2 via ch5 (voir figure 4.10). La figure 4.11 montre les statecharts de Controller1 et Controller2. Avant l'exécution, nous supposons que le nombre de jetons disponibles sur ch1 et ch2 est égal à 0, et 7 respectivement.

Pour réaliser notre objectif, nous avons adopté l'environnement Rhapsody 7.2 d'IBM (à l'origine, c'était une marque de commerce déposée de Telelogic Inc.) [105]. Rhapsody offre un environnement de conception visuel pour créer les exigences et les modèles des systèmes en utilisant les diagrammes UML™ et SysML™. Rhapsody permet au concepteur du système embarqué d'analyser, concevoir, mettre en œuvre et de tester son système. Il permet aussi la génération automatique de tests et le reverse ingénierie. En utilisant Rhapsody, nous pouvons facilement modéliser et simuler notre application.

Les codes Maude et SystemC sont générés automatiquement sous forme de fichiers textes grâce à l'interpréteur VB (Visual Basic) intégré dans Rhapsody (voir les figures 4.12 et 4.13). Ce dernier offre une API pour le parsing de modèles UML et la collecte de données. Bien sûr, on peut utiliser la représentation textuelle XMI pour générer le code, mais nous préférons d'utiliser le VB, pour sa simplicité. La spécification Maude et le résultat de réécriture de notre exemple, sont montrés dans les figures 4.14 et 4.15 respectivement.

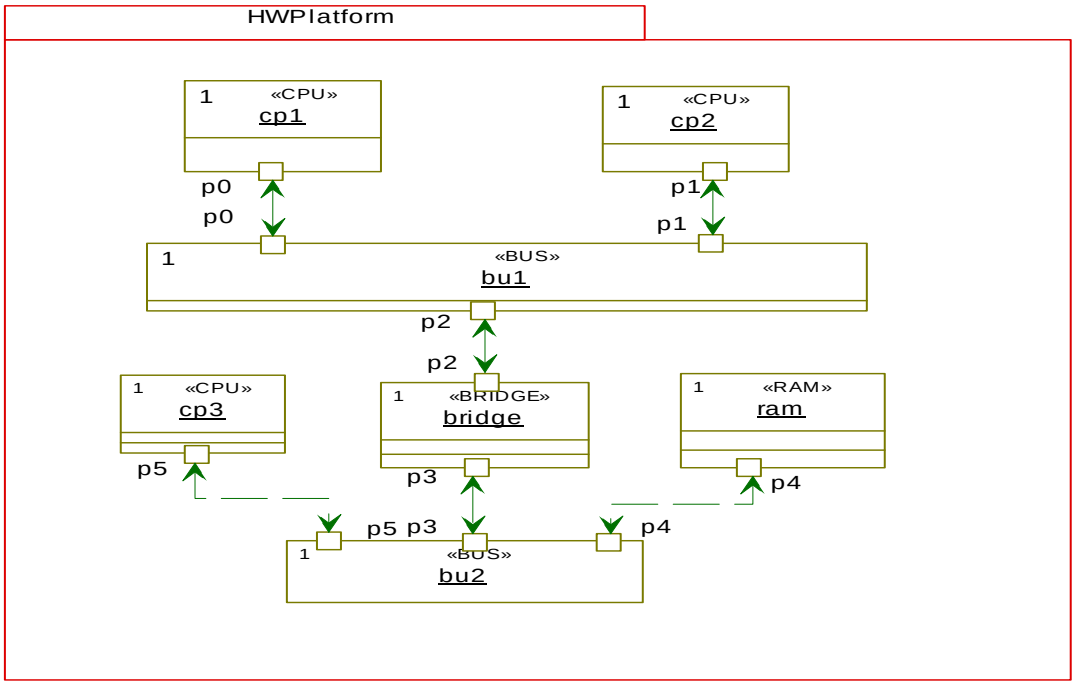


Figure 4.9. Modélisation UML d'architecture pour l'exemple.

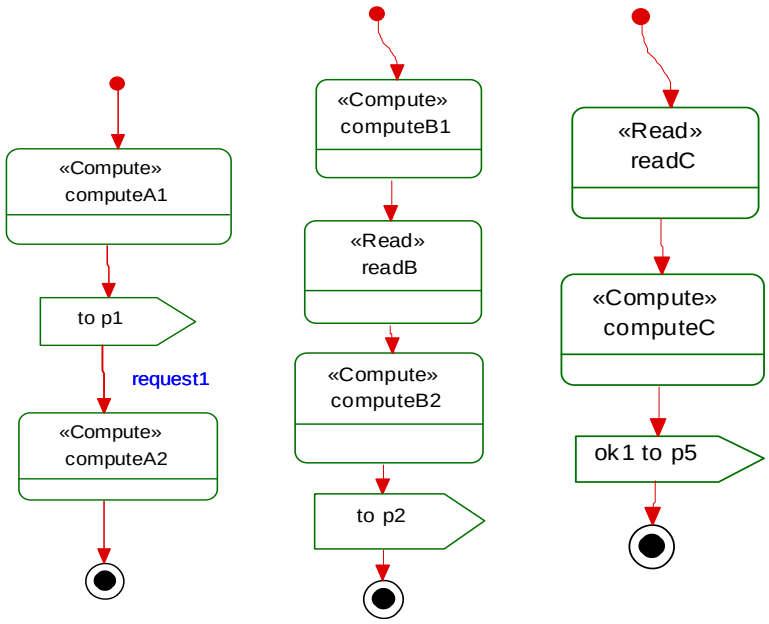


Figure 4.10. Modélisation de comportements internes pour les tâches A, B et C.

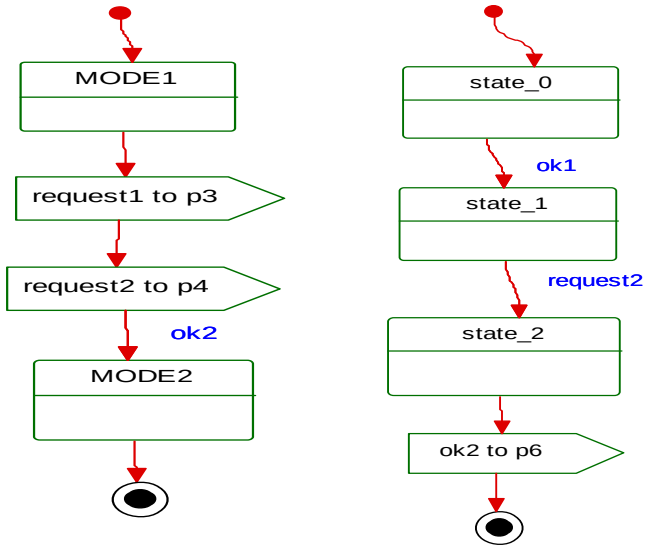


Figure 4.11. Modélisation de comportements internes pour les deux contrôleurs.

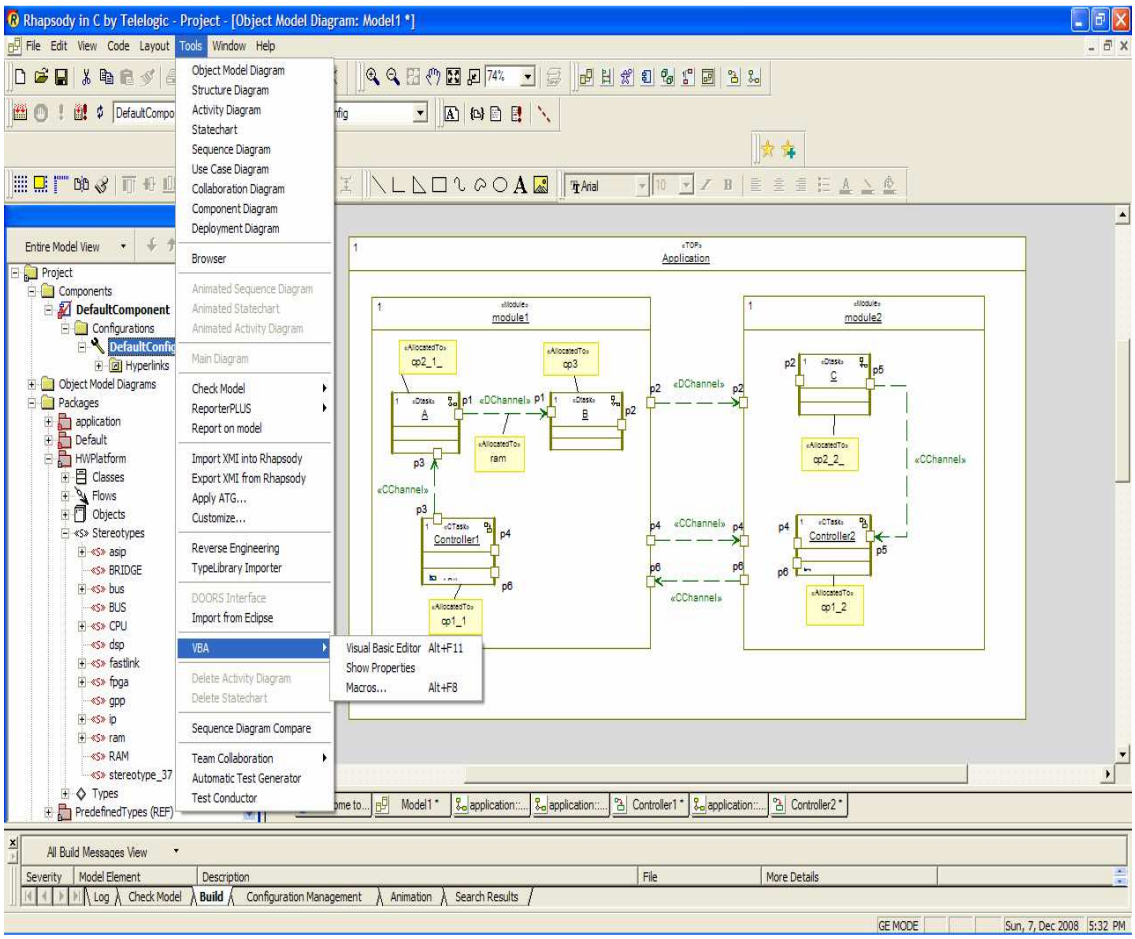


Figure 4.12. Modélisation avec Rhapsody.

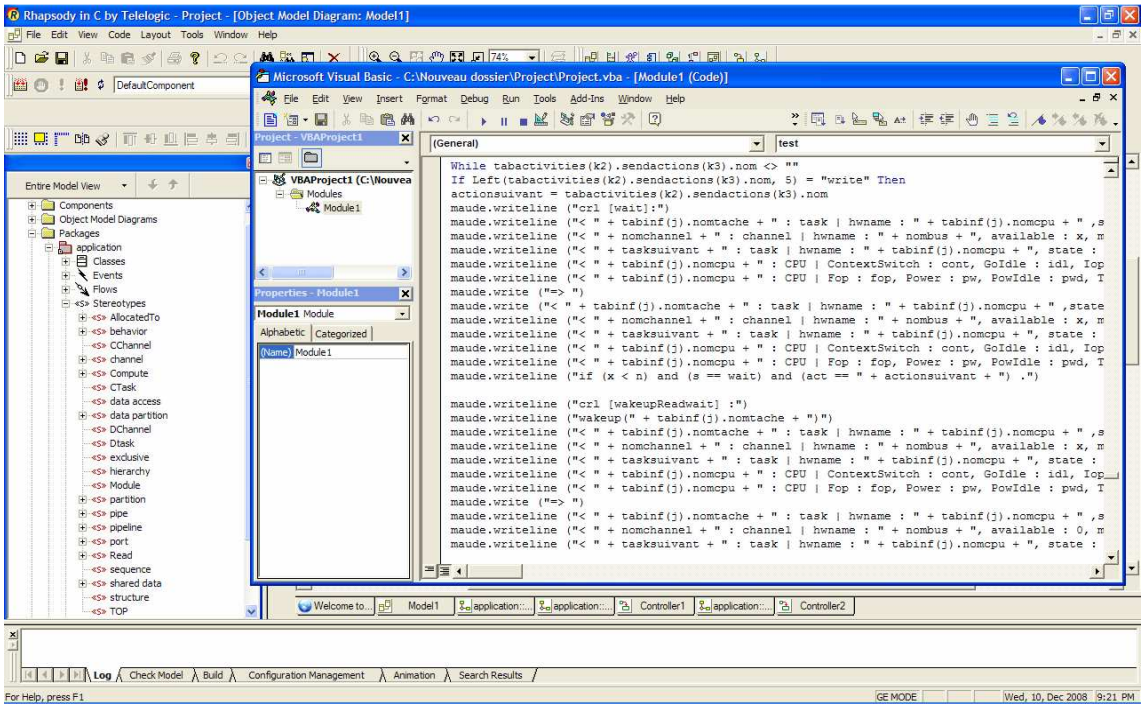


Figure 4.13. Programmation en VB du Rhapsody.

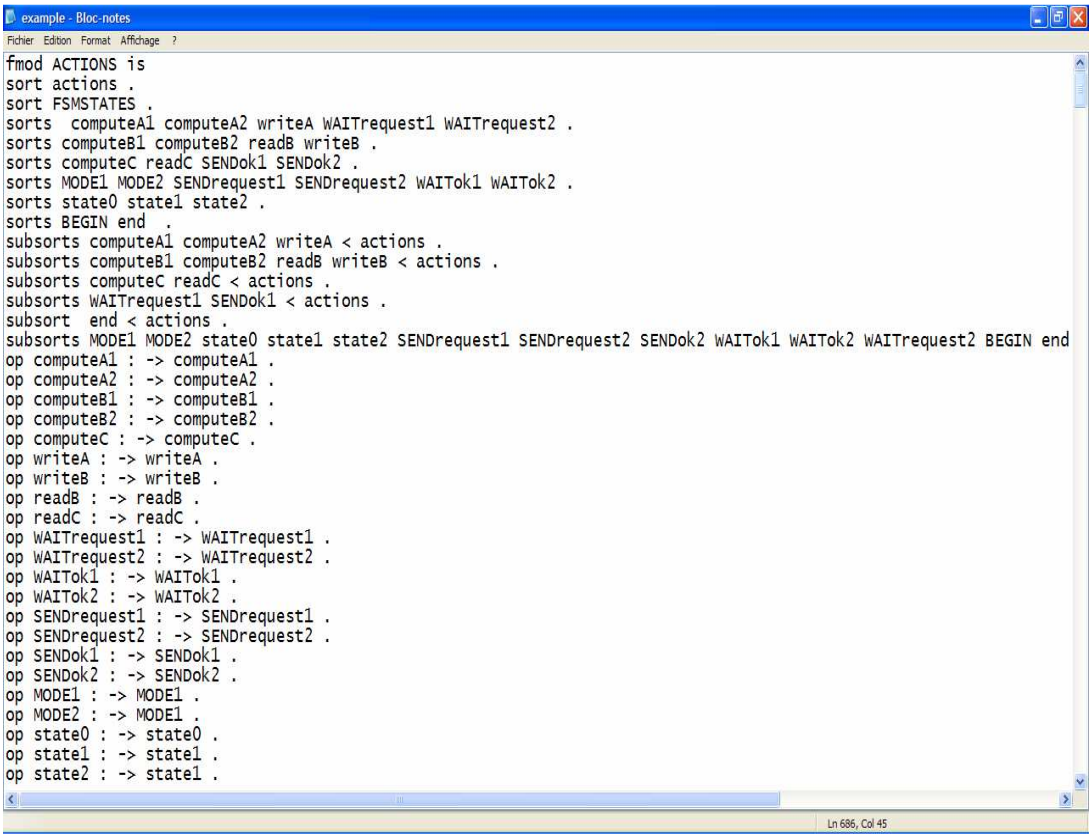


Figure 4.14. Un fragment du code Maude pour l'exemple.

Figure 4.15. Résultat de réécriture pour l'exemple.

9. Passage de modèles UML vers SystemC

Depuis que SystemC supporte la technologie objet, il semble un bon choix pour la mise en œuvre de modèles UML en vu de leurs simulation fonctionnelle et temporelle (éventuellement leurs synthèse). Afin de générer automatiquement du code SystemC à partir de modèles UML (diagrammes de structure, l'activité des diagrammes et des statecharts), nous devons raffiner les modèles UML initiaux en ajoutant des informations détaillées concernant le type de données transmises entre les tâches et les tailles des FIFOs attachées aux canaux virtuels. Cette information est présentée comme un paramètre aux ports UML (type de données) et aux canaux de données (tailles des FIFOs).

UML	**SystemC**
TOP object	sc_main()
Compsite object (module)	module
Port	port
Data flow driven task object	SC_THREAD process
Control driven task object	Sub-module with two SC_METHOD processes: GetNextState and SetState
FSM states	Enum type
Input events	Sensitivity list
Input event	sc_in<type>
Output event	sc_out<type>
Control channel	sc_signal<bool>
Read action	fifo.read();
Write action	fifo.write()
External input data port	sc_fifo_in<T>
External output data port	sc_fifo_out<T>
Computation action	Function
Internal Data channel	sc_fifo<T>

Tableau 4.2. Correspondance entre UML et SystemC.

Le processus de transformation se fait suivant les règles de transformation indiquées dans le tableau 4.2. Nous implémentons les tâches dominées par les données comme des processus SC_THREAD. Ce choix est dû au fait que, dans le modèle de calcul KPN, une tâche représente une boucle infinie qui lit les données d'entrées, les traiter, puis écrire les résultats sur les canaux de sortie tout en accédant aux ressources (FIFO) bloquantes.

Les canaux de données sont implémentés comme des fifo SystemC. Les lectures et les écritures sur les canaux sont implémentées comme des méthodes de lecture et d'écriture bloquantes sur des *fifo* SystemC. Chaque action de calcul à forte granularité dans le diagramme d'activité est implémentée en tant que fonction SystemC. Chaque tâche de contrôle est implémentée comme un module SystemC composé de deux processus SC_METHOD : *GetNextState* qui calcule le nouvel état de la FSM en se basant sur l'état actuel et l'événement d'entrée et *SetState* qui copie l'état suivant dans l'état actuel à tout front montant (positif) d'horloge. La Figure 4.16 donne le fichier d'en-tête en SystemC de la tâche

de l'exemple de la figure 4.7 (b). Les événements d'entrée *ev1*, *ev2* et *ev3* sont déclarés en tant que ports d'entrée de type booléen et *ev5* en tant que port de sortie.

Nous rajoutons deux signaux pour mémoriser l'état courant et l'état suivant. Le processus *GetNextState* est sensible aux signaux *ev1*, *ev2*, *ev3*, et *current_state*. Le processus *SetState* est sensible au front montant de signal d'horloge. Figure 4.18 illustre l'implémentation du statechart. Nous constatons que, lorsque la FSM est à l'état *state2* et l'événement *ev3* se produit, FSM met à jour son port de sortie à vrai (c'est-à-dire l'envoi de ev5). Les figures 4.17 et 4.19 montrent les fichiers d'en-tête et d'implémentation de la tâche de l'exemple de la figure 4.7 (a). *task* est implémentée en tant que SC_THREAD avec un signal sensible (*ev5*). La tâche lit les données sous forme de jetons à partir du ch1 puis l'écrit sur le canal ch2. Dans cet exemple, nous supposons que la tâche est définie à l'intérieur d'un module appelé *module*. Les figures 4.20, 4.21 et 4.22 illustrent le code SystemC pour le fichier d'en-tête, le fichier d'implémentation de module1, et de module *TOP* respectivement.

```
// Satechart.h
# include  "systemc.h"
Enum states {
BEGIN, STATE0, STATE1, STATE2, END } ;
SC_MODULE (Satechart) {
sc_in<bool>  ev1 ; // input port
sc_in<bool>  ev2 ;
sc_in<bool>  ev3 ;
sc_out<bool>  ev5 ; // output port
sc_signal<states> next_state;  // signal
sc_signal<states> current_state;
void GetNextState();
void SetState();
SC_CTOR(Satechart) {
SC_METHOD(GetNextState);
sensitive << ev1<< ev2 << ev3 << current_state;
SC_METHOD(SetState);
sensitive_pos (clk) ;
}
};
```

Figure 4.16. Fichier d'entête de la machine à états finis.

```
// module1.h
# include "system.h"
SC_MODULE (module1) {
sc_out<bool>  evt5;
sc_fifo<int> ch1 ;
sc_fifo<int> ch2 ;
void ComputeDCT();
void ComputeINV();
void Task();
SC_CTOR(module) {
```

```
SC_THREAD(Task);
Sensitive << ev5;
```

Figure 4.17. Fichier d'entête pour la tâche dominée par les données.

```
// Satechart.cc
# include  " Satechart.h"
void Satechart::GetNextState() {
//
switch (current_state) {
case BEGIN:
next_state = STATE0;
break;
case STATE0:
if ( ev1 == true)
next_state = STATE1;
break;
case STATE1:
if (ev2 == true)
next_state = STATE2;
break;
case STATE2:
if  (ev3 == true) {
ev5 = true;
next_state =  END;
}
break;
}
}
void Satechart::SetState() {
current_state = next_state;
}
```

Figure 4.18. Implémentation de la machine à états finis.

```
// module.cc
# include "module.h"
void module::ComputeDCT()
{
// code for ComputeDCT
}
void module::ComputeINV()
{
// code for ComputeINV
}
void module::Task() {
while (true) {
sample1 = ch.read();
wait (ev5);
ComputeDCT();
ch2.write();
ComputeINV();
}
}
```

Figure 4.19. Implémentation de la tâche dominée par les données.

```
// module1.h
# include "system.h"
# include "Controller1.h"
SC_MODULE (module1) {
sc_in<bool> ok2;
sc_out<bool>  request2;
sc_fifo<int> ch1 ;
sc_fifo_out<int> ch2 ;
Controller1 *C1;
sc_signal<bool> ch3;
sc_signal<bool> ch4;
sc_signal<bool> ch6;
void A();
void B();
void ComputeA1();
void ComputeA2();
void ComputeB1();
void ComputeB2();
SC_CTOR(module1) {
C1 = new Controller1("C1");
C1-> ok2(ch6);
C1-> request1(ch3);
C1-> request2(ch4);
SC_THREAD(A);
sensitive << request1;
SC_THREAD(B);
}
}
```

Figure 4.20. Fichier d'entête du module1

```
// module1.cc
# include "module1.h"
void module1::ComputeA1()
{
// code for ComputeA1()
}
void module1::ComputeA2()
{
// code for ComputeA2
}
void module1::ComputeB1()
{
// code for ComputeB1()
}
void module1::ComputeB2()
{
// code for ComputeB2()
}
void module1::A() {
while (true) {
ComputeA1();
```

```
ch1.write(sample);
wait (request1);
ComputeA2();
}
}
void module1::B() {
while (true) {
ComputeB1();
sample1 = ch1.read();
ComputeB2();
ch2.write(sample2);
}
}
void Controller1::GetNextState1() {
//
switch (current_state1) {
case BEGIN:
next_state1 = MODE1;
break;
case MODE1:
request1 = true;
request2 = true;
next_state1= MODE1;
break;
case MODE1:
if (ok2 )
next_state1 = MODE2;
break;
case MODE2:
next_state1 =  END;
break;
}
}
void Controller1::SetState1() {
current_state1 = next_state1;
}
```

Figure 4.21. Fichier d'implémentation du module1

```
// main.cc
# include "module1.h"
# include "module2.h"
int sc_main(int argc, char* argv[]) {
sc_signal<bool> OK2;
sc_signal<bool> REQUEST2;
sc_fifo<int> CH2(10) ;
module1 M1("module1");
M1.ok2(OK2);
M1.request2(REQUEST2);
M1.ch2(CH2);
module2 M2("module2");
M2.ok2(OK2);
```

```
M2.request2(REQUEST2);
M2.ch2(CH2);
return(0);
}
```

Figure 4.22. Implémentation du module *TOP*

10. Conclusion

Dans ce chapitre, nous avons présenté une technique de modélisation et de vérification formelle pour les systèmes embarqués mixtes. Nous avons identifie deux types des tâches et des canaux : dominées par contrôle et données. Les tâches dominées par contrôle sont modélisées par le bais des statecharts avec zéro délai. Les tâches dominées par les données sont modélisées à l'aide des diagrammes d'activité avec opérations à forte granularité. A partir de ce modèle, une spécification Maude et un code SystemC sont générés automatiquement. Nous avons utilisé le code Maude pour vérifier formellement quelques propriétés du système et estimer la consommation d'énergie à un niveau élevé d'abstraction. En revanche, le code SystemC est utilisé pour faire des simulations.

