Modélisation et Simulation Rapide au niveau cycle pour l'Exploration Architecturale de Systèmes Intégrés sur puce

Richard Buchmann

▶ **To cite this version:**

Richard Buchmann. Modélisation et Simulation Rapide au niveau cycle pour l'Exploration Architecturale de Systèmes Intégrés sur puce. Micro et nanotechnologies/Microélectronique. Université Pierre et Marie Curie - Paris VI, 2006. Français. NNT : . tel-00426066

HAL Id: tel-00426066
https://theses.hal.science/tel-00426066v1

Submitted on 23 Oct 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE DE DOCTORAT DE L'UNIVERSITÉ PARIS VI

SPÉCIALITÉ INFORMATIQUE

Présentée par Richard BUCHMANN

Pour obtenir le titre de

DOCTEUR DE L'UNIVERSITÉ PARIS VI

MODÉLISATION ET SIMULATION RAPIDE AU NIVEAU CYCLE POUR L'EXPLORATION ARCHITECTURALE DE SYSTÈMES INTÉGRÉS SUR PUCE

Soutenue le 5 décembre 2006, devant le jury composé de

M. Frédéric PÉTROT	Rapporteur
M. Tanguy RISSET	Rapporteur
M. Alain GREINER	Directeur de thèse
M. François CHAROT	Examinateur
M. Jean-Marie CHESNEAUX	Examinateur
M. Bruno JEGO	Examinateur
M. Olivier TEMAM	Examinateur

Résumé

La modélisation d'un système intégré sur puce nécessite la spécification de l'application logicielle et la modélisation de l'architecture matérielle puis le déploiement du logiciel sur ce matériel. L'objectif du concepteur de systèmes intégrés est de trouver la meilleure solution de déploiement pour optimiser les critères de surface de silicium, de consommation d'énergie, et de performances. Ces critères sont le plus souvent évalués par simulation.

En raison du grand nombre de paramètres de l'architecture matérielle et des choix dans le déploiement du logiciel sur l'architecture, le temps nécessaire pour les simulations est important. Les outils permettant de réduire ce temps présentent un grand intérêt.

Cette thèse présente des principes et des outils pour faciliter le développement des architectures matérielles et pour accélérer la simulation de modèles d'architectures synchrones décrites en langage SystemC, précis au cycle près et au bit près. Ce document est constitué de quatre chapitres :

- La modélisation de composants matériels en SystemC sous la forme d'automates synchrones communicants (CFSM) ;
- La génération de modèles SystemC, pour la simulation, à partir de descriptions synthétisables VHDL au niveau RTL ;
- La vérification des règles d'écriture des modèles SystemC ;
- La simulation rapide à l'aide d'une technique d'ordonnancement totalement statique.

Ces outils permettent au concepteur de construire rapidement une architecture matérielle à l'aide de composants synthétisables au niveau RTL et de composants SystemC, respectant le modèle des CFSM. SystemCASS simule une telle architecture avec une accélération supérieure à un facteur 12 par rapport à un simulateur à échéancier dynamique.

Mots clés : Exploration Architecturale, SOC, Modélisation, Simulation, Ordonnancement, Automate à États Finis

Abstract

System On Chip modeling is based on software specification, hardware modeling, and software to hardware mapping. The system designer goal is to find the best mapping that matches specifications while optimizing performances, silicium area, and energy consumption.

The same system designer is faced with architectural exploration issues due to the important number hardware/software parameters. Architectural exploration is time consuming, and any tool that can reduce or ease the development process is of paramount interest.

This thesis presents principles and tools to facilitate hardware development and to speed up synchronous hardware simulation. The targeted simulation platform is based on SystemC language and relies on bit/cycle accurate models. Four chapters present :

- The use of communicating synchronous finite state machines as an effective means to model hardware components and platform (CFSM) ;
- SystemC model generation from synthesizable VHDL description at RTL level ;
- Writing rules checking of SystemC models ;
- Fast simulation using entirely static scheduling.

These tools allow the system designer to build an hardware architecture using synthesizable components at RTL level, and SystemC components, based on CFSM model. SystemCASS simulates such architecture 12 faster than a simulator using a dynamic scheduling.

Keywords : Architectural Exploration, SOC, Modeling, Simulation, Scheduling, Finite State Machine

Remerciements

Je tiens à remercier Alain Greiner pour avoir dirigé ma thèse, et qui a permis de la financer. J'ai apprécié tout particulièrement ses conseils durant la rédaction. Sans ces précieux conseils, le manuscrit aurait eu un tout autre visage. Je remercie également Frédéric Pétrot et Tanguy Risset pour avoir accepté la lourde tâche de rapporteur, ainsi que François Charot, Jean-Marie Chesneaux, Bruno Jego, et Olivier Temam, pour avoir accepté d'être membre du jury.

Lorsque j'étais étudiant en master, j'ai très apprécié les qualités pédagogiques de Frédéric Pétrot et d'Alain Greiner. Je tiens à leur exprimer toute ma gratitude car ils ont largement influencé mon choix de spécialisation en CAO pour les systèmes intégrés sur puce.

Je remercie Ludovic Jacomme et toute l'équipe ASIM, pour leur disponibilité, leur sérieux, et leur sympathie. Une question reste rarement sans réponse. Il est toujours agréable de travailler au quotidien avec des collègues que l'on apprécie. C'est pourquoi, je remercie particulièrement mes collègues doctorants (ou ex-doctorants...) : Emmanuel Viaud, Etienne Faure, François Donnet, Maxime Palus, Pascale Gomez, et Wahid Bahroun.

Je remercie également mon entourage proche pour m'avoir soutenu durant les moments difficiles. Certains de mes proches ont eu la gentillesse de participer à la relecture. Cet effort est d'autant plus louable qu'ils ne travaillent pas dans le domaine des systèmes intégrés.

Lors de la 1ère année d'études universitaires, il n'est pas rare de voir une amitié forte naître entre deux personnes. Celle qui me lie avec Sébastien ne fait pas exception. Celle-ci, je cite, "s'étend au-delà du jour sidéral". Encore une vieille histoire...

Je remercie, de tout cœur, Juliana qui m'a accompagné dès le début de ma thèse, et qui a fait preuve d'une patience infinie. La patience est une qualité qui s'apprécie...

Table des matières

Table des matières **vii**

1 Problématique **1**

1.1 Modélisation d’un système intégré sur puce 2
1.2 Exploration Architecturale 4
1.3 Simulation architecturale 5
1.4 Conclusions 10

2 Modélisation CABA **13**

2.1 État de l’art 14
2.2 Modèle d’architecture en CFSM 21
2.3 Représentation en SystemC 27
2.4 Conclusions 37

3 Ordonnancement statique **39**

3.1 État de l’art 40
3.2 Hypothèses de modélisation des modules 45
3.3 Construction de l’ordre d’exécution des processus 46
3.4 Résultats expérimentaux 53
3.5 Conclusions 63

4 Génération de modèle en CFSM **65**

4.1 État de l’art 66
4.2 Principe de la méthode proposée 72
4.3 Techniques utilisées par VASY 79
4.4 Construction d’un automate et génération 89
4.5 Résultats expérimentaux 98
4.6 Conclusions 101

5 Vérification sémantique 103
5.1 Approche dynamique 104
5.2 Principe de la vérification sémantique 107
5.3 Construction de l'arbre de syntaxe abstraite 109
5.4 Construction du support des variables 110
5.5 Identification des constantes architecturales 116
5.6 Vérification des contraintes de modélisation 116
5.7 Mise en œuvre 116
5.8 Résultats expérimentaux 117
5.9 Conclusions 121

6 Conclusions et perspectives 123
6.1 Bilan des travaux 124
6.2 Conclusions générales 126
6.3 Perspectives 126

A Annexe : Support d'une expression i
A.1 Définition du support simple d'une expression i
A.2 Définition du support étendu d'une expression i

B Acronymes iii

Bibliographie v

Liste des tableaux vii

Table des figures ix

Listings xiii

Chapitre 1

Problématique

Sommaire

1.1 Modélisation d'un système intégré sur puce 2
1.1.1 Spécification de l'application logicielle 2
1.1.2 Modélisation de l'architecture matérielle 2
1.1.3 Déploiement du logiciel sur l'architecture matérielle 3
1.2 Exploration Architecturale 4
1.3 Simulation architecturale 5
1.3.1 Niveaux d'abstraction des modèles matériels avec représentation du temps 6
1.3.2 Moteur de simulation 8
1.3.3 Modélisation pour l'ordonnancement statique 9
1.3.4 Vérification sémantique 9
1.3.5 Génération de modèles pour la simulation rapide 10
1.4 Conclusions 10

La simulation architecturale est une méthode incontournable pour l'évaluation et la validation des systèmes intégrés sur puce. La recherche sur les méthodes de simulation a un impact majeur sur l'exploration architecturale.

Au cours de ce chapitre, nous présentons la modélisation d'un système intégré sur puce, puis l'exploration architecturale. Les problèmes abordés dans cette thèse sont résumés à la fin du chapitre.

1.1 Modélisation d'un système intégré sur puce

Les applications multimédia (télévision numérique, DVD, ...) ou les applications orientées télécommunications (processeurs réseau, terminaux mobiles) sont implantées sur des architectures matérielles, intégrant à l'intérieur d'une même puce, du logiciel qui s'exécute sur un ou plusieurs processeurs et un petit nombre d'accélérateurs matériels.

La modélisation d'un système intégré sur puce nécessite la modélisation de l'architecture matérielle et la description de l'application logicielle puis le déploiement du logiciel sur ce matériel. Les sections suivantes illustrent ces différentes étapes en s'appuyant sur l'exemple d'une application de décompression *Motion JPEG* (MJPEG).

1.1.1 Spécification de l'application logicielle

Pour exploiter le parallélisme gros grain des applications logicielles, les applications sont souvent décrites sous la forme d'un graphe de tâches (processus ou threads). Les tâches communiquent entre elles par des canaux de communication point à point. Les noeuds du graphe représentent les tâches. Les arcs représentent les canaux de communication. Le modèle théorique est celui des réseaux de processus de Kahn [Kah74].

Les applications sont généralement spécifiées entièrement sous forme logicielle pour pouvoir être directement exécutées et validées sur une station de travail. Le langage C/C++ est couramment utilisé pour la spécification.

La figure 1.1 présente une application de décompression d'un flux d'images MJPEG. La tâche *Traffic Generator* fournit l'image dont les informations, contenues dans l'entête, sont lues par la tâche *Demux*. La tâche Variable Length Decoder *(VLD)* effectue un décodage d'Huffman. La tâche Inverse Quantification *(IQ)* multiplie les données par un quantum puis la tâche ZigZag *(ZZ)* change l'ordre des données. La tâche Inverse Discrete Cosine Transform *(IDCT)* calcule la transformation inverse permettant de reconstruire un bloc de l'image. Enfin, la tâche *LineBuilder* construit les lignes d'image. La tâche *Display* affiche l'image résultant de la décompression.

1.1.2 Modélisation de l'architecture matérielle

L'architecture matérielle est typiquement multiprocesseur, comportant plusieurs bancs mémoire, un ou plusieurs coprocesseurs spécialisés, organisés autour d'un micro réseau d'interconnexions ou d'un bus.

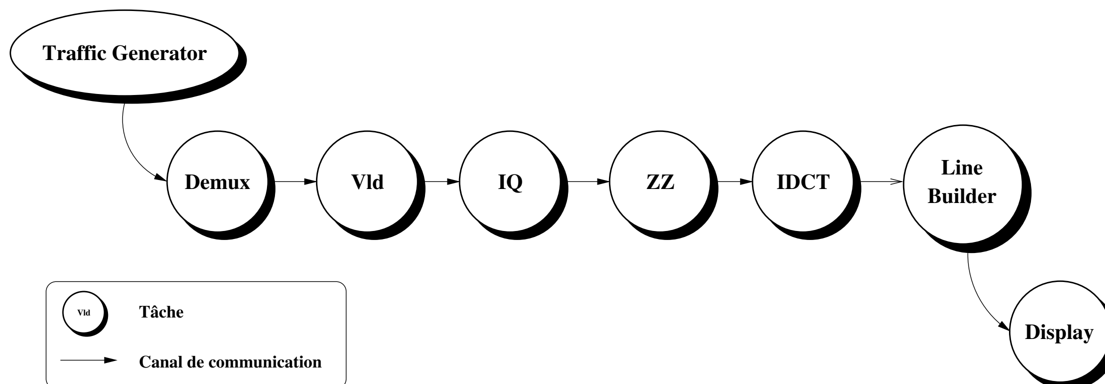

FIG. 1.1 – Graphe de tâches de l'application de décompression MJPEG

L'architecture est représentée sous la forme d'un schéma dans lequel les composants matériels sont connectés par un signal ou bien par une nappe de signaux. Les modèles de composants matériels sont appelés *modules*.

Chaque module est décrit à l'aide d'un langage de description de matériel tels que *Very High Speed Integrated Circuits Hardware Description Language* (VHDL), Verilog, ou SystemC.

La figure 1.2 présente un exemple d'architecture matérielle à quatre processeurs programmables connectés par un micro réseau générique à interface *Virtual Component Interface* [All97] (VCI). Chaque processeur possède son propre cache de données et d'instructions. L'architecture contient également un coprocesseur et un timer dont les signaux d'interruption sont connectés aux processeurs. Les signaux d'horloge et de reset sont connectés à chacun des composants. Ces signaux ne sont pas représentés sur la figure.

1.1.3 Déploiement du logiciel sur l'architecture matérielle

Les tâches peuvent être implantées soit en matériel, soit en logiciel. Le déploiement consiste à définir le placement des tâches logicielles sur les processeurs programmables et/ou sur les accélérateurs matériels.

L'exemple, figure 1.3, décrit un déploiement possible de l'application de décompression MJPEG sur une architecture matérielle multiprocesseur. Les tâches logicielles *DEMUX*, *VLD*, *IQ*, *ZZ*, et *LineBuilder* sont déployées sur des processeurs programmables. Les tâches *Traffic Generator* et *Display* qui correspondent aux entrées et sorties du flux vidéo sont implantées sur des composants matériels. De plus, la tâche *IDCT* qui est très gourmande en temps de calcul, est implantée sur un accélérateur matériel.

Le résultat du déploiement est une solution au problème de placement des tâches logicielles sur une architecture matérielle. La recherche de solutions de déploiement s'appelle l'exploration architecturale.

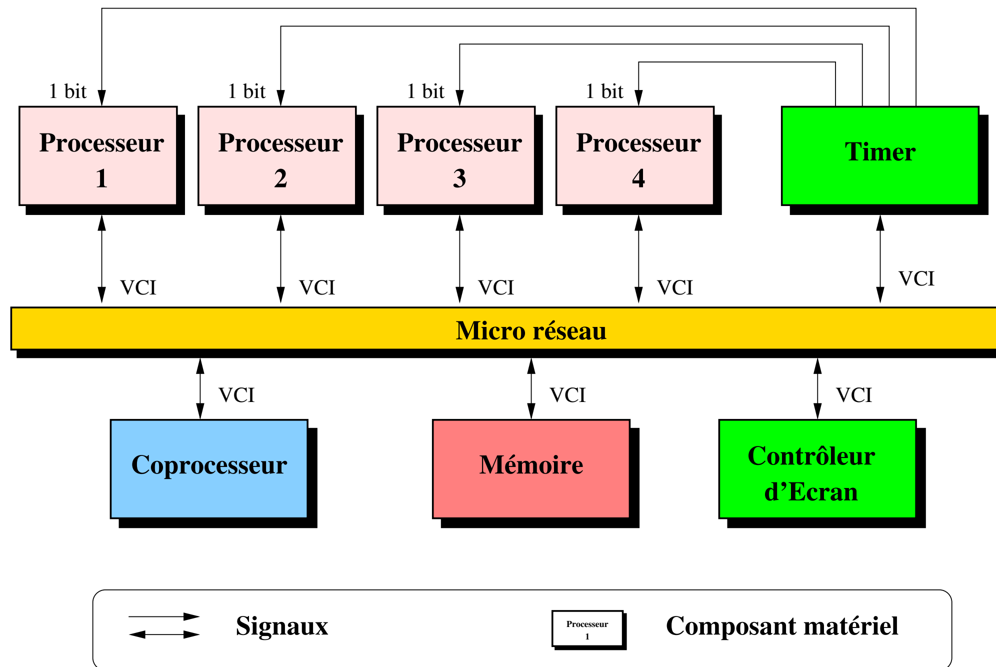

FIG. 1.2 – Architecture matérielle multiprocesseur

1.2 Exploration Architecturale

L'exploration architecturale consiste à chercher la meilleure solution de déploiement d'une application logicielle sur une architecture matérielle paramétrable. Les principaux critères d'optimisation sont :

- la surface de silicium occupée ;
- la consommation d'énergie du système intégré ;
- les performances en nombre de cycles.

L'architecture matérielle possède de nombreux paramètres tels que le nombre et la puissance des processeurs programmables, la capacité et la stratégie des caches, la latence et le débit des canaux de communication, le type de composant d'interconnexion, etc.

En raison du grand nombre de paramètres de l'architecture matérielle et des choix dans le déploiement du logiciel sur l'architecture, l'espace des solutions est gigantesque.

L'exploration architecturale est un problème d'optimisation NP-difficile. Ce problème est généralement résolu manuellement par le concepteur. Celui-ci évalue et compare la qualité de l'architecture pour un nombre restreint de solutions. Le concepteur obtient ainsi une bonne solution de déploiement mais il n'a toutefois pas la garantie que cette solution soit la meilleure.

Une des principales méthodes pour évaluer la surface, la consommation, et les performances d'une solution est la simulation architecturale. Le travail d'optimisation se déroule en plusieurs

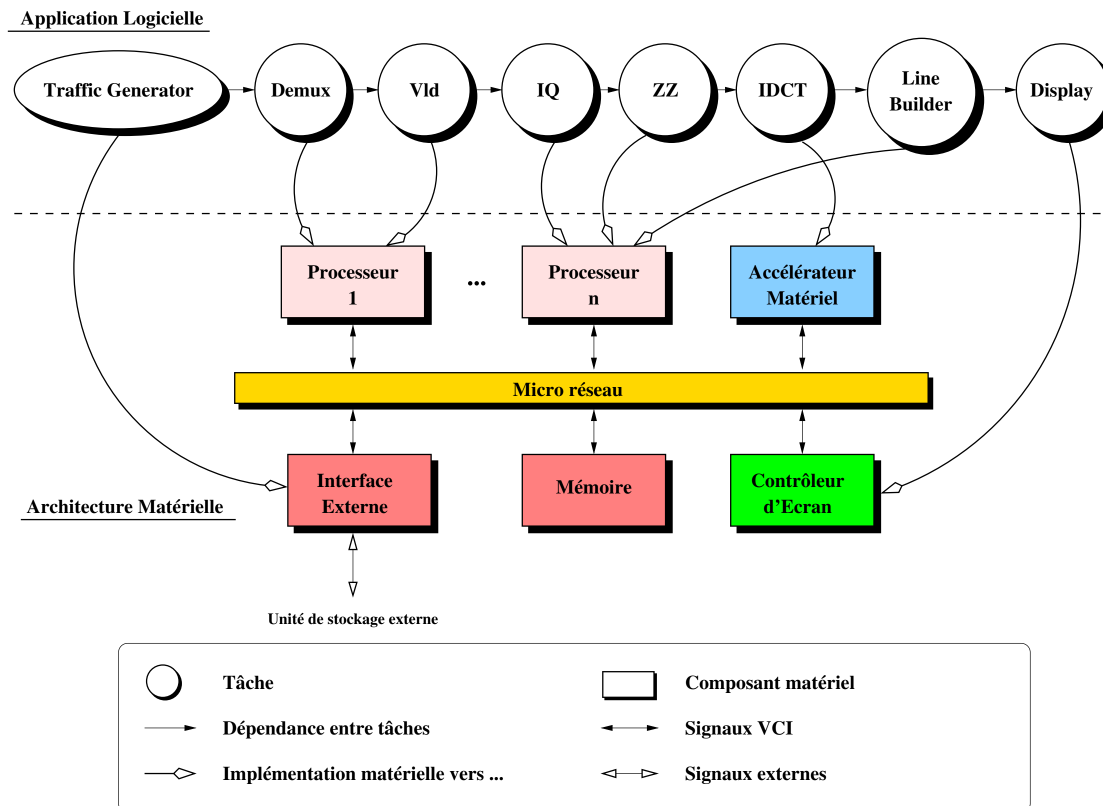

FIG. 1.3 – Déploiement d'une application MJPEG sur une architecture matérielle

passes au cours desquelles le concepteur simule, observe puis ajuste l'architecture.

1.3 Simulation architecturale

La simulation architecturale est la fonction de coût de l'exploration architecturale. La simulation de systèmes intégrés sur puce est de type temps discret. Autrement dit, l'algorithme calcule, de manière itérative, l'état du système pour chaque unité de temps.

L'accroissement de la taille et de la complexité des systèmes intégrés sur puce implique des temps de simulation très importants pouvant atteindre plusieurs jours de calculs. L'exploration architecturale s'en trouve alors très ralentie.

L'objectif de cette thèse est d'accélérer l'exploration architecturale pour converger plus rapidement vers un bon choix d'architecture. Pour cela, nous souhaitons réduire les temps de simulation.

La durée de la simulation dépend principalement de la vitesse d'exécution des modèles, ainsi que du moteur de simulation. Les sections suivantes traitent du choix du niveau d'abstraction des

modèles matériels puis du moteur du simulateur pour la simulation rapide.

1.3.1 Niveaux d'abstraction des modèles matériels avec représentation du temps

Nous pouvons distinguer quatre principaux niveaux d'abstraction pour la modélisation de composants matériels :

- le niveau porte ;
- le niveau *Register Transfer Level* (RTL) ;
- le niveau *Cycle Accurate, Bit Accurate* (CABA) ;
- le niveau *Transaction Level Modeling with Time* (TLM-T).

Chacun des niveaux de modélisation offre un compromis que nous présentons dans les sections suivantes.

Niveau porte

Un composant matériel est décrit comme une interconnexion de portes logiques caractérisées pour un procédé de fabrication choisi. Pour chaque type de porte, la surface de silicium occupée, les capacités, et les temps de propagation du courant électrique sont connus.

La simulation à partir d'une telle description est lente. La modélisation au niveau porte permet d'obtenir des informations telle que la fréquence d'horloge du circuit par exemple.

Niveau RTL

L'horloge et les registres de l'architecture sont identifiés. Les modèles au niveau RTL décrivent le comportement en termes de transferts entre registres.

La simulation de composants à ce niveau est plus rapide que celle des composants au niveau porte. De plus, ce type de modèle est en règle générale synthétisable : les outils de synthèse logique sont capables de convertir un modèle de niveau RTL vers le niveau porte.

Niveau CABA

Les modèles au niveau CABA sont précis au bit et au cycle près sur les interfaces. Un tel composant peut être vu comme une boite noire. Ces modèles ne sont pas directement synthétisables.

La simulation au niveau CABA est plus rapide qu'au niveau RTL.

Niveau TLM-T

Les communications entre les modules matériels sont modélisées sous forme de transactions datées.

La simulation au niveau TLM-T est la plus rapide à l'heure actuelle. La fonctionnalité de l'architecture est fidèlement modélisée mais la précision en nombre de cycles n'est pas garantie.

Choix du niveau d'abstraction pour l'exploration architecturale

La vitesse de simulation et la précision des évaluations dépendent du niveau d'abstraction des modèles de l'architecture matérielle. La figure 1.4 récapitule le compromis entre la vitesse de simulation et la précision des modèles en fonction du niveau d'abstraction.

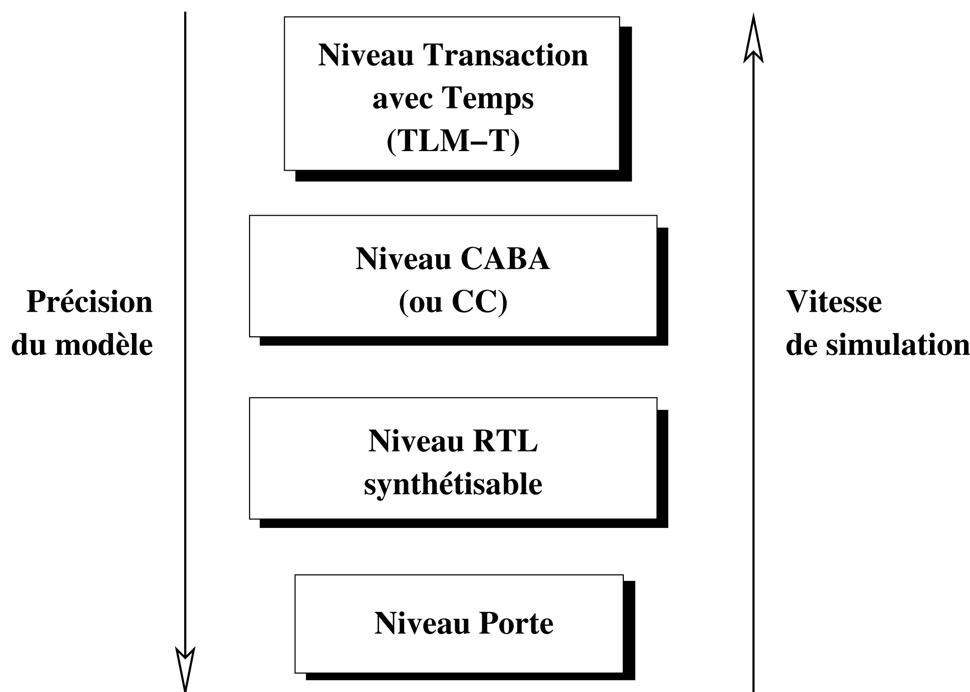

FIG. 1.4 – Compromis performance/précision en fonction du niveau d'abstraction

Le niveau TLM-T ne garantit pas une évaluation précise des performances du système au cycle près. En effet, les dates d'occurrence des transactions peuvent accepter une erreur de plusieurs cycles.

Les trois autres niveaux de modélisation - porte, RTL synthétisable et CABA - permettent d'obtenir une évaluation précise de la consommation d'énergie du système intégré, et des performances temporelles du système. Le niveau CABA offre une rapidité d'évaluation supérieure aux deux autres.

Nous nous intéressons dans cette thèse au niveau d'abstraction CABA car ce niveau offre un bon compromis pour l'exploration architecturale.

Nous avons vu que la vitesse d'exécution dépend du niveau d'abstraction mais aussi du moteur de simulation.

1.3.2 Moteur de simulation

Les architectures que nous souhaitons simuler contiennent uniquement des modules décrits au niveau CABA. Chacun de ces modules possède un ou plusieurs processus décrivant son comportement. Le principal rôle du cœur du simulateur est l'ordonnancement de l'exécution de ces processus sur la station de travail.

L'ordonnancement consiste à établir un ordre d'exécution séquentiel des processus. Si cet ordre est construit au cours de l'exécution de la simulation, on dit que l'ordonnancement est dynamique.

Le cœur du simulateur inclut généralement un moteur de simulation à évènements discrets. Ce type de moteur permet d'obtenir un ordonnancement dynamique. Lorsque la valeur d'un signal ou d'un port *I* change, un évènement $Event_I$ est généré. Chaque processus est associé à un ensemble d'évènements, que l'on appelle sa liste de sensibilité. Lorsqu'un évènement de la liste est généré, l'échéancier exécute le processus correspondant.

L'exécution d'un processus est susceptible de modifier certains signaux qui généreront à leur tour des évènements. La simulation continue tant que des évènements sont générés. Le système est dit stable lorsqu'aucun évènement n'est en attente au temps courant, et le simulateur peut continuer en incrémentant le temps simulé.

Un échéancier dynamique produit un ordonnancement au fur et à mesure de la simulation. Malheureusement, les informations dont il dispose sont insuffisantes pour déduire un ordre optimal. Le schéma 1.5 illustre, à l'aide d'une architecture à deux modules, ce problème. Chaque module possède un seul processus. Nous avons donc deux processus : *A* et *B*. L'évènement $Event_{I2}$ est dans la liste de sensibilité du processus *A*. Les évènements $Event_{I1}$ et $Event_{S1}$ sont dans la liste de sensibilité du processus *B*.

Si les valeurs des signaux *I1* et *I2* changent simultanément, deux évènements $Event_{I1}$ et $Event_{I2}$ sont générés. Les processus *A* et *B* sont réveillés, au même moment, respectivement par $Event_{I2}$ et $Event_{I1}$. Lorsque le processus *A* s'exécutera, le signal *S1* sera modifié et produira un évènement $Event_{S1}$ qui réveillera une deuxième fois le processus *B*. Selon l'implémentation de l'échéancier, l'ordre effectif des réveils de processus sera un ordre parmi les deux suivants :

- *B*, *A*, puis *B* de nouveau ;
- *A*, *B*, puis *B* de nouveau.

L'ordre résultant d'un ordonnancement dynamique introduit un réveil inutile du processus *B*. Or, ces réveils inutiles augmentent le temps d'exécution et donc diminuent la vitesse de simulation. Un ordre optimal serait : *A* puis *B*.

C'est pourquoi nous nous intéressons aux techniques d'ordonnancement statique, où un ordre optimal d'activation des processus est calculé, une fois pour toute, avant la simulation. Un tel ordonnancement nécessite des informations supplémentaires. Nous souhaitons donc enrichir les modèles par l'ajout d'informations utilisables par un outil d'ordonnancement statique.

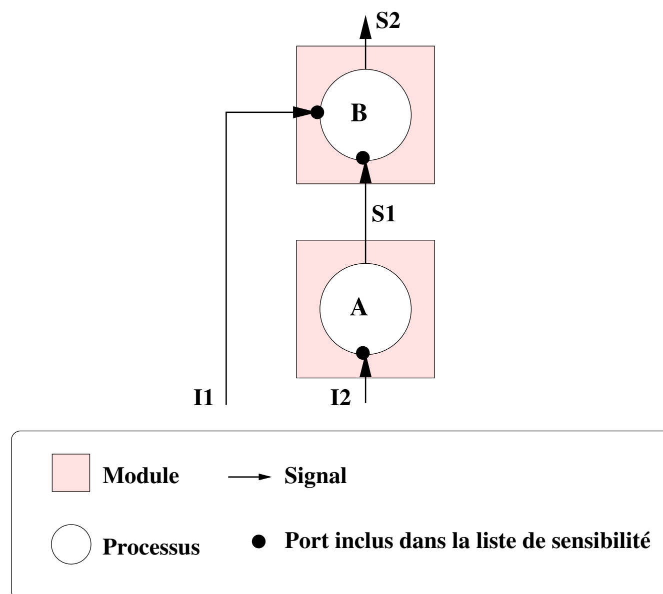

FIG. 1.5 – Exemple d'architecture matérielle impliquant des réveils inutiles de processus

1.3.3 Modélisation pour l'ordonnancement statique

Le rôle d'un échéancier à ordonnancement statique est de déterminer un ordre optimal d'évaluation des processus avant l'exécution de la simulation. Pour cela, l'échéancier a besoin d'identifier les relations de précédence des processus. Ces relations de précédence ne sont pas fournies par le concepteur. Ces informations doivent donc être construites par l'échéancier.

Une méthode consiste à expliciter, pour chaque module, les dépendances existantes entre les signaux de sortie et les signaux d'entrée. Dans cette thèse, nous abordons les contraintes que doivent respecter les modèles pour garantir la calculabilité d'un ordonnancement statique optimal. L'échéancier repose sur l'hypothèse que les informations supplémentaires explicitant les relations de dépendance entre signaux de sorties et signaux d'entrée d'un même module sont exactes. La section suivante propose une méthode pour vérifier cette hypothèse.

1.3.4 Vérification sémantique

Nous venons de proposer un échéancier à ordonnancement statique reposant sur un certain nombre d'hypothèses. Les hypothèses posées sont les suivantes :

- les modèles sont CABA ;
- les modèles fournissent des informations de dépendances des signaux d'entrée/sortie permettant de calculer l'ordonnancement statique.

Si ces informations de dépendance ne sont pas vérifiées, l'échéancier raisonne à partir de données erronées. Le simulateur devient alors non déterministe. Les résultats de simulation se-

ront donc potentiellement incorrects.

La vérification sémantique des modèles avant simulation permet de prévenir l'occurrence de ce cas de figure. La vérification sémantique est une des contributions de cette thèse.

1.3.5 Génération de modèles pour la simulation rapide

De nombreux modèles synthétisables de niveau RTL existent dans l'industrie. Étant destinés à la synthèse logique, ces modèles offrent de très mauvaises performances en terme de vitesse de simulation.

L'écriture manuelle d'un modèle CABA à partir d'une version RTL peut être longue et fastidieuse. Les erreurs humaines sont fréquentes. La génération automatique par un outil permettrait de réduire les temps de conversion ainsi que les risques d'erreurs.

Nous nous intéressons donc à la génération automatique de modèles de simulation de niveau CABA à partir des modèles RTL synthétisables. Ce travail constitue l'une des contributions de cette thèse.

1.4 Conclusions

Au cours de ce chapitre, les différentes étapes de l'exploration architecturale des systèmes intégrés sur puce ont été brièvement présentées. Nous avons décrit une des méthodes de modélisation du matériel, et de déploiement du logiciel sur le matériel. Nous avons ensuite identifié le problème de la durée de l'exploration architecturale qui devient excessive en raison du grand nombre de paramètres du système.

Étant donné que la durée de l'exploration architecturale dépend de la vitesse de la simulation architecturale, le problème revient à accélérer l'exécution de la simulation. Nous avons discerné deux facteurs majeurs : le niveau d'abstraction des modules et le moteur de simulation.

Le niveau de modélisation qui nous intéresse est le niveau CABA. La restriction quant au niveau de modélisation permet d'envisager l'emploi d'un simulateur spécialisé orienté vers la simulation rapide.

Nous avons identifié alors l'ordonnancement statique des processus comme étant une bonne source d'accélération. Cette accélération peut être obtenue à condition d'enrichir les modèles avec des informations explicitant les relations de dépendance entre les signaux d'entrée/sortie d'un même module. Si ces informations sont partielles ou incorrectes, le déterminisme de la simulation est compromis. Nous avons alors discerné deux besoins. Le premier besoin est d'établir une méthode de modélisation où les dépendances entre les signaux d'entrée/sortie apparaissent clairement. Le second besoin est de vérifier les relations de dépendance fournies.

L'exploration architecturale est limitée par la disponibilité des modèles de simulation rapide. Or, nous avons constaté que les modèles RTL synthétisables sont très répandus. La génération automatique de modules CABA à partir de ces modèles RTL est donc un enjeu important.

Au cours de ce chapitre, nous avons soulevé les questions suivantes auxquelles nous essaierons de répondre :

- Comment modéliser, de manière efficace, le comportement d'un composant matériel au niveau CABA ?
- Comment accélérer la simulation de composants CABA ?
- Quelles sont les règles de validité d'un modèle de simulation rapide ?
- Comment vérifier automatiquement le respect de ces règles ?
- Comment générer automatiquement les modèles de simulation rapide ?

Les chapitres suivants tentent d'apporter des réponses aux questions que nous avons posé tout au long de cette problématique.

Plan de la thèse

Nous présentons d'abord, dans le chapitre 2, une méthode de modélisation de composants CABA pour la simulation rapide.
L'algorithme d'ordonnancement abordé au chapitre 3 permet une accélération significative lors de la simulation de ces modèles.
Le chapitre 4 décrit une méthode de génération de modèles CABA pour accroître le nombre de modèles disponibles.
Dans le chapitre 5, nous décrivons une liste de règles d'écriture. Ces règles d'écriture permettent de vérifier si un modèle respecte notre méthode de modélisation.
Les résultats expérimentaux sont exposés et commentés dans chacun des chapitres.
Nous concluons ensuite sur l'intérêt des méthodes proposées. Puis nous exposons plusieurs perspectives à nos travaux.

Chapitre 2

Modélisation précise au cycle et au bit près sur les interfaces

Sommaire

2.1 État de l'art 14

2.1.1 CASS : Modélisation d'architectures multiprocesseurs 16

2.1.2 FastSysC : Modélisation au niveau micro-architecture 19

2.1.3 Problèmes non résolus 21

2.2 Modèle d'architecture en CFSM 21

2.2.1 Modélisation d'un composant matériel 22

2.2.2 Relations de dépendances combinatoires entre les signaux 26

2.2.3 Intérêt du modèle des automates synchrones communicants 26

2.3 Représentation en SystemC 27

2.3.1 Représentation du comportement d'un composant matériel 27

2.3.2 Définition de l'interface matérielle 27

2.3.3 Représentation des fonctions 28

2.3.4 Représentation des listes de sensibilité 28

2.3.5 Représentation des dépendances entre les signaux d'entrée et les signaux de sortie 29

2.3.6 Représentation des registres 30

2.3.7 Exemples 30

2.4 Conclusions 37

Lors de l'exploration architecturale, le concepteur de systèmes intégrés cherche la meilleure solution de déploiement d'une application logicielle sur une architecture matérielle paramétrable. Pour cela, le concepteur modifie les paramètres de l'architecture et lance de nombreuses simulations jusqu'à l'obtention d'une solution satisfaisant les contraintes.

La durée des simulations devient excessive à cause de l'accroissement de la taille et de la complexité des systèmes intégrés sur puce. La vitesse de simulation est donc une caractéristique importante que nous cherchons à maximiser.

L'architecture matérielle d'un système intégré sur puce est constituée de plusieurs composants matériels. Les principaux composants sont des processeurs, des bancs mémoire, des coprocesseurs spécialisés, des contrôleurs de bus et/ou des réseaux d'interconnexions. Ces composants sont interconnectés par des signaux.

Le comportement de chaque composant matériel est décrit, dans le simulateur, sous la forme d'un ou plusieurs processus. Le principal rôle de ces processus est d'évaluer les signaux de sortie des composants en fonction des valeurs des signaux qui se présentent sur les entrées, et en fonction de leur état interne. Sachant que le niveau d'abstraction joue un rôle majeur dans la vitesse d'exécution, nous nous intéressons uniquement à la modélisation au niveau CABA.

Le paramétrage de l'architecture matérielle est généralement facilité par l'utilisation de composants matériels réutilisables. La réutilisabilité des composants dépend de l'approche de modélisation adoptée. Nous apporterons donc une attention particulière aux critères de réutilisabilité lors de la modélisation de composants matériels.

Ce chapitre présente l'état de l'art en matière de modélisation de composants matériels précis au cycle et au bit près. Nous développons ensuite notre méthode de modélisation orientée vers la simulation rapide. Enfin, une représentation en SystemC sera décrite et accompagnée d'exemples.

2.1 État de l'art

Les modèles synthétisables RTL décrivent explicitement l'architecture interne, et en particulier les registres définissant l'état interne des composants. L'architecture interne décrite par un modèle RTL synthétisable est strictement identique à l'architecture interne de la réalisation matérielle.

Les modèles au niveau CABA sont précis au bit et au cycle près sur les interfaces. Un tel module se comporte comme une boîte noire. L'architecture interne d'un modèle CABA peut utiliser une représentation des états internes différente de la réalisation matérielle. La modélisation CABA de l'architecture interne des composants peut donc être simplifiée par rapport au modèle RTL synthétisable. Cependant, les comportements d'un même composant décrit au niveau RTL et au niveau CABA sont identiques aux interfaces.

Nous présentons, à titre d'exemple, deux implémentations possibles d'un même composant matériel : le processeur MIPS R3000 [Kan88][Goo92]. La première implémentation est au niveau RTL et décrit une architecture interne identique à la réalisation matérielle. La seconde

implémentation est au niveau CABA et modélise une architecture interne différente tout en respectant le comportement sur les interfaces.

