Université de Montréal

Covérification des Systèmes Intégrés

par

Mostafa Azizi

Département d'Informatique et de Recherche Opérationnelle
Faculté des Arts et des Sciences

Thèse présentée à la Faculté des Études Supérieures
en vue de l'obtention du grade de Philosophiæ Doctor (Ph. D.)
en Informatique

Décembre 2000

Université de Montréal

Faculté des Études Supérieures

Cette thèse intitulée :

Covérification des Systèmes Intégrés

présentée par :

Mostafa Azizi

a été évaluée par un jury composé des personnes suivantes :

Mme Rachida Dssouli	Présidente – Rapporteuse
M. El Mostapha Aboulhamid	Directeur de recherche
M. Sofiène Tahar	Codirecteur de recherche
M. Guy Bois	Membre du jury
M. Claude Thibeault	Examinateur externe
M. Georges Azuelos	Représentant du doyen

Thèse acceptée le : 05 juillet 2001

Sommaire

Au cours des décennies passées, la conception des composantes logicielle et matérielle d'un système intégré s'effectuait séparément. Puis, le logiciel (abrégé par L) obtenu était exécuté sur le matériel (abrégé par M) prototype. Si les contraintes de la spécification requises pour le système ne sont pas satisfaites, le processus de conception est réitéré dans l'espoir de retrouver un bon prototype. Cette technique de vérification des propriétés d'un tel système s'est avérée lourdement coûteuse en termes du temps de réalisation et du coût du système global. Pour remédier à une telle situation, plusieurs compagnies et universités se sont penchées sur l'élaboration d'environnements de codesign L/M ayant de meilleures performances. Aussi, ces derniers devront-ils intégrer des outils efficaces et fiables pour accomplir des tâches de covérification. Afin d'atteindre un niveau maximum de succès, la covérification doit être exécutée en concurrence avec le codesign; en d'autres mots, il est nécessaire qu'elle soit intégrée au codesign dès ses premières phases.

La covérification, a priori, peut être effectuée par trois techniques principales : « bread-boarding », cosimulation et covérification formelle. La première est utilisée rarement ou lorsque les autres méthodes échouent. La troisième technique, elle et la vérification formelle se chevauchent largement et elle nécessite d'être revue et développée. Quant à la deuxième, la cosimulation, elle est abondamment utilisée pour les systèmes mixtes à tel point qu'on confond les termes covérification et cosimulation; elle consiste à simuler simultanément les parties logicielle et matérielle d'un système tout en assurant la communication entre les deux. Nous proposons dans cette thèse d'améliorer cette technique par une méthodologie combinant les concepts de base de la simulation, des inspirations de la vérification formelle et des techniques de la programmation orientée objet.

Notre méthodologie de covérification est basée sur le concept du «multithreading». Dans cette approche, la partie logicielle est décrite par un ensemble de threads communicants, l'interaction entre les deux parties logicielle et matérielle s'effectue via des registres, et des threads additionnels sont utilisés pour exprimer des assertions de vérification. Finalement, la cosimulation du code global est exécuté comme un modèle unifié (homogène) à l'aide du compilateur Java ou comme un modèle distribué (hétérogène) à l'aide d'un environnement de covérification.

Partant d'une spécification d'un système L/M, le concepteur travaille à parfaire sa conception et à coder correctement son implémentation. Mais comme personne n'est parfaite, il y a toujours une probabilité de présence d'erreurs au moins implicites (de fonctionnalité) dans le code (des centaines de lignes) bien que sa compilation soit accomplie avec succès (c'est-à-dire sa syntaxe est correcte). Cette situation de doute exige une validation de l'implémentation par rapport à sa spécification. Pour ce faire, des propriétés extraites de la spécification doivent être satisfaites par l'implémentation globale des deux parties logicielle et matérielle, y compris leur interface de communication.

Les propriétés décrivent un tout ou une partie de la spécification du système L/M. Leurs expressions sont formulées en combinant éventuellement des paramètres de la partie logicielle seule, de la partie hardware seule ou des deux à la fois. Ces propriétés sont définies selon les formats d'écriture d'une propriété en CPL («Coverification Properties Language»). CPL est un langage simple que nous avons développé afin de décrire, séparément de l'implémentation, l'ensemble des propriétés à covérifier par notre outil JACOV («JAva COVerification tool»). Ces propriétés contenues dans un fichier sont ensuite traduites et rangées dans des threads de Java.

La covérification des propriétés susmentionnées est effectuée par un processus de simulation/cosimulation. Étant donné que le système et ses propriétés à covérifier sont répartis en threads, nous amorçons la simulation par une séquence significative de vecteurs de test. Cette séquence devrait être choisie de manière à ce que les propriétés réagissent à l'application de chacun de ses éléments (de la séquence). Les réponses de toutes les propriétés recueillies durant la simulation sont analysées pour conclure des chances de réussite de la covérification. Si une des propriétés a fait preuve d'un comportement incorrect, alors ceci est un signe suffisant de la présence d'une anomalie au sein de l'implémentation (sachant bien sûr que toutes les propriétés sont correctes des points de vue de la syntaxe et de la logique). Toutefois, si toutes les propriétés ont satisfait les tests qu'elles ont subis, alors la covérification est prometteuse mais elle ne garantit pas que l'implémentation soit absolument correcte.

Avant de conclure cette thèse, nous avons appliqué notre méthodologie sur un modèle d'un système à plusieurs processeurs et à mémoire partagée afin de contrôler la cohérence des données des différentes caches (chaque processeur en possède une). Nous avons observé et illustré les traces du flux de données du système dans les deux cas : sans et avec protocole de cohérence; les résultats obtenus sont très positifs. Nous avons fait aussi une vérification formelle du même système par la technique de «model checking» offerte par l'outil VIS («Verification Interacting with Synthesis tool») et montré l'aspect de complémentarité entre la covérification et la vérification formelle surtout quand la réalisation de cette dernière s'avère très coûteuse ou impossible.

Mots clés

Covérification L/M

Vérification formelle

Cosimulation

Threads

Propriétés

Systèmes intégrés

Systèmes L/M

Simulation séquentielle

Simulation distribuée

Spécification

Test

Table des matières

Liste des tableaux

Liste des figures

Liste des sigles et abréviations

AMC	Applied Microsystems Corporation
ASIC	Application Specific Integrated Circuit
BDD	Binary Decision Diagrams
CDFG	Control Data Flow Graphs
CFSM	Codesign Finite State Machine
CPL	Coverification Properties Language
CPU	Central Processing Unit
CTL	Computation Tree Logic
DS	Distributed Simulation
EDA	Eagle Design Automation
EFSM	Extended Finite State Machine
FIFO	First Input First Output
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
HDL	Hardware Description Language
HLL	High Level Language
HOL	High Order Logic
HW	Hardware
ICE	In-Circuit Emulator
ILA	Iterative Logic Array
IP	Intellectual Property
ISS	Instruction-set Simulator
JACOV	JAva-COVerification tool
L	Logiciel
LTS	Labeled Transition Space
M	Matériel
NORTEL	NORthern TELecom, Canada
RAM	Random Access Memory
RFV	Register For Verification
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
RTL	Register Transfer Level

SS	Sequential Simulation
SW	Software
VHDL	VHSIC Hardware Description Language
VIS	Verification Interacting with Synthesis tool
VIS-Verilog	Sous-ensemble de Verilog admissible par VIS
VSP	Virtual Software Processor

Liste des contributions[1]

1. M. Azizi, E. -M. Aboulhamid, and S. Tahar, "Sequential and Distributed Simulations using Java Threads", Proceedings of the International Conference on Parallel Computing in Electrical Engineering (PARELEC'2000), IEEE Computer Society, pp. 237-241, Trois-Rivières, Quebec, Canada, August 27-30, 2000 (ISBN 0-7695-0759-X)

2. E. -M. Aboulhamid, M. Azizi, and X. Song, "Hardware Design and Verification Methods", Book chapter in the Encyclopedia of Computer Science and Technology, Vol. 42, pp. 67-91, A. Kent and J. G. Williams Eds., Marcel Dekker, Inc., New York, USA, 2000 (ISBN 0-8247-2295-7)

3. M. Azizi, E. -M. Aboulhamid et S. Tahar, "Entre la covérification concurrente et l'ordonnancement des threads", Colloque de la Microélectronique, Congrès ACFAS'2000, Université de Montréal, Montréal, Canada, 10-14 mai 2000 (http://www.acfas.ca/)

4. M. Azizi, E. -M. Aboulhamid, and S. Tahar, "Properties Coverification in HW/SW Systems", Proceedings of the 2nd International Conference of Electronic Circuits and Systems (ECS'99), pp. 80-83, Bratislava, Slovakia, September 6-8, 1999 (http://www.ieee.org)

5. M. Azizi, E. -M. Aboulhamid, and S. Tahar, "Multithreading-Based Coverification Technique of HW/SW Systems", Proceedings of the International Conference of Parallel and Distributed Processing Techniques and Applications (PDPTA'99), CSREA Press, Vol. IV, pp. 1999-2005, Las Vegas (Nevada), USA, June 27-30, 1999 (ISBN 1-892512-14-9)

6. M. Azizi, E. -M. Aboulhamid et S. Tahar, "Covérification des systèmes HW/SW basée sur le concept de «multithreading»", Communication au congrès ACFAS'99, Université d'Ottawa, Ottawa, Canada, 10-14 mai 1999 (http://www.acfas.ca/)

7. M. Azizi, O. Ait-Mohamed, and X. Song, "Cache Coherence Protocol Verification of a Multiprocessor System with Shared Memory", Proceedings of the 10th International Conference of Microelectronics (ICM'98), pp. 99-102, Mounastir, Tunisia, December 17-19, 1998 (http://www.ieee.org)

[1] Cette liste couvre presque toutes nos contributions durant la période de notre inscription au doctorat (allant de septembre 1996 à décembre 2000).

8. M. Azizi, E. -M. Aboulhamid, and S. Tahar, “Multi-threading Based Coverification of Embedded Systems”, Workshop on Block Based Design (BBD’98), Ottawa, Canada, October 1998 (http://www.cmc.ca/)

9. M. Azizi, “Cosimulation using Eaglei Environment”, Internal Report, NORTEL Corkstown, Nepean, Canada, March 1998 (5T53/4 : team of Cosimulation & Emulation)

10. M. Azizi et E. -M. Aboulhamid et I. Bennour, “Covérification des systèmes intégrés : Étude et Analyse”, Communication au congrès ACFAS’98, Université Laval, Québec, Canada, mai 1998 (http://www.acfas.ca)

11. M. Azizi, E. -M. Aboulhamid et I. Bennour, “Covérification des systèmes intégrés”, Actes du Colloque International du Traitement d’Image et des Systèmes de Vision Artificielle (TISVA’98), pp. 234-240, Oujda, Maroc, avril 1998

12. M. Azizi, Covérification des systèmes intégrés, Thèse de doctorat, ~180 pages, Université de Montréal, Montréal, Canada (dépôt en décembre 2000)

« **Louange à Allah, Seigneur de l'univers.** », *traduction du Coran (Sourate 1, Verset 2)*

« **Cette grâce vient d'Allah. Et Allah suffit comme Parfait Connaisseur.** », *traduction du Coran (Sourate 4, Verset 70)*

«**Soyez en quête du savoir du berceau à la tombe.** », *traduction d'un hadith du prophète Mohammed (sa)*

À la mémoire de mes chers parents bba Abd Allah et mma Rahma !

À toute ma famille !

À tous les bons de ce monde !

Remerciements

Nous tenons à exprimer notre reconnaissance envers :

- *monsieur El Mostapha Aboulhamid, professeur agréé à l'Université de Montréal et notre directeur de recherche,*

- *monsieur Sofiène Tahar, «associate professor» à la «Concordia University» et notre codirecteur de recherche,*

  tous deux pour leur soutien continu, leurs conseils avisés et la patience dont ils ont dû faire preuve à notre égard.

- *madame Rachida Dssouli (Université de Montréal), monsieur Guy Bois (École Polytechnique de Montréal), monsieur Claude Thibeault (École de Technologie Supérieure de Montréal) et tous les autres membres du jury pour avoir accepté d'examiner cette thèse.*

- *l'ACDI, le GRIAO, le Ministère de l'Enseignement Supérieur du Maroc et la FES de l'Université de Montréal pour leur soutien financier.*

- *monsieur Parviz Youssefpur, manager à NORTEL, pour nous avoir offert un stage au sein des groupes : «5T53 : Methodologies of Design Systems» et «5T54 : Cosimulation & Emulation» à «NORTEL Semi-conductors» (Ottawa), pour la période allant de novembre 97 à mars 98.*

- *messieurs Shailish Saturwala et Imed Bennour, respectivement senior et junior à NORTEL, pour nous avoir supervisé durant notre séjour à NORTEL.*

- *tous nos enseignants depuis la maternelle.*

- *nos sœurs et frères pour leur encouragement permanent.*

- *ma femme pour sa compréhension et son soutien continu.*

- *nos collègues au DIRO (Université de Montréal) et à l'École Supérieure de Technologie d'Oujda (Maroc).*

Chapitre 1

Introduction et Motivations

Depuis quelques années, de nombreuses prédictions sur le futur de l'ordinateur font la une des médias de l'informatique. Ces prédictions, en fait, reposent sur des tentatives récentes de mise au point de nouveaux ordinateurs en faisant appel à des technologies substitutives de la micro-électronique de manière complète ou au moins partielle. Alors, nous entendons de nos jours parler de l'ordinateur optique basé sur la photonique, de l'ordinateur moléculaire basé sur la chimie des molécules, de l'ordinateur biologique basé sur l'exploitation des neurones vivants, de l'ordinateur à base d'ADN et de l'ordinateur quantique basé sur la physique quantique de la matière [8]. Ces différents genres d'ordinateurs sont tous en phase de recherche et leur émergence éventuelle sur le marché public serait tributaire du dilemme performance/coût et d'une compétitivité convaincante avec l'ordinateur micro-électronique. En attendant ce que ramènent les vagues du futur pour savoir davantage sur la relève de la micro-électronique, cette dernière n'a pas encore dit son dernier mot et semble capable de persévérer bien longtemps.

La micro-électronique connaît actuellement une révolution sans pareille sur tous les plans, industriels et académiques. Ceci a donné naissance à des systèmes très complexes aussi bien au niveau de leurs conceptions qu'au niveau de leurs fonctionnements. Pour veiller sur la crédibilité de leurs conceptions, ces systèmes doivent être soumis à des processus de vérification et de test assez rigoureux. Une telle tâche n'est point évidente vu l'accroissement rapide de la complexité et de l'hétérogénéité des « nouveau-nés » micro-électroniques.

En fonction de leurs spécifications et des performances désirées, différentes architectures sont considérées pour réaliser ces systèmes. Notre intérêt de recherche est concentré sur les systèmes dits mixtes[2] ou intégrés. Il s'agit de systèmes à deux dimensions, une logicielle et une autre matérielle. De tels systèmes trouvent leur application dans plusieurs domaines tels que la Télécommunication, l'Aéronautique, la Robotique, l'Automobile, la Médecine, etc.

La conception des composantes logicielle et matérielle d'un système intégré, au cours des décennies passées, s'effectuait de manière séparée. Puis, la partie logicielle obtenue était exécutée sur le prototype de la partie matérielle. Si les contraintes de la spécification requises pour le système ne sont pas satisfaites par le design, le processus est réitéré dans l'espoir de retrouver un bon prototype. Cette technique de vérification des propriétés d'un tel système s'est avérée lourdement coûteuse en termes de temps de réalisation et de coût du système global. Pour remédier à une telle situation, plusieurs compagnies et universités ont axé leurs intérêts sur l'élaboration d'environnements de codesign de meilleures performances. Aussi, ces derniers devront-ils intégrer des outils efficaces et fiables pour accomplir des tâches de covérification. Dans l'objectif d'atteindre un niveau maximum de succès, la covérification doit être exécutée en concurrence avec le codesign; en d'autres mots, il est nécessaire qu'elle soit intégrée au codesign dès ses premières phases.

[2] Nous entendons par système mixte, un système à deux parties logicielle et matérielle. Il ne faut pas faire la confusion avec un système à deux parties analogique et numérique.

La covérification d'un système intégré, étant une nouvelle perception de la vérification de ses deux parties constituantes au fur et à mesure que le design se réalise, revêt un intérêt potentiel dans le monde des circuits intégrés dédiés à des applications spécifiques (ASIC) ou générales (GPIC). L'utilisation de ces derniers est très abondante, mais la maîtrise de leur complexité n'est guère à la portée des concepteurs puisqu'elle se complique davantage à chaque jour. Ce qui explique la demande pressante d'outils puissants pour la conception, la vérification et la réalisation de tels systèmes. Comme remède, de nombreuses tentatives académiques et industrielles ont fait l'objet de plusieurs articles sans réussir toutefois à satisfaire les exigences de qualité et d'efficacité. De notre côté, nous tentons d'apporter une modeste contribution dans ce domaine en proposant un nombre de techniques d'amélioration de la covérification.

A priori, la covérification peut être effectuée par trois techniques principales : « bread-boarding » (c'est-à-dire vérifier après le prototypage), cosimulation et covérification formelle. La première n'est utilisée que peu ou lorsque les autres méthodes échouent (dans le cas d'un système matériel, on parle de « testing »). La troisième technique, la covérification formelle, se chevauche largement avec la vérification formelle et elle nécessite d'être revue et développée. Quant à la deuxième, la cosimulation, elle est souvent utilisée pour les systèmes mixtes à tel point qu'on confond les termes covérification et cosimulation. Elle consiste à simuler simultanément les parties du système, logicielle et matérielle, tout en assurant la communication entre les deux. Nous proposons d'améliorer cette technique par une méthodologie combinant les concepts de base de la simulation, des inspirations de la vérification formelle et des techniques de la programmation orientée objet.

Cadre de la thèse

Cette thèse met l'emphase sur une nouvelle piste de recherche : la covérification des systèmes L/M. La littérature est très riche de travaux sur la vérification des systèmes micro-électroniques à caractère matériel (ASICs, VLSI,...).

Toutefois, à cause de certaines raisons de partitionnement lors de la conception d'un système donné, des fonctions spécifiques sont réalisées par un programme (microcode) et non par un circuit (câblage). Un tel processus de conception est connu sous le nom de « HW/SW-codesign ». Dans cette situation, le système en cours de conception est mixte; il a deux parties : une logicielle et une autre matérielle. Qu'allons nous faire pour vérifier un tel genre de systèmes? Faut-il appliquer les techniques de vérification du matériel en faisant abstraction de la partie logicielle ou vice versa? Ou avant tout, est-ce que les techniques antérieures de vérification formelle sont applicables même sur un système mixte en cours de conception? La majorité des techniques de vérification formelle font souvent appel à la théorie des automates (ou équivalents), qui sert comme une piste de décollage vers une abstraction souvent très floue d'un point de vue pratique. Le concepteur soucieux des devenirs concrets des signaux internes ou externes du système, opte généralement pour une simulation ou un test afin de traquer explicitement les flux de contrôle et de données du système. En outre, dans le cas des systèmes L/M, il n'y a pratiquement aucune méthode (ou outil) formelle pour une telle vérification. Cela est dû à la difficulté de « matérialiser »[3] la partie logicielle afin de profiter des outils et des techniques disponibles de la vérification matérielle. En attendant que le comportement logiciel soit efficacement reproduit par des composants matériels, l'intérêt est maintenant concentré sur la covérification par cosimulation. Notre thèse s'inscrit alors dans ce cadre. Elle dévoile les secrets de cet axe de recherche tout en y effectuant les contributions listées ci-après.

Contributions[4]

Nos principales contributions sont résumées par les points suivants :

i. Identification de la covérification : nous avons classifié les techniques de cosimulation existantes en littérature récente tout en présentant les outils qui leur

[3] C'est-à-dire donner un comportement matériel à la partie logicielle pour en fin de compte manipuler le système mixte (hétérogène) comme un système matériel (homogène).
[4] Voir la liste des contributions, page xviii.

sont associés. En plus, nous avons donné plus de sens au terme « covérification » en expliquant ses objectifs, ses techniques et ses domaines d'application.

ii. Méthodologie de covérification basée sur le «multithreading» : nous avons proposé une méthodologie basée sur le concept du «multithreading» pour effectuer une covérification des systèmes L/M. Elle évalue par cosimulation les propriétés du système étudié. Cette méthodologie se prête aussi bien aux systèmes mixtes (à deux dimensions logicielle et matérielle) qu'aux systèmes non mixtes (à une seule dimension, soit logicielle soit matérielle).

iii. Spécification des propriétés : les propriétés à covérifier sont décrites dans un langage simple que nous avons développé spécialement pour cette fin. Ce langage portant le nom CPL («Coverification Properties Language»), permet à son utilisateur de spécifier les propriétés du système dans une syntaxe similaire à celles des langages de haut niveau (Java par exemple). Les propriétés décrites en CPL devront être automatiquement converties en threads et intégrées au design du système, sans toutefois le modifier.

iv. Cosimulations séquentielle et distribuée : sous une optique de covérification, nous avons réécrit et implémenté avec succès les algorithmes classiques des simulations, séquentielle et distribuée, en utilisant les threads. Les résultats obtenus sont très prometteurs.

v. *Tests orientés par des propriétés :* nous avons tenté de classifier les vecteurs de test à la lumière de leurs activations des propriétés considérées. Un vecteur de test est qualifié de « bon » si son application à l'entrée du système fait activer un bon nombre de propriétés. Dans une première simulation, une séquence aléatoire de vecteurs de test est appliquée au système. Pour chaque vecteur, nous générons un rapport sur l'activation des propriétés. Seuls les bons vecteurs de la séquence courante seront considérés pour les futures simulations. Les bons vecteurs sont soumis à des fonctions de transformation déterministes, aléatoires ou mixtes dans

l'espoir d'améliorer la qualité des vecteurs de test générés. Une fonction d'évaluation s'occupe de la tâche de sélection des bons vecteurs de test.