L'une des particularités de notre approche est l'abstraction de données. La technique d'abstraction de données nous a permet de faire des vérifications et des simulations plus rapides.

Cinquième chapitre

Intégration d'UML dans un flot de co-conception visant une architecture reconfigurable

Sommaire

1. Introduction

Dans ce chapitre, nous présentons une méthodologie intégrant UML et les outils de synthèse de haut niveau et de Co-simulation ciblant les plateformes reconfigurables.
Notre flot essaye de tirer profit du standard UML et des outils commerciaux de synthèse de haut niveau et de co-simulation visant les architectures reconfigurables et en particulier les plateformes Xilinx.

Une particularité de notre flot est le fait qu'il prenne en compte la modélisation de haut niveau de l'architecture incluant les pilotes pour la partie logicielle et les adaptateurs (wrappers) pour la partie matérielle. Ce chapitre est organisé comme suit : dans un premier temps, nous mettons la perspective sur les travaux voisins, ensuite nous discutons en détail notre flot. Une étude de cas est présentée. Il s'agit du standard H264 pour compression/décompression vidéo. Dans cette étude de cas, nous avons transformé la fonction de transformation inverse au matériel en utilisant les outils CatapultC et XPS. Toutes les autres fonctions sont implémentées en logiciel.

2. Travaux voisins

Selon le type d'entrée de flot de conception, nous pouvons classer les flots de Co-conception existants ciblant les architectures reconfigurables en trois grandes catégories: les langages de programmation classiques, UML, et UML avec SystemC.
L'utilisation des langages de programmation est un moyen de capturer la spécification du système, qui a l'avantage d'être exécutable. Afin de surmonter certains des inconvénients de l'utilisation de langages de programmation classiques pour la spécification au niveau système, UML est utilisé comme entrée de flot. En outre, certaines approches utilisent UML avec SystemC pour la spécification exécutable de SOC. Par la suite, nous allons étudier les travaux existants en se basant sur la catégorie d'entrée de flot de conception.
Les langages de programmation classiques les plus utilisés dans les spécifications des SOC sont C / C + +, et Java. Plusieurs approches utilisant le langage C pour des coprocesseurs reconfigurables ont été proposées [30, 80, 104]. La compilation des applications C à un processeur et co-processeur reconfigurable est proposée par Callahan et al. [30]. Leur partitionnement matériel / logiciel se fait au niveau bloc de base. Nimble [93] est un cadre de conception au niveau système qui compile automatiquement les applications C en un code exécutable pour architecture embarquée reconfigurable. Dans ce cadre, l'algorithme de

partitionnement matériel/logiciel effectue un partitionnement à granularité fine (au niveau boucles et blocs de base) de l'application.

Le flot de conception basé Java pour la conception des systèmes réseau reconfigurables est d'abord proposé par Fleischmann et al. [52]. L'environnement de Co-conception appelé JACoP contient un gestionnaire dynamique pour l'ordonnancement de méthodes, soit sur la machine virtuelle de Java (JVM) ou sur le matériel reconfigurable.

SystemC est récemment utilisé comme un langage de spécification au niveau système. Pelkonen et al. [51] ont proposé une méthodologie de modélisation au niveau système du matériel dynamiquement reconfigurable en utilisant SystemC. Cette méthode permet aux utilisateurs d'explorer l'espace de conception au niveau système, sans avoir besoin d'associer le design à une implémentation technologique réelle.

Toutefois, cette méthode n'est pas complète et elle requiert une validation de résultats pour des investigations ultérieures. L'utilisation d'UML comme entrée de la Co-conception de systèmes embarqués est encore assez nouvelle. Dans [10, 11] Beierlein et al. ont présenté un environnement de Co-conception basé UML pour les architectures dynamiquement reconfigurables, appelé compilateur de modèle pour architectures configurables (MOCCA). Ils utilisent le langage de modélisation UML dans toutes les phases de développement, de la spécification à la synthèse. Le concept de Co-conception, Architecture dirigée par les modèles (MDA), et conception basée plate-forme sont utilisés dans l'approche de développement proposée. Ils ont étendu UML par un langage d'action afin de permettre l'exécution, la vérification et la simulation, l'exploration d'espace de conception, et l'automatisation de la synthèse.

L'adoption d'UML, avec SystemC dans le processus de conception de SOC est présentée par Zhu et al. [135]. Le processus de conception présenté est appelé SLOOP (System Level design with Object-Oriented Process) qui permet l'évaluation des performances au niveau système. Ils ont également étendu UML en utilisant le mécanisme de stéréotype. Toutefois, les outils qui sont nécessaires dans SLOOP sont encore au cours de développement. Un translateur UML vers SystemC pour la conception de SOC est proposé par Nguyen et al. [105]. L'utilisation de diagrammes de séquence étendus pour spécifier les jeux de test et leur traduction en SystemC est l'un de ses objectifs futurs.

Contrairement à ces approches, nous proposons un partitionnement matériel / logiciel au niveau méthode en s'appuyant sur les diagrammes de séquence et d'objets d'UML.

3. Flot proposé

Nous avons développé un flot de conception qui intègre UML et une collection d'outils de simulation et de synthèse ciblant des architectures reconfigurables. La figure 5.1 montre la vue détaillée de notre méthodologie, qui est séparée en quatre phases: la spécification, l'exploration, la génération et l'intégration.

Notre flot commence par l'établissement de modèles UML structurels et fonctionnels en utilisant l'environnement Rhapsody7.2.

Le modèle structurel d'application est présenté à travers le diagramme d'objets. Le modèle fonctionnel est présenté au moyen des diagrammes de séquence. Le code de méthodes des classes est implémenté dans le langage de programmation C.

Parallèlement au modèle d'application, l'architecture matérielle est également modélisée avec le diagramme de structure d'UML avec un ensemble défini de stéréotypes et de méthodes qui caractérisent les composants matériels et la topologie de l'architecture cible.

Dans notre cas, l'architecture cible est une plate-forme reconfigurable (Xilinx) avec des cœurs de processeur (PowerPC, Microblaze), HW programmable (Virtex4), périphériques, interfaces HW / SW (FSL, APU, OPB, PLB), cœurs IP, et un support de système d'exploitation (OS) pour accélérateurs matériels reconfigurables (linux embarqué).

Afin d'être capable de générer des adaptateurs HW (wrappers) pour accélérateurs matériels et des pilotes SW pour méthodes logiciels de manière automatique, nous avons inclut des modèles des adaptateurs et des pilotes dans l'architecture.

Après la génération de l'exécutable et la vérification du comportement fonctionnel de l'application, l'étape suivante consiste à l'exploration SW / HW. Dans cette étape, le concepteur profile le code exécutable de l'application et analyse la charge de communication entre les méthodes. Selon les résultats de profilage et de la charge de communication, le concepteur identifie les méthodes qui seront implémentées en matériel et choisit le système de communication le plus adéquat (point à point ou mémoire partagée).

Le partitionnement HW / SW se fait au niveau méthode dans le diagramme de séquence et/ou diagramme d'objets par définition, d'un nouveau stéréotype appelé "HW" avec un seul paramètre qui spécifie le nom d'IP. À ce stade, le concepteur peut estimer les performances du système en tenant compte de services de qualité offerts par les composants de la plateforme.

Les méthodes HW et leurs adaptateurs (wrappers) servent d'entrée pour l'outil commercial de synthèse de haut niveau CatapultC pour la génération automatique de code VHDL à partir de

code C [34]. En utilisant un tel outil, le concepteur peut optimiser sa conception en appliquant certaines techniques d'optimisation de synthèse de haut niveau telles que le déroulement de boucles etc. et de générer le schéma RTL du design. Il peut aussi faire des estimations plus exactes sur le temps d'exécution, la surface et la consommation d'énergie.

Les adaptateurs HW sont générés automatiquement en code C à partir de la deuxième étape en utilisant le nom d'une macro *wrapper* écrite dans l'interpréteur VB qui est intégré dans Rhapsody. D'un autre côté, les méthodes logicielles sont adaptées par des méthodes d'insertion des appels à leurs pilotes qui sont générés automatiquement à partir de la deuxième étape. Le processus de génération de pilotes se fait par l'intermédiaire d'une macro nommée *driver* écrite dans l'interpréteur VB. Le résultat de cette étape est une plate-forme qui dépend du code C pour les méthodes SW et un code VHDL pour les méthodes HW.

La dernière étape consiste en l'intégration de codes C / VHDL en utilisant l'outil XPS.

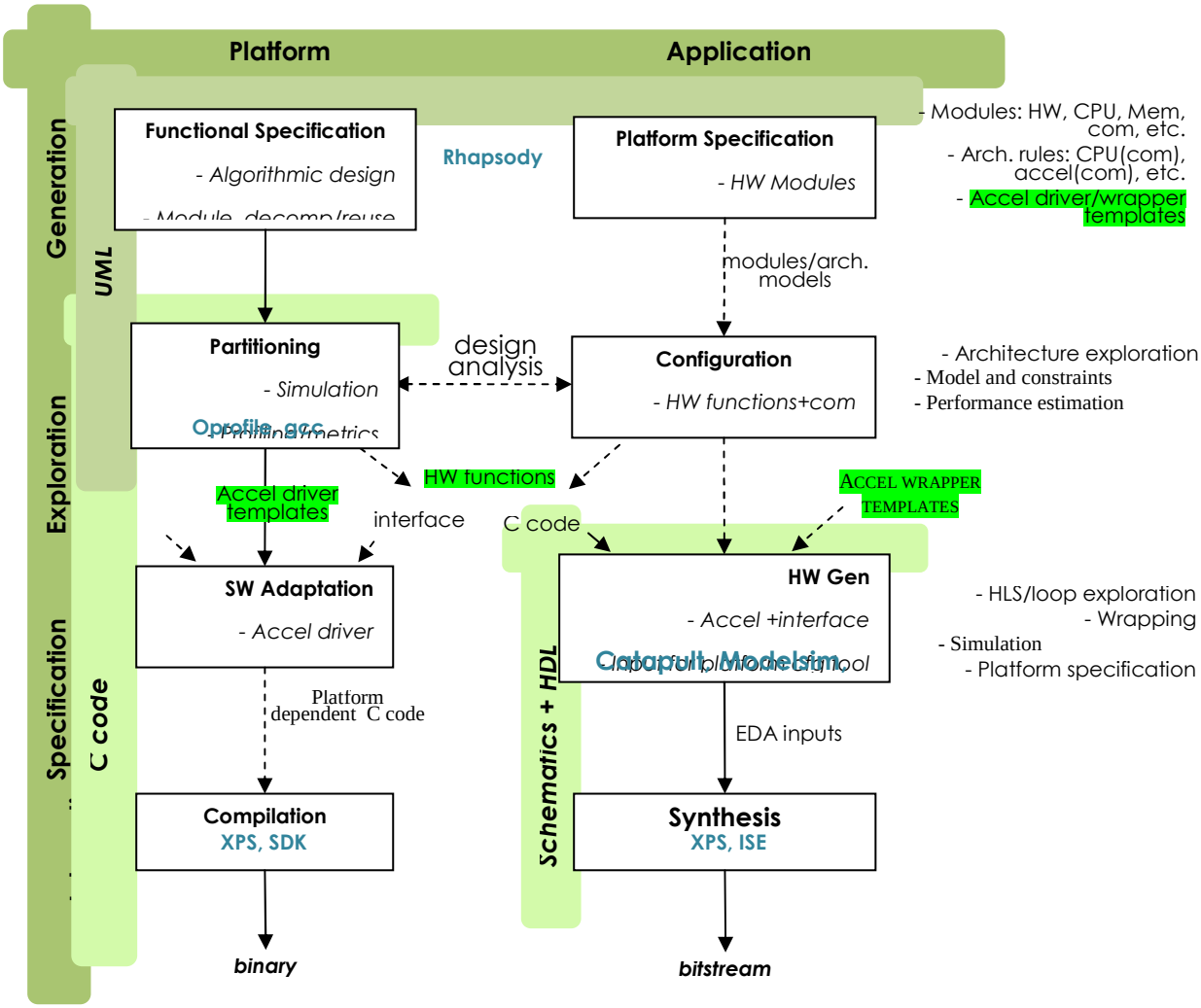


Figure 5.1. Flot proposé.

Les modèles d’application s’appuient sur deux diagrammes UML: diagramme d’objets pour la modélisation de l'architecture du logiciel, et le diagramme de séquence pour la modélisation du comportement collaboratif de l’application. Nous adoptons Rhapsody2 [108]. Cet outil peut générer du code C, C + +, Ada ou Java à partir des diagrammes UML.

Rhapsody en C vise le développement de logiciels embarqués et permet de créer un modèle indépendant des plateformes. Il porte sur la génération automatique de code C à partir de diagrammes UML, y compris la configuration de l'environnement et de génération de *makefile*. Cette approche permet de mettre en place un compilateur multi cibles et un éditeur de lien pour la plate-forme cible. Les modèles visuels peuvent être exécutés et animés. Il fournit également un bon support pour un processus de développement itératif et souple.

Rhapsody permet la réutilisation du code en l'enveloppant dans des bibliothèques liées au modèle. Il est possible de configurer la manière dont le code Rhapsody est généré en changeant les propriétés de projet, du paquetage ou de classe. Dans certains cas, cela pourrait être nécessaire pour optimiser la taille du code.

Nous avons choisi le diagramme de séquence d'abord parce qu'il supporte la hiérarchie, la modélisation de l'exécution parallèle, séquentielle et le contrôle. Il permet également au concepteur d’analyser les dépendances entre les méthodes et la charge de communication.

Le diagramme de séquence est utilisé aussi pour le partitionnement HW / SW visuel par définition d'un stéréotype appelé "HW" afin de déterminer les méthodes qui seront mises en oeuvre dans le matériel. Les comportements internes des méthodes sont introduits directement en C.

L'objectif de la modélisation de la plate-forme cible à ce niveau d'abstraction est de fournir aux concepteurs HW / SW un modèle visuel de l'architecture matérielle qui sera utilisé lors de la phase finale du flot. Un tel modèle visuel va aider les concepteurs à comprendre les compromis qui existe entre l'application et l'architecture matérielle, de faire le partitionnement HW / SW au niveau UML, et la génération automatique des adaptateurs (wrappers) pour accélérateurs matériels et les pilotes logiciels.

Notre plate-forme cible est une plate-forme de type Xilinx incluant des processeurs Microblaze, PowerPC et FPGA Virtex4 / 5 [132].

Microblaze est un processeur soft RISC standard de 32-bit ; style Harvard, qui est spécialement conçu pour les architecture FPGA basées Virtex et Spartan-II/3. Ses 32 registres (de 32 bits) sont des tables lookup (LUT) à base de RAM. Il garantit un très court temps d'accès au registre. Pour la mémoire, soit un bloc RAM sur puce (on-chip) ou mémoire hors puce (off-chip) peuvent être utilisés. Le temps d'accès au bloc RAM sur puce est minime car

il y a des ressources de routage dédiées pour y accéder. En raison du fait que Microblaze utilise les ressources FPGA disponibles de façon très efficace, il est possible que son horloge puisse atteindre jusqu'à 150 MHz.

Microblaze est utilisé dans différents domaines, tels que les applications de réseau, applications de télécommunication, de contrôle et de marché de consommation.

En général, il existe deux façons d'intégrer un cœur IP personnalisé dans un système embarqué à base de processeur Microblaze. Une première solution est de connecter l'IP sur l'OPB (On-chip Peripheral Bus). L'OPB fait partie de bus sur puce standard d'IBM *Core ConnectTM*. La deuxième manière est de connecter l'IP de l'utilisateur au bus dédié du Microblaze ; c'est le Fast Simplex Link (FSL).

Microblaze contient huit interfaces FSL de type entrée et huit sorties. Les canaux FLS sont des interfaces de données unidirectionnelles point à point dédiés. Les interfaces FSL du Microblaze ont une taille de 32 bits. En outre, les mêmes canaux peuvent être utilisés pour transmettre ou recevoir des mots de données ou de contrôle. Un bit séparé indique si le mot transmis (reçu) est une information de contrôle ou de données.

Les performances de l'interface FSL peut atteindre jusqu'à 300 Mo / sec. Ce débit dépend de dispositif cible lui-même. Le système de bus FSL est idéal pour des communications Microblaze à Microblaze ou les entrée/sortie de flux de données.

Les principales caractéristiques de l'interface de FLS sont:

- Communication unidirectionnelle point à point.
- Mécanisme de communication non partagé et non soumis à l'arbitrage.
- Support pour communication de données et de contrôle.
- Communication basée FIFO.
- Taille des données configurables.
- 600 MHz en mode autonome (standalone operation).
- Le bus FSL est piloté par un maître et pilote un esclave.

Si l'application est critique en temps, l'IP utilisateur doit être connecté au bus système FSL, sinon, il peut être connecté en tant que maître ou esclave sur l'OPB. Si l'IP est connecté à FSL, il est alors possible d'utiliser des fonctions prédéfinies C afin d'utiliser l'IP utilisateur dans l'application logiciel. Pour chaque type des composants matériels, nous définissons un ensemble de stéréotypes.

Nous avons inclut dans notre modèle d'architecture des modèles (templates) pour les pilotes logiciels et les adaptateurs (wrappers) matériel. Un template nommé *wrapper* possède une méthode qui lit la signature de la méthode HW et effectue les étapes suivantes:

1. Créer les registres de la taille appropriée pour les paramètres d'entrée et la valeur de retour.
2. Ajouter un registre de contrôle de 1 bit (GO) - ce qui signifie que les entrées sont en place et le traitement peut commencer.
3. Ajouter un registre d'état de 1 bit (DONE), qui indique que la valeur de retour est prête pour la lecture par l'application hôte.

Le code de l'interface est écrit en C et ajouté automatiquement au code de la méthode HW. De manière similaire, un template nommé *driver possède* une méthode qui cherche les méthodes HW, puis il effectuera les étapes suivantes:

1. Ajouter le symbole de commentaire // avant l'appel de méthode logiciel pour le désactiver.
2. Ajouter une instruction pour charger la méthode HW requise en FPGA.
3. Ajouter les instructions nécessaires du protocole handshake pour passer les valeurs d'entrée au matériel et récupérer le résultat.

4. Etude de cas

Afin de valider notre méthodologie, Nous avons choisi le décodeur vidéo H264 comme étant une étude de cas. Il s'agit d'une application de traitement de signal intensif. Son diagramme de blocs fonctionnels est illustré en troisième chapitre (figure 3.11).

Dans un premier temps, nous avons modélisé cette application en utilisant l'outil Rhapsody. Deux diagrammes UML sont utilisés : le diagramme d'objets pour modéliser la structure du décodeur et les diagrammes de séquences pour modéliser la collaboration des objets, ainsi que le partitionnement HW/SW. Les méthodes des classes sont implémentées directement en langage C. Pour obtenir un code qui est proche du code original. Nous avons personnalisé le processus de génération de code en éliminant le code non nécessaire (comme l'allocation dynamique des objets). La figure 5.2 illustre le diagramme d'objets du décodeur H264. Son diagramme de séquence principal est présenté dans la figure 3.12 du troisième chapitre. Les figures 5.3, 5.4, et 5.5 montrent la classe '*coretrans*' ayant la fonction de transformation inverse (enter) stéréotypée par 'HW', le modèle UML du processeur Microblaze, et le modèle UML d'un IP respectivement.

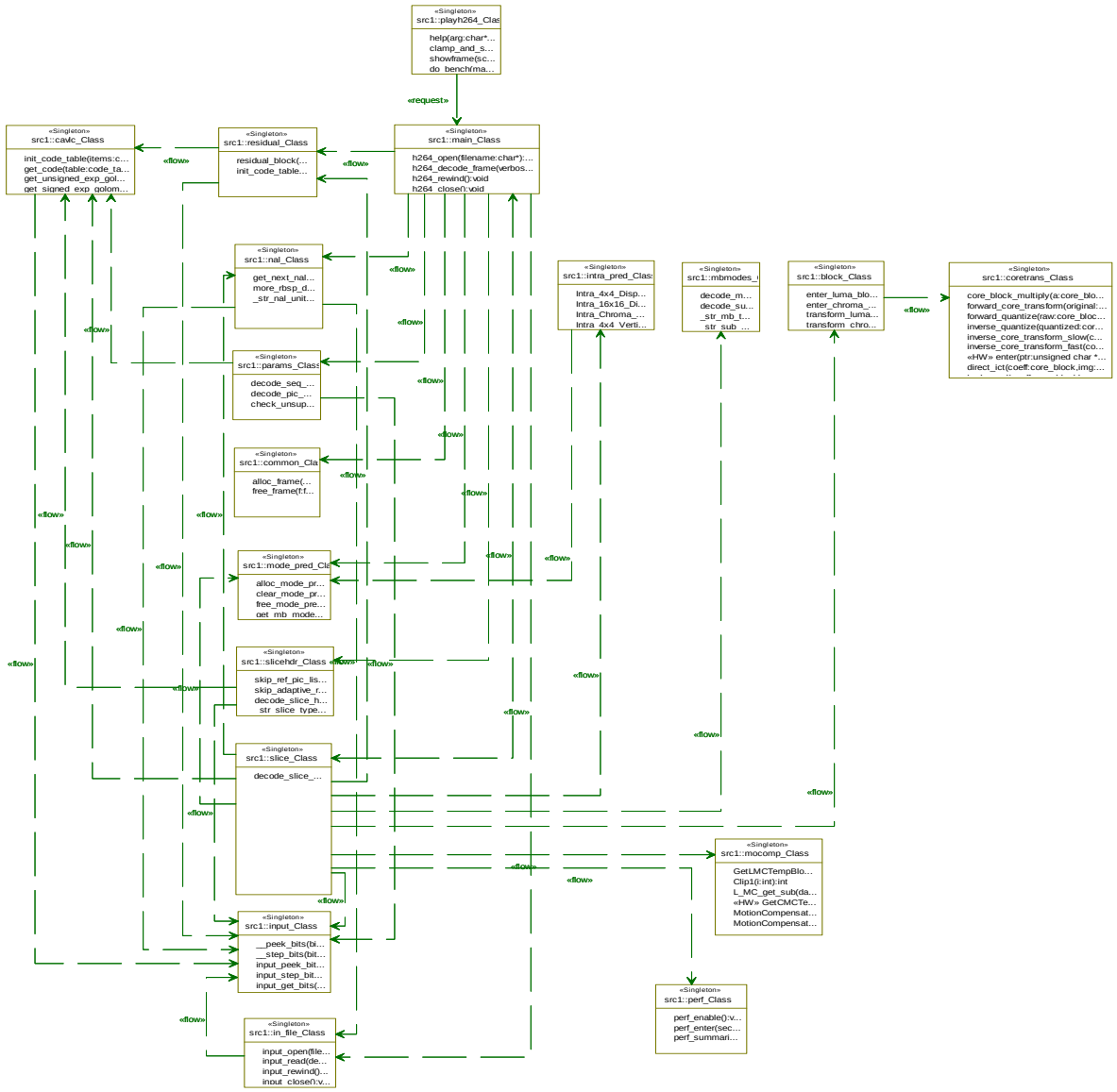


Figure 5.2. Diagramme d'objets de décodeur H264.