La réalisation matérielle du processeur pipeline *MIPS R3000* comporte cinq étages de pipeline : *instruction fetch* (I), *decode* (D), *execute* (E), *memory access* (M), *write back* (W). Chaque instruction du processeur est découpée en cinq parties I, D, E, M et W correspondant aux cinq étages du pipeline. L'exécution d'une partie nécessite un cycle. La latence d'exécution d'une instruction est donc de cinq cycles.

Chaque étage du pipeline travaille, en parallèle, sur une instruction différente. Cinq cycles sont nécessaires pour évaluer complètement une instruction. Le processeur traite cinq instructions en cinq cycles. Le débit d'un tel processeur est donc d'une instruction par cycle.

Modélisation en RTL synthétisable du MIPS R3000 L'implémentation RTL synthétisable modélise fidèlement chaque étage du pipeline.

La figure 2.1 illustre l'architecture interne d'un *MIPS R3000* dont le modèle est strictement identique à la réalisation matérielle. Au cycle t, le modèle évalue l'étage F pour l'instruction $i+4$, l'étage D pour l'instruction $i+3$, l'étage E pour l'instruction $i+2$, l'étage M pour l'instruction $i+1$, l'étage W pour l'instruction i.

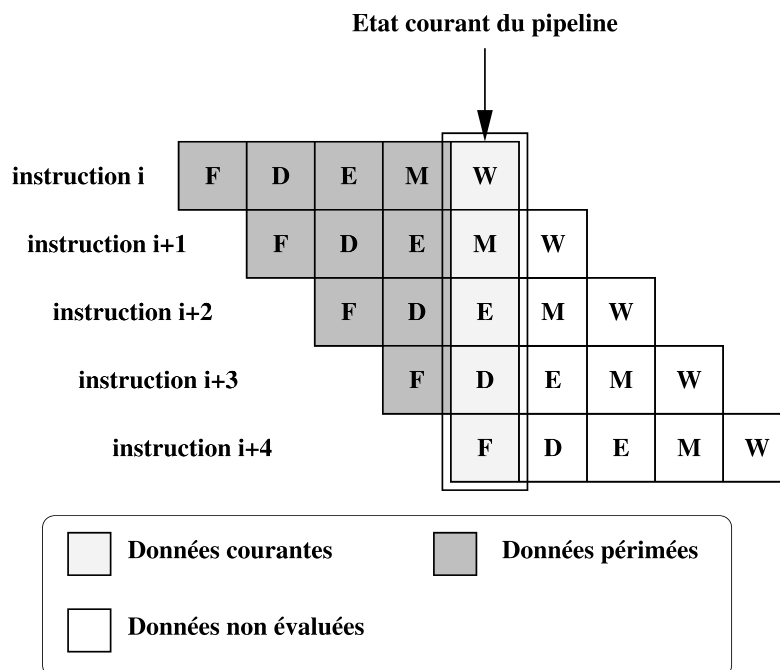

FIG. 2.1 – Modélisation au niveau RTL d'un processeur à 5 étages de pipeline

Modélisation CABA du MIPS R3000 avec 2 étages de pipeline Une implémentation CABA possible modélise une architecture interne simplifiée comportant un nombre d'étages de pipeline inférieur à la réalisation matérielle dès lors que l'on peut respecter le comportement sur les interfaces. Cette modélisation CABA comporte seulement deux étages :

- Le premier modélise l'*instruction fetch* (F),
- Le second modélise les quatre autres étages du pipeline : *decode*, *execute*, *memory access*, et *write back*.

La figure 2.2 illustre la modélisation du MIPS R3000 en 2 étages de pipeline. Le deuxième étage est noté IE' pour *instruction execute*. Le comportement aux interfaces est identique à la réalisation matérielle.

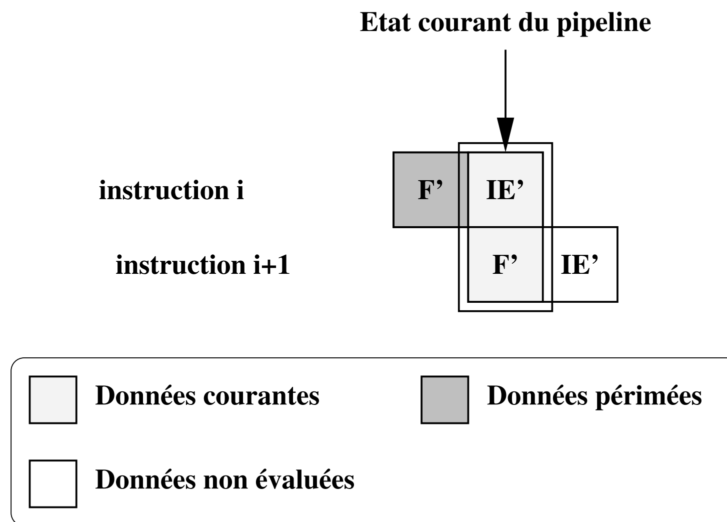

FIG. 2.2 – Modélisation simplifiée du MIPS R3000 en 2 étages de pipeline

L'architecture interne à deux étages permet d'exécuter en parallèle deux instructions en deux cycles. Le débit est également d'une instruction par cycle. Une telle architecture respecte fidèlement le comportement de la réalisation matérielle.

Dans les sections suivantes, nous présenterons deux approches de modélisation. La première vise la modélisation de systèmes multiprocesseurs dans le cadre d'une simulation rapide. La deuxième vise la modélisation de processeur au niveau micro-architecture. Les temps d'exécution des simulations sur la station de travail sont comparés afin de quantifier l'efficacité des deux approches.

2.1.1 CASS : Modélisation d'architectures multiprocesseurs

Le simulateur *Cycle Accurate System Simulator* (CASS) [PHG97][Den01], proposé par Denis Hommais et Frédéric Pétrot, cible la simulation rapide de systèmes intégrés sur

puce. Pour cela, les modules doivent respecter des contraintes de modélisation que l'échéancier [CM81][SMB87][WM90] exploite afin de construire un ordonnancement quasi statique.

CASS accepte des modèles écrits en langage *C* respectant un modèle formel imposé. Le modèle formel est celui des automates synchrones communicants [Moo56][Mea55][MP63][eDS64]. La figure 2.3 représente un ensemble fini d'automates communicants. Chaque automate comporte des entrées, des registres, des sorties, et deux types de fonctions. Tout module, ou toute architecture matérielle peut se ramener à ce type de représentation. Cette représentation autorise alors une optimisation importante de l'échéancier.

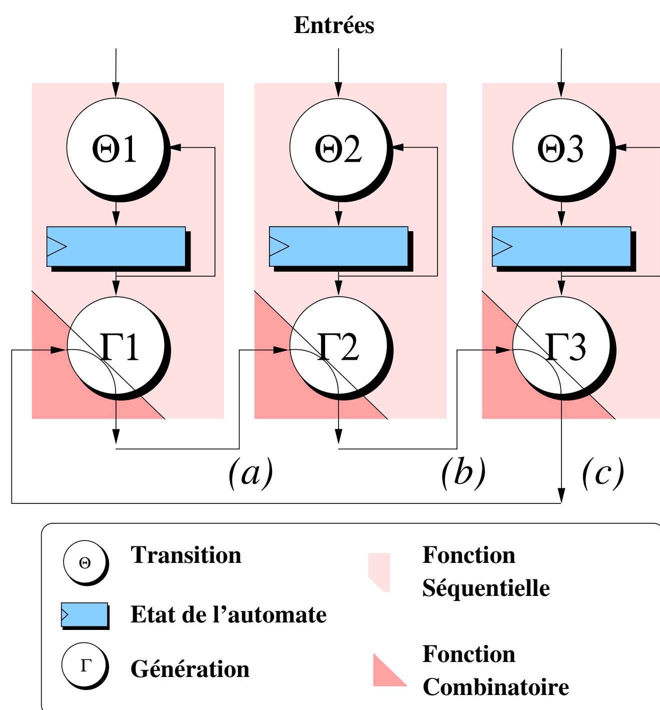

FIG. 2.3 – Automates à états finis communicants

Pour chaque composant matériel, le concepteur écrit deux fonctions. Ces deux fonctions modélisent d'une part, le comportement séquentiel, et, d'autre part, le comportement combinatoire. Le comportement séquentiel est décrit sous la forme d'un processus synchrone avec l'horloge de l'architecture simulée. Ce processus séquentiel calcule les états internes du composant et les signaux de sortie qui ne dépendent que de l'état interne. Le processus combinatoire calcule les signaux de sortie qui dépendent directement des valeurs des entrées.

La fonction séquentielle implémente les transitions d'état de l'automate et la fonction de génération des sorties de Moore. La fonction combinatoire implémente la génération des sorties de Mealy.

Le concepteur doit fournir explicitement la liste des sorties de Mealy pour chaque module.

L'échéancier de CASS raisonne sur les dépendances entre modules pour obtenir les relations de précédence entre processus. Cet échéancier génère alors un ordonnancement quasi statique comportant :

- Une liste statique des appels de fonction séquentielle dans n'importe quel ordre ;
- Une boucle de relaxation contenant une liste ordonnée des appels de fonction combinatoire.

L'ordonnancement généré par CASS a donc les propriétés suivantes :

- Les fonctions séquentielles sont exécutées une et une seule fois par cycle ;
- Les fonctions combinatoires sont exécutées au moins une fois par cycle, jusqu'à stabilité.

Toute la partie combinatoire de chaque module est modélisée par une seule et même fonction. Il y a donc autant de fonctions combinatoires que de modules dans l'architecture matérielle. Cette fonction calcule, à chaque exécution, tous les signaux de sortie du module qui dépendent directement des entrées. Les signaux de sortie d'un même module ne peuvent donc être évalués indépendamment. Lorsqu'un signal de sortie doit être évalué, la fonction combinatoire est exécutée, provoquant ainsi l'évaluation de l'ensemble des signaux du module. Dans un tel cas, l'ordonnancement quasi statique est susceptible d'exécuter plusieurs fois la même fonction combinatoire. L'exécution répétée d'une fonction combinatoire implique alors l'évaluation redondante de certaines expressions. Ce phénomène ralentit la simulation. L'exemple suivant illustre le problème.

La figure 2.4 présente une architecture matérielle comportant deux fonctions combinatoires. La fonction combinatoire F_a possède deux sorties S_2 et S_4. La sortie S_2 dépend de S_1 ; et la sortie S_4 dépend de S_3. La fonction combinatoire F_b calcule la valeur du signal S_3 en fonction de S_2.

La simulation de l'architecture matérielle, pour un cycle d'horloge donné, nécessite l'évaluation de tous les signaux. L'ordre optimal d'évaluation des signaux de cet exemple est trivial : S_1, S_2, S_3, puis S_4. L'ordre optimal d'exécution des fonctions est : F_a, F_b, puis F_a. Nous pouvons remarquer que l'exécution des fonctions provoque une évaluation redondante des signaux S_2 et S_4.

Bilan

La séparation de processus en deux parties, l'une séquentielle, l'autre combinatoire, permet de construire un ordonnancement quasi statique performant. Les résultats expérimentaux ont montré que cet ordonnancement permet une simulation extrêmement rapide - jusqu'à dix fois plus rapide qu'un moteur utilisant un échéancier à ordonnancement dynamique. Ce choix comporte cependant un défaut : certaines fonctions combinatoires sont parfois évaluées plusieurs fois au cours du même cycle, avant d'atteindre la stabilité du système. Bien que l'ordonnancement des fonctions soit optimal, la simulation souffre de l'évaluation redondante de certains signaux. Plus les fonctions combinatoires sont coûteuses et/ou interdépendantes, plus les calculs redondants sont pénalisants en terme de vitesse d'exécution.

La modélisation au niveau micro-architecture introduit des modèles dont le comportement est majoritairement combinatoire. CASS est donc peu performant pour traiter ce type de modé-

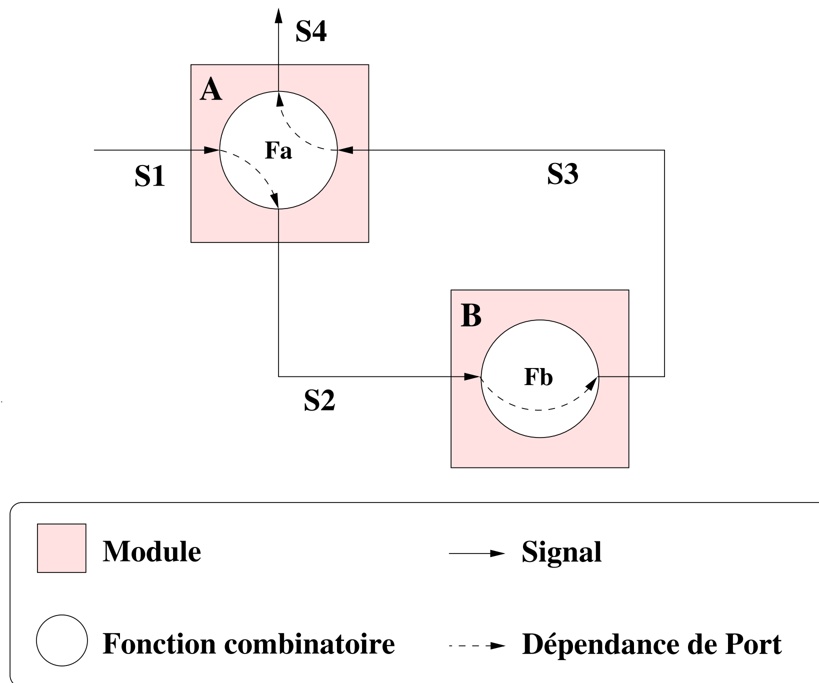

FIG. 2.4 – Exemple d'architecture matérielle problématique pour CASS

lisation.

2.1.2 FastSysC : Modélisation au niveau micro-architecture

Le simulateur *FastSysC* [PMT04], présenté dans la thèse de Gilles Mouchard à l'université de Paris XI, cible la simulation de modèles de processeur au niveau micro-architecture. Ces processeurs sont généralement des architectures complexes de type super-scalaires. La description de tels processeurs se présente sous la forme d'une hiérarchie de modules interconnectés par des signaux internes. Les modules élémentaires de la hiérarchie implémentent des mécanismes tels que le décodage d'instruction, l'exécution d'opération arithmétique, la prédiction de branchement, l'exécution d'instructions dans le désordre, l'exécution spéculative, ou encore l'exécution multi-contextes. La plupart de ces modules décrivent un comportement combinatoire et la majorité des signaux sont pilotés par des processus combinatoires.

FastSysC vise donc l'accélération de la simulation de modèles de processeur ayant les propriétés suivantes :

- La micro-architecture est très détaillée ;
- Les modules sont précis au cycle sur les interfaces.

La modélisation de modèles de processeur au niveau micro-architecture introduit un nombre important de signaux pilotés par des processus combinatoires. Un échéancier à ordonnancement

dynamique ne peut déterminer un ordre optimal d'exécution de ces processus. Certains processus sont alors exécutés plusieurs fois au cours du même cycle. La simulation se révèle très lente dans ces conditions.

FastSysC repose sur un modèle de calcul de type évènements discrets mais son moteur de simulation tente d'accélérer la simulation en réduisant la redondance des exécutions de processus à l'aide d'un échéancier travaillant en deux temps. Dans un premier temps, l'échéancier utilise un ordonnancement incomplet statique, dit acyclique. L'ordonnancement étant incomplet, les signaux ne sont pas tous évalués. Dans un second temps, l'échéancier continue l'exécution des processus jusqu'à stabilité du système à l'aide d'un ordonnancement dynamique classique. L'objectif du premier ordonnancement statique incomplet est de réduire le coût de l'ordonnancement dynamique.

Pour atteindre cet objectif, *FastSysC* impose, au concepteur de modèles, de décrire explicitement les relations de dépendances entre les signaux d'entrée et les signaux de sortie. L'échéancier exploite ces informations avant l'exécution de la simulation pour construire un ordonnancement statique incomplet.

Comme CASS, *FastSysC* définit deux types de processus : les processus séquentiels et les processus combinatoires. Le type du processus est obtenu par le simulateur en consultant sa liste de sensibilité.

Les processus séquentiels sont synchrones et n'ont pas de dépendances de données entre eux. L'ordre d'exécution des processus séquentiels n'a donc aucune importance. Les processus sont exécutés une seule fois par cycle, dans n'importe quel ordre. Des informations supplémentaires sont nécessaires pour ordonnancer efficacement les processus combinatoires. *FastSysC* propose au concepteur d'ajouter explicitement des informations considérées comme facultatives.

Le concepteur a la possibilité de spécifier explicitement la liste des dépendances entre les signaux d'entrée et les signaux de sortie pour chaque processus combinatoire. Ces informations supplémentaires permettent de construire un ordonnancement acyclique des signaux avant simulation. Si la liste des dépendances est incomplète ou erronée, l'ordonnancement acyclique ne garantit pas l'évaluation correcte de chaque signal. Quel que soit le résultat de ce premier ordonnancement, l'échéancier à ordonnancement dynamique continue l'exécution des processus jusqu'à stabilité du système. Cet échéancier à ordonnancement dynamique assure que l'exécution de la simulation soit correcte et constitue donc le garde-fou de l'ordonnancement acyclique.

L'échéancier en deux temps de *FastSysC* raisonne sur les dépendances entre signaux ainsi que sur les listes de sensibilité pour générer un ordonnancement tel que :

- Les fonctions séquentielles sont exécutées une et une seule fois par cycle ;
- Les fonctions combinatoires sont exécutées exactement une fois, puis à chaque fois qu'un événement apparaît sur leur liste de sensibilité.

Bilan

FastSysC est un moteur pour la simulation de modèles de processeur détaillés au niveau micro-architecture. Ce moteur se présente sous la forme d'une *Application Program Inter-*

face (API) SystemC implémentant un sous-ensemble du langage. La solution apportée par *FastSysC* pour augmenter la vitesse de simulation consiste à modifier le moteur à évènements. Pour cela, le concepteur fournit une liste de dépendances entre les signaux d'entrée et les signaux de sortie pour les processus combinatoires. À l'aide de ces informations, le moteur construit une partie de l'ordonnancement de manière statique. La simulation continue jusqu'à stabilité avec un ordonnancement dynamique. L'échéancier de FastSysC apporte un facteur d'accélération de la simulation entre deux et trois par rapport à un simple échéancier dynamique.

2.1.3 Problèmes non résolus

La modélisation proposée pour CASS permet des simulations très rapides si l'architecture comporte peu de signaux pilotés par des processus combinatoires, et si ces processus combinatoires nécessitent peu de calculs. *FastSysC* propose une approche de modélisation qui permet l'accélération d'architecture comportant de nombreux signaux pilotés par des processus combinatoires.

La modélisation de composants matériels proposée dans le cadre des simulateurs CASS et *FastSysC* ne résout pas les problèmes suivants :

- Comment modéliser des composants matériels pour éviter complètement l'évaluation redondante des signaux ?
- Comment modéliser la partie combinatoire des composants matériels pour permettre un ordonnancement totalement statique ?

La contribution de ce chapitre s'inspire des travaux présentés dans cet état de l'art et propose une approche de modélisation pour répondre aux problèmes jusqu'alors non résolus : éviter complètement l'évaluation redondante des signaux et permettre un ordonnancement totalement statique.

2.2 Modèle d'architecture matérielle en automates synchrones communicants

L'architecture matérielle des systèmes intégrés sur puce que nous cherchons à modéliser est généralement multiprocesseur, et comporte plusieurs bancs mémoire, un ou plusieurs coprocesseurs spécialisés, organisés autour d'un micro réseau d'interconnexions ou d'un bus. Les modèles CABA des composants matériels utilisent une représentation des états internes, simplifiée par rapport au modèle RTL synthétisable, si possible. Notre méthode de modélisation poursuit deux objectifs :

- Écrire des modèles de simulation rapide ;
- Réduire le temps nécessaire pour construire une architecture complète multiprocesseur.

Une solution consiste à exploiter une bibliothèque de composants matériels réutilisables. La construction d'une architecture matérielle se résume alors à paramétrer et à connecter les modules de la bibliothèque selon les besoins. C'est l'idée de base du projet *System on Chip*

LIBrary (SoCLIB).

La constitution d'une telle bibliothèque requiert un effort de modélisation supplémentaire. Le coût des efforts peut être cependant partagé par les différents utilisateurs de la bibliothèque.

Notre approche de modélisation de composants réutilisables repose sur le modèle formel des automates. Une architecture matérielle multiprocesseur se comporte comme un ensemble d'automates synchrones communicants. Nous montrerons que ce type de modélisation offre des opportunités d'accélération de la simulation.

Dans les sections suivantes, nous étudions la modélisation d'un composant matériel, puis la communication entre ces composants. Nous concluons ensuite sur l'intérêt qu'apporte ce type de modélisation.

2.2.1 Modélisation d'un composant matériel

La description d'une architecture matérielle est composée de deux types de composant. Les composants élémentaires décrivent le comportement du matériel. Les composants structurels décrivent une hiérarchie de composants structurels ou élémentaires.

Chaque composant élémentaire est modélisé par un ou plusieurs automates à états finis synchrones (*Communicating Finite State Machine* (CFSM)), communiquant entre eux par des signaux internes. Un module (ou modèle de composant) est caractérisé par un ensemble de ports, de registres, et de processus décrivant le comportement du ou des automates. Chaque port/registre/processus est identifié par le concepteur et déclaré explicitement comme tel.

Nous distinguons trois types de fonctions : transition, génération de *Moore*, génération de *Mealy*. Chaque module possède une unique fonction de transition, une unique fonction de génération de Moore, et enfin, $n \geq 0$ fonctions de génération de Mealy.

La figure 2.5 représente un ensemble fini d'automates communicants. Un module, ou une architecture multiprocesseur complète peut se ramener à ce type de représentation.

Chacune des fonctions a un rôle bien défini que nous décrivons dans les sections suivantes.

Fonction de transition

Le rôle de la fonction de transition est de déterminer le nouvel état du ou des automates du module. En pratique, cette fonction calcule les nouvelles valeurs à écrire dans les registres en fonction des valeurs courantes, contenues dans les registres, et des valeurs présentes sur les ports d'entrée.

Le calcul des valeurs futures des registres s'effectue en parallèle dans chacun des automates. Tous les registres mémorisent leur nouvelle valeur simultanément de manière synchrone. L'implémentation de la fonction de transition utilise un langage séquentiel tel que *C++* pour décrire un comportement parallèle ce qui pose un problème quand un même module contient plusieurs automates.

La figure 2.6 illustre deux automates communicants via leurs registres. L'automate A dépend des entrées I, et des registres REG_a et REG_b. De même, l'automate B dépend également

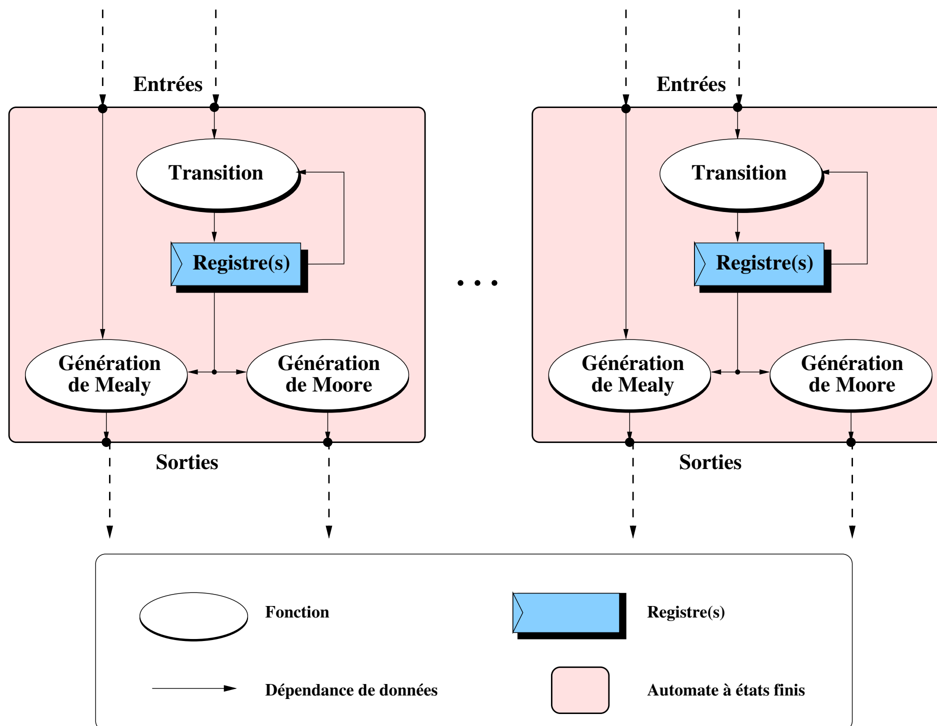

FIG. 2.5 – Automates à états finis communicants

des entrées I, et des registres REG_a et REG_b. La fonction de transition de ces deux automates comporte les affectations séquentielles suivantes :

- $REG_a = f_a(REG_a, REG_b, I)$
- $REG_b = f_b(REG_a, REG_b, I)$

Les affectations sont concurrentes. Si les écritures dans les registres étaient immédiates, la valeur courante d'une partie des registres serait écrasée par sa valeur future. Les transitions d'état des automates deviendraient alors erronées, et ce, quel que soit l'ordre d'exécution de ces deux affectations.

Les écritures séquentielles dans les registres doivent donc être postées (ou retardées avant d'être utilisée dans la 2ème affectation). Les écritures ne sont effectives qu'après l'exécution de toutes les fonctions de transition d'un même module.

La table 2.1 récapitule les opérations autorisées en fonction des types de données.

L'implémentation d'un reset synchrone se situe typiquement dans la fonction de transition.

Fonctions de génération

Les fonctions de génération assurent la mise à jour des sorties du module. Il en existe deux types : les fonctions de génération de **Moore** et de **Mealy**.

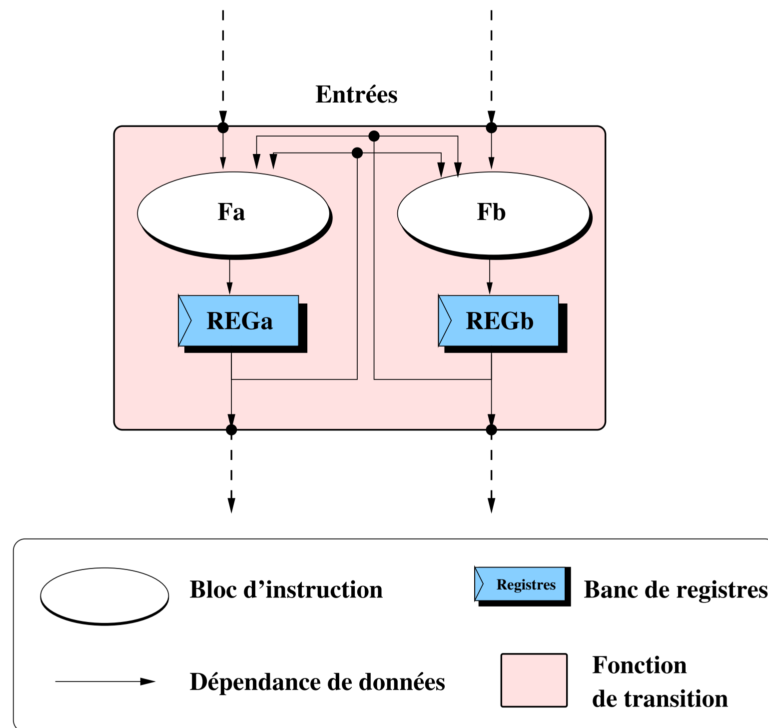

FIG. 2.6 – Fonction de transition comportant deux automates interdépendants

Type de données	Opérations autorisées
Entrées	Lectures
Sorties	Aucune
Registres	Lectures immédiates et Écritures retardées

TAB. 2.1 – Opérations autorisées dans une fonction de transition

Les deux sections suivantes décrivent la différence entre ces deux types de fonction de génération. La figure 2.5 illustre cette différence.

Fonction de génération de Moore

La fonction de génération de Moore permet de calculer les valeurs à émettre sur les ports de sortie qui dépendent uniquement des valeurs contenues dans les registres.

La table 2.2 fournit la liste des opérations autorisées en fonction des types de données.

Type de données	Opérations autorisées
Entrées	Aucune
Sorties	Écritures
Registres	Lectures

TAB. 2.2 – Opérations autorisées dans une fonction de génération de Moore

Fonction de génération de Mealy

La fonction de génération de **Mealy**, ou fonction combinatoire, permet de calculer les valeurs à émettre sur les ports de sortie quand celles-ci dépendent des valeurs contenues dans les registres et des valeurs présentes sur les ports d'entrée.

La table 2.3 est proche de la précédente table 2.2. La différence tient dans le fait que la lecture des entrées est autorisée.

Type de données	Opérations autorisées
Entrées	Lectures
Sorties	Écritures
Registres	Lectures

TAB. 2.3 – Opérations autorisées dans une fonction de génération de Mealy

Le comportement combinatoire d'un composant matériel est décrit par $n \geq 0$ fonctions de génération de Mealy. Chaque fonction de Mealy évalue au moins un signal de sortie de Mealy. Le concepteur peut écrire autant de fonctions de Mealy que nécessaire. Cependant, un signal est piloté par une seule fonction de Mealy. Le nombre maximum de fonctions de Mealy est, par conséquent, égal au nombre de signaux de sortie de Mealy du module.

La modélisation en plusieurs fonctions de Mealy permet d'évaluer séparément les différentes sorties du module. Le principal intérêt est d'éviter les évaluations redondantes d'expressions en cas d'exécutions répétées de la fonction.

La figure 2.4 présentée au début du chapitre illustrait le problème des évaluations redondantes. La figure 2.7 présente une autre modélisation de cette architecture dans laquelle le module A possède deux fonctions de génération de Mealy séparées F'_a et F''_a. Cette modélisation donne la possibilité d'obtenir un ordre d'exécution des fonctions pour lequel chaque signal est évalué exactement une fois : F'_a, F_b puis F''_a.

Pour chaque fonction de génération de Mealy, le concepteur doit fournir des informations, nécessaires à l'ordonnancement, sous la forme de relations de dépendances entre les signaux de Mealy.

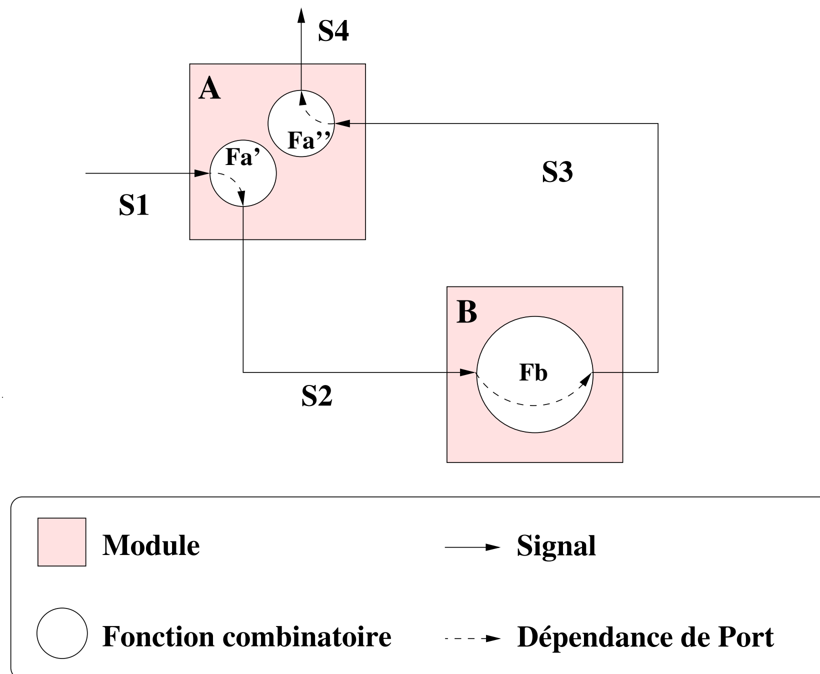

FIG. 2.7 – Exemple d'architecture matérielle avec trois fonctions de Mealy

2.2.2 Relations de dépendances combinatoires entre les signaux

La description des relations de dépendances entre les signaux de Mealy est une information dédiée au moteur de simulation. Ces informations permettent au moteur de simulation de construire un ordonnancement optimal.

Les relations de dépendances entre les signaux d'entrée et les signaux de sortie sont présentes uniquement dans les fonctions de génération de Mealy. Les fonctions de transition et les fonctions de génération de Moore comportent uniquement des dépendances entre des registres et des signaux. Par construction, les fonctions de Mealy sont donc les seules fonctions à posséder des dépendances entre signaux.

Pour fournir au moteur de simulation les informations nécessaires à l'ordonnancement, nous demandons au concepteur de déclarer explicitement, comme *FastSysC*, les dépendances entre les signaux d'entrée et les signaux de sortie pour chaque fonction de Mealy. Nous présentons dans le chapitre 3 une méthode et un outil permettant de vérifier la conformité de ces informations de dépendances par rapport au code des fonctions.

2.2.3 Intérêt du modèle des automates synchrones communicants

La modélisation CABA en automates synchrones communicants oblige le concepteur à modéliser le comportement d'un composant matériel à l'aide de $n+2$ processus. Les dépendances

de données entre ces processus sont identifiées. L'ordre d'exécution des processus est alors en partie déterminé. En effet, les fonctions de génération dépendent des registres. Les fonctions de génération sont donc exécutées après les fonctions de transition.

La modélisation en automates permet d'obtenir les garanties suivantes :

- Toutes les fonctions sont exécutées une et une seule fois ;
- Les fonctions de l'architecture sont exécutées dans l'ordre suivant :
  1. Transitions,
  2. Générations de Moore,
  3. Générations de Mealy.

Le principal intérêt du modèle en automate est la minimisation du nombre d'appels de fonction. Une représentation en SystemC est présentée dans la section suivante pour illustrer notre modélisation en automate.

2.3 Représentation en SystemC

SystemC est un langage de description de matériel. Ce langage se présente sous la forme d'une bibliothèque de classes C++ open-source.

Le succès de SystemC a conduit ce langage à devenir un nouveau standard dans le domaine de la conception au niveau système. De nombreux outils commerciaux (*Visual Elite FastC*, *ModelSim*, *Riviera*, *Visual Sim* etc.) supportent maintenant SystemC.

Nous avons mené de nombreuses expérimentations pour valider la modélisation CFSM CABA présentée dans ce chapitre. Nous avons développé des composants en SystemC respectant le modèle des CFSM afin de constituer une bibliothèque de composants réutilisables. La représentation en SystemC, utilisée lors de nos expérimentations, est décrite dans les sections suivantes.

2.3.1 Représentation du comportement d'un composant matériel

Un composant matériel élémentaire est représenté par une structure C++ héritant de la classe *sc_module*. Un module peut contenir :

- Des ports pour la communication entre modules ;
- Des fonctions pour décrire le comportement ;
- Des signaux pour modéliser des registres ;
- Des variables *C++* pour modéliser des constantes architecturales.

Le listing **2.1** présente un squelette de module SystemC.

2.3.2 Définition de l'interface matérielle

L'interface matérielle d'un module est composée d'un ou plusieurs ports. Chaque port est caractérisé par une direction et un type de données.

```
1  struct exemple : sc_module
2  {
3    // ports
4    // fonctions
5    // registres
6    // constantes architecturales
7
8    exemple (sc_module_name n) : sc_module (n)
9    {
10     // Déclaration des fonctions
11     // Pour chaque fonction , déclarations :
12     // – de la liste de sensibilité
13     // – des dépendances entre les ports de l'interface
14   }
15 }
```

Listings 2.1 – Description d'un module SystemC

Les ports sont des variables *C++*. Les types de port sont notés :

- *sc_in* : pour les ports d'entrée ;
- *sc_out* : pour les ports de sortie.

La modélisation CABA est précise au bit sur les interfaces. Les types de données utilisées sont les *sc_uint<N>* afin de préciser explicitement le nombre de bits.

2.3.3 Représentation des fonctions

Un module possède un constructeur, une fonction de transition, une fonction de génération de Moore, et $n \geq 0$ fonctions de génération de Mealy.

Le constructeur permet de déclarer les autres fonctions et d'initialiser des constantes architecturales telles que le numéro d'identification d'un processeur, la taille d'un cache, ou le nombre de ports VCI d'un micro réseau. Les registres ne sont pas initialisés dans le constructeur. Ils sont initialisés dans la fonction de transition par un mécanisme de reset si nécessaire.

Les fonctions de transition et de génération d'un module sont déclarées, dans le constructeur, en tant que *SC_METHOD*.

2.3.4 Représentation des listes de sensibilité

Il existe trois types de fonctions pour implémenter un automate. Les listes de sensibilité associées aux fonctions permettent de distinguer ces trois types de la façon suivante :

- La fonction de transition est sensible uniquement au front montant de l'horloge ;
- La fonction de génération de Moore est sensible uniquement au front descendant de l'horloge ;

- La fonction de génération de Mealy est sensible au front descendant de l'horloge et à au moins un port d'entrée.

Si une fonction de Mealy est déclarée pour chaque signal de sortie de Mealy, il est possible de rendre totalement explicite les dépendances combinatoires entre les signaux d'entrée et les signaux de sortie. L'ordonnancement est ainsi facilité.

2.3.5 Représentation des dépendances entre les signaux d'entrée et les signaux de sortie

Les dépendances entre les signaux d'entrée et les signaux de sortie sont déclarées explicitement par le concepteur. Chacune des dépendances est déclarée séparément sous la forme d'un couple : le port *A* dépend du port *B*.

Le listing 5.11 montre une implémentation des déclarations de dépendances pour un cache d'instructions et de données CABA. Dans cet exemple, le cache possède une fonction de Mealy pour le cache d'instructions, et une autre fonction de Mealy pour le cache de données.

Les déclarations de dépendances entre les ports d'entrée et les ports de sortie ne sont pas incluses dans la norme du langage *SystemC*. La syntaxe de ces déclarations constitue une extension de *SystemC*. Des directives de pré-compilation désactivent le code non standard si le moteur de simulation n'est pas compatible avec cette extension.

```
1    SC_METHOD ( generation_Mealy_instruction );
2     sensitive_neg << CLK;
3     sensitive     << ICACHE.ADDRESS;
4  #if defined(PORT_DEPENDANCIES_ENABLED)
5             ICACHE.MISS          (ICACHE.ADDRESS);
6             ICACHE.INSTRUCTION   (ICACHE.ADDRESS);
7  #endif
8
9    SC_METHOD ( generation_Mealy_donnee );
10    sensitive_neg << CLK;
11    sensitive     << DCACHE.TYPE;
12    sensitive     << DCACHE.ADDRESS;
13    sensitive     << DCACHE.REQUEST;
14    sensitive     << DCACHE.UNCACHED;
15 #if defined(PORT_DEPENDANCIES_ENABLED)
16            DCACHE.MISS      (DCACHE.TYPE);
17            DCACHE.MISS      (DCACHE.ADDRESS);
18            DCACHE.MISS      (DCACHE.REQUEST);
19            DCACHE.MISS      (DCACHE.UNCACHED);
20            DCACHE.READ_DATA(DCACHE.ADDRESS);
21 #endif
```

Listings 2.2 – Déclaration des dépendances entre les ports d'entrée et les ports de sortie

2.3.6 Représentation des registres

La fonction de transition est une fonction C++ qui décrit des affectations concurrentes sur les registres. La fonction de transition est donc une description séquentielle d'un comportement parallèle. Pour résoudre ce problème, les registres nécessitent un mécanisme d'écriture retardée (ou postée). Le type *sc_signal* implémente ce type de mécanisme.

Les registres sont des variables C++, membres du module et de type *sc_signal<type de la donnée mémorisée>*. Les types de données utilisés sont choisis par le concepteur selon l'implémentation de l'architecture interne du module.

2.3.7 Exemples

Deux modèles *SystemC* sont présentés, à titre d'exemples, dans les sections suivantes :
- Un additionneur accumulateur,
- Un coprocesseur *Direct Memory Access* (DMA) simplifié à interface VCI.