Plan de la thèse

Notre thèse est organisée de la manière suivante. Le chapitre 1 présente une introduction générale au sujet de cette thèse tout en discutant ses motivations et sa problématique. Le chapitre 2 rapporte les travaux publiés dans les domaines de la conception et de la vérification du matériel. Ce chapitre permet au lecteur de situer le thème de la covérification par rapport à ces axes de recherche, et de comprendre nos propos dans les chapitres suivants. Nous résumons dans le chapitre 3 les principaux techniques et outils contemporains de la cosimulation tout en faisant la liaison avec le codesign et la covérification des systèmes L/M. En plus, nous discutons dans ce chapitre l'ordonnancement des threads en Java et comment ces derniers pourront être utilisés pour faire de la covérification concurrente. Au chapitre 4, nous proposons une méthodologie de covérification basée sur le principe du «multithreading». Cette méthodologie est considérée comme un apport innovateur dans cet axe de recherche. Basée sur des threads, une réimplantation des algorithmes classiques des simulations séquentielle et distribuée est discutée au sein du chapitre 5. Le chapitre 6 explique comment spécifier les propriétés à covérifier selon notre méthodologie. Nous avons défini pour cette fin une syntaxe baptisée CPL. Le chapitre 7 expose deux applications, l'une de vérification et l'autre de covérification. Par-là, nous visons à démontrer le rôle complémentaire de la covérification par rapport à la vérification formelle. Le dernier chapitre conclut notre thèse et suggère des axes de recherche futurs pour donner suite à notre travail.

Chapitre 2

Méthodes de conception et de vérification des circuits intégrés

2.1. Introduction

Les systèmes et les composants micro-électroniques sont de plus en plus intégrés dans différents systèmes dédiés ou au sein d'applications importantes telles que les télécommunications, l'avionique, l'automatique, la robotique, l'appareillage électronique, etc. Certains de ces systèmes micro-électroniques sont très complexes; ils peuvent être perçus comme étant une interconnexion de blocs contenant chacun des millions de transistors. Souvent, lorsque les concepteurs sont en train de concevoir un matériel pour une application spécifique, ils choisissent des composants préconçus, déjà testés et essayés. La réutilisation du design devient de plus en plus populaire puisqu'elle permet de réduire aussi bien le temps de réalisation d'un circuit que son coût. Les tâches de test et de vérification débutent maintenant au niveau comportemental où l'intégration des composants est une importante solution. Le concepteur dispose d'une spécification complète ou partielle du système global ou du composant spécifique qui doit être conçu à partir d'ingrédients micro-électroniques

disponibles. La spécification est réalisée à l'aide du paradigme des langages de description de matériel (HDL); elle peut être exécutée, simulée ou éventuellement soumise à des outils d'évaluation, d'estimation ou de vérification. La spécification est alors supposée être sans fautes et elle peut être utilisée comme un modèle de référence pour vérifier la conformité de toute implémentation par rapport à la spécification originale. Dû à la complexité ascendante des systèmes micro-électroniques récents, de nouveaux axes de recherche ont vu le jour pour combler les lacunes des anciennes techniques. Nous citons à titre d'exemple le codesign L/M, la réutilisation du design (« *design reuse* ») et la vérification formelle.

Afin de situer la covérification par rapport aux autres tâches de conception et rendre ainsi nos propos compréhensibles pour un lecteur non averti, ce chapitre donne un aperçu sur les techniques de conception et de vérification. La section 2.2 traite les différentes étapes du processus de conception. Une vue succincte sur les langages HDL fait l'objet de la section 2.3. Dans la section 2.4, nous exposons très brièvement l'objectif de la synthèse des systèmes numériques. La vérification basée sur la simulation ou le test, est présentée dans la section 2.5; nous y discutons aussi les techniques de simulation, de conception testable et de génération automatique de test, ainsi que les modèles de fautes. La vérification formelle du matériel est rapportée dans la section 2.6. La section 2.7 conclut ce chapitre.

2.2. Processus de conception

Le processus de conception est un ensemble d'actions, ordonnées et bien définies, à exécuter par l'outil ou l'opérateur de conception pour satisfaire une spécification d'un système, des consignes ou des objectifs prédéfinis. Ces actions sont étroitement dépendantes de la nature du système en cours de conception. D'où, il n'y a pas une méthode unique pour concevoir des systèmes différents, cependant la philosophie y demeure globalement la même. Le processus de conception accueille à son entrée la spécification d'un produit et génère à sa sortie l'implémentation de celui-ci. Fournie par le client, la spécification englobe toutes ou quelques consignes que l'implémentation du produit doive satisfaire. L'implémentation peut être

élaborée à n'importe quel niveau d'abstraction dépendamment de là où les paramètres des propriétés de la spécification seront bien explicites et le test d'équivalence de l'implémentation avec la spécification pourrait être accompli sans aucun problème.

Les processus de conception sont classifiés en deux grandes catégories :

- La conception personnalisée (« *custom design* ») : elle est coûteuse et requiert des efforts colossaux, alors son utilisation est limitée à des applications spécifiques telles que la conception des unités de processeurs.
- La conception semi-personnalisée (« *semi-custom design* ») : elle représente avec la méthodologie de la conception réutilisable une bonne alternative de la conception personnalisée. La conception semi-personnalisée peut être répartie en deux types : le design à base de cellules et le design à base de tableaux. Le premier s'effectue en utilisant des cellules de la librairie (des cellules); quant au second, il utilise des modules MPGAs et FPGAs [9].

Généralement, le processus de conception comprend les étapes suivantes :

- *Modélisation* : Partant d'une spécification dans un langage naturel ou d'une description écrite dans un langage de spécification, un modèle HDL du système est établi. Un tel modèle doit être validé avant de procéder à l'étape suivante. Par ailleurs, les langages VHDL et Verilog sont actuellement les plus populaires des langages de description du matériel.
- *Synthèse* : Le modèle HDL obtenu est optimisé à la lumière de certaines contraintes telles que le temps d'exécution, la surface du circuit et son coût. Pour ce faire, des outils de synthèse permettant de faire l'optimisation nécessaire et de générer le schéma du circuit, sont largement utilisés (Synopsys, Cadence, etc.).
- *Validation* : Cette étape finale vérifie si le modèle implémenté reproduit exactement la fonctionnalité du modèle spécifié (modèle de référence). Après l'étape de validation, viennent les phases de mise en prototype et de fabrication.

2.3. Langages de description HDL et modélisation

Un modèle reproduit, selon ses raisons d'être, certaines caractéristiques du système original. Il peut être élaboré à n'importe quel niveau d'abstraction et sous un format textuel (code), graphique (graphe) ou mixte.

2.3.1. Niveaux d'abstraction

L'abstraction est un concept basé sur le fait d'ignorer ou de supprimer dans un modèle certains détails du système original afin de simplifier sa taille (du modèle), d'en déduire des conclusions d'ordre général et aussi pour réduire le volume du travail requis. On distingue plusieurs niveaux d'abstraction :

- *Niveau système* : on essaye à ce niveau d'identifier l'architecture globale du système en spécifiant ses principaux éléments et leurs communications bilatérales et avec l'environnement. À ce niveau, on ne s'intéresse pas aux détails des composants du système, une vue globale suffit.
- *Niveau comportemental* : la fonctionnalité d'un composant matériel à ce niveau est décrite depuis ses ports d'entrée jusqu'à ses ports de sortie. Le « timing » n'est pas requis dans le modèle; cependant, il peut y être considéré afin de servir de repère pour certaines exécutions.
- *Niveau transfert de registres (RTL)* : à ce niveau, on suppose que le système est synchrone. Des contraintes sur le « timing » et les ressources du système sont définies. Le comportement de chaque composant est donc décrit pour chaque cycle d'horloge. Les opérations des flux de contrôle et de données sont accomplies à l'aide des registres.
- *Niveau porte* : le système à ce niveau est globalement vu comme étant un circuit séquentiel. Ce dernier pourrait comprendre des bascules, des loquets et des portes logiques. Un tel modèle peut être dérivé d'une description RTL tout en faisant usage des librairies des différentes technologies (TTL, CMOS, etc.). Ces dernières, créées sous une optique de réutilisation, regroupent des modèles « clé en main » de bascules et de portes.

- *Niveau transistor et mise en carte* : à ce niveau, le système paraît comme étant une carte portant plusieurs composants actifs (transistors) et passifs (résistances, capacités, etc.). Une optimisation des emplacements des composants sur la surface offerte s'en suit.

2.3.2. Langages standards de description de matériel (HDL)

L'objectif de ces langages est de faciliter les tâches de modélisation des composants numériques et d'en élaborer une documentation exécutable. La maintenance du code HDL et l'insertion de nouveaux composants au sein d'un design existant, sont devenues relativement faciles et requièrent sensiblement moins de temps. Les HDLs sont indépendants de la technologie, cependant ils permettent d'adapter un même modèle à différentes technologies. Les HDLs les plus fréquemment utilisés sont Verilog et VHDL.

A. Verilog

Verilog [10, 11] est un langage de description de matériel dédié en premier lieu à la modélisation et à la simulation des circuits logiques avec un degré d'efficacité satisfaisant. Il est conçu à base du langage C. En comparaison avec VHDL, Verilog est un langage faiblement typé. Il permet d'implémenter des exécutions séquentielles et parallèles. Un composant en Verilog est décrit par l'objet *module* (figure 2.1) dont il faut déclarer les entrées-sorties et spécifier la fonction. Nous illustrons ci-après sur la figure 2.2 une structure typique d'un programme Verilog.

Un module peut aussi faire appel aux services d'autres modules en les instantiant dans son corps. Des sous-programmes, tels que *function* et *task*, sont déclarés à l'extérieur du module où ils sont appelés à la manière de C.

```
module logical_adder (I1, I2, O);
input I1, I2;
output O;
reg O;
always  (I1 or I2)
begin
if  (I1) O=1 else if (I2) O=1 else O=0;
end
endmodule
```

Figure 2.1 : Additionneur logique

```
module  name_component  (inputs and outputs list);
input …, …, …; // list of inputs
output …, …, …; // list of outputs
reg  …, …, …; // list of registers
wire …, …, …; // list of wire parameters
initial     // block of initialization
begin
…          // instructions of initialization to be run once
end

always  // block that are always running
begin
…           // instructions of always block
end
endmodule // the end of module name_component

task name_task  // declaration of procedure (void function in C)
begin
… // task body
end

function name_function // declaration of function (type function in C)
begin
… // function body
end
```

Figure 2.2 : Structure générale d'un programme en Verilog

B. VHDL

VHDL («VHSIC Hardware Description Language») [12-26] est un langage de description de matériel selon la norme IEEE. Il est généralement utilisé pour modéliser, à différents niveaux d'abstraction, des composants et des systèmes numériques tels que des puces, des interfaces, des cartes, etc. Étant un standard IEEE, VHDL vient renforcer le développement, la simulation, la vérification, la synthèse et le test des designs matériels en permettant une description commune et compatible pour tous leurs outils respectifs.

Fortement typé, VHDL est conçu à base d'Ada. C'est un langage très puissant grâce à ses qualités de programmation moderne, à ses capacités d'implémenter des designs à grande échelle et à son aptitude à réutiliser des anciens designs. Dans la littérature, il y a suffisamment d'ouvrages récents dont [12-26], qui examinent la modélisation, la simulation et la synthèse à l'aide de du langage VHDL. Dans ce qui suit, nous présentons sommairement l'essentiel de ce langage.

La plus importante caractéristique du langage VHDL est sa faculté de séparer la spécification de l'interface (*entity*) de celle de son corps (*architecture*). Ceci permet à une seule interface d'avoir plusieurs implémentations de son corps. Le code VHDL pourrait être donc développé de manière modulaire. Aussi, de nouvelles unités élémentaires conçues en VHDL peuvent-elles enrichir la librairie de VHDL en les stockant dans des modules connus sous le nom de *packages*. Ces derniers sont éventuellement réutilisés dans des designs ultérieurs. Voici les différentes composantes d'un design en VHDL :

- *Entity :* Elle décrit l'interface d'un composant, toutefois elle est indépendante de son implémentation. Dans cette unité, sont déclarés tous les ports d'entrée et de sortie d'où communique le composant avec son environnement.
- *Architecture :* Elle décrit une implémentation possible d'une entité déclarée. Cette architecture peut être réalisée selon trois styles différents : architectures structurelle,

comportementale et mixte. Le style comportemental comprend aussi deux sous-styles : algorithmique et orienté flux de données.

- *Configuration :* Elle relie les références de l'interface locale (entity) et de ses architectures aux unités préconçues de la librairie VHDL (prédéfinie ou enrichie par l'utilisateur). Plusieurs architectures pourront être associées à une même interface, alors la configuration s'avère indispensable pour identifier laquelle d'entre elles il faut utiliser lorsque l'interface est instantiée.

- *Package :* Il englobe des informations prédéfinies communes à plusieurs unités de design, telles que des composants préconçus, des fonctions, des types, des signaux, des constantes, etc. Un package en VHDL, comme en d'autres langages, cache les détails aux utilisateurs et permet la réutilisation du code VHDL.

2.4. Synthèse

La synthèse est définie comme étant le processus de raffinage partant d'un modèle abstrait jusqu'à un modèle plus détaillé de plus bas niveau d'abstraction. Le raffinage prend en considération toutes les contraintes, que ce soit celles formulées par l'utilisateur ou celles émanant des règles de conception. Durant ce processus, des paramètres de performance sont optimisés pour satisfaire les contraintes mises en jeu. Parmi ces paramètres, nous en citons : la capacité de traitement (« *throuput rate* »), la latence (« *latency* »), le cycle de l'horloge (« *clock cycle* »), la surface du circuit (« *circuit area* »), la puissance (« *power* »), la testabilité, et ainsi de suite. On distingue dans la synthèse trois types [17] : la synthèse comportementale, la synthèse logique et la synthèse au niveau circuit.

2.5. Vérification par simulation et test

La simulation est l'approche la plus répandue pour la vérification des systèmes numériques. Ce type de vérification est basé sur un modèle HDL du système étudié. Ce modèle décrit le comportement global ou partiel du système. Il est stimulé par une série de vecteurs de test et les résultats obtenus (à ses sorties) sont comparés avec des valeurs prévues dans les mêmes conditions de la manipulation.

L'avantage majeur de la vérification par simulation apparaît dans le fait qu'elle ne nécessite pas un prototype matériel et qu'elle est souvent implémentée par les outils de conception assistée par ordinateur (« CAD tools »). Cependant, cette méthodologie est gourmande en terme de temps. En outre, elle requiert une génération de stimuli, les appliquer à l'entrée du système et ensuite en entamer la tâche de vérification.

La communauté du test a défini les niveaux d'abstraction suivants :

- Le *niveau système,* où le système numérique est présenté comme des unités échangeant des messages entre elles (chaque unité est une sorte de boîte noire dont le comportement est décrit par ses entrées-sorties).
- Le *niveau processeur*, où le système est vu comme un ensemble de programmes et de structures de données.
- Le *niveau instruction*, où le système est considéré comme une unité capable d'exécuter un ensemble d'instructions ayant un effet sur un nombre de registres et de ports.
- Le *niveau registre*, où le système est perçu comme étant composé de deux parties : la partie de contrôle qui peut être décrite par une FSM ou par une structure logique aléatoire; la partie du chemin de données est constituée des unités fonctionnelles et des registres.
- Le *niveau logique*, où le système est vu comme une interconnexion de portes logiques et de bascules.

À chaque niveau d'abstraction, des suites de test sont éventuellement générées et différents aspects du système peuvent être simulés. Le modèle peut être compilé en un code objet qui est très efficace mais de moindre flexibilité que la simulation à événements. Actuellement, les environnements CAD utilisent une approche mixte qui fait un compromis entre l'efficacité et la flexibilité.

La simulation peut être aussi effectuée en étapes successives : Dans une première étape, la fonctionnalité du système est vérifiée sans se préoccuper des caractéristiques temporelles du design. Une fois que cette étape est accomplie et qu'un modèle structurel est automatiquement ou manuellement dérivé, les contraintes temporelles sont vérifiées. Différents modèles de « timing » sont

éventuellement utilisés. Le modèle « unit-delay » où chaque porte logique élémentaire fait un délai d'une unité de temps, est le modèle le plus simple et le plus rapide en terme de temps de simulation. Il donne une idée primaire sur la performance du système simulé. Il existe d'autres modèles plus élaborés qui tiennent compte aussi des raccords d'interconnexion et considèrent des délais plus exacts pour les portes logiques. De tels modèles sont plus précis en terme de performance globale du système, toutefois leur usage requiert des librairies spécifiques des différentes technologies et des environnements CAD sophistiqués.

2.5.1. Modélisation des fautes

Un fonctionnement anormal observable d'un système est appelé une *erreur*. Une erreur est une conséquence d'une *faute* dans le système. Les fautes peuvent résulter des causes suivantes :

- *erreurs de conception* telles que des spécifications incomplètes, des contradictions entre les différents niveaux d'abstraction ou des violations des règles de conception.
- *erreurs de fabrication* telles que des connexions incorrectes ou des composants non convenables.
- *défauts de fabrication* lors du processus de fabrication.
- *défaillances physiques* résultant du vieillissement des composants et des facteurs environnementaux.

Les fautes peuvent être permanentes, intermittentes ou transitoires. Il est très difficile d'examiner toutes ces sortes de fautes dans un modèle abstrait du système sous test; par exemple, si nous avons un modèle au niveau porte logique, alors il est très compliqué voire impossible de modéliser des erreurs d'alignement de la gaufre (« *mask alignment error* »). Les *fautes logiques* représentent des effets des fautes physiques. L'ensemble des fautes logiques possibles est appelé un *modèle de fautes*.

Le modèle des fautes suppose que, dans un circuit combinatoire, chaque ligne peut être « collée » de façon permanente à une des valeurs logiques 0 ou 1. On dit que les fautes sont de types collé-0 ou collé-1. Si un circuit donné possède n lignes, chacune d'elles peut soit fonctionner correctement, soit être collée-0, soit être collée-

1, alors le nombre de tous les cas possibles à considérer est 3^n-1 [33]. Dans le cas où seulement une ligne serait responsable d'induire le circuit en faute, le modèle est dit « collé à une seule faute », autrement il est dit « collé à fautes multiples ». Détecter une faute *f* dans un circuit combinatoire est un problème NP-complet [34, 35, 36]. Le modèle « collé à une seule faute » est le plus utilisé grâce au fait qu'il est simple à manipuler et indépendant des technologies. Ce modèle a été étendu aux niveaux élevés d'abstraction comme le RTL. D'autres modèles de fautes ont été proposés, tels que les fautes du pont et les fautes du délai [33].

2.5.2. Techniques de simulation des fautes

L'objectif de la simulation des fautes est d'évaluer efficacement la couverture d'une séquence de test. La couverture peut être définie comme le pourcentage de fautes couvertes par la séquence, étant donné un modèle de faute.

La simulation des fautes, de manière séquentielle, transforme le modèle du circuit sans fautes *C* pour représenter le circuit fautif C_f résultant d'une faute *f*. Avec cette technique, une seule faute est simulée à la fois, d'où le processus doit être répété pour chaque faute; ceci le rend inapplicable pour des circuits réels. La simulation des fautes parallèles simule simultanément le circuit bon et un nombre de circuits fautifs. Les réponses de tous ces circuits aux même vecteurs de test sont emmagasinées dans des cases-mémoires contiguës. La simulation des fautes déductives, basée sur la simulation du bon circuit, déduit le comportement de tous les circuits fautifs théoriques; seul un sous-ensemble de ces derniers est examiné à cause de la limitation de mémoire. La simulation des fautes concurrentes observe si des résultats des circuits fautifs concordent ou pas avec ceux du bon circuit (sain); et elle continue l'analyse juste pour les circuits dont les sorties sont incorrectes [33].

2.5.3. Génération automatique des patrons de test

La génération automatique de tests est une importante tâche dans le flux de conception. Elle est très difficile parce qu'elle est un problème NP-dur même pour les circuits combinatoires sous le modèle « collé à une seule faute ». Les outils automatiques s'intéressent aussi bien aux circuits combinatoires qu'aux parties combinatoires des circuits séquentiels. Les techniques de la conception prévoyant la testabilité, comme nous allons le voir dans la section 2.5.5, réduisent le problème de test des circuits séquentiels à un problème de test des circuits combinatoires. Comme il a été mentionné précédemment, des techniques de moindre coût telles que la génération aléatoire de test suivie de la simulation de fautes, peuvent être utilisées en première étape, ensuite viennent les méthodes déterministes pour élargir la couverture. Parmi les méthodes déterministes les plus connues dans le domaine de la génération de test pour les circuits combinatoires sous un modèle « collé à une seule faute », nous citons (à titre d'exemple) le D-algorithme [37], PODEM (« Path-Oriented Decision Making ») [38] et FAN (« FANout-Oriented Test Generation ») [39].