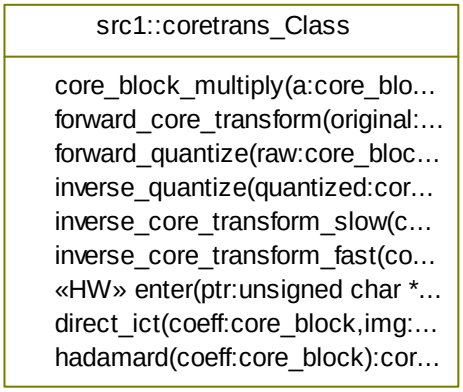


Figure 5.3. La fonction de transformation inverse (enter) est stéréotypée par "HW".

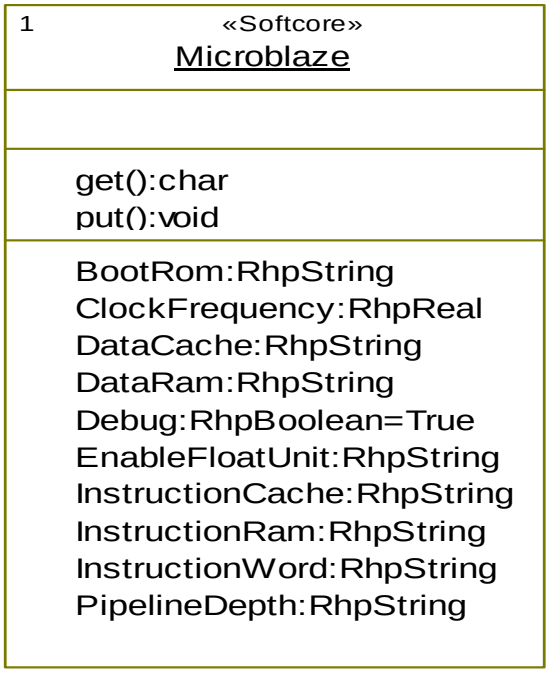


Figure 5.4. Modélisation UML du MicroBlaze.

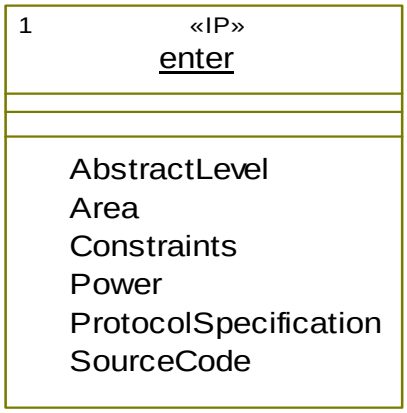


Figure 5.5. Modélisation UML d'un IP.

Le tableau 5.1 donne les résultats du profilage pour les différentes fonctions du H264.

%	Nom du symbole	Fonctionnalité
25.4309	GetLMCTempBlock	Motion compensation
21.0332	L_MC_get_sub	Motion compensation
10.7746	Clip1	Motion compensation
9.6356	MotionCompensateTB	Motion compensation
6.9927	GetCMCTempBlock	Motion compensation
5.1527	inverse_quantize	Inverse quantization
5.1203	direct_ict	Inverse transform
4.2957	Enter	Inverse transform
3.3019	coeff_scan	Inverse Scan
1.1246	get_code	CAVLC
0.8510	decode_slice_data	--
0.6445	enter_luma_block	--
0.6193	residual_block	--
0.5533	FillMVs	Mode prediction
0.5017	MotionCompensateMB	Motion compensation
0.3613	input_get_one_bit	--

Tableau 5.1. Résultat du profilage de différentes fonctions de H264.

4.1. Génération du code VHDL

Cette section se concentre sur la fonction de transformation inverse et sa traduction au matériel en utilisant les outils de HLS avec la logique nécessaire à l'interfaçage du Microblaze. Tout d'abord, on génère un code VHDL RTL à partir du code C pur en utilisant l'outil Mentor CatapultC.

L'utilisation d'un tel outil aide largement le problème de génération de code VHDL, mais l'analyse manuelle est susceptible d'être inévitable afin d'aborder efficacement le problème délicat des interfaces.

Dans notre cas, la fonction transformation inverse manipule des petits blocs 4x4 homogènes, alors une communication point à point semble une solution plus adéquate. Dans ce cas particulier, le traitement en lui-même est rapide (quelques cycles) par rapport aux transferts de données (16 données * 2).

Nous avons défini un modèle d'enveloppe VHDL, qui étend les modèles FSL originaux fournis par les outils Xilinx, et en mesure d'examiner le temps de traitement de l'accélérateur avec des contraintes sur les transferts de données. Nous avons aussi développé des modèles

d’enveloppes des fonctions pour l'accélérateur afin d’être appelées à partir du code de l'application. Les deux modèles ont été développés avec le souci d'être applicable dans le cas général, certains cas peuvent nécessiter plus ou moins d’adaptation ; ça dépend des besoins de données de communication.

Dans notre étude de cas (figure 5.6), le code VHDL RTL est généré par un outil HLS (CatapultC) avec des interfaces simples: 16 ports block_in, 16 ports block_out plus un signal d’horloge *clk* et un signal reset *rst*. Ces signaux sont ensuite enveloppés pour être compatibles avec l’interface FSL et qui rendent possible la connexion d'un coprocesseur au processeur Microblaze.

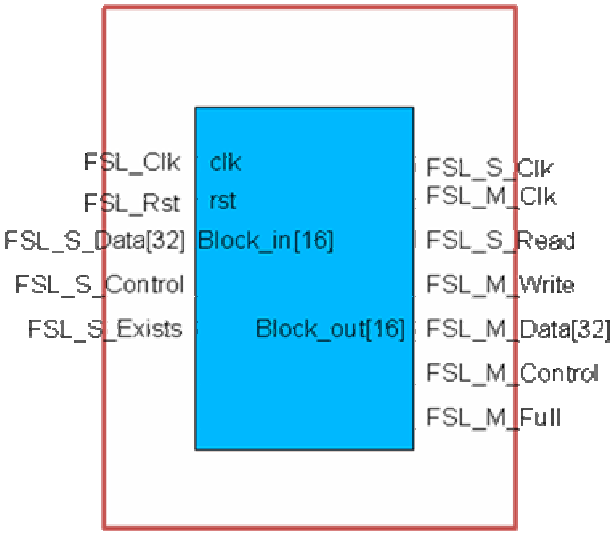


Figure 5.6. Résultat de synthèse de la fonction transformation inverse.

4.2. Exploration de l’espace de conception

Nous exposons ici l'exploration de l’espace de conception (DSE) des résultats sur la fonction de transformation inverse en utilisant cette méthodologie. Le processus du DSE est fondé sur la capacité de l'outil HLS à générer des solutions à différents compromis surface / performance (sans optimisation, le déroulement des boucles et le pipelining des boucles).

Dans la pratique, il suffit de créer une nouvelle RTL pour la fonction, remplacer l'ancien code VHDL et régénérer le système. Le reste du code sera maintenu ce que simplifie extrêmement l'évaluation des alternatives du parallélisme des fonctions HW.

Une évaluation réaliste de l'accélération doit tenir compte de temps de transferts de données vers / depuis le coprocesseur. Dans cette évaluation, nous avons comparé le code C de la transformée inverse s’exécutant sur le PowerPC, avec un code C en appel d'un coprocesseur

avec exactement les mêmes ensembles de données. Le résultat est une accélération de 2,41, alors que sur un simple point de vue de traitement, le traitement est xx fois plus vite. Cela est dû à la communication de données qui sont séquentielles avec FSL.

5. Conclusion

Dans ce chapitre, nous avons présenté une méthodologie intégrant UML et les outils CAO de synthèse de haut niveau et de co-simulation ciblant une architecture reconfigurable. Les outils que nous avons utilisé sont Rhapsody pour la modélisation UML, Catapult C pour transformer le code C en VHDL et l'outil XPS se Xilinx pour la synthèse et la co-simulation.

L'application et la plateforme matérielle sont modélisées au niveau UML en s'appuyant sur les diagrammes de classes et d'architecture d'UML2.

Le partitionnement se fait au niveau méthode de classe en utilisant le diagramme de séquence ou d'objets. Pour ce faire, nous avons défini un nouveau stéréotype nommé "HW" désignant le nom d'IP implémentant la fonction matérielle.

L'intégration d'UML avec les outils de synthèse de haut niveau, apporte plusieurs avantages. Premièrement elle aide les concepteurs qui ne sont pas spécialistes en matériel d'utiliser les outils du CAO ciblant les architectures réconfigurables efficacement. Deuxièmement, l'utilisation d'UML accroît la réutilisation du code et la productivité.

Sixième chapitre

Définition d’une sémantique opérationnelle pour l’ordonnanceur SystemC

Sommaire

1. Introduction

SystemC est une bibliothèque de classes C++ et une méthodologie qui permet aux concepteurs d'exploiter les outils de développement du standard C++ pour créer un modèle du SOC au niveau système, le simuler rapidement afin de valider et d'optimiser le design, explorer différents algorithmes, et fournir aux équipes de développement matériel et logiciel une spécification exécutable du système [70].

SystemC supporte des modèles à différents niveaux d'abstraction, allant de modèles fonctionnels de haut niveau, aux modèles RTL (Register Transfer Level) et le raffinement itératif de modèles de haut niveau en modèles de plus bas niveaux d'abstraction. La bibliothèque de classes du SystemC fournit les données nécessaires pour construire le modèle d'architecture du système, y compris le temps, la concurrence, et le comportement réactif; des aspects qui sont absents du standard C ++. Différentes versions du langage ont fait leur apparition, mais nous considérons uniquement SystemC2.0.

En dépit d'être utilisé pour analyser et vérifier les SOCs (dû à son moteur de simulation à événements discrets), SystemC manque une sémantique formelle permettant une analyse formelle.

Dans ce chapitre, nous présentons notre approche de formalisation de l'ordonnanceur SystemC en utilisant le langage Maude. Notre idée consiste donc à transformer la dynamique d'un programme SystemC en un ensemble de règles de réécriture exprimées dans le langage Maude. Ce passage nous permet d'exploiter les capacités du langage Maude entre autre la vérification de modèles (model cheker).

2. SystemC: aspects structurels et dynamiques

L'aspect structurel du SystemC est basé sur le concept du module. Un module SystemC contient des ports, des interfaces, des canaux, des processus, et éventuellement d'autres modules. En d'autres termes, les modules permettent l'expression de l'hiérarchie (voir les figures 6.1 et 6.2). Un port est un objet à travers lequel un module peut accéder à l'interface du canal. Mais les modules peuvent également accéder à l'interface du canal directement. Un port qui est connecté à un canal par le biais d'une interface ne voit que les méthodes de ce canal qui sont définies par l'interface. Une interface fournit un ensemble de déclarations de méthode, mais ne fournit aucune implémentation de méthode. Les interfaces sont utilisées pour définir l'ensemble des méthodes que les canaux doivent implémenter. Un canal

implémente une ou plusieurs interfaces, et sert de conteneur pour la communication et la synchronisation.

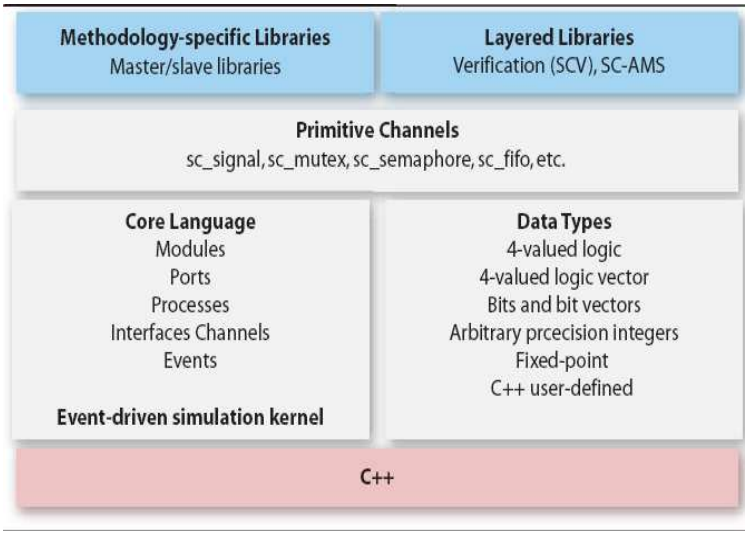


Figure 6.1. Infrastructure du SystemC [55].

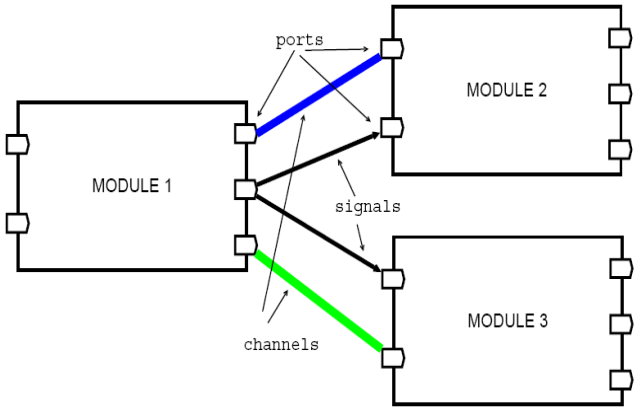


Figure 6.2. Aspects structurels du SystemC [55].

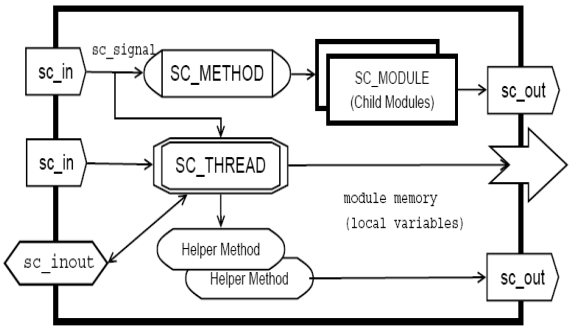


Figure 6.3. Un Module SystemC [55].

On note que le canal n'est pas nécessairement une connexion point à point ; il peut connecter plus de deux modules. Un canal peut être primitif ou hiérarchique. Un canal hiérarchique est un module, qui peut contenir des processus et des autres modules, et il peut accéder directement aux autres canaux.

En SystemC, les comportements concurrents sont modélisés en utilisant les processus. Un processus peut être vu comme un fil de contrôle, qui reprend l'exécution quand certains événements se produisent ou que certains signaux se changent, puis suspend l'exécution après l'exécution de certaines actions. En SystemC, un processus ne peut être sensible qu'aux changements de valeurs des signaux particuliers, et l'ensemble des signaux pour lesquels le processus est sensible (la liste de sensibilité), doit être pré-spécifié avant le début de simulation. Deux types de sensitivités sont pris en compte dans SystemC: la sensitivité statique où l'ensemble des signaux est fixé et la sensitivité dynamique où l'ensemble de signaux peut varier dans le temps sous le contrôle du processus lui-même.

Les processus SystemC s'exécutent simultanément et peuvent suspendre et reprendre l'exécution à des points spécifiés par l'utilisateur. Ces processus qui requièrent leur propre pile d'exécution sont appelés "SC_THREADs". Quand le seul signal qui déclenche un processus est le signal d'horloge "clk", nous obtenons ce que nous appelons SC_CTHREAD (clocked thread). Les processus qui n'ont pas vraiment besoin d'une pile d'exécution indépendante sont appelés "SC_METHODs".

Un processus SC_METHOD s'exécute en un temps de simulation égal à zéro et retourne le contrôle au noyau de simulation et ne peut pas être suspendue avec l'instruction *wait*. C'est le seul type de processus qui peut être utilisé pour la synthèse RTL. Un SC_THREAD est plus lent qu'un SC_METHOD dans la simulation et utilisé pour la modélisation des "test benches".

En SystemC, l'utilisation de *notify()* avec un argument de temps égal à zéro provoque l'événement dans la phase d'évaluation de prochain delta cycle; *notify()* avec un argument de temps cause l'événement à être notifié dans un temps spécifié dans le futur; *notify()* sans arguments cause les processus qui sont sensibles à l'événement à être immédiatement prêts pour l'exécution (c'est-à-dire prêt à s'exécuter dans la phase d'évaluation actuelle). Ce dernier mécanisme est utile pour la modélisation de systèmes logiciels et des systèmes d'exploitation qui manquent la notion de delta cycles. Un appel à *request_update ()* cause la méthode *update ()* à être appelée dans la phase de mise à jour de delta cycle actuel.

Plusieurs notifications pour un même événement, sont réglées conformément à la règle suivante: *temporisé* < *delta* < *immédiat*

Une notification immédiate est toujours prise en compte avant un delta-cycle qui est à son tour se traite avant une notification temporisée.

L'ordonnanceur SystemC contrôle la date et l'ordre d'exécution du processus, gère les notifications d'événements et gère les mises à jour des canaux. Il supporte la notion de delta-cycles. Un delta-cycle se compose de l'exécution de phase d'évaluation et de phase de mise à jour. Il y peut être un nombre variable de cycles delta pour chaque temps de simulation. Les processus SystemC sont non-préemptifs. Cela signifie que les processus s'exécutent sans interruption. Une fois démarre, l'ordonnanceur continu son exécution jusqu'à ce qu'il n'y ait pas plus d'événements, ou un processus arrête l'exécution explicitement, ou une exception se produit. La fonction *sc_main()* joue le même rôle que celui de la fonction *main()* du C++. La phase d'élaboration est définie comme l'exécution de la fonction *sc_main()* à partir du début de *sc_main()* à la première invocation de l'ordonnanceur. Au cours de l'élaboration, les éléments structurels du système sont créés et reliés suivant l'hiérarchie du système. La structure du système est créée lors du temps d'élaboration et ne change pas pendant la simulation. Avant la première invocation de l'ordonnanceur, l'initialisation est la première étape de la simulation. Chaque processus est exécuté (on peut désactiver l'initialisation d'un processus avec des appels à méthode *dont_initialize()*) lors de l'initialisation. L'ordre d'exécution des processus est indéterminé, mais l'exécution de deux simulations en utilisant la même version du même simulateur doit produire des résultats identiques. La figure 6.4 présente un exemple d'un programme SystemC.

En SystemC, le Non-déterminisme est introduit parce que l'ordre d'exécution de processus au sein d'une phase de simulation spécifique est indéterminé. Dans un système bien conçu, cet aspect de SystemC n'aura pas d'incidence sur le comportement global du système. Mais dans un système mal conçu, il est possible que cela pourrait entraîner des effets indésirables: le système peut se comporter de manière imprévisible d'une simulation à une autre ou pire encore, une implémentation du système dérivée de la spécification peut ne pas fonctionner correctement [55].

```
# include “systemc . h”
SC_MODULE(my_module ) {
sc _in<bool > port 1;
sc_out<bool > port 2;
sc_event e2, e3; / / events declaration
sc_signal<int > count ; / / intern channel
void proc1 ( ) {
if ( count. read ( ) < 10 ) {
count.write (count.read ( ) + 1);
e2.notify ( ); / / immediate notification
} else {
e3.notify (5 , SC_NS ); / / timed notification
}
}
void proc2 ( ) {
if ( count.read ( ) < 11 ) {
count.write (count . read ( ) + 2 ) ;
} else {
e3.notify (4 , SC_NS ); / / timed notification
}
}
void proc3 ( ) { count.write ( 0 ) ; }
}
SC_CTOR( my_module ) {
count.write (0) ;
SC_METHOD( proc1 ) ; sensitive << count ;
SC_METHOD( proc2 ) ; sensitive << e2 ;
dont_initialize ( );
SC_METHOD( proc3 ) ; sensitive << e3 ;
dont_initialize ( );
}
};
```

Figure 6.4. Un programme SystemC [32].

L'algorithme complet de l'ordonnanceur est le suivant:

1) Phase d'initialisation : Exécuter tous les processus (sauf les SC_CTHREAD) dans un ordre quelconque.
2) Phase d'évaluation : Sélectionner un processus qui est prêt à l'exécution et reprendre son exécution. Cela peut conduire à l'occurrence des notifications d'événements immédiates ce qui résulte en l'exécution de processus supplémentaires dans cette même phase.
3) S'il y a encore des processus qui sont prêt à l'exécution, aller à l'étape 2.
4) Phase de la mise à jour : exécuter les appels en attente de mise à jour résultant des appels *request_update*() de l'étape 2.
5) S'il y a des notifications retardées (*delayed*) en attente, déterminer les processus qui sont prêts à l'exécution en raison des notifications retardées et passer à l'étape 2.
6) S'il n'y a plus des notifications temporisées, la simulation est terminée.
7) Avancer le temps de simulation courant à la première notification temporisée en attente.
8) Déterminer les processus qui sont prêts à l'exécution en raison des événements qui ont des notifications en attente au temps courant. Aller à l'étape 2.

3. Travaux voisins

Bien que, les designs SystemC soient exécutables, on peut affirmer qu'il existe une difficulté à les analyser. Cela est dû au fait que la sémantique du SystemC est définie de façon informelle, ce qui rend SystemC pas très convenable pour les techniques des vérifications formelles.
La définition d'une sémantique opérationnelle n'est pas nouvelle [31, 32, 110, 111, 112], par exemple, le langage ASM est utilisée pour définir la sémantique de simulation SystemC [31, 110], comme la sémantique du SpecC [102], l'équivalent du SystemC, ou la sémantique de VHDL. Malheureusement, ces travaux ne considèrent que l'ordonnanceur de SystemC1.0 qui est vraiment différent de la version 2.0. Les principaux objectifs de ces travaux sont la définition d'une spécification précise des implémentations futures de l'ordonnanceur ou d'étudier l'interopérabilité du SystemC avec Verilog, SpecC et VHDL.
Notre objectif est de fournir une sémantique formelle à l'ordonnanceur SystemC exprimée en langage Maude.
Notre approche est similaire, à [49], en termes des objectifs. Essentiellement, elle consiste à l'application de techniques formelles pour la vérification et la validation des programmes SystemC.

4. Une sémantique basée sur la logique de réécriture pour SystemC

Dans cette section, nous essayons d'expliquer en détail la sémantique de l'ordonnanceur SystemC et sa sémantique opérationnelle exprimée en Maude.

Une spécification SystemC est composée de deux parties: une partie statique définissant la structure du système et les types de données, et une partie dynamique, qui définit le comportement du système.

La partie statique de spécifications SystemC peut être modélisée en utilisant les méthodes de spécification algébrique fournies par Maude. La dynamique d'une spécification SystemC est modélisée à l'aide des règles de réécriture. De cette façon, nous intégrons les aspects statiques et dynamiques d'un modèle SystemC au sein d'un seul cadre sémantique. L'objectif de ce travail est l'aspect dynamique.

Comme indiqué précédemment, SystemC comprend trois types de processus qui sont SC_METHODs, SC_THREADs et SC_CTHREADS.

Le comportement d'un SC_METHOD ressemble à une procédure dans le sens, il retourne le contrôle quand il termine l'exécution. Son exécution est déclenchée chaque fois au moins quand l'un des signaux définis dans sa liste de sensibilité se change. Une fois que l'exécution commence, il ne peut pas être interrompu. SC_METHOD ne peut pas comprendre une instruction *wait*.