Chaque exemple est accompagné d'une description de l'architecture interne et du code source SystemC.

Additionneur accumulateur 32 bits

La structure contient un port de sortie *s* et quatre ports d'entrée : les deux opérandes *a* et *b*, le bit de sélection *sel*, et l'horloge *clk*. Le module contient un registre 32 bits mémorisant sur front montant.

La figure 2.8 présente l'architecture interne d'un additionneur accumulateur 32 bits. Le listing **2.3** est l'implémentation correspondante qui respecte la modélisation en SystemC au niveau CABA sous la forme d'automates synchrones communicants.

Les fonctions *transition* et *MOORE_generation* portent des noms explicites. Le moteur de simulation identifie le type des fonctions à l'aide de leur liste de sensibilité : *transition* est sensible au front montant de l'horloge ; et *MOORE_generation* est sensible au front descendant.

Notre méthodologie de modélisation en SystemC permet de décrire des composants matériels aussi petit qu'un additionneur accumulateur. L'écriture d'un tel module nécessite seulement quelques minutes.

DMA simplifié

Cet exemple est la description d'un composant simplifiée de type DMA. L'objectif de ce composant est d'implanter, en matériel, un mécanisme rapide de copie mémoire. Ce composant comporte deux interfaces VCI "initiateur" (ou maître). La première permet d'envoyer les requêtes de lecture des données d'un banc mémoire. La deuxième interface permet d'envoyer les requêtes d'écriture vers un autre banc mémoire. Le listing **2.4** déclare les ports externes du composant.

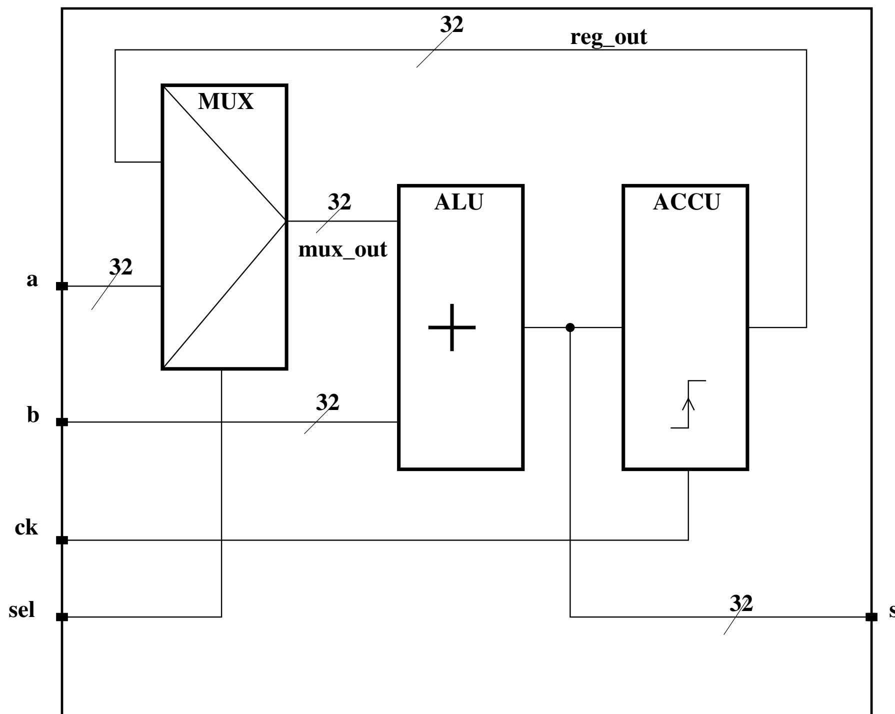

FIG. 2.8 – Additionneur accumulateur 32 bits

Le module possède deux automates synchrones et une mémoire tampon permettant de stocker une paire adresse/donnée. Le premier automate gère l'interface VCI pour les lectures. Le second gère l'interface pour les écritures. La figure 2.9 présente l'architecture interne du composant.

Le listing **2.5**. déclare les registres et les noms des états des automates. Les registres sont du type *sc_signal<...>*.

Les fonctions de transition des deux automates sont implémentées dans la fonction *transition*. Les fonctions de génération sont, quant à elle, dans la fonction *MOORE_generation*. L'implémentation des fonctions est fournie dans les listings **2.6** et **2.7**. Nous utilisons l'instruction *switch(X)* pour déterminer l'état courant de l'automate *X*. En fonction de cet état, nous effectuons le calcul du nouvel état dans la fonction de transition, et le calcul des sorties de l'automate dans la fonction de génération.

Notre méthodologie de modélisation permet de décrire des composants matériels complexes tels que des coprocesseurs DMA, des micro réseaux, ou des processeurs programmables.

```
1  struct addaccu : sc_module
2  {
3    sc_in <bool>    clock;
4    sc_in <int>     a;
5    sc_in <int>     b;
6    sc_in <bool>    select;
7    sc_out<int>     s;
8
9    sc_signal<int>  reg;
10
11   void transition ()
12   {
13     reg = ((select.read())?reg:a.read()) + b.read();
14   }
15
16   void MOORE_generation ()
17   {
18     s = reg.read();
19   }
20
21   SC_HAS_PROCESS (addaccu);
22   addaccu (sc_module_name n) : sc_module (n)
23   {
24     SC_METHOD(transition);
25     sensitive_pos << clock;
26     SC_METHOD(MOORE_generation);
27     sensitive_neg << clock;
28   }
29 };
```

Listings 2.3 – Additionneur accumulateur 32 bits décrit en SystemC CABA

```
1  #include "soclib_vci_interfaces.h"
2
3  template <unsigned short ADDRSIZE,
4            unsigned short CELLSIZE,
5            unsigned short ERRLEN>
6  struct VCI_MOVER : sc_module
7  {
8          //////////////////////
9          // EXTERNAL PORTS
10         //////////////////////
11         sc_in<bool>                                             CLK;
12         sc_in<bool>                                             RESETN;
13         ADVANCED_VCI_INITIATOR<ADDRSIZE,CELLSIZE,ERRLEN>        VCI[2];
14
15 ...
16
17 }; // fin struct VCI_MOVER
```

Listings 2.4 – DMA simplifié décrit en SystemC CABA : déclaration des ports externes

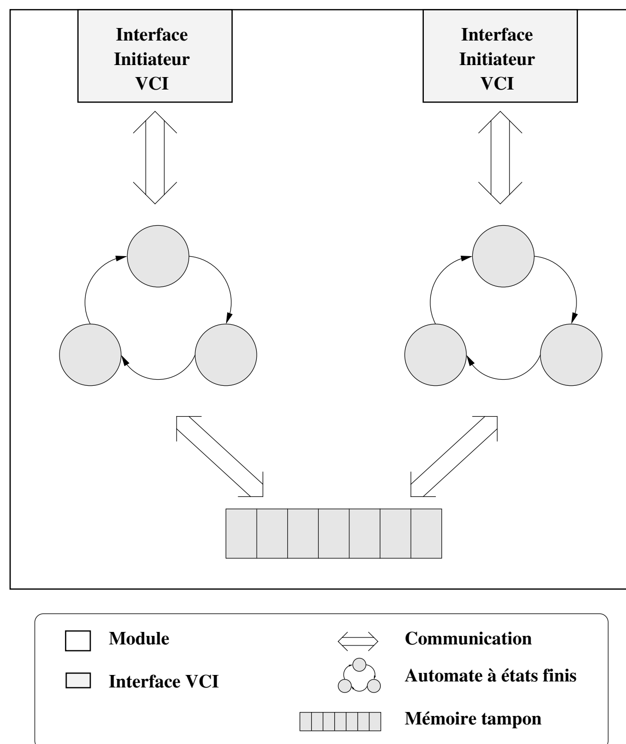

FIG. 2.9 – DMA simplifié à interface VCI

```
1  //////////////////////////
2  // FSM STATES
3  //////////////////////////
4  enum {
5  IDLE      = 0,
6  REQUEST   = 1,
7  RESPONSE  = 2
8  };
9  //////////////////////////
10 // REGISTERS
11 //////////////////////////
12 sc_signal<bool>                        BUF_VALID;
13 sc_signal<sc_uint<8 * CELLSIZE> >      BUF_DATA;
14 sc_signal<sc_uint<ADDRSIZE> >          BUF_ADDRESS;
15 sc_signal<sc_uint<ADDRSIZE> >          ADDRESS_COUNTER;
16 sc_signal<int>                         SOURCE_FSM_STATE;
17 sc_signal<int>                         DEST_FSM_STATE;
```

Listings 2.5 – DMA simplifié décrit en SystemC CABA : déclaration des registres

```
1  void transition()
2  {
3    if (RESETN == false) // initialisation
4    {
5      SOURCE_FSM_STATE   = IDLE;
6      DEST_FSM_STATE     = IDLE;
7      BUF_VALID          = false;
8      ADDRESS_COUNTER = 0;
9    } else {
10     switch(SOURCE_FSM_STATE) { // automate SOURCE_FSM_STATE
11     case IDLE:
12       SOURCE_FSM_STATE = REQUEST;
13       break;
14     case REQUEST:
15       if (VCI[0].CMDACK == true)
16         SOURCE_FSM_STATE = RESPONSE;
17       break;
18     case RESPONSE:
19       if ((BUF_VALID == false)
20       && (VCI[0].RSPVAL == true))
21       {
22         BUF_DATA        = VCI[0].RDATA;
23         BUF_ADDRESS     = ADDRESS_COUNTER;
24         BUF_VALID       = true;
25         SOURCE_FSM_STATE = IDLE;
26         ADDRESS_COUNTER = ADDRESS_COUNTER + 1;
27       }
28       break;
29     }
30     switch(DEST_FSM_STATE) { // automate DEST_FSM_STATE
31     case IDLE:
32       if (BUF_VALID == true)
33         DEST_FSM_STATE = REQUEST;
34       break;
35     case REQUEST:
36       if (VCI[1].CMDACK == true)
37       {
38         BUF_VALID = false;
39         DEST_FSM_STATE = RESPONSE;
40       }
41       break;
42     case RESPONSE:
43       if (VCI[1].RSPVAL == true)
44         DEST_FSM_STATE = IDLE;
45       break;
46     }
47   } // fin else RESETN
48 }
```

Listings 2.6 – DMA simplifié décrit en SystemC CABA : fonction de transition

```
1  void MOORE_generation()
2  {
3    switch(SOURCE_FSM_STATE) { // interface VCI[0]
4      case IDLE:
5        VCI[0].CMDVAL = false;
6        VCI[0].RSPACK = false;
7        break;
8      case REQUEST:
9        VCI[0].CMDVAL = true;
10       VCI[0].RSPACK = false;
11       VCI[0].ADDRESS = ADDRESS_COUNTER;
12       VCI[0].CMD     = VCI_CMD_READ;
13       break;
14     case RESPONSE:
15       VCI[0].CMDVAL = false;
16       if (BUF_VALID == false)
17         VCI[0].RSPACK = true;
18       else
19         VCI[0].RSPACK = false;
20       break;
21   } // fin switch SOURCE_FSM_STATE
22   switch(DEST_FSM_STATE) { // interface VCI[1]
23     case IDLE:
24       VCI[1].CMDVAL = false;
25       VCI[1].RSPACK = false;
26       break;
27     case REQUEST:
28       VCI[1].CMDVAL = true;
29       VCI[1].RSPACK = false;
30       VCI[1].ADDRESS = BUF_ADDRESS;
31       VCI[1].WDATA = BUF_DATA;
32       VCI[1].CMD = VCI_CMD_WRITE;
33       break;
34     case RESPONSE:
35       VCI[1].CMDVAL = false;
36       VCI[1].RSPACK = true;
37       break;
38   } // fin switch DEST_FSM_STATE
39 }
```

Listings 2.7 – DMA simplifié décrit en SystemC CABA : fonction de génération

2.4 Conclusions

Au cours de ce chapitre, nous avons défini le type d'architecture matérielle que nous cherchons à simuler : une architecture multiprocesseur composée de module CABA. Nous avons ensuite dégagé deux objectifs. L'un est d'écrire des modèles rapides en simulation ; l'autre est de réduire le temps nécessaire à la construction d'une architecture matérielle.

Deux approches de modélisation ont été présentées dans l'état de l'art. CASS [PHG97] utilise le modèle des automates synchrones communicants pour accélérer la vitesse en simulation. *FastSysC* [PMT04] enrichit les modèles afin de réduire le nombre d'exécution des processus.

La méthodologie présentée s'inspire de ces deux travaux. Elle repose sur la modélisation de composants réutilisables exploitant le modèle formel des automates à états finis communicants et sur la description explicite des relations entre les ports d'entrées et les ports de sortie au sein d'un même module.

Nous avons développé des composants matériels respectant la méthodologie présentée dans ce chapitre. La bibliothèque **SoCLIB** [SoC03] découle directement du travail sur la méthode de modélisation. Cette bibliothèque est constituée de composants validés, rapides à la simulation, et inter-opérables. Le concepteur a ainsi la possibilité de construire rapidement des architectures matérielles pour l'exploration architecturale.

Dans les chapitres suivants, les travaux présentés exploitent la bibliothèque SoCLIB. Nous avons ainsi pu obtenir de nombreux résultats expérimentaux.

Chapitre 3

Ordonnancement totalement statique pour la simulation

Sommaire

3.1 État de l'art . . . 40
3.1.1 FastSysC : ordonnancement mixte . . . 41
3.1.2 CASS : ordonnancement quasi statique . . . 42
3.1.3 Problèmes non résolus . . . 44
3.2 Hypothèses de modélisation des modules . . . 45
3.2.1 Modélisation CABA en automates synchrones communicants . . . 45
3.2.2 Identification du signal d'horloge . . . 45
3.2.3 Dépendances entre les ports d'entrée et les ports de sortie . . . 45
3.3 Construction de l'ordre d'exécution des processus . . . 46
3.3.1 Construction du graphe de dépendances entre les ports . . . 47
3.3.2 Transformation en graphe de dépendances entre signaux . . . 47
3.3.3 Construction de l'ordre d'exécution des processus . . . 48
3.3.4 Propriétés de l'ordre d'exécution des processus . . . 51
3.4 Résultats expérimentaux . . . 53
3.4.1 Architectures modélisées pour les simulations . . . 54
3.4.2 Bilan des résultats . . . 63
3.5 Conclusions . . . 63

L'exploration architecturale requiert de nombreuses simulations dont la durée est très importante. La vitesse de la simulation architecturale dépend principalement du moteur du simulateur. Dans ce chapitre, nous nous intéressons donc à l'impact du moteur sur la vitesse d'exécution.

Les architectures que nous souhaitons simuler contiennent uniquement des modules décrits au niveau CABA sous la forme d'automates synchrones communicants. Chacun de ces modules possède un ou plusieurs processus décrivant son comportement.

Le principal rôle du moteur de simulation est de réaliser l'ordonnancement des processus. Ce rôle est assuré par l'échéancier qui calcule un ordre d'exécution séquentiel des processus sur la station de travail. Le coût de l'échéancier et le coût d'exécution des processus ont un impact important sur la vitesse de simulation. Nous cherchons donc à réduire ces coûts à l'aide d'un échéancier à ordonnancement statique.

L'état de l'art dans le domaine de l'ordonnancement est présenté dans la section suivante. Puis, nous détaillons une méthode permettant de construire un ordonnancement statique optimal des processus. Nous illustrons ensuite la méthode par quelques exemples de taille réelle. Les résultats expérimentaux sont présentés à la fin de ce chapitre.

3.1 État de l'art

Les modules que nous cherchons à simuler sont au niveau CABA et respectent la modélisation en CFSM. Le comportement de chaque composant matériel est décrit, dans le simulateur, sous la forme d'un ou plusieurs processus. La simulation consiste à exécuter de manière séquentielle les processus sur la station de travail. L'ordre d'exécution est calculé par l'échéancier à l'aide d'un algorithme d'ordonnancement.

Deux catégories d'ordonnancement existent :

- L'ordonnancement dynamique : l'ordre d'exécution des processus est déterminé, au fur et à mesure, en fonction des évènements qui se produisent pendant la simulation ;
- L'ordonnancement statique : l'ordre d'exécution des processus est déterminé définitivement avant la simulation.

L'ordonnancement dynamique implique le calcul de l'ordre d'exécution des processus au cours de la simulation. Le surcoût de l'ordonnancement est alors proportionnel à la durée de la simulation. L'exécution d'un échéancier à ordonnancement dynamique devient donc coûteux. De plus, les échéanciers à ordonnancement dynamique proposent généralement un ordre d'exécution sous-optimal.

L'ordonnancement statique consiste à calculer l'ordre d'exécution des processus avant la simulation. Le coût de l'ordonnancement est payé une seule fois, et ce, quelle que soit la durée totale de la simulation. Autrement dit, le coût est constant par rapport à la durée de la simulation et devient donc négligeable.

Dans l'état de l'art, nous présentons deux méthodes d'ordonnancement. La première propose une solution mixte statique/dynamique. La deuxième décrit un échéancier à ordonnancement quasi statique.

3.1.1 FastSysC : ordonnancement mixte

Le simulateur *FastSysC* [PMT04] repose sur le modèle de calcul à évènements discrets. Les composants matériels ainsi que le schéma d'interconnexion sont écrits en SystemC.

L'objectif de ce simulateur est de réduire le coût de l'ordonnancement dynamique. Pour cela, la méthode proposée est de réduire le nombre de réveil de processus, autrement dit le nombre d'exécution de chaque fonction.

L'échéancier de *FastSysC* minimise le nombre d'exécutions de processus à l'aide d'un échéancier travaillant en deux temps. Dans un premier temps, l'échéancier utilise un ordonnancement statique incomplet, dit acyclique. L'ordonnancement étant incomplet, les signaux ne sont pas encore tous évalués. Dans un second temps, l'échéancier continue l'exécution des processus jusqu'à stabilité du système à l'aide d'un ordonnancement dynamique classique. L'objectif du premier ordonnancement statique est de réduire le coût de l'ordonnancement dynamique.

Pour atteindre cet objectif, *FastSysC* impose une méthode de modélisation du matériel pour obtenir des informations sur les relations de dépendances entre les signaux d'entrée et les signaux de sortie. L'échéancier exploite ces informations avant l'exécution de la simulation pour construire l'ordonnancement statique incomplet.

FastSysC propose au concepteur de déclarer explicitement les dépendances entre les signaux d'entrée et les signaux de sortie. Ces relations de dépendances permettent de construire un graphe de signaux. Les nœuds sont les signaux, et les arcs orientés indiquent une relation de dépendance. A chaque arc est associé le processus pilote.

Si le graphe contient un ou plusieurs cycles, *FastSysC* retire un des arcs arbitrairement jusqu'à l'obtention d'un graphe acyclique. L'échéancier procède ensuite à un tri topologique. L'ordonnancement des processus est obtenu en réveillant les processus permettant d'évaluer les signaux dans l'ordre.

La figure 3.1 présente une architecture matérielle comportant quatre modules. Chaque module possède un unique processus. Les dépendances de données sont indiquées par des flèches en pointillé. L'échéancier construit le graphe de dépendances entre signaux illustré par la figure 3.2. L'ordonnancement des signaux est alors :

1. $S1$ et $S2$;
2. $S3$;
3. $S4$;
4. $S5$.

L'algorithme d'ordonnancement [Gil04] remplace les signaux par les processus pilotant ces signaux. L'ordonnancement des processus est donc :

1. A et D ;
2. B ;
3. C ;
4. B.

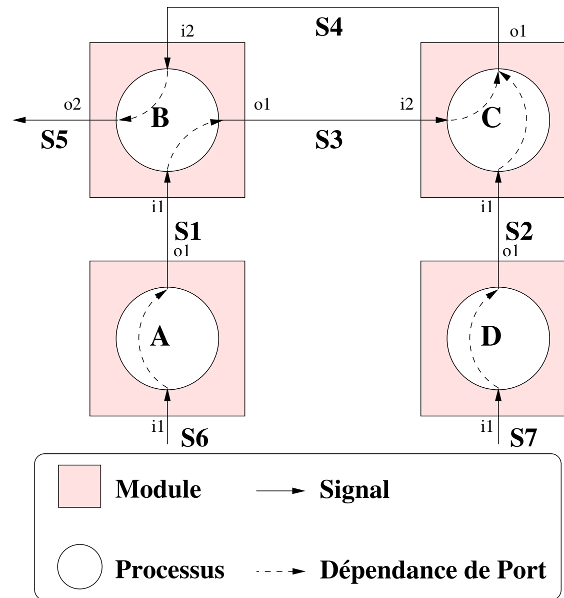

FIG. 3.1 – Exemple : Architecture

L'ordonnancement obtenu est optimal. Cependant, le processus *B* doit être exécuté deux fois. Les signaux *S*3 et *S*5 sont alors évalués chacun deux fois. Ces évaluations redondantes sont dues à une mauvaise description du comportement du module *B* et pourraient être évitées.

Bilan

FastSysC propose un échéancier mixte statique/dynamique et obtient un facteur d'accélération de la simulation entre deux et trois par rapport à un simple échéancier dynamique. Cependant, ce moteur n'exploite pas suffisamment les informations statiques tout en payant le prix fort d'une partie dynamique. En effet, l'ordonnancement statique est incomplet. L'échéancier dynamique est toujours présent et implique par conséquent un coût important en temps d'exécution.

3.1.2 CASS : ordonnancement quasi statique

Le simulateur CASS [PHG97][RFA02], repose sur le modèle de calcul des automates synchrones communicants présenté dans le chapitre 2.

Les composants sont modélisés en langage *C*. Le schéma d'interconnexion est décrit en VHDL. Le fichier VHDL permet de fournir également des méta-données au moteur de simulation.

Le concepteur doit fournir explicitement la liste des sorties de Mealy pour chaque composant matériel. Ces informations permettent de construire, avant la simulation, un graphe de processus dans lequel les nœuds sont les processus, et les arcs sont les signaux pilotés.

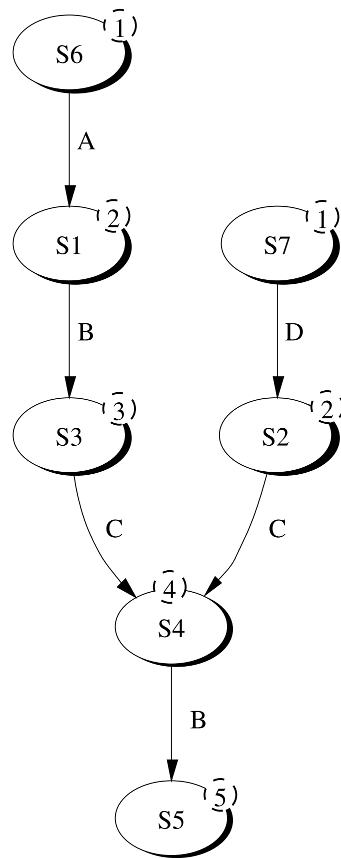

FIG. 3.2 – Exemple : Graphe de dépendances de signaux triés

L'échéancier procède à un tri topologique des processus pour établir un ordre d'exécution. Pour chaque cycle du graphe, l'échéancier génère une boucle de relaxation. La boucle de relaxation exécute chaque processus présent dans le cycle jusqu'à stabilité du système.

Reprenons l'architecture de la figure 3.1. Du point de vue de CASS, les dépendances entre signaux d'entrée et signaux de sortie ne sont pas connues. Cependant, les signaux $S1$, $S2$, $S3$, $S4$, $S5$ sont tout de même identifiés comme signaux de sortie de Mealy. L'échéancier construit ensuite le graphe de dépendances entre processus illustré en figure 3.3. Un nœud est créé pour chaque processus combinatoire (A, B, C, D). Le signal $S1$ est une entrée de B et une sortie de Mealy de A donc un arc est ajouté pour relier le nœud A vers le nœud B. Pour chaque sortie de Mealy, des arcs sont ainsi ajoutés si nécessaires. Le graphe de processus permet de déduire l'ordonnancement suivant :

1. D'abord D et A dans n'importe quel ordre ;
2. Puis B et C jusqu'à stabilité du système.

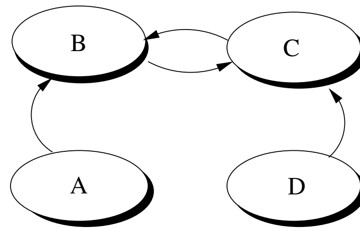

FIG. 3.3 – Exemple : Graphe de dépendances entre processus

Une exécution résultant de l'ordonnancement de CASS pourrait être : *A*, *D*, *B*, *C*, *B*, *C*, *B*, *C*. Cet ordonnancement est sous-optimal puisque les processus *B* et *C* sont exécutés plus de fois que nécessaire. Le coût de ces exécutions superflues sont cependant plus faibles que le coût d'un ordonnancement dynamique.

Bilan

CASS propose un échéancier statique et obtient une accélération qui peut atteindre un facteur dix par rapport à un échéancier dynamique. Malheureusement, les informations de dépendances sous la forme d'une liste de sortie de Mealy sont insuffisantes pour déterminer un ordre d'exécution minimal. Le graphe de processus construit à partir de ces informations peut contenir des cycles alors que le graphe des signaux n'en contient pas. Les figures 3.3 et 3.2 illustrent la différence entre le graphe de processus et le graphe de signaux. Plus le nombre de cycles dans le graphe augmente, plus l'ordonnancement provoque des exécutions superflues. L'accélération attendue décroît alors de manière significative.

3.1.3 Problèmes non résolus

FastSysC exploite les informations sur les dépendances entre les signaux d'entrée et les signaux de sortie pour construire un ordonnancement mixte rapide. Si les informations sur les dépendances sont erronées ou incomplètes, l'échéancier à ordonnancement dynamique garantit que l'évaluation des signaux est correcte.

Le simulateur CASS construit un ordonnancement quasi statique très rapide lorsqu'il existe peu de dépendances combinatoires entre les modules.

Certains problèmes ne sont pas résolus par les simulateurs CASS et *FastSysC*. Cela nous amène à se poser les questions suivantes :

- Comment construire un ordonnancement totalement statique ?
- Comment garantir, avant simulation, que tous les signaux soient évalués une seule fois par cycle ?
- Comment garantir, par construction, que l'ordonnancement est bien optimal ?

La contribution qui fait l'objet de ce chapitre s'inspire des travaux effectués dans le cadre des simulateurs CASS et *FastSysC* et propose un simulateur à ordonnancement totalement statique

sans boucle de relaxation.

3.2 Hypothèses de modélisation des modules

La contribution présentée dans ce chapitre propose une technique d'ordonnancement statique permettant de simuler des architectures multiprocesseurs pouvant être décrites jusqu'au niveau micro-architecture. L'ordonnancement proposé est inspiré par les méthodes employées dans CASS et *FastSysC*. Cet ordonnancement raisonne sur le graphe de dépendances entre signaux et est complètement statique, sans boucle de relaxation.

Les hypothèses de modélisation sur lesquelles repose notre ordonnancement statique portent sur trois points :

- Le niveau de modélisation CABA/CFSM ;
- L'identification du signal d'horloge ;
- Les dépendances combinatoires entre les signaux.

Si ces trois hypothèses sont respectées, la méthode d'ordonnancement présentée dans ce chapitre est applicable.

3.2.1 Modélisation CABA en automates synchrones communicants

La modélisation CABA en automates synchrones communicants est décrite en détail dans le chapitre 2 (page 26). Les modèles en automate sont décrits sous forme de processus. Les dépendances de données entre ces processus sont identifiées explicitement. L'ordre d'exécution des processus est alors en partie déterminé.

Les modèles, que nous souhaitons ordonnancer, doivent respecter cette modélisation pour obtenir les garanties suivantes :

- L'exécution unique de chaque fonction est nécessaire et suffisante ;
- Les fonctions de l'architecture doivent être exécutées dans l'ordre suivant :
  1. Toutes les fonctions de transition,
  2. Toutes les fonctions de générations de Moore,
  3. Toutes les fonctions de générations de Mealy ;

3.2.2 Identification du signal d'horloge

Le signal d'horloge doit être explicitement désigné par le concepteur. Ce signal permet de synchroniser la mémorisation dans les registres. Cette information permet, à l'échéancier, de déterminer le type de chaque processus : transition, ou génération.

3.2.3 Dépendances entre les ports d'entrée et les ports de sortie

Le concepteur de module doit déclarer explicitement les dépendances combinatoires entre les ports d'entrée et les ports de sortie au sein de chaque processus. Par construction, il n'existe

pas de dépendances entre les ports d'entrée et les ports de sortie dans les fonctions de transition et de génération de Moore. Il en existe au moins une dans chaque fonction de génération de Mealy. (Cf. figure 2.5 page 23).

Les dépendances entre les ports sont fournies à l'échéancier par le concepteur. L'échéancier construit alors les dépendances combinatoires entre les signaux.

3.3 Méthode de construction de l'ordre d'exécution des processus

La construction de l'ordre d'exécution des processus se déroule après l'élaboration du schéma d'interconnexion des composants mais avant la simulation. La construction se compose de trois étapes illustrées par la figure 3.4. Nous construisons d'abord un graphe de dépendances entre les ports que nous transformons ensuite en graphe de dépendances entre les signaux, pour finalement établir l'ordre d'exécution des processus.

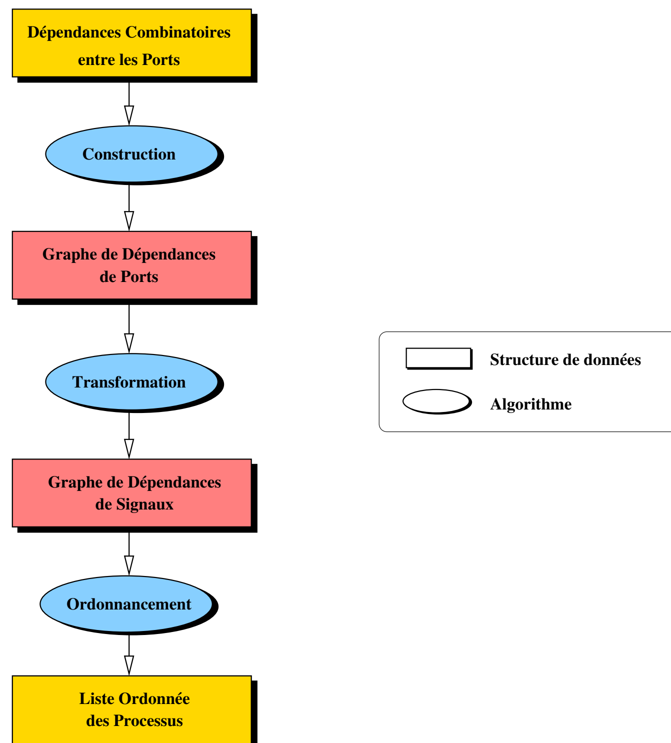

FIG. 3.4 – Méthode pour l'ordonnancement totalement statique

Les sections suivantes présentent ces trois étapes en s'appuyant sur l'exemple d'une architecture simpliste.

3.3.1 Construction du graphe de dépendances entre les ports

Le graphe de dépendances entre les ports d'entrée et les ports de sortie d'un même composant exprime les dépendances combinatoires locales dans les fonctions de Mealy. Les noeuds du graphe sont des ports d'entrée ou des ports de sortie. Les arcs sont orientés et expriment une relation de dépendance. A chaque arc est associé le processus pilote.

Un port est soit une entrée, soit une sortie mais ne peut pas être les deux. Le graphe est donc orienté et acyclique.

La figure 3.5 représente le graphe construit à partir des informations sur les dépendances entre les ports, illustrées par les flèches en pointillées de la figure 3.1. Par exemple, le port de sortie $o1$ du module B est piloté par le processus B et dépend du port d'entrée $i1$.

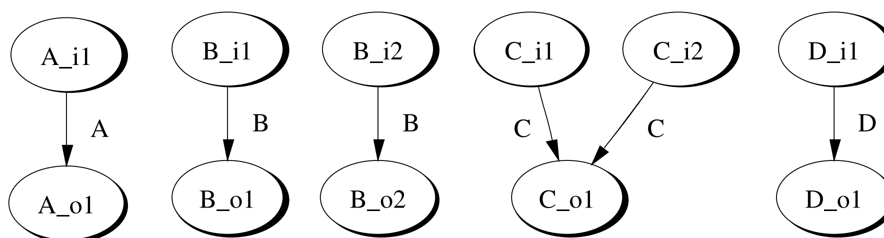

FIG. 3.5 – Graphe de dépendances entre les ports

3.3.2 Transformation en graphe de dépendances entre signaux

L'ordonnancement proposé dans ce chapitre raisonne sur le graphe de dépendances entre signaux. Ce graphe est obtenu à partir du graphe de dépendances entre ports.

Le schéma d'interconnexion de l'architecture à simuler associe un signal à chaque port. Pour construire le graphe de dépendances entre les signaux, il suffit de regrouper tous les ports connectés à un même signal en un seul nœud de type signal.

Les noeuds du graphe de signaux sont tous les signaux du système. Un arc est orienté et exprime une dépendance combinatoire entre deux signaux. Une étiquette est associée à chaque arc pour identifier le processus pilote. Le graphe est orienté.

La fonction 1 définit une implémentation possible pour construire le graphe de dépendances entre signaux. La figure 3.6 représente le graphe de dépendances entre signaux construit à partir du graphe de dépendances entre ports de la figure 3.5.

Les graphes contenant un ou plusieurs cycles sont considérés comme erronés. S'il existe un cycle dans le graphe, le concepteur de modèle matériel doit corriger la description de l'architecture pour éliminer tout cycle de dépendances combinatoires entre signaux.

Le graphe de dépendances entre signaux, contrairement au graphe de dépendances entre processus (voir figure 3.3), ne comporte pas de cycle. Puisque ce graphe ne comporte pas de cycles, il est possible de déterminer un ordonnancement statique pour l'évaluation des processus.

Fonction 1 Transformation d'un graphe de dépendances entre ports G_{ports} en graphe de dépendances entre signaux $G_{signaux}$

Require: $Graphe_{ports}$ est acyclique
  $G_{signaux} \Leftarrow \emptyset$
  for all A_{ij} arc de G_{ports} orienté de $Port_i$ vers $Port_j$ **do**
    Créer un nœud signal N_1 où N_1 est connecté à $Port_i$
    Créer un nœud signal N_2 où N_2 est connecté à $Port_j$
    Ajouter dans $G_{signaux}$ un arc orienté A de N_1 vers N_2
  end for
  return $G_{signaux}$

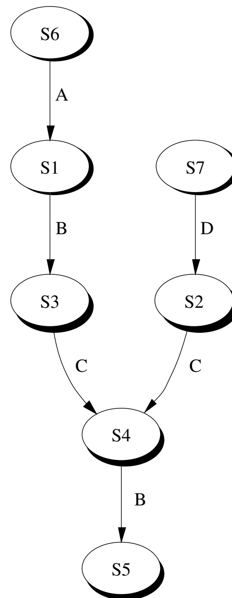

FIG. 3.6 – Exemple : Graphe de dépendances entre signaux

Cependant, un même processus devra être évalué plusieurs fois. Dans l'exemple de la figure 3.6, le processus B doit être évalué deux fois.

3.3.3 Construction de l'ordre d'exécution des processus

Nous voulons construire une liste ordonnée des processus à exécuter, à partir du graphe acyclique de dépendances combinatoires entre signaux. La construction de l'ordre d'exécution des processus n'est pas triviale, car un même processus peut être attaché à plusieurs arcs du graphe de dépendances entre signaux. Cette construction se déroule de la manière suivante :

1. S'il existe un processus pour lequel tous les signaux pilotés dépendent exclusivement de

signaux source[1], nous le sélectionnons et passons à l'étape 3.

2. Nous indiquons, au concepteur, que l'ordonnancement ne peut être optimal car certains signaux seront évalués plusieurs fois au cours du même cycle de simulation. Nous proposons une liste de processus à modifier pour une éventuelle optimisation. Nous sélectionnons ensuite un processus qui pilote au moins un nœud qui dépend exclusivement de signaux source[2] ;
3. Le processus sélectionné est ajouté dans la liste ordonnée des processus à exécuter ;
4. Nous supprimons les arcs sortant des sources, qui sont étiquetés par le processus sélectionné, puis nous supprimons tous les sommets de degré 0 (toutes les sources n'ayant plus d'arcs sortants) ;
5. La construction de l'ordre d'exécution répète les opérations précédentes (étapes 1 à 4) et s'arrête lorsque le graphe de dépendances entre signaux est vide.

La figure 3.1 présente un exemple d'architecture. Cette architecture est décrite par quatre processus :

- A évalue le signal $S1$ en fonction du signal $S6$;
- B évalue les signaux $S3$ et $S5$ respectivement en fonction des signaux $S1$ et $S4$;
- C évalue le signal $S4$ en fonction des signaux $S2$ et $S3$;
- D évalue les signaux $S2$ en fonction du signal $S7$.

Le graphe initial de dépendances entre signaux est illustré par la figure 3.6. L'ordonnancement nécessite cinq itérations :

1. À la première itération, nous pouvons sélectionner le processus A car A dépend exclusivement d'un seul signal source du graphe. Le processus A est donc ajouté dans la liste $L = A$. L'arc étiqueté A est supprimé. Le nœud $S6$ n'a ni arcs entrants ni arcs sortants. $S6$ est donc supprimé. Le nœud $S1$ devient alors une source. Le graphe obtenu est illustré par la figure 3.7 ;
2. À la seconde itération, le processus D dépend d'un seul signal source. Le nœud $S7$ est supprimé. D est ajouté dans la liste $L = A, D$. Le graphe obtenu est illustré par la figure 3.8.
3. À la troisième itération, aucun processus ne satisfait le critère suivant : le processus sélectionné dépend exclusivement de signaux sources. L'ordonnancement ne peut être optimal. Nous sélectionnons alors B car ce processus pilote le signal $S3$ qui dépend exclusivement de nœud source. Le processus B est ajouté dans la liste $L = A, D, B$. L'arc entre $S1$ et $S3$, ainsi que le nœud $S1$ sont supprimés. Le nœud $S3$ devient, à son tour, une source. La figure 3.9 présente le graphe résultant ;
4. À la quatrième itération, le processus C ne dépend que de signaux source. C est donc ajouté dans la liste $L = A, D, B, C$. Les deux arcs étiquetés C sont supprimés, ainsi que les nœuds $S2$ et $S3$. $S4$ est maintenant une source, comme indiqué sur la figure 3.10.

[1] Le graphe étant acyclique, il existe toujours au moins un nœud source.

[2] Il existe toujours un processus satisfaisant ce critère, car, dans notre graphe orienté acyclique, il existe toujours au moins un nœud d'ordre 2.

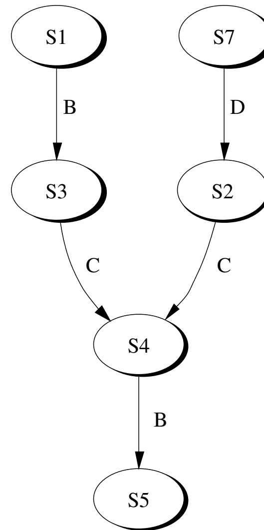

FIG. 3.7 – Exemple : Graphe de dépendances combinatoires entre signaux après ordonnancement de A

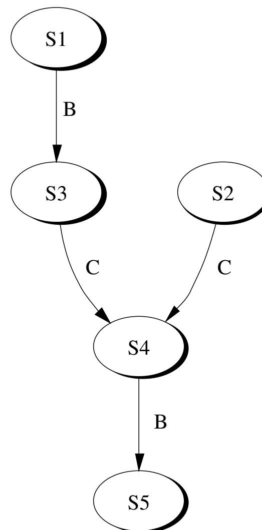

FIG. 3.8 – Exemple : Graphe de dépendances combinatoires entre signaux après ordonnancement de D

5. À la cinquième et dernière itération, le processus B est de nouveau sélectionné. Après suppression des arcs et des nœuds, le graphe est alors vide. La construction de l'ordonnancement est terminée. L'ordre d'exécution est $L = A, D, B, C, B$.