2.5.4. Test fonctionnel

Un test fonctionnel teste une implémentation contre sa spécification fonctionnelle. C'est un test de conformité entre l'implémentation et sa spécification. Il est fréquemment appliqué aux niveaux d'abstraction plus élevés que le niveau porte. Le diagnostic, processus de détection et de localisation des erreurs, est plus important à ces niveaux parce que les erreurs sont de type *erreurs de conception* qui doivent être corrigées avant de procéder à la suite.

Les techniques du test exhaustives ne sont rien d'autre qu'une illustration du test fonctionnel appliqué aux petits circuits combinatoires et séquentiels. Les techniques du test pseudo – exhaustives sont applicables aux circuits qui peuvent être répartis en blocs caractérisés chacun par ses entrées-sorties; ces blocs doivent être de petite taille pour pouvoir les tester exhaustivement. Les circuits logiques itératifs organisés en table (ILAs) sont les candidats parfaits pour le cas du test pseudo-

exhaustif; ils peuvent être perçus comme étant une interconnexion régulière de cellules identiques (par exemple, un additionneur *ripple-carry*) [33].

2.5.5. Conception prévoyant la testabilité (DFT)

La testabilité est une mesure de la facilité avec laquelle on peut tester un circuit ou un système. Elle prend en compte le coût de la génération des suites de test et leur application au système à tester. Le coût et le temps associés au test peuvent souvent dépasser ceux des autres tâches requises durant l'accomplissement de la conception. L'objectif principal du DFT consiste en la réduction de ces paramètres. En résumé, la testabilité d'un circuit est influencée par trois facteurs :

- *La contrôlabilité*, elle mesure le degré de facilité d'affecter une valeur spécifique à n'importe quel nœud dans le système.
- *L'observabilité*, elle mesure le degré de facilité de déduire la valeur de n'importe quel nœud du système juste en observant ses sorties.
- *La prédictibilité*, elle mesure le degré de facilité d'obtenir la valeur d'une sortie en réponse à une application éventuelle d'une valeur donnée à l'entrée du système.

Plusieurs techniques ad hoc de DFT ont été développées et elles sont bien résumées dans la référence [33]; ces techniques en général tentent d'atteindre les objectifs suivants :

- Améliorer l'observabilité et la contrôlabilité en rendant possible l'accès direct aux points d'observation (sorties et autres) et aux points de contrôle (entrées et autres).
- Faciliter l'initialisation des circuits en prévoyant une entrée *RESET* par exemple.
- Permettre la désactivation des horloges internes durant la campagne de test.
- Faciliter la répartition d'un circuit en plusieurs sous-circuits lors de la génération de test.
- Éviter l'usage de la logique redondante.
- Développer ou fournir une logique capable de casser les boucles de feedback.

La technique de conception basée sur le test (« scan-based design ») constitue l'une des techniques DFT les plus réussies parce qu'elle possède de bonnes caractéristiques en termes de temps de test, de borne supérieure d'aire, et de pattes I/O réservées au test. Elle est basée sur l'usage des registres de « scan » capables d'effectuer des opérations de décalage et de chargement parallèle [33]. LSSD appartient à cette classe de techniques; elle est utilisée dans des architectures IBM [40]. Des standards de DFT ont été développés aussi pour servir au test au niveau des cartes (« board level »); nous mentionnons à titre d'exemple le « JTAG Boundary Scan standard » [41, 42].

Nous concluons cette section en citant les techniques de compression qui consistent à compresser la réponse du circuit dans ce qu'on appelle une signature, et l'autotest intégré (BIST) qui appartient à une autre classe de techniques; pour avoir plus d'explications sur ces techniques, le lecteur est invité à consulter la référence [33].

2.6. Vérification formelle du matériel

La vérification est une activité de conception très importante. Comme le nombre de portes croit très rapidement et la complexité des circuits intègres devient de plus en plus exigeante, les défis à concevoir un circuit intégré sont sans précédents. Les approches de vérification classiques (simulation et test) ne sont pas très efficaces pour des systèmes complexes. C'est très coûteux, voire impossible, de tester ou de simuler toutes les entrées possibles d'un système dont le nombre d'entrées est excessivement grand. La simulation et le test ne garantissent pas un design sans erreurs, d'autant plus si le design fait intervenir des boîtes noires telles que les blocs à propriétés intellectuelles (« Bloc-based design (BBD) » ou « Intellectual Property Blocs (IPs) »).

Les techniques de la vérification formelle [36, 43] cherche à démontrer si un design est correct ou non sans avoir besoin d'une simulation exhaustive. La vérification formelle d'un système se fait à la manière des preuves mathématiques des théorèmes, sans se soucier des valeurs des entrées.

Une implémentation représente le design du système qui devrait être vérifié afin de savoir s'il (le design) correspond ou pas à sa description initiale

(spécification). La spécification représente les propriétés auxquelles se réfère le processus de vérification pour conclure de la conformité d'un système. Elle peut être exprimée de différentes manières telles qu'une description comportementale, une description structurelle abstraite ou un ensemble de propriétés temporelles. Du fait que le raisonnement est formel, un formalisme est indispensable pour exprimer les trois entités : l'implémentation, la spécification et la relation entre elles. Les représentations d'une implémentation peuvent être des réseaux de transistors/portes, des automates à états finis, une description en logique, et ainsi de suite [17]. Les formalismes utilisés pour représenter des spécifications sont classifiés en (*i*) automates de logique et (*ii*) théorie des langages. La logique en question inclut la logique propositionnelle, la logique des prédicats de premier ordre, la logique d'ordre élevé et la logique modale (la logique temporelle, etc.) [19, 44].

Les méthodes de la vérification formelle peuvent être classées en deux grandes catégories : (i) la vérification interactive utilisant des 'prouveurs' de théorèmes (« theorem-prover ») et (ii) la vérification basée sur l'énumération des états des FSMs.

2.6.1. « Model-Checking »

Les méthodes « model-checking » et « equivalence checking » font partie de la catégorie de vérification basée sur des machines à états finis (FSM). Elles requièrent une exploration exhaustive de l'espace des états du modèle. Malheureusement, cette exploration provoque, dans le cas des systèmes de grande taille, un problème d'explosion d'états. Ceci est donc une limitation de l'usage de ces méthodes. Pour en remédier, des tentatives d'amélioration ont été proposées dans les références suivantes [45-50]. Pour alléger le problème d'explosion, McMillan propose dans [45] de ne pas représenter explicitement les états du modèle; le groupe de Bryant a proposé le ROBDD (« Reduced Ordered Binary Decision Diagrams ») [46] pour ordonner et réduire les digrammes de décision binaires. Cette technique est connue sous le nom de « symbolic model checking » du fait que des variables symboliques sont utilisées pour représenter les états du système au lieu des variables numériques. Bien que l'emploi du ROBDD pour explorer implicitement l'espace des

états, ait élargi le domaine d'application de l'approche « model checking », cette dernière demeure inadéquate pour la vérification des circuits à grand chemin de données. Ceci est dû au fait que les ROBDDs sont considérés au niveau logique booléen, ce qui veut dire qu'une variable individuelle est nécessaire pour chaque bit d'une donnée; donc, le problème d'explosion d'états persiste encore pour les circuits à grand chemin de données. Par contre les ROBDDs se prêtent bien à la vérification des chemins de contrôle. Alors, des méthodes plus efficaces doivent être développées pour vérifier aussi bien les chemins de données que leurs interactions avec les chemins de contrôle d'un design. La technique de MDG (« Multiway Decision Graphs ») [47, 48] est proposée dans ce cadre, cependant son utilisation reste très limitée.

La technique de « model checking » a été appliquée pour vérifier automatiquement des designs de complexité significative tels que la vérification du protocole de cohérence des caches d'un système multiprocesseur [45, 51].

L'avantage le plus important du « model checking » est que cette technique peut être complètement automatique. Aussi, se prête-t-elle bien à la vérification du contrôle des systèmes. Cependant, son inconvénient est qu'elle est plus restrictive en application et faible en vérification hiérarchique en comparaison avec le « theorem-proving ».

2.6.2. « Theorem-Proving »

L'approche la plus générale de vérification est de formuler les conditions de conformité d'un système comme des théorèmes à démontrer, et de générer ensuite mécaniquement des preuves pour ces théorèmes en utilisant un outil dédié à ce genre de tâches. La technique de « theorem-proving » utilise des formalismes puissants tels que la logique d'ordre élevé [52], qui permettent de postuler le problème de vérification à différents niveaux d'abstraction. Cette approche a connu des succès significatifs dans la vérification des designs des microprocesseurs [44, 53]. Cependant, la vérification basée sur cette technique est moins automatique [52, 54].

De nos jours, la majorité des systèmes de «theorem-proving » sont semi-automatiques et exigent plus d'efforts du côté de l'utilisateur afin de développer des

spécifications pour chaque composant et à superviser le processus d'inférence. D'où, seuls les experts et les avertis sont capables d'utiliser cette technique sans trop de peine.

2.7. Conclusion

Nous avons présenté au sein de ce chapitre un aperçu global sur les techniques de conception, telles que la modélisation, la synthèse, la simulation, le test et la vérification formelle. Pour surmonter les limitations et faire face aux complexités excessivement ascendantes des systèmes, de nouvelles approches combinant différentes méthodes classiques surgissent dans le champ de la micro-électronique; le codesign L/M, la réutilisation de design, le développement des modèles et des blocs matériels à propriétés intellectuelles, la coverification L/M et la révision des méthodes traditionnelles semblent être le point de visée auquel vont concourir les intérêts futurs dans le domaine de conception et de vérification des systèmes numériques.

Chapitre 3

Introduction au codesign et à la covérification des systèmes intégrés

3.1. Introduction

La covérification L/M d'un système intégré est une nouvelle perception de la vérification de ses deux parties constituantes au fur et à mesure que sa conception se réalise. Elle revêt un intérêt potentiel dans le monde des circuits intégrés dédiés à des applications spécifiques (ASICs). L'utilisation de ces derniers est très abondante mais la maîtrise de leur complexité se complique davantage de jour en jour, ce qui explique la demande pressante pour le développement d'outils puissants pour la conception, la vérification et la réalisation de tels systèmes. De nombreuses tentatives académiques et industrielles n'arrivant pas à satisfaire les exigences de qualité et d'efficacité, ont fait l'objet des articles suivants [36, 51, 61-71].

Le reste de ce chapitre est organisé comme suit : à la section 3.2, nous exposons le problème de la covérification d'une manière générale, puis à la section 3.3, nous examinons brièvement le processus du codesign L/M. Nous faisons un survol des principales techniques de covérification dans la section 3.4 tout en

discutant l'applicabilité des méthodes formelles à la covérification. La section 3.5 présente des techniques de cosimulation. Nous décrivons brièvement à la section 3.6 quelques environnements dédiés au codesign et à la covérification. Nous discutons dans la section 3.7 comment faire de la covérification ordonnancée à l'aide des threads. La section 3.8 conclut ce chapitre.

3.2. Motivation et position du problème

3.2.1. Terminologie

- Un système intégré : c'est un système formé d'une composante logicielle et d'une autre matérielle. C'est un assemblage d'unités de traitement (processeurs), d'ASICs et de mémoires d'une part et d'un programme pilote supervisant le fonctionnement du système global d'autre part. Un tel système est dit interactif s'il interagit avec son environnement via des capteurs et des actionneurs (figure 3.1).

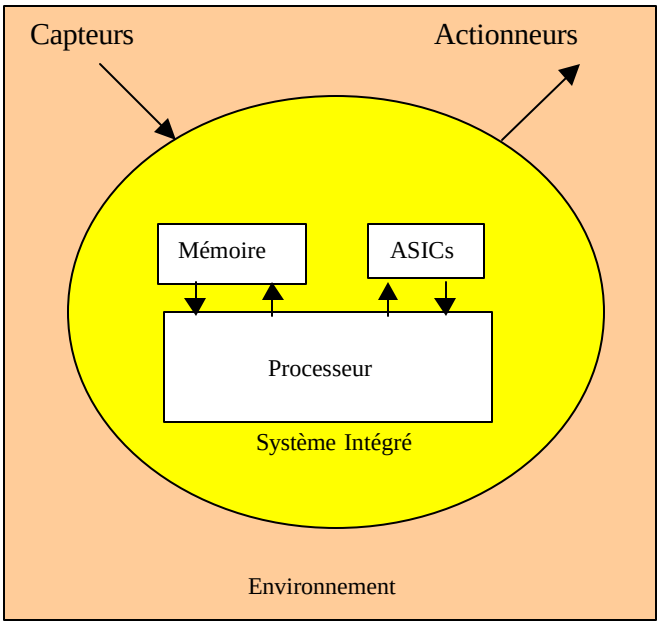


Figure 3.1 : Un système intégré

- Le codesign d'un système intégré (ou codesign L/M) : C'est la conception en

parallèle des deux composantes, logicielle et matérielle, du système en vue de faciliter la vérification de leur compatibilité et accélérer le prototypage.

- La covérification d'un système intégré : C'est la vérification simultanée et parallèle de ses deux sous-systèmes : matériel et logiciel. Pour augmenter les chances de succès d'un codesign, la covérification doit être mise en œuvre dès ses premiers pas (codesign).
- La covérification formelle : Nous avons introduit ce nouveau terme pour signifier la chose suivante : l'ensemble de toutes les preuves mathématiques utilisées pour vérifier la validité d'un codesign L/M.
- La cosimulation : C'est le fait d'élaborer, à partir de la spécification d'un système intégré, un code compilé pour sa partie logicielle et une description HDL pour sa partie matérielle.
- Une propriété : C'est une caractéristique extraite de la spécification du système et que l'implémentation doit satisfaire à tout prix. Elle est représentée par une expression logique ou une condition dont il faut vérifier la validité. Pour ce faire, elle doit être écrite dans un format vérifiable au moyen d'un langage spécifique.

3.2.2. Position du problème

Face aux besoins pressants et croissants de vérifier le design d'un système L/M avant son prototypage, les concepteurs et les développeurs ont été convaincus de la nécessité d'une élaboration d'un environnement de conception commun pour les deux parties logicielle et matérielle. De nombreuses tentatives visant un tel objectif ont été publiées dans des articles tels que [67, 71-76]. En résumé, tous les auteurs de ces articles proposent d'écrire la composante logicielle dans un langage de programmation de haut niveau (tel que C), la partie matérielle dans un langage de description de matériel (tels que VHDL ou Verilog) et établissent des unités ou des modules pour assurer la communication ou l'interfaçage entre les deux parties (figure 3.2).

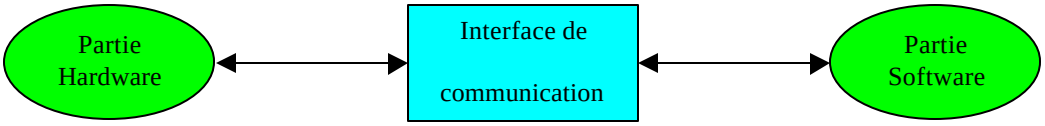


Figure 3.2 : Représentation globale d'un système intégré

Traditionnellement, la covérification d'un système conçu est accomplie après sa fabrication, c'est-à-dire une fois que le prototype du système a été réalisé, son logiciel est exécuté sur lui. Et à ce stade, la propriété «être correcte » est décidée éventuellement « vraie » s'il n'y a aucune détection d'erreurs, autrement le composant est recyclé. L'attente de la mise en prototype du système afin de vérifier sa fonctionnalité globale, constitue un grand risque et peut probablement causer d'énormes pertes pécuniaires et de temps. Donc, il s'est avéré indispensable d'introduire le processus de covérification dès les premières étapes de la conception.

Un environnement universel de covérification performant devrait posséder, entre autres, les caractéristiques suivantes [70] :

- Exécution du code en temps réel.
- Manipulation de tout type de processeurs.
- Capacités robustes de « debugging ».
- Temps de réitération rapide.
- Information exacte du « timing ».
- Prédiction exacte du comportement du système.
- Pas de collisions de communication.
- Fiabilité.

Pour simuler les composantes logicielles et matérielles d'un système intégré, les concepteurs devront élaborer des modèles relatifs à tous les composants et à leur intercommunication. En d'autres mots, ces modèles ont à décrire le comportement des différentes parties du système par l'énumération de tous ses états de fonctionnement et de leurs transitions. Étant donné un système intégré, faut-il aborder sa coverification par simulation ou par méthodes formelles? Par voie

formelle, un système intégré *S* est décrit par un graphe d'états *G* = < *GS, GH, GC*> où *GS, GH* et *GC* sont respectivement les sous-graphes des parties fondamentales logicielle et matérielle, et du module de leur communication. Comment extraire, de la spécification globale, un ensemble de propriétés à vérifier sur une représentation adéquate de *S* et élaborer des algorithmes d'analyse d'accessibilité plus efficaces en termes de temps et d'espace mémoire? Une telle tâche est non évidente à cause de la grandeur et de la complexité de ce type de systèmes; déjà les méthodes formelles de vérification du matériel souffrent de certains inconvénients tels que le problème d'explosion des états, alors le handicap de ces méthodes serait davantage apparent dans le cas de systèmes plus complexes dont les systèmes intégrés. Par voie non formelle, le design global est soumis au processus de simulation. Un ensemble de vecteurs de test est appliqué à l'entrée du système et les résultats recueillis à sa sortie sont comparés à d'autres prévus afin de conclure sur certaines de ses propriétés. Cette méthodologie a des problèmes de temps d'exécution et de choix de vecteurs de test, et en plus la décision de vérification n'est souvent pas absolue. Nous proposons dans cette thèse une voie de vérification mixte combinant des ingrédients des deux méthodologies : de simulation et de vérification formelle.

3.3. Codesign L/M

Le codesign est le développement simultané de composants matériels et logiciels pour obtenir le design complet d'un système. Ce processus comprend les tâches de spécification, de conception, de synthèse et de validation des systèmes mixtes L/M. Entre autres, la méthodologie du codesign vient répondre à la complexité ascendante des systèmes soumis au processus de conception, ainsi qu'au besoin de mettre au point un prototype précoce pour valider la spécification et fournir à la clientèle un « feedback » durant le processus de conception en générant une architecture abstraite autour de processeurs inter-communicants qui implémentent la spécification initiale. Cette architecture peut être raffinée davantage afin de produire finalement une architecture hétérogène plus complète, englobant des modules matériels, logiciels et de communication.

La majorité des systèmes intégrés récemment conçus possèdent de telles architectures hétérogènes et ils sont composés d'entités standards et personnalisées (« *custom* »), aussi bien matérielles que logicielles. Souvent, le partitionnement L/M est effectué manuellement par le concepteur du système. Cependant, l'accroissement hâtif des pressions sur la réduction du temps de réalisation, de la complexité des systèmes à concevoir et du volume du logiciel dans un système intégré, a poussé les concepteurs à développer des processus semi-automatiques de partitionnement L/M et des outils d'estimation de performance. En général, les équipes de conception du logiciel et du matériel utilisaient séparément des outils dédiés pour concevoir les parties logicielle et matérielle, devant œuvrer au sein d'un même système. C'est à la fin du cycle de conception que les deux parties sont mises ensemble, et ainsi qu'une évaluation du système global pour diagnostiquer et corriger des défaillances éventuelles est possible. Alors pour réduire le temps et découvrir des incompatibilités entre le matériel et le logiciel avant la fin de la conception, une mise en communication des deux environnements du matériel et du logiciel est fortement requise. C'est pour cette raison qu'aujourd'hui des méthodes courantes essayent de retarder autant que possible le partitionnement et d'assurer donc en permanence, même après le partitionnement, une communication entre les outils de développement des deux parties (ou entre les deux équipes de matériel et de logiciel). Les figures 3.3 et 3.4 illustrent deux processus possibles de codesign proposés en littérature [27, 28]. Les étapes du flux de codesign présentées par ces figures se résument par ce qui suit :

- *Saisie de la spécification* : il s'agit de la génération du modèle et de la simulation de ce dernier.
- *Validation de la spécification* : elle est effectuée par des techniques de simulation ou de vérification.
- *Exploration* : plusieurs alternatives de codesign sont explorées afin d'en extraire la meilleure qui satisfasse les contraintes prédéfinies. Ces alternatives sont classées selon certains critères tels que la performance et le coût.
- *Raffinage de la spécification* : à la lumière des résultats obtenus lors de l'étape d'exploration, la spécification initiale est raffinée en vue d'obtenir une nouvelle

description optimisée. La vérification de l'équivalence entre cette dernière et la spécification initiale est accomplie par des techniques de cosimulation.

- *Implémentation L/M* : le système est partitionné en blocs matériels et logiciels décrits au niveau fonctionnel. Les blocs matériels sont soumis aux outils de synthèse; quant aux blocs logiciels, ils sont développés en utilisant les techniques du génie logiciel.

Bien que le codesign soit récent, il fait déjà l'objet de plusieurs ouvrages et publications dont nous citons les références [3, 13, 18, 27-32].