Contrairement à un SC_METHOD, un SC_THREAD peut comporter plusieurs instructions *wait* et il représente généralement une boucle infinie. Le tableau 6.1 donne les différentes sémantiques de l'instruction *wait*.

wait	**Sémantique**
wait()	wait on events in sensitivity list.
wait(e1)	wait on event e1.
wait(e1 | e2 | e3)	wait on events e1, e2, or e3.
wait(e1 & e2 & e3)	wait on events e1, e2, and e3.
wait(200, SC_NS)	wait for 200 ns.
wait(200, SC_NS, e1)	wait on event e1, timeout after 200 ns.
wait(200, SC_NS, e1 | e2 | e3)	wait on events e1, e2, or e3, timeout after 200 ns.
wait(200, SC_NS, e1 & e2 & e3)	wait on events e1, e2, and e3, timeout after 200 ns.

Tableau 6.1. Les différentes sémantiques de l'instruction *wait* [55].

Une fois que l'exécution commence, elle se poursuit jusqu'à la première instruction *wait*. Dans ce cas, l'ordonnanceur bloque le processus en cours d'exécution et commute le contexte pour exécuter un autre processus qui est prêt. Lorsqu'un ou plusieurs événements de la liste de sensibilité se produisent, SC_THREAD reprend son exécution jusqu'à la prochaine instruction *wait*.

En SystemC, les processus communiquent via des canaux. Les canaux sont utilisés pour la communication entre les processus au sein des modules et entre les modules. A l'intérieur d'un module un processus peut accéder directement à un canal. Si un canal est connecté à un port d'un module, le processus accède au canal par l'intermédiaire du port. Il existe deux classes de canaux : primitif et hiérarchique.

Les canaux primitifs n'ont pas de structure visible, pas de processus et ne peuvent pas accéder directement aux autres canaux primitifs. Il existe plusieurs types de canaux primitifs en SystemC: *sc_signal*, *sc_buffer*, *sc_signal_rv*, *sc_fifo*, *sc_mutex*, et *sc_semaphore*.

Les canaux hiérarchiques sont des modules qui peuvent contiennent des processus et des sous-modules. Ils peuvent directement accéder à d'autres canaux. Dans ce travail, nous nous somme intéressés aux canaux de type *sc_signal* (signaux).

Les signaux sont utilisés pour décrire le matériel et sont généralement utiles pour la modélisation RTL. Ils sont utilisés soit pour la communication point à point ou multipoints. Les signaux ne sont pas résolus : une seule écriture est autorisée. Un signal est titulaire d'une valeur de données typée. Les signaux implémentent la sémantique évaluer-mettre à jour

(evaluate-update). Un signal possède une valeur actuelle, la nouvelle valeur et l'ancienne valeur.

Au cours de la phase d'évaluation de delta-cycle, l'écriture d'une valeur sur le signal implique l'affectation de cette valeur à sa nouvelle valeur. La dernière écriture gagne quand plusieurs écritures successives ont lieu. Au cours de la phase de mise à jour de delta-cycle, si la nouvelle valeur est différente de la valeur actuelle, alors la valeur actuelle est déplacée à l'ancienne valeur et la nouvelle valeur est déplacée à la valeur actuelle et un événement se produit.

Un canal *sc_buffer* se comporte exactement comme un signal, sauf que l'écriture d'une valeur est toujours mise à jour et l'événement *value_changed_event* se produit, même lorsque la nouvelle valeur est égale à la valeur actuelle. Le canal *sc_signal_rv* peut avoir plus d'une écriture. Un canal *sc_fifo* implémente une FIFO. Il s'agit d'une connexion point à point et peut seulement être connecté à un seul port d'entrée et un port de sortie. Le canal *sc_mutex* est utilisé pour l'accès sécurisé à une ressource partagée, et en fin le canal *sc_semaphore* est similaire au canal *sc_mutex* mais pour un nombre limité des accès simultanés.

4.1. Spécification de processus et de comportements

Afin d'être capable de transformer les programmes SystemC en spécifications Maude, nous devons abstraire les comportements des processus. Ainsi, chaque comportement de processus est modélisé comme une machine à états finis où chaque état appartient à l'un des états génériques suivants: *TNotify*, *Inotify*, *Update*, *Stop*, *Wait*, *BEGIN* et *END* où *TNotify* désigne une notification temporisée, *Inotify* désigne une notification immédiate, *Update* désigne une mise à jour du canal (écriture), *Wait* désigne l'instruction d'attente, et *Stop* désigne l'arrêt explicite de la simulation. BEGIN et END désignent les états de début et de fin respectivement. En d'autres termes, nous prenons en compte que les états ayant une influence sur le comportement global du système. Parfois, nous sommes obligés d'ajouter un état de lecture d'un canal pour tester sa valeur actuelle. En utilisant Maude, les comportements de *SC_METHODs*, et *SC_THREADs* sont spécifiés comme un ensemble de règles de réécriture fini et infini respectivement. Chaque règle de réécriture correspond à un état de transition. Afin de distinguer entre les états, nous attachons à chaque état de notification le nom de l'événement et le nom du canal à chaque état de mise à jour ou de lecture.

**** Declaration des états des processus*
sort ProcessesStates .

sorts Begin End UpdateCH1 INotifyEv1 TNotifyEv2 WaitEv1 ReadCH2.
subsorts Begin End UpdateCH1 INotifyEv1 TNotifyEv2 WaitEv1 ReadCH2 < ProcessesStates
op Begin : -> Begin .
op End : -> End .
op UpdateCH1 : -> UpdateCH1 .
op INotifyEv1 : -> INotifyEv1 .
op TNotifyEv2 : -> TNotifyEv2 .
op WaitEv1 : -> WaitEv1 .
op ReadCH2 -> ReadCH2 .

Les processus sont déclarés comme étant des instances des classes Maude. Depuis, que les processus au sein de différents modules peuvent avoir le même nom, nous attachons aux noms des processus les noms des modules incluant ces processus. Chaque processus dispose de trois attributs qui sont la liste de sensitivité, le type de processus, et l'état actuel du processus. La liste de sensitivité est implémentée comme une liste de chaînes des caractères.

**** Declaration de la classe Process*
op PROCESS : -> Cid [ctor] .

**** Declaration de types de processus*
sort ProcessType .
sorts METHOD THREAD CTHREAD .
subsorts METHOD THREAD CTHREAD < ProcessType .
op METHOD : -> METHOD .
op THREAD : -> THREAD .
op CTHREAD : -> CTHREAD .

**** Declaration de la liste de sensitivité*
sort EventList .
subsort String < EventList .
op nil : -> EventList [ctor] .
op _,_ : EventList EventList -> EventList [ctor assoc id: nil] .

**** Declaration des attributs des processus*
op SensitivityList :_ : EventList -> Attribute [ctor gather (&)] .

op PType :_ : ProcessType -> Attribute [ctor gather (&)] .
op state :_ : ProcessesStates -> Attribute [ctor gather (&)] .

Le déclenchement de processus est spécifié en utilisant le concept de message Maude. Nous définissons un message générique: *start* pour déclencher l'exécution d'un processus.

**** Declaration de message*
op start : Oid -> Msg [ctor] .

Voici deux exemples de règles de réécriture Maude:

rl [startproc1] :
start(proc1)
< proc1 : PROCESS | PType : METHOD ,SensitivityList : l, state : BEGIN >
=>
< proc1 : PROCESS | PType : METHOD, SensitivityList : l, state : INotifyEv1 > .

rl [end] :
< proc1 : PROCESS | PType : METHOD, SensitivityList : l, state : INotifyEv1 >
=>
< proc1 : PROCESS | PType : METHOD, SensitivityList : (“Ev2”), state : END > .

Ces deux règles concernent la machine d'état du processus SC_METHOD qui génère une notification immédiate (INotifyEv1), puis se termine. La deuxième règle permet de changer la liste de sensitivité (la liste par défaut) à une nouvelle liste de sensitivité, y compris "ev2" (i.e. liste de sensitivité dynamique).

rl [startproc2] :
start(proc2)
< proc2 : PROCESS | PType : THREAD ,SensitivityList : l, state : BEGIN >
=>
< proc2 : PROCESS | PType : THREAD, SensitivityList : l, state : WaitEv1 > .

rl [end] :
< proc2 : PROCESS | PType : THREAD ,SensitivityList : l, state : WaitEv1 >

=>

< proc2 : PROCESS | PType : THREAD, SensitivityList : l, state : END > .

rl [loop] :

< proc2 : PROCESS | PType : THREAD, SensitivityList : l, state : END >

=>

< proc2 : PROCESS | PType : THREAD, SensitivityList : l, state : BEGIN > start(proc2) .

Ces trois règles spécifient un processus SC_THREAD qui attend l'événement "ev1". On note qu'un processus SC_THREAD est implémenté comme une boucle infinie.

4.2. Spécification de canaux

Les canaux primitifs sont déclarés comme des instances des classes Maude. Chaque canal a au moins six attributs qui sont le type de canal, l'ancienne valeur, la valeur actuelle, la nouvelle valeur, et les deux processus liés par ce canal.

**** Declaration d'une classe CHANNEL*

op CHANNEL : -> Cid [ctor] .

**** Declaration de types des canaux*

sort ChannelType .

sorts SIGNAL BUFFER FIFO MUTEX SEMAPHORE.

subsorts SIGNAL BUFFER FIFO MUTEX SEMAPHORE < ChannelType .

op SIGNAL : -> SIGNAL .

op BUFFER : -> BUFFER .

op FIFO : -> FIFO .

op MUTEX : -> MUTEX .

op SEMAPHORE : -> SEMAPHORE .

**** Declaration des attributs du cannal*

op CHType :_ : ChannelType -> Attribute [ctor gather (&)] .

op Old :_ : String -> Attribute [ctor gather (&)] .

op Current :_ : String -> Attribute [ctor gather (&)] .

op New :_ : String -> Attribute [ctor gather (&)] .

op End1 :_ : Oid -> Attribute [ctor gather (&)] .

op End2 :_ : Oid -> Attribute [ctor gather (&)] .

End1 et *End2* spécifient les deux processus liés par le canal. Dans le cas, le canal relie plus de deux processus, nous devons rajouter plus d'attributs *Endx*.

4.3. Spécification d'exécution d'ordonnanceur

Avant de spécifier la sémantique de l'ordonnanceur, il faut définir un ensemble de listes et de fonctions. Les tableaux 6.2 et 6.3 donnent un aperçu sur les principales listes et fonctions Maude que nous avons utilisé dans notre sémantique.
En plus de ces listes et fonctions, on définit un objet *Timer* qui garde trace de temps courant (CTime), de temps précédant (PTime), et de temps de simulation (STime).

**** Declaration de la classe Timer Class & ses attributs.*
op TIMER : -> Cid [ctor] .
op CTime :_ : Int -> Attribute [ctor gather (&)] .
op PTime :_ : Int -> Attribute [ctor gather (&)] .
op STime :_ : Int -> Attribute [ctor gather (&)] .

Nom de liste	Rôle
NOTRUNNABLE	Stocke la liste des processus qui ne sont pas prêt pour l'exécution
IRUNNABLE	Stocke la liste des processus qui sont déclenchés par une notification immédiate
DRUNNABLE	Stocke la liste des processus qui sont déclenchés par la mise jour d'un canal (delayed notifications)
TRUNNABLE	Stocke la liste des processus qui sont déclenchés par une notification temporisée
IMMEDIATE	Stocke la liste de notifications immédiates
DELAYED	Stocke la liste de notifications retardées (delayed notifications)
TIMED	Stocke la liste de notifications temporisées (timed notifications)
TEVENTS	Stocke la liste des notifications temporisées et les temps correspondants
FTIMES	Stocke les temps futurs
INTERM	Liste intermédiaire

Tableau 6.2. L'ensemble de listes implémentées en Maude.

Nom de fonction	Rôle
first	Retourne le premier élément d'une liste
addb	Ajouter un élément au début d'une liste
adde	Ajouter un élément à la fin d'une liste
delete	Supprimer le premier élément d'une liste
pdelete	Supprimer un élément donné d'une liste
tdelete	Supprimer un élément donné de la liste TEVENTS
in	Retourne vrai, si un élément donné appartient à une liste donnée
size	Retourne la taille d'une liste
min	Retourne le minimum d'une liste
orsensible	Retourne vrai si un processus est sensible à un événement
extractevent	Retourne le nom d'événement pour une valeur de temps donnée

extracttime	Retourne la valeur de temps pour un événement donné
extract	Extraire un événement de la liste TEVENTS
rand	Générer un nombre aléatoire

Tableau 6.3. La liste de fonctions implémentées en Maude.

Ci-dessous, nous présentons quelques règles de réécriture spécifiant la sémantique d'ordonnancement :

*rl [notify1] : ***1*
< proc1 : PROCESS | PType : METHOD, SensitivityList : l, state : INotifyEv1 >
< notif : LIST | IMMEDIATE : imm >
=>
< proc1 : PROCESS | PType : METHOD, SensitivityList : l, state : TNotifyEv4 >
< notif : LIST | IMMEDIATE : adde(imm,"ev3") > .

*rl [notify4] : ***2*
< proc1 : PROCESS | PType : METHOD, SensitivityList : l, state : TNotifyEv4 >
< tevents : LIST | TEVENTS : tevent >
< times : LIST | FTIMES : ft >
=>
< proc1 : PROCESS | PType : METHOD, SensitivityList : l, state : UpdateCH1 >
< tevents : LIST | TEVENTS : adde(tevent,["ev4",8]) >
< times : LIST | FTIMES : adde(ft,8) > .

*rl [update1] : ***3*
< proc1 : PROCESS | PType : METHOD, SensitivityList : l, state : UpdateCH1 >
< ch1 : CHANNEL | CHType : SIGNAL, End1 : proc1, End2 : proc2, Old : old, Current : current, New : new >
< dnotif : LIST | DELAYED : delay >
=>
if (current =/= new) then
< proc1 : PROCESS | PType : METHOD, SensitivityList : l, state : END >
< ch1 : CHANNEL | CHType : SIGNAL, End1 : proc1, End2 : proc2, Old : current, Current :

new, New : rand >

< dnotif : LIST | DELAYED : adde(delay,"updatech1") >

else

< proc1 : PROCESS | PType : METHOD, SensitivityList : l, state : END >

< ch1 : CHANNEL | CHType : SIGNAL, End1 : proc1, End2 : proc2, Old : old, Current : current, New : new >

< dnotif : LIST | DELAYED : delay > fi .

*rl [reset] : ***4*

< proc1 : PROCESS | PType : METHOD, SensitivityList : l, state : END >

< notrunnable : LIST | NotRunnable : lpn >

=>

< proc1 : PROCESS | PType : METHOD, SensitivityList : l, state : BEGIN >

< notrunnable : LIST | NotRunnable : adde(lpn,proc1) > .

*crl [METHODImmediateresume] : ***5*

< notif : LIST | IMMEDIATE : imm >

< irunnable : LIST | IRunnable : lp >

< notrunnable : LIST | NotRunnable : lpn >

< proc1 : PROCESS | PType : METHOD, SensitivityList : l, state : BEGIN >

=>

< notif : LIST | IMMEDIATE : imm >

< irunnable : LIST | IRunnable : adde(lp,proc1) >

< notrunnable : LIST | NotRunnable : pdelete(proc1,lpn) >

< proc1 : PROCESS | PType : METHOD, SensitivityList : l, state : BEGIN >

if (orsensible(l,imm)) and (in(proc1,lpn)) and (imm =/= nil) .

*crl [resumeMethods] :***6*

< notif : LIST | IMMEDIATE : imm >

< irunnable : LIST | IRunnable : lp >

=>

< notif : LIST | IMMEDIATE : imm >

< irunnable : LIST | IRunnable : delete(lp) > start(first(lp)) if (imm == nil) and (lp =/= nil)

.

*crl [settime] :***7*

< timer : TIMER | CTime : ct, PTime : pt, STime : st >

< times : LIST | FTIMES : ft >

< tnotif : LIST | TIMED : tim >

< dnotif : LIST | DELAYED : delay >

< drunnable : LIST | DRunnable : lp >

< notif : LIST | IMMEDIATE : imm >

< irunnable : LIST | IRunnable : lp1 >

=>

< dnotif : LIST | DELAYED : delay >

< drunnable : LIST | DRunnable : lp >

< notif : LIST | IMMEDIATE : imm >

< irunnable : LIST | IRunnable : lp1 >

< timer : TIMER | CTime : ct + min(ft), PTime : ct, STime : st >

< tnotif : LIST | TIMED : tim >

< times : LIST | FTIMES : ft > timed(tnotif) if (delay == nil) and (lp == nil) and (imm == nil) and (lp1 == nil) and (ft =/= nil) and (ct < st) .

*crl [mindate] :***8*

timed(tnotif)

< tevents : LIST | TEVENTS : tevent >

< times : LIST | FTIMES : ft >

< tnotif : LIST | TIMED : tim >

< timer : TIMER | CTime : ct, PTime : pt, STime : st >

=>

< tevents : LIST | TEVENTS : tdelete(min(ft),tevent) >

< times : LIST | FTIMES : pdelete(min(ft),ft) >

< tnotif : LIST | TIMED : adde(tim,extract(min(ft),tevent)) >

< timer : TIMER | CTime : ct, PTime : pt, STime : st > timed(tnotif) if ((ct - pt) == min(ft)) and (tevent =/= nil) .

*crl [resumeMethods] :***9*

< tnotif : LIST | TIMED : tim >

< trunnable : LIST | TRunnable : lp >

< dnotif : LIST | DELAYED : delay >

< drunnable : LIST | DRunnable : lp1 >

< notif : LIST | IMMEDIATE : imm >

< irunnable : LIST | IRunnable : lp2 >

< timer : TIMER | CTime : ct, PTime : pt, STime : st >

=>

< dnotif : LIST | DELAYED : delay >

< drunnable : LIST | DRunnable : lp1 >

< notif : LIST | IMMEDIATE : imm >

< irunnable : LIST | IRunnable : lp2 >

< tnotif : LIST | TIMED : tim >

< trunnable : LIST | TRunnable : delete(lp) >

< timer : TIMER | CTime : ct, PTime : pt, STime : st >

start(first(lp)) if (tim == nil) and (lp =/= nil) and (delay == nil) and (lp1 == nil) and (imm == nil) and (lp2 == nil) .

La première règle permet de mémoriser les notifications immédiates dans la liste IMMEDIATE.

La deuxième règle permet de mémoriser les notifications temporisées (événements et temps) et les temps correspondants dans les listes TEVENTS et FTIMES respectivement.

La troisième règle permet de mémoriser une notification tardive (delayed notification) qui résulte d'une mise à jour du canal (écriture). La mise à jour d'événement est considérée que si la valeur actuelle du canal (SIGNAL) est différente de sa nouvelle valeur. Dans ce cas, l'ancienne valeur devient la valeur actuelle, la valeur actuelle devient la nouvelle valeur, et la nouvelle valeur est créée de façon aléatoire en utilisant la fonction *rand*.

La quatrième règle permet d'ajouter un processus SC_METHOD à la liste *NotRunnable* dés qu'il atteint l'état final (fin d'exécution) et de réinitialiser son état à l'état de départ BEGIN. Nous appliquons la même règle pour les SC_THREADs mais pour chaque état *Wait*.

La cinquième règle permet d'ajouter les processus SC_METHOD qui sont sensibles aux notifications immédiates à la liste *IRunnable*. La même idée est appliquée pour les SC_METHODs qui sont sensibles aux notifications tardives et temporisées.

La sixième règle permet de déclencher des processus SC_METHOD qui sont sensibles aux notifications immédiates et de les supprimer de la liste IRunnable. Nous appliquons le même

principe pour les processus SC_METHOD qui sont sensibles aux notifications tardives et temporisées. On note qu'avant le déclenchement des processus, toutes les notifications immédiates doivent être supprimées de la liste IMMEDIATE. Afin de préserver la sémantique d'ordonnanceur SystemC, nous devons vérifier que les listes *IRunnable* et *IMMEDIATE* sont vides avant de déclencher les processus qui sont sensibles aux notifications tardives et les listes *DRunnable* et *DELAYED* sont vides avant de déclencher les processus qui sont sensibles aux notifications temporisées.

La septième règle permet de mettre à jour les temps courant et précédant de l'objet Timer, et déclenche l'ordonnanceur pour traiter les notifications temporisées en générant le message *timed*. Notons que, avant de faire ça, toutes les listes des notifications immédiates et tardives doivent être vides.

La huitième règle permet de mémoriser des événements qui se produisent au même moment dans la liste *TIMED* et supprimer ces événements et les temps correspondants de deux listes *TEVENTS* et *FTIMES* respectivement.

La neuvième règle permet le déclenchement des processus qui sont sensibles aux notifications temporisées.

La fin de l'exécution de l'ordonnanceur est spécifiée explicitement par l'introduction de temps de simulation ou de l'instruction (Stop) au sein de comportement d'un processus.

Les règles de réécriture décrites ci-dessus spécifient les phases d'évaluation et de mise à jour de l'ordonnanceur SystemC. La phase d'initialisation est spécifiée en tant que configuration initiale du système. Ainsi, les processus déclarés par *dont_initialize* sont mis dans la liste *NOTRUNNABLE*. Pour indiquer le fait que le processus est exécutable, nous devons produire son message *start* dans la configuration initiale. Toutes les initialisations concernant les processus, les canaux et les listes sont aussi mises dans la configuration initiale. Enfin, nous constatons que *wait(t)* où t est une constante non nulle qui peut être remplacée par *timeout.notify (t)* et *wait(timeout)*. De même, quand un processus SC_METHOD est sensible à un signal d'horloge, nous introduisons au début du processus *clk.notify(t)* où t est la période de l'horloge (par défaut t égal à un).

4.4. Spécification de propriétés

L'objectif de transformation de programmes SystemC en langage Maude est d'exploiter son vérificateur de modèles (model Checker) pour vérifier formellement l'exactitude des programmes SystemC vis-à-vis de certaines propriétés. Nous nous sommes intéressés aux deux propriétés : la première propriété intéressante que nous devrions vérifier est le fait que la

simulation se termine uniquement quand il n'ya plus de processus exécutables ; c'est une propriété importante de la simulation qui est mentionnée dans le manuel de référence du SystemC. La seconde propriété consiste à vérifier si tous les processus qui sont sensibles aux notifications immédiates s'exécutent avant ceux qui sont sensibles aux notifications tardives avant ceux qui sont sensibles aux notifications temporisées. Bien que, cette propriété soit définie implicitement dans les règles de réécriture, en ajoutant certaines conditions, on n'est pas sûr si cette propriété est toujours vérifiée. Cela est dû au fait que les règles de réécriture sont exécutées de façon non-déterministe.

Pour réaliser notre objectif, nous avons défini cinq propriétés nommées *SimulationEnd*, *RunnableIsEmpty*, *Immediat*, *Delayed* et *Timed*.

En Utilisant Maude, nous pouvons définir les cinq propriétés comme suit:

```
var cf : Configuration .
op SimulationEnd : Configuration -> Prop .
op RunnableIsEmpty : Configuration -> Prop .
op Immediate  : Configuration -> Prop .
op Delayed : Configuration -> Prop .
op Timed : Configuration -> Prop .

ceq < timer : TIMER | CTime : ct, PTime : pt, STime : st > cf
|= SimulationEnd (< timer : TIMER | CTime : ct, PTime : pt, STime : st > cf) = true if ct > st
or ct == st  .
ceq < irunnable : LIST | IRunnable : lp > < drunnable : LIST | DRunnable : lp1 > <
trunnable : LIST | TRunnable : lp2 > cf
|= RunnableIsEmpty (< irunnable : LIST | IRunnable : lp > < drunnable : LIST | DRunnable
: lp1 > < trunnable : LIST | TRunnable : lp2 > cf) = true if (lp == nil) and (lp1 == nil) and
(lp2 == nil) .
ceq < irunnable : LIST | IRunnable : lp >   cf
|= Immediate (< irunnable : LIST | IRunnable : lp > cf) = true if
lp == nil .
ceq < drunnable : LIST | DRunnable : lp > cf
|= Delayed (< drunnable : LIST | DRunnable : lp > cf) = true if
 lp =/= nil .
ceq < trunnable : LIST | TRunnable : lp >  cf
|= Timed (< trunnable : LIST | TRunnable : lp > cf)  = true if
```

lp =/= nil .