Nous proposons un algorithme d'ordonnancement composé de deux fonctions. La fonction 2 sélectionne un processus et retourne la sélection. La fonction 3 tient à jour la liste ordonnée des

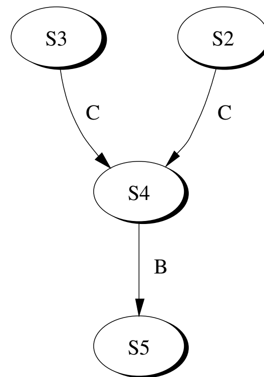

FIG. 3.9 – Exemple : Graphe de dépendances combinatoires entre signaux après ordonnancement de *B*

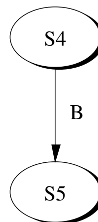

FIG. 3.10 – Exemple : Graphe de dépendances combinatoires entre signaux après ordonnancement de *C*

processus et réduit le graphe de dépendances entre signaux.

La figure 3.1 illustre une architecture matérielle dont la description introduit la nécessité d'une évaluation multiple d'un processus. Pour le constater, il suffit de consulter la figure 3.6 représentant le graphe de dépendances entre signaux. Le processus *B* pilote deux signaux : *S*3 et *S*5. Les deux signaux ne peuvent être évalués correctement en une seule exécution de *B*. Le processus *B* doit donc être exécuté deux fois par cycle pour évaluer *S*3 dans un premier temps, puis *S*5 dans un second temps.

Notre méthode d'ordonnancement détecte ce problème et avertit le concepteur. Pour résoudre le problème, le concepteur doit modifier le module *B*. Il suffit de diviser le processus *B* en deux autres processus. La figure 3.11 montre l'architecture modifiée. Un processus *B*1 pilote le signal *S*3, et un autre processus *B*2 pilote le signal *S*5. Les processus *B*1 et *B*2 seront alors ordonnancés séparément.

3.3.4 Propriétés de l'ordre d'exécution des processus

Un modèle est optimisé lorsqu'un seul réveil de processus par cycle est suffisant pour évaluer chaque signal de sortie. Dans le cas où tous les modèles sont optimisés, l'ordonnancement généré a les propriétés suivantes :

- L'ordre exact d'exécution des processus est connu avant simulation ;

Fonction 2 Sélection du processus à ordonnancer

Require: $Liste_{puits}$ contient les puits du $Graphe_{signaux}$
Ensure: Le processus sélectionné pilote au moins un signal puits du graphe $Graphe_{signaux}$

```
for all S tel que S ∈ Liste_puits do
   P =getDriver(S)
   if drivesOnlySinks(P) then
      return P
   end if
end for
print Unable to select a process avoiding multiple signal evaluation.
S = getSinkRandomly(Liste_puits)
return getDriver(S)
```

Fonction 3 Construction de l'ordre d'exécution des processus

Require: $Graphe_{signaux}$ est acyclique
Ensure: Tous les processus apparaissent exactement une fois dans $Liste_{processus}$

```
Liste_processus ⇐ ∅
repeat
   Liste_signaux ⇐ get_sinks(Graphe_signaux)
   P ⇐ select_driver_process(Liste_signaux)
   Insert P into Liste_processus
   for all S tel que S ∈ Liste_signaux do
      if P =get_driver(S) then
         Remove S from Graphe_signaux
      end if
   end for
until Graphe_signaux ⊆ ∅
return Liste_processus
```

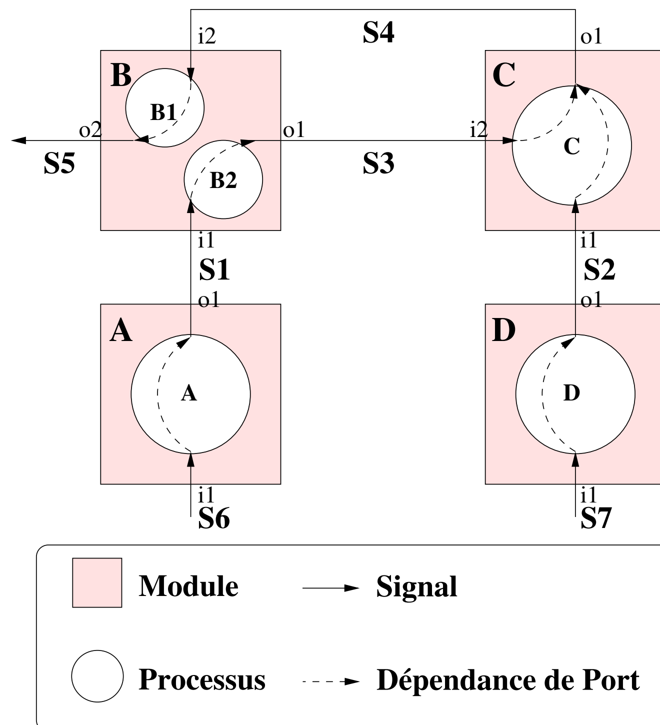

FIG. 3.11 – Exemple : Architecture modifiée pour satisfaire le critère de l'algorithme d'ordonnancement totalement statique

- Toutes les fonctions (transitions, générations de Moore et de Mealy) sont exécutées une et une seule fois par cycle ;
- Chaque signal est évalué une seule fois par cycle.

L'échéancier proposé est totalement statique et évite complètement les évaluations redondantes. La simulation a alors un coût minimal d'exécution.

3.4 Résultats expérimentaux

Le moteur de simulation développé dans le cadre de cette thèse, s'appelle *SystemC Accurate System Simulator* (SystemCASS). Celui-ci est décliné en deux versions. La première version raisonne sur le graphe de modules et possède un échéancier quasi statique qui génère un ordonnancement des processus combinatoires identique à l'ordonnancement généré par CASS. La seconde version raisonne sur le graphe de signaux et possède un échéancier totalement statique qui implémente la méthode d'ordonnancement présentée dans le paragraphe 3.3 de ce chapitre. Le simulateur SystemCASS a été comparé avec le simulateur SystemC à ordonnancement dynamique d'*Open SystemC Initiative* (OSCI). Le simulateur FastSysC, présenté dans l'état de l'art,

implémente un sous-ensemble trop restreint du langage SystemC pour compiler nos modèles de simulation. Nous n'avons donc pas effectué de simulations avec FastSysC. La table 3.1 résume les différences entre les trois simulateurs comparés dans nos résultats expérimentaux.

Simulateur	Raisonne sur...	Type d'ordonnancement
SystemC 2.1.v1	les évènements	Dynamique
SystemCASS quasi statique	le graphe de processus	Quasi Statique
SystemCASS totalement statique	le graphe de signaux	Totalement Statique

TAB. 3.1 – Simulateurs comparés lors des expérimentations

Nous avons simulé trois architectures pour comparer les temps de simulation. Les modèles utilisés par SystemCASS et par SystemC 2.1.v1 sont strictement identiques. Les déclarations des dépendances entre ports ne font pas parties de la norme SystemC. Ces déclarations sont utilisées par le simulateur SystemCASS mais pas par le simulateur d'OSCI. Les déclarations de dépendances entre ports sont donc activées ou désactivées à l'aide de macros.

Les comparaisons sont effectuées sur les temps de simulation. Ces temps de simulation sont mesurés après la phase d'initialisation. La phase d'initialisation est plus longue dans le cadre d'un simulateur à ordonnancement statique mais la durée de l'initialisation reste négligeable devant les temps de simulation.

Les sections suivantes décrivent les architectures modélisées ainsi que leurs applications, puis les résultats expérimentaux obtenus. Nous commentons ensuite les résultats.

3.4.1 Architectures modélisées pour les simulations

Nous avons simulé trois architectures :

- Architecture multiprocesseur simple :
  une application de routage de paquets Ethernet déployée sur une architecture multiprocesseur ;
- Architecture multiprocesseur clusterisée :
  une application de stéréo-vision est déployée sur une architecture à clusters multiprocesseurs ;
- Micro-architecture de processeur complexe :
  une application de raytracer est déployée sur un processeur Java décrit au niveau micro-architecture.

Chaque architecture est détaillée dans les sections suivantes. Les résultats expérimentaux sont présentés et commentés.

Architecture multiprocesseur simple

Ce travail a été effectué dans le cadre de la thèse d'Etienne Faure au *Laboratoire d'Informatique de Paris 6* (LIP6). L'architecture est constituée de composants respectant le protocole

de communication VCI provenant de la bibliothèque SoCLIB. L'architecture est multiprocesseur et organisée autour d'un micro réseau d'interconnexions. Le nombre de processeurs dans l'architecture est paramétrable. Les composants *input_engine* et *output_engine* assurent la communication avec l'extérieur. La figure 3.12 illustre l'architecture sur laquelle est déployée une application gigabit Ethernet. Le déploiement de l'application a fait l'objet de deux articles dans des conférences internationales : [FGPB04], et [FBGP04].

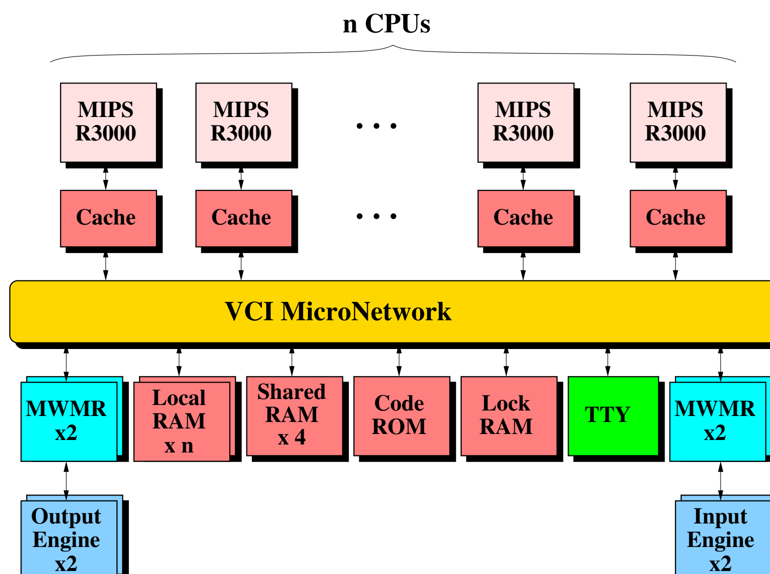

FIG. 3.12 – PAPR : Architecture matérielle

Les dépendances combinatoires de l'architecture se situent au niveau de l'interface entre les processeurs et leurs mémoires caches. Le plus long chemin dans le graphe de dépendances entre signaux est de longueur 1. Tous les processus combinatoires pilotent uniquement des signaux puits du graphe. L'ordonnancement des processus est donc simple dans ce cas : les processus combinatoires peuvent être exécutés dans n'importe quel ordre.

Le nombre de processeurs n connectés au micro réseau est un paramètre structurel de l'architecture. La figure 3.13 montre les vitesse de simulation en fonction du nombre de processeurs, obtenus par SystemC OSCI et par les deux versions de SystemCASS. L'ordre d'exécution des processus combinatoires n'ayant pas d'importance, l'ordonnancement totalement statique obtient des performances très similaires à l'ordonnancement quasi statique.

Le rapport entre la vitesse de simulation de SystemCASS et celle de SystemC d'OSCI augmente lorsque la taille de l'architecture augmente. La figure 3.14 illustre la croissance du rapport en fonction du nombre de processeurs.

Pour cette architecture, les performances d'un ordonnancement quasi statique sont très proches d'un ordonnancement totalement statique. Finalement, la simulation de cette architecture permet d'observer le gain d'un moteur à ordonnancement statique par rapport à un moteur

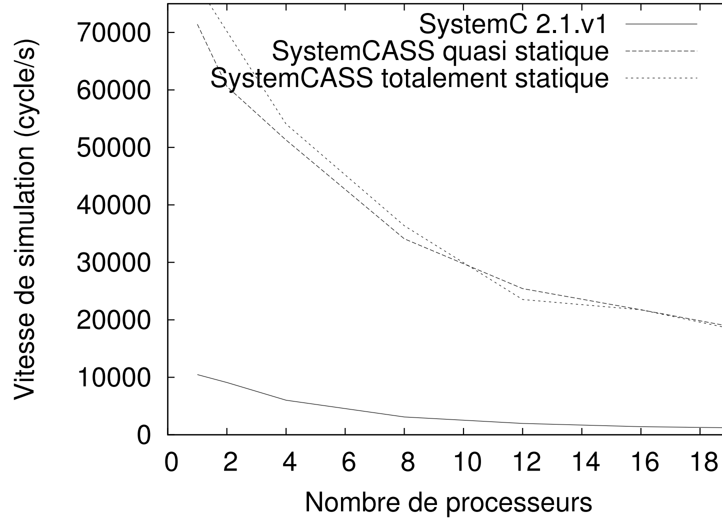

FIG. 3.13 – PAPR : Vitesse de simulation en fonction du nombre de processeurs

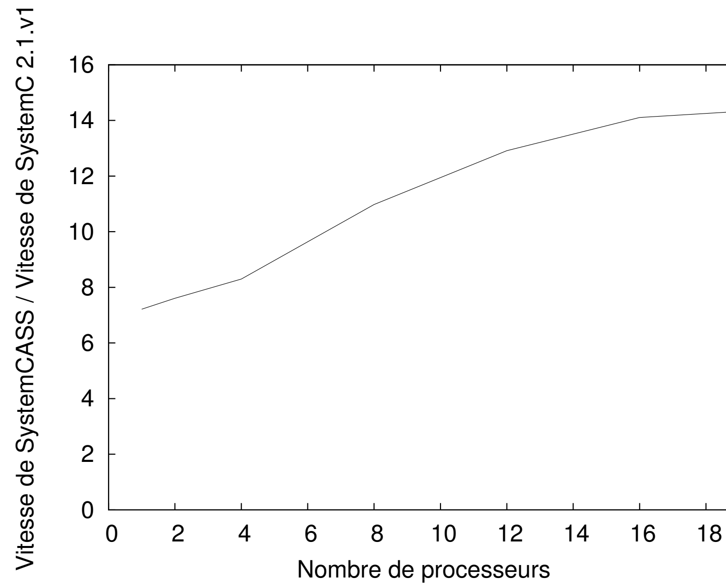

FIG. 3.14 – PAPR : Vitesse de SystemCASS par rapport à SystemC 2.1.v1 en fonction du nombre de processeurs

à ordonnancement dynamique.

Architecture multiprocesseur clusterisée

Ce travail a été effectué dans le cadre de la thèse de Mathieu Carrier au LIP6. L'application Stéréo-vision[GPC+06] a pour objectif de détecter les obstacles devant un véhicule routier avant collision. Cette application nécessite de lourds calculs et possède des fortes contraintes de temps. Pour tenir ces contraintes, l'application multithread est déployée sur une architecture à clusters multiprocesseurs. Les clusters communiquent entre eux à travers un micro réseau global. La figure 3.15 présente l'architecture à huit clusters interconnectés par le micro réseau global.

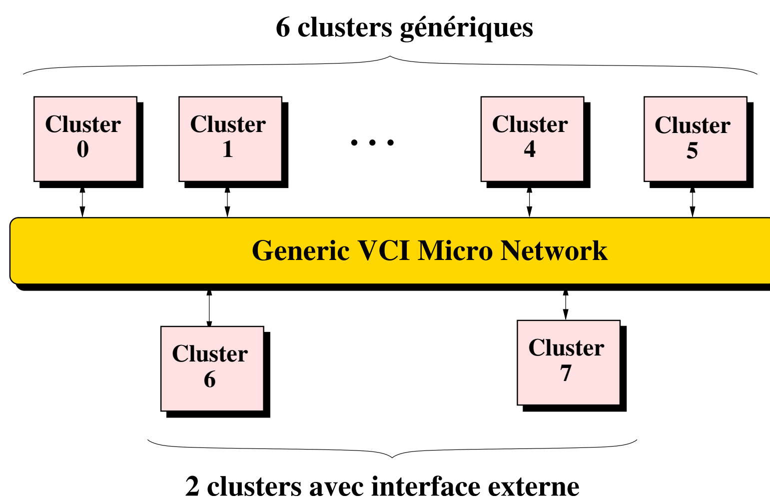

FIG. 3.15 – Stéréo-vision : Architecture matérielle générale

Tous les composants au sein du même cluster sont interconnectés à l'aide d'un crossbar local. L'architecture possède donc un réseau d'interconnexion à deux niveaux.

Chaque cluster contient quatre processeurs programmables avec leurs caches, des bancs mémoires, un timer, une console d'affichage *TeleTYpewriter* (TTY), et un micro réseau local. Certains clusters possèdent un coprocesseur pour gérer les accès à l'interface externe.

Les micro réseaux locaux sont des crossbars complètement combinatoires. Cependant, les composants connectés aux micro réseaux ont un comportement synchrone. Les dépendances combinatoires présentes dans cette architecture sont situées entre les signaux en entrée d'un micro réseau et ses signaux de sortie. Le plus long chemin combinatoire dans le graphe de dépendances entre signaux est donc toujours de longueur 1. Les processus combinatoires peuvent alors être exécutés dans n'importe quel ordre. La figure 3.17 montre la vitesse de simulation de SystemCASS et de SystemC 2.1.v1 d'OSCI. SystemCASS est au moins 10 fois plus rapide. Les versions quasi statique et totalement statique de SystemCASS obtiennent des performances similaires.

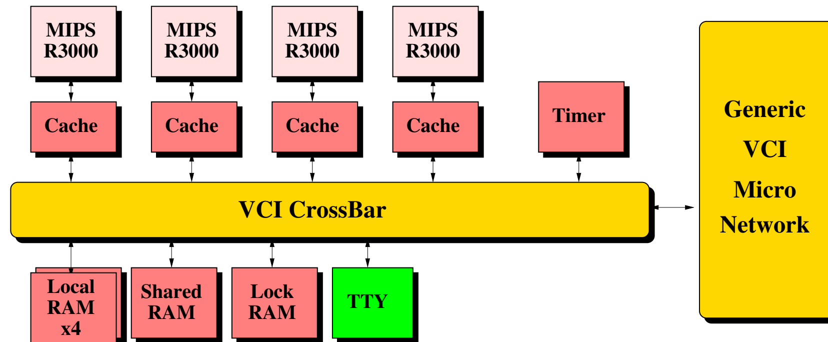

FIG. 3.16 – Stéréo-vision : Architecture matérielle d'un cluster

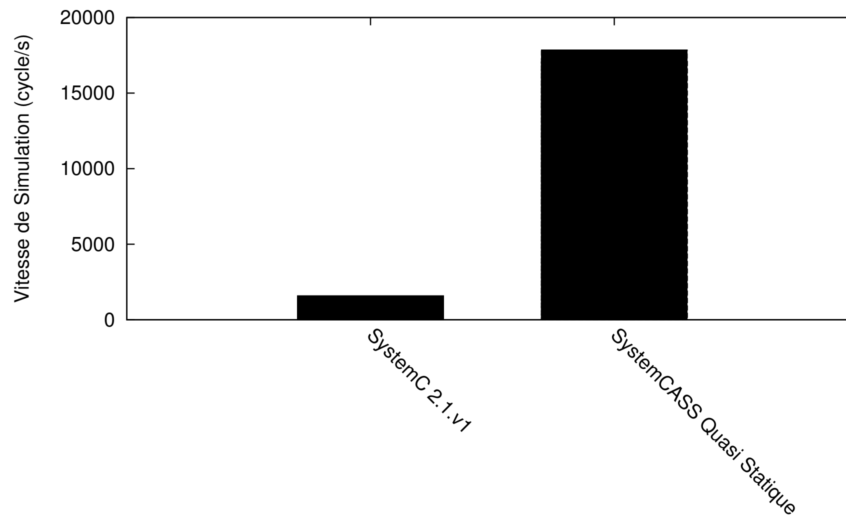

FIG. 3.17 – Stéréo-vision : Résultats de simulation

Micro-architecture de processeur complexe

Ce travail a été effectué dans le cadre de la thèse de Maxime Palus au LIP6. L'application de raytracer est un moteur de calcul d'image 3D. Cette application est exécutée sur une architecture composée d'un processeur Java, d'une mémoire cache, et d'un banc de mémoire vive. Le processeur Java est décrit au niveau micro-architecture, sous la forme de modules élémentaires connectés. La figure 3.18 illustre l'organisation des modules composant le processeur. Les principaux modules sont le décodeur d'instruction, l'unité d'exécution, l'unité de regroupement d'instruction et le prédicteur de branchement.

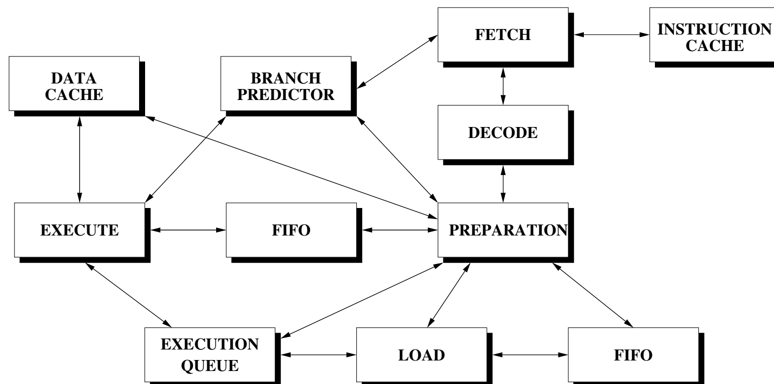

FIG. 3.18 – Processeur Java : Architecture interne

Pour tenir les contraintes de performances du processeur, la plupart des modules élémentaires décrivent un comportement combinatoire. Le graphe de dépendances combinatoires entre signaux contient un grand nombre de dépendances. La vitesse de simulation dépend alors principalement de la capacité du simulateur à optimiser l'ordonnancement des processus.

L'architecture contient environ 400 dépendances combinatoires pour près de 100 signaux. 33 processus doivent être ordonnancés pour évaluer ces signaux.

La figure 3.19 montre les vitesses de simulation pour SystemC 2.1.v1 d'OSCI, et pour les deux versions de SystemCASS.

Pour cette architecture, le simulateur à ordonnancement quasi statique est 7 fois plus rapide que le simulateur à ordonnancement dynamique. Ce gain attendu est principalement dû à l'absence d'échéancier en cours de simulation.

Le simulateur à ordonnancement quasi statique raisonne sur le graphe de dépendances entre processus. Le graphe obtenu, illustré par la figure 3.20, possède des cycles. Les processus contenus dans un cycle doivent être évalués jusqu'à stabilité. L'ordonnancement généré contient une boucle de relaxation dont le coût est important.

L'échéancier à ordonnancement totalement statique raisonne sur le graphe de dépendances entre signaux. L'ordonnancement totalement statique n'a pas besoin de boucle de relaxation et est donc encore plus rapide. Les résultats expérimentaux montrent que l'ordonnancement totalement statique est 1,6 fois plus rapide qu'un ordonnancement quasi statique, soit 12 fois plus rapide qu'un ordonnancement dynamique. La figure 3.21 illustre une partie du graphe de dépendances entre signaux (environ 60 dépendances sur les 400) pour montrer qu'il existe de nombreuses dépendances dans notre description du processeur Java.

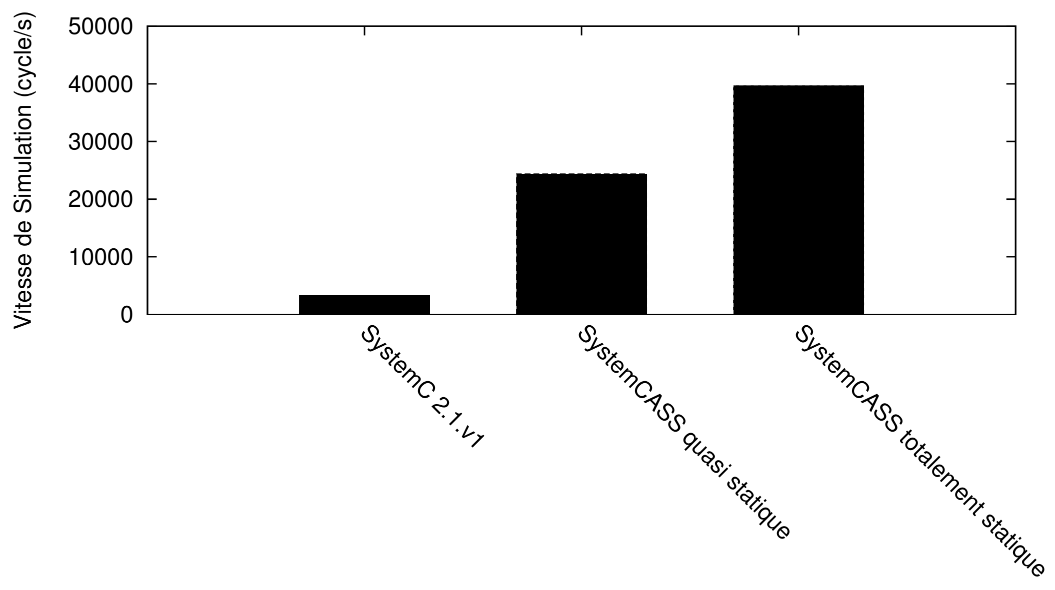

FIG. 3.19 – Processeur Java : Comparaison des vitesses de simulation

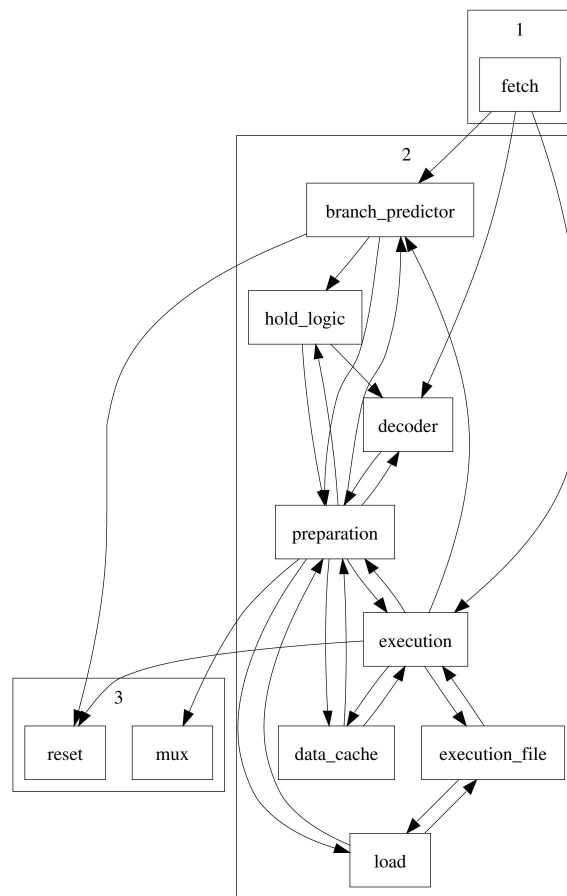

FIG. 3.20 – Processeur Java : Graphe de dépendances entre modules

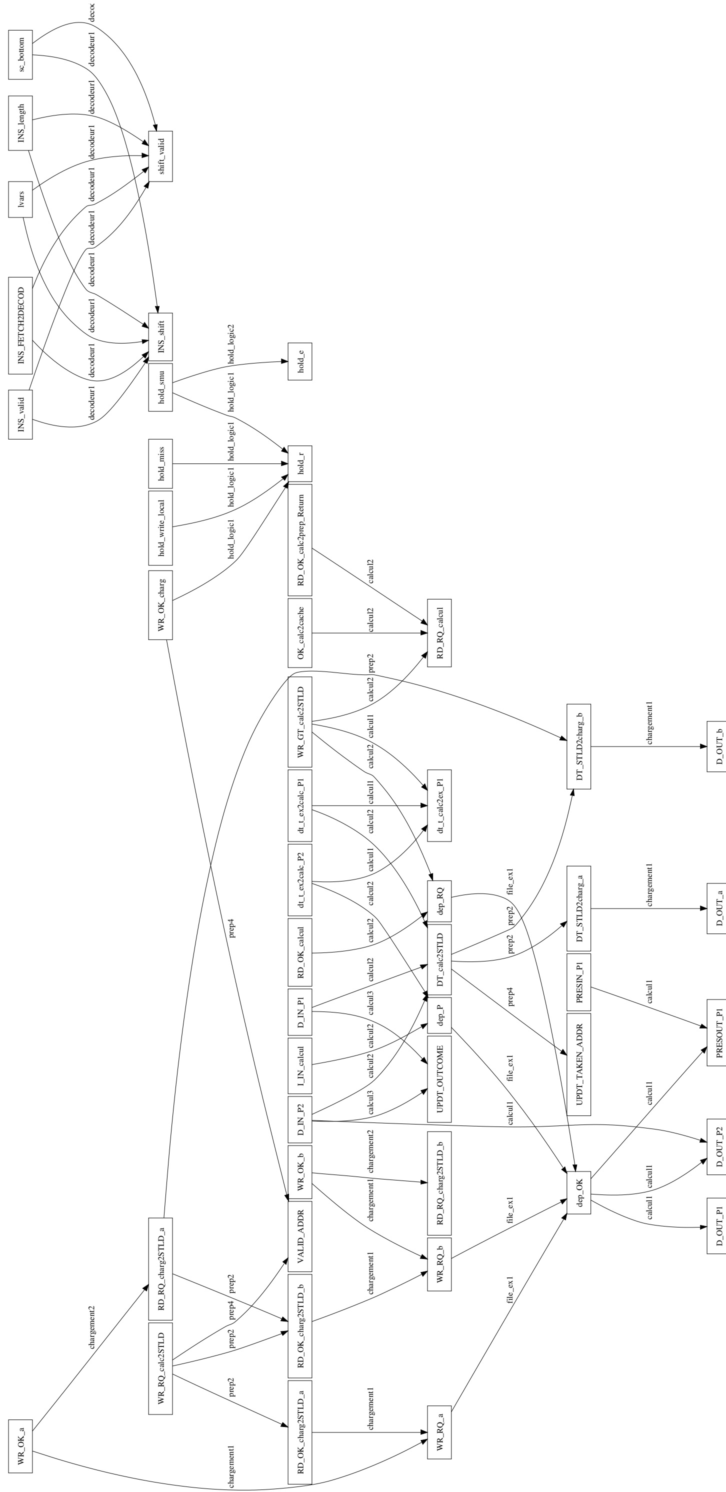

FIG. 3.21 – Processeur Java : Sous partie du graphe de dépendances entre signaux

3.4.2 Bilan des résultats

Les résultats expérimentaux montrent que le moteur de simulation présenté dans ce chapitre apporte une accélération qui peut dépasser un facteur 12 par rapport à un échéancier dynamique.

De plus, nous avons montré que la méthode totalement statique améliore sensiblement les performances par rapport à la méthode quasi statique dans le cas où l'architecture simulée comporte un grand nombre de dépendances combinatoires entre signaux.

Les vitesses enregistrées sont suffisantes pour travailler sur l'exploration architecturale de systèmes intégrés sur puce. Les travaux sur PAPR, sur la stéréo-vision, et sur le raytracer ont ainsi été facilité par l'accélération de la simulation apportée par SystemCASS.

3.5 Conclusions

La technique d'ordonnancement présentée s'inspire des deux échéanciers CASS et *FastSysC*. La méthode proposée va plus loin, et permet d'obtenir un ordonnancement totalement statique et minimal.

Cette méthode cible la simulation d'architecture matérielle multiprocesseur décrite au niveau CABA et même jusqu'au niveau micro-architecture. Les modèles de composants doivent respecter le modèle des automates synchrones communicants et fournir la liste des dépendances combinatoires entre les ports d'entrées et les ports de sortie.

Le moteur de simulation SystemCASS a été développé pour évaluer l'efficacité de la méthode. Les résultats expérimentaux montrent une accélération importante par rapport au simulateur SystemC de référence. De plus, l'ordonnancement totalement statique s'est révélé jusqu'à 60% plus rapide que l'ordonnancement quasi statique.

Cependant, le concepteur doit payer le prix : celui-ci doit décrire ses composants sous la forme des CFSM puis fournir explicitement, pour chaque processus, les dépendances entre les ports d'entrée et les ports de sortie. Le chapitre suivant présente une méthode ainsi qu'un outil de génération automatique de modèles de simulation respectant le modèle des CFSM.

Par ailleurs, si les dépendances entre ports ne sont pas fournies, l'ordonnancement totalement statique est impossible. Si elles sont erronées, la simulation est non déterministe. Pour éviter ce type de comportement, nous proposons, dans les chapitres suivants, une méthode ainsi qu'un outil de vérification de cohérence des modèles de simulation respectant le modèle des CFSM.

Chapitre 4

Génération automatique de modèle de simulation respectant le modèle des CFSM

Sommaire

4.1 État de l'art . . . 66
- 4.1.1 Verilog2SC ou VHDL2SC : Conversion par analyse syntaxique . . . 67
- 4.1.2 VASY : Conversion par analyse sémantique . . . 68
- 4.1.3 V2CASS : Génération de modèle C++ . . . 69
- 4.1.4 Problèmes non résolus . . . 71

4.2 Principe de la méthode proposée . . . 72
- 4.2.1 Construction d'un graphe de contrôle . . . 72
- 4.2.2 Réduction du graphe de contrôle . . . 74
- 4.2.3 Construction d'une liste d'assignations concurrentes . . . 75
- 4.2.4 Identification des registres . . . 77
- 4.2.5 Construction d'une représentation en CFSM . . . 77
- 4.2.6 Génération d'une description SystemC CABA en CFSM . . . 78

4.3 Techniques utilisées par VASY . . . 79
- 4.3.1 Construction d'un graphe de contrôle en réseau de Pétri . . . 79
- 4.3.2 Réduction du graphe de contrôle en réseau de Pétri . . . 82
- 4.3.3 Construction d'une liste d'assignations concurrentes . . . 87
- 4.3.4 Identification des registres . . . 87

4.4 Construction d'un automate et génération . . . 89
- 4.4.1 Construction du modèle CFSM . . . 89
- 4.4.2 Ordonnancement des assignations concurrentes . . . 92
- 4.4.3 Génération d'un automate en SystemC . . . 93

4.5 Résultats expérimentaux . . . 98
- 4.5.1 Architecture du coprocesseur Hadamard . . . 98
- 4.5.2 Limitation : Descriptions VHDL utilisant des tableaux . . . 100

4.6 Conclusions . . . 101

Lors de l'exploration architecturale, le concepteur de systèmes intégrés évalue et compare un nombre restreint de solutions de déploiement. Pour cela, il construit différentes architectures matérielles à partir de modèles de composants disponibles.

De nombreux modèles synthétisables de niveau RTL existent dans l'industrie. Étant destinés à la synthèse logique, ces modèles sont généralement écrits en VHDL ou Verilog, et offrent de très mauvaises performances en terme de vitesse de simulation. Or, la simulation d'architecture nécessite des temps de simulation très importants pouvant atteindre plusieurs jours de calculs. Il est donc préférable d'utiliser des modèles rapides en simulation.

La conversion manuelle d'un modèle au niveau RTL vers un modèle de simulation rapide est longue (plusieurs jours à plusieurs mois) et sujette à de nombreuses erreurs humaines. La conversion automatique, par un outil informatique, ne nécessite que très peu de temps (quelques secondes à quelques minutes) et supprime les risques d'erreurs humaines.

Les chapitres précédents ont décrit dans un premier temps une approche de modélisation basée sur les automates synchrones communicants ; puis, dans un second temps, une technique d'accélération de la simulation applicable à ces modèles. Pour bénéficier du gain apporté par cette accélération de la simulation, décrite dans les chapitres précédents, nous souhaitons obtenir automatiquement des modèles de simulation respectant la modélisation en CFSM.

L'objectif de ce chapitre est de présenter une méthode de génération automatique de modèles rapides pour la simulation à partir de modèles synthétisables de niveau RTL. La première section présente l'état de l'art en matière de génération de modèles précis au bit et au niveau cycle sur les interfaces. Nous développons ensuite notre méthode de génération.

4.1 État de l'art

Dans ce chapitre, nous poursuivons deux objectifs. D'une part, nous cherchons à décrire tous les modèles de composants dans un langage unique : le langage SystemC. Pour cela, nous devons résoudre un problème de syntaxe en utilisant des outils de conversion. D'autre part, nous souhaitons bénéficier de l'accélération de la simulation applicable aux modèles décrits sous la forme des CFSM. Pour cela, nous devons abstraire la description initiale pour construire un automate dont le comportement est équivalent à celui du modèle synthétisable de niveau RTL.

La génération automatique de modèles SystemC pour la simulation rapide, à partir de modèles synthétisables au niveau RTL, est constituée de deux problèmes distincts :

- La conversion de syntaxe : du langage VHDL vers le langage SystemC ;
- L'abstraction des modèles : du niveau RTL vers le niveau CFSM.

Dans l'état de l'art, nous présentons les outils suivants :

- Verilog2SC propose une analyse syntaxique pour convertir une description VHDL en SystemC sans changer son niveau d'abstraction.
- VASY utilise une analyse sémantique pour générer une description synthétisable VHDL en une autre description VHDL en utilisant un sous-ensemble différent du langage.
- V2CASS est un outil d'abstraction permettant de construire un automate à états finis à

partir d'une description synthétisable au niveau RTL. L'automate est ensuite généré en C/C++.

4.1.1 Verilog2SC ou VHDL2SC : Conversion par analyse syntaxique

Verilog2SC est un outil de conversion de description Verilog en description SystemC. Verilog2SC [LMAH02] repose sur une analyse syntaxique de la description initiale pour procéder à une conversion par mot clef. Il existe des outils similaires, tel que VHDL2SC[AFF+01], pour convertir des modèles VHDL en SystemC.

L'outil possède une table de conversion de mot clef. A chaque mot clef Verilog est associé un mot clef SystemC.

La table 4.1 fournit une liste non exhaustive d'association de mots clefs entre Verilog et SystemC.

Mot clef Verilog	**Mot clef SystemC**
module	SC_MODULE
input	sc_in
output, inout	sc_inout
wire, reg	bool, sc_logic
integer	int
real	double
case statement	C/C++ switch statement
assign	processus C++ (void assign_XXX () ...)
...	...

TAB. 4.1 – Liste non exhaustive d'association de mots clefs Verilog/SystemC

La conversion consiste à remplacer les mots clefs Verilog de la description initiale par les mots clefs SystemC à l'aide de la table d'association. Un processus est ainsi créé pour chaque mot clef *assign* rencontré. L'outil de conversion génère ensuite un constructeur par module. Le constructeur d'un module associe une liste de sensibilité à chaque processus.

La figure 4.1 illustre la conversion par mots clefs de Verilog2SC à l'aide d'un exemple. Le composant *test* Verilog est une porte logique *AND*. Ce composant possède deux entrées *a* et *b*, ainsi qu'une sortie *c* d'un bit. Le modèle SystemC décrit l'interface en utilisant les types standards *sc_in* et *sc_inout*. L'affectation de la sortie *c* est décrite sous la forme d'une méthode SystemC.