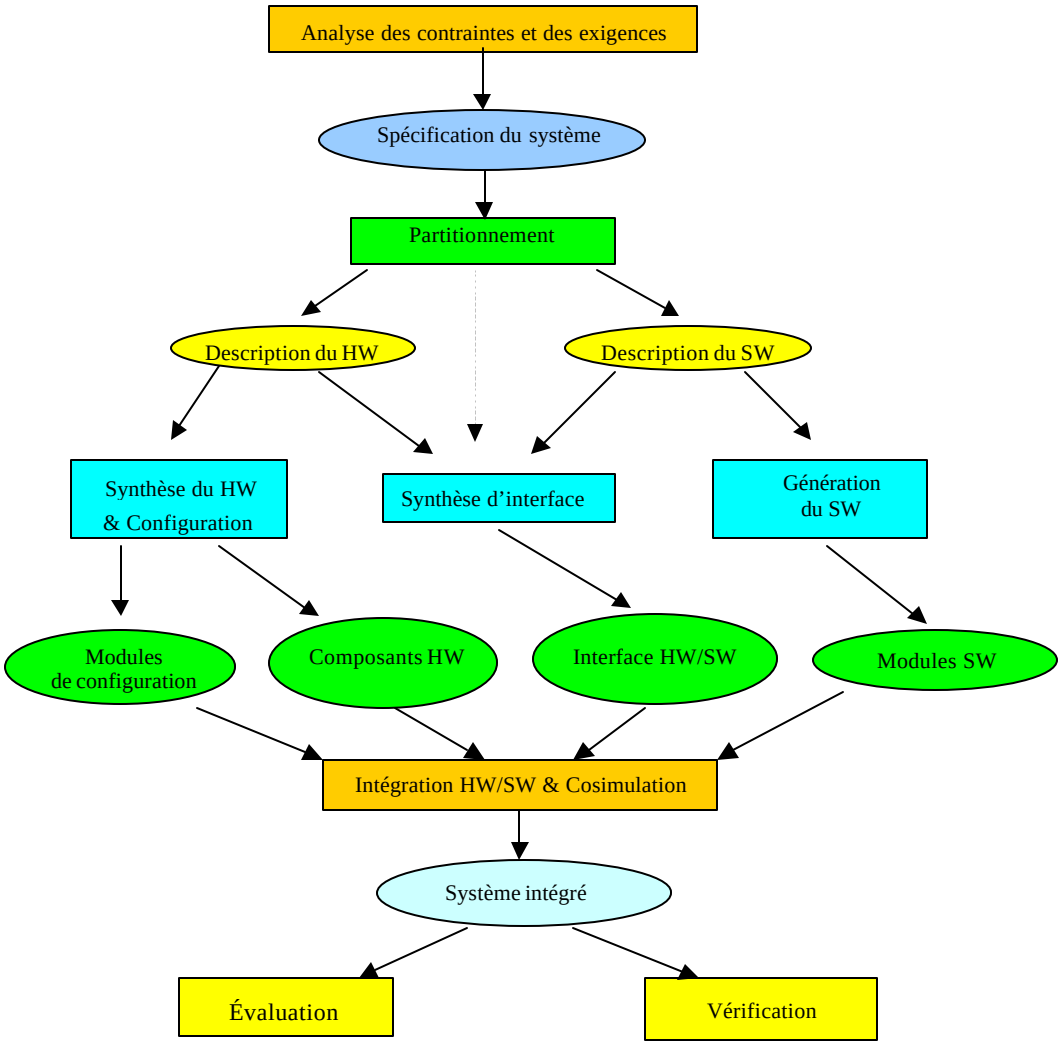


Figure 3.3 : Flux conventionnel du codesign

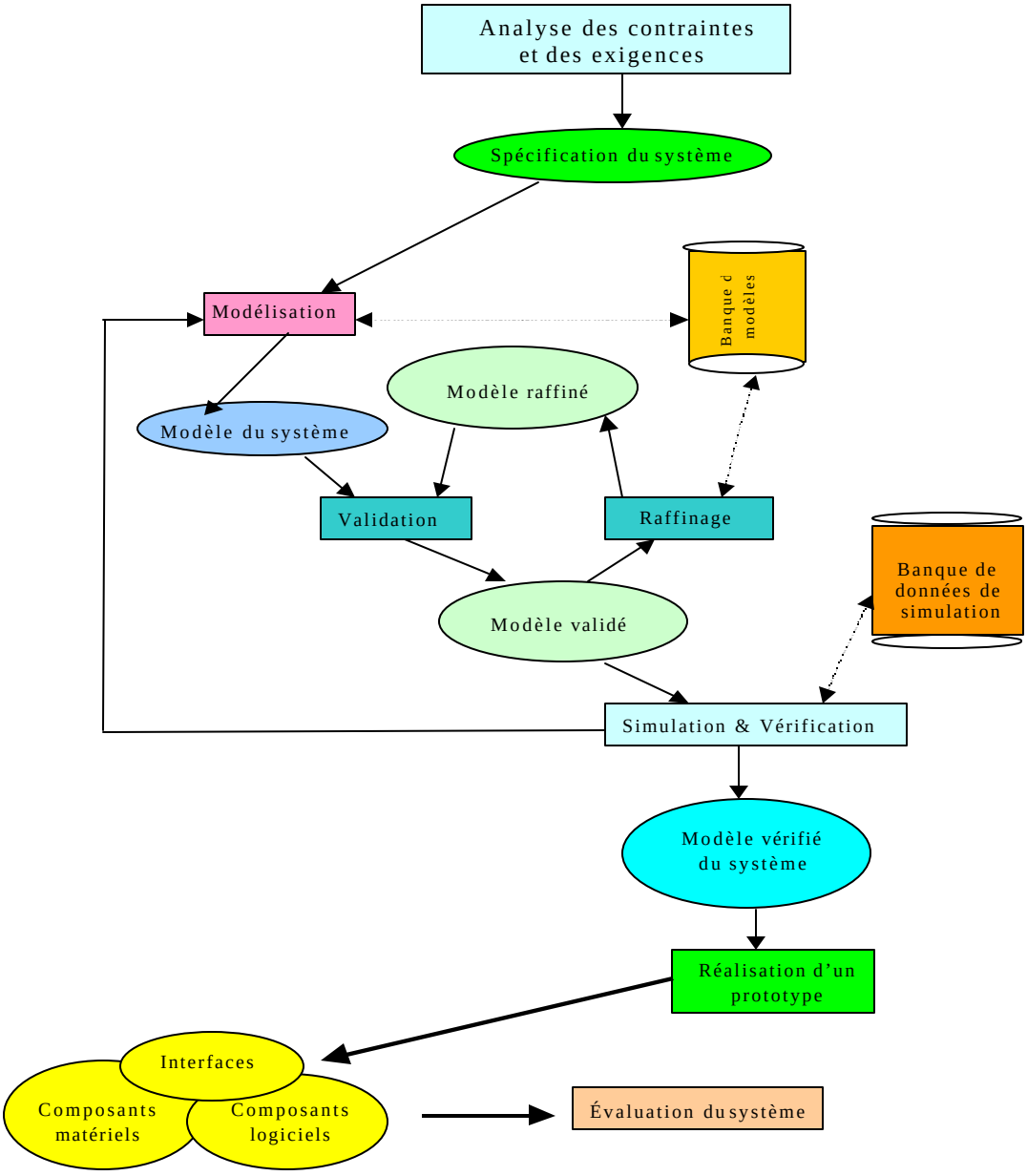


Figure 3.4 : Flux de codesign basé sur un modèle

3.4. Aperçu sur les techniques de covérification

Il est important, avant d'examiner ces techniques, d'évoquer brièvement les différents types de représentations mathématiques d'un système donné.

3.4.1. Techniques de représentation

Différents modèles pour la spécification et la vérification d'un système ont été élaborés afin de simplifier et de bien manipuler les tâches de conception et de validation. Parmi ces modèles, nous citons :

- les machines à états finis étendues (EFSMs) [77],
- les machines à états finis dédiées au codesign (CFSMs) [62],
- les systèmes à transitions étiquetées (LTSs) [68],
- les modèles à diagrammes d'états (State-chart models) [75],
- et les réseaux de Petri [66] :
    - Systèmes Place/Transition,
    - Systèmes colorés,
    - Systèmes Prédicat/Transition.

3.4.2. Mécanismes de la covérification

Au cours de cette section, nous allons répondre aux questions suivantes :

- Comment covérifier?
- Quelles sont les propriétés à vérifier?
- Quels sont les systèmes à covérifier?

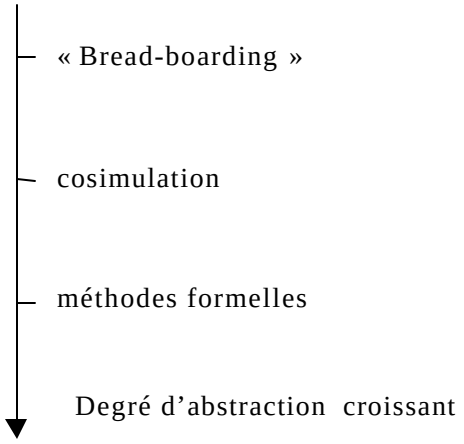


Figure 3.5 : Niveaux d'abstraction des trois méthodologies de covérification

En général, la covérification peut être effectuée par trois techniques principales : par cosimulation, par des méthodes formelles et par « bread-boarding ». Nous classons les trois techniques de covérification selon leurs degrés d'abstraction comme l'indique la figure 3.5.

A. Covérification par cosimulation

Cette technique (figure 3.6) simule les deux parties logicielle et matérielle tout en assurant la communication entre elles; cette mise en communication entre le logiciel et le matériel est appelée intégration virtuelle. Dans le cas d'un modèle unifié [82], uniquement un seul simulateur est utilisé pour faire la cosimulation; cependant, quand on a affaire à un modèle distribué [67, 73, 74, 83-85], deux ou plusieurs simulateurs sont requis dépendamment du nombre de langages qui sont utilisés pour décrire les parties du système intégré. Grâce aux outils de cosimulation, l'intégration entre le matériel et le logiciel commence assez tôt avant la phase du prototypage; les propriétés du système L/M sont vérifiées par des simulations séquentielle ou distribuée (chapitre 5). Quelques exemples de covérification par simulation sont présentés en littérature, tels que la covérification d'une boucle à verrouillage de phase à NORTEL [101] et l'estimation de performance de la covérification du design d'un système à 24 processeurs RISC à SIEMENS [103].

Le modèle élaboré est stimulé par une séquence d'entrées efficace; puis la séquence des sorties générées est comparée à la séquence des sorties prévues. Cette technique souffre essentiellement de la difficulté de trouver une séquence d'entrées plus appropriées pour mener ce test, c'est-à-dire celle qui incite le système à générer une séquence de sorties riche en information « covérificationnelle[5] ».

B. Covérification par «bread-boarding»

Cette technique (figure 3.7) consiste à mettre en circuit imprimé un prototype du système en question puis à lui appliquer des tests spécifiques de bon fonctionnement. Il est clair que cette méthode requiert toujours un prototype pour son accomplissement. Cependant, une telle exigence la rend impraticable pour les

[5] C'est à dire une information qui nous aide à prendre des décisions sur la conformité du système.

systèmes complexes dont les prototypes sont loin d'être facilement réalisables. Aussi, les pertes en termes de temps et de semi-conducteur, sont-elles énormes si la vérification s'avère négative.

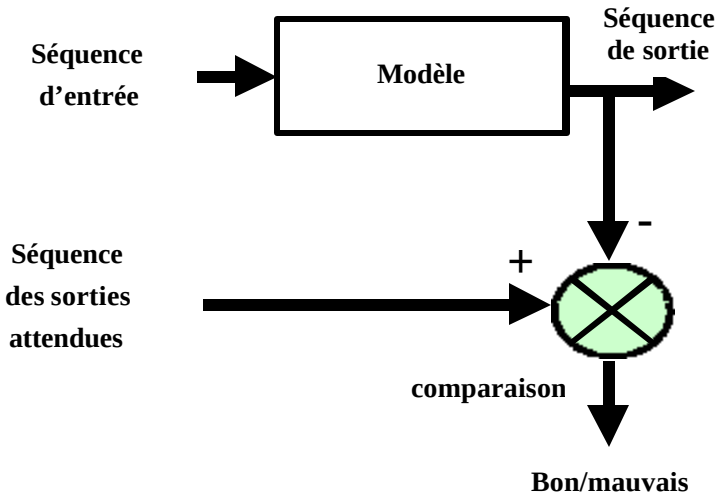


Figure 3.6 : Technique de covérification par cosimulation

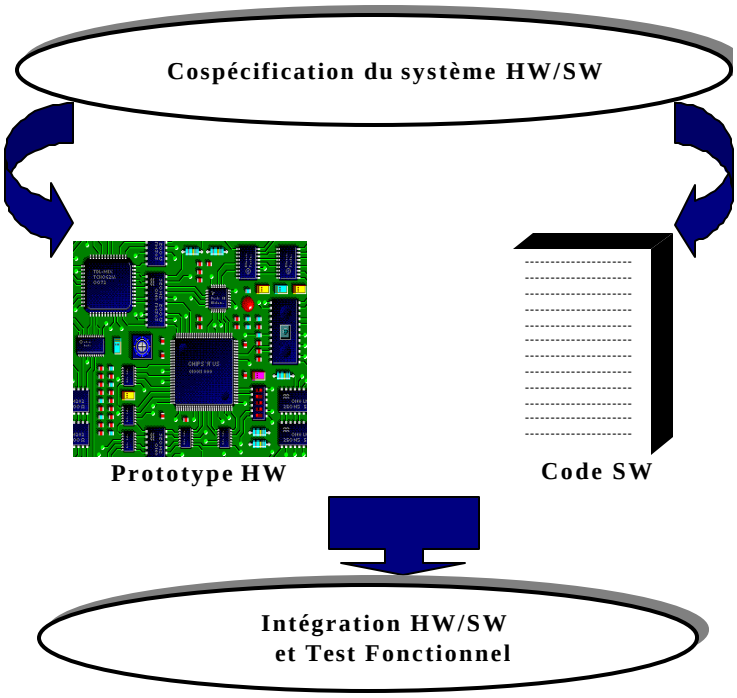


Figure 3.7 : Technique de covérification par « Bread-boarding »

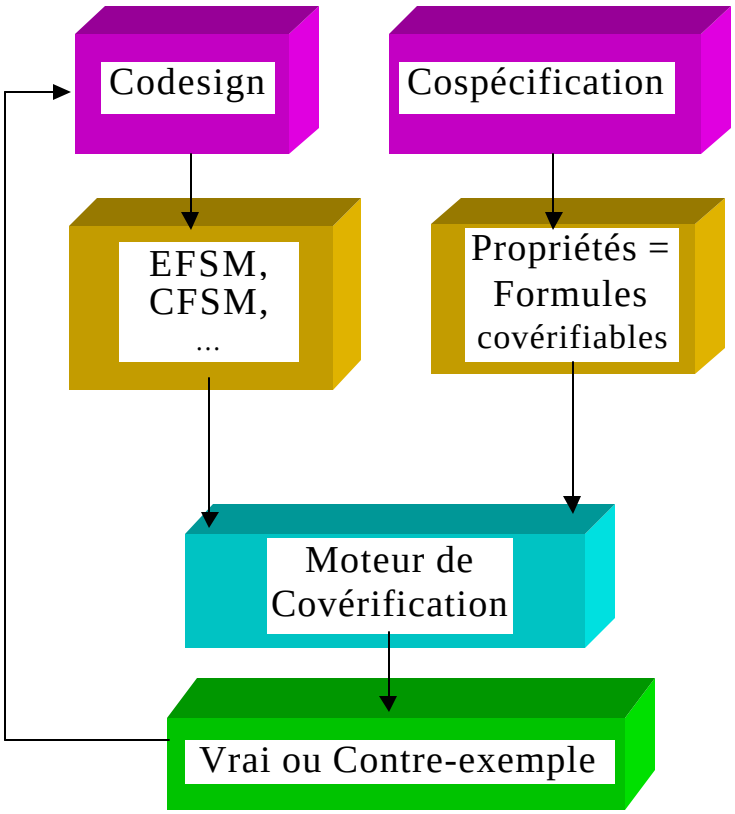


Figure 3.8 : Technique de covérification formelle

C. Covérification formelle

Avec celle-ci, nous n'avons ni à retrouver de performants stimuli ni à manipuler un «bread-boarding » mais seulement à utiliser des techniques mathématiques pour prouver la vérité de propriétés bien définies (figure 3.8). Il est nécessaire que ces dernières soient exprimées dans un langage compatible avec celui de la représentation du design. Cette catégorie de covérification attire de plus en plus l'attention des chercheurs sauf que la majorité des articles publiés jusqu'à présent traitent de la vérification soit du matériel, soit du logiciel et non pas la covérification proprement dite (le mariage L/M). L'inconvénient de cette technique consiste en son niveau d'abstraction plus élevé, en d'autres termes, on ne dispose plus de détails de réalisation du système. Parmi les méthodes de vérification les plus répandues, nous mentionnons le «model checking» et le «theorem proving». Certains travaux tentant

de vérifier les systèmes L/M à l'aide de ces dernières techniques font abstraction du caractère logiciel en le considérant juste comme une boîte noire au sein de la partie matérielle et vérifient l'ensemble comme étant un système matériel homogène [78, 100, 129].

3.4.3. Types d'erreurs

Comme un système intégré est composé de deux parties inter-communicantes, des modules logiciels et des composants matériels, intuitivement, toutes les fautes possibles peuvent être classifiées de la manière suivante :

- *erreurs M-M* : Ce sont des erreurs ayant lieu au sein de la partie matérielle.
- *erreurs L-L* : Ce sont des erreurs localisées au sein de la partie logicielle.
- *erreurs L-M* : Ce sont des erreurs relatives aux signaux de communication allant de la partie logicielle vers la partie matérielle.
- *erreurs M-L* : Ce sont des erreurs relatives aux signaux de communication allant de la partie matérielle vers la partie logicielle.

Pour plus d'abstraction, il est possible de percevoir les erreurs L-M et M-L sous un même type que nous identifions par : *erreurs de communication*.

3.4.4. Propriétés à vérifier

Les propriétés à vérifier (« correctness properties ») sont classifiées en deux catégories significatives : les propriétés de sûreté («safety properties») et les propriétés de vivacité (« liveness properties »). La première catégorie signifie que jamais une chose mauvaise n'aura lieu pendant le fonctionnement; quant à la seconde, elle signifie qu'éventuellement de bonnes choses auront lieu. Ces propriétés sont d'un haut niveau d'abstraction et nécessitent alors d'être détaillées en fonction de l'application en cours et de la description du système.

Les propriétés sont généralement décrites par des expressions logiques dans un langage approprié. Elles sont extraites du cahier de spécification du système en question. Si elles sont satisfaites par l'implémentation, alors la vérification relative à ces contraintes est décidée positive, sinon il faut revoir le design.

3.4.5. Applicabilité des méthodes de vérification à la covérification

La covérification est appliquée à des systèmes mixtes (composés chacun de deux parties matérielle et logicielle). Après notre analyse des différentes méthodes d'accessibilité pratiquées dans le domaine de la vérification, nous concluons que toutes ces méthodes font appel uniquement au graphe d'états du système et non pas au système lui-même (c'est-à-dire il y a une indépendance explicite du système). Autrement dit, ces méthodes ne perçoivent que le graphe d'états établi et font abstraction du système réel, elles ne sont pas sensibles à l'homogénéité ou à l'hétérogénéité du système original. Alors, lorsque la représentation d'un système intégré est achevée, donnant ainsi naissance à un graphe descriptif des entités : matérielles, logicielles et de leur intercommunication, les méthodes d'accessibilité lui sont applicables sans aucune restriction théorique. Cependant, à cause de leurs niveaux d'abstraction assez élevés, certains aspects de bas niveau sont ignorés dans les modèles de base; en conséquence, pour des vérifications pratiques, on fait souvent recours à des techniques moins abstraites telles que la simulation ou l'émulation.

3.5. Cosimulation : techniques et technologie

3.5.1. Définition

La cosimulation L/M, comme elle a été définie par William D. Bishop et al [81], est une technique de simulation d'un système en utilisant des entités logicielles et matérielles. Le problème de base en cosimulation réside dans la manière de réconcilier deux spécifications :

- pour exécuter le logiciel aussi rapidement que possible sur une machine hôte qui peut être même plus rapide que le CPU cible.
- pour conserver la synchronisation des simulations logicielle et matérielle de façon à ce qu'elles interagissent semblablement dans le système cible (réel).

3.5.2. Techniques de cosimulation

Gupta et al [82] ont proposé une technique basée sur un seul simulateur pour le matériel et le logiciel à la fois. Ce simulateur utilise une seule file d'événements et un modèle de processeur cible. Cette technique ne tient pas compte d'éventuelles différences entre les comportements matériel et logiciel. Une autre technique visant à améliorer les performances de cosimulation a fait l'objet des articles [67, 83, 84]. Cette technique lie faiblement un simulateur de matériel avec un processus de logiciel. La synchronisation est assurée par l'usage des standards de communication inter-processus offerts par le système d'exploitation de la machine hôte. Un des inconvénients de cette technique est que les horloges relatives aux parties logicielle et matérielle sont non synchronisées. Ceci requiert l'usage de protocoles correctifs. Ce problème est dû au fait que le matériel est plus rapide que le logiciel. Afin de surmonter l'inconvénient de celle-ci, une autre technique est proposée dans [85]. Elle consiste à mémoriser une trace de temps pour le logiciel et une autre pour le matériel et à utiliser des mécanismes divers pour les synchroniser périodiquement. Si le logiciel est *maître* alors il décide quand envoyer un message, mentionné par le cycle horloge courant du logiciel, au simulateur de matériel. Si le temps du matériel est déjà en avance, le simulateur peut remonter dans le temps, caractéristique que peu de simulateurs de matériel possèdent actuellement. Si le matériel est *maître* alors le simulateur de matériel appelle des procédures de communication qui, à leur tour, appellent le code logiciel de l'utilisateur.