[] (SimulationEnd(initial) -> RunnableIsEmpty(initial)).

Cette propriété exprime qu'à partir de la configuration initiale, toujours si la simulation atteint sa fin, alors toutes les listes de processus exécutables sont vides.

[] (Delayed(initial) -> not(Immediate (initial))).

Cette propriété exprime qu'à partir de la configuration initiale, toujours si des processus exécutables qui sont déclenchés par des notifications tardives existent, alors la liste des processus exécutables qui sont déclenchés par des notifications immédiates est vide.

[] (Timed(initial) -> (Immediate (initial)) /\ (~(Delayed(initial)))).

Cette propriété exprime qu'à partir de la configuration initiale, toujours si des processus exécutables qui sont déclenchés par des notifications temporisées existent, alors la liste des processus exécutables qui sont déclenchés par des notifications tardives est vide.

On note que le symbole [] signifie globalement: la propriété doit être vérifiée pour chaque état de chaque chemin de calcul. Le symbole -> signifie "implique". / \ Et ~ et sont les opérateurs logiques "and" et "not" respectivement. *initial* est la configuration initiale.

5. Etude de cas

Afin de valider notre sémantique, nous avons choisi le protocole de données simplex (SDP), comme étant une étude de cas [117].

Le SDP est un protocole de données utilisé pour transférer des données d'un dispositif vers un autre dans une seule direction. Le SDP peut détecter des erreurs de transfert, et il peut renvoyer les paquets de données pour mener à bien le transfert de données si des erreurs sont détectées.

La description SystemC contient le bloc Emetteur (*transmit)*, le bloc Receveur (*receiver)*, le bloc *channel*, le bloc *Timer* et le bloc d'affichage (*display)* (voir figure 6.5). L'émetteur envoie des paquets de données au canal. Ce dernier reçoit les paquets et les envoie au récepteur. Le canal peut introduire des erreurs pour représenter le taux d'erreur réel de support physique. Le récepteur reçoit les paquets de données de canal et les analyse pour les erreurs. Si le paquet de données n'a pas d'erreurs, le récepteur génère un paquet d'accusé de réception et envoie ce paquet au canal. Le canal reçoit le paquet d'accusé de réception et l'envoie à l'émetteur. Le canal peut introduire des erreurs lors de l'envoi du paquet d'accusé de réception. Après la réception de paquet d'accusé de réception pour le paquet de données

précédemment envoyé, l'émetteur envoie le prochain paquet. Ce processus continue jusqu'à ce que tous les paquets de données soient envoyés. Le bloc *timer* génère des événements d'expiration de temps (timeout). Les paquets sont générés par une fonction aléatoire dans le bloc d'émetteur et sont envoyés par le biais du canal au récepteur. Le récepteur envoie les données à l'écran pour l'affichage [117].

L'application comprend cinq processus SC_METHOD, un processus SC_THREAD et sept canaux (signaux). Nous avons défini trois autres objets: PACK pour spécifier l'objet paquet de données, COUNT pour spécifier un objet compteur, et RAN pour spécifier un objet aléatoire. Nous définissons également un nouveau type *paquet*. Ce dernier contient des informations sur les données à transmettre (buffer), le numéro de séquence (seq), et *retry*. Les données sont générées de manière aléatoire en utilisant la fonction *rand*. La figure 6.6 illustre un fragment du code Maude pour l'exemple du SDP. Les résultats du model checker sont présentés en figure 6.7.

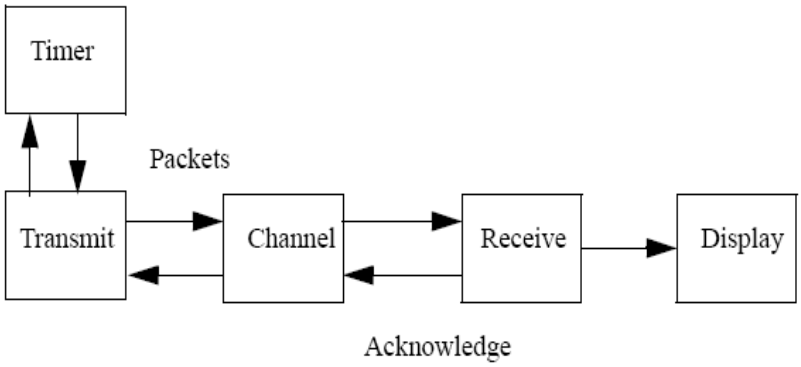


Figure 6.5. Diagramme de blocs du SDP [117].

sort packet .

op (_,_,_) : Int Int Int -> packet .

< TransmitSendData : PROCESS1 | PType : METHOD, SensitivityList : ("clock","timeout"), state : BEGIN >

< ChannelReceiveData : PROCESS2 | PType : METHOD, SensitivityList : ("tpackin"), state : BEGIN >

< ChannelSendAck : PROCESS3 | PType : METHOD, SensitivityList : ("rpackin"), state : BEGIN >

< ReceiverReceiveData : PROCESS4 | PType : METHOD, SensitivityList : ("rclock"), state : BEGIN >

< DisplayPrintData : PROCESS5 | PType : METHOD, SensitivityList : ("dout"), state : BEGIN >

< TimerRunTimer : PROCESS6 | PType : THREAD, SensitivityList : ("clock","starttimer"), state : BEGIN >

< PACK : PACKET | buffer : rand, framenum : 1, retry : 0 >

< COUNT : COUNTER | value : 0 >

< RAN : RAND | value : rand >

< TPACKIN : CHANNEL1 | CHType : SIGNAL, Oldpacket : (0,1,0) , Currentpacket : (0,1,0) , Newpacket : (0,1,0) , End1 : TransmitSendData, End2 : ChannelReceiveData >

< TIMEOUT : CHANNEL2 | CHType : SIGNAL, Old : "false" , Current : "false" , New : "false" , End1 : TransmitSendData, End2 : TimerRunTimer >

< TPACKOUT : CHANNEL3 | CHType : SIGNAL, Oldpacket : (0,1,0) , Currentpacket : (0,1,0) , Newpacket : (0,1,0), End1 : TransmitSendData, End2 : ChannelSendAck >

< STARTTIMER : CHANNEL4 | CHType : SIGNAL, Old : "false", Current : "false", New : "false", End1 : TransmitSendData, End2 : TimerRunTimer >

< RPACKIN : CHANNEL5 | CHType : SIGNAL, Oldpacket : (0,1,0), Currentpacket : (0,1,0), Newpacket : (0,1,0), End1 : ChannelSendAck, End2 : ReceiverReceiveData >

< RPACKOUT : CHANNEL6 | CHType : SIGNAL, Oldpacket : (0,1,0), Currentpacket : (0,1,0), Newpacket : (0,1,0),End1 : ChannelReceiveData, End2 : ReceiverReceiveData >

< DOUT : CHANNEL7 | CHType : SIGNAL, Old : "" , Current : "", New : "" ,End1 : ReceiverReceiveData, End2 : DisplayPrintData >

< Timer : TIMER | CTime : 0, PTime : 0, STime : 1000 > .

Les résultats montrent bien que toutes les propriétés sont toujours satisfaites (vraies).

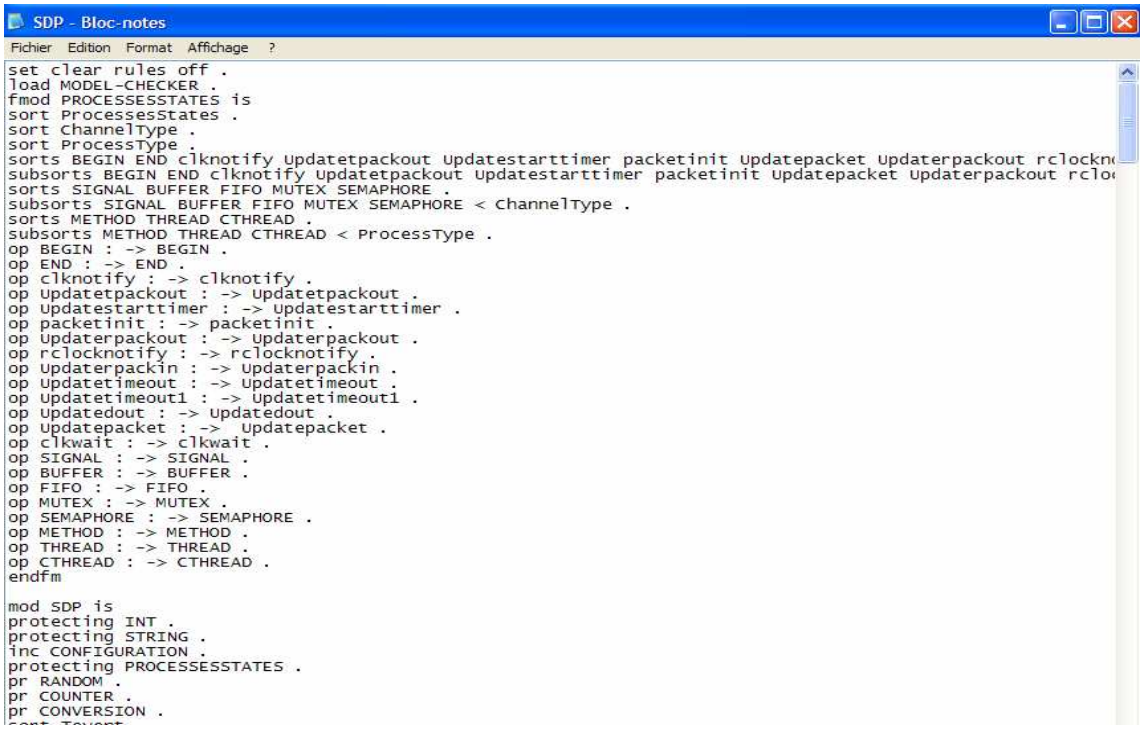


Figure 6.6. Fragment du code Maude pour le SDP.

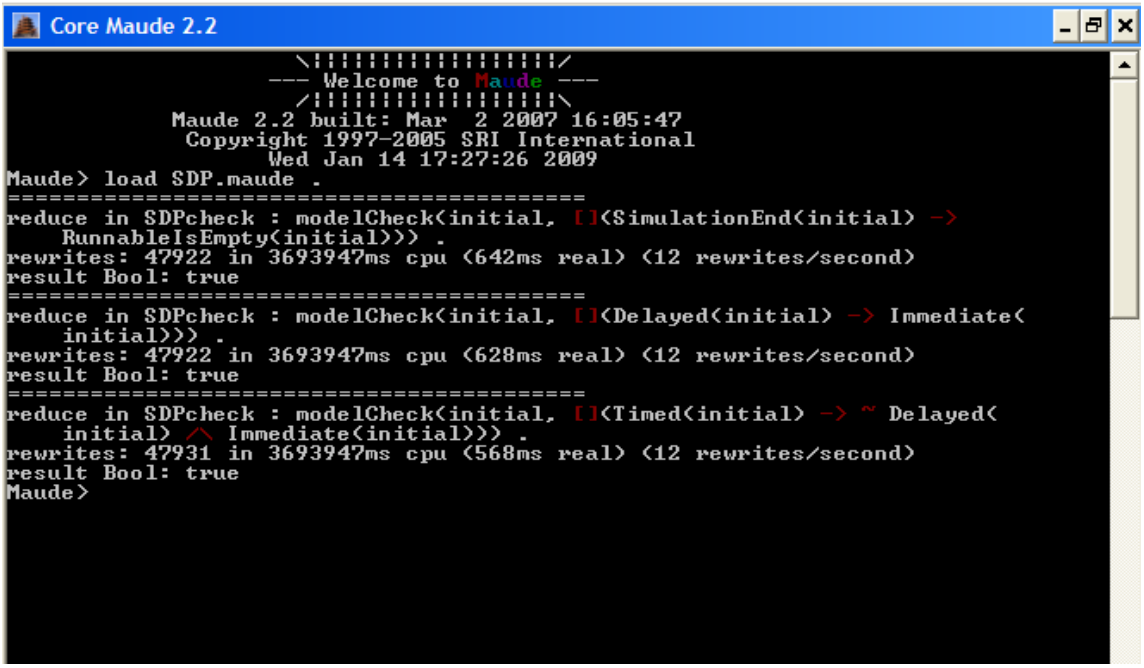


Figure 6.7. Résultats du model checker.

D’après les résultats du model checker, nous pouvons constater que toutes les propriétés que nous avons définit sont satisfaites pour l’exemple du SDP.

6. Conclusion

Dans ce chapitre, nous avons défini une sémantique opérationnelle pour l’ordonnanceur SystemC. Notre sémantique proposée s’appuie sur la logique de réécriture et implémentée en langage Maude. Nous nous sommes intéressés à l’aspect dynamique de l’ordonnanceur. Par conséquent, nous pouvons constater que la vérification formelle peut renforcer la simulation et permet éventuellement de détecter des erreurs qui ne peuvent pas être détectées par le moteur de simulation du SystemC.

Septième Chapitre

Conclusion et Perspectives

1. Conclusion

Nous avons proposé dans cette thèse une collection de méthodes et d'outils qui aident les concepteurs des systèmes multiprocesseurs mono puce à modéliser, estimer les performances, simuler et vérifier de manière formelle les MPSOCs à un niveau élevé d'abstraction. Nous avons également développé un flot qui permet l'intégration d'UML avec les outils CAD de synthèse de haut niveau et de co-simulation ciblant les architectures réconfigurables.

L'abstraction est un point clé de notre travail. Pour cela, nous nous sommes basés sur l'UML comme point d'entrée de nos outils développés.

Puisque UML ne propose aucune méthodologie de conception, nous avons suivi les principes de la méthode dite en Y pour systèmes embarqués. Nous avons donc séparé entre modèles d'application, d'architecture, d'association, de traitement orienté données, de traitement orienté contrôle, de calcul et de communication. Cette séparation des préoccupations va certainement accroître la réutilisation et minimiser les coûts de la maintenance.

La première contribution à consisté à développer un profil UML pour estimer les performances au pire de cas (temps d'exécution et consommation d'énergie) d'une application s'exécutant sur MPSOC.

Ce profil propose un ensemble de stéréotypes permettant la modélisation des différents types du parallélisme qu'on trouve dans une application embarquée typique comme le parallélisme des tâches et de données. Le profil propose aussi un ensemble d'heuristiques pour passer d'un modèle séquentiel vers un modèle concurrent hiérarchique, pour l'association et pour l'optimisation.

De manière similaire, Le profil définit un ensemble des stéréotypes pour modéliser les composants matériels de l'architecture comme CPU, FPGA, IP, Bus, Mémoire, etc.…, et l'association de l'application à l'architecture (Mapping). Nous avons modélisé l'architecture du MPSOC de sorte qu'elle soit générique. Notons que les valeurs de performances obtenues sont relatives dans le sens où elles sont utilisées pour comparer entre les solutions possibles résultant de partitionnement. Dans notre cas, c'est le concepteur qui partitionne l'application selon son expérience.

La deuxième contribution porte sur la modélisation, la vérification formelle et la simulation des systèmes embarqués mixte. Pour ce faire, nous avons utilisé les diagrammes d'états transitions à zéro délai et les diagrammes d'activités d'UML2 avec actions à forte

granularité pour modéliser le traitement dominé par contrôle et le traitement dominé par les données respectivement.

Les actions à forte granularité désignent des calculs incluant un certain nombre des opérations élémentaires, des opérations de lecture à partir d'un canal ou l'écriture sur un canal abstrait. Les données transmises sur les canaux sont des données abstraites et elles sont exprimées en termes de jetons de données. Cette abstraction de données facilite largement la simulation et la vérification formelle.

La communication entre tâches se fait via des canaux abstraits. Deux types des canaux sont distingués: canaux transportant les données de contrôle et ceux qui transportent les données proprement dites. A partir des modèles UML, une spécification Maude est générée.

Chaque transition dans le diagramme d'états transitions ou d'activité est spécifiée en tant qu'une règle de réécriture.

Nous utilisons cette spécification formelle d'un coté pour estimer la consommation d'énergie à un niveau élevé d'abstraction et pour vérifier formellement quelques propriétés de l'autre coté.

Afin de créer un pont entre UML et les langages standards de simulation et de synthèse au niveau système, nous avons transformé les modèles UML vers le langage SystemC. Cette transformation nécessite un raffinement des modèles UML initiaux.

Les tâches dominées par le contrôle sont implémentées à l'aide des machines à états finis et les tâches dominées par les données comme étant des *Threads* SystemC. Les canaux des données sont implémentés comme des *fifos* SystemC et les canaux de contrôle comme étant des signaux.

Les codes Maude et SystemC sont générés automatiquement sous forme des fichiers textes en utilisant l'API VB qui est intégrée dans l'environnement Rhapsody.

La troisième contribution concerne l'intégration d'UML avec les outils CAD de synthèse de haut niveau et de co-simulation ciblant les architectures réconfigurables.

Le flot proposé utilise UML comme point d'entrée pour modéliser l'application et l'architecture. Le partitionnement SW/HW est également modélisé en utilisant le diagramme de séquence ou le diagramme d'objets d'UML2. Pour ce faire, nous avons défini un nouveau stéréotype appelé « HW » qui désigne le nom d'IP implémentant la fonction matérielle.

Le partitionnement se fait en se basant sur les résultats du profilage. Le choix de topologie de communication s'appuie sur l'analyse de la charge de communication entre les méthodes des classes.

Nous avons exploité une collection d'outils CAD commerciaux comme Catapult C pour générer du code VHDL à partir du code C et l'outil XPS pour la synthèse et la co-simulation ciblant une architecture réconfigurable de Xilinx.

Parmi les avantages des architectures réconfigurables, nous citons notamment la capacité de faire des prototypages le plus tôt possible et de modifier l'architecture matérielle à la demande. Une autre raison est le coût raisonnable de ce type d'architectures.

Nous avons pris en compte la modélisation de haut niveau des adaptateurs incluant les pilotes pour la partie logicielle et des modèles d'enveloppe (wrappers) pour la partie matérielle. Le code des adaptateurs est généré automatiquement en C et intégré dans le corps des méthodes des classes qui communiquent avec les méthodes implémentées en matériel.

Pour valider notre méthodologie, nous avons choisi le standard H264 pour compression/décompression vidéo comme étant une étude de cas. Il s'agit d'une application du traitement intensif avec peu de contrôle.

La quatrième contribution est liée aux aspects spécification et vérification formelle de l'ordonnanceur SystemC. L'objectif de cette contribution est de proposer une sémantique opérationnelle formelle pour l'ordonnanceur SystemC.

La sémantique que nous avons proposé s'appuie essentiellement sur la logique de réécriture et englobe l'aspect statique et dynamique du SystemC.

L'utilisation de la logique de réécriture nous permet de formuler dans le même cadre formel les deux aspects structurel et dynamique.

L'aspect statique est formulé en tant qu'équations algébriques implémentées en langage Maude. En revanche, l'aspect dynamique se formule par le bais des règles de réécriture implémentées en Maude.

Notre choix de langage Maude se justifié par sa puissance d'exprimer la concurrence et l'indéterminisme de façon intuitive et naturelle, sa simplicité et ses capacités pour faire des simulations et des vérifications formelles en un temps raisonnable.

2. Perspectives

Les perspectives envisageables sont :

1. Enrichir les modèles UML par des données plus réalistes concernant le temps d'exécution des actions et des transitions dans les diagrammes d'états transitions et la consommation d'énergie ainsi que les diagrammes d'activité.
2. Raffiner le modèle d'architecture en vue de synthèse et prendre en considération des topologies de communication plus élaborées comme les réseaux sur puce (NOC).
3. Annoter les modèles UML à partir des résultats obtenus au bas niveau (back annotation).
4. Automatiser le partitionnement SW/HW au niveau UML en donnant la main au concepteur de prendre ses décisions de manière interactive vis à vis des solutions qui répondent à ses besoins.
5. Compléter la sémantique opérationnelle proposée pour l'ordonnanceur SystemC et découvrir d'autres propriétés pour vérification formelle.
6. Appliquer les principes de développement dirigé par les modèles (MDD : Model Driven development) pour générer de manière automatique du code écrit en langages spécifiques aux MPSOCs au niveau transactionnel comme SystemC et SystemVerilog ou au niveau RTL comme VHDL et Verilog à partir des modèles UML.
7. Appliquer les principes de la programmation orientée aspect (AOP) pour décorréler les aspects fonctionnels des aspects non fonctionnels.

Bibliographie

[1] I. Ahmed, Y. He, and M. L. Liou. Video compression with parallel processing. In Parallel Computing Journal, Vol. 28, pp.1039-1078, 2002.

[2] P. Alexander, D. Barton, and R. Karnath. System Specification in Rosetta. In Proceedings of IEEE Engineering of Computer Based Systems Symposium (ECBS), 2000.

[3] D. Amyot and A. Eberlein. An Evaluation of Scenario Notations and Construction Approaches for Telecommunication Systems Development. In Telecommunications Systems Journal, 24(1), pp. 61-94, September 2003.

[4] L. Appvrille, M. Waseem,R. Ameur Boulifa, S. Coudert, and R. Pacalet. Abstract application modeling for system design space exploration. Euromicro Conference on Digital System Design (DSD'06), Dubrovnik, Croatia, August 2006.

[5] L. Appvrille, M. Waseem,R. Ameur Boulifa, S. Coudert, and R. Pacalet. A UML-based Environment for System Design Space Exploration. 13th IEEE International Conference on Electronics, Circuits and Systems (ICECS'2006), Nice, France, December 2006.

[6] M. Auguin. Systèmes sur Puce: vers l'exploration en milieu complexe. Ecole Thématique sur les Systèmes Enfouis, I3S, Université de Nice Sophia Antipolis, CNRS, INRIA, Novembre 2003.

[7] F. Balarian and al. Hardware/Software codesign of embedded system, The Polis approach. Kluwer Academic publishers, 1997.

[8] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: An Integrated Electronic System Design Environment. Published by the IEEE Computer Society, April 2003.

[9] I. D. Bates, E. G. Chester, and D. J. Kinniment. A statechart based HW/SW codesign system. In the Proceedings of the seventh international workshop on Hardware/software codesign, Rome, Italy, May 3-5, 1999.

[10] T. Beierlein, D. Frohlich, and B. Steinbach. UML-based co-design for run-time reconfigurable architectures. In Proc. of the Forum on Specification and Design Languages (FDL'03), pp. 5–19, September 2003.

[11] T. Beierlein, D. Frohlich, and B. Steinbach. Model-driven compilation of UML-models for reconfigurable architectures. In Proc. of the Second RTAS Workshop on Model-Driven Embedded Systems (MoDES'04), May 2004.

[12] R. Ben Atitallah, P. Boulet, A. Cuccuru, J.L. Dekeyser, A. Honré, O. Labbani, S. Le Bleu, P. Marquet, E. Piel, J. Taillard, and H. Yu. INRIA. Rapport technique 0342, Gaspard2 UML profile documentation., September 2007.

[13] A. Benveniste and P. Le Guernic. Hybrid Dynamical Systems Theory and the SIGNAL Language. IEEE Tr. on Automatic Control, 35(5), pp. 525-546, May 1990.

[14] G. Berry and G. Gonthier, The Esterel synchronous programming language: Design, semantics, implementation. Science of Computer Programming, 19(2), pp. 87-152, 1992.