Bilan

La conversion par mots clefs permet de convertir un sous-ensemble seulement de la syntaxe Verilog vers le langage SystemC. La simple conversion par mots clefs n'est pas capable de

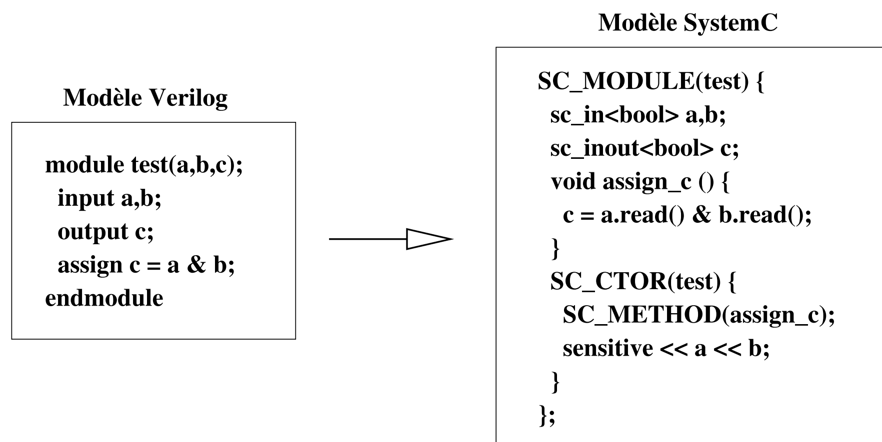

FIG. 4.1 – Verilog2SC : Exemple de la conversion d'un composant

traiter un sous-ensemble du langage Verilog suffisamment important pour convertir complètement des modèles de taille réelle. Le langage VHDL possède une sémantique assez différente du C/C++. En effet, certains mots clefs VHDL tels que *guarded* et *after* n'ont pas d'équivalents ni en C/C++, ni en SystemC.

De plus, un outil de conversion par analyse syntaxique ne permet pas de reconstruire les automates synchrones qui sont à la base des modèles CFSM.

4.1.2 VASY : Conversion par analyse sémantique

Les outils de synthèse logique reposent souvent sur une analyse syntaxique de la description du matériel. L'analyse reconnaît des constructions prédéfinies pour guider la synthèse notamment en permettant l'identification des registres. Chaque outil de synthèse possède ses propres constructions prédéfinies.

Lorsque le concepteur souhaite utiliser un autre outil de synthèse, certaines constructions prédéfinies utilisées dans la description initiale ne sont pas reconnues par le nouvel outil. Un travail de réécriture des modèles est alors nécessaire. L'outil *VHDL Analyzer for SYnthesis* (VASY), développé dans le cadre de la thèse [Lud99] de Ludovic Jacomme, a pour objectif d'automatiser ce travail de réécriture.

VASY repose sur une analyse sémantique de la description initiale sans utiliser aucune convention syntaxique. Les éléments mémorisants et l'horloge sont ensuite identifiés par une méthode formelle.

La méthode de VASY se déroule en trois grandes étapes successives :

1. Construction du graphe de contrôle représentant la sémantique de simulation, en utilisant le formalisme des réseaux de Pétri ;
2. Identification formelle des éléments mémorisants et de l'horloge ;

3. Génération d'une description synthétisable en utilisant les constructions syntaxiques reconnues par l'outil de synthèse ciblé.

Nous détaillons cette méthode dans les parties 4.2 et 4.3.

Bilan

La conversion de syntaxe par analyse sémantique est puissante. Cependant, la description générée par VASY représente le comportement du composant matériel au niveau RTL. En effet, VASY n'élève pas le niveau d'abstraction de la description initiale. VASY ne répond donc que partiellement à nos objectifs que sont, d'une part, la conversion de syntaxe et, d'autre part, l'abstraction de modèle du niveau RTL vers le niveau CFSM.

4.1.3 V2CASS : Génération d'un modèle comportemental décrit en C/C++

V2CASS [Lud99] est un outil de conversion de modèle synthétisable VHDL en modèle comportemental écrit en C/C++. Le modèle VHDL doit être décrit sous la forme d'un processus à plusieurs points de suspension (un point de suspension est défini par le mot clé *wait* du VHDL). La description VHDL initiale modélise une machine à états. Chaque point de suspension du processus représente un état de la machine. Chaque signal affecté représente une sortie de la machine.

Lorsque le processus se réveille, il s'exécute pour ensuite s'endormir à nouveau sur l'un des points de suspension. Le passage d'un point de suspension à un autre correspond à une transition d'état de l'automate.

La génération du modèle comportemental en langage C/C++ se déroule en trois étapes successives :

1. Construction d'un graphe de contrôle représentant le processus à plusieurs points de suspension ;
2. Construction d'un automate reproduisant le comportement du processus ;
3. Génération des expressions de l'automate en langage C/C++.

Un automate d'états est caractérisé par : <États, Entrées, Sorties, Transition, Génération>. La construction de l'automate consiste à déterminer l'ensemble des états, l'ensemble des entrées, l'ensemble des sorties, les conditions de transition entre chaque état, et enfin, l'ensemble des affectations présentes sur les différentes transitions.

Ensemble des états de l'automate

L'ensemble des états de l'automate est construit en créant un état $state_i$ pour chaque point de suspension w_i du graphe de contrôle.

La figure 4.2 illustre la construction de l'ensemble des états de l'automate décrit par un processus à plusieurs points de suspension. Dans cet exemple, le composant matériel possède trois états $W1$, $W2$ et $W3$.

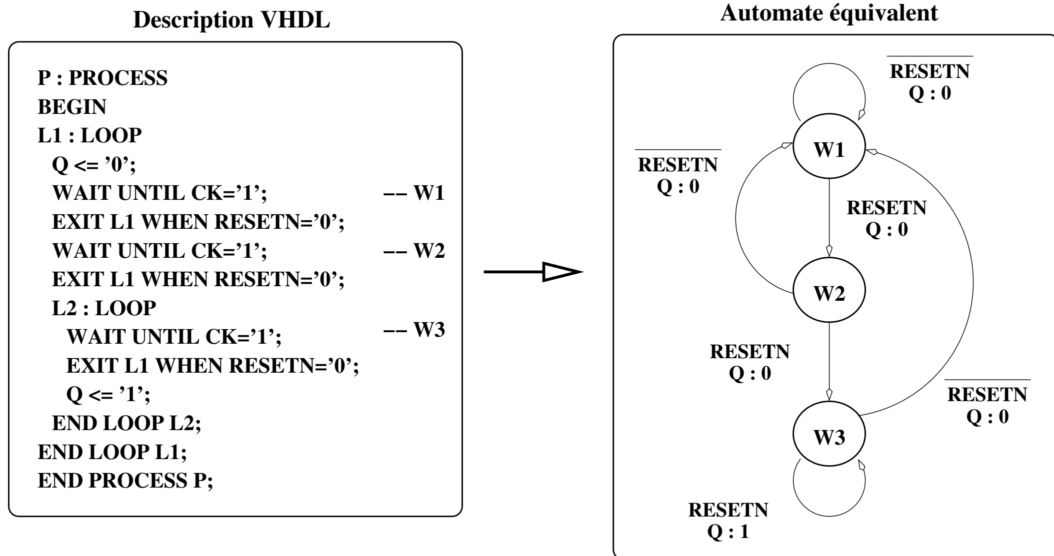

FIG. 4.2 – Construction de l'ensemble des états de l'automate correspondant à un processus à plusieurs points de suspension

Fonction de transition de l'automate

V2CASS énumère tous les chemins allant du point w_i au point w_j. La condition de transition entre $state_i$ et $state_j$ est équivalente à la condition d'exécution du chemin allant du point de suspension w_i au point de suspension w_j.

La transition de w_i vers w_j est notée $T_{(i,j)}$. Nous obtenons, pour notre exemple, les fonctions de transition suivantes :

- $T_{(1,1)} = \overline{RESETN}$
- $T_{(1,2)} = RESETN$
- $T_{(1,3)} = 0$
- $T_{(2,1)} = \overline{RESETN}$
- $T_{(2,2)} = 0$
- $T_{(2,3)} = RESETN$
- $T_{(3,1)} = \overline{RESETN}$
- $T_{(3,2)} = 0$
- $T_{(3,3)} = RESETN$

Fonction de génération de l'automate

La fonction de génération est l'ensemble des affectations exécutées sur le chemin allant du point w_i au point w_j.

L'ensemble des affectations présentes sur la transition de w_i vers w_j est notée $G_{(i,j)}$. L'affec-

tation de l'expression e dans la variable x est notée (x,e). Nous obtenons, pour notre exemple, les fonctions de transition suivantes :

- $G_{(1,1)} = \{(Q,0)\}$
- $G_{(1,2)} = \{(Q,0)\}$
- $G_{(1,3)} = \emptyset$
- $G_{(2,1)} = \{(Q,0)\}$
- $G_{(2,2)} = \emptyset$
- $G_{(2,3)} = \{(Q,0)\}$
- $G_{(3,1)} = \{(Q,0)\}$
- $G_{(3,2)} = \emptyset$
- $G_{(3,3)} = \{(Q,1)\}$

Bilan

V2CASS propose un générateur de modèle C/C++ décrit sous la forme d'un automate à états finis synchrones communiquant par des signaux. Cet outil est capable de convertir uniquement des descriptions comportementales qui sont sous la forme de processus à plusieurs points de suspension. Malheureusement, ce style d'écriture est peu utilisé dans l'industrie.

4.1.4 Problèmes non résolus

Notre objectif est la réécriture de modèles pour la simulation rapide. Les outils de conversion par analyse syntaxique, tels que Verilog2SC et VHDL2SC, ne traitent qu'un sous ensemble restreint du langage. La conversion par analyse sémantique, proposée par VASY, est capable de traiter un sous ensemble plus important du langage.

V2CASS est un outil capable de construire un automate à états finis synchrone. Malheureusement, la méthode ne peut convertir que les modèles à un processus comportant plusieurs points de suspension. La description d'un composant matériel est généralement composée d'assignations concurrentes, et de processus à un seul point de suspension.

La génération de modèle de niveau CFSM pose des problèmes non résolus par ces outils :

- Comment construire un (ou plusieurs) automate(s) représentant le comportement d'un composant décrit sous la forme d'assignations concurrentes, et de processus à un seul point de suspension ?
- Comment identifier les ports de Mealy d'un automate et comment construire la liste de dépendances entre les ports d'entrée et les ports de sortie au sein d'un même composant décrit au niveau RTL ?

Ce chapitre tente de répondre à ces questions en proposant une méthode ainsi qu'un outil.

4.2 Principe de la méthode proposée

La méthode permet de générer un modèle SystemC de type CFSM, à partir d'une description synthétisable VHDL de niveau RTL. Le modèle généré est constitué de plusieurs processus décrivant le comportement du composant matériel. Le modèle généré comporte également une liste de dépendances entre les signaux d'entrée et les signaux de sortie pour chaque processus. Cette liste est nécessaire dans le cas où nous souhaiterions utiliser l'ordonnancement totalement statique, décrit au chapitre 3. La méthode de génération proposée s'appuie sur les techniques d'analyse sémantique fournies par l'outil VASY.

La génération de modèle rapide pour la simulation se déroule en six étapes successives :

1. Construction d'un graphe de contrôle (par processus) représentant la description RTL ;
2. Réduction du ou des graphes de contrôle ;
3. Construction d'une liste d'assignations concurrentes décrivant le comportement global du composant matériel ;
4. Identification formelle des registres ;
5. Construction d'une représentation en CFSM dont le comportement est équivalent à celui de la description RTL initiale ;
6. Génération d'une description SystemC sous la forme d'un automate.

La figure 4.3 illustre les étapes de la méthode proposée. Les rectangles représentent les données sous forme de fichiers ou de structures. Les cercles représentent les algorithmes.

Les six étapes de la méthode sont présentées brièvement dans les sections suivantes.

4.2.1 Construction d'un graphe de contrôle pour chaque processus

Le comportement d'un composant matériel modélisé en VHDL au niveau RTL est généralement décrit par des assignations concurrentes et des processus synchrones à un seul point de suspension. Les assignations concurrentes sont traitées plus loin. Pour ce qui concerne les processus, la description VHDL est analysée pour construire un graphe de contrôle pour chaque processus. Ce graphe de contrôle est représenté par un réseau de Pétri.

Un réseau de Pétri est un graphe bipartite orienté. Il est composé de deux types de nœuds : les places et les transitions. Une place représente une instruction ou un bloc d'instructions. Une transition représente une relation de séquentialité. Une transition d'une place A vers une place B indique que les instructions de la place A sont exécutées avant celles de la place B. Un réseau de Pétri permet ainsi de décrire la partie contrôle d'un processus.

Nous enrichissons notre modèle en ajoutant, aux places et aux transitions, des informations sous forme d'étiquettes. Une étiquette, associée à une place, fait office de commentaire. Trois informations importantes sont associées à une transition :

- Une liste de sensibilité ;
- Une liste d'affectations séquentielles où chaque variable n'est affectée qu'une seule fois ;
- Une condition de franchissement.

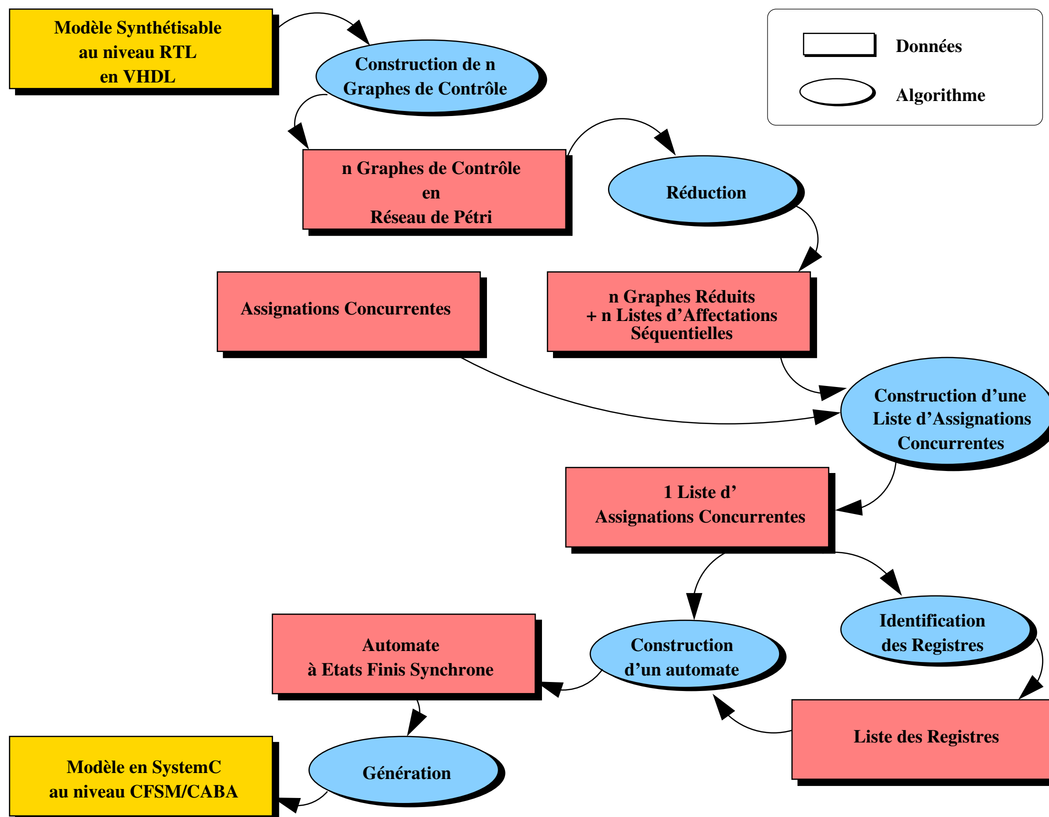

FIG. 4.3 – Méthode pour la génération de modèle rapide pour la simulation

Les listes de sensibilité sont représentées par une liste de signaux. Les conditions de franchissement, et les affectations sont des expressions, arithmétiques et/ou logiques, représentées par une structure de données baptisée VEX.

Les *VHDL EXpression* (VEX) permettent de décrire des expressions arithmétiques et logiques, au niveau bit ou au niveau vecteur de bits. La structure de données est constituée de nœuds. Un nœud représente un opérateur ou une opérande. Un nœud opérateur contient une liste chaînée d'opérandes.

La figure 4.4 illustre la structure de donnée VEX correspondant à l'expression $(reg+1)\&i(0)$. Le nœud racine de cette expression est un nœud représentant l'opérateur & de concaténation de bits. Ce nœud contient une liste chaînée de deux opérandes : la sous-expression $(reg+1)$ de largeur 3 bits et l'opérande $i(0)$, de largeur 1 bit.

La figure 4.5 présente un réseau de Pétri après l'étape de compilation. Le processus possède un unique point de suspension, une affectation, et une instruction conditionnelle. *asg* contient une liste ordonnée d'affectations uniques ; *g* une condition de franchissement ; et *w* une liste de sensibilité.

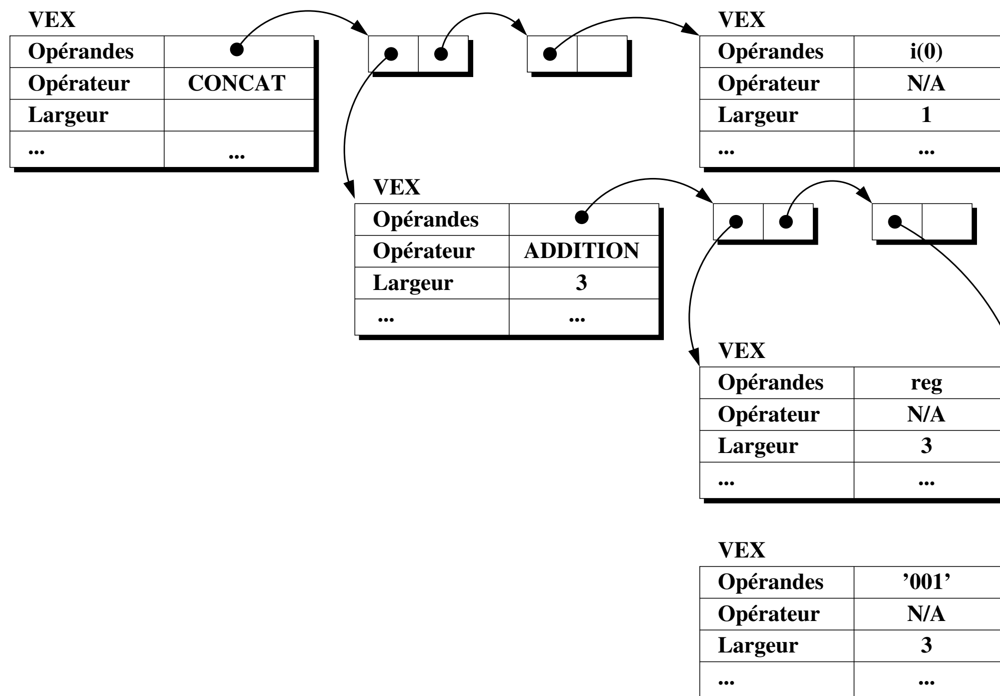

FIG. 4.4 – Représentation d'une expression par la structure de données VEX

Le nombre de chemins d'exécution et la taille du réseau de Pétri dépendent de la complexité de la description VHDL initiale.

4.2.2 Réduction du graphe de contrôle

La complexité de l'analyse sémantique dépend du nombre de chemins d'exécution du graphe à explorer. L'étape de réduction cherche à réduire le nombre de chemins d'exécution ainsi que la taille du réseau de Pétri pour faciliter l'analyse sémantique. Cette étape effectue également les renommages de variables nécessaires pour respecter la condition d'affectation unique.

La réduction du graphe de contrôle vise à obtenir une forme canonique. La figure 4.6 illustre un réseau de Pétri après l'étape de réduction. Le graphe réduit possède alors seulement deux places et deux transitions.

La transition t_w représente l'unique point de suspension du processus. La condition de franchissement de t_w contient la liste de sensibilité du processus. La transition t_{asg} représente le comportement du processus sous la forme d'une seule liste ordonnée d'affectations uniques.

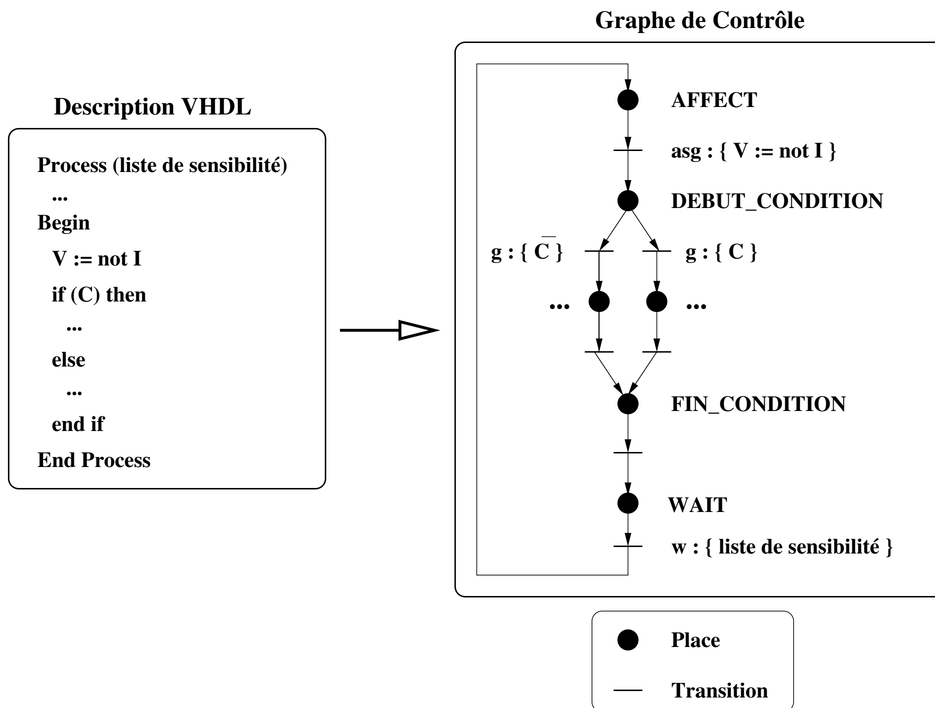

FIG. 4.5 – Graphe de contrôle représenté par un réseau de Pétri

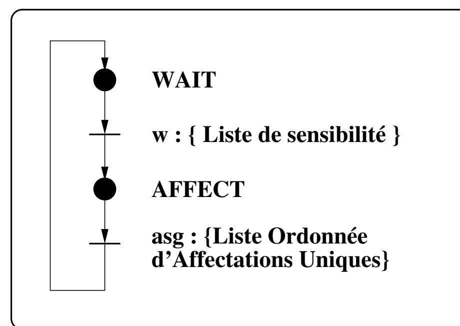

FIG. 4.6 – Graphe de contrôle réduit représenté par un réseau de Pétri

4.2.3 Construction d'une liste d'assignations concurrentes décrivant le comportement global du composant

La description VHDL initiale comporte des processus synchrones, et des assignations concurrentes. Un processus est représenté, après construction puis réduction, sous la forme d'un graphe de contrôle réduit. Un graphe de contrôle réduit contient une liste ordonnée d'affectations uniques représentant le comportement d'un processus. Toute assignation concurrente (à un

signal ou à un registre) peut être considérée comme un processus élémentaire. Un composant matériel est donc représenté par un ensemble de listes ordonnées d'affectations uniques, et une liste d'assignations concurrentes.

Le listing **4.1** montre un exemple de description VHDL que nous souhaitons traiter. $input1$ et $input2$ sont deux ports d'entrée. $temp$ est une variable temporaire. $output1$ et $output2$ sont deux ports de sortie. $register1$ est un registre. Trois assignations concurrentes calculent les valeurs de $temp$, de $output1$, et de $output2$. Un processus synchrone modifie la nouvelle valeur du registre $register1$ sur le front montant de l'horloge. Dans cet exemple, tous les registres, et tous les ports hormis l'horloge, sont 4 bits.

```
1  process (ck)
2  begin
3    if (ck='1' and ck'event) then
4      register1 <= register1 - input1;
5    end if;
6  end process;
7
8  output1 <= register1;
9  temp    <= input2 + register1;
10 output2 <= temp and "1001";
```

Listings 4.1 – Description VHDL synthétisable au niveau RTL

L'objectif de la troisième étape de la méthode est de construire une liste d'assignations concurrentes représentant le comportement global du composant. Pour cela, il suffit de fusionner les assignations concurrentes, et les listes ordonnées d'affectations uniques de chaque processus en une seule liste d'assignations concurrentes. La nouvelle liste d'assignations concurrentes ainsi créée n'est pas ordonnée.

La figure 4.7 représente la liste d'assignations concurrentes correspondant à la description VHDL du listing **4.1**. Les écritures dans les variables sont concurrentes. La mémorisation du registre $register1$ est synchrone avec l'horloge.

$$output1 := register1$$
$$temp := input2 + register1$$
$$output2 := temp \ and \ "1001"$$
$$register1 := (CK \ and \ CK.event())?(register1 - input1) : (register1)$$

FIG. 4.7 – Liste d'assignations concurrentes décrivant le comportement global d'un composant

Les ports, et la direction des ports sont connus. Cependant, les registres, ne sont pas identifiés, alors que les informations sont nécessaires pour la construction de l'automate.

4.2.4 Identification des registres

Cette étape consiste à analyser la liste d'assignations concurrentes afin de déterminer pour chaque variable si elle se comporte ou non comme un registre.

Un registre a deux comportements possibles :

- Le registre conserve sa valeur lorsque la condition de mémorisation c^{mem} est vraie ;
- Le registre enregistre une nouvelle valeur dans le cas contraire.

Les deux conditions sont mutuellement exclusives. La somme des conditions vaut 1.

$$c^{mem} \cdot c^{write} = 0$$
$$c^{mem} + c^{write} = 1$$

Pour déterminer si une variable est un registre, il suffit de retrouver la condition d'écriture et la condition de mémorisation. Si la condition de mémorisation est toujours fausse, la variable affectée correspond à un élément purement combinatoire. Si la condition est toujours vraie, la variable affectée est une constante.

Dans le cadre de la génération d'automate, nous cherchons à identifier les registres à écriture synchrone. Notre méthode de génération n'accepte pas les registres asynchrones.

Les écritures dans les registres étant synchrones, la condition de mémorisation c_{mem} de chaque registre est un front d'une horloge unique CK. De plus, la liste de sensibilité du processus pilotant chacun des registres doit inclure ce signal d'horloge CK. Si un registre ne respecte pas ces contraintes, la génération d'automate s'arrête.

4.2.5 Construction d'une représentation en CFSM

Les étapes 4.2.1, 4.2.2, 4.2.3, et 4.2.4 nous ont permis de générer une représentation totalement parallèle, sous forme d'un ensemble d'assignations concurrentes, en partant d'un ensemble de processus. Nous devons maintenant effectuer un travail inverse, en reconstruisant $2+n$ processus séquentiels. En effet, un automate est décrit par trois types de fonction : transition, génération de Moore, génération de Mealy. La modélisation SystemC que nous proposons utilise $2+n$ processus : un processus de type transition, un processus de type génération de Moore, et n processus de type génération de Mealy (un par port de sortie de Mealy). La construction de l'automate consiste à classer et ordonnancer les assignations concurrentes, obtenues à l'étape précédente, dans les $2+n$ processus du modèle des CFSM. L'ensemble des assignations concurrentes peut être représenté par un graphe orienté acyclique. Le classement des assignations est obtenu par parcours de ce graphe, en analysant les dépendances entre variables et les types de ces variables. Ceci est décrit en détail dans la section 4.4.

La figure 4.8 illustre le classement des assignations concurrentes de la figure 4.7 dans le modèle des CFSM. L'assignation du registre $register1$ est classée dans la fonction de transition. L'assignation de $output1$ est classée dans la fonction de génération de Moore car le port de sortie $output1$ dépend uniquement de registres. L'assignation de $output2$ et celle de $temp$ sont classées

dans la fonction de génération de Mealy car le port de sortie $output2$ dépend de $temp$ qui dépend lui même directement d'une entrée.

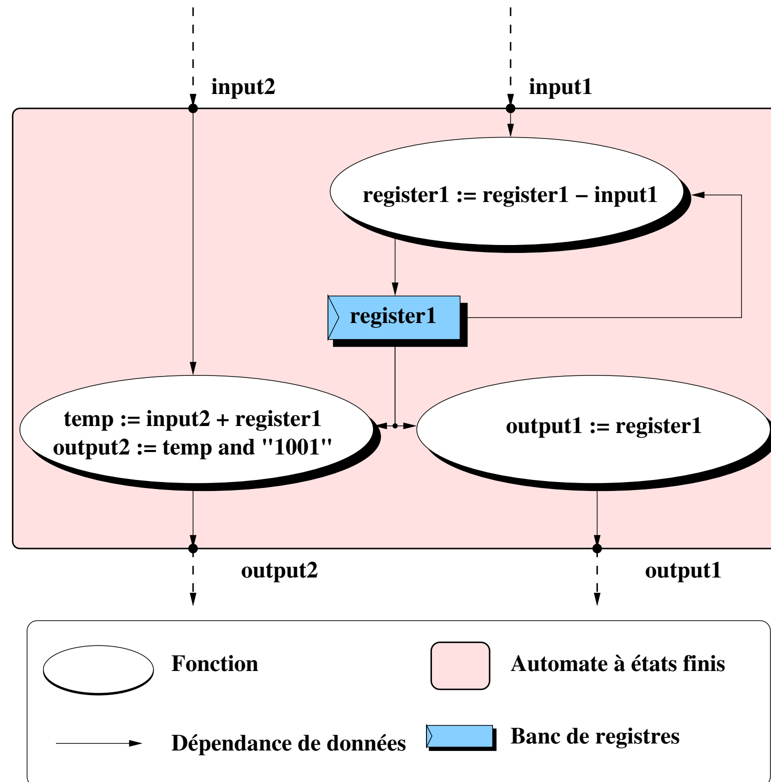

FIG. 4.8 – Classement et ordonnancement des assignations concurrentes dans le modèle des CFSM

4.2.6 Génération d'une description SystemC CABA en CFSM

La dernière étape utilise la représentation en CFSM précédemment construite pour générer une description SystemC. La description définit :

- L'interface du composant matériel au bit près ;
- Le comportement du matériel au cycle près à l'aide des processus définis ci-dessus ;
- Une liste de sensibilité par processus ;
- Les dépendances entre les ports d'entrée et les ports de sortie au sein de chaque processus.

La description SystemC ainsi générée respecte la modélisation présentée au chapitre 2 et est compatible avec les moteurs de simulation SystemC d'OSCI et SystemCASS décrits dans le chapitre 3.

4.3 Techniques utilisées par VASY

La méthode proposée est composée de six étapes successives. Les quatre premières sont des techniques implantées par l'outil VASY : construction du graphe de contrôle de chaque processus, réduction des graphes de contrôle, construction d'une liste d'assignations concurrentes décrivant le comportement global du matériel, puis identification des registres.

Cette section constitue un rappel des techniques utilisées par l'outil VASY. Ces techniques sont détaillées dans la thèse de Ludovic Jacomme [Lud99].

4.3.1 Construction d'un graphe de contrôle en réseau de Pétri

Le graphe de contrôle est d'abord créé avec une seule place représentant l'état initial. Toutes les instructions séquentielles sont ensuite traduites en un ensemble de places et de transitions reproduisant le comportement du processus. Un processus décrivant le comportement d'un composant matériel ne se termine pas. Un arc relie donc la dernière instruction à la première place modélisant ainsi l'exécution sans fin du processus.

La figure 4.9 représente un graphe de contrôle sous la forme d'un réseau de Pétri. La place $P1$ et la transition $T1$ représentent le début du processus. La transition Tn modélise la boucle d'exécution du processus.

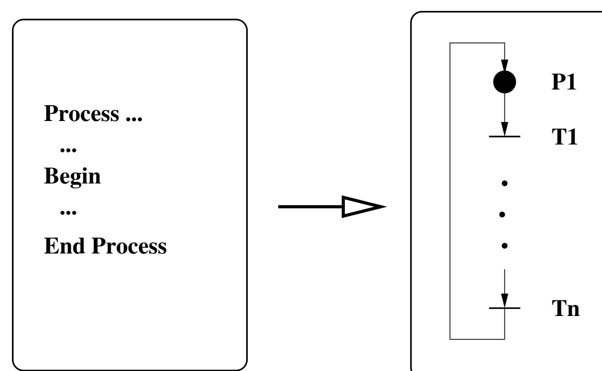

FIG. 4.9 – Graphe de contrôle d'un processus sous la forme d'un réseau de Pétri

Il existe quatre types d'instruction :

- Les affectations séquentielles
- Les instructions conditionnelles
- Les instructions de boucle
- Les instructions de suspension

Affectations séquentielles

Chaque affectation séquentielle est traduite en une place étiquetée $AFFECT$ et une transition. L'expression de l'affectation est associée à la transition et est représentée par des VEX.

La figure 4.10 illustre la description VHDL de deux affectations séquentielles et sa représentation en réseau de Pétri.

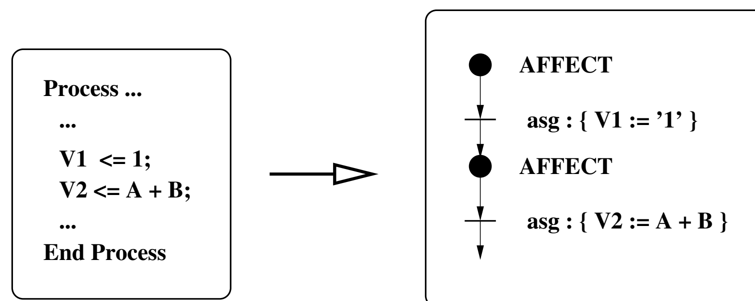

FIG. 4.10 – Réseau de Pétri représentant deux affectations séquentielles

Instructions conditionnelles

Les instructions conditionnelles, telles que $IF/THEN/ELSE$ ou $SWITCH/CASE$, permettent de choisir un chemin d'exécution parmi $n \geq 2$ en fonction d'une condition. Pour représenter ce type d'instruction, nous ajoutons deux places de sorte que tous les chemins d'exécution commencent par la place $DEBUT\ CONDITION$ et terminent par la place $FIN\ CONDITION$. La première transition de chaque chemin d'exécution est associée à une expression conditionnelle représentée par des VEX.

La figure 4.11 illustre la description VHDL d'une instruction conditionnelle de type $CASE/SWITCH$ et sa représentation en réseau de Pétri.

Instructions de boucle

Les instructions de boucle sont *while*, for, $loop$, *next*, et *exit*. Nous détaillons ici uniquement l'instruction *while* à titre d'exemple.

La représentation de *while* commence par une place de type $DEBUT\ BOUCLE$ et deux transitions avec expressions conditionnelles représentant la condition de la boucle. Ces transitions mènent vers deux chemins d'exécution :

- L'une modélise le corps de la boucle et commence par une place de type $LOOP$;
- L'autre modélise la fin de la boucle et est marquée par une place de type $END\ LOOP$.

La figure 4.12 montre une instruction *while* en VHDL et sa représentation en réseau de Pétri. Nous pouvons distinguer les deux chemins d'exécution :

- Le chemin de $LOOP$ jusqu'à $WHILE$ constitue le corps de la boucle.

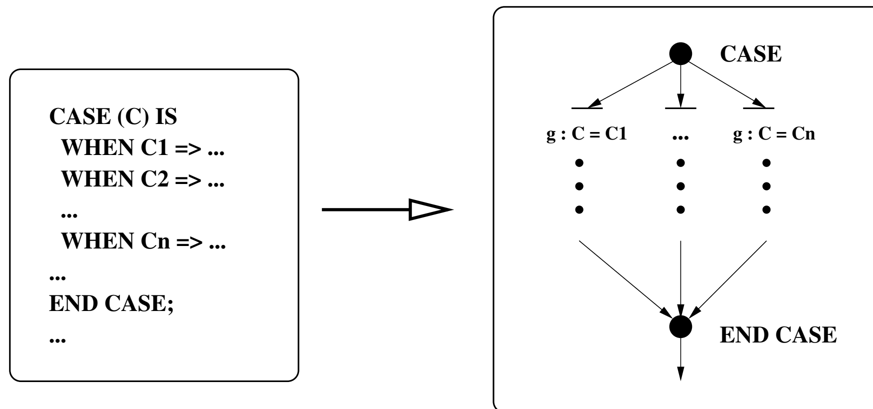

FIG. 4.11 – Réseau de Pétri représentant une instruction conditionnelle

- Le chemin de *WHILE* jusqu'à *END LOOP* constitue la condition de fin de boucle.

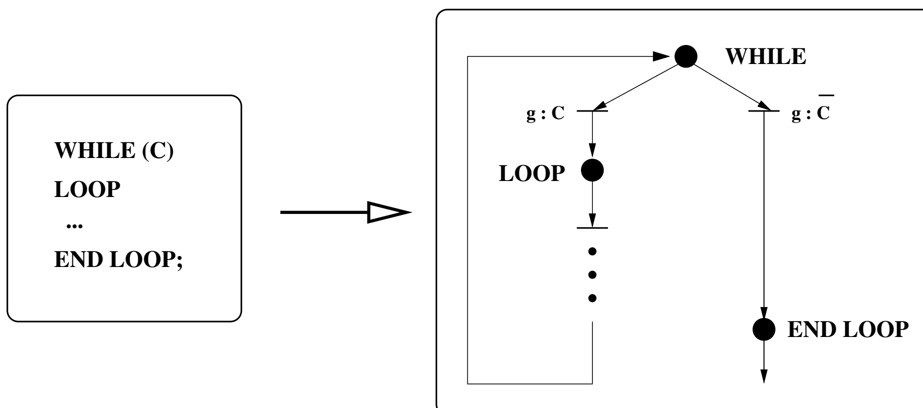

FIG. 4.12 – Réseau de Pétri représentant une instruction de boucle

Instructions de suspension

Les instructions de suspension permettent de suspendre l'exécution du processus courant. Le processus reprend son exécution lorsqu'un évènement apparaît sur un des ports spécifiés en paramètres. Cette liste de ports est la liste de sensibilité.

L'instruction *WAIT* est représentée par une place de type *WAIT* et une transition associée à une liste de sensibilité.

La figure 4.13 illustre un point de suspension, défini en VHDL par une instruction *WAIT ON*, et sa représentation en réseau de Pétri. L'exécution du processus reprend lorsqu'un évènement apparaît sur le port d'entrée *CLK*.

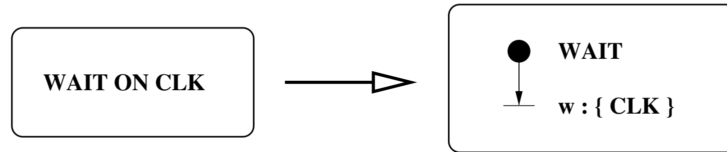

FIG. 4.13 – Réseau de Pétri représentant une instruction de suspension

4.3.2 Réduction du graphe de contrôle en réseau de Pétri

L'objectif est d'obtenir un graphe de contrôle canonique. Pour cela, VASY applique une suite de techniques de réduction sur le graphe de contrôle construit à partir de la description VHDL.

Les techniques de réduction sont :

- La réduction locale séquentielle ;
- La réduction locale latérale ;
- La réduction globale ;
- La réduction des boucles.

Les quatre techniques de réductions sont présentées dans les sections suivantes. Les propriétés du graphe réduit sont ensuite décrites.

Réductions locales séquentielles

La réduction séquentielle consiste à fusionner deux transitions, $t1$ et $t2$, de type $AFFECT$ en une seule transition, t_{res}, de type $AFFECT$. La réduction séquentielle peut s'appliquer lorsque les deux transitions consécutives t_1 et t_2 vérifient les conditions suivantes :

- t_1 et t_2 sont des transitions de type $AFFECT$ représentant des instructions d'affectation ;
- t_2 est l'unique successeur de t_1 ;
- La condition de franchissement de t_1 vers t_2 est toujours vraie.

Une variable peut être affectée plusieurs fois sur un même chemin d'exécution. Ce cas apparaît notamment lorsque le concepteur souhaite réutiliser une variable temporaire pour des usages différents. Dans les listes d'affectations uniques asg^1 et asg^2, respectivement associées à $t1$ et $t2$, les variables sont affectées une seule fois. Cependant, une variable peut être affectée à la fois dans $t1$ et dans $t2$.