3.5.3. Technologie de la cosimulation L/M

Dans cette section, nous présentons plusieurs possibilités offertes par la technologie actuelle pour modéliser et cosimuler un système L/M. Ces techniques font appel à des émulateurs, à des simulateurs ou à des modules réutilisables. Elles peuvent être basées sur des modèles virtuels (uniquement du code) ou sur des modèles réels (code et circuits). Rappelons que le modèle parfait d'un système donné est ce système lui-même, c'est pour cette raison que tous les progrès des développeurs visent la mise au point d'un modèle très proche du système original des points de vue de la structure et de la fonctionnalité.

- *Modèles fonctionnels pour l'exécution du code objet* [29]

Le programme de simulation est transformé en code objet et mémorisé dans un modèle HDL de la mémoire du processeur. Après, ce code objet est exécuté sur un modèle fonctionnel du microprocesseur. Les principaux inconvénients de cette approche sont la lenteur de la simulation et la complexité du modèle du microprocesseur.

- *Modèles matériels pour l'exécution du code objet* [29]

Pour réduire le temps d'exécution de la méthode présentée ci-dessus, cette approche propose de remplacer les modèles du microprocesseur et de sa mémoire par des puces réelles qui communiquent avec le simulateur. Les inconvénients de cette méthode sont les suivants :

- Le coût d'établissement de la communication entre le simulateur et les puces n'est pas compensé par une amélioration de la simulation.
- Le simulateur n'arrive pas à gérer des signaux échangés avec les puces au rythme dicté par ces derniers, ce qui cause des problèmes de congestion de signaux (« bottleneck »).

- *Utilisation des boîtes noires pour le matériel dans des programmes du logiciel* [29]

Le programme peut dépendre de certains nouveaux composants matériels dont le code (modèle) est non disponible. Alors, on associe à ces derniers des boîtes noires, appelées «stubs», obtenues par émulation. L'inconvénient de cette technique est que souvent les «stubs» ne reproduisent pas correctement les fonctions matérielles requises.

- *Prototypage du matériel* [29, 72]

Cette technique tend à décaler les phases de développement du logiciel et du matériel de sorte que le prototype matériel soit disponible avant le code du logiciel et alors quand ce dernier est réalisé, on l'exécute directement sur le matériel (prototypes FPGA). L'avantage de celle-ci est qu'elle donne plus de garantie sur la

fonctionnalité du logiciel sur le matériel. Par ailleurs, son inconvénient apparaît dans la lenteur du cycle de développement à cause de la séquentialité des conceptions du matériel et du logiciel, et en le coût d'un recyclage éventuel.

- *VSP* [29, 72, 86]

Cette technique a été créée par EDA (« Eagle Design Automation »). Elle met en ensemble les deux codes logiciel et matériel pour effectuer une simulation complète du système. Tous les modèles HDL ordinaires sont utilisables sauf que pour le composant *processeur*, on associe un modèle appelé : VSP («Virtual Software Processor »). VSP reproduit la fonctionnalité requise du processeur et offre des services de communication et de synchronisation avec le code logiciel. On distingue trois types de modèles VSP :

- *VSP/Link* : ce type possède la performance la plus haute; il accepte en entrée un code HLL (tel que C) et l'exécute conjointement sur le matériel (virtuel). Il utilise un modèle fonctionnel de bus (BFM : « Bus Functional Model ») pour décrire le bus du processeur mis en œuvre. Avec cette technique, on peut simuler toutes les transactions échangées entre le processeur et ses périphériques via le bus.
- *VSP/Sim* : celui-ci utilise un simulateur de jeu d'instructions (ISS) et le lie à un simulateur de matériel via une interface d'Eaglei (environnement de covérification). À l'aide de cette technique, on peut exécuter un ensemble d'instructions du processeur étudié.
- *VSP/Tap* : il connecte un « In-Circuit Emulator » (ICE) à un simulateur de matériel via la technologie de communication d'Eaglei ou d'autres. Le modèle du processeur peut être construit à l'aide des FPGAs.

Remarque

Une nouvelle technique connue sous le nom de « CoreTap » vient d'être élaboré par AMC (« Applied Microsystems Corporation »). Elle consiste à utiliser un processeur réel à la place du modèle VSP.

Une comparaison entre ces modèles est illustrée par le tableau suivant (tableau 3.1) [29] :

Modèle	VSP	Détails internes	Code supporté	Vitesse
VSP/Link	L	Néant	C, C++	Très rapide
VSP/Sim	L	Plusieurs	C, C++, Ass.	Rapide
VSP/Tap	M	Tous	C, C++, Ass.	Rapide

Tableau 3.1 : Les différents types de VSP

3.6. Quelques environnements de codesign et de covérification

Nous discutons brièvement ci-dessous de quelques environnements de codesign et de covérification. Il faut noter que leurs versions étudiées ici sont sorties au cours des années 90, il se peut donc que nos remarques formulées ci-dessous aient été déjà intégrées dans les nouvelles versions (ou futures).

3.6.1. POLIS

POLIS [87] est un environnement de codesign pour les systèmes intégrés interactifs. Aussi, permet-il la spécification, la cosimulation, la synthèse et l'analyse de performances des systèmes L/M. POLIS est axé sur une seule représentation, celle des machines à états finis dédiées au codesign (CFSM). Un système modélisé en POLIS est équivalent à un réseau de CFSMs dont chacune décrit un de ses composants. Les concepteurs, dans l'environnement POLIS, écrivent leurs spécifications en un langage de haut niveau (tels que Esterel, Verilog, VHDL, etc.). Le code HLL obtenu est traduit en CFSMs. Pour faire de la vérification, POLIS possède un traducteur de CFSMs en FSMs auxquelles sont applicables les outils (non intégrés à POLIS) de la vérification ordinaire. POLIS nécessite d'être doté d'un outil de vérification en vue d'effectuer entièrement et rapidement toutes les tâches désirées.

3.6.2. Ptolemy

Ptolemy [62] est un environnement de codesign permettant la cosimulation et la co-synthèse des systèmes intégrés. Il traite le système à concevoir comme étant une collection hiérarchique d'objets décrits à différents niveaux d'abstraction et utilise des modèles sémantiques pour communiquer avec chacun d'eux (objets) :

- *Domaine* : Chaque niveau d'abstraction avec son propre modèle est appelé domaine (exemple : flux de données, logique, etc.). Chaque domaine inclut un ordonnanceur d'exécution des « stars ».
- *Stars* : Ce sont des objets atomiques représentant les primitives du domaine (exemple : opérateurs de flux de données, portes logiques, etc.).
- *Galaxies* : Ce sont des collections de stars et de sous-galaxies.

Remarque

Les CFSMs peuvent être implémentées en Ptolemy par l'emploi de la notion du domaine à événements discrets (DED).

Comme en POLIS, il n'y a pas d'outils de vérification intégrés au sein de l'environnement Ptolemy. Il serait bien bénéfique dans le futur de doter Ptolemy d'un outil puissant de covérification.

3.6.3. CVE-Seamless

Seamless [88] est un environnement de covérification, récemment élaboré pour répondre aux besoins industriels et aux exigences sévères de performance. Il offre des services précieux dont :

- la réduction du temps de développement des systèmes intégrés.
- la réduction du nombre d'itérations d'un prototype matériel.
- le support des codes HLL et assembleur.
- L'accès à plus de 30 processeurs.
- le support de QuickHDL™, de QuicksimII, de Verilog-XL, des simulateurs logiques Leapfrog et de l'émulateur SimExpress.

Seamless crée un prototype virtuel d'un système intégré, ce qui permet aux équipes de conception de vérifier l'interaction L-M assez tôt dans le processus de conception, c'est-à-dire lorsque les erreurs sont encore moins coûteuses à corriger. Une telle covérification précoce fournit beaucoup plus de flexibilité aux concepteurs afin de raffiner et d'optimiser leur design et d'y apporter des modifications éventuellement impossibles plus tard.

Seamless élimine le temps de retard qui aurait lieu lorsque le test du logiciel doit attendre qu'un prototype du matériel soit réalisé. Il permet une description du matériel depuis le niveau comportemental et jusqu'au niveau portes logiques, et une description du logiciel en C++ et en assembleur.

3.6.4. Eaglei

Nous avons déjà eu l'occasion lors d'un stage [6] à NORTEL (Corkstown-Ottawa) de travailler sur un projet de covérification d'un système de télécommunication (appelé LUNAR) à l'aide de l'environnement Eaglei [89], un produit de l'ex-compagnie iViewLogic (actuellement Synopsys). Le système étudié reçoit continuellement des trames sérielles, les traite puis les génère sous forme de nouvelles trames en série ou de paquets d'octets (trames parallèles). La figure 3.9 illustre les principales composantes de l'environnement de cosimulation Eaglei. Ce dernier utilise la technologie VSP (section 3.5.3).

Après la mise en marche de cette cosimulation, nous avons noté les observations suivantes :

- le temps de cosimulation est assez long (quelques dizaines de minutes par vecteur test),
- le nombre de stimuli possibles est très élevé (~ 2^{160} entrées possibles) et une sélection bien ciblée de quelques vecteurs de test est indispensable pour accélérer la cosimulation et estimer sa crédibilité.

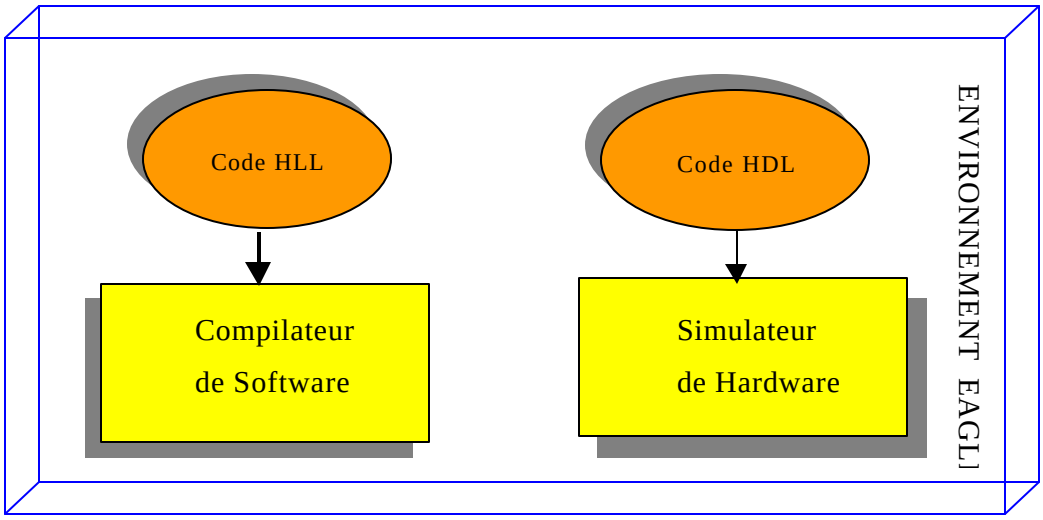


Figure 3.9 : Vue simplifiée d'Eaglei

3.7. Covérification ordonnancée

Le processus de notre méthodologie de covérification L/M fait appel à des objets concurrents : les threads de Java. Ces derniers sont ordonnancés à l'aide de la combinaison de deux techniques : celle du tourniquet et celle des priorités [92]. La première concerne les threads à priorités identiques; elle les fait passer en morceaux à l'unité d'exécution (CPU) en leur allouant le même temps d'exécution. Quant à la seconde, elle classe les threads par ordre de priorités. Le thread de priorité maximale est invité en premier à l'unité d'exécution, et ainsi de suite.

Le processus de covérification est qualifié d'ordonnancé si l'exécution de ses événements obéit à une règle d'ordre. Cette dernière est implémentée sous le nom d'*ordonnanceur*. Elle pourrait faire appel à une référence de temps (cas avec temporisation) ou non (cas sans temporisation). En respectant cette règle, la succession des événements aurait au moins du sens, du point de vue fonctionnement. L'ordonnanceur supervise l'ensemble des threads en les passant à l'unité d'exécution selon un critère de classement donné. La communication entre les threads, dépendamment de la spécification du système, pourrait être synchrone ou asynchrone [92].

3.7.1. Cas sans temporisation

Nous entendons par covérification sans temporisation, la covérification dont l'ordonnancement n'est pas basé sur la cadence d'une ou de plusieurs horloges. Seules la réception et l'émission d'un événement sont éventuellement mises en jeu. La consommation d'un résultat par un bloc du programme ne peut être effectuée qu'après la réception de celui-ci. La synchronisation est assurée par des rendez-vous entre les processus.

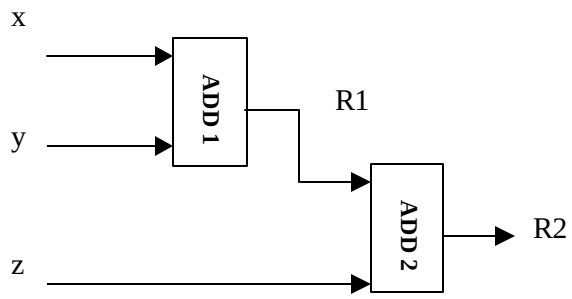


Figure 3.10 : Circuit combinatoire

```
Public main ()
{
...
Scheduler ordonnanceur=new Scheduler();
ADD ADD1=new ADD (x, y, R1);
ADD ADD2=new ADD (z, R1, R2);
...
ordonnanceur.addfirst(ADD1);
ordonnaceur.addnext(ADD2);
...
ordonnanceur.start();
}
```

Figure 3.11 : Ordonnancement sans horloge

Selon un ordre d'opérations dicté par l'utilisateur, l'ordonnanceur tâche de respecter cette directive en contrôlant le flux de données. Par exemple, dans le circuit combinatoire de la figure 3.10, nous (ou l'ordonnanceur) commençons toujours par effectuer la première addition, puis en suit la seconde (figure 3.11). Il importe de noter que les additionneurs *ADD1* et *ADD2* sont deux instances de l'objet *ADD*; ce

dernier a été engendré en héritant de l'objet *Thread*. De même, « *ordonnanceur* » est une instance de « *Scheduler* », objet dérivant de la classe mère *Thread*.

3.7.2. Cas avec temporisation

L'ordonnancement, dans ce cas, exploite les variables du temps. Chaque thread possède sa propre horloge locale. Cette dernière cadence les opérations intra threads. Une horloge globale temporise les opérations inter threads. Les horloges locales sont indépendantes à moins que des synchronisations émanant du fonctionnement s'imposent. Alors, dans cette situation, des attentes des deux partenaires inter communicants, l'envoyeur et le receveur, pourront être indispensables.

En résumé, la covérification temporisée est effectuée cycle par cycle d'horloge. Toutes les activités ayant lieu au sein de chaque cycle sont mises en évidence grâce à un affichage direct, un enregistrement dans un fichier ou à une consommation révélatrice de certains résultats (cas des propriétés de contrôle par exemple).

3.8. Conclusion

Nous avons tout le long de ce chapitre tenté d'introduire le lecteur aux concepts de la covérification des systèmes intégrés. En effet, après avoir expliqué le pourquoi et les objectifs de ce nouvel horizon de recherche, nous avons énuméré les différentes représentations mathématiques susceptibles de décrire un système intégré, puis nous avons exposé les grandes catégories des techniques de la covérification tout en discutant les avantages et les inconvénients de leurs applications. Aussi, du fait que tout système (en particulier un système intégré) est modélisable par un graphe d'états, nous avons mené une discussion sur les techniques d'accélération de son accessibilité. Ensuite, nous avons présenté différentes techniques de cosimulation des systèmes intégrés. Enfin, nous avons exposé succinctement quelques environnements de codesign et de coverification tels que POLIS, Ptolemy, Seamless et Eaglei.

En conclusion, nous retenons que le domaine de la covérification est encore à ses premières phases, aussi bien en terme de publications qu'en terme d'outils dédiés; il nécessite d'être perfectionné davantage. Dans le chapitre suivant, nous discutons comment les threads pourront mieux servir un processus de covérification.

Chapitre 4

Méthodologie de covérification basée sur le « mutlithreading »

4.1. Introduction

Dans ce chapitre, nous faisons le point sur une importante catégorie des méthodologies de conception : il s'agit de la conception des systèmes mixtes L/M et leur covérification L/M. Antérieurement (selon la méthodologie classique de conception), les sous-systèmes logiciel et matériel étaient conçus séparément, et des problèmes se manifestèrent éventuellement au moment de l'intégration. Cela conduisait à un processus itératif non acceptable en termes de coût et de temps de réalisation. Pour surpasser ces difficultés, l'intégration L/M est requise assez tôt dans le processus de conception et la covérification doit être effectuée en concurrence avec le processus de codesign. On peut décrire les deux parties du système intégré, logicielle et matérielle, en utilisant seulement un langage de description de matériel HDL, et après on le soumet à un outil de vérification formelle pour s'assurer de sa conformité. Cependant, la description du logiciel dans ce cas est très limitée et

plusieurs aspects de traitement sont ignorés. Par exemple, l'outil VIS[6] [98] accepte juste un sous-ensemble de Verilog; malgré son extension par la fonction de non-déterminisme et les variables symboliques, les différents aspects du logiciel ne peuvent être complètement représentés à l'aide de VIS-Verilog. En conséquence, la vérification des systèmes L/M va être incomplète. D'où des techniques efficaces de description et de vérification sont sérieusement indispensables.

Nous présentons dans ce chapitre une méthodologie de covérification basée sur le concept du « multithreading » [99]. Dans cette approche, la partie logicielle est décrite par un ensemble de threads communicants, l'interaction entre les parties logicielle et matérielle est clairement définie et des threads additionnels sont utilisés pour exprimer des assertions de vérification. Finalement, la cosimulation du code global est exécutée comme un modèle unifié[7] (homogène) à l'aide du compilateur Java ou comme un modèle distribué (hétérogène) à l'aide d'un environnement de covérification tel que CVE-Seamless [88] ou Eaglei [89].

4.2. Technique de covérification basée sur le «multithreading»

Le codesign d'un système intégré commence par le partitionnement de celui-ci en des sous-systèmes logiciels et matériels. Ensuite, les équipes de conception de logiciel et de matériel effectuent respectivement la conception de la partie logicielle et de la partie matérielle. Simultanément, une interface assurant la communication entre le logiciel et le matériel est synthétisée. Lorsque la conception de toutes ces parties est presque finie, les concepteurs assemblent ces dernières pour obtenir l'implémentation recherchée du système global. La question qui se pose à ce niveau est « est-ce que cette implémentation remplit bien les consignes de la spécification?». Pour répondre à cette question, au moins partiellement, nous effectuons une cosimulation d'une telle implémentation avec un nombre de vecteurs de test.

[6] VIS : Version 1.3
[7] Les deux parties sont décrites par un code HLL.

La covérification d'un système L/M consiste à vérifier si l'implémentation de la partie logicielle répond bien à sa spécification, si de même la description de la partie matérielle satisfait sa spécification et si leur intégration respecte aussi les exigences de la spécification globale. Les figures 4.1 et 4.2 illustrent bien les architectures typiques, comportementale et structurelle, des systèmes intégrés auxquels nous avons affaire. La partie logicielle contient le code du système plus des données. Le code renferme des fonctions d'adressage de la mémoire, d'interfaçage avec la partie matérielle et d'autres sous-programmes implémentant des fonctionnalités propres au système qu'on a décidées de réaliser « logiciellement » lors du partitionnement L/M. La partie matérielle est un ensemble de modules de circuits, de registres, de blocs à propriété intellectuelle (IP), etc. L'interface entre les deux parties est montée autour d'un processeur qui exécute le code logiciel et gère les signaux du matériel. Ce type de systèmes, à cause de leur complexité et de leur hétérogénéité, défie les méthodes contemporaines de vérification et de simulation.

Notre méthodologie de covérification est basée sur la technique de cosimulation et sur certains aspects de la vérification formelle. Le mélange des philosophies de simulation et de vérification formelle pourrait s'apparenter à la dénomination de «vérification semi-formelle» [109].