[15] M. K. Bhatti and L. Apvrille. Modeling and simulation of SoC hardware Architecture for Design Space Exploration. In SAME 2007 Forum. Academic Posters Session. LaboSOC GET/ENST Paris, Sophia Antipolis, France, October 3-4, 2007.

[16] K. Bondalapati and V. K. Prasanna. Reconfigurable computing systems. Proceedings of the IEEE, 90(7), pp. 1201–1217, July 2002.

[17] G. Booch, J. Rumbaugh, and I. Jacobson. Unified Modeling Language User Guide. Addison-Wesley, 1999.

[18] F. Boutekkouk and M. Benmohammed. A Novel UML2.0-based approach for System On a Chip modeling and Co-design. In VLSI-SOC PhD Forum, Nice, France, 2006.

[19] F. Boutekkouk and M. Benmohammed. An Agent-Based Framework for SOCs Design. In SNIB'06 (Séminaire National en Informatique de Biskra), May 2-4, 2006.

[20] F. Boutekkouk and M. Benmohammed. Analyse dirigée par les scénarios de l'ordonnaçabilité d'une application embarquée temps réel. Dans CGE'05. Ecole militaire polytechnique, Bordj El Bahri, Alger, 16-17 Avril, 2007.

[21] F. Boutekkouk, and M. Benmohammed. Modeling and simulation of embedded systems using UML2.0 and the Y-chart approach. In the fourth workshop on Object-oriented Modeling of Embedded Real Time Systems (OMER4), Paderborn, Germany, October 30-31, 2007.

[22] F. Boutekkouk et M. Benmohammed. Un outil d'aide à la conception des systèmes embarqués en suivant la méthode en Y. Dans MOAD'07, Méthodes et Outils d'Aide à la Décision. Béjaia, Algérie, 18-20 Novembre, 2007.

[23] F. Boutekkouk, S. Bilavarn, M. Auguin, and M. Benmohammed. UML profile for estimating Application Worst Case Execution Time on System-On-Chip. In the International Symposium on System-on-Chip (SOC2008). Tampere, Finland, November 5-6, 2008.

[24] F. Boutekkouk and M. Benmohammed. Maude-based design space exploration of embedded systems. In the 20th International Conference in Microelectronics (ICM'08), University of Sharjah. Sharjah, UAE, December 14-17, 2008.

[25] F. Boutekkouk, M. Benmohammed, S. Bilavarn, and M. Auguin. UML2.0 profiles for Embedded Systems and Systems On a Chip (SOCs). In JOT (Journal of Object Technology), January 2009.

[26] F. Boutekkouk and M. Benmohammed. Control/Data Driven Embedded Systems High Level Modeling, Formal Verification, and Simulation. In IRECOS (International Review on Computers and Software), Vol. 4, N. 1, January 2009.

[27] F. Boutekkouk, M. Benmohammed, S. Bilavarn, and M. Auguin. Formal semantics for SystemC scheduler. In IRECOS (International Review on Computers and Software), Vol. 4, N. 1, Mars 2009.

[28] F. Boutekkouk, M. Benmohammed, S. Bilavarn, and M. Auguin. UML for modelling and performance estimation of embedded systems. In JOT (Journal of Object Technology), Mars 2009.

[29] F. Boutekkouk and M. Benmohammed. UML Modeling and Formal Verification of control/data driven Embedded Systems. In the 14th International Conference on Engineering of Complex Systems, Potsdam, Germany, June 2009.

[30] T. J. Callahan and J. Wawrzynek. Instruction-level parallelism for reconfigurable computing. In Proc. of the 8th International Workshop on Field-Programmable Logic and Applications, From FPGAs to Computing Paradigm, pp. 248–257. Springer-Verlag, Berlin, August 1998.

[31] D. Cansell, J.-F. Culat, D. Méry, and C. Proch. Derivation of SystemC code from abstract system models. In Forum on specification & Design Languages - FDL'04, Lille, France, September 2004.

[32] D. Cansell, D. Méry, and C. Proch. Modelling SystemC Scheduler by refinement. In IEEE ISOLA workshop on Leveraging Applications of Formal Methods, Verification, and Validation (ISOLA'05), Columbia, U.S.A, 2005.

[33] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: A Declarative Language for Programming Synchronous Systems. Conference Record of the 14th Annual ACM Symp. on Principles of Programming Languages, Munich, Germany, January 1987.

[34] CatapultC, http://www.mentor.com/products/esl/high_level_synthesis/.

[35] C. Chavet. Synthèse automatique d'interfaces de communication matérielles pour la conception d'applications du domaine du traitement du signal. Thèse de l'université de Bretagne Sud. N° d'ordre 94, Octobre 2007.

[36] R. Chen, M. Sgroi, L. Lavagno, G. Martin, A. Sangiovanni-Vincentelli, and J. Rabaey. UML AND PLATFORM-BASED DESIGN. In "UML for Real", Edited by B. Selic, L. Lavagno, G. Martin, pp. 107-126, Kluwer Academic Publishers, May 2003.

[37] M. Clavel. Maude: Specifcation and programming in rewriting logic. Internal report, SRI International, 1999.

[38] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and C. Talcott. Maude Manual, version 2.3, January 2007.

[39] K. Compton and S. Hauck. Reconfigurable computing: A survey of systems and software. ACM Computing Surveys, 34(2), pp.171–210, June 2002.

[40] P. Coste, F. Hessel, P. Le Marrec, Z. Sugar, M. Romdhani, R. Suescun, N. Zergainoh, and A.A. Jerraya. Multilanguage design of Heterogeneous system. In Proceedings of the seventh international workshop on Hardware/software codesign, Rome, Italy, pp. 54-58, 1999.

[41] P. Coste, F. Hessel, and A.A. Jerraya. Multilanguage Codesign Using SDL and Matlab. In *SASIMI'00*, Kyoto, Japan, April 2000.

[42] DaRT. Dataparallelism for Real-time futurs. INRIA. Theme 1C. Activity Report, 2003.

[43] J.M. Daveau. Spécification système et synthèse de la communication pour le Co-Design Logiciel/Matériel. Ph.D. Thesis INPG, TIMA Laboratory, Decembre, 1997.

[44] G. de Jong. A UML-Based Design Methodology for Real-Time and Embedded Systems. In Proceedings of the Design Automation and Test in Europe Conference (DATE'02), Paris, France, 2002.

[45] C. Dorotska, D. Frohlich, and B. Steinbach. Synthesis of UML-Models for Reconfigurable Hardware. In proceeding, 2nd UML for SoC Design Workshop at 42nd Design Automation Conference (DAC), Anaheim, California, 2005.

[46] J. Dunagan, K. Trivedi, R. Geist and V. Nicola. Extended Stochastic Petri nets: Applications and analysis. Proc. Performance 84, Paris (France), pp. 507-519, December 1984,

[47] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli. Design of embedded systems: formal models, validation, and synthesis. In Proceedings of the IEEE, 85(3), pp. 366-390, March 1997.

[48] R. Ernst, J. Henkel, and T. Benner. Hardware-Software Cosynthesis for Microcontrollers, IEEE Journal Design and Test of Computers, pp. 64-75, December 1993.

[49] J. Fellmuth, P. Herber, and S. Glesner. Model Checking SystemC Designs Using Timed Automata. Embedded systems week, Atlanta, Georgia, USA. October 19-24, 2008.

[50] J. M. Fernandes, R.J. Machado, and H.D. Santos. Modeling industrial Embedded Systems with UML. Reconfigurable embedded systems : development methodologies for real-time applications, 2000.

[51] J. Fleischmann, K. Buchenrieder, and R. Kress. A hardware/software prototyping environment for dynamically reconfigurable embedded systems. In Proc. of the 6th International Workshop on Hardware/software Codesign, IEEE Computer Society, pp. 105–109, March 1998.

[52] J. Fleischmann, K. Buchenrieder, and R. Kress. Java driven codesign and prototyping of networked embedded systems. In Proc. of the 36th ACM/IEEE Design Automation Conference (DAC'99), ACM Press, pp. 794–797, June 1999.

[53] W. Fornaciari, P. Gubian, D. Sciuto, and C. Silvano. Power Estimation of Embedded Systems: a Hardware/Software Codesign approach. In IEEE transactions on VLSI Systems, 6(2), June 1998.

[54] D. Frohlich. Object-Oriented Development for Reconfigurable Architectures. Dissertation. Von der Fakultat fur Mathematik und Informatic. Der Technischien Universitat Bergakademie Freiberg. June 2007.

[55] Functional specification for SystemC 2.0. www.systemc.org.

[56] D.D. Gajski, R. Doemer, and J. Zhu. IP-Centric Methodology and Design With the SpecC Language. In System Level Design of Embedded Systems. Irvine, California, USA, 1998.

[57] D.D. Gajski, F. vahid, S. Narayan, and J. Gong. Specification and Design of Embedded Systems. Published by Prentice Hall. Englewood, Newjersey 07632, 1994.

[58] D.D. Gajski, F. Vahid, S. Narayan, and J. Gong. SpecSyn: An environment supporting the specify-explore-refine paradigm for hw/sw system design. IEEE transactions on VLSI Systems, 6(1), pp. 84-100, 1998.

[59] J. Gary, P. Topiwala, and A. Luthra. The H.264/AVC Advanced Video Coding Standard: Overview and Introduction to the Fidelity Range Extensions. Presented at the SPIE Conference on Applications of Digital Image Processing XXVII Special Session on Advances in the New Emerging Standard: H.264/AVC, August, 2004.

[60] A.Gerbi and k. Ferhat. UML Profiles for Real-Time Systems and their Applications. Journal paper. In JOT, 5(4), pp. 149-169, May-June 2006.

[61] A. Ghosh, J. Kunkel, and S. Liao. Hardware Synthesis from C/C++. In Proceedings of the DATE99 conference, pp.382-383, March 1999.

[62] M. Gordon, W. Thies, and S. Amarasinghe. Exploiting Coarse Grained Task, Data, and Pipeline Parallelism in Stream Programs. ASPLOS06, San Jose, California, USA, October 21-25, 2006.

[63] P. Green, M. Edwards, and S. Essa. UML for System-Level Design. In Forum on Design Languages, Proceedings of FDL 2001, Lyon, France, September 3-7, 2001.

[64] P. Green and M. Edwards. The modelling of embedded systems using HASOC. Department of computation, UMIST, manchesterm UK, 2001.

[65] R. Gupta and G. De Micheli. Hardware-Software Cosynthesis for Digital Systems. In IEEE Journal Design and Test of Computers, pp. 29-41, September 1993.

[66] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and Verifying Real- Time Systems by Means of the Synchronous Data-Flow Language LUSTRE. In IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems, 18(9), September 1992.

[67] N. Harman. Verifying a simple Pipelined Microprocessor Using Maude. In LNCS ISSN 0302-9743, Vol. 2267, pp. 128-151, 2005.

[68] R. Hartenstein. A Decade of Reconfigurable Computing: A Visionary Perspective. In *Proc.* DATE '01, Munchen, March 13-16, 2001.

[69] C. Hoare. Communicating Sequential Processes. Prentice Hall, 1985.

[70] IEEE Standard SystemC® language Reference Manual. www.systemc.org.

[71] ITEA. Information Technology For European Advancement. MARTES. Model-Based Approach for Real-Time Embedded Systems development. Title: Current limitations of best practices. Deliverable ID: 1.1, Version: 1.0. Editor Kari Tiensyrja. Status: Final. Confidentiality: Public. March 31, 2006.

[72] A.A. Jerraya, and W. Wolf. Multiprocessor systems on chip. Morgan Kaufmann publishers 2005.

[73] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In Proc. of the IFIP Congress 74, North-Holland Publishing Co., 1974.

[74] T. Kangas, P. Kukkala, H. Orsila, E. Saminen, M. Hannikainen, and T.D. Hamalainen. UML-Based Multiprocessor SOC Design. In ACM transactions on Embedded computing Systems, 5(2), pp. 281-320, May 2006.

[75] I. Karkowski and H. Corpooral. Design space exploration algorithm for heterogeneous multi processor embedded system design. In DAC'98, San Francisco, CA, June 1998.

[76] M. Katelman and J. Meseguer. A rewriting semantics for ABEL with Application to Hardware/Software Co-design and analysis. In ENTCS 176(4), July 2007.

[77] K. Keutzer, S. Malik, R. Newton, j. Rabaey, and A. Sangiovanni-Vincentelli. Sytem level design: orthgonalization of concerns and Platform-Based Design. In IEEE transactions on computer-aided design of circuits and systems, 19(12) ,December 2000.

[78] B. Kienhuis, Ed F. Deprettere, P.V. Wolf, and K. Vissers. A Methodology to design Programmable Embedded Systems. The Y-chart approach. In LNCS series vol. 2268, by Springer Verlag ©, pp.18-37, 2001.

[79] B. Kienhuis, E. Rijpkema, and E. Deprettere. Compaan: deriving process networks from Matlab for embedded signal processing architectures. International Conference on Hardware Software Codesign, Proceedings of the eighth international workshop on Hardware/software codesign, San Diego, California, pp.13-17, May 3-5, 2000.

[80] S. Kimura, M. Yukishita, Y. Itou, A. Nagoya, M. Hirao, and K.Watanabe. A hardware/software codesign method for a general purpose reconfigurable co-processor. In Proc. of the 5th International Workshop on Hardware/Software Co-design (CODES/CASHE'97), IEEE Computer Society, pp. 147–151, March 1997.

[81] D. Kirovski and M. Potkonjak. System level synthesis of low power hard real time systems. In Design Automation Conference (DAC'97), Anaheim, June 1997.

[82] P. Kukkala, J. Riihimaki, M. Hannikainen, T.D. Hamalainen, and K.Kronlof. UML2.0 Profile for Embedded System Design. In Proceedings of the Design, Automation and Test in Europe Conference end Exhibition (DATE'05), 2005.

[83] L. Lagadec. Abstraction, modélisation et outils de CAO pour les architectures réconfigurables. Ph.D. Thesis, Université Renne 1, Décembre 2000.

[84] E.A. Lee and A. Sangiovanni-Vincentelli. A Framework for Comparing Models of Computation. In IEEE Transactions on computer-aided design of integrated circuits and systems, 17(12), December 1998.

[85] E. A. Lee. Computing for Embedded Systems. IEEE Instrumentation and Measurement, Technology Conference, Budapest, Hungary, May 21-23, 2001.

[86] E.A. Lee and T. M. Parks. Dataflow Process Networks. In Proceedings of the IEEE, 83(5), pp. 773-801, May 1995.

[87] E.A. Lee and D. G. Messerschmitt. Synchronous Data Flow. In Proceedings of the IEEE, 75(9), September 1987.

[88] E.A. Lee and D. G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. IEEE Trans. on Computers, January 1987.

[89] E.A. Lee and A. Sangiovanni-Vincentelli. Comparing Models of Computation . Proceedings of ICCAD, pp.10-14, November 1996.

[90] Y. Le Moullec, J. P. Diguet, and J. L. Philippe. Design-Trotter: a Multimedia Embedded Systems Design Space Exploration Tool. IEEE Workshop on Multimedia Signal Processing (MMSP'02), St. Thomas, US Virgin Islands, December 9-11, 2002.

[91] J. Liu, X. Liu, T-K J. Koo, B. Sinopoli, S. Sastry, and E.A. Lee. A Hierarchical Hybrid System Model and Its Simulation. Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720 Availlale on line at http ://ptolemy.eecs.berkeley.edu/papers/.

[92] X. Liu, J. Liu, J. Eker, and E.A. Lee. Heterogeneous Modeling and Design of Control Systems. Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720 Software-Enable Control : Information technology for Dynamical Systems Tariq Samad and Garry Balas (eds.), Wiley-IEEE Press, April 2003.

[93] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood. Hardware-software co-design of embedded reconfigurable architectures. In Proc. of the 37th ACM/IEEE Design Automation Conference (DAC'00), IEEE Computer Society, pp. 507–512, June 2000.

[94] D. Lyonnard, S. Yoo, A. Baghdadi, and A. Jerraya. Automatic Generation of Application-Specific Architectures for Heterogeneous MPSoC through Combination of Processors. SLS Group, TIMA Laboratory Colloque CAO 2002, Paris, France, 2002.

[95] P. Magarshack and P.G. Paulin. System-on-Chip Beyond the Nanometer Wall. In DAC'03, Anaheim, California, USA. June 2-6, 2003.

[96] F. Maraninchi. The Argos Language: Graphical Representation of Automata and Description of Reactive Systems. In Proc. of the IEEE Workshop on Visual Languages, Kobe, Japan, October 1991.

[97] G. Martin, L. Lavagno, and J. Louis –Guerin. Embedded UML: a merger of real-time UML and co-design. In Proceedings of CODES 2001, Copenhagen, pp.23-28, April 2001.

[98] G. Martin. UML for embedded systems specification and design: motivation and overview. Cadence design systems, 2002.

[99] T. McCombs. Maude 2.0 Primer, Version 1.0. International report. SRI. International 2003.

[100] S.J. Mellor, J.R. Wolf, and C. McCausland. Why Systems-on-Chip Needs More UML like a Hole in the Head. In Proceedings of the Design, Automation and Test in Europe (DATE'05), 2005.

[101] J. Meseguer. Rewriting as a unified model of concurrency. In proceedings of the Concurr'90 Conference, Springer LNCS Vol. 458, Amsterdam, pp. 384-400, 1990.

[102] W. Mueller, R. Domer, and A. Gerstlauer. The formal execution semantics of SpecC. In ISSS '02: Proceedings of the 15th international symposium on System Synthesis ACM Press, New York, USA, pp.150-155, 2002.

[103] T. Murata. Petri Nets: Properties, analysis and applications. Proc. IEEE, 77(4), pp. 541-580, April 1989.

[104] N. Narasimhan, V. Srinivasan, M. Vootukuru, J. Walrath, S. Govindarajan, and R. Vemuri. Rapid prototyping of reconfigurable coprocessors. In Proc. of the International Conference on Application Specific Systems, Architectures, and Processors (ASAP), IEEE Press, pp.303–312, August 1996.

[105] K.D. Nguyen, Z. Sun, P. S. Thiagarajan, and W. F. Wong. Model-driven SoC design via executable UML to SystemC. In Proc. of the 25th IEEE International Real-Time Systems Symposium (RTSS'04), IEEE Computer Society, pp. 459–468, December 2004.

[106] PTOLEMY II, Heterogeneous Concurrent Modeling And Design In Java, Volume 1: Introduction To Ptolemy II, University of California at Berkeley, July 16, 2003.

[107] Ptolemy Project, http://Ptolemy.eecs.berkely.edu.

[108] Rhapsody case tool reference manual. I-Logix Inc. http://www.ilogix.com.

[109] E.Riccobene, P. Scandura, A. Rosti, and S. Bocchino. A SOC Design Methodology Involving a UML2.0 Profile for SystemC. In Proceedings of the Design, Automation and Test in Europe Conference end Exhibition (DATE'05), 2005.

[110] J. Ruf, D. Hoffmann, J. Gerlach, T. Kropf, W. Rosenstiehl, and W. Mueller. The simulation semantics of SystemC. In DATE'01, Proceedings of the conference on Design, automation and test in Europe, IEEE Press, pp. 64-70, 2001.

[111] J. Ruf, D. Hoffmann, T. Kropf, and W. Rosenstiel. Simulation-guided property checking based on a multi-valued AR-automata. In DATE'01, Proceedings of the conference on Design, automation and test in Europe, IEEE Press, pp. 742-748, 2001.

[112] A. Salem. Formal semantics of synchronous SystemC. In DATE'03, Proceedings of the conference on Design, Automation and Test in Europe, IEEE Computer Society, pp. 376-381, 2003.

[113] T. Schattkowsky. UML2.0 Overview and Perspectives in SOC Design. In Proceedings of the Design, Automation and Test in Europe (DATE'05), 2005.

[114] J. Sifakis. Use of Petri nets for performance evaluation. In Measuring, Modelling and Evaluating Computer Systems, ed. H. Beilmer and E. Gelenbe. North Holland, 1977.

[115] S. Stuijk. Predictable Mapping of Streaming Applications on Multiprocessors. PhD thesis. Ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, October 25, 2007.

[116] F. Symons. Introduction to numerical Petri Nets, a general graphical model of concurrent processing systems. Australian Telecomm. Res., Vol. 14, January 1980.

[117] SystemC, Version 2.0, User's Guide. www.systemc.org.

[118] Systems Modeling Language (SysML) Specification. OMG document: ad/2006-03-08-01, version 1. Draft, April 2006.

[119] SystemVerilog 3.1 - Accellera's Extensions to Verilog, http://www.eda.org/sv/SystemVerilog_3.1_final.pdf

[120] J. Teich, T. Blickle, and L. Thiele. An evolutionary approach to system level synthesis. In Codes/CASHE'97, Braunschweig, Germany, March 1997.

[121] UML Profile for MARTE, Beta 1. OMG Adopted Specification, ptc/07-08-04, August 2007.

[122] UML Profile for System on a Chip (SOC). OMG Available Specification, version 1.0.1 formal /06-08-01, August 2006.

[123] UML Profile for MARTE, Beta 1. OMG Adopted Specification, ptc/07-08-04, August 2007.

[124] UML Profile for System on a Chip (SOC). OMG Available Specification, version 1.0.1 formal /06-08-01, August 2006.

[125] F. Vahid. Modifying Min-Cut for hardware and software functional partitioning. In Workshop on Hardware/Software Codesign, Braunschweig, Germany, 1997.

[126] W. Van der Aalst. Interval timed coloured petri nets and their analysis. Application and theory of Petri Nets 1993, Proceedings 14th International Conference, Chicago, Illinois, USA, vol. 691, pp. 453-472, 1993.

[127] Y. Vanderperren, and W. Dehaene. SysML and Systems Engineering Applied to UML-Based SOC Design. In Proc. 2nd UML-SOC Workshop at 42nd DAC, Anaheim (CA), USA, 2005.

[128] D. Verkest, J. Cockx, F. Potargent, G. De Jong Alcatel, F. Wellesplein and H. De Man. On the use of C++ for system-on-chip design. 1998.

[129] A. Viehl, O. Bringmann, and W. Rosentiel. Performance Analysis of Sequence Diagrams for SOC design. In proceeding, 2nd UML for SoC Design Workshop at 42nd Design Automation Conference (DAC), Anaheim, California, 2005.

[130] W. Wolf. The Future of Multiprocessor Systems-on-Chips, DAC 2004, San Diego, California, USA, June 7-11, 2004.

[131] Xilinx, Virtex-4 Family Overview, Advanced Product Sprcification, Data Sheet DS 112 (v1.1), September 10, 2004.

[132] Xilinx, Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet, Product Specification, June 2004.

[133] N. E. Zergainoh, G. F. Marchioro, J. M. Daveau, and A.A. Jerraya. Using SDL for Hardware/ Software Co-design of an ATM Network Interface Card. 1st Workshop of the SDL Forum Society an SDL and MSC, Berlin, June 1998.

[134] Q. Zhu, A. Matsuda, S. Kuwamura, T. Nakata, and M. Shoji. An object-oriented design process for system-on-chip using UML. In Proc. of the 15th International Symposium on System Synthesis (ISSS'02), ACM Press, pp. 249–254, October 2002.

[135] Q. Zhu, R. Oishi, T. Hesegawa, and T. Nakata. Integrating UML into SOC Design Process. In Proceedings of the Design, Automation and Test in Europe (DATE'05), 2005.