Les listes d'affectations uniques asg^1 et asg^2 sont concaténées pour former une nouvelle liste ordonnée d'affectations équivalente asg^{res}.

Si une même variable v est affectée dans les deux transitions $t1$ et $t2$, VASY procède à des renommages de variable pour conserver la propriété : chaque variable est affectée une seule fois. La première affectation de la variable v dans asg^{res} est renommée en v'. Toutes les occurrences de v dans le membre de droite d'une affectation située entre les affectations $asg^{res}_{v'}$ et asg^{res}_{v} sont également renommées en v'.

La figure 4.14 illustre le résultat d'une réduction séquentielle. Le graphe de contrôle initial contient deux transitions. La première est associée à l'affectation $V1 := A$. La seconde est associée à l'affectation $V2 := B$. La transition résultant de la réduction contient alors deux affectations séquentielles $V1 := A$ et $V2 := B$.

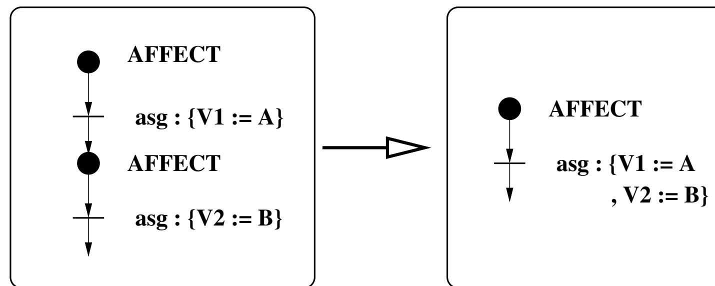

FIG. 4.14 – Réduction séquentielle du graphe de contrôle

Un deuxième exemple, illustré par la figure 4.15, présente une réduction séquentielle nécessitant le renommage des variables qui sont affectées plusieurs fois sur le même chemin d'exécution. La première transition est associée aux affectations séquentielles $V := A; B := V$. La seconde transition est associée aux affectations séquentielles $C := V; V := V + D; E := V$. La variable V est affectée deux fois. VASY procède donc à un renommage de V. Après renommage puis concaténation, nous obtenons une seule liste ordonnée d'affectations uniques : $V' := A; B := V'; C := V'; V := V' + D; E := V$.

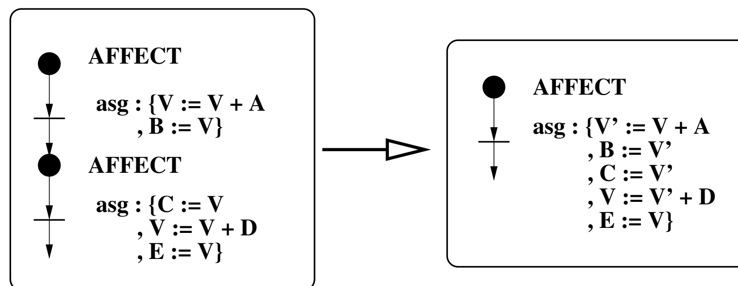

FIG. 4.15 – Réduction séquentielle du graphe de contrôle (2)

Réductions locales latérales

La réduction latérale consiste à fusionner un ensemble $T_{lateral} = \{t_1, ..., t_n\}$ de transition en une seule transition t_{res}. Une nouvelle liste ordonnée d'affectations uniques est associée à la transition résultante t_{res}. Cette liste d'affectations uniques est équivalente aux listes d'affectations de $t_1, ..., t_n$.

Une même variable v peut être affectée par plusieurs chemins d'exécution t_i. L'affectation de la variable v sur le chemin t_i est notée asg_v^i. La méthode de VASY est de remplacer ces affectations par une seule affectation utilisant une expression conditionnelle, telle que :

$$asg_v^{res} = (g_1?asg_v^1 : ...(g_i?asg_v^i : ...(g_n?asg_v^n : v)))$$

La réduction latérale peut s'appliquer lorsque toutes les transitions de $t_1,...,t_n$ vérifient les conditions suivantes :

- Toutes les transitions t_i représentent des instructions d'affectations ;
- Toutes les transitions t_i possèdent le même prédécesseur et le même successeur ;
- Toutes les transitions t_i ont chacune une expression conditionnelle mutuellement exclusive g_i :

$$g_i \cdot g_j = 0 \text{ si } i \neq j$$
$$\sum_i^n g_i = 1$$

La figure 4.16 illustre le résultat d'une réduction latérale. Le graphe de contrôle initial contient deux places *IF* et *END IF*. Deux transitions décrivent deux chemins d'exécution disjoints possibles. Après réduction, le graphe de contrôle contient une place et une seule transition T_{res}. L'affectation de V associée à la transition T_{res} est équivalente à l'exécution des places *IF* à *END IF*.

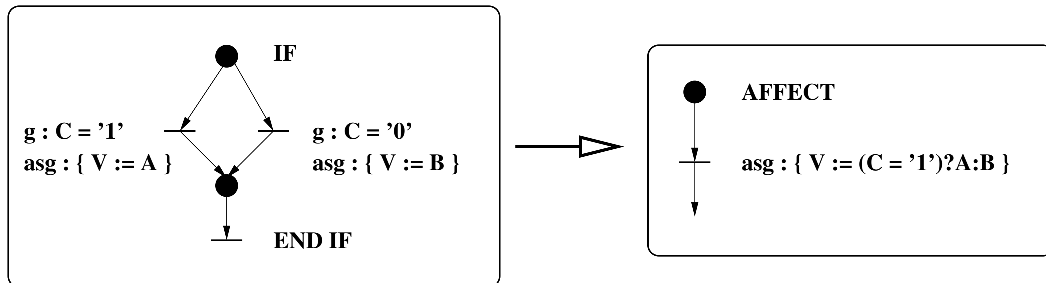

FIG. 4.16 – Réduction latérale du graphe de contrôle

Réductions globales

La réduction globale consiste à énumérer tous les chemins d'exécution du graphe de contrôle puis à construire un nouveau graphe de contrôle. Ce nouveau graphe est appelé graphe de contrôle semi réduit.

Le graphe de contrôle d'un processus à un seul point de suspension comporte une seule place t_w de type *WAIT*. L'énumération de tous les chemins d'exécution consiste à chercher tous les chemins possibles allant de t_w à t_w.

La figure 4.17 illustre un exemple d'énumération des chemins d'exécution. Le processus comporte deux instructions conditionnelles *IF* imbriquées. Dans le graphe de contrôle de ce processus, il existe trois chemins allant de t_w à t_w.

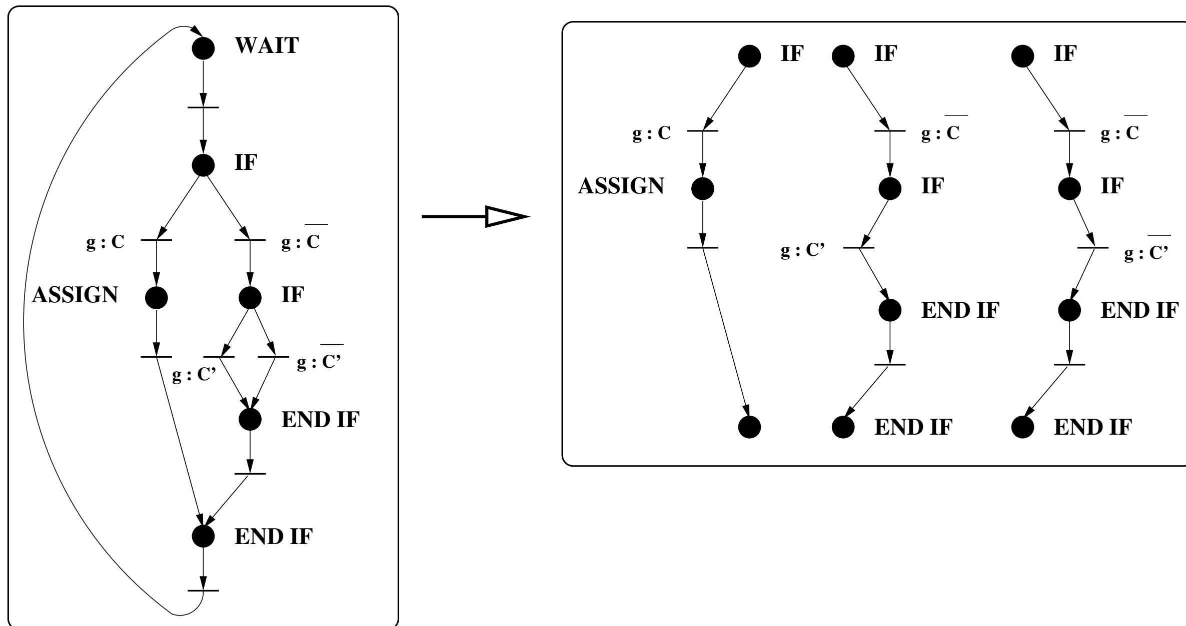

FIG. 4.17 – Énumeration des chemins du graphe de contrôle

Le graphe de contrôle semi réduit est illustré par la figure 4.18. et est construit comme suit :

- Le début du processus est représenté par une place de type *WAIT* et par une transition contenant la liste de sensibilité du processus ;
- Une place de type $AFFECT$ et une transition contenant toutes les affectations de variables qui sont effectuées sur les différents chemins d'exécution ;
- Deux places $DEBUT\ CONDITION$ et $FIN\ CONDITION$ sont séparées par $n \geq 2$ transitions représentant les n chemins d'exécution possibles ;
- Une transition relie la dernière place de type $FIN\ CONDITION$ avec la première place du processus de type $WAIT$.

Le graphe de contrôle semi réduit peut ensuite être réduit par une réduction latérale puis une réduction séquentielle.

Le graphe 4.19 illustre la réduction finale du graphe de contrôle semi réduit. Le résultat des réductions permet d'obtenir un graphe avec seulement deux places et deux transitions. La première place représente le point de suspension du processus. Le deuxième représente l'ensemble des opérations décrivant le comportement du processus. La deuxième place est reliée à la première par une transition.

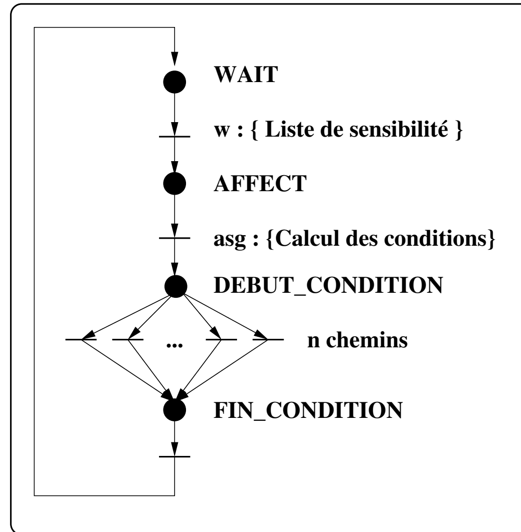

FIG. 4.18 – Graphe de contrôle semi réduit

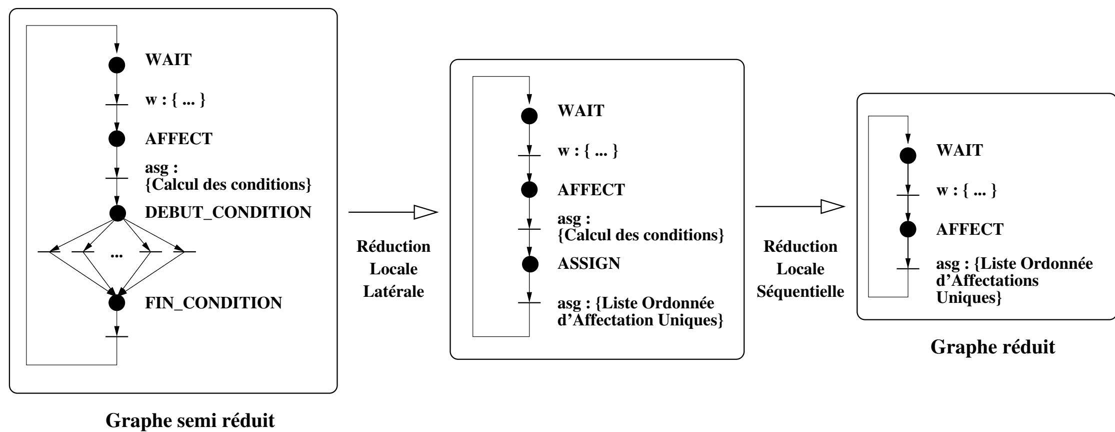

FIG. 4.19 – Réduction finale du graphe de contrôle réduit

Réduction des boucles

Les réductions locales séquentielles et latérales ne permettent pas de réduire complétement un graphe de contrôle comportant une boucle.

La figure 4.12 montre un graphe de contrôle pour lequel les réductions échouent. La condition de fin de boucle *WHILE* est calculée par la transition t_{cond}. La transition t_{cond} possède deux flèches sortantes. L'une mène à l'exécution du corps de boucle. L'autre mène à la fin de la boucle. La réduction latérale ne peut pas réduire ces deux chemins d'exécution car les deux chemins ont une place qui succède différente. Si le nombre d'itérations de la boucle n'est pas

connu, la réduction globale ne peut pas énumérer tous les chemins d'exécution.

Nous ne nous intéressons qu'à des descriptions VHDL synthétisables. Pour qu'une description VHDL soit synthétisable, il faut que les bornes de chaque boucle puissent être déterminées statiquement. Nous pouvons alors dérouler complètement les boucles. Une boucle ainsi déroulée est ensuite réduite à l'aide des techniques de réductions locales.

En pratique, VASY ne sait dérouler que les boucles FOR dont les bornes sont constantes. Les boucles $WHILE$ et $LOOP$ ne sont pas déroulées car les bornes sont difficiles à déterminer statiquement.

Propriétés du graphe de contrôle réduit

Le graphe de contrôle réduit est obtenu par construction puis réduction du graphe de contrôle semi réduit correspondant à la description initiale VHDL. Toutes les descriptions VHDL synthétisables RTL peuvent être représentées sous la forme d'un graphe de contrôle réduit. Ce graphe, illustré par la figure 4.6, contient uniquement deux transitions t_{asg} et t_w reliées par des arcs à deux places de type $WAIT$ et $AFFECT$. La transition t_w représente le point de suspension et contient la liste de sensibilité du processus. La transition t_{asg} décrit le comportement du processus à l'aide d'une liste ordonnée d'affectations uniques.

Les affectations sont séquentielles. Chaque variable n'est affectée qu'une seule fois.

Le graphe de contrôle a la propriété d'être canonique et minimal. Ces propriétés sont exploitées lors de la construction de l'automate dans les étapes suivantes.

4.3.3 Construction d'une liste d'assignations concurrentes décrivant le comportement global du composant

La description VHDL initiale comporte des assignations concurrentes et des processus synchrones. Pour chaque processus de la description VHDL initiale, il existe un graphe de contrôle réduit contenant, dans la transition t_{asg}, une liste ordonnée d'affectations uniques. Les listes ordonnées d'affectations uniques, et les assignations concurrentes sont fusionnées en une seule liste d'assignations concurrentes.

Une même variable doit être affectée par un seul processus, ou bien par une seule assignation concurrente. En effet, si un même nom de variable locale apparaît dans plusieurs processus, celle-ci est renommée en prenant en compte son contexte d'utilisation. La nouvelle liste d'assignations concurrentes conserve ainsi la propriété suivante : chaque variable est affectée une seule fois.

4.3.4 Identification des registres

L'identification des registres consiste à analyser toutes les assignations concurrentes décrivant le comportement global du matériel. Chaque bit d'une même variable est susceptible d'avoir un comportement différent. Les expressions d'assignation sont donc analysées bit par bit. Pour cela, les expressions d'assignation VEX sont d'abord converties en *Reduced Ordered Binary*

Decision Diagram(s) (ROBDD). VASY raisonne ensuite sur les ROBDD représentant les expressions d'assignation d'un bit.

Un registre est une variable capable de conserver sa valeur. Le support étendu[1] de l'assignation du bit b d'une variable v comprend v :

$$v_b \in SuppExt(v_b)$$

Pour retrouver la condition de mémorisation et la condition d'écriture du bit b d'une variable v, il suffit d'effectuer une décomposition de Shannon[Mor82]. Une décomposition de Shannon peut être notée comme ceci :

$$f(v_b,x_1,...,x_n) = (v_b \wedge f^+_{v_b}(x_1,...,x_n)) \vee (\bar{v_b} \wedge f^-_{v_b}(x_1,...,x_n))$$
avec :
$$f^+_{v_b}(x_1,...,x_n) = f(1,x_1,...,x_n)$$
$$f^-_{v_b}(x_1,...,x_n) = f(0,x_1,...,x_n)$$

La condition de mémorisation est la condition pour laquelle $f_{v_b} = v_b$. Cette condition s'écrit

$$c^{mem} = f^+_{v_b}(x_1,...,x_n) \wedge \overline{f^-_{v_b}(x_1,...,x_n)}$$

VASY construit d'abord le graphe de décision binaire de la fonction $f(v_b,x_1,...,x_n)$. Il calcule ensuite les fonctions $f^+_{v_b}$ et $f^-_{v_b}$, puis la condition de mémorisation c_{mem}. VASY distingue trois cas de figure :

- Si $c^{mem} = 1$, alors $f(v_b) = v_b$. La variable v est donc une constante.
- Si $c^{mem} = 0$, alors v n'est pas un registre puisque cette variable ne conserve jamais sa valeur.
- Dans tous les autres cas, v est un registre.

Nous pouvons distinguer deux types de registre :

- Les registres synchrones ;
- Les registres asynchrones.

La méthode d'identification des registres, présentée ici, ne permet pas de faire la distinction entre des registres synchrones et des registres asynchrones. Nous pouvons appliquer la méthode décrite dans la thèse de Ludovic Jacomme pour déterminer formellement ces propriétés.

L'identification des registres permet d'obtenir une liste des registres employés dans la description VHDL initiale du matériel. La liste d'assignations concurrentes, où tous les registres sont identifiés, permet de construire une représentation respectant le modèle des automates communicants.

[1] La définition du support étendu est donnée dans l'annexe A.

4.4 Construction d'un automate et génération du modèle en SystemC

Cette section décrit les deux dernières étapes qui constituent l'une des contributions de cette thèse : la construction du modèle CFSM puis la génération de la description SystemC correspondante.

4.4.1 Construction du modèle CFSM

A l'issue du traitement effectué par VASY, le comportement du composant matériel est décrit par une seule liste d'assignations concurrentes. De plus, les variables de type registre sont identifiées.

L'objectif de cette étape est de construire l'automate CFSM sous la forme de $2+n$ processus. La construction consiste alors à classer et à ordonnancer les assignations concurrentes dans ces $2+n$ processus.

Le classement d'une assignation est déterminé en fonction des dépendances de données vis à vis des registres et/ou des ports d'entrée. Nous effectuons donc une analyse des dépendances afin de déterminer, pour chaque assignation, le classement dans un processus et l'ordre d'exécution au sein de ce processus. L'analyse se déroule en trois étapes successives : la construction d'un graphe de dépendances entre variables, l'étiquetage des nœuds pour les associer à un ou plusieurs processus de l'automate, puis l'ordonnancement des assignations concurrentes associées au même processus.

Construction du graphe de dépendances entre variables

Le graphe de dépendance est un graphe orienté. Les nœuds sont des variables intermédiaires, des ports, des entrées de registre ou des sorties de registre. Un arc orienté relie A vers B lorsque A est une opérande de l'expression d'assignation de B. Un tel arc signifie : "A fait partie du support[2]simple de B". Un nœud source du graphe est donc un port d'entrée ou une sortie de registre. Un nœud puits du graphe est un port de sortie ou une entrée de registre. Le graphe de dépendance est construit à partir de la liste d'assignations concurrentes décrivant le comportement global du composant matériel.

La figure 4.20 illustre le graphe de dépendance construit à partir de la liste d'assignations concurrentes de la figure 4.7 (page 76). L'entrée du registre $register1$ dépend de la sortie du registre $register1$ et de l'entrée $input1$. $output2$ dépend uniquement de $temp$.

Pour construire le graphe, il suffit de parcourir la liste d'assignations concurrentes pour identifier les dépendances entre variables. Nous créons un nœud pour toutes les variables intermédiaires et pour toutes les variables de type port ou registre. Pour chaque opérande présent dans le membre de droite de chaque assignation, nous créons un arc orienté entre le nœud représentant cette opérande et le nœud représentant la variable affectée.

[2]Voir la définition en annexe A.

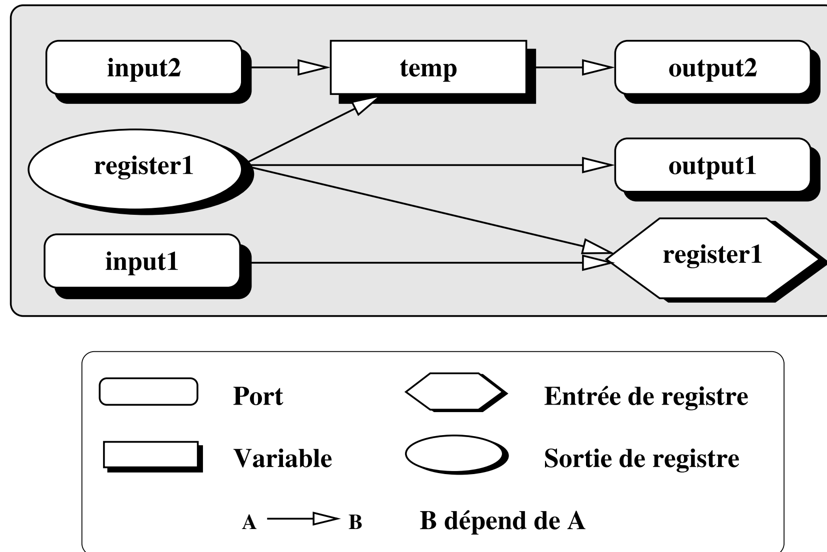

FIG. 4.20 – Graphe de dépendances entre variables

La fonction 4 présente une implémentation simple pour construire le graphe de dépendances entre variables.

Fonction 4 Construction du graphe de dépendances entre variables à partir d'une liste d'assignations concurrentes

---

Require: L_{asg} contient une liste d'assignations concurrentes.
  $G \Leftarrow \emptyset$
  for all expression d'assignation $asg_v \in L_{asg}$ **do**
    Ajouter, dans G, le nœud v
    for all variable $v' \in Supp(v)$ **do**
      Ajouter, dans G, le nœud v'
      Ajouter, dans G, un arc orienté de v' vers v
    end for
  end for
  return G

Pour simplifier le problème de l'étiquetage des nœuds et de l'ordonnancement des assignations concurrentes dans les étapes suivantes, le graphe de dépendance doit être acyclique. S'il existe un cycle dans le graphe de dépendances entre variables, nous stoppons l'analyse et nous avertissons l'utilisateur.

Étiquetage des nœuds du graphe de dépendances entre variables

L'étiquetage répond à deux objectifs :

- Identifier les dépendances combinatoires entre les ports d'entrée et les ports de sortie du composant matériel ;
- Construire $2+n$ listes d'assignations concurrentes distinctes correspondant aux fonctions de transition, de génération de Moore, et aux n fonctions de génération de Mealy.

Rappelons que pour faciliter l'ordonnancement statique de la simulation, il existe une fonction de génération de Mealy indépendante pour chaque signal de sortie de type Mealy.

L'étiquette associée à un nœud indique le rôle que joue la variable dans les trois types de fonctions définies ci-dessus. Une même variable peut recevoir plusieurs étiquettes. Nous souhaitons appliquer, à chaque variable v, les règles suivantes :

- Si v est un registre, alors v et toutes les variables de son support étendu $SuppExt(v)$ sont étiquetées *Transition* ;
- Si v est un port de sortie :
  - S'il existe un port d'entrée dans le support étendu $SuppExt(v)$, alors v et toutes les variables de son support étendu sont étiquetés *Génération de Mealy pour v* ;
  - Sinon, v et toutes les variables incluses dans $SuppExt(v)$ sont étiquetées *Génération de Moore* ;

Pour étiqueter tous les nœuds du graphe de dépendances entre signaux, nous devons d'abord déterminer l'étiquette associée à chaque puits. Nous propageons ensuite l'étiquette de chaque puits vers ses prédécesseurs. La propagation des étiquettes s'arrête aux sources.

La fonction 5 présente une implémentation de l'étiquetage des nœuds. La liste des puits est parcourue dans n'importe quel ordre. Pour chaque puits, nous analysons le support étendu et le type du puits pour déterminer l'étiquette. Cette étiquette est ensuite propagée récursivement du puits jusqu'aux sources.

Fonction 5 Étiquetage des nœuds du graphe de dépendances entre variables

---

Require: G représente le graphe de dépendances entre variables.
  for all Nœud puits v de G **do**
    if getType(v) = REGISTER **then**
      Étiquette récursivement v et tous ses prédécesseurs en tant que $TRANSITION$
    else if getType(v) = OUTPUT_PORT **then**
      if $\exists i, i \in SuppExt(v)$ et getType(i) = INPUT_PORT **then**
        Étiquette récursivement v et tous ses prédécesseurs en tant que $MEALY_v$
      else
        Étiquette récursivement v et tous ses prédécesseurs en tant que $MOORE$
      end if
    end if
  end for
  return G

La figure 4.21 illustre le graphe de dépendances entre variables étiquetées construit à partir du graphe de la figure 4.20. *register*1 est un registre donc l'entrée de *register*1 est étiquetée *Transition*. *output*1 dépend seulement de *register*1 qui est un registre donc *output*1 et la sortie de *register*1 sont étiquetés *Génération de Moore*. La sortie *output*2 dépend de *temp* qui dépend lui même de *input*2 et *register*1. Il existe un chemin de *input*2 à *output*2. *output*2 dépend donc d'au moins une entrée. *output*2, *temp*, et *input*2 sont alors étiquetés *Génération de Mealy*. La figure 4.8 illustre le classement de ces assignations dans les processus du modèle en CFSM.

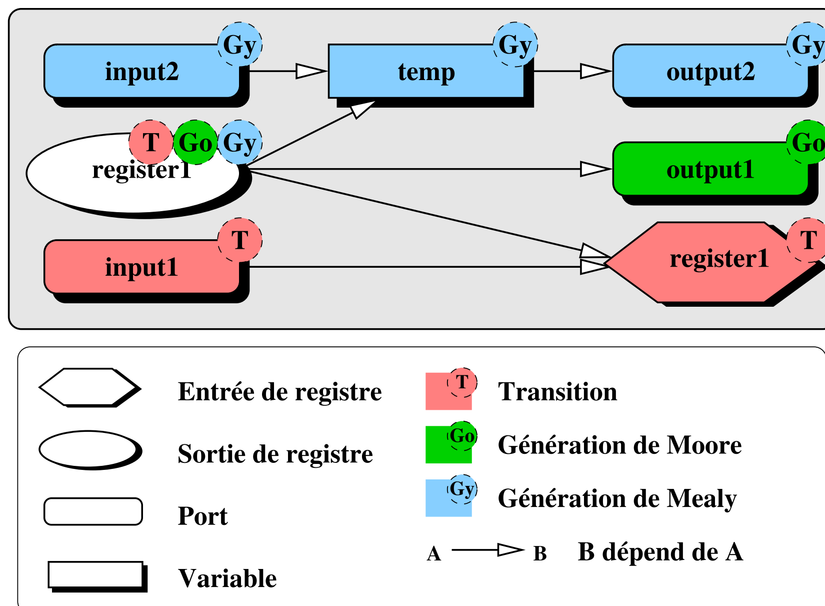

FIG. 4.21 – Graphe de dépendances entre variables étiquetées

Un processus s'exécutant de manière séquentielle, l'ordre d'exécution des assignations concurrentes est important. L'étape suivante permet d'ordonnancer les assignations concurrentes au sein d'un même processus.

4.4.2 Ordonnancement des assignations concurrentes

Nous effectuons un tri topologique du *Directed Acyclic Graph* (DAG) de dépendances entre variables. Ainsi, à chaque nœud est associé un numéro d'ordre. Il suffit alors de trier les $2+n$ listes d'assignations concurrentes, en fonction des numéros, pour obtenir $2+n$ listes ordonnées d'affectations uniques.

La figure 4.22 illustre le DAG de notre exemple après le tri topologique. Les numéros associés aux nœuds sont croissants des sources jusqu'aux puits. Le numéro du nœud *temp* est inférieur au numéro du nœud *output*2. La liste ordonnée d'affectations uniques du processus *output*2_*gen* est donc :

1. $temp := input2 + register1$
2. $output2 := temp\ and$ "1001"

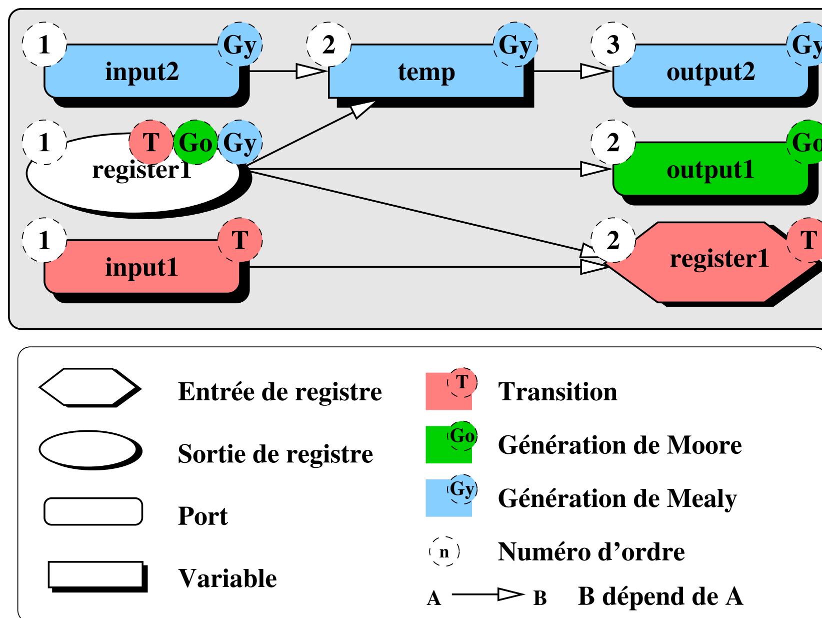

FIG. 4.22 – Graphe de dépendances entre variables étiquetées, après tri topologique

Le comportement de la description VHDL initiale est finalement représenté par un automate. Cet automate est décrit par $2+n$ processus. Chaque processus est associé à une liste ordonnée d'affectations uniques.

4.4.3 Génération d'un automate en SystemC

Un modèle en SystemC est représenté par une structure C++ héritant de la classe *sc_module*. Cette structure contient :

- Des variables de type *sc_in* et *sc_out* décrivant les ports du composant matériel ;
- Des variables de type *sc_signal* pour modéliser les registres,
- $2+n$ méthodes correspondant aux processus qui décrivent le comportement du matériel,
- Un constructeur permettant :
  - D'associer une liste de sensibilité à chaque processus ;
  - De décrire les dépendances entre les ports d'entrée et les ports de sortie au sein d'un même processus.

La représentation en SystemC d'un composant matériel généré par notre méthode est décrite dans les sections suivantes.

Définition de l'interface matériel au bit près

L'interface du composant est définie explicitement par le concepteur dans la description VHDL initiale. Chaque port de l'interface est caractérisé par une direction et un type de données. Les structures de données internes du compilateur conservent ces informations et permettent de générer une interface SystemC identique au bit près.

En SystemC, les ports d'entrée et les ports de sortie sont déclarés respectivement en tant que *sc_in* et *sc_out*. Le type de données VHDL est converti en type SystemC comme indiqué sur la table 4.2.

VHDL	**SystemC**
bit	bool
std_logic	bool
bit_vector	sc_uint<N>
std_logic_vector	sc_uint<N>
integer	sc_uint<N>

TAB. 4.2 – Conversion des types de données VHDL vers SystemC

Description du comportement au cycle près

Le modèle formel des automates communicants distingue trois types de fonctions : la fonction de transition, la fonction de génération de Moore, et les fonctions de génération de Mealy.

La fonction de transition est décrite par un processus unique contenant toutes les expressions d'affectation étiquetées $TRANSITION$. De même, la fonction de génération de Moore est décrite par un processus unique contenant toutes les affectations étiquetées $MOORE$.

Les fonctions de génération de Mealy sont décrites par $n \geq 0$ processus combinatoires. Un tel processus pilote un seul port de sortie étiqueté $MEALY$. Il y a alors autant de processus combinatoires générés que de ports de sortie étiquetés $MEALY$. Le comportement d'un composant matériel est donc décrit par $2+n$ processus.

La génération de modèle SystemC consiste à regrouper les expressions d'affectation dans les fonctions spécifiées par les étiquettes. Le générateur doit respecter l'ordre séquentiel d'évaluation des variables intermédiaires, des ports et des registres. Pour cela, il suffit de parcourir simplement les listes ordonnées d'affectations uniques.

Liste de sensibilité et dépendances de ports

À chaque processus est associée une liste de sensibilité. Nous supposons que les registres mémorisent les valeurs en entrée au front montant. Ainsi, un processus correspondant à la fonction de transition du modèle en automate est réveillé au front montant de l'horloge. Le processus correspondant à la fonction de génération de Moore est réveillé au front descendant de l'horloge

pour fournir la garantie que tous les registres aient été mis à jour avant de commencer le calcul des signaux de sortie.

Les processus correspondant aux fonctions de génération de Mealy impliquent des dépendances entre les signaux d'entrée et les signaux de sortie du module. L'analyse du graphe de dépendances permet de déterminer la liste de sensibilité et la liste des dépendances entre ports au sein d'un même processus. Dans le cadre de notre méthode, un processus combinatoire ne pilote qu'un seul port de sortie à la fois. Un tel processus possède seulement des dépendances entre l'unique port de sortie piloté O_{Mealy} et tous les ports d'entrée inclus dans le support étendu de O_{Mealy}.

La table 4.3 résume le contenu de la liste de sensibilité et les dépendances entre ports en fonction du type de processus.

Processus	**Liste de sensibilité**	**Dépendance entre ports**
Transition	Front montant de l'horloge	N/A
Génération de Moore	Front descendant de l'horloge	N/A
Génération du signal de Mealy O_{Mealy}	Front descendant de l'horloge et les ports d'entrée inclus dans le support de O_{Mealy}	Dépendances entre O_{Mealy} et les ports d'entrée inclus dans le support de O_{Mealy}

TAB. 4.3 – Listes de sensibilité et dépendances entre ports en fonction du type de processus

Exemple de modèle généré

Nous reprenons l'exemple de la description VHDL fournie par le listing **4.1** présenté au début du chapitre. Après analyse puis génération par notre méthode, nous obtenons une description SystemC dont le comportement est équivalent à la description initiale.

Le listing **4.2** contient la liste des ports, des registres, et des processus, ainsi qu'un constructeur.

Le listing **4.3** fournit une implémentation du processus SystemC correspondant à la fonction de transition de l'automate. La nouvelle valeur du registre $register1$ est calculée bit par bit.

Le listing **4.4** présente le processus décrivant la génération de Moore du composant matériel. Celui-ci calcule la sortie $output1$ en fonction de $register1$.

Le listing **4.5** illustre la description SystemC d'une fonction de génération de Mealy. Le processus $output2_gen$ permet de calculer la sortie $output2$ en fonction de $register1$ et de $input2$.

Le constructeur, présenté dans le listing **4.6**, déclare les processus décrivant le comportement du matériel avec leur liste de sensibilité associée. Les dépendances entre les ports d'entrée et les ports de sortie sont déclarées de façon optionnelle : le port de sortie $output2$ affecté par le processus $output2_gen$ dépend du port d'entrée $input2$. Ces déclarations permettent de simuler avec SystemCASS.

```
1  // generated by xbn_drive
2  // Jun  8 2006 12:00:58
3  #include <systemc.h>
4  struct exemple : sc_module {
5  // PORT(s) list
6  // INPUT PORTS
7  sc_in<sc_uint<4> > input2;
8  sc_in<sc_uint<4> > input1;
9  sc_in_clk ck;
10 // OUTPUT PORTS
11 sc_out<sc_uint<4> > output2;
12 sc_out<sc_uint<4> > output1;
13 // REGISTERS
14 sc_signal<sc_uint<4> > register1;
15 // Forward Declarations
16 SC_HAS_PROCESS(exemple);
17 exemple (sc_module_name);
18 void transition ();
19 void moore_generation ();
20 void output2_gen ();
21 };
```

Listings 4.2 – Interface SystemC générée respectant la modélisation en automate

```
1  void
2  exemple::transition ()
3  {
4  register1.write (((sc_uint<4>)register1.read().range (3,0) /* WIDTH = 4 */
5    - (sc_uint<4>)input1.read().range (3,0) /* WIDTH = 4 */));
6  }
```

Listings 4.3 – Fonction de transition générée en SystemC

```
1  void
2  exemple::moore_generation ()
3  {
4  output_port1.write (register1.read().range (3,0) /* WIDTH = 4 */);
5  }
```

Listings 4.4 – Fonction de génération de Moore générée en SystemC

```
1  void
2  exemple::output2_gen ()
3  {
4  sc_uint<4> temp = (((sc_uint<4>)input2.read().range (3,0) /* WIDTH = 4 */
5              + (sc_uint<4>)register1.read().range (3,0) /* WIDTH = 4 */));
6  output_port2.write (((sc_uint<4>)temp.read().range (3,0) /* WIDTH = 4 */
7              & (sc_uint<4>)(9)));
8  }
```

Listings 4.5 – Fonction de génération de Mealy générée en SystemC

```
1  exemple::exemple (sc_module_name)
2    : output2("output2"),output1("output1"),input2("input2"),
3      input1("input1"),ck("ck"),register1("register1")
4  {
5    SC_METHOD (transition);
6     sensitive << ck.pos();
7    SC_METHOD (moore_generation);
8     sensitive << ck.neg();
9    SC_METHOD (output2_gen);
10    sensitive << ck.neg() << input2;
11 #if defined(PORT_DEPENDANCIES_ENABLED)
12    output2 (input2);
13 #endif
14 }
```

Listings 4.6 – Constructeur généré en SystemC

Propriétés du modèle SystemC

Le modèle généré respecte la modélisation en automates synchrones communicants.

Les signaux de Mealy de l'automate sont pilotés par des processus combinatoires élémentaires. Les dépendances entre les ports d'entrée et l'unique port de sortie d'un tel processus sont déclarées explicitement dans le constructeur du module. Ces informations permettent d'utiliser le simulateur à ordonnancement totalement statique présenté au chapitre 3.

L'échéancier à ordonnancement totalement statique raisonne sur le graphe de dépendances entre signaux. Si ce graphe possède un cycle, l'échéancier refuse de produire un ordonnancement. Sachant que ce graphe est sans cycle, et que chaque processus combinatoire ne pilote qu'un seul signal de sortie, l'ordonnancement devient trivial. L'ordre d'exécution des processus est alors optimal, sans évaluation redondante des signaux.

4.5 Résultats expérimentaux

Le générateur de modèle SystemC, développé dans le cadre de cette thèse, s'appelle RTL2CFSM. Celui-ci a permis de générer, avec succès, le séquenceur du coprocesseur Hadamard[AJFA00] mais l'outil a également essuyé quelques échecs lors de la conversion de descriptions comportant des tableaux.

Dans les sections suivantes, nous présentons le coprocesseur Hadamard, puis nous critiquons les résultats obtenus. Enfin, nous expliciterons les problèmes rencontrés avec les tableaux VHDL.

4.5.1 Architecture du coprocesseur Hadamard

Le coprocesseur Hadamard est une implémentation matérielle de la transformée d'Hadamard utilisée pour la compression d'image. L'algorithme de compression calcule la matrice $H*P*H$, où H est la matrice d'Hadamard, et P est la matrice de l'image à compresser. Le produit de matrices est obtenu par une succession d'additions et de soustractions.