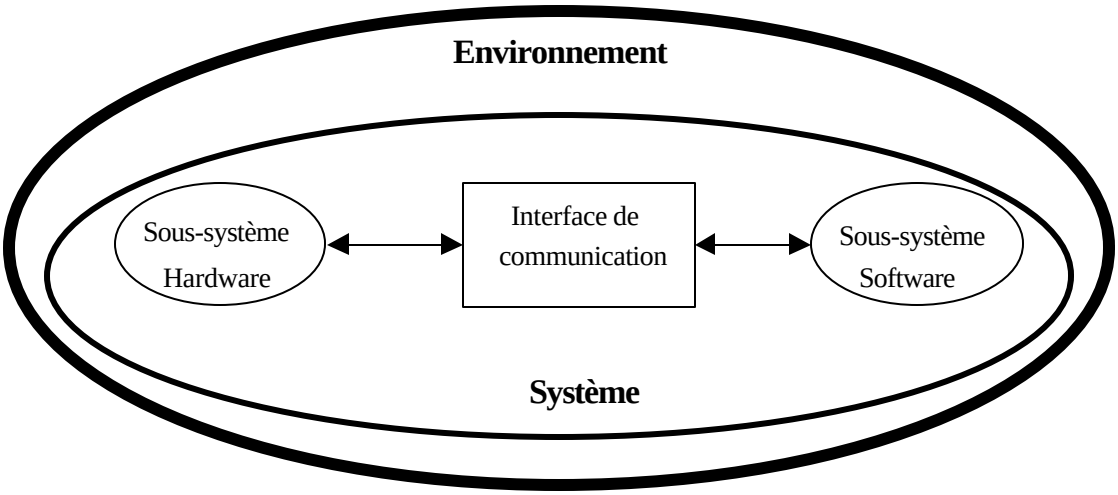


Figure 4.1 : Architecture comportementale d'un système L/M

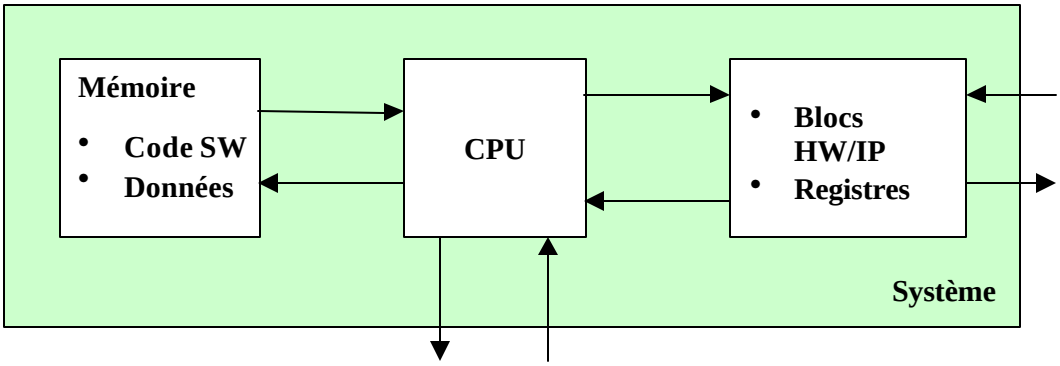


Figure 4.2 : Architecture structurelle d'un système L/M

La technique de covérification basée sur le «multithreading» que nous proposons est une approche qui tente d'améliorer et de mieux organiser le processus de covérification en utilisant le concept du «multithreading» et en orientant la cosimulation par des propriétés du système intégré étudié. Cette technique est composée de quatre étapes ordonnées :

Étape 1 : Organiser la partie logicielle en threads.
Étape 2 : Mettre les signaux révélateurs du matériel dans des registres pour fin d'observation.
Étape 3 : Décrire les propriétés en threads.
Étape 4 : Cosimuler le système avec ses propriétés en tenant compte des étapes précédentes.

4.2.1. Première étape : Mise en threads de la partie logicielle

À l'instar de la partie matérielle qui est décrite par des « *modules* » en Verilog ou par des «*entities* » en VHDL et dont l'exécution est parallèle et guidée par des événements, nous parallélisons la partie logicielle à l'aide des threads (figure 4.3). Alors, nous organisons minutieusement la partie logicielle en un nombre convenable de threads. Au pire des cas, un seul thread englobera tout le code du

logiciel. L'emploi des threads augmente le degré de visibilité au sein de ce code et facilite sa manipulation pour ajouter des blocs de code, localiser et corriger d'éventuelles erreurs (par un utilisateur).

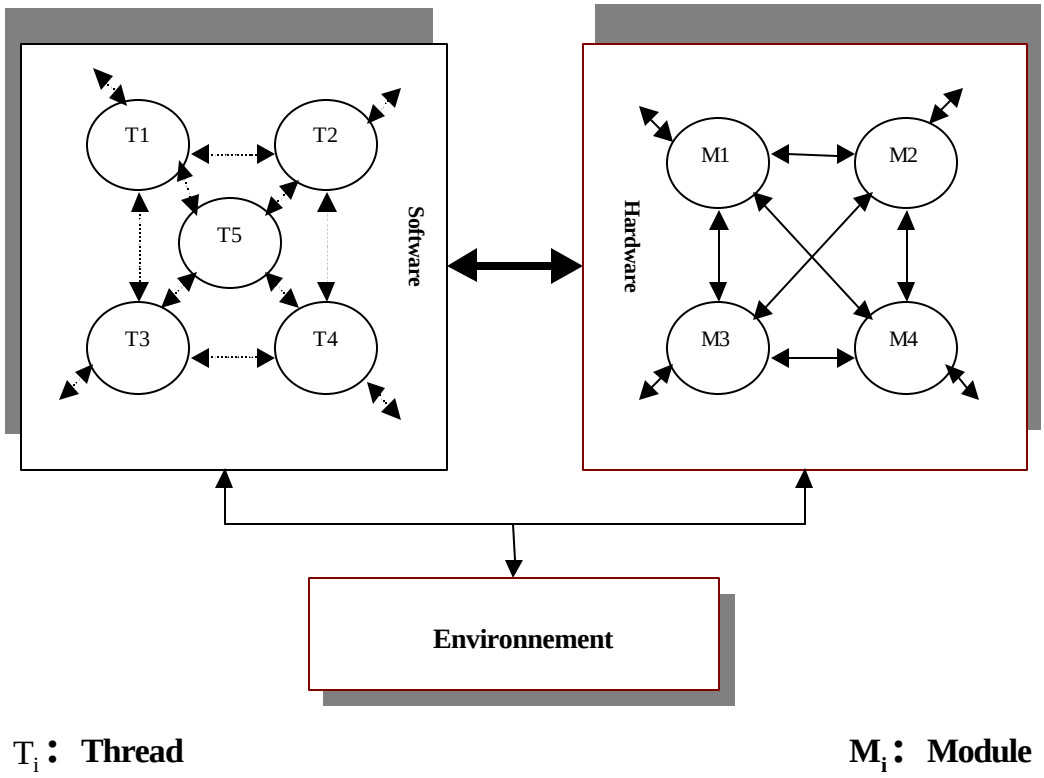


Figure 4.3 : Organisation des codes HLL et HDL

Supposons que *C_SW* est le code de la partie logicielle, l'opérateur chargé de la covérification effectuera lui-même (au moins pour l'instant) l'opération de la mise en threads de *C_SW*. Par exemple, il le répartit en deux threads, $C\_SW = \{T_1, T_2\}$. T_1 et T_2 sont donc des threads de la partie logicielle auxquels seront ajoutés d'autres threads porteurs de propriétés (troisième étape). Il faut noter que pour l'instant la description du logiciel en threads est faite de manière non automatique. Le problème de la mise en threads du logiciel (mise en concurrence ou « threadabilité ») est similaire à celui de la parallélisation d'un programme. Nous n'avons pas étudié ce problème dans cette thèse. C'est l'opérateur, grâce à son expérience, qui va prendre

la décision de décrire son logiciel en combien de threads. Ceci dépendra principalement de la nature du code du logiciel (dépendances de données et de contrôle) et de l'architecture de la machine hôte (nombre d'unités de traitement). La partie matérielle, elle aussi, est une collection d'entités concurrentes dont le nombre est tributaire de la stratégie de conception mise en œuvre et de la fonctionnalité du matériel. Soit, par exemple, *C_HW* le code de la partie matérielle réparti en trois modules, $C\_HW = \{M_1, M_2, M_3\}$. Cette partie, selon les objectifs de la covérification, sera enrichie de quelques registres dédiés à cette fin (voir la deuxième étape).

Le système intégré, après cette réorganisation, est perçu comme étant un ensemble d'objets concurrents de différents types. Ces objets se communiquent entre eux soit directement (même type) soit indirectement (types différents) via un moyen de communication offert par l'environnement de travail (Sockets ou RPC). Cette répartition du code procure une facilité de « debugging », d'ajout et de réutilisation.

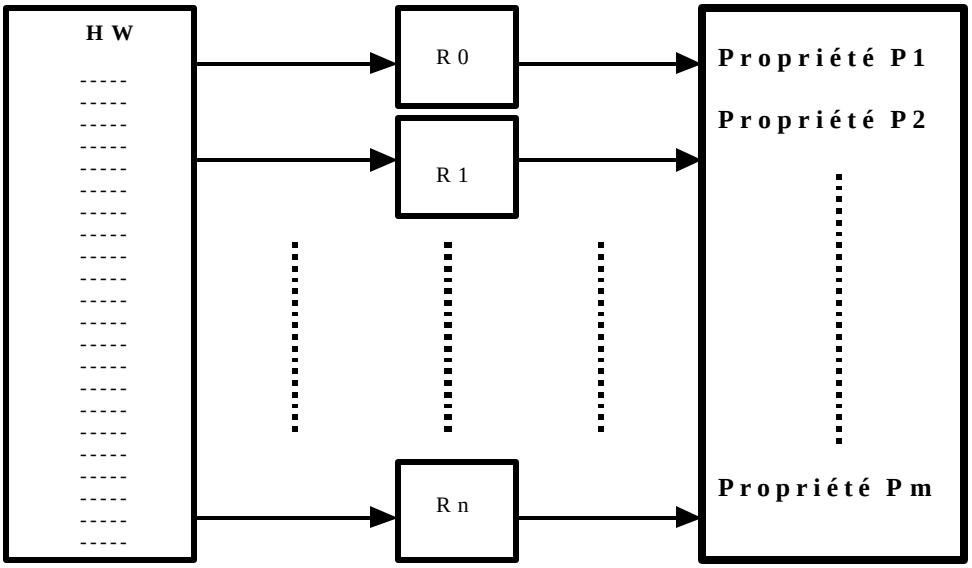


Figure 4.4 : Registres auxiliaires pour des fins de covérification

4.2.2. Deuxième étape : Registres de surveillance

Cette étape est considérée dans le but de rendre accessibles au logiciel les paramètres du matériel dont dépendent certaines propriétés (figure 4.4). En effet, les propriétés à vérifier sont définies du côté du logiciel et certaines d'entre elles supervisent en tout ou en partie le comportement du matériel. Alors, des paramètres du côté matériel, tels que des variables d'états par exemple, sont sauvegardés dans des registres réservés à la vérification (RFV). Dans certains cas, ces registres font partie intégrante du design original; cependant, quand il y a un manque de registres libres, nous considérons des registres additionnels appartenant à une mémoire auxiliaire (ne faisant pas partie du design). Ces registres sont adressables à partir de la partie logicielle et ainsi les signaux du matériel mis en registres sont observables. C'est un transfert d'information du matériel au logiciel.

Supposons que nous voulons suivre la trace d'un signal intermédiaire *SI* au sein d'un module matériel. Produit par un sous-bloc matériel, *SI* est capté au bout d'une sortie intermédiaire sans registre associé. Pour pouvoir lire sa valeur à partir de la partie logicielle, nous sommes mené à créer dans le code de la partie matérielle un registre *R_SI* image de *SI* qui reflète son évolution. *R_SI* est accessible en écriture (au moins) du côté matériel et en lecture (au moins) du côté logiciel. Toutes les propriétés décrites dans la partie logicielle dépendant directement de SI seront explicitement exprimées en fonction de son registre image *R_SI*. Cependant, dans le cas où un signal posséderait son propre registre image accessible en lecture du côté logiciel, nous n'avons pas besoin d'en créer un, nous utilisons ce même registre quand c'est nécessaire. Tel est le cas par exemple des registres de configuration qui sont accessibles en écriture/lecture depuis la partie logicielle. Ces derniers sont utilisés par exemple comme indicateurs ou identificateurs d'un mode de fonctionnement de la puce mère. On peut les réécrire à tout moment pour basculer d'un mode de fonctionnement à un autre (cas de LUNAR [6] par exemple).

Les registres éventuellement ajoutés pour des fins de covérification seront éliminés de la partie matérielle à la fin du processus de covérification à moins qu'il s'agisse d'une conception pour covérifiabilité (DFC : «Design For Coverifiability»)[8]. Dans ce dernier cas, tous les registres sont conservés et la stratégie de covérification développée lors du codesign L/M demeure valable même sur le système réel (après la fabrication du prototype).

4.2.3. Troisième étape : Spécification des propriétés

Extraites de la spécification du système, les propriétés sont de nature soit séquentielle soit concurrente. Aussi, peuvent-elles concerner la partie logicielle uniquement, la partie matérielle uniquement ou les deux ensemble.

Deux propriétés dont l'exécution doit se faire en concurrence (simultanément), sont dites concurrentes, autrement elles sont dites séquentielles. Deux propriétés concurrentes sont contenues dans deux threads différents, tandis que deux ou plusieurs propriétés séquentielles peuvent cohabiter ensemble dans un même thread et même en présence d'une propriété concurrente.

Sans faire aucun changement du code logiciel original (c'est-à-dire le logiciel du système intégré comme il a été initialement spécifié), nous l'augmentons par des threads additionnels pour les fins suivantes : gérer les accès *lecture/écriture* des registres et valider les propriétés du système (figure 4.5). Les propriétés concurrentes sont décrites chacune par un thread de même priorité que les autres. Les propriétés séquentielles sont rassemblées toutes ou juste une partie dans un thread. L'ensemble de ces propriétés est exécuté en pleine concurrence.

Supposons que nous avons des propriétés qui sont exprimées en fonction des registres images de certains signaux de la partie matérielle. Alors, pour évaluer ces propriétés, il faut lire les registres en question. Un thread peut être chargé de lire tous

[8] DFC est un champ brut qui pourrait être développé à la manière de DFT («Design For Testability»).

les registres images à chaque cycle de simulation ou bien pour chaque propriété, son thread porteur lit ses registres paramètres juste avant son évaluation.

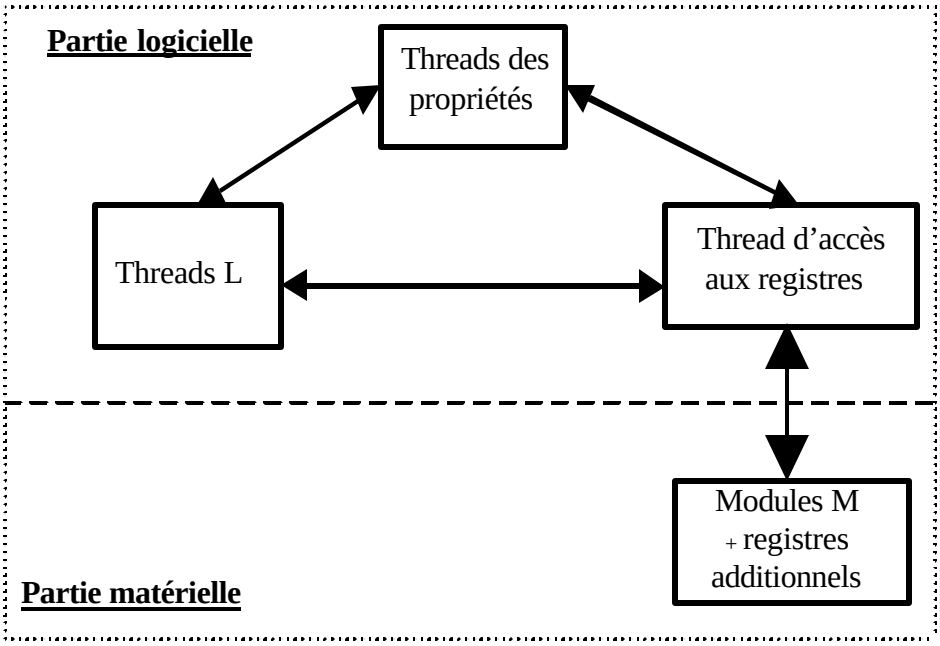


Figure 4.5 : Logiciel original augmenté par d'autres threads

4.2.4. Quatrième étape : Cosimulation du système

Nous regroupons le logiciel original et les threads résultant de la troisième étape. De même, le matériel original est éventuellement augmenté par des registres comme nous l'avons expliqué à la deuxième étape. Le système global obtenu est donc composé du logiciel original mis en threads, des threads porteurs des propriétés, du matériel original décrit en modules et d'éventuels registres images. Ensuite, nous cosimulons l'ensemble à l'aide d'un environnement de covérification (figure 4.6).

Avec un tel environnement, le simulateur de matériel va nous réfléchir la fonctionnalité de la partie matérielle, et simultanément l'exécution de la partie

logicielle va nous faire parvenir des rapports sur l'évaluation des propriétés. La communication entre les deux parties se fait via un modèle de microprocesseur qui assimile le processeur de la machine hôte pour satisfaire les spécifications requises du modèle. Par exemple, lors de la covérification du système LUNAR à NORTEL, nous avons travaillé avec un modèle VSP_mc68360 (Motorola) sur une station Sun dotée d'un processeur ULTRASPARC. La technique de cosimulation VSP/Link offerte par l'environnement de covérification Eaglei, nous permettait à l'aide d'un modèle fonctionnel de bus (BFM) décrivant les transactions sur les entrées/sorties du microprocesseur mc86360, de reproduire la fonctionnalité de ce dernier en faisant appel aux services du processeur de la machine hôte.

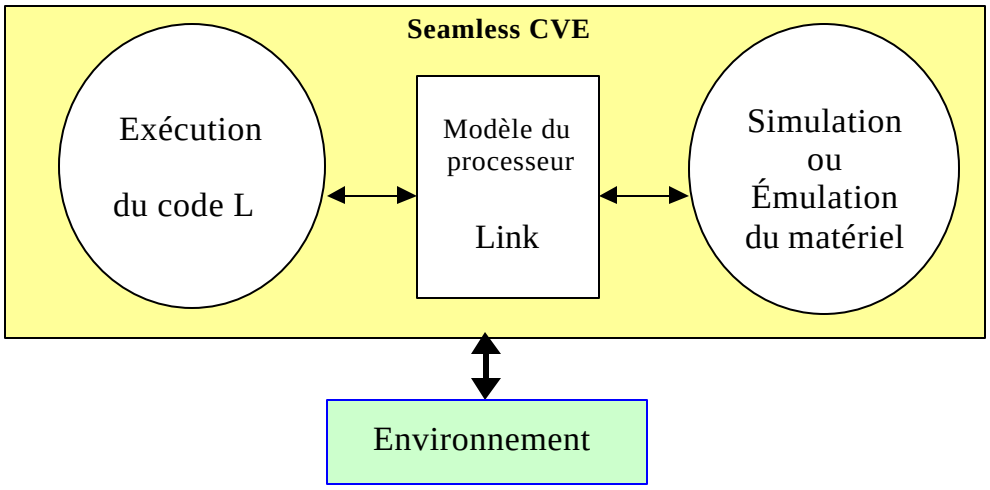


Figure 4.6 : Vue comportementale de cosimulation

Un modèle BFM d'un microprocesseur englobe la fonctionnalité de son bus. Il peut seulement exécuter les transactions du bus (communication via bus) sur le bus du microprocesseur mais il ne peut exécuter des instructions (appartenant au code du microprocesseur modélisé). Un BFM est donc un modèle de microprocesseur abstrait qu'on peut utiliser pour vérifier comment un microprocesseur et ses périphériques interagissent. Le BFM est une composante clé dans toute solution de covérification du fait qu'il représente l'interface entre les parties logicielle et matérielle. Pour exécuter des instructions du code assembleur d'un microprocesseur donné, on utilise

un simulateur de jeu d'instructions (ISS : «Instruction Set Simulation») qui traduit de telles instructions en instructions admises par la machine hôte (section 3.5).

4.3. Exemple d'illustration : un contrôleur à logique floue

Un contrôleur flou est un élément de la boucle de régulation d'un système automatisé. Son unité centrale exécute un code basé sur la logique floue. Ce contrôleur est un système L/M comme l'indique la figure 4.7. La partie logicielle n'est rien d'autre qu'un algorithme flou qui génère des actions spécifiques en réponse aux mesures relevées par les capteurs et appliquées à l'entrée du contrôleur. Dans notre cas, chacun des sous-modules SW_i reçoit les écarts aux consignes des grandeurs suivantes : la température, la pression et l'humidité; en contre-partie, le système génère la valeur correspondante du courant électrique. La partie matérielle (figure 4.8) est composée des vecteurs de registres nécessaires pour la mémorisation des mesures et des consignes de l'opérateur homme/machine, et de leurs écarts (c'est-à-dire entre les mesures et les consignes).