Annexe

Code VHDL de différents composants générés par XPS.

```
------------------------------------------------------------------------------
----

-- inverse_transform_0_to_microblaze_0_wrapper.vhd

------------------------------------------------------------------------------
----

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

library UNISIM;
use UNISIM.VCOMPONENTS.ALL;

library fsl_v20_v2_10_a;
use fsl_v20_v2_10_a.All;

entity inverse_transform_0_to_microblaze_0_wrapper is
  port (
    FSL_Clk : in std_logic;
    SYS_Rst : in std_logic;
    FSL_Rst : out std_logic;
    FSL_M_Clk : in std_logic;
    FSL_M_Data : in std_logic_vector(0 to 31);
    FSL_M_Control : in std_logic;
    FSL_M_Write : in std_logic;
    FSL_M_Full : out std_logic;
    FSL_S_Clk : in std_logic;
    FSL_S_Data : out std_logic_vector(0 to 31);
    FSL_S_Control : out std_logic;
    FSL_S_Read : in std_logic;
    FSL_S_Exists : out std_logic;
    FSL_Full : out std_logic;
    FSL_Has_Data : out std_logic;
    FSL_Control_IRQ : out std_logic
  );
end inverse_transform_0_to_microblaze_0_wrapper;

architecture STRUCTURE of inverse_transform_0_to_microblaze_0_wrapper is

  component fsl_v20 is
    generic (
      C_EXT_RESET_HIGH : integer;
      C_ASYNC_CLKS : integer;
      C_IMPL_STYLE : integer;
      C_USE_CONTROL : integer;
      C_FSL_DWIDTH : integer;
      C_FSL_DEPTH : integer
    );
    port (
      FSL_Clk : in std_logic;
      SYS_Rst : in std_logic;
      FSL_Rst : out std_logic;
      FSL_M_Clk : in std_logic;
```

```
      FSL_M_Data : in std_logic_vector(0 to C_FSL_DWIDTH-1);
      FSL_M_Control : in std_logic;
      FSL_M_Write : in std_logic;
      FSL_M_Full : out std_logic;
      FSL_S_Clk : in std_logic;
      FSL_S_Data : out std_logic_vector(0 to C_FSL_DWIDTH-1);
      FSL_S_Control : out std_logic;
      FSL_S_Read : in std_logic;
      FSL_S_Exists : out std_logic;
      FSL_Full : out std_logic;
      FSL_Has_Data : out std_logic;
      FSL_Control_IRQ : out std_logic
    );
  end component;

  attribute x_core_info : STRING;
  attribute x_core_info of fsl_v20 : component is "fsl_v20_v2_10_a";

begin

  inverse_transform_0_to_microblaze_0 : fsl_v20
    generic map (
      C_EXT_RESET_HIGH => 0,
      C_ASYNC_CLKS => 0,
      C_IMPL_STYLE => 0,
      C_USE_CONTROL => 1,
      C_FSL_DWIDTH => 32,
      C_FSL_DEPTH => 16
    )
    port map (
      FSL_Clk => FSL_Clk,
      SYS_Rst => SYS_Rst,
      FSL_Rst => FSL_Rst,
      FSL_M_Clk => FSL_M_Clk,
      FSL_M_Data => FSL_M_Data,
      FSL_M_Control => FSL_M_Control,
      FSL_M_Write => FSL_M_Write,
      FSL_M_Full => FSL_M_Full,
      FSL_S_Clk => FSL_S_Clk,
      FSL_S_Data => FSL_S_Data,
      FSL_S_Control => FSL_S_Control,
      FSL_S_Read => FSL_S_Read,
      FSL_S_Exists => FSL_S_Exists,
      FSL_Full => FSL_Full,
      FSL_Has_Data => FSL_Has_Data,
      FSL_Control_IRQ => FSL_Control_IRQ
    );

end architecture STRUCTURE;

------------------------------------------------------------------------------
----
-- inverse_transform_0_wrapper.vhd
------------------------------------------------------------------------------
----
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

library UNISIM;
use UNISIM.VCOMPONENTS.ALL;
```

```
library inverse_transform;
use inverse_transform.All;

entity inverse_transform_0_wrapper is
  port (
    FSL_Clk : in std_logic;
    FSL_Rst : in std_logic;
    FSL_S_Clk : out std_logic;
    FSL_S_Read : out std_logic;
    FSL_S_Data : in std_logic_vector(0 to 31);
    FSL_S_Control : in std_logic;
    FSL_S_Exists : in std_logic;
    FSL_M_Clk : out std_logic;
    FSL_M_Write : out std_logic;
    FSL_M_Data : out std_logic_vector(0 to 31);
    FSL_M_Control : out std_logic;
    FSL_M_Full : in std_logic
  );
end inverse_transform_0_wrapper;

architecture STRUCTURE of inverse_transform_0_wrapper is

  component inverse_transform is
    port (
      FSL_Clk : in std_logic;
      FSL_Rst : in std_logic;
      FSL_S_Clk : out std_logic;
      FSL_S_Read : out std_logic;
      FSL_S_Data : in std_logic_vector(0 to 31);
      FSL_S_Control : in std_logic;
      FSL_S_Exists : in std_logic;
      FSL_M_Clk : out std_logic;
      FSL_M_Write : out std_logic;
      FSL_M_Data : out std_logic_vector(0 to 31);
      FSL_M_Control : out std_logic;
      FSL_M_Full : in std_logic
    );
  end component;

  attribute x_core_info : STRING;
  attribute x_core_info of inverse_transform : component is
"inverse_transform_v";

begin

  inverse_transform_0 : inverse_transform
    port map (
      FSL_Clk => FSL_Clk,
      FSL_Rst => FSL_Rst,
      FSL_S_Clk => FSL_S_Clk,
      FSL_S_Read => FSL_S_Read,
      FSL_S_Data => FSL_S_Data,
      FSL_S_Control => FSL_S_Control,
      FSL_S_Exists => FSL_S_Exists,
      FSL_M_Clk => FSL_M_Clk,
      FSL_M_Write => FSL_M_Write,
      FSL_M_Data => FSL_M_Data,
      FSL_M_Control => FSL_M_Control,
      FSL_M_Full => FSL_M_Full
    );
```

```
end architecture STRUCTURE;



-------------------------------------------------------------------------------
----
-- microblaze_0_to_inverse_transform_0_wrapper.vhd
-------------------------------------------------------------------------------
----
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

library UNISIM;
use UNISIM.VCOMPONENTS.ALL;

library fsl_v20_v2_10_a;
use fsl_v20_v2_10_a.All;

entity microblaze_0_to_inverse_transform_0_wrapper is
  port (
    FSL_Clk : in std_logic;
    SYS_Rst : in std_logic;
    FSL_Rst : out std_logic;
    FSL_M_Clk : in std_logic;
    FSL_M_Data : in std_logic_vector(0 to 31);
    FSL_M_Control : in std_logic;
    FSL_M_Write : in std_logic;
    FSL_M_Full : out std_logic;
    FSL_S_Clk : in std_logic;
    FSL_S_Data : out std_logic_vector(0 to 31);
    FSL_S_Control : out std_logic;
    FSL_S_Read : in std_logic;
    FSL_S_Exists : out std_logic;
    FSL_Full : out std_logic;
    FSL_Has_Data : out std_logic;
    FSL_Control_IRQ : out std_logic
  );
end microblaze_0_to_inverse_transform_0_wrapper;

architecture STRUCTURE of microblaze_0_to_inverse_transform_0_wrapper is

  component fsl_v20 is
    generic (
      C_EXT_RESET_HIGH : integer;
      C_ASYNC_CLKS : integer;
      C_IMPL_STYLE : integer;
      C_USE_CONTROL : integer;
      C_FSL_DWIDTH : integer;
      C_FSL_DEPTH : integer
    );
    port (
      FSL_Clk : in std_logic;
      SYS_Rst : in std_logic;
      FSL_Rst : out std_logic;
      FSL_M_Clk : in std_logic;
      FSL_M_Data : in std_logic_vector(0 to C_FSL_DWIDTH-1);
      FSL_M_Control : in std_logic;
      FSL_M_Write : in std_logic;
      FSL_M_Full : out std_logic;
      FSL_S_Clk : in std_logic;
      FSL_S_Data : out std_logic_vector(0 to C_FSL_DWIDTH-1);
```

```
      FSL_S_Control : out std_logic;
      FSL_S_Read : in std_logic;
      FSL_S_Exists : out std_logic;
      FSL_Full : out std_logic;
      FSL_Has_Data : out std_logic;
      FSL_Control_IRQ : out std_logic
    );
  end component;

  attribute x_core_info : STRING;
  attribute x_core_info of fsl_v20 : component is "fsl_v20_v2_10_a";

begin

  microblaze_0_to_inverse_transform_0 : fsl_v20
    generic map (
      C_EXT_RESET_HIGH => 0,
      C_ASYNC_CLKS => 0,
      C_IMPL_STYLE => 0,
      C_USE_CONTROL => 1,
      C_FSL_DWIDTH => 32,
      C_FSL_DEPTH => 16
    )
    port map (
      FSL_Clk => FSL_Clk,
      SYS_Rst => SYS_Rst,
      FSL_Rst => FSL_Rst,
      FSL_M_Clk => FSL_M_Clk,
      FSL_M_Data => FSL_M_Data,
      FSL_M_Control => FSL_M_Control,
      FSL_M_Write => FSL_M_Write,
      FSL_M_Full => FSL_M_Full,
      FSL_S_Clk => FSL_S_Clk,
      FSL_S_Data => FSL_S_Data,
      FSL_S_Control => FSL_S_Control,
      FSL_S_Read => FSL_S_Read,
      FSL_S_Exists => FSL_S_Exists,
      FSL_Full => FSL_Full,
      FSL_Has_Data => FSL_Has_Data,
      FSL_Control_IRQ => FSL_Control_IRQ
    );

end architecture STRUCTURE;


------------------------------------------------------------------------------
----
-- microblaze_0_wrapper.vhd
------------------------------------------------------------------------------
----
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

library UNISIM;
use UNISIM.VCOMPONENTS.ALL;

library microblaze_v6_00_b;
use microblaze_v6_00_b.All;

entity microblaze_0_wrapper is
  port (
```

```
CLK : in std_logic;
RESET : in std_logic;
INTERRUPT : in std_logic;
DEBUG_RST : in std_logic;
EXT_BRK : in std_logic;
EXT_NM_BRK : in std_logic;
DBG_STOP : in std_logic;
MB_Halted : out std_logic;
INSTR : in std_logic_vector(0 to 31);
I_ADDRTAG : out std_logic_vector(0 to 3);
IREADY : in std_logic;
IWAIT : in std_logic;
INSTR_ADDR : out std_logic_vector(0 to 31);
IFETCH : out std_logic;
I_AS : out std_logic;
DATA_READ : in std_logic_vector(0 to 31);
DREADY : in std_logic;
DWAIT : in std_logic;
DATA_WRITE : out std_logic_vector(0 to 31);
DATA_ADDR : out std_logic_vector(0 to 31);
D_ADDRTAG : out std_logic_vector(0 to 3);
D_AS : out std_logic;
READ_STROBE : out std_logic;
WRITE_STROBE : out std_logic;
BYTE_ENABLE : out std_logic_vector(0 to 3);
DM_ABUS : out std_logic_vector(0 to 31);
DM_BE : out std_logic_vector(0 to 3);
DM_BUSLOCK : out std_logic;
DM_DBUS : out std_logic_vector(0 to 31);
DM_REQUEST : out std_logic;
DM_RNW : out std_logic;
DM_SELECT : out std_logic;
DM_SEQADDR : out std_logic;
DOPB_DBUS : in std_logic_vector(0 to 31);
DOPB_ERRACK : in std_logic;
DOPB_MGRANT : in std_logic;
DOPB_RETRY : in std_logic;
DOPB_TIMEOUT : in std_logic;
DOPB_XFERACK : in std_logic;
IM_ABUS : out std_logic_vector(0 to 31);
IM_BE : out std_logic_vector(0 to 3);
IM_BUSLOCK : out std_logic;
IM_DBUS : out std_logic_vector(0 to 31);
IM_REQUEST : out std_logic;
IM_RNW : out std_logic;
IM_SELECT : out std_logic;
IM_SEQADDR : out std_logic;
IOPB_DBUS : in std_logic_vector(0 to 31);
IOPB_ERRACK : in std_logic;
IOPB_MGRANT : in std_logic;
IOPB_RETRY : in std_logic;
IOPB_TIMEOUT : in std_logic;
IOPB_XFERACK : in std_logic;
DBG_CLK : in std_logic;
DBG_TDI : in std_logic;
DBG_TDO : out std_logic;
DBG_REG_EN : in std_logic_vector(0 to 4);
DBG_CAPTURE : in std_logic;
DBG_UPDATE : in std_logic;
Trace_Instruction : out std_logic_vector(0 to 31);
Trace_Valid_Instr : out std_logic;
```

```
Trace_PC : out std_logic_vector(0 to 31);
Trace_Reg_Write : out std_logic;
Trace_Reg_Addr : out std_logic_vector(0 to 4);
Trace_MSR_Reg : out std_logic_vector(0 to 10);
Trace_New_Reg_Value : out std_logic_vector(0 to 31);
Trace_Exception_Taken : out std_logic;
Trace_Exception_Kind : out std_logic_vector(0 to 3);
Trace_Jump_Taken : out std_logic;
Trace_Delay_Slot : out std_logic;
Trace_Data_Address : out std_logic_vector(0 to 31);
Trace_Data_Access : out std_logic;
Trace_Data_Read : out std_logic;
Trace_Data_Write : out std_logic;
Trace_Data_Write_Value : out std_logic_vector(0 to 31);
Trace_Data_Byte_Enable : out std_logic_vector(0 to 3);
Trace_DCache_Req : out std_logic;
Trace_DCache_Hit : out std_logic;
Trace_ICache_Req : out std_logic;
Trace_ICache_Hit : out std_logic;
Trace_OF_PipeRun : out std_logic;
Trace_EX_PipeRun : out std_logic;
Trace_MEM_PipeRun : out std_logic;
FSL0_S_CLK : out std_logic;
FSL0_S_READ : out std_logic;
FSL0_S_DATA : in std_logic_vector(0 to 31);
FSL0_S_CONTROL : in std_logic;
FSL0_S_EXISTS : in std_logic;
FSL0_M_CLK : out std_logic;
FSL0_M_WRITE : out std_logic;
FSL0_M_DATA : out std_logic_vector(0 to 31);
FSL0_M_CONTROL : out std_logic;
FSL0_M_FULL : in std_logic;
FSL1_S_CLK : out std_logic;
FSL1_S_READ : out std_logic;
FSL1_S_DATA : in std_logic_vector(0 to 31);
FSL1_S_CONTROL : in std_logic;
FSL1_S_EXISTS : in std_logic;
FSL1_M_CLK : out std_logic;
FSL1_M_WRITE : out std_logic;
FSL1_M_DATA : out std_logic_vector(0 to 31);
FSL1_M_CONTROL : out std_logic;
FSL1_M_FULL : in std_logic;
FSL2_S_CLK : out std_logic;
FSL2_S_READ : out std_logic;
FSL2_S_DATA : in std_logic_vector(0 to 31);
FSL2_S_CONTROL : in std_logic;
FSL2_S_EXISTS : in std_logic;
FSL2_M_CLK : out std_logic;
FSL2_M_WRITE : out std_logic;
FSL2_M_DATA : out std_logic_vector(0 to 31);
FSL2_M_CONTROL : out std_logic;
FSL2_M_FULL : in std_logic;
FSL3_S_CLK : out std_logic;
FSL3_S_READ : out std_logic;
FSL3_S_DATA : in std_logic_vector(0 to 31);
FSL3_S_CONTROL : in std_logic;
FSL3_S_EXISTS : in std_logic;
FSL3_M_CLK : out std_logic;
FSL3_M_WRITE : out std_logic;
FSL3_M_DATA : out std_logic_vector(0 to 31);
FSL3_M_CONTROL : out std_logic;
```

```
FSL3_M_FULL : in std_logic;
FSL4_S_CLK : out std_logic;
FSL4_S_READ : out std_logic;
FSL4_S_DATA : in std_logic_vector(0 to 31);
FSL4_S_CONTROL : in std_logic;
FSL4_S_EXISTS : in std_logic;
FSL4_M_CLK : out std_logic;
FSL4_M_WRITE : out std_logic;
FSL4_M_DATA : out std_logic_vector(0 to 31);
FSL4_M_CONTROL : out std_logic;
FSL4_M_FULL : in std_logic;
FSL5_S_CLK : out std_logic;
FSL5_S_READ : out std_logic;
FSL5_S_DATA : in std_logic_vector(0 to 31);
FSL5_S_CONTROL : in std_logic;
FSL5_S_EXISTS : in std_logic;
FSL5_M_CLK : out std_logic;
FSL5_M_WRITE : out std_logic;
FSL5_M_DATA : out std_logic_vector(0 to 31);
FSL5_M_CONTROL : out std_logic;
FSL5_M_FULL : in std_logic;
FSL6_S_CLK : out std_logic;
FSL6_S_READ : out std_logic;
FSL6_S_DATA : in std_logic_vector(0 to 31);
FSL6_S_CONTROL : in std_logic;
FSL6_S_EXISTS : in std_logic;
FSL6_M_CLK : out std_logic;
FSL6_M_WRITE : out std_logic;
FSL6_M_DATA : out std_logic_vector(0 to 31);
FSL6_M_CONTROL : out std_logic;
FSL6_M_FULL : in std_logic;
FSL7_S_CLK : out std_logic;
FSL7_S_READ : out std_logic;
FSL7_S_DATA : in std_logic_vector(0 to 31);
FSL7_S_CONTROL : in std_logic;
FSL7_S_EXISTS : in std_logic;
FSL7_M_CLK : out std_logic;
FSL7_M_WRITE : out std_logic;
FSL7_M_DATA : out std_logic_vector(0 to 31);
FSL7_M_CONTROL : out std_logic;
FSL7_M_FULL : in std_logic;
ICACHE_FSL_IN_CLK : out std_logic;
ICACHE_FSL_IN_READ : out std_logic;
ICACHE_FSL_IN_DATA : in std_logic_vector(0 to 31);
ICACHE_FSL_IN_CONTROL : in std_logic;
ICACHE_FSL_IN_EXISTS : in std_logic;
ICACHE_FSL_OUT_CLK : out std_logic;
ICACHE_FSL_OUT_WRITE : out std_logic;
ICACHE_FSL_OUT_DATA : out std_logic_vector(0 to 31);
ICACHE_FSL_OUT_CONTROL : out std_logic;
ICACHE_FSL_OUT_FULL : in std_logic;
DCACHE_FSL_IN_CLK : out std_logic;
DCACHE_FSL_IN_READ : out std_logic;
DCACHE_FSL_IN_DATA : in std_logic_vector(0 to 31);
DCACHE_FSL_IN_CONTROL : in std_logic;
DCACHE_FSL_IN_EXISTS : in std_logic;
DCACHE_FSL_OUT_CLK : out std_logic;
DCACHE_FSL_OUT_WRITE : out std_logic;
DCACHE_FSL_OUT_DATA : out std_logic_vector(0 to 31);
DCACHE_FSL_OUT_CONTROL : out std_logic;
DCACHE_FSL_OUT_FULL : in std_logic
```

```
  );
end microblaze_0_wrapper;

architecture STRUCTURE of microblaze_0_wrapper is

  component microblaze is
    generic (
      C_SCO : integer;
      C_DATA_SIZE : integer;
      C_DYNAMIC_BUS_SIZING : integer;
      C_FAMILY : string;
      C_INSTANCE : string;
      C_AREA_OPTIMIZED : integer;
      C_D_OPB : integer;
      C_D_LMB : integer;
      C_I_OPB : integer;
      C_I_LMB : integer;
      C_USE_MSR_INSTR : integer;
      C_USE_PCMP_INSTR : integer;
      C_USE_BARREL : integer;
      C_USE_DIV : integer;
      C_USE_HW_MUL : integer;
      C_USE_FPU : integer;
      C_UNALIGNED_EXCEPTIONS : integer;
      C_ILL_OPCODE_EXCEPTION : integer;
      C_IOPB_BUS_EXCEPTION : integer;
      C_DOPB_BUS_EXCEPTION : integer;
      C_DIV_ZERO_EXCEPTION : integer;
      C_FPU_EXCEPTION : integer;
      C_PVR : integer;
      C_PVR_USER1 : std_logic_vector(0 to 7);
      C_PVR_USER2 : std_logic_vector(0 to 31);
      C_DEBUG_ENABLED : integer;
      C_NUMBER_OF_PC_BRK : integer;
      C_NUMBER_OF_RD_ADDR_BRK : integer;
      C_NUMBER_OF_WR_ADDR_BRK : integer;
      C_INTERRUPT_IS_EDGE : integer;
      C_EDGE_IS_POSITIVE : integer;
      C_RESET_MSR : std_logic_vector;
      C_OPCODE_0x0_ILLEGAL : integer;
      C_FSL_LINKS : integer;
      C_FSL_DATA_SIZE : integer;
      C_ICACHE_BASEADDR : std_logic_vector;
      C_ICACHE_HIGHADDR : std_logic_vector;
      C_USE_ICACHE : integer;
      C_ALLOW_ICACHE_WR : integer;
      C_ADDR_TAG_BITS : integer;
      C_CACHE_BYTE_SIZE : integer;
      C_ICACHE_USE_FSL : integer;
      C_ICACHE_LINE_LEN : integer;
      C_DCACHE_BASEADDR : std_logic_vector;
      C_DCACHE_HIGHADDR : std_logic_vector;
      C_USE_DCACHE : integer;
      C_ALLOW_DCACHE_WR : integer;
      C_DCACHE_ADDR_TAG : integer;
      C_DCACHE_BYTE_SIZE : integer;
      C_DCACHE_USE_FSL : integer;
      C_DCACHE_LINE_LEN : integer
    );
    port (
      CLK : in std_logic;
```

```
RESET : in std_logic;
INTERRUPT : in std_logic;
DEBUG_RST : in std_logic;
EXT_BRK : in std_logic;
EXT_NM_BRK : in std_logic;
DBG_STOP : in std_logic;
MB_Halted : out std_logic;
INSTR : in std_logic_vector(0 to 31);
I_ADDRTAG : out std_logic_vector(0 to 3);
IREADY : in std_logic;
IWAIT : in std_logic;
INSTR_ADDR : out std_logic_vector(0 to 31);
IFETCH : out std_logic;
I_AS : out std_logic;
DATA_READ : in std_logic_vector(0 to 31);
DREADY : in std_logic;
DWAIT : in std_logic;
DATA_WRITE : out std_logic_vector(0 to 31);
DATA_ADDR : out std_logic_vector(0 to 31);
D_ADDRTAG : out std_logic_vector(0 to 3);
D_AS : out std_logic;
READ_STROBE : out std_logic;
WRITE_STROBE : out std_logic;
BYTE_ENABLE : out std_logic_vector(0 to 3);
DM_ABUS : out std_logic_vector(0 to 31);
DM_BE : out std_logic_vector(0 to 3);
DM_BUSLOCK : out std_logic;
DM_DBUS : out std_logic_vector(0 to 31);
DM_REQUEST : out std_logic;
DM_RNW : out std_logic;
DM_SELECT : out std_logic;
DM_SEQADDR : out std_logic;
DOPB_DBUS : in std_logic_vector(0 to 31);
DOPB_ERRACK : in std_logic;
DOPB_MGRANT : in std_logic;
DOPB_RETRY : in std_logic;
DOPB_TIMEOUT : in std_logic;
DOPB_XFERACK : in std_logic;
IM_ABUS : out std_logic_vector(0 to 31);
IM_BE : out std_logic_vector(0 to 3);
IM_BUSLOCK : out std_logic;
IM_DBUS : out std_logic_vector(0 to 31);
IM_REQUEST : out std_logic;
IM_RNW : out std_logic;
IM_SELECT : out std_logic;
IM_SEQADDR : out std_logic;
IOPB_DBUS : in std_logic_vector(0 to 31);
IOPB_ERRACK : in std_logic;
IOPB_MGRANT : in std_logic;
IOPB_RETRY : in std_logic;
IOPB_TIMEOUT : in std_logic;
IOPB_XFERACK : in std_logic;
DBG_CLK : in std_logic;
DBG_TDI : in std_logic;
DBG_TDO : out std_logic;
DBG_REG_EN : in std_logic_vector(0 to 4);
DBG_CAPTURE : in std_logic;
DBG_UPDATE : in std_logic;
Trace_Instruction : out std_logic_vector(0 to 31);
Trace_Valid_Instr : out std_logic;
Trace_PC : out std_logic_vector(0 to 31);
```

```
Trace_Reg_Write : out std_logic;
Trace_Reg_Addr : out std_logic_vector(0 to 4);
Trace_MSR_Reg : out std_logic_vector(0 to 10);
Trace_New_Reg_Value : out std_logic_vector(0 to 31);
Trace_Exception_Taken : out std_logic;
Trace_Exception_Kind : out std_logic_vector(0 to 3);
Trace_Jump_Taken : out std_logic;
Trace_Delay_Slot : out std_logic;
Trace_Data_Address : out std_logic_vector(0 to 31);
Trace_Data_Access : out std_logic;
Trace_Data_Read : out std_logic;
Trace_Data_Write : out std_logic;
Trace_Data_Write_Value : out std_logic_vector(0 to 31);
Trace_Data_Byte_Enable : out std_logic_vector(0 to 3);
Trace_DCache_Req : out std_logic;
Trace_DCache_Hit : out std_logic;
Trace_ICache_Req : out std_logic;
Trace_ICache_Hit : out std_logic;
Trace_OF_PipeRun : out std_logic;
Trace_EX_PipeRun : out std_logic;
Trace_MEM_PipeRun : out std_logic;
FSL0_S_CLK : out std_logic;
FSL0_S_READ : out std_logic;
FSL0_S_DATA : in std_logic_vector(0 to C_FSL_DATA_SIZE-1);
FSL0_S_CONTROL : in std_logic;
FSL0_S_EXISTS : in std_logic;
FSL0_M_CLK : out std_logic;
FSL0_M_WRITE : out std_logic;
FSL0_M_DATA : out std_logic_vector(0 to C_FSL_DATA_SIZE-1);
FSL0_M_CONTROL : out std_logic;
FSL0_M_FULL : in std_logic;
FSL1_S_CLK : out std_logic;
FSL1_S_READ : out std_logic;
FSL1_S_DATA : in std_logic_vector(0 to C_FSL_DATA_SIZE-1);
FSL1_S_CONTROL : in std_logic;
FSL1_S_EXISTS : in std_logic;
FSL1_M_CLK : out std_logic;
FSL1_M_WRITE : out std_logic;
FSL1_M_DATA : out std_logic_vector(0 to C_FSL_DATA_SIZE-1);
FSL1_M_CONTROL : out std_logic;
FSL1_M_FULL : in std_logic;
FSL2_S_CLK : out std_logic;
FSL2_S_READ : out std_logic;
FSL2_S_DATA : in std_logic_vector(0 to C_FSL_DATA_SIZE-1);
FSL2_S_CONTROL : in std_logic;
FSL2_S_EXISTS : in std_logic;
FSL2_M_CLK : out std_logic;
FSL2_M_WRITE : out std_logic;
FSL2_M_DATA : out std_logic_vector(0 to C_FSL_DATA_SIZE-1);
FSL2_M_CONTROL : out std_logic;
FSL2_M_FULL : in std_logic;
FSL3_S_CLK : out std_logic;
FSL3_S_READ : out std_logic;
FSL3_S_DATA : in std_logic_vector(0 to C_FSL_DATA_SIZE-1);
FSL3_S_CONTROL : in std_logic;
FSL3_S_EXISTS : in std_logic;
FSL3_M_CLK : out std_logic;
FSL3_M_WRITE : out std_logic;
FSL3_M_DATA : out std_logic_vector(0 to C_FSL_DATA_SIZE-1);
FSL3_M_CONTROL : out std_logic;
FSL3_M_FULL : in std_logic;
```

```
  FSL4_S_CLK : out std_logic;
  FSL4_S_READ : out std_logic;
  FSL4_S_DATA : in std_logic_vector(0 to C_FSL_DATA_SIZE-1);
  FSL4_S_CONTROL : in std_logic;
  FSL4_S_EXISTS : in std_logic;
  FSL4_M_CLK : out std_logic;
  FSL4_M_WRITE : out std_logic;
  FSL4_M_DATA : out std_logic_vector(0 to C_FSL_DATA_SIZE-1);
  FSL4_M_CONTROL : out std_logic;
  FSL4_M_FULL : in std_logic;
  FSL5_S_CLK : out std_logic;
  FSL5_S_READ : out std_logic;
  FSL5_S_DATA : in std_logic_vector(0 to C_FSL_DATA_SIZE-1);
  FSL5_S_CONTROL : in std_logic;
  FSL5_S_EXISTS : in std_logic;
  FSL5_M_CLK : out std_logic;
  FSL5_M_WRITE : out std_logic;
  FSL5_M_DATA : out std_logic_vector(0 to C_FSL_DATA_SIZE-1);
  FSL5_M_CONTROL : out std_logic;
  FSL5_M_FULL : in std_logic;
  FSL6_S_CLK : out std_logic;
  FSL6_S_READ : out std_logic;
  FSL6_S_DATA : in std_logic_vector(0 to C_FSL_DATA_SIZE-1);
  FSL6_S_CONTROL : in std_logic;
  FSL6_S_EXISTS : in std_logic;
  FSL6_M_CLK : out std_logic;
  FSL6_M_WRITE : out std_logic;
  FSL6_M_DATA : out std_logic_vector(0 to C_FSL_DATA_SIZE-1);
  FSL6_M_CONTROL : out std_logic;
  FSL6_M_FULL : in std_logic;
  FSL7_S_CLK : out std_logic;
  FSL7_S_READ : out std_logic;
  FSL7_S_DATA : in std_logic_vector(0 to C_FSL_DATA_SIZE-1);
  FSL7_S_CONTROL : in std_logic;
  FSL7_S_EXISTS : in std_logic;
  FSL7_M_CLK : out std_logic;
  FSL7_M_WRITE : out std_logic;
  FSL7_M_DATA : out std_logic_vector(0 to C_FSL_DATA_SIZE-1);
  FSL7_M_CONTROL : out std_logic;
  FSL7_M_FULL : in std_logic;
  ICACHE_FSL_IN_CLK : out std_logic;
  ICACHE_FSL_IN_READ : out std_logic;
  ICACHE_FSL_IN_DATA : in std_logic_vector(0 to 31);
  ICACHE_FSL_IN_CONTROL : in std_logic;
  ICACHE_FSL_IN_EXISTS : in std_logic;
  ICACHE_FSL_OUT_CLK : out std_logic;
  ICACHE_FSL_OUT_WRITE : out std_logic;
  ICACHE_FSL_OUT_DATA : out std_logic_vector(0 to 31);
  ICACHE_FSL_OUT_CONTROL : out std_logic;
  ICACHE_FSL_OUT_FULL : in std_logic;
  DCACHE_FSL_IN_CLK : out std_logic;
  DCACHE_FSL_IN_READ : out std_logic;
  DCACHE_FSL_IN_DATA : in std_logic_vector(0 to 31);
  DCACHE_FSL_IN_CONTROL : in std_logic;
  DCACHE_FSL_IN_EXISTS : in std_logic;
  DCACHE_FSL_OUT_CLK : out std_logic;
  DCACHE_FSL_OUT_WRITE : out std_logic;
  DCACHE_FSL_OUT_DATA : out std_logic_vector(0 to 31);
  DCACHE_FSL_OUT_CONTROL : out std_logic;
  DCACHE_FSL_OUT_FULL : in std_logic
);
```

```
  end component;

  attribute x_core_info : STRING;
  attribute x_core_info of microblaze : component is "microblaze_v6_00_b";

begin

  microblaze_0 : microblaze
    generic map (
      C_SCO => 0,
      C_DATA_SIZE => 32,
      C_DYNAMIC_BUS_SIZING => 1,
      C_FAMILY => "virtex4",
      C_INSTANCE => "microblaze_0",
      C_AREA_OPTIMIZED => 0,
      C_D_OPB => 1,
      C_D_LMB => 1,
      C_I_OPB => 1,
      C_I_LMB => 1,
      C_USE_MSR_INSTR => 1,
      C_USE_PCMP_INSTR => 1,
      C_USE_BARREL => 0,
      C_USE_DIV => 0,
      C_USE_HW_MUL => 1,
      C_USE_FPU => 0,
      C_UNALIGNED_EXCEPTIONS => 0,
      C_ILL_OPCODE_EXCEPTION => 0,
      C_IOPB_BUS_EXCEPTION => 0,
      C_DOPB_BUS_EXCEPTION => 0,
      C_DIV_ZERO_EXCEPTION => 0,
      C_FPU_EXCEPTION => 0,
      C_PVR => 0,
      C_PVR_USER1 => X"00",
      C_PVR_USER2 => X"00000000",
      C_DEBUG_ENABLED => 1,
      C_NUMBER_OF_PC_BRK => 2,
      C_NUMBER_OF_RD_ADDR_BRK => 0,
      C_NUMBER_OF_WR_ADDR_BRK => 0,
      C_INTERRUPT_IS_EDGE => 0,
      C_EDGE_IS_POSITIVE => 1,
      C_RESET_MSR => X"00000000",
      C_OPCODE_0x0_ILLEGAL => 0,
      C_FSL_LINKS => 1,
      C_FSL_DATA_SIZE => 32,
      C_ICACHE_BASEADDR => X"00000000",
      C_ICACHE_HIGHADDR => X"3FFFFFFF",
      C_USE_ICACHE => 0,
      C_ALLOW_ICACHE_WR => 1,
      C_ADDR_TAG_BITS => 0,
      C_CACHE_BYTE_SIZE => 8192,
      C_ICACHE_USE_FSL => 1,
      C_ICACHE_LINE_LEN => 4,
      C_DCACHE_BASEADDR => X"00000000",
      C_DCACHE_HIGHADDR => X"3FFFFFFF",
      C_USE_DCACHE => 0,
      C_ALLOW_DCACHE_WR => 1,
      C_DCACHE_ADDR_TAG => 0,
      C_DCACHE_BYTE_SIZE => 8192,
      C_DCACHE_USE_FSL => 1,
      C_DCACHE_LINE_LEN => 4
    )
```

```
port map (
  CLK => CLK,
  RESET => RESET,
  INTERRUPT => INTERRUPT,
  DEBUG_RST => DEBUG_RST,
  EXT_BRK => EXT_BRK,
  EXT_NM_BRK => EXT_NM_BRK,
  DBG_STOP => DBG_STOP,
  MB_Halted => MB_Halted,
  INSTR => INSTR,
  I_ADDRTAG => I_ADDRTAG,
  IREADY => IREADY,
  IWAIT => IWAIT,
  INSTR_ADDR => INSTR_ADDR,
  IFETCH => IFETCH,
  I_AS => I_AS,
  DATA_READ => DATA_READ,
  DREADY => DREADY,
  DWAIT => DWAIT,
  DATA_WRITE => DATA_WRITE,
  DATA_ADDR => DATA_ADDR,
  D_ADDRTAG => D_ADDRTAG,
  D_AS => D_AS,
  READ_STROBE => READ_STROBE,
  WRITE_STROBE => WRITE_STROBE,
  BYTE_ENABLE => BYTE_ENABLE,
  DM_ABUS => DM_ABUS,
  DM_BE => DM_BE,
  DM_BUSLOCK => DM_BUSLOCK,
  DM_DBUS => DM_DBUS,
  DM_REQUEST => DM_REQUEST,
  DM_RNW => DM_RNW,
  DM_SELECT => DM_SELECT,
  DM_SEQADDR => DM_SEQADDR,
  DOPB_DBUS => DOPB_DBUS,
  DOPB_ERRACK => DOPB_ERRACK,
  DOPB_MGRANT => DOPB_MGRANT,
  DOPB_RETRY => DOPB_RETRY,
  DOPB_TIMEOUT => DOPB_TIMEOUT,
  DOPB_XFERACK => DOPB_XFERACK,
  IM_ABUS => IM_ABUS,
  IM_BE => IM_BE,
  IM_BUSLOCK => IM_BUSLOCK,
  IM_DBUS => IM_DBUS,
  IM_REQUEST => IM_REQUEST,
  IM_RNW => IM_RNW,
  IM_SELECT => IM_SELECT,
  IM_SEQADDR => IM_SEQADDR,
  IOPB_DBUS => IOPB_DBUS,
  IOPB_ERRACK => IOPB_ERRACK,
  IOPB_MGRANT => IOPB_MGRANT,
  IOPB_RETRY => IOPB_RETRY,
  IOPB_TIMEOUT => IOPB_TIMEOUT,
  IOPB_XFERACK => IOPB_XFERACK,
  DBG_CLK => DBG_CLK,
  DBG_TDI => DBG_TDI,
  DBG_TDO => DBG_TDO,
  DBG_REG_EN => DBG_REG_EN,
  DBG_CAPTURE => DBG_CAPTURE,
  DBG_UPDATE => DBG_UPDATE,
  Trace_Instruction => Trace_Instruction,
```

```
Trace_Valid_Instr => Trace_Valid_Instr,
Trace_PC => Trace_PC,
Trace_Reg_Write => Trace_Reg_Write,
Trace_Reg_Addr => Trace_Reg_Addr,
Trace_MSR_Reg => Trace_MSR_Reg,
Trace_New_Reg_Value => Trace_New_Reg_Value,
Trace_Exception_Taken => Trace_Exception_Taken,
Trace_Exception_Kind => Trace_Exception_Kind,
Trace_Jump_Taken => Trace_Jump_Taken,
Trace_Delay_Slot => Trace_Delay_Slot,
Trace_Data_Address => Trace_Data_Address,
Trace_Data_Access => Trace_Data_Access,
Trace_Data_Read => Trace_Data_Read,
Trace_Data_Write => Trace_Data_Write,
Trace_Data_Write_Value => Trace_Data_Write_Value,
Trace_Data_Byte_Enable => Trace_Data_Byte_Enable,
Trace_DCache_Req => Trace_DCache_Req,
Trace_DCache_Hit => Trace_DCache_Hit,
Trace_ICache_Req => Trace_ICache_Req,
Trace_ICache_Hit => Trace_ICache_Hit,
Trace_OF_PipeRun => Trace_OF_PipeRun,
Trace_EX_PipeRun => Trace_EX_PipeRun,
Trace_MEM_PipeRun => Trace_MEM_PipeRun,
FSL0_S_CLK => FSL0_S_CLK,
FSL0_S_READ => FSL0_S_READ,
FSL0_S_DATA => FSL0_S_DATA,
FSL0_S_CONTROL => FSL0_S_CONTROL,
FSL0_S_EXISTS => FSL0_S_EXISTS,
FSL0_M_CLK => FSL0_M_CLK,
FSL0_M_WRITE => FSL0_M_WRITE,
FSL0_M_DATA => FSL0_M_DATA,
FSL0_M_CONTROL => FSL0_M_CONTROL,
FSL0_M_FULL => FSL0_M_FULL,
FSL1_S_CLK => FSL1_S_CLK,
FSL1_S_READ => FSL1_S_READ,
FSL1_S_DATA => FSL1_S_DATA,
FSL1_S_CONTROL => FSL1_S_CONTROL,
FSL1_S_EXISTS => FSL1_S_EXISTS,
FSL1_M_CLK => FSL1_M_CLK,
FSL1_M_WRITE => FSL1_M_WRITE,
FSL1_M_DATA => FSL1_M_DATA,
FSL1_M_CONTROL => FSL1_M_CONTROL,
FSL1_M_FULL => FSL1_M_FULL,
FSL2_S_CLK => FSL2_S_CLK,
FSL2_S_READ => FSL2_S_READ,
FSL2_S_DATA => FSL2_S_DATA,
FSL2_S_CONTROL => FSL2_S_CONTROL,
FSL2_S_EXISTS => FSL2_S_EXISTS,
FSL2_M_CLK => FSL2_M_CLK,
FSL2_M_WRITE => FSL2_M_WRITE,
FSL2_M_DATA => FSL2_M_DATA,
FSL2_M_CONTROL => FSL2_M_CONTROL,
FSL2_M_FULL => FSL2_M_FULL,
FSL3_S_CLK => FSL3_S_CLK,
FSL3_S_READ => FSL3_S_READ,
FSL3_S_DATA => FSL3_S_DATA,
FSL3_S_CONTROL => FSL3_S_CONTROL,
FSL3_S_EXISTS => FSL3_S_EXISTS,
FSL3_M_CLK => FSL3_M_CLK,
FSL3_M_WRITE => FSL3_M_WRITE,
FSL3_M_DATA => FSL3_M_DATA,
```

```
FSL3_M_CONTROL => FSL3_M_CONTROL,
FSL3_M_FULL => FSL3_M_FULL,
FSL4_S_CLK => FSL4_S_CLK,
FSL4_S_READ => FSL4_S_READ,
FSL4_S_DATA => FSL4_S_DATA,
FSL4_S_CONTROL => FSL4_S_CONTROL,
FSL4_S_EXISTS => FSL4_S_EXISTS,
FSL4_M_CLK => FSL4_M_CLK,
FSL4_M_WRITE => FSL4_M_WRITE,
FSL4_M_DATA => FSL4_M_DATA,
FSL4_M_CONTROL => FSL4_M_CONTROL,
FSL4_M_FULL => FSL4_M_FULL,
FSL5_S_CLK => FSL5_S_CLK,
FSL5_S_READ => FSL5_S_READ,
FSL5_S_DATA => FSL5_S_DATA,
FSL5_S_CONTROL => FSL5_S_CONTROL,
FSL5_S_EXISTS => FSL5_S_EXISTS,
FSL5_M_CLK => FSL5_M_CLK,
FSL5_M_WRITE => FSL5_M_WRITE,
FSL5_M_DATA => FSL5_M_DATA,
FSL5_M_CONTROL => FSL5_M_CONTROL,
FSL5_M_FULL => FSL5_M_FULL,
FSL6_S_CLK => FSL6_S_CLK,
FSL6_S_READ => FSL6_S_READ,
FSL6_S_DATA => FSL6_S_DATA,
FSL6_S_CONTROL => FSL6_S_CONTROL,
FSL6_S_EXISTS => FSL6_S_EXISTS,
FSL6_M_CLK => FSL6_M_CLK,
FSL6_M_WRITE => FSL6_M_WRITE,
FSL6_M_DATA => FSL6_M_DATA,
FSL6_M_CONTROL => FSL6_M_CONTROL,
FSL6_M_FULL => FSL6_M_FULL,
FSL7_S_CLK => FSL7_S_CLK,
FSL7_S_READ => FSL7_S_READ,
FSL7_S_DATA => FSL7_S_DATA,
FSL7_S_CONTROL => FSL7_S_CONTROL,
FSL7_S_EXISTS => FSL7_S_EXISTS,
FSL7_M_CLK => FSL7_M_CLK,
FSL7_M_WRITE => FSL7_M_WRITE,
FSL7_M_DATA => FSL7_M_DATA,
FSL7_M_CONTROL => FSL7_M_CONTROL,
FSL7_M_FULL => FSL7_M_FULL,
ICACHE_FSL_IN_CLK => ICACHE_FSL_IN_CLK,
ICACHE_FSL_IN_READ => ICACHE_FSL_IN_READ,
ICACHE_FSL_IN_DATA => ICACHE_FSL_IN_DATA,
ICACHE_FSL_IN_CONTROL => ICACHE_FSL_IN_CONTROL,
ICACHE_FSL_IN_EXISTS => ICACHE_FSL_IN_EXISTS,
ICACHE_FSL_OUT_CLK => ICACHE_FSL_OUT_CLK,
ICACHE_FSL_OUT_WRITE => ICACHE_FSL_OUT_WRITE,
ICACHE_FSL_OUT_DATA => ICACHE_FSL_OUT_DATA,
ICACHE_FSL_OUT_CONTROL => ICACHE_FSL_OUT_CONTROL,
ICACHE_FSL_OUT_FULL => ICACHE_FSL_OUT_FULL,
DCACHE_FSL_IN_CLK => DCACHE_FSL_IN_CLK,
DCACHE_FSL_IN_READ => DCACHE_FSL_IN_READ,
DCACHE_FSL_IN_DATA => DCACHE_FSL_IN_DATA,
DCACHE_FSL_IN_CONTROL => DCACHE_FSL_IN_CONTROL,
DCACHE_FSL_IN_EXISTS => DCACHE_FSL_IN_EXISTS,
DCACHE_FSL_OUT_CLK => DCACHE_FSL_OUT_CLK,
DCACHE_FSL_OUT_WRITE => DCACHE_FSL_OUT_WRITE,
DCACHE_FSL_OUT_DATA => DCACHE_FSL_OUT_DATA,
DCACHE_FSL_OUT_CONTROL => DCACHE_FSL_OUT_CONTROL,
```

```
      DCACHE_FSL_OUT_FULL => DCACHE_FSL_OUT_FULL
    );

end architecture STRUCTURE;
```