La figure 4.23 illustre l'architecture du coprocesseur Hadamard. L'architecture comporte six composants élémentaires, représentés par des rectangles. Un arc orienté de A vers B traduit une dépendance de donnée de A en fonction de B. Le coprocesseur est décrit par les composants suivants :

- Un bloc de *RAM* pour stocker l'image à compresser ;
- Un bloc *MAT* pour conserver les valeurs de la matrice H ;
- Un bloc de calcul arithmétique 8 bits ;
- Un bloc $registerfile$ pour stocker les résultats de calcul intermédiaires ;
- Un bloc $Counters$ gère trois index nécessaires lors des opérations sur les matrices ;
- Un bloc *Seq* commande les autres blocs. C'est le séquenceur du coprocesseur.

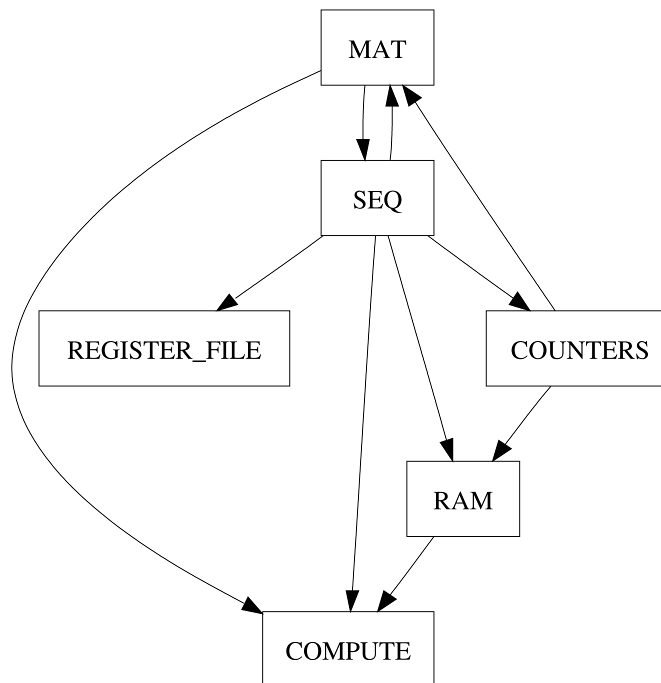

FIG. 4.23 – Hadamard : Architecture interne du coprocesseur

Expérimentations

Nous disposons des descriptions synthétisables VHDL du coprocesseur Hadamard sous la forme de processus synchrones à un seul point de suspension. Nous avons effectué des conversions du bloc *Seq* uniquement.

Le bloc *Seq* est décrit par deux processus. L'un calcule les valeurs futures des registres, ainsi que les valeurs sur les ports de sortie. Ce processus est réveillé à chaque fois qu'un évènement apparaît sur l'un des ports d'entrée ou sur l'un des registres. L'autre processus se contente de mettre à jour les valeurs des registres au front montant de l'horloge.

L'outil de conversion RTL2CFSM a permis de convertir le bloc *Seq*. Le comportement du modèle généré est ensuite vérifié par simulation en comparant avec les traces de simulation de la description VHDL initiale. La liste de dépendances entre les ports d'entrée et les ports de sortie a été construite puis vérifiée. Cette liste permet alors d'utiliser le simulateur à ordonnancement totalement statique décrit dans le chapitre 3.

Résultats de la conversion et Bilan

Le coprocesseur est un exemple académique. Le bloc *Seq* a une taille nettement inférieure à un exemple de taille réelle. La conversion manuelle nécessite seulement une ou deux demi

journées. La conversion automatique prend moins de deux secondes.

La conversion à l'aide d'un outil de génération est très avantageuse en temps de développement. Le concepteur a ensuite la possibilité d'utiliser les modèles générés dans le cadre de l'exploration architecturale. Ces modèles sont compatibles avec le simulateur SystemCASS et permettent d'obtenir de bonnes performances à l'aide de l'ordonnancement totalement statique. Ces modèles ont cependant deux inconvénients : la lisibilité et la vitesse de simulation de la description.

Lors de l'analyse sémantique, les variables existantes sont renommées et des variables intermédiaires sont ajoutées. Les noms de variables deviennent peu lisibles. De plus, les optimisations booléennes rendent les opérations bit à bit complexes et difficiles à lire par le concepteur.

Lors de la génération, RTL2CFSM décrit les opérations arithmétiques et logiques à l'aide des opérateurs redéfinis par SystemC. Ces opérateurs sont parfois plus lents que les opérateurs C/C++. Malheureusement, le générateur automatique n'est pas capable de déterminer lorsqu'un opérateur C/C++ est plus avantageux qu'un opérateur SystemC. Lors d'une conversion manuelle, le concepteur est capable de vectoriser les opérations et de choisir les opérateurs les plus rapides. Une description écrite manuellement par un concepteur est donc plus rapide, en simulation, que la description générée par RTL2CFSM.

4.5.2 Limitation : Descriptions VHDL de composants matériels utilisant des tableaux

Beaucoup de composants matériels utilisent des FIFO, de la RAM, et/ou des bancs de registres. La description VHDL de ces éléments matériels comporte généralement des tableaux.

Rappelons que l'analyse sémantique de VASY travaille, bit par bit, sur l'affectation de chaque élément des tableaux. Or, VASY ne traite pas les tableaux, car cela consommerait beaucoup trop de mémoire pour construire la liste d'assignations concurrentes représentant le comportement global du matériel. Les dimensions d'un tableau posent donc un véritable problème pour l'analyse sémantique.

La lecture indexée dans un tableau est traduite en expressions conditionnelles. Une telle opération nécessite autant d'expressions conditionnelles imbriquées que d'éléments dans le tableau. L'expression $dout := mem(idx)$ devient ainsi :

$$dout := (idx == 0)?(mem_0) : (...((idx == i)?(mem_i) : (...(mem_n))))$$

Une telle expression est trop gourmande en mémoire. L'analyse sémantique est alors impossible.

La description d'un composant de taille réelle comporte souvent des tableaux. Or, l'analyse sémantique décrite dans ce chapitre n'est pas capable de traiter correctement ces tableaux. Les perspectives de recherche pour la génération de modèle en CFSM cherchent à résoudre ce problème.

Perspectives

La solution envisagée est de proposer au concepteur d'instancier des composants prédéfinis, tels que des FIFO, des blocs de RAM, et des bancs de registres. Les composants prédéfinis constitueraient une bibliothèque pour faciliter l'analyse sémantique. Ceci revient en pratique à interdire l'utilisation des tableaux dans la description comportementale.

VASY n'effectuerait pas les étapes de construction et de réduction de graphes de contrôle des composants de la bibliothèque. Chacun de ces composants serait associé à une liste d'assignations concurrentes prédéfinie. Ces assignations concurrentes seraient alors insérées dans la liste d'assignations concurrentes décrivant le comportement global du matériel.

Les difficultés de l'analyse sémantique face aux tableaux seraient alors contournées.

4.6 Conclusions

La méthode proposée dans ce chapitre permet de générer automatiquement un modèle de simulation SystemC à partir d'une description synthétisable RTL. Le modèle de simulation est enrichi par la liste de dépendances entre ports afin de profiter de l'accélération du simulateur à ordonnancement totalement statique présenté au chapitre 4.

Le principe de la méthode repose sur une analyse sémantique de la description VHDL initiale pour construire un automate dont le comportement est équivalent. L'automate est représenté par des affectations uniques. La génération de la description SystemC revient à générer ces affectations en expressions C/C++.

Les résultats expérimentaux portent sur une architecture simple de coprocesseur et ont permis de valider l'approche. Le travail de réécriture peut être entièrement automatique. La conversion automatique comporte des avantages importants notamment en terme de temps de travail. Les modèles générés ont cependant deux inconvénients : la lisibilité et la vitesse de simulation de la description.

En fin de compte, un modèle généré par notre méthode est extrêmement intéressant à court terme pour réaliser une exploration architecturale. Cependant, un tel modèle se révèle difficile à exploiter par la suite. Un travail de réécriture manuel du modèle doit alors être envisagé.

De plus, la méthode de génération présentée comporte une limitation importante : les tableaux ne sont pas traités correctement par l'analyse sémantique. L'une des perspectives d'amélioration, de la génération de modèle SystemC en automate, est de résoudre ce problème.

Chapitre 5

Vérification sémantique des modèles à automates synchrones communicants

Sommaire

5.1 Approche dynamique 104
5.1.1 Vérification à l'exécution 104
5.1.2 Problèmes non résolus 106
5.2 Principe de la vérification sémantique 107
5.3 Construction de l'arbre de syntaxe abstraite 109
5.4 Construction du support des variables 110
5.4.1 Opérateurs d'affectation 111
5.4.2 Opérations arithmétiques et logiques 112
5.4.3 Instructions conditionnelles 113
5.4.4 Instructions de boucle 114
5.4.5 Appels de fonction ou de méthode 115
5.5 Identification des constantes architecturales 116
5.6 Vérification des contraintes de modélisation 116
5.7 Mise en œuvre 116
5.8 Résultats expérimentaux 117
5.8.1 Modèles de composants matériels vérifiés 118
5.8.2 Bilan des résultats 121
5.9 Conclusions 121

L'approche présentée dans cette thèse consiste à modéliser les composants matériels élémentaires sous la forme d'automates synchrones communicants puis à simuler l'architecture à l'aide d'un moteur de simulation optimisé pour ce type de modélisation. Ce moteur de simulation repose sur le fait que tous les modèles de composants matériels respectent le modèle formel des automates communicants pour construire un ordonnancement optimal, totalement statique. L'ordonnancement statique accélère la vitesse de simulation de manière importante pour gagner un ordre de grandeur par rapport au moteur de simulation standard d'OSCI.

L'objectif de ce chapitre est de présenter une méthode de vérification capable de déterminer si un modèle de composant matériel respecte la modélisation en CFSM. La vérification d'un modèle repose sur une analyse avant simulation afin de conserver des vitesses de simulation élevées.

Ce chapitre présente deux méthodes de vérification des modèles SystemC écrits sous la forme des CFSM. Deux outils distincts ont été développés. L'un est un outil de vérification à l'exécution. L'autre est un outil de vérification sémantique. Les résultats expérimentaux sont ensuite décrits à la fin du chapitre.

5.1 Approche dynamique

L'objectif de ce chapitre est de déterminer si la description SystemC d'un composant matériel respecte bien le modèle formel des automates synchrones communicants présenté au chapitre 2.

À notre connaissance, il n'existe pas de travaux permettant d'atteindre cet objectif. Nous commençons par présenter une méthode "naïve" de vérification à l'exécution qui a le mérite de préciser formellement les contraintes qui doivent être respectées. Nous exposons ensuite les problèmes non résolus par cette méthode.

5.1.1 Vérification à l'exécution

La vérification se déroule au cours de l'exécution normale de la simulation. La description SystemC du comportement d'un matériel est exécutée de manière séquentielle. A chaque lecture ou écriture dans un port ou un registre, nous souhaitons vérifier si l'opération respecte le modèle formel des automates synchrones communicants.

Les ports et les registres d'un module sont identifiés explicitement par les types SystemC : *sc_in*, *sc_out*, et *sc_signal*. Les opérations d'évaluation et d'affectation des ports, et des registres sont respectivement redéfinies par les méthodes $read()$ et $write()$ de l'API SystemC. C'est à dire qu'à chaque fois qu'un port ou un registre est affecté, à l'aide de l'opérateur $=$, la méthode $write()$ est automatiquement appelée. De même, à chaque fois qu'un port ou un registre est évalué, la méthode $read()$ est appelée.

La méthode de vérification consiste à insérer des vérifications dans les méthodes $read()$ et $write()$. Cela nécessite donc des modifications dans le simulateur. La vérification à l'exécution

permet alors de vérifier, à chaque appel de méthode $read()$ et $write()$, si l'opération respecte le modèle en CFSM ou pas.

La méthode nécessite également un indicateur pour déterminer le type du processus en cours d'exécution. Le simulateur possède donc une variable $CURRENT_PROCESS_TYPE$. Cette variable a quatre valeurs possibles :

- $INITIALISATION$
- $TRANSITION$
- $GENERATION_MOORE$
- $GENERATION_MEALY$

.

À l'initialisation, le simulateur positionne $CURRENT_PROCESS_TYPE$ à la valeur $INITIALISATION$, puis il exécute le constructeur de chaque module. Ensuite, à chaque cycle, le simulateur fixe $CURRENT_PROCESS_TYPE$ à la valeur $TRANSITION$ et appelle toutes les fonctions de transition décrivant l'architecture matérielle. Puis, le simulateur fixe l'indicateur à la valeur $GENERATION_MOORE$ et ainsi de suite.

La table 5.1 présente les contraintes qui doivent être vérifiées à l'exécution. *OK* indique que l'exécution de la méthode est autorisée car celle-ci respecte le modèle des automates. Lorsque la méthode $sc_signal < T >:: read()$ est appelée, le type du processus courant (eq. $CURRENT_PROCESS_TYPE$) est testé. Si le type est $TRANSITION$, $GENERATION_MOORE$, ou $GENERATION_MEALY$, l'exécution continue. Sinon, l'exécution s'arrête et un message d'erreur permet au concepteur d'identifier la ligne fautive.

Type SystemC	**Méthode**	*INIT.*	*TRANSITION*	*GEN.MOORE*	*GEN.MEALY*
sc_in	write ()	-	-	-	-
	read ()	-	OK	-	OK
sc_out	write ()	-	-	OK	OK
	read ()	-	-	-	-
sc_signal	write ()	-	OK	-	-
	read ()	-	OK	OK	OK

TAB. 5.1 – Contraintes vérifiées à l'exécution

Il est impossible de redéfinir les opérateurs d'évaluation et d'affectation associées aux variables de type standard C/C++. En effet, les opérateurs, associés aux types int, $double$, ou encore $bool$, ne peuvent être redéfinis. Le concepteur a donc la possibilité de déclarer et d'utiliser des variables échappant aux vérifications à l'exécution.

Le listing **5.1** illustre une description SystemC qui ne respecte pas la modélisation en automates synchrones communicants. Le composant $exemple$ possède un port de sortie $output1$ et une variable membre reg de type int. La fonction de génération de Moore effectue deux opérations. La première est $reg++$. Le type de la variable reg est un type standard C/C++. L'affectation de cette variable ne peut pas être vérifiée à l'exécution. Malheureusement, cette af-

fectation ne respecte pas les contraintes de modélisation des fonctions de génération. La seconde opération affecte *reg* à la sortie *output*1. Une telle opération est autorisée. Cependant, la valeur affectée ne respecte pas les contraintes sur les dépendances de données imposées par le modèle en CFSM. La vérification à l'exécution n'est pas capable de détecter cette seconde erreur de modélisation.

```
1  struct exemple : sc_module {
2    sc_in <bool> clock;
3    sc_out<int>  output1;
4    int reg;
5    exemple::moore_generation ()
6    {
7      reg++; // Affectation non vérifiable
8      output1 = reg; // Problème non detecté
9    }
10   ...
11 };
```

Listings 5.1 – Description SystemC ne respectant pas la modélisation en automates

Limitations de l'approche par simulation

Pour mener à bien nos expérimentations, nous avons modifié le simulateur SystemCASS à ordonnancement totalement statique, présenté au chapitre 3. La méthode de vérification à l'exécution est simple et rapide à mettre en œuvre. Cependant, celle-ci comporte trois inconvénients importants :

- Les opérations sur des variables de type standard ne sont pas vérifiables ;
- Comme pour toute méthode reposant sur la simulation, le chemin d'exécution dépend du jeu de données. Par conséquent, la vérification de tous les cas possibles d'exécution n'est donc pas garantie.
- Les vérifications à chaque appel de méthode *read*()/*write*() réduisent considérablement la vitesse de simulation.

5.1.2 Problèmes non résolus

La vérification à l'exécution ralentit les simulations de manière trop importante. De plus, les résultats ne sont pas satisfaisants.

Les problèmes non résolus par la vérification à l'exécution sont :

- Comment vérifier le comportement des variables de type standard dans une description SystemC ?
- Comment vérifier tous les cas d'exécution possibles ?

- Comment identifier, formellement, les opérations qui ne respectent pas le modèle des automates synchrones communicants ?

Les sections suivantes présentent une méthode de vérification sémantique à l'aide d'une analyse statique à la compilation. Celle-ci tente de répondre aux problèmes énumérés ci-dessus. Nous décrivons ensuite les résultats obtenus, puis nous terminons par la conclusion de ce chapitre.

5.2 Principe de la vérification sémantique de modèle à automates

La vérification d'un modèle à automates consiste à analyser sa description pour indiquer s'il respecte, ou non, le modèle formel des automates synchrones communicants. Une telle description est caractérisée par un ensemble de ports, de registres, de constantes architecturales et de processus décrivant le comportement du ou des automates.

La méthode de vérification sémantique s'appuie sur une analyse des dépendances entre variables dans un même processus. Nous distinguons cinq types de variable :

- Les ports d'entrée ;
- Les ports de sortie ;
- Les registres ;
- Les variables intermédiaires ;
- Les constantes architecturales.

Le concepteur identifie explicitement les ports, et les registres, à l'aide des types prédéfinis par SystemC. Par contre, les variables intermédiaires et les constantes architecturales doivent être identifiées automatiquement au cours de l'analyse. Si le type d'une variable n'est pas l'un des cinq types cités précédemment, la description SystemC n'est pas vérifiable.

Les règles d'écriture que nous souhaitons vérifier sont les suivantes :

- Les dépendances entre variables de type port et registre respectent le modèle formel des automates synchrones communicants, présenté au chapitre 2 ;
- Une même variable est pilotée par un seul processus ;
- Les constantes architecturales sont des variables membres. Leurs valeurs sont soit connues à la compilation, soit initialisées par le constructeur ;
- Les variables intermédiaires sont déclarées en tant que variables locales aux processus.

Si l'une de ces règles n'est pas vérifiée, la description analysée ne respecte pas le modèle formel des automates synchrones communicants. Les expressions ne respectant pas les contraintes sont détectées durant l'analyse. L'outil de vérification localise alors l'expression fautive et affiche la ligne mise en cause pour aider le concepteur à corriger les erreurs de modélisation.

Pour analyser les dépendances entre variables, nous raisonnons sur le support étendu[1] des variables. L'analyse est statique et elle se déroule à la compilation en quatre étapes successives :

1. Construction de l'arbre de syntaxe abstraite représentant la description SystemC du composant matériel à vérifier ;

[1] La définition du support étendu est fournie en annexe A.

2. Construction du support étendu de chaque variable ;

3. Identification des constantes architecturales, et des variables intermédiaires ;

4. Vérification de l'application des contraintes de modélisation.

La figure 5.1 illustre les différentes étapes de la méthode. Nous détaillons ces étapes dans les sections suivantes.

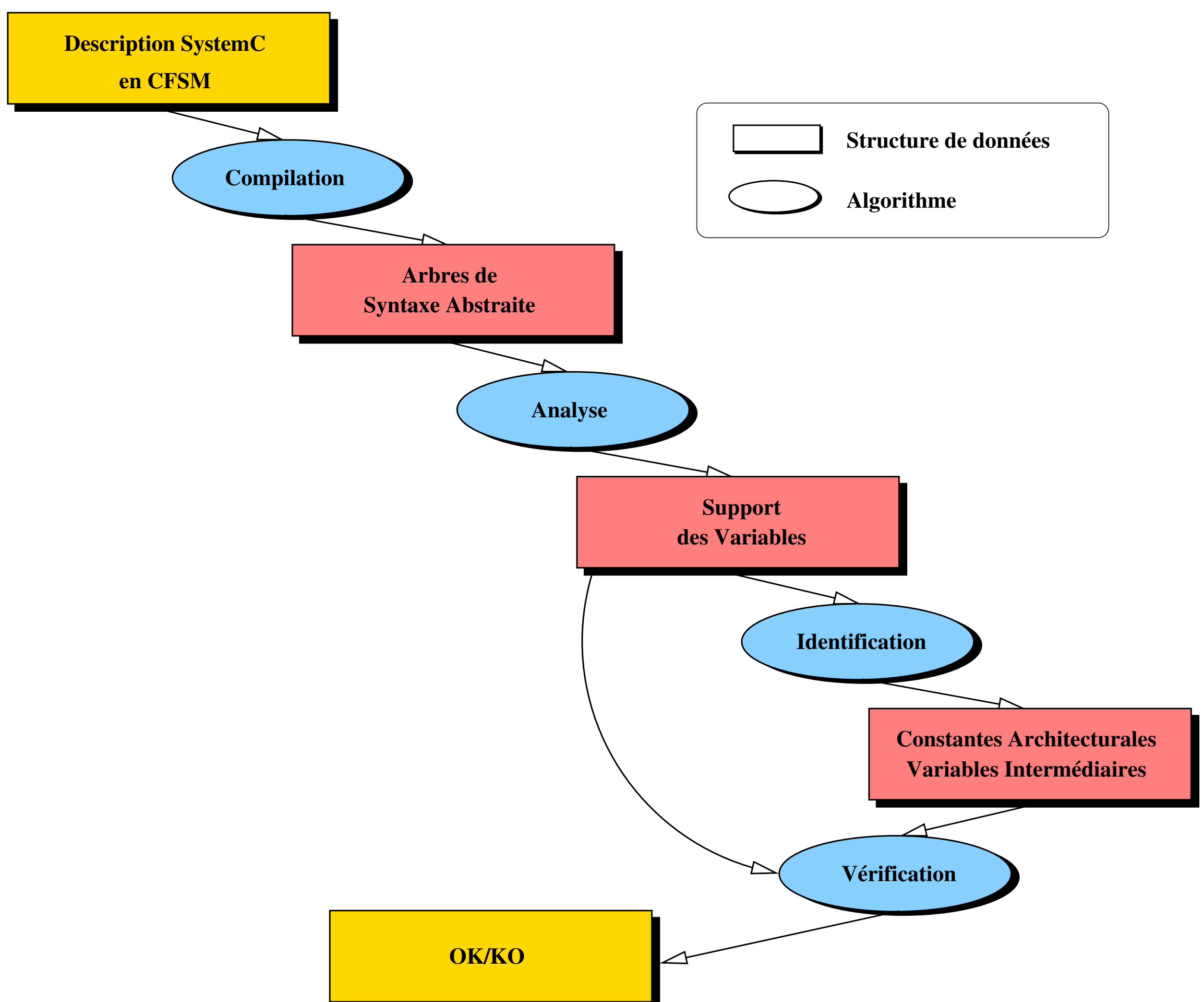

FIG. 5.1 – Méthode pour la vérification statique par analyse sémantique

5.3 Construction de l'arbre de syntaxe abstraite

La description SystemC d'un composant matériel est composée :

- De processus décrivant le comportement du matériel ;
- D'un constructeur qui initialise les constantes architecturales et qui identifie les processus décrivant le comportement.

Cette description est analysée pour construire un *Arbre de Syntaxe Abstraite* (AST) pour le constructeur, et pour chaque processus. Cette étape est effectuée par un compilateur tel que *gcc*.

Le corps d'une fonction C/C++ est représenté par un AST. La racine de l'arbre est le point d'entrée de la fonction. Un nœud non feuille représente un opérateur. Une feuille représente une variable ou une constante. Chaque nœud est associé à une étiquette indiquant le numéro de ligne du fichier source. Un arc $A \rightarrow B$ signifie que B est une opérande de l'opérateur A.

Les opérateurs les plus courants sont :

- Les opérateurs arithmétiques, tels que $+-/*$, et *mod* ;
- Les opérateurs logiques binaires, tels que *and*, *or*, *xor*, $<<$ et $>>$;
- L'opérateur point virgule (;) ; de composition d'instructions ;
- Les opérateurs de boucle, tels que *while* et for ;
- Les instructions conditionnelles, tels que if et *switch/case*.

La figure 5.2 illustre l'arbre de syntaxe abstraite représentant l'expression suivante : $reg_A = reg_A + (port_I * 2)$. Le nœud racine représente l'opérateur $=$, redéfini par l'API SystemC, pour l'écriture dans les registres. La racine possède deux fils. Le fils de gauche est une feuille. Cette feuille correspond à la variable affectée, reg_A, qui est de type registre. Le fils de droite décrit l'expression arithmétique $reg_A + (port_I * 2)$ qui sera affectée dans reg_A.

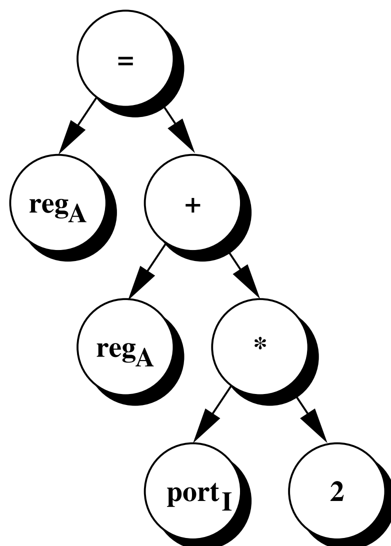

FIG. 5.2 – Arbre de syntaxe abstraite représentant une affectation en SystemC

Après construction de l'arbre de syntaxe abstraite, par le compilateur, nous pouvons construire le support des variables.

5.4 Construction du support des variables

Pour construire le support des variables utilisées dans la description d'un composant matériel, nous analysons, séparément, l'AST du constructeur, ainsi que l'AST de chaque processus.

L'ordre d'exécution des instructions dans une fonction C/C++ est modifié par les instructions conditionnelles et les instructions de boucle, or le résultat de l'exécution de ces instructions dépend des données. L'ordre d'exécution des instructions lors de la simulation dépend donc du jeu de données. Le support des variables, que nous souhaitons construire, doit être indépendant du jeu de données.

Le nombre de chemins d'exécution possibles peut être important. Ainsi, il n'est pas raisonnable d'énumérer tous les chemins pour construire le support des variables. L'approche présentée propose de construire le support des variables de manière pessimiste.

L'analyse d'un AST commence par la racine de l'arbre qui représente l'entrée de la fonction C/C++. Le parcours de l'arbre doit respecter ensuite un ordre, indépendant du jeu de données. Nous détaillons par la suite comment parcourir l'arbre.

Avant de parcourir l'AST d'un processus ou d'un constructeur, nous nous intéressons à l'état initial du support étendu des variables. Si une variable n'est pas modifiée par le processus, la variable conserve sa valeur. L'état initial du support étendu d'une variable $SuppExt(v)$ est donc $\{v\}$.

L'expression résultant d'une opération dépend, d'une part, des opérandes, et d'autre part, des conditions d'exécution de l'opération. Le support étendu d'une expression inclut donc les opérandes de l'expression, ainsi que les opérandes intervenant dans les conditions de son exécution. Lors du parcours de l'arbre, nous devons conserver le support de la condition d'exécution de chaque bloc d'instruction. Le support de la condition d'exécution d'un bloc est initialisé à l'ensemble vide et celui-ci est ensuite mis à jour au fur et à mesure du parcours de l'arbre.

État initial :
$\forall v, SuppExt(v) = \{v\}$
$SuppExt_{bloc} = \emptyset$

La figure **5.2** présente une description SystemC comportant un automate à états synchrone. Le registre fsm_state conserve l'état de la machine à états. Le reset synchrone (ligne 3 et 4) est décrit à l'aide d'une instruction conditionnelle. La condition dépend de la variable *resetn*. Quel que soit le chemin d'exécution emprunté, nous savons que l'initialisation du registre fsm_state dépend du résultat de l'expression conditionnelle. Le registre fsm_state dépend du support de la condition d'exécution du bloc $bloc1$ correspondant au reset synchrone. Cela se traduit par $resetn \in SuppExt(bloc1)$.

Dans notre exemple, la transition de l'automate est décrite à l'aide d'une instruction *switch/case* (ligne 6 à 15). Le comportement de l'automate doit être analysé en étudiant chacune des transitions d'état. Les sections suivantes expliquent comment calculer le support étendu de chaque variable et le support de chaque bloc d'instructions. A l'issue de l'analyse de notre exemple, le support étendu de fsm_state est :
$SuppExt(fsm_state) = SuppExt(bloc1) \cup SuppExt(bloc2) \cup SuppExt(bloc3) \cup SuppExt(bloc4)$
$SuppExt(fsm_state) = \{resetn, fsm_state, eop\}$

```
1  exemple::transition ()
2  {
3    if (resetn)
4      fsm_state = IDLE; // SuppExt(bloc1) = {resetn}
5    else
6      switch (fsm_state) {
7      case IDLE :
8        fsm_state = WAIT; // SuppExt(bloc2) = {resetn, fsm_state}
9        break;
10     case READ :
11       buffer = val; // SuppExt(bloc3) = {resetn, fsm_state}
12       if (eop)
13         fsm_state = IDLE; // SuppExt(bloc4) = {resetn, fsm_state, eop}
14     ...
15     }
16   ...
17 }
```

Listings 5.2 – Description SystemC illustrant le support d'exécution

Lors du parcours, nous mettons à jour le support de chaque variable et le support d'exécution de chaque bloc d'instructions. Les instructions peuvent être classées dans cinq catégories :

- Les opérateurs d'affectation ;
- Les opérateurs arithmétiques et logiques ;
- Les instructions conditionnelles ;
- Les instructions de boucle ;
- Les appels de fonction.

5.4.1 Opérateurs d'affectation

Une instruction d'affectation est un opérateur binaire de la forme $A = B$. La figure 5.3 illustre la représentation d'une affectation.

L'exécution d'une telle opération écrase le support de A par l'union du support de B et du support de la condition d'exécution de l'affectation :

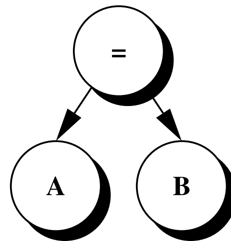

FIG. 5.3 – Arbre de syntaxe abstraite représentant une affectation

$$SuppExt(Affectation) = SuppExt(A) = SuppExt(B) \cup SuppExt_{bloc}$$

Les descriptions SystemC comportent généralement des tableaux (en particulier pour représenter les mémoires). Les dimensions d'un tableau posent un véritable problème pour l'analyse des supports. Pour éviter ce problème, nous considérons un seul support pour l'ensemble des éléments consistuant un même tableau. Le support du i-ème élément d'un tableau T est l'union du support de l'index i et du support de l'ensemble des éléments qui constituent le tableau :

$$SuppExt(T[i]) = SuppExt(T) \cup SuppExt(i).$$

L'instruction d'affectation dans un des éléments constituant un tableau est de la forme $T[i] = B$. L'exécution de cette affectation ajoute le support de i et le support de B dans celui de T :

$$SuppExt(T) = SuppExt(T) \cup SuppExt(i) \cup SuppExt(B)$$

5.4.2 Opérations arithmétiques et logiques

Les opérations binaires arithmétiques et logiques sont représentées par un nœud père et deux fils, comme indiqué sur la figure 5.4.

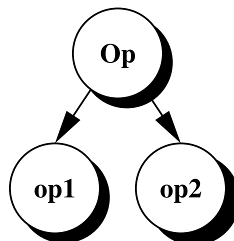

FIG. 5.4 – Arbre de syntaxe abstraite représentant une opération binaire, arithmétique ou logique

Le support de l'expression $op1(Op)op2$ est l'union des trois supports : le support de la première opérande, le support de la seconde opérande, et le support de la condition d'exécution de l'opération :

$$SuppExt(Resultat) = SuppExt(Operande1) \cup SuppExt(Operande2) \cup SuppExt_{bloc}$$

5.4.3 Instructions conditionnelles

Les instructions conditionnelles sont les instructions C/C++ $if/else$ et $switch/case$. Ces instructions créent des chemins d'exécution qui sont empruntés en fonction du résultat d'expressions conditionnelles.

La figure 5.5 illustre l'arbre de syntaxe abstraite d'une instruction $if/else$. Cette instruction créé un ou deux chemins d'exécution supplémentaire(s). Le nœud père if possède trois nœuds fils :

Condition Ce nœud représente l'expression conditionnelle.

Then Ce nœud décrit un bloc d'instructions qui est exécuté lorsque le résultat de l'expression conditionnelle est $true$.

Else Le nœud *else* spécifie un second bloc d'instructions. La spécification du second bloc d'instructions est facultative. Si le résultat de l'expression conditionnelle est $false$, ce second bloc d'instructions est exécuté.

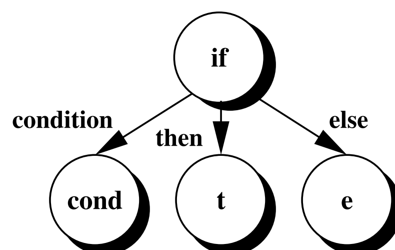

FIG. 5.5 – Arbre de syntaxe abstraite représentant une instruction conditionnelle

L'exécution d'un bloc d'instructions dépend de l'expression conditionnelle. Le résultat de cette expression conditionnelle n'est pas connu à la compilation. En effet, le résultat n'est connu qu'à l'exécution. Nous souhaitons construire le support des variables résultant du branchement et de l'exécution de l'un de ses blocs. Pour cela, il suffit de construire le support de chaque variable pour tous les blocs d'instructions exécutables, puis de fusionner tous ces supports.

Le support des expressions évaluées dans un bloc dépend du résultat de l'expression conditionnelle. Nous calculons donc le nouveau support étendu des conditions d'exécution $SuppExt'_{bloc}$, puis nous parcourons chacun des blocs possibles en considérant ce nouveau support d'exécution. Le support de chaque variable résultant de l'exécution d'une instruction conditionnelle et de l'un de ses blocs est l'union des supports résultant de l'exécution de chaque bloc.

$SuppExt'_{bloc} = SuppExt_{bloc} \cup SuppExt_{condition}$
$\forall v, SuppExt(v) = SuppExt^{then}(v) \cup SuppExt^{else}(v)$ avec
$SuppExt^{then}(v)$, support étendu de v après l'exécution du bloc *then* ;
$SuppExt^{else}(v)$, support étendu de v après l'exécution du bloc *else*.

Les instructions $switch/case$ sont similaires aux instructions $if/then/else$. Les instructions $switch/case$ créent $n > 0$ bloc(s) d'instructions.

5.4.4 Instructions de boucle

Les instructions de boucle sont représentées par les instructions *while* et *for*. La figure 5.6 illustre l'arbre de syntaxe abstraite d'une boucle *while*. Le nœud père *while* possède seulement deux fils : l'un décrit le corps de boucle ; l'autre représente la condition d'exécution du corps de la boucle sous la forme d'une expression conditionnelle.

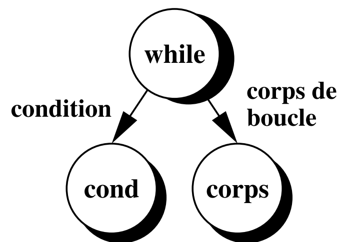

FIG. 5.6 – Arbre de syntaxe abstraite représentant une boucle

Le corps de boucle est exécuté tant que le résultat de l'expression conditionnelle est *true*. Le support des expressions évaluées dans le corps de la boucle dépend donc du résultat de l'expression conditionnelle.

Le nombre d'itérations de la boucle n'est pas connu à la compilation. Le support étendu des variables est construit de manière à indiquer toutes les dépendances possibles quel que soit le nombre d'itérations de la boucle. Ainsi, le support étendu d'une variable v est l'union du support de v après une itération, puis deux, etc. jusqu'à n itérations.

Le support d'une variable résultant de l'exécution de la boucle est construit de manière itérative :

1. Le support étendu d'une variable v avant exécution de la boucle est noté $SuppExt_v^0$;
2. Nous calculons d'abord le nouveau support étendu d'exécution du corps de boucle $SuppExt_{bloc}^i$. Nous parcourons ensuite le corps de boucle avec ce nouveau support d'exécution.
3. Nous construisons le nouveau support étendu, noté $SuppExt_v^i$, de chaque variable v résultant de l'exécution de la i-ème itération ;

4. Nous répétons n fois l'étape 2 et 3 jusqu'à temps que le support de chaque variable soit stable : $\forall v, SuppExt^{n}(v) \equiv SuppExt^{n-1}(v)$

Le support d'une variable v après l'exécution de la boucle, noté $SuppExt^{final}(v)$, est l'union des supports de v résultant de l'exécution des n itérations.

$$SuppExt^{i}_{bloc} = SuppExt^{i-1}_{bloc} \cup SuppExt_{condition}$$
$$\forall v, SuppExt^{final}(v) = \bigcup_{n\geq 0}^{i} SuppExt^{i}(v)$$

5.4.5 Appels de fonction ou de méthode

Les appels de fonction (ou de méthode) sont autorisés seulement si :

- La fonction appelée ne possède pas de paramètres ;
- La fonction appelée n'introduit pas de récursion ;
- La méthode est appliquée à l'instance courante du composant matériel : le contexte de la méthode est donc obligatoirement *this*.

Les appels ne respectant pas ces contraintes ne sont pas traités et l'analyse du support des variables s'arrête.

La figure 5.7 illustre un arbre de syntaxe abstraite représentant un appel de méthode. Le nœud père comprend trois fils :

Contexte Ce nœud détermine l'instance de l'objet auquel s'applique la méthode. Le pointeur d'instance doit être égal à *this*.

Fonction Ce nœud identifie la fonction appelée.

Paramètres Ce nœud établit la liste des paramètres utilisés lors de l'appel de fonction. La liste doit être vide.

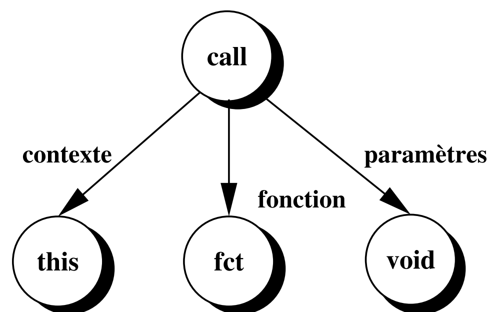

FIG. 5.7 – Arbre de syntaxe abstraite représentant un appel de méthode

Le parcours de l'arbre continue à partir de la racine de l'arbre de syntaxe abstraite représentant le corps de la fonction appelée. En fin de compte, le corps de la fonction appelée est simplement inclus dans la fonction appelante.

5.5 Identification des constantes architecturales et des variables intermédiaires

Une variable intermédiaire conserve sa valeur au plus pendant un cycle. Elle ne doit donc pas conserver sa valeur d'un cycle à l'autre. Pour éviter ce cas de figure, nous appliquons la contrainte d'écriture suivante : une variable intermédiaire doit être une variable locale aux processus. Les valeurs des variables intermédiaires sont ainsi perdues à chaque exécution complète d'un processus. Si une variable locale est lue avant d'être affectée, le compilateur est capable de l'identifier.

Un composant matériel est représenté par une classe SystemC de type *sc_module*. Une constante architecturale est une variable membre. Celle-ci doît être initialisée dans le constructeur et elle ne doit pas être modifiée au cours de la simulation.

Toutes les variables locales aux processus sont ainsi identifiées en tant que variables intermédiaires. Toutes les variables membres, qui ne sont ni de type registre, ni de type port, sont identifiées en tant que constantes architecturales.

5.6 Vérification de l'application des contraintes de modélisation

Les données nécessaires pour la vérification de l'application des contraintes de modélisation sont :

- Le support de chaque variable intervenant dans la description du composant matériel ;
- La liste des variables intermédiaires et des constantes architecturales identifiées.

Notre méthode de vérification consiste à vérifier le respect des contraintes d'écriture en automates synchrones communicants en analysant le support des variables. La table 5.2 présente les contraintes sur le support des variables en fonction du type de processus à vérifier.

Si le support des variables ne répond pas aux contraintes, la description SystemC du composant ne respecte pas la modélisation présentée dans le chapitre 2. La simulation du composant produira alors un comportement non déterministe. Le non déterminisme conduira à des erreurs en simulation.