4.4. Implémentation et résultats

4.4.1. Spécification des propriétés

La spécification du système est traduite en un ensemble de propriétés. Chacune d'elles est décrite par un code HLL de manière à ce qu'il soit possible de la valider en concurrence avec le reste des propriétés. Les paramètres de chacune de ces propriétés pourraient appartenir à la partie matérielle seule (il s'agit de propriétés du matériel), à la partie logicielle seule (il s'agit de propriétés du logiciel) ou à toutes les deux (il s'agit de propriétés mixtes L/M). Pour chaque vecteur de test appliqué au modèle du système, toutes les propriétés considérées sont exécutées pour décider de leur validité (valide ou bien invalide). Des propriétés du contrôleur présenté à la section 4.3 sont listées ci-dessous avec leurs pseudo-codes :

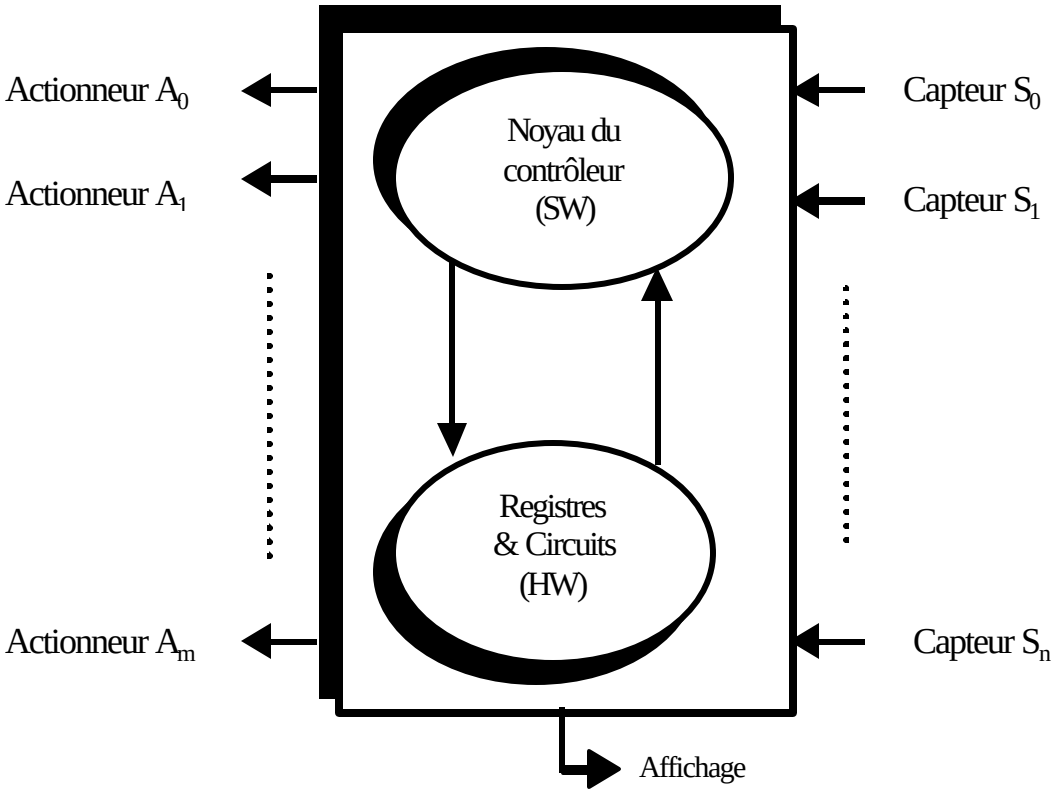


Figure 4.7 : Contrôleur à logique floue

Exemple de propriétés

P_1 - Each measure must be updated every T_c period

P_2 - Each action must be updated every T_a period

*P_3 - Always **not** (high_current **and** very_hot)*

Leurs expressions en HLL

/ Thread Body of P_1 */*

{…

while Tc_spent

{ read_measures();

test_update_measures();}

…}

/ Thread Body of P_2 */*

{…

while Ta_spent

{ read_actions();

test_update_actions();}

…}

/ Thread Body of P_3 */*

{…

*assertion1=**not** (high_current **and** very_hot);*

test_assertion (assertion1);

…}

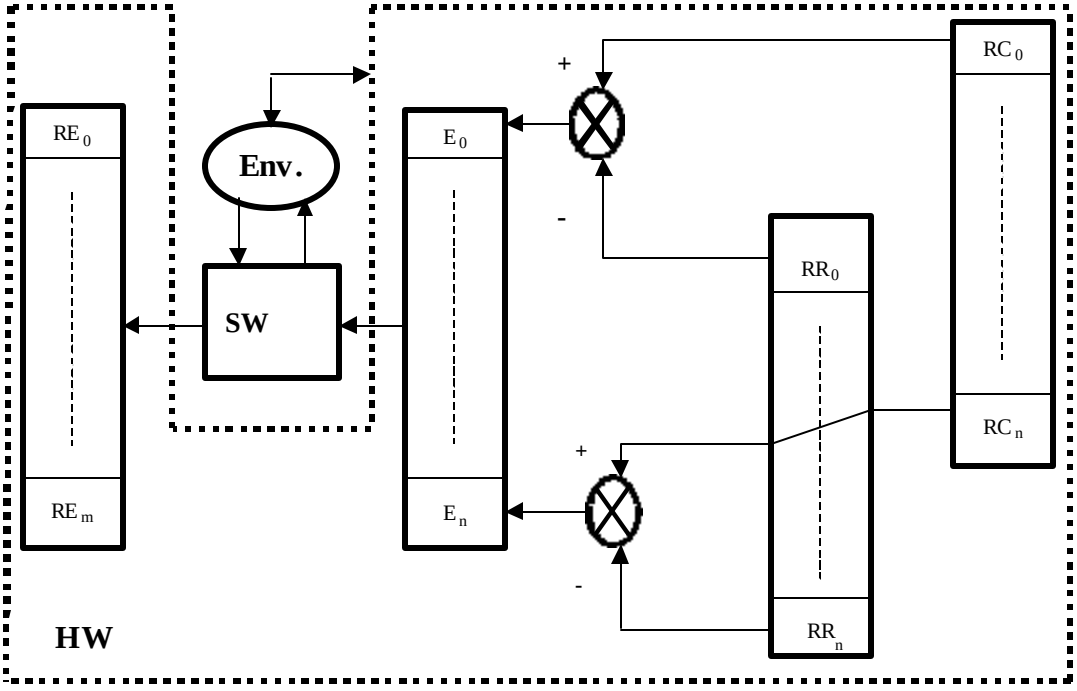


Figure 4.8 : Description de la partie matérielle

4.5.2. Structure du code

Le code L/M est organisé comme l'indique la figure 4.9. Nous avons utilisé un modèle unifié pour tout le système afin de mener une simulation fonctionnelle. Sous cette optique, la partie matérielle est perçue comme étant un ensemble de registres accessibles en lecture et/ou en écriture. Dans notre exemple, la partie logicielle est répartie en un nombre de sous-modules indépendants SW_i décrits chacun par un thread correspondant. Par ailleurs, la partie matérielle est contrôlée par d'autres threads. Les propriétés L/M sont casées dans des threads tout en tenant compte de leurs dépendances.

Nous avons considéré des générateurs aléatoires pour produire, à chaque période d'échantillonnage, des mesures des grandeurs à capter : la température, l'humidité et la pression. Les écarts de ces mesures par rapport à leurs consignes sont stockés dans des registres accessibles (figure 4.8).

Nous avons réparti la partie logicielle originale en quatre threads dont trois s'occupent chacun du traitement relatif à l'une des trois grandeurs (température, humidité et pression). Le quatrième thread calcule l'intensité du courant électrique en fonction des écarts des trois autres grandeurs et l'écrit dans le registre équivalent. Les propriétés P_1, P_2 et P_3 sont décrites chacune dans un thread. P_1 est une propriété du matériel (capteurs), P_2 est une propriété du logiciel (algorithme) et P_3 est une propriété L/M. P_1 et P_2 sont évaluées respectivement aux bouts des intervalles de temps T_c et T_a, où T_c est la période des instants de lecture des mesures des trois grandeurs et T_a est la période des instants d'application de la commande (courant électrique). P_3 doit être vraie à tout instant, elle exprime le fait que le contrôleur ne doit jamais générer une intensité forte du courant ($\in$ l'intervalle donné *high_current* = $[I_{max}, \infty]$) si la température actuelle est assez élevée ($\in$ l'intervalle donné *very_hot* = $[T_{max}, \infty]$).

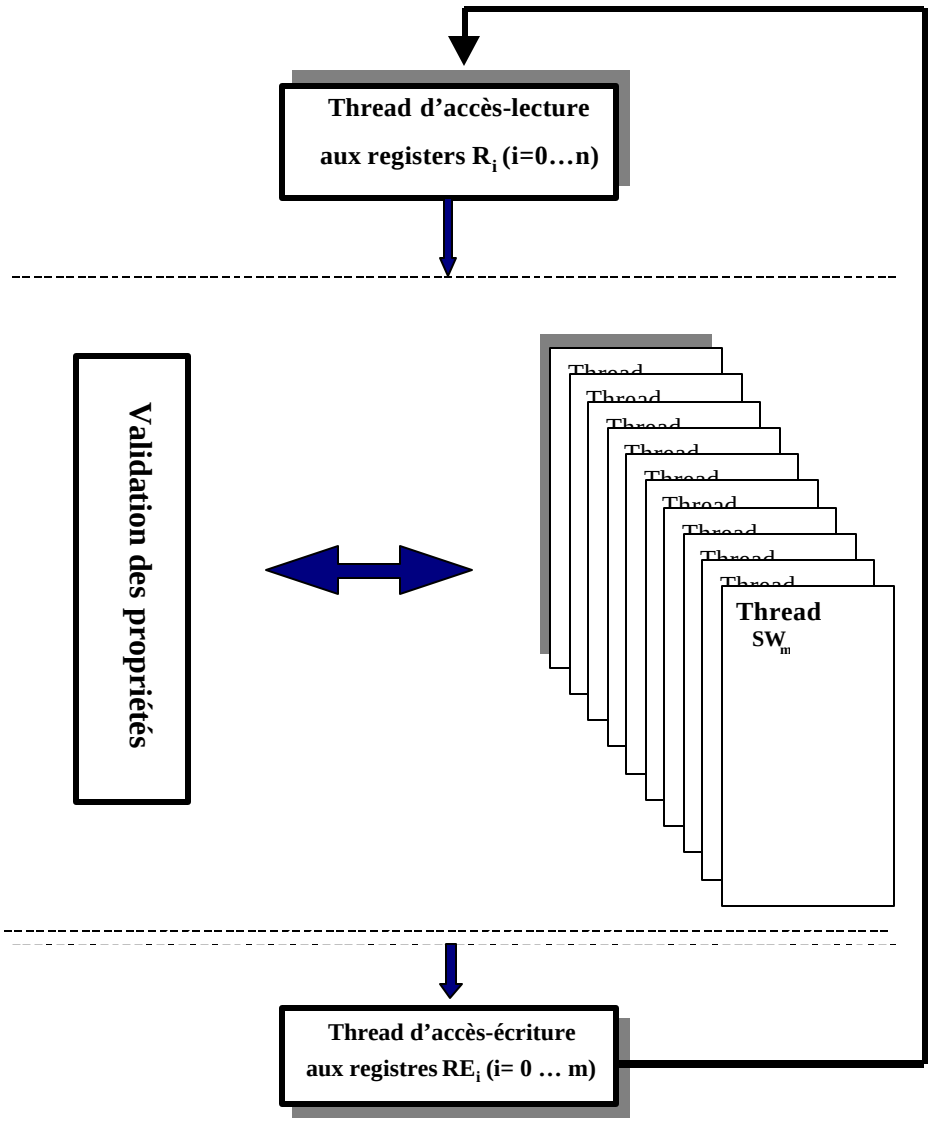


Figure 4.9 : Vue globale de la partie logicielle étendue

4.5.3. Simulation et test

Pour effectuer la simulation de notre contrôleur, nous avons généré des vecteurs de test de manière aléatoire. Un vecteur de test dans notre exemple n'est rien d'autre qu'un triplet composé des valeurs instantanées des trois grandeurs, la température, l'humidité et la pression. Au bout de chaque période prédéfinie T_c, ces

vecteurs, à tour de rôle, sont appliqués à l'entrée du système et ensuite les propriétés sont évaluées, soit valides (1), soit invalides (0). Une batterie de plus de mille vecteurs de test a été considérée pour les propriétés mentionnées à la section 4.4.1, et les résultats sont illustrés par les figures 4.10-(a), 4.10-(b) et 4.10-(c).

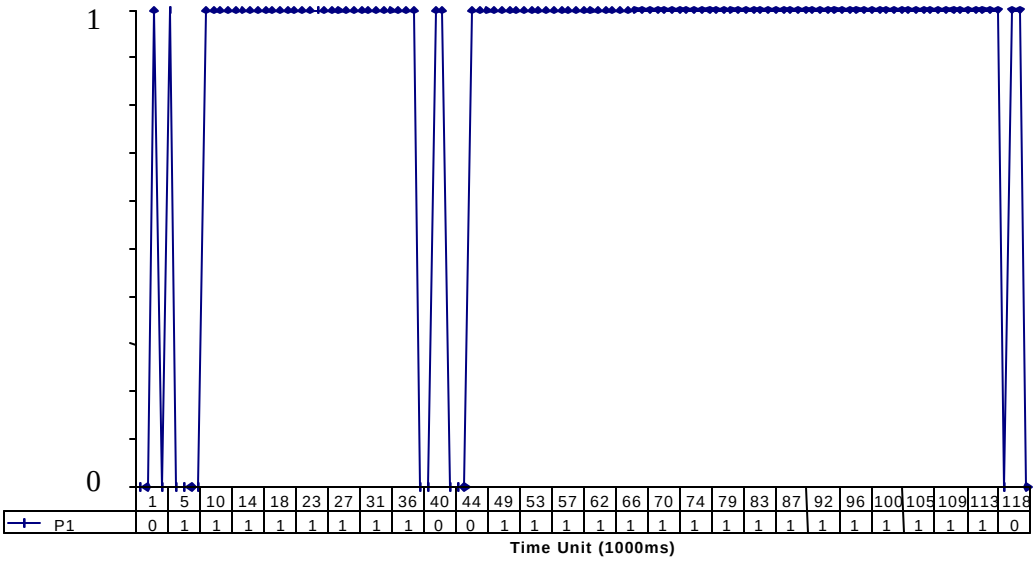
Ces figures présentent les états de validation des trois propriétés P_1, P_2 et P_3 en réaction à l'application des différents vecteurs de test. P_1 par exemple est non satisfaite (sa valeur est nulle) par les vecteurs de test d'indices 1, 40, 44 et d'autres. P_2 est non validée par exemple par les vecteurs d'indices 1, 43, 66 et d'autres. P_3 est aussi non satisfaite par certains vecteurs tels que 18, 35 et 104. Les causes d'invalidation de ces propriétés face à certains vecteurs de test peuvent éventuellement provenir soit des vecteurs de test qui sont en dehors des marges admises par la spécification du système (première cause) soit des erreurs au sein du code du contrôleur (deuxième cause) soit de la non convenance de l'architecture de la machine hôte à l'exécution des applications à threads multiples (troisième cause).

Selon la spécification, lorsqu'un vecteur de test excède la marge permise, les registres correspondants ne sont pas réécrits et gardent en conséquence leurs valeurs précédentes. Ce qui fait que la mise à jour des registres dans ce cas n'a pas eu lieu, alors la propriété P_1 serait invalide.

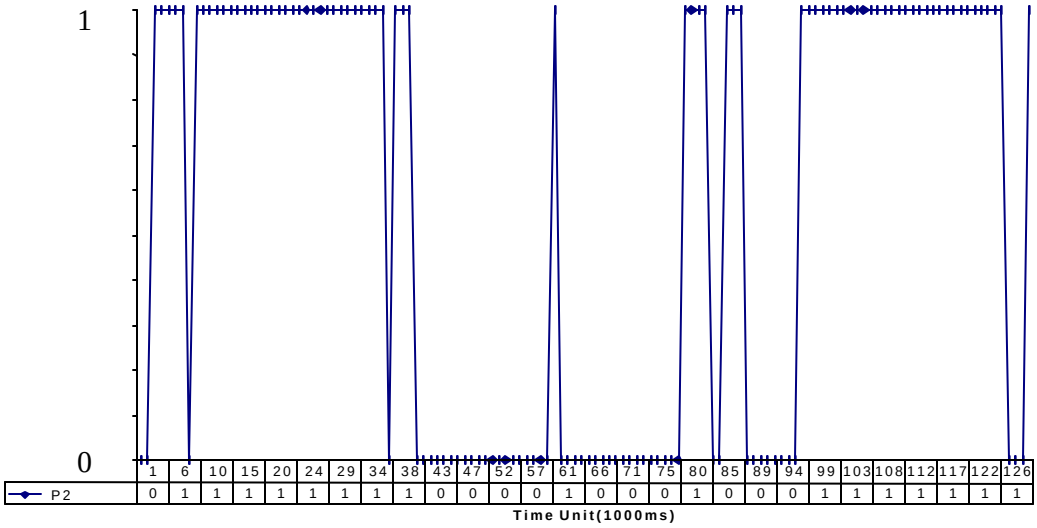
De même, si les calculs de l'intensité du courant électrique aboutissent à une valeur interdite par la spécification, alors aucune réécriture n'est faite et le registre correspondant conserve son contenu. Dans ce cas, la propriété P_2 est jugée invalide. En outre, si le code du contrôleur ne respecte pas les périodes T_c et T_a pour lire les mesures et écrire les actions (on suppose que le temps de calcul est négligeable), les propriétés seront invalides ou mal évaluées.

Afin d'éliminer les erreurs émanant de la première cause, il faut réécrire les propriétés concernées pour qu'elles soient en concordance complète avec la spécification (surtout pour les conditions aux limites et les exceptions). Quant au remède de la deuxième cause, il faut réviser notamment le code des threads en relation avec les propriétés invalides. La troisième cause pourrait entraver tout

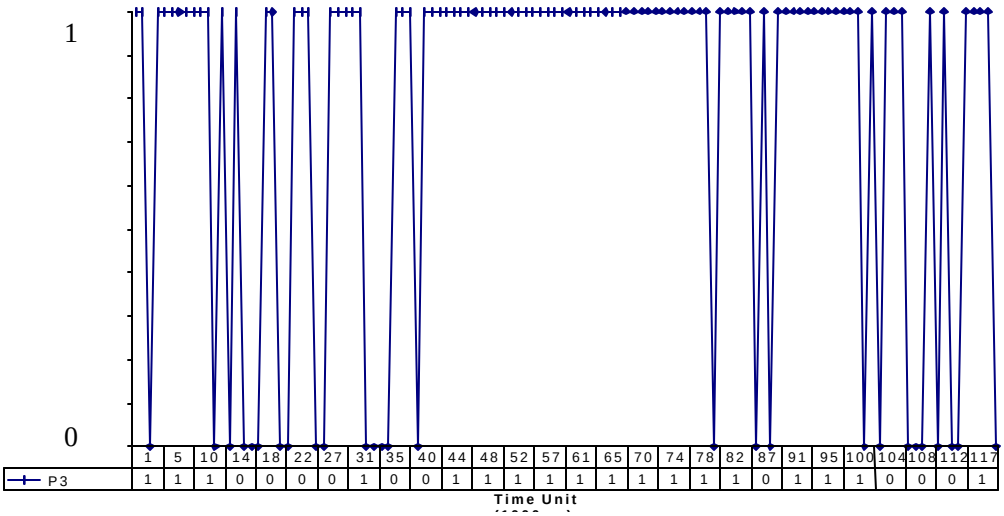
développement dans le domaine du « multithreading ». Ceci est le cas lorsque le nombre de threads mis en jeu est assez important et le temps d'attente d'un thread avant de se voir passer en exécution (dans le cas d'une machine monoprocesseur) devient en conséquence assez long. Un remède pour éliminer cette cause est d'utiliser des architectures dédiées à ce genre d'applications ou au moins les exécutent avec succès. Récemment, Sun Microsystems a annoncé la fabrication de processeurs spécialisés très performants pour les applications Java (type processeur Java). Cela pourrait résoudre ce problème mais aucune expérimentation n'est faite pour en confirmer.



(a) : Validation de la propriété P1 vs différents vecteurs de test



(b) : Validation de la propriété P2 vs différents vecteurs de test



(c) : Validation de la propriété P3 vs différents vecteurs de test

Figure 4.10 : Validation des propriétés vs des vecteurs de test

Remarques

- La référence du temps, l'horloge, qui a été arbitrairement utilisée dans l'implémentation ci-dessus est l'horloge de la machine hôte. Alors, la cadence de la simulation sera rigidement dépendante de celle de l'horloge de la machine. Cela entraînera une consommation de temps réel inutile. Pour cette raison, nous avons implémenté nos propres objets « *Clock* » dont les détails sont fournis au chapitre 5.

- Une machine dont l'architecture est dédiée à l'exécution des applications basées sur le « multithreading », serait la machine idéale pour la mise en œuvre de notre méthodologie de covérification.

4.6. Conclusion

Dans ce chapitre, nous avons présenté une méthodologie de covérification basée sur le concept du «multithreading» et sur la technologie de cosimulation. Cette approche permet une bonne structuration du code logiciel et une flexibilité particulière pour définir et superviser les propriétés L/M. La manipulation de ces dernières est rendue facile; elles sont toutes testées du côté logiciel, y compris celles concernant la partie matérielle. En plus, en appliquant cette technique, nous pouvons guider le processus de cosimulation et permettre une tolérance aux fautes en définissant bien les propriétés du système. Cette technique peut accélérer la tâche de covérification dans la mesure où la machine hôte est un système multiprocesseur à mémoire partagée, sachant que les machines séquentielles (modèle de Von Newmann) ne se prêtent pas bien à la programmation concurrente. Notre approche est étroitement liée à la cosimulation; elle constitue une contribution non négligeable dans le domaine de la covérification L/M, et d'autant plus qu'il y a absence de méthodes formelles spécifiques.