Acronymes

AOP	Aspect Oriented Programming
ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Instruction Set Processor
BRBW	Blocking Read Blocking Write
BRNW	Blocking Read Non Blocking Write
CAO	Conception Assistée par Ordinateur
CDFG	Control Data Flow Graph
CFG	Control Flow Graph
CFSM	Concurrent Finite State Machine
CGA	Coarse Grained Action
CPU	Central Processing Unit
CSP	Communicating Sequential Processes
CWL	Component Wrapper Language
DART	Data Parallelism for Real Time
DFG	Data Flow Graph
DMA	Direct Memory Access
DRM	Detailed Resource Modelling
DSE	Design Space Exploration
DSP	Digital Signal Processor
FPGA	Field Programmable Gate Array
FSL	Fast Simplex Link
FSM	Finite State Machine
GCM	Generic Component Modelling
GPP	General Purpose Processor
GRM	Generic Resource Modelling
HASOC	Hardware and Software Objects On Chip
HCDFG	Hierarchic Control Data Flow Graph
HDL	Hardware Description Language
HFSM	Hierarchic Finite State Machine
HLS	High Level Synthesis
HRM	Hardware Resource Modelling

ILP	Instruction Level Parallelism
IP	Intellectual Property
KPN	Kahn Process Network
MART	Modeling and Analysis of Real Time Embedded Systems
MDA	Model Driven Architecture
MDD	Model Driven Development
MMU	Memory Management Unit
MOC	Model Of Computation
MPSOC	Multi Processors System On Chip
NFP	Non Functional Property
NOC	Network On Chip
NRBW	Non Blocking Read Blocking Write
NRNW	Non Blocking Read Non Blocking Write
OCL	Object Constraint Language
OOSE	Object Oriented Software Engineering
OMG	Object Modeling Group
OMT	Object Modeling Technique
OPB	On-chip Peripheral Bus
PAM	Performance Analysis Modelling
PSM	Program State Machine
QOS	Quality Of Service
RAM	Random Access Memory
ROM	Read Only Memory
RS	Reactive Synchronous
RSM	Software Resource Modelling
RTL	Register Transfer Level
RTOS	Real Time Operating System
SAM	Schedulability Analysis Modelling
SDF	Synchronous Data Flow
SDL	System Description Language
SDP	Simplex Data Protocol
SLOOP	System Level design with Object Oriented Process
SOC	System On Chip
SRM	Software Resource Modelling

SysML	System Modeling Language
TLM	Transaction Level Modeling
TML	Task Modeling Language
TUT	Technology University of Tampere
UML	Unified Modeling Language
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLSI	Very Large Scale Integration
VSL	Value Specification Language
WCET	Worst Case Execution Time