La méthode est capable d'identifier l'expression qui ne respecte pas les contraintes d'écriture. Le concepteur est alors en mesure de corriger la description du composant matériel.

5.7 Mise en œuvre de l'outil de vérification

Notre outil de vérification est composé de deux modules. Le premier est une extension du compilateur GCC permettant de générer un fichier XML qui contient l'arbre de syntaxe abstraite représentant la description C/C++. Le deuxième module lit le fichier XML, construit les supports de variables, identifie les constantes architecturales et vérifie l'application des contraintes d'écriture. La figure 5.8 illustre le découpage de l'outil en deux modules distincts.

Type du processus	**Type de variable**	**Support**
Transition	Registre	$\subset$ {Registres, Constantes, Entrées}
	Entrée	-
	Sortie	-
	Constante	-
Génération de Moore	Registre	-
	Entrée	-
	Sortie	$\subset$ {Registres, Constantes}
	Constante	-
Génération de Mealy	Registre	-
	Entrée	-
	Sortie	$\subset$ {Registres, Constantes, Entrées}
	Constante	-
Constructeur	Registre	-
	Entrée	-
	Sortie	-
	Constante	$\subset$ {Constantes, paramètres}

TAB. 5.2 – Contraintes sur le support des variables en fonction du type de processus

Le plug-ins *GCCXML* standard ne génère que le prototype des classes et des fonctions C/C++. Nous avons donc étendu ce plug-ins pour générer l'AST complet du corps des fonctions.

Le module de vérification charge le fichier XML puis recherche les constructeurs des structures héritant de *sc_module*. Il parcourt ensuite le(s) constructeur(s) pour identifier les processus à vérifier. Le module de vérification vérifie ensuite les processus en utilisant la méthode présentée dans ce chapitre.

5.8 Résultats expérimentaux

Nous avons développé un outil implantant la méthode de vérification sémantique présentée. L'outil se nomme *Cfsm RUles cheCKER* (CRUCKER). Celui-ci a permis de vérifier les contraintes d'écriture de l'ensemble des modèles SoCLIB, à quelques exceptions près, soit plus d'une dizaine de modèles. Notre outil prototype a rencontré quelques difficultés pour traiter les modèles de micro réseaux d'interconnexions. Ces problèmes concernent le module de compilation. Ils n'ont pas été résolus, faute de temps. Cependant, ils ne remettent pas en cause la méthode de vérification.

Nous présentons les résultats obtenus pour les deux modèles suivants :

- Le composant *MultiRAM* : contrôleur de mémoire multisegment ;

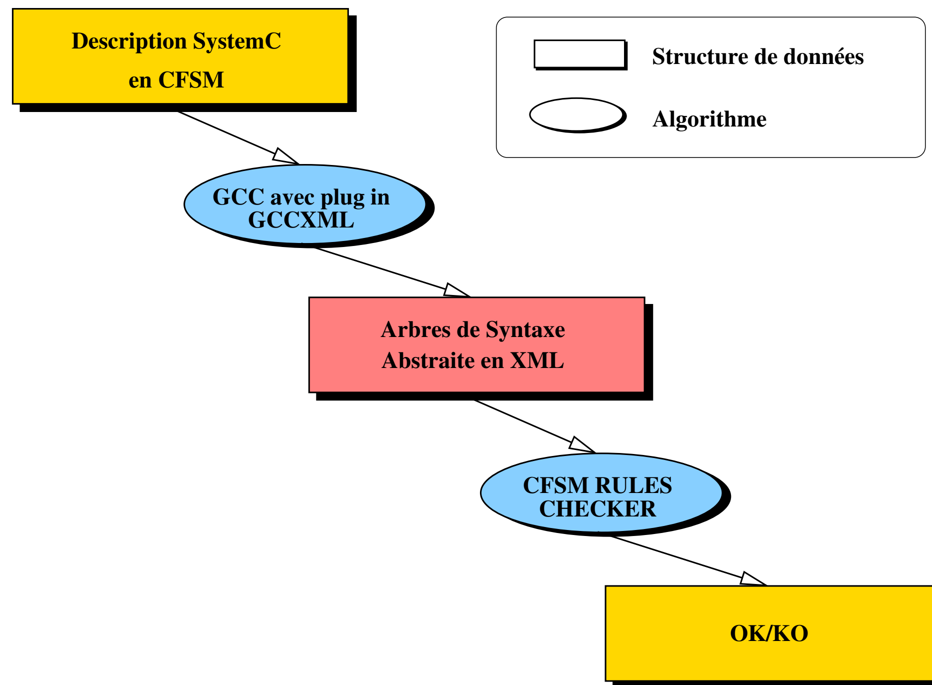

FIG. 5.8 – Flot de de la chaîne d'outil de vérification des modèles à automates synchrones communicants

- Le composant *XCache* : cache d'instruction et de données.

5.8.1 Modèles de composants matériels vérifiés

La description des composants matériels *MultiRAM* et *XCache* sont présentés dans les sections suivantes.

MultiRAM

La *MultiRAM* est un contrôleur de mémoire possédant plusieurs segments mémoires. Celui-ci communique à l'aide d'une interface VCI complètement synchrone. Alors qu'en règle générale les registres sont modélisés par des variables de type *sc_signal*, dans le cas de la *MultiRAM*, les éléments mémorisants de la mémoire sont représentés sous la forme d'un tableau d'entiers.

Le composant comporte un automate et une mémoire tampon de type FIFO de profondeur deux. À l'aide de cette mémoire tampon, l'automate est capable de recevoir une requête d'accès à la mémoire et d'envoyer, en même temps, des paquets réponses VCI à chaque cycle. Le composant *MultiRAM* est illustré par la figure 5.9.

L'automate de la *MultiRAM* ne possède pas de dépendances combinatoires entre ses ports d'entrée et ses ports de sortie. Sa description ne comporte donc pas de fonctions de génération

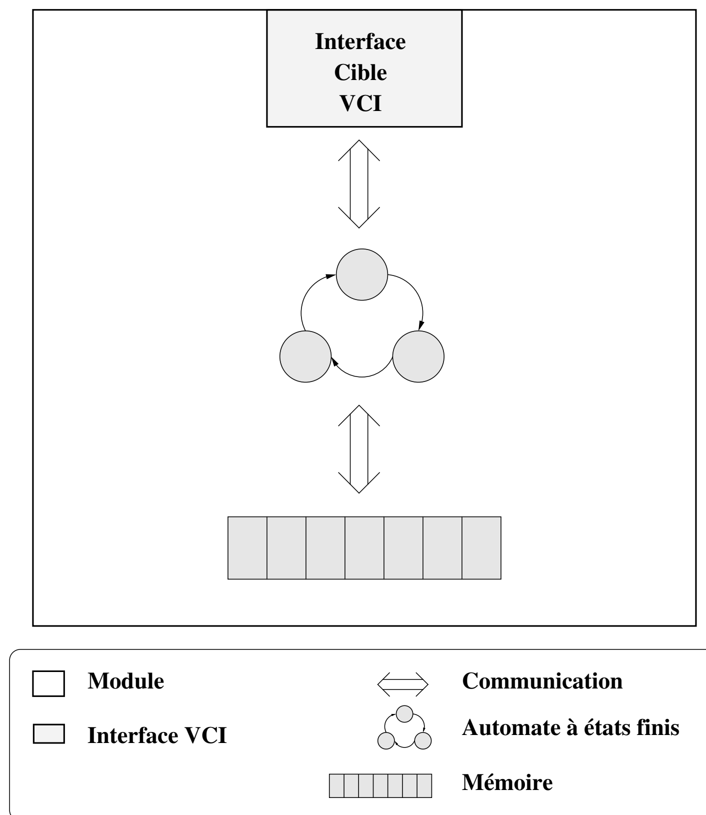

FIG. 5.9 – Contrôleur mémoire multi segment à interface VCI

de Mealy à vérifier.

L'interface VCI est paramétrable à l'aide de paramètres de template. Ces paramètres sont des constantes architecturales. Le vérificateur identifie ces constantes ainsi que les éléments mémorisants de la mémoire et valide correctement la description. Si nous modifions la description pour y introduire une erreur ne respectant pas les contraintes d'écriture en automate, le vérificateur trouve l'erreur et affiche le numéro de ligne responsable.

XCache

Le composant matériel *XCache* est à la fois un cache de données, et un cache d'instructions. La figure 5.10 présente une vue générale de l'architecture du composant. *XCache* possède deux interfaces. L'une est VCI et se connecte à un composant d'interconnexion tels qu'un bus ou un micro réseau. L'autre interface se connecte à un processeur MIPS, SPARC ou OpenRISC de la bibliothèque SoCLIB. Quatre automates décrivent le comportement du matériel :

- L'automate *FSM_DONNEES* contrôle l'interface du cache de donnée ;

- L'automate *FSM_INSTRUCTIONS* contrôle l'interface du cache instruction ;
- L'automate *FSM_REQUETES* contrôle l'interface de requête VCI ;
- L'automate *FSM_REPONSES* contrôle l'interface de réponse VCI.

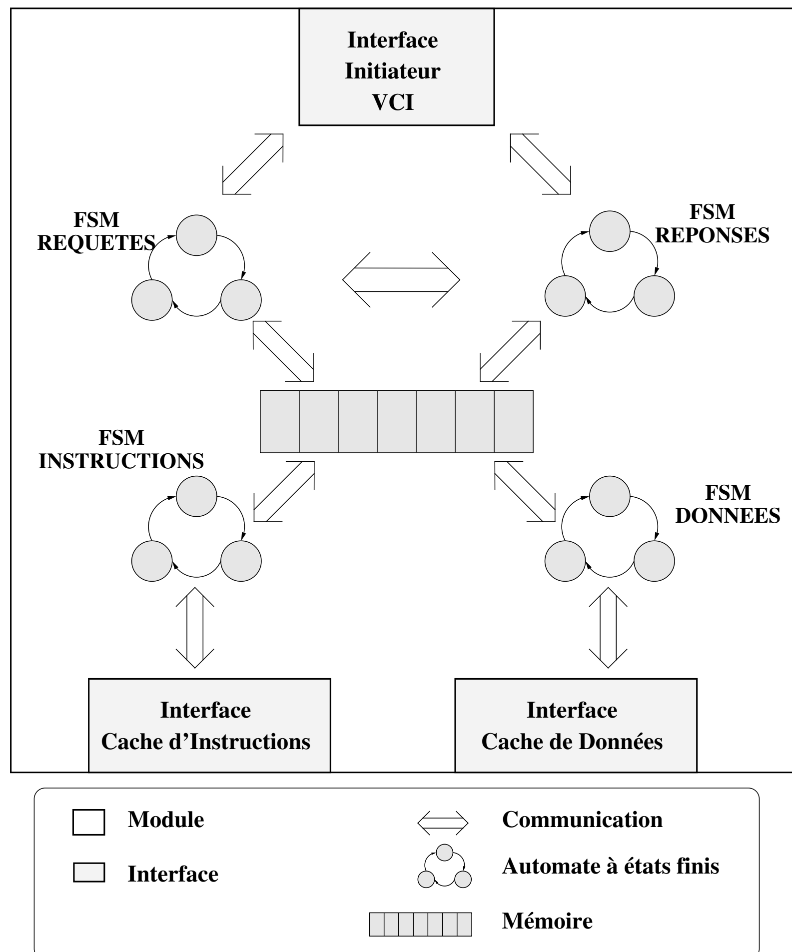

FIG. 5.10 – Mémoire cache de données et d'instructions communiquant par interface VCI

Dans le *XCache*, il existe quatre ports de sortie de Mealy :
- Le port FRZ de l'interface du cache d'instructions ;
- Le port INS contenant le code instruction ;
- Le port FRZ de l'interface du cache de données ;
- Le port $RDATA$ contenant une donnée.

Ces ports de sortie dépendent directement de certains ports d'entrée, de manière combinatoire. Par exemple, le port de sortie INS dépend du port d'entrée ADR définissant l'adresse de l'instruction, et du port d'entrée REQ spécifiant si la requête est valide. La figure 5.11 illustre

toutes les dépendances combinatoires entre les ports d'entrée et les ports de sortie du composant *XCache*.

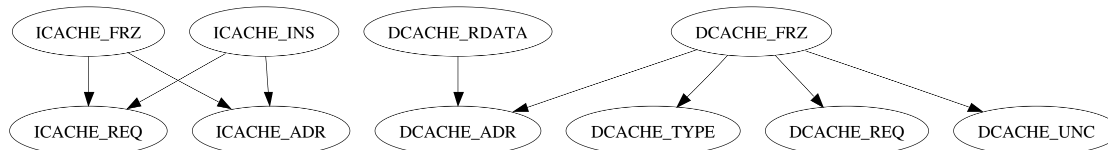

FIG. 5.11 – Graphe de dépendances combinatoires entre les ports d'entrée et les ports de sortie du XCache

La description SystemC du *XCache* possède un processus de type génération de Mealy permettant de décrire les dépendances combinatoires entre les ports d'entrée et les ports de sortie. Le vérificateur vérifie correctement ce processus, ainsi que les processus de type transition et génération de Moore.

5.8.2 Bilan des résultats

L'outil de vérification sémantique par analyse statique a permis de vérifier les composants de la bibliothèque SoCLIB. L'analyse nécessite quelques secondes pour identifier les expressions qui ne respectent pas le modèle formel des automates synchrones communicants.

Lorsqu'une erreur de modélisation est détectée, le message d'erreur indique la cause et le numéro de ligne dans la description.

Les résultats de la vérification sémantique de modèle à automate sont corrects. La méthode comporte cependant quelques inconvénients essentiellement dûs aux incompatibilités à la compilation.

Il existe un décalage entre la norme C/C++ et son application. En effet, les compilateurs interprètent la norme de manière différente, ce qui implique des problèmes de compilation avec certaines descriptions SystemC.

De plus, les outils permettant de manipuler l'AST d'une fonction C/C++ sont peu répandus. Par conséquent, la mise en œuvre d'un outil de vérification par analyse sémantique n'est pas aisée.

5.9 Conclusions

Lors de l'exploration architecturale, le concepteur cherche à simuler l'architecture matérielle le plus rapidement possible. Le simulateur SystemCASS permet d'obtenir une vitesse de simulation importante à l'aide d'un échéancier à ordonnancement totalement statique.

Pour utiliser l'ordonnancement totalement statique de SystemCASS, tous les modèles de l'architecture doivent respecter la modélisation en CFSM présentée au chapitre 2. Si un modèle ne respecte pas les contraintes de modélisation, les résultats de simulation fournis par SystemCASS peuvent être incorrects.

Ce chapitre a présenté deux méthodes de vérification. La première est une vérification à l'exécution. Les vérifications au cours de l'exécution ralentissent considérablement la simulation. De plus, il existe des erreurs de modélisation qui ne peuvent être détectées par cette méthode.

La deuxième méthode de vérification repose sur une vérification sémantique par analyse statique. La description du matériel est d'abord compilée pour construire un arbre de syntaxe abstraite pour chaque processus et pour chaque constructeur. Les arbres de syntaxe abstraite sont ensuite analysés pour construire le support des variables. Enfin, le support des variables est vérifié pour déterminer si le modèle respecte, ou non, le modèle formel des automates synchrones communicants.

Deux modèles de la bibliothèque SoCLIB ont été vérifiés correctement par le prototype d'outil de vérification. La méthode de vérification sémantique s'est révélée rapide et efficace. Cependant, le module assurant la compilation de la description C/C++ souffre de quelques problèmes de compatibilité.

Chapitre 6

Conclusions et perspectives

Sommaire

6.1 Bilan des travaux **124**
6.1.1 Modélisation précise au cycle et au bit près sur les interfaces 124
6.1.2 Ordonnancement totalement statique pour la simulation 124
6.1.3 Génération automatique de modèle de simulation respectant le modèle des CFSM 125
6.1.4 Vérification sémantique des modèles à automates synchrones communicants 125
6.2 Conclusions générales **126**
6.3 Perspectives **126**

Modéliser un système intégré sur puce consiste à modéliser l'architecture matérielle et à décrire l'application logicielle puis à déployer ce logiciel sur ce matériel. Le concepteur de système cherche à optimiser le système intégré lors de l'exploration architecturale. Les principaux critères d'optimisation sont la surface de silicium occupée, la consommation d'énergie du système intégré, et les performances en nombre de cycles. L'évaluation d'une solution de déploiement est obtenue par simulation architecturale.

L'objectif de cette thèse est d'accélérer la simulation d'architectures complexes décrites en langage SystemC, avec des modèles précis au bit près et au cycle près. Dans les sections suivantes, nous dressons un bilan des travaux réalisés. Nous concluons ensuite sur l'intérêt des méthodes proposées. Enfin, nous présentons quelques perspectives de recherche à la suite de ces travaux.

6.1 Bilan des travaux

Au cours de cette thèse, nous avons proposé de faciliter le développement des architectures matérielles et d'accélerer leur simulation. Nous avons alors présenté des méthodes ainsi que des outils à travers quatre chapitres :

- Modélisation précise au cycle et au bit près sur les interfaces, respectant le modèle des automates synchrones communicants (CFSM) ;
- Ordonnancement totalement statique pour la simulation ;
- Génération automatique de modèle de simulation respectant le modèle des CFSM ;
- Vérification sémantique des modèles en CFSM.

Nous rappelons ci-dessous le bilan pour chaque chapitre.

6.1.1 Modélisation précise au cycle et au bit près sur les interfaces

Les architectures matérielles que nous cherchons à simuler sont multiprocesseurs. Nous avons décrit une approche de modélisation exploitant le modèle formel des automates synchrones communicants, permettant une modélisation précise au cycle près et au bit près. À la description du matériel, nous ajoutons la description explicite des dépendances combinatoires entre les ports d'entrées et les ports de sortie au sein d'un même module.

Les composants de la bibliothèque SoCLIB respectent la méthodologie proposée. Les composants peuvent être directement simulés en utilisant le simulateur événementiel de la distribution SystemC standard.

6.1.2 Ordonnancement totalement statique pour la simulation

Les architectures que nous souhaitons simuler contiennent uniquement des modules respectant le modèle des automates synchrones communicants.

Le comportement d'un composant matériel est décrit par un ensemble de processus. La simulation architecturale d'un tel composant nécessite un ordonnancement des processus sur la

station de travail.

L'objectif de l'ordonnancement est de calculer un ordre d'exécution des processus. Les simulateurs d'architecture utilisent généralement un échéancier à ordonnancement dynamique. Ce type d'échéancier coûte cher en terme de performance.

Nous avons proposé un ordonnancement totalement statique exploitant les propriétés des modules décrits sous la forme des CFSM. Les relations entre les ports d'entrée et les ports de sortie au sein d'un même module sont fournies explicitement par le concepteur. Ces informations permettent de construire un ordonnancement totalement statique, dont le coût en terme de performance est nul.

Le simulateur à ordonnancement totalement statique, SystemCASS, permet une accélération d'un ordre de grandeur par rapport au simulateur SystemC d'OSCI, et une accélération significative par rapport à d'autres simulateurs tels que FastSystemC ou CASS.

6.1.3 Génération automatique de modèle de simulation respectant le modèle des CFSM

Pour accélérer la construction d'une architecture matérielle en vue de l'exploration architecturale, nous souhaitons utiliser les modèles déjà existants : il existe de nombreux modèles synthétisables de niveau RTL dans l'industrie. Cependant, ceux-ci sont généralement écrits en VHDL ou en Verilog. Ces modèles offrent de très mauvaises performances en terme de vitesse de simulation.

Notre objectif est de convertir automatiquement les modèles VHDL au niveau RTL pour obtenir des modèles SystemC respectant le principe des automates synchrones communicants. Ces modèles sont alors compatibles avec le simulateur SystemC d'OSCI, ainsi qu'avec SystemCASS. Ceux-ci sont alors exploitables pour accélérer l'exploration architecturale.

La méthode proposée repose sur une analyse sémantique de la description VHDL initiale pour construire un automate dont le comportement est équivalent. Les premiers résultats expérimentaux sont encourageants. Malheureusement, l'analyse des tableaux VHDL pose problème. Les modèles comportant des tableaux ne peuvent être analysés par notre méthode. Nous avons alors présenté des perspectives de recherche pour résoudre ce problème.

6.1.4 Vérification sémantique des modèles à automates synchrones communicants

Dans le cas où les modèles ne respectent pas l'approche de modélisation en CFSM, l'ordonnancement totalement statique du simulateur SystemCASS conduit à des erreurs de simulation. L'objectif est de vérifier les modèles avant simulation.

La méthode de vérification proposée repose sur une vérification sémantique par analyse statique. Les résultats expérimentaux ont montré que la méthode permet de détecter toutes les erreurs de modélisation en quelques secondes.

6.2 Conclusions générales

Au cours de cette thèse, nous avons proposé des méthodes pour faciliter le développement des architectures matérielles : modélisation sous la forme d'automates synchrones communicants, génération automatique de modèle de simulation, et vérification sémantique des modèles. Nous avons également proposé des méthodes pour accélérer la simulation : ordonnancement totalement statique.

La bibliothèque de composants, SoCLIB, ainsi que trois outils (RTL2CFSM, CRUCKER, SystemCASS) ont été mis au point pour valider nos méthodes. Ceux-ci ont montré leur efficacité gràce à des résultats expérimentaux très satisfaisants.

Plusieurs dizaines de contributeurs participent maintenant au développement de SoCLIB et SystemCASS. SoCLIB et SystemCASS sont utilisés depuis quelques années dans des travaux de recherche, tels que des stages M2 recherche, et des thèses. Les principaux avantages, appréciés par les concepteurs, sont l'inter-opérabilité des composants SoCLIB et la rapidité de simulation de SystemCASS.

6.3 Perspectives

Comme nous l'avons évoqué dans le chapitre 4, la méthode de génération automatique de modèle de simulation, à partir de descriptions VHDL synthétisables, a des difficultés pour traiter des architectures de taille réelle. En effet, les tableaux VHDL posent de serieux problèmes lors de l'analyse sémantique.

La solution envisagée a été brièvement présentée. Celle-ci consiste à remplacer l'usage des tableaux, dans la description des composants matériels, par des composants matériels prédéfinis fournissant le même comportement que les tableaux. Il suffit alors d'instancier ces composants au lieu d'utiliser les tableaux VHDL.

L'outil d'analyse sauterait l'étape de compilation des composants prédéfinis. Cette solution permettrait de rendre l'outil RTL2CFSM capable de traiter n'importe quelle description VHDL de composant matériel de taille réel, au prix de quelques modifications du code VHDL.

Annexe A

Annexe : Support d'une expression

Nous définissons deux types de support d'expression :
- Le support simple ;
- Le support étendu.

A.1 Définition du support simple d'une expression

Le support simple $Supp(e)$ d'une expression est l'ensemble des variables qui interviennent directement dans le calcul de l'expression e.

$$\forall e, e = f(x_1, ..., x_n),$$
$$Supp(e) = \{x_1, ..., x_n\}$$

La figure A.1 présente un graphe de dépendances entre variables ainsi que le support simple de chaque variable. A dépend de B et C. C dépend de A et D. Les supports simples de A et C sont :
$Supp(A) = \{B, C\}$
$Supp(C) = \{A, D\}$

A.2 Définition du support étendu d'une expression

Le support étendu d'une expression est l'ensemble des variables qui interviennent directement dans le calcul de l'expression e ou qui interviennent dans le calcul de ses sous expressions.

$$SuppExt(e) = Supp(e) \bigcup_v SuppExt(v)$$
$$\text{avec } v \in SuppExt(e)$$

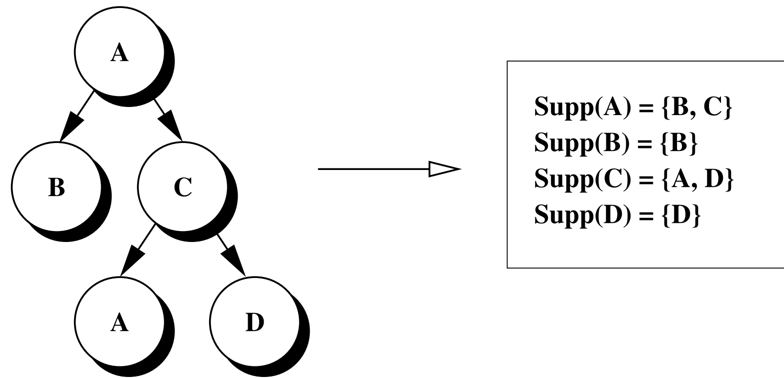

FIG. A.1 – Graphe de dépendances entre variables et support simple des variables

Nous reprenons l'exemple présenté dans la section A.1. La figure A.2 présente le support étendu de chaque variable. Rappelons que A dépend de B et C qui dépend lui même de A et D. Nous avons les supports étendus suivants :
$SuppExt(A) = \{A,B,C,D\}$
$SuppExt(C) = \{A,D\}$

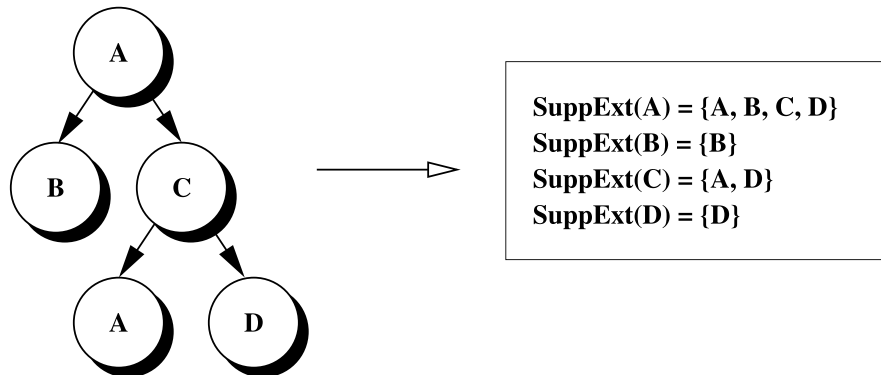

FIG. A.2 – Graphe de dépendances entre variables et support étendu des variables

Annexe B

Acronymes

ABL *Arbre Binaire Lisp*

API *Application Program Interface*

AST *Arbre de Syntaxe Abstraite*

BDD *Binary Decision Diagram(s)*

CABA *Cycle Accurate, Bit Accurate*

CASS *Cycle Accurate System Simulator*

CFSM *Communicating Finite State Machine*

DAG *Directed Acyclic Graph*

DMA *Direct Memory Access*

FSM *Finite State Machine*

CRUCKER *Cfsm RUles cheCKER*

FIFO *First In First Out*

IDCT *Inverse Discrete Cosine Transform*

IQ *Inverse Quantification*

GCC *Gnu Compiler Collection*

LIP6 *Laboratoire d'Informatique de Paris 6*

MIPS *Million Instructions Per Second*

MJPEG *Motion JPEG*

OSCI *Open SystemC Initiative*

PAPR *Programmation et Architecture des Processeurs Réseaux*

RISC *Reduced Instruction Set Chip*

ROBDD *Reduced Ordered Binary Decision Diagram(s)*

RTL *Register Transfer Level*

RTL2CFSM *RTL To CFSM*

SoCLIB *System on Chip LIBrary*

SystemCASS *SystemC Accurate System Simulator*

TLM-T *Transaction Level Modeling with Time*

TTY *TeleTYpewriter*

V2CASS *VASY To CASS*

VASY *VHDL Analyzer for SYnthesis*

VCI *Virtual Component Interface* [All97]

Verilog2SC *Verilog To SystemC*

VEX *VHDL EXpression*

VHDL *Very High Speed Integrated Circuits Hardware Description Language*

VHDL2SC *VHDL To SystemC*

VLD *Variable Length Decoder*

XML *eXtended Markup Language*

ZZ *ZigZag*

Bibliographie

[AFF+01] N. Agliada, A. Fin, F. Fummi, M. Martignano, and G. Pravdelli. On the reuse of vhdl modules into systemc designs. In *Forum on Design Languages (FDL01)*, 2001.

[AJFA00] Zerrouki Amal, Dunoyer Julien, Wajsbürt Franck, and Derieux Anne. Design of the hadamard coprocessor with the alliance cad system carried by post-graduating students. In *3rd European Workshop on Microelectronics Education (EWME)*, pages 265–268, 2000.

[All97] VSI Alliance. *VSIA Architecture Document*, 1997. http ://www.vsia.org.

[CM81] K. M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Commun. ACM*, 24(4) :198–206, 1981.

[Den01] Hommais Denis. *Une méthode d'évaluation et de synthèse des communications dans les systèmes intégrés matériel-logiciel*. PhD thesis, 2001.

[eDS64] M. O. Rabin et D. Scott. Sequential machines. pages 63–91, 1964.

[FBGP04] E. Faure, S. Berrayana, D. Genius, and F. Pétrot. Application télécom pour processeur réseau. In *Sciences Electroniques Technologies de l'Information et des Télécommunications (SETIT'04)*, mars 2004.

[FGPB04] E. Faure, D. Genius, F. Pétrot, and S. Berrayana. Modular on chip multi processor for routing applications. In *EUROPAR 04*, pages 847–855, 2004.

[Gil04] Mouchard Gilles. *Modélisation de Processeurs et de Systèmes*. PhD thesis, 2004.

[Goo92] James Goodman. *A programmer's view of computer architecture : with examples from the MIPS RISC architecture*. Fort worth : Saunders college pub. edition, 1992.

[GPC+06] A. Greiner, F. Petrot, M. Carrier, M. Benabdenbi, R. Chotin-avot, and R. Labayrade. Mapping an obstacles detection, stereo vision-based, software application on a multi-processor system-on-chip. In *Proceedings of Intelligent Vehicles Symposium 2006*, pages 370–376. IEEE, 2006.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In *IFIP Congress : Information Processing*. North-Holland Publishing Co., 1974.

[Kan88] Gerry Kane. *MIPS RISC architecture*. Englewood Cliffs, NJ : Prentice-Hall, 1988.

[LMAH02] O. F. Nadjarbashi L. Mahmoudi Ayough, A. Haj Abutalebi and S. Hessabi. Verilog2sc : A methodology for converting verilog. hdl to systemc. In *Proceedings of the 11th International HDL Conference (HDL Con 2002)*, pages 211–217, 2002.

[Lud99] Jacomme Ludovic. *Analyse sémantique de descriptions VHDL synchrones en vue de la synthèse*. PhD thesis, 1999.

[Mea55] G. Mealy. A method for synthesizing sequential circuits. pages 1045–1079, 1955.

[Moo56] E. Moore. Gedanken experiments on sequential machines. pages 129–153, 1956.

[Mor82] Bernard M. E. Moret. Decision trees and diagrams. *ACM Comput. Surv.*, 14(4) :593–623, 1982.

[MP63] M. Rabin et E. Shamir M. Perles. The theory of definite automata. pages 233–243, 1963.

[PHG97] Frederic Petrot, Denis Hommais, and Alain Greiner. Cycle precise core based hardware/software system simulation with predictable event propagation. In *Proceeding of the* 23rd *Euromicro Conference*, pages 182–187, Budapest, Hungary, September 1997. ASIM/LIP6/UPMC, IEEE.

[PMT04] Daniel Gracia Perez, Gilles Mouchard, and Olivier Temam. A new optimized implemention of the systemc engine using acyclic scheduling. In *Design, Automation and Test in Europe Conference and Exhibition Volume I (DATE'04)*, page 10552, Paris, France, 2004. ALCHEMY INRIA Futurs & LRI, Paris South University, IEEE.

[RFA02] Buchmann Richard, Petrot Frederic, and Greiner Alain. Pilotage evenementiel versus ordonnancement statique. In *Troisième colloque du GDR CAO de circuits et systèmes integres, Paris, France*, pages 151–154. ASIM/LIP6/UPMC, 2002.

[SMB87] S. P. Smith, M. R. Mercer, and B. Brodk. Demand driven simulation : Backsim. In *DAC '87 : Proceedings of the 24th ACM/IEEE conference on Design automation*, pages 181–187, New York, NY, USA, 1987. ACM Press.

[SoC03] SoCLIB. A modelisation & simulation plat-form for system on chip, 2003. http ://-soclib.lip6.fr.

[WM90] Zhicheng Wang and Peter M. Maurer. Lecsim : a levelized event driven compiled logic simulation. In *DAC '90 : Proceedings of the 27th ACM/IEEE conference on Design automation*, pages 491–496, New York, NY, USA, 1990. ACM Press.

Liste des tableaux

2.1 Opérations autorisées dans une fonction de transition 24
2.2 Opérations autorisées dans une fonction de génération de Moore 25
2.3 Opérations autorisées dans une fonction de génération de Mealy 25

3.1 Simulateurs comparés lors des expérimentations 54

4.1 Liste non exhaustive d'association de mots clefs Verilog/SystemC 67
4.2 Conversion des types de données VHDL vers SystemC 94
4.3 Listes de sensibilité et dépendances entre ports en fonction du type de processus 95

5.1 Contraintes vérifiées à l'exécution 105
5.2 Contraintes sur le support des variables en fonction du type de processus 117

Table des figures

1.1 Graphe de tâches de l'application de décompression MJPEG 3
1.2 Architecture matérielle multiprocesseur 4
1.3 Déploiement d'une application MJPEG sur une architecture matérielle 5
1.4 Compromis performance/précision en fonction du niveau d'abstraction 7
1.5 Exemple d'architecture matérielle impliquant des réveils inutiles de processus . 9

2.1 Modélisation au niveau RTL d'un processeur à 5 étages de pipeline 15
2.2 Modélisation simplifiée du MIPS R3000 en 2 étages de pipeline 16
2.3 Automates à états finis communicants 17
2.4 Exemple d'architecture matérielle problématique pour CASS 19
2.5 Automates à états finis communicants 23
2.6 Fonction de transition comportant deux automates interdépendants 24
2.7 Exemple d'architecture matérielle avec trois fonctions de Mealy 26
2.8 Additionneur accumulateur 32 bits 31
2.9 DMA simplifié à interface VCI 33

3.1 Exemple : Architecture 42
3.2 Exemple : Graphe de dépendances de signaux triés 43
3.3 Exemple : Graphe de dépendances entre processus 44
3.4 Méthode pour l'ordonnancement totalement statique 46
3.5 Graphe de dépendances entre les ports 47
3.6 Exemple : Graphe de dépendances entre signaux 48
3.7 Exemple : Graphe de dépendances combinatoires entre signaux après ordonnancement de A 50
3.8 Exemple : Graphe de dépendances combinatoires entre signaux après ordonnancement de D 50
3.9 Exemple : Graphe de dépendances combinatoires entre signaux après ordonnancement de B 51
3.10 Exemple : Graphe de dépendances combinatoires entre signaux après ordonnancement de C 51

3.11 Exemple : Architecture modifiée pour satisfaire le critère de l'algorithme d'ordonnancement totalement statique . . . 53
3.12 PAPR : Architecture matérielle . . . 55
3.13 PAPR : Vitesse de simulation en fonction du nombre de processeurs . . . 56
3.14 PAPR : Vitesse de SystemCASS par rapport à SystemC 2.1.v1 en fonction du nombre de processeurs . . . 56
3.15 Stéréo-vision : Architecture matérielle générale . . . 57
3.16 Stéréo-vision : Architecture matérielle d'un cluster . . . 58
3.17 Stéréo-vision : Résultats de simulation . . . 58
3.18 Processeur Java : Architecture interne . . . 59
3.19 Processeur Java : Comparaison des vitesses de simulation . . . 60
3.20 Processeur Java : Graphe de dépendances entre modules . . . 61
3.21 Processeur Java : Sous partie du graphe de dépendances entre signaux . . . 62

4.1 Verilog2SC : Exemple de la conversion d'un composant . . . 68
4.2 Construction de l'ensemble des états de l'automate correspondant à un processus à plusieurs points de suspension . . . 70
4.3 Méthode pour la génération de modèle rapide pour la simulation . . . 73
4.4 Représentation d'une expression par la structure de données VEX . . . 74
4.5 Graphe de contrôle représenté par un réseau de Pétri . . . 75
4.6 Graphe de contrôle réduit représenté par un réseau de Pétri . . . 75
4.7 Liste d'assignations concurrentes décrivant le comportement global d'un composant . . . 76
4.8 Classement et ordonnancement des assignations concurrentes dans le modèle des CFSM . . . 78
4.9 Graphe de contrôle d'un processus sous la forme d'un réseau de Pétri . . . 79
4.10 Réseau de Pétri représentant deux affectations séquentielles . . . 80
4.11 Réseau de Pétri représentant une instruction conditionnelle . . . 81
4.12 Réseau de Pétri représentant une instruction de boucle . . . 81
4.13 Réseau de Pétri représentant une instruction de suspension . . . 82
4.14 Réduction séquentielle du graphe de contrôle . . . 83
4.15 Réduction séquentielle du graphe de contrôle (2) . . . 83
4.16 Réduction latérale du graphe de contrôle . . . 84
4.17 Énumeration des chemins du graphe de contrôle . . . 85
4.18 Graphe de contrôle semi réduit . . . 86
4.19 Réduction finale du graphe de contrôle réduit . . . 86
4.20 Graphe de dépendances entre variables . . . 90
4.21 Graphe de dépendances entre variables étiquetées . . . 92
4.22 Graphe de dépendances entre variables étiquetées, après tri topologique . . . 93
4.23 Hadamard : Architecture interne du coprocesseur . . . 99

5.1 Méthode pour la vérification statique par analyse sémantique 108
5.2 Arbre de syntaxe abstraite représentant une affectation en SystemC 109
5.3 Arbre de syntaxe abstraite représentant une affectation 112
5.4 Arbre de syntaxe abstraite représentant une opération binaire, arithmétique ou logique 112
5.5 Arbre de syntaxe abstraite représentant une instruction conditionnelle 113
5.6 Arbre de syntaxe abstraite représentant une boucle 114
5.7 Arbre de syntaxe abstraite représentant un appel de méthode 115
5.8 Flot de de la chaîne d'outil de vérification des modèles à automates synchrones communicants 118
5.9 Contrôleur mémoire multi segment à interface VCI 119
5.10 Mémoire cache de données et d'instructions communiquant par interface VCI . 120
5.11 Graphe de dépendances combinatoires entre les ports d'entrée et les ports de sortie du XCache 121

A.1 Graphe de dépendances entre variables et support simple des variables ii
A.2 Graphe de dépendances entre variables et support étendu des variables ii

Listings

2.1 Description d'un module SystemC . . . 28
2.2 Déclaration des dépendances entre les ports d'entrée et les ports de sortie . . . 29
2.3 Additionneur accumulateur 32 bits décrit en SystemC CABA . . . 32
2.4 DMA simplifié décrit en SystemC CABA : déclaration des ports externes . . . 33
2.5 DMA simplifié décrit en SystemC CABA : déclaration des registres . . . 34
2.6 DMA simplifié décrit en SystemC CABA : fonction de transition . . . 35
2.7 DMA simplifié décrit en SystemC CABA : fonction de génération . . . 36
4.1 Description VHDL synthétisable au niveau RTL . . . 76
4.2 Interface SystemC générée respectant la modélisation en automate . . . 96
4.3 Fonction de transition générée en SystemC . . . 96
4.4 Fonction de génération de Moore générée en SystemC . . . 96
4.5 Fonction de génération de Mealy générée en SystemC . . . 97
4.6 Constructeur généré en SystemC . . . 97
5.1 Description SystemC ne respectant pas la modélisation en automates . . . 106
5.2 Description SystemC illustrant le support d'exécution . . . 111

Liste des algorithmes

1 Transformation d'un graphe de dépendances entre ports G_{ports} en graphe de dépendances entre signaux $G_{signaux}$ 48
2 Sélection du processus à ordonnancer 52
3 Construction de l'ordre d'exécution des processus 52
4 Construction du graphe de dépendances entre variables à partir d'une liste d'assignations concurrentes 90
5 Étiquetage des nœuds du graphe de dépendances entre variables 91