Chapitre 5

Simulations séquentielle et distribuée basées sur des threads

5.1. Introduction

Théoriquement, la simulation distribuée possède une multitude d'avantages par rapport à la simulation séquentielle. Cependant, son application n'est pas évidente et dépend potentiellement de la disponibilité des composants matériels et logiciels qui permettent la mise en exécution de ce genre de techniques. Les threads communicants du langage Java peuvent être considérés pour concevoir des simulateurs distribués. Ces derniers seront portables puisque Java est indépendant de l'environnement de la machine hôte. En outre, un simulateur à base de threads pourrait bénéficier de la puissance d'une machine multiprocesseur dédiée à ce type d'applications (si disponible) afin de satisfaire les contraintes de performance. Dans ce cas (d'une machine multiprocesseur), les threads indépendants sont exécutés en parallèle sur plusieurs processeurs; tandis que dans le cas d'une architecture monoprocesseur, tous les threads sont ordonnancés pour s'exécuter sur un seul processeur. Un gain de performance est possible avec l'usage d'une architecture

parallèle, mais ceci reste tributaire du degré de dépendance entre les threads et aussi de la capacité du système d'exploitation à affecter, de manière optimale, les threads actifs sur les processeurs disponibles. Ce dernier problème ne fait pas l'objet de notre thèse et reste donc ouvert aux progrès futurs. Par ailleurs, la tâche la plus importante de la simulation distribuée consiste en la gestion des événements et du temps. La manipulation des événements doit prendre en considération leurs instants d'occurrence et l'avancement des horloges.

Le processus de simulation a fait l'objet de plusieurs publications. Nous sommes particulièrement intéressé par les références [102, 96]. Ces derniers résument la théorie de la simulation à événements, de laquelle nous sommes inspirés pour insérer les threads de Java (voir chapitre 4) dans ce genre de simulation. Récemment, Java a sensiblement suscité l'intérêt des chercheurs et il a été effectivement utilisé pour décrire des systèmes L/M [95, 96]; cependant, les problèmes relevant de la gestion du temps et des événements n'ont pas été traités.

Nous avons implémenté les algorithmes des deux techniques de simulation, séquentielle et distribuée, au moyen des threads de Java2; un «package » appelé *covérification* a été développé pour cette fin. Ce « package » regroupe un ensemble d'objets et de méthodes tels que l'horloge et ses méthodes, la queue et ses méthodes, etc.

Ce chapitre est organisé comme suit : les sections 5.2 et 5.3 exposent respectivement les simulations séquentielle et distribuée ainsi que leurs algorithmes. Nous décrivons dans la section 5.4, l'implémentation de ces algorithmes au moyen des threads de Java. Avant de conclure, nous discutons dans la section 5.5 la gestion des horloges, les difficultés rencontrées et la validité de ces algorithmes.

5.2. Simulation séquentielle (SS)

La covérification est basée sur la technique de simulation/cosimulation. Dans notre travail, nous introduisons les threads comme objets animateurs de cette dernière. Comment gérer le « timing » et l'échange des messages entre les threads? Une telle question est très importante pour le déroulement correct d'une simulation. Pour y répondre, nous avons mené l'étude suivante afin de mettre au clair la faisabilité d'une telle simulation et les difficultés de sa réalisation.

5.2.1. Notation

La notation suivante est utilisée dans le reste de ce chapitre :

- T_i : un thread *i*.
- L_i : la liste des événements de T_i
- $M_i = (m, t)_i$: un message *i* dont le contenu est m et l'instant d'occurrence est *t*.
- $(M_j, T_j) \in L_i$: signifie que le message $M_j = (m, t)_j$ doit être envoyé au thread T_j par le thread T_i (*Li* est associé à *Ti*) à l'instant *t*.
- $L_i .M_j$: le message M_j de la liste L_i.
- $C_{A \to B}$: le canal qui véhicule le message du thread envoyeur *A* au thread receveur *B*.
- *Clock* : l'horloge globale du système en cours de simulation.
- $Clock_i$: l'horloge locale de T_i.

5.2.2. Algorithme SS

L'algorithme de la simulation séquentielle (SS) consiste à simuler le fonctionnement d'un système à l'aide d'un ensemble de threads communiquant entre eux via un seul canal. Ceci signifie que le processus de communication est sérialisé; tous les messages sont ordonnancés selon la cadence d'une horloge globale. L'algorithme SS est présenté ci-dessous en pseudo-code de Java :

Algorithme SS :

// Début

1: User_Constraint_ or_ end end_of_SS;

2: Init

{

4: Clock=0;

5: Déclarer et créer le canal de simulation: Channel_SS;

6: Déclarer et créer les différents threads {T_1, T_2, T_3,…};

7: Timed_ordered_messages_list L_i = {(M_j,T_j) / où le message M_j=(m,t)$_j$ est composé de son contenu m et du temps de son occurrence t, le thread T_j est le receveur (l'émetteur par défaut est T_i)};

// L_i appartient au corps de T_i , c'est une liste locale.

}

11: while (not (end_of_SS))

{

// Éxecuter tous les threads T_1, T_2,…

14: Produire les messages et les ranger dans L_i ;

15: If (Sending)

{

16: Retirer et transmettre le message M_j=(m,t)$_j$ à l'instant t_j via channel_SS pour chaque liste L_i de chaque thread T_i selon l'horloge globale Clock;

18: Clock = t_j;

}

20: if (Receiving)

Recevoir M_l;

}

// Fin

Interprétation de SS

À la ligne 1, nous spécifions la variable *end_of_SS* qui va jouer le rôle d'indicateur de fin de simulation. Le bloc *Init* renferme toutes les initialisations et les déclarations nécessaires pour la simulation séquentielle. À la ligne 4, nous initialisons la variable d'horloge *Clock* à 0. Nous créons les objets *Channel_SS* et les threads ainsi que leurs listes d'événements aux lignes 5, 6 et 7. Le déroulement de la simulation est décrit par les lignes allant de 11 à la fin. Tant que la condition d'arrêt de la simulation est non vraie, tous les threads mis en jeu sont actifs. Les événements produits par chacun d'eux sont rangés par ordre dans sa liste d'événements. Un événement dont l'instant est le plus petit, est émis au premier et *Clock* est avancée à cet instant. *Clock* est une horloge globale qui est visible par tous les threads.

5.2.3. Exemple d'une ligne d'assemblage

Il s'agit d'une ligne d'assemblage composée de plusieurs unités de travail situées dans une chaîne de production donnée. Ces unités s'échangent des services (produits) auxquels nous associons des événements (messages). Dans cet exemple, nous considérons cinq unités dont *Source*, *A*, *B*, *C* et *Sortie* (figure 5.1). Chacune d'elles est représentée par un thread. Alors, les threads mis en jeu sont {*Source*, *A*, *B*, *C*, *Sortie*} et leurs listes d'événements respectives sont les suivantes :

$$L_{Source} = \{((-, 5), A), ((-, 7), A), ((-, 30), A), ((-, 32), A)\}$$
$$L_{A} = \{((-, 9), B), ((-, 19), B), ((-, 31), B), ((-, 37), B)\}$$
$$L_{B} = \{((-, 21), C), ((-, 36), C), ((-, 38), C), ((-, 45), C)\}$$
$$L_{C} = \{((-, 23), Sortie), ((-, 39), Sortie), ((-, 40), Sortie), ((-, 49), Sortie)\}$$
$$L_{Sortie} = \{\}$$

Les résultats de cette simulation sont présentés dans le tableau 5.1. Le premier événement a eu lieu à l'instant 5 et suivent les autres à raison d'un par cycle d'horloge. Le dernier message a été émis à l'instant 49.

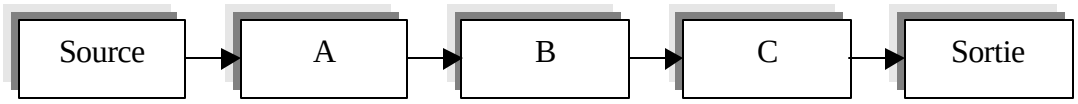


Figure 5.1 : Ligne d'assemblage

Horloge	**L_{Source}**	**L_A**	**L_B**	**L_C**	**Message envoyé**	**Thread émetteur**
<5	((-, 5), A) ((-, 7), A) ((-, 30), A) ((-, 32), A)	((-, 9), B) ((-, 19), B) ((-, 31), B) ((-, 37), B)	((-, 21), C) ((-, 36), C) ((-, 38), C) ((-, 45), C)	((-, 23),Sortie) ((-, 39),Sortie) ((-, 40),Sortie) ((-, 49),Sortie)	-	-
5	((-, 5), A) ((-, 7), A) ((-, 30), A) ((-, 32), A)	((-, 9), B) ((-, 19), B) ((-, 31), B) ((-, 37), B)	((-, 21), C) ((-, 36), C) ((-, 38), C) ((-, 45), C)	((-, 23),Sortie) ((-, 39),Sortie) ((-, 40),Sortie) ((-, 49),Sortie)	((-, 5), A)	Source
7	((-, 7), A) ((-, 30), A) ((-, 32), A)	((-, 9), B) ((-, 19), B) ((-, 31), B) ((-, 37), B)	((-, 21), C) ((-, 36), C) ((-, 38), C) ((-, 45), C)	((-, 23),Sortie) ((-, 39),Sortie) ((-, 40),Sortie) ((-, 49),Sortie)	((-, 7), A)	Source
9	((-, 30), A) ((-, 32), A)	((-, 9), B) ((-, 19), B) ((-, 31), B) ((-, 37), B)	((-, 21), C) ((-, 36), C) ((-, 38), C) ((-, 45), C)	((-, 23),Sortie) ((-, 39),Sortie) ((-, 40),Sortie) ((-, 49),Sortie)	((-, 9), B)	A
19	((-, 30), A) ((-, 32), A)	((-, 19), B) ((-, 31), B) ((-, 37), B)	((-, 21), C) ((-, 36), C) ((-, 38), C) ((-, 45), C)	((-, 23),Sortie) ((-, 39),Sortie) ((-, 40),Sortie) ((-, 49),Sortie)	((-, 19), B)	A
21	((-, 30), A) ((-, 32), A)	((-, 31), B) ((-, 37), B)	((-, 21), C) ((-, 36), C) ((-, 38), C) ((-, 45), C)	((-, 23),Sortie) ((-, 39),Sortie) ((-, 40),Sortie) ((-, 49),Sortie)	((-, 21), C)	B
23	((-, 30), A) ((-, 32), A)	((-, 31), B) ((-, 37), B)	((-, 36), C) ((-, 38), C) ((-, 45), C)	((-, 23),Sortie) ((-, 39),Sortie) ((-, 40),Sortie) ((-, 49),Sortie)	((-, 23), Sortie)	C
30	((-, 30), A)	((-, 31), B)	((-, 36), C)	((-, 39),Sortie)	((-, 30), A)	Source

	((-, 32), A)	((-, 37), B)	((-, 38), C) ((-, 45), C)	((-, 40),Sortie) ((-, 49),Sortie)		
31	((-, 32), A)	((-, 31), B) ((-, 37), B)	((-, 36), C) ((-, 38), C) ((-, 45), C)	((-, 39),Sortie) ((-, 40),Sortie) ((-, 49),Sortie)	((-, 31), B)	A
32	((-, 32), A)	((-, 37), B)	((-, 36), C) ((-, 38), C) ((-, 45), C)	((-, 39),Sortie) ((-, 40),Sortie) ((-, 49),Sortie)	((-, 32), A)	Source
36	-	((-, 37), B)	((-, 36), C) ((-, 38), C) ((-, 45), C)	((-, 39),Sortie) ((-, 40),Sortie) ((-, 49),Sortie)	((-, 36), C)	B
37	-	((-, 37), B)	((-, 38), C) ((-, 45), C)	((-, 39),Sortie) ((-, 40),Sortie) ((-, 49),Sortie)	((-, 37), B)	A
38	-	-	((-, 38), C) ((-, 45), C)	((-, 39),Sortie) ((-, 40),Sortie) ((-, 49),Sortie)	((-, 38), C)	B
39	-	-	((-, 45), C)	((-, 39),Sortie) ((-, 40),Sortie) ((-, 49),Sortie)	((-, 39), Sortie)	C
40	-	-	((-, 45), C)	((-, 40),Sortie) ((-, 49),Sortie)	((-, 40), Sortie)	C
45	-	-	((-, 45), C)	((-, 49),Sortie)	((-, 45), C)	B
49	-	-	-	((-, 49),Sortie)	((-, 49), Sortie)	C
>49	-	-	-	-	-	-

Tableau 5.1 : Étapes d'exécution de l'exemple 5.2.3 en appliquant l'algorithme SS

5.3. Simulation distribuée (DS)

En plus de ce qu'il a été précisé dans la section précédente, la covérification peut faire appel, selon la nature du système étudié, à la technique de simulation distribuée à base de threads. Dans ce cas, chacun des threads aura sa propre horloge

et l'échange des messages devient très rigoureux. Nous présentons dans la suite l'algorithme DS que nous avons implémenté ainsi que les résultats obtenus.

5.3.1. Algorithme DS

// Début

1: User_Constraint_ or_ end end_of_DS;

2: Init

{

4: Déclarer et créer les différents threads {T_1, T_2, T_3,…};

// L_i appartient au corps de T_i , c'est une liste locale.

6: Timed_ordered_messages_list L_i= {(M_j,T_j) où le message M_j=(m,t)$_j$ est composé de son contenu m et du temps de son occurrence t, le thread T_j est le receveur (l'émetteur par défaut est T_i)};

// for i=0 to n, n est le nombre de threads

10: Initialiser toutes les horloges locales $clock_i = 0^+$;

11: Déclarer et créer les canaux {C_1, C_2,…} pour tous les threads émetteurs;

}

13: while (not (end_of_DS))

{

// Éxecuter tous les threads T_1, T_2,…

16: for (each thread)

{

18: if (sending)

{

20: Retirer et transmettre le message "earliest" M_j=(m,t)$_j$ à l'instant t_j via C_i pour chaque liste L_i de chaque thread T_i selon l'horloge locale $clock_i$;

23: $clock_i = t_j$;

}

25: if (receiving)

```
                    {
                      27: Recevoir le message Mj=(m,t)j ;
                      28 : clocki = tj;
                    }
        }
}
// Fin
```

Interprétation de DS

Nous commençons par créer les objets nécessaires au sein du bloc *Init*. Toutes les horloges $clock_i$ des différents threads sont initialisées. Ensuite, chaque thread fait avancer son horloge locale, au cours de la simulation, aux instants d'émission des événements. De même, à la réception d'un événement, l'horloge locale du thread récepteur est avancée à son instant et les événements éventuellement « plus tôt » sont émis en conséquence.

5.3.2. Exemple d'application

Nous appliquons cet algorithme sur le même exemple de la section 5.2.3 afin de voir la différence entre les résultats des deux algorithmes, SS et DS. L'exécution obtenue est résumée par le tableau 5.2. Il est bien visible que cette simulation s'est effectuée en 7 cycles au lieu de 16 cycles en simulation séquentielle (tableau 5.1). Ce qui montre dans ce cas que la technique de simulation distribuée a réduit de moitié (ou presque) le temps d'exécution obtenu en simulation séquentielle. Cependant, une telle performance n'est pas toujours garantie et elle est tributaire de la nature du système étudié si sa structure se prête très bien ou non à l'application d'une simulation distribuée.

Étape	**L_{Source}**	**L_A**	**L_B**	**L_C**	**Messages envoyés**	**Thread émetteur**
0	((-, 5), A) ((-, 7), A) ((-, 30), A) ((-, 32), A)	((-, 9), B) ((-, 19), B) ((-, 31), B) ((-, 37), B)	((-, 21), C) ((-, 36), C) ((-, 38), C) ((-, 45), C)	((-, 23),Sortie) ((-, 39),Sortie) ((-, 40),Sortie) ((-, 49),Sortie)	-	-
1	((-, 5), A) ((-, 7), A) ((-, 30), A) ((-, 32), A)	((-, 9), B) ((-, 19), B) ((-, 31), B) ((-, 37), B)	((-, 21), C) ((-, 36), C) ((-, 38), C) ((-, 45), C)	((-, 23),Sortie) ((-, 39),Sortie) ((-, 40),Sortie) ((-, 49),Sortie)	((-, 5), A) ((-, 7), A)	Source
2	((-, 30), A) ((-, 32), A)	((-, 9), B) ((-, 19), B) ((-, 31), B) ((-, 37), B)	((-, 21), C) ((-, 36), C) ((-, 38), C) ((-, 45), C)	((-, 23),Sortie) ((-, 39),Sortie) ((-, 40),Sortie) ((-, 49),Sortie)	((-, 9), B) ((-, 30), A)	A Source
3	((-, 32), A)	((-, 19), B) ((-, 31), B) ((-, 37), B)	((-, 21), C) ((-, 36), C) ((-, 38), C) ((-, 45), C)	((-, 23),Sortie) ((-, 39),Sortie) ((-, 40),Sortie) ((-, 49),Sortie)	((-, 19), B) ((-, 32), A) ((-, 21), C)	A Source B
4	-	((-, 31), B) ((-, 37), B)	((-, 36), C) ((-, 38), C) ((-, 45), C)	((-, 23),Sortie) ((-, 39),Sortie) ((-, 40),Sortie) ((-, 49),Sortie)	((-, 31), B) ((-, 37), B)	A A
5	-	-	((-, 36), C) ((-, 38), C) ((-, 45), C)	((-, 23),Sortie) ((-, 39),Sortie) ((-, 40),Sortie) ((-, 49),Sortie)	((-, 36), C) ((-, 23),Sortie)	B C
6	-	-	((-, 38), C) ((-, 45), C)	((-, 39),Sortie) ((-, 40),Sortie) ((-, 49),Sortie)	((-, 38), C) ((-, 39),Sortie)	B C
7	-	-	((-, 45), C)	((-, 40),Sortie) ((-, 49),Sortie)	((-, 45), C) ((-, 40),Sortie) ((-, 49),Sortie)	B C
8	-	-	-	-	-	-

Tableau 5.2 : Étapes d'exécution de l'exemple 5.2.3 en appliquant l'algorithme DS

Remarque

Nous avons toutefois détecté et résolu un problème dû à l'arrêt des horloges locales des threads. Nous en donnons plus de détails à la section 5.5.2.

5.4. Validité des algorithmes SS et DS

En général, pour simuler un système physique, on fait appel à un simulateur. Ce dernier est un système logique composé de processus (threads dans notre cas). On dit qu'un système logique simule correctement un système physique s'il est possible pour le système logique de prédire la séquence exacte des transmissions de messages dans le système physique [102]. Il se pourrait que le système simulant ne reproduit pas tous les détails du système simulé et ne s'exécute pas à la même vitesse de ce dernier; cependant, l'ordre des événements à transmettre doit être obligatoirement respecté. De manière formelle, si t_1, t_2, …, t_n sont respectivement les instants de transmission des messages m_1, m_2, …, m_n dans le système physique et $t_1 \leq t_2 \leq \ldots \leq t_n$, alors le système logique devrait être capable de générer la séquence ordonnée suivante : $\{(t_1, m_1), (t_2, m_2), \ldots, (t_n, m_n)\}$. Nous considérons les deux théorèmes suivants [102] comme preuves formelles de validité des deux algorithmes SS et DS. Le premier démontre que la simulation séquentielle basée sur la stratégie de SS est correcte; de même, le second prouve que la simulation distribuée implémentée par DS est correcte.

Théorème (cas de SS) : Soit (t, m) un élément de la liste des événements tel que $t<t'$ pour tout autre élément (t', m') de la liste des événements. Alors, le message m est transmis à l'instant t dans le système physique et aucun message n'est transmis à un instant t'', où $Clock \leq t'' < t$.

Preuve : Si le message m n'est pas transmis à l'instant t, cela doit être parce qu'un autre message a été transmis à un instant t'', où $Clock \leq t'' \leq t$ et par conséquent, la

transmission de *m* est annulée. Or, ceci contredit le fait que (*t*, *m*) est l'événement à transmettre le plus tôt (« *the earliest event* »). ÿ

Théorème (cas de DS) : La simulation est correcte à tout point.

Preuve : La simulation est clairement correcte, par définition, quand il n'y a aucun message à transmettre durant la simulation (*cas 0*). Supposons qu'une simulation est correcte jusqu'à un point donné (*cas n-1*). Le message prochain m_i dans la simulation est correctement transmis par un thread T_i . Puisque la simulation était correcte jusqu'à la transmission de m_i , alors T_i a reçu des séquences d'entrée correctes. D'où, la simulation incluant le message m_i est correcte (*cas n*). Ainsi, par induction, la simulation globale est correcte. ÿ

5.5. Implémentation

5.5.1. Structure

Nous avons implémenté les deux algorithmes SS et DS en Java (version 1.2). Le moteur de l'application est principalement basé sur la gestion des threads communicants et concurrents. Nous avons développé en Java un « *package* » nommé *coverification* qui encapsule une multitude de classes et d'objets tels que les horloges, les listes d'événements, les messages, etc. Ce module doit être importé au programme principal pour permettre l'usage de ses éléments.

Dans cette implémentation, chaque thread possède sa propre liste d'événements. Cette dernière est organisée en deux sous-listes : la première, *ls*, abrite tous les messages à émettre par le thread propriétaire; la seconde, *lr*, recueille tous les messages émis par les autres threads au thread propriétaire (figure 5.2). Les messages à envoyer sont spécifiés par l'utilisateur ou éventuellement produits par le thread propriétaire dépendamment de la fonctionnalité locale décrite par ce dernier. L'utilisateur de notre code est donc invité à définir le comportement des différentes entités du système étudié en réécrivant une méthode pré-baptisée *behavior()*. Cette

dernière appartient à une classe mère qui définit la communication globale entre les threads et gère la sauvegarde des résultats générés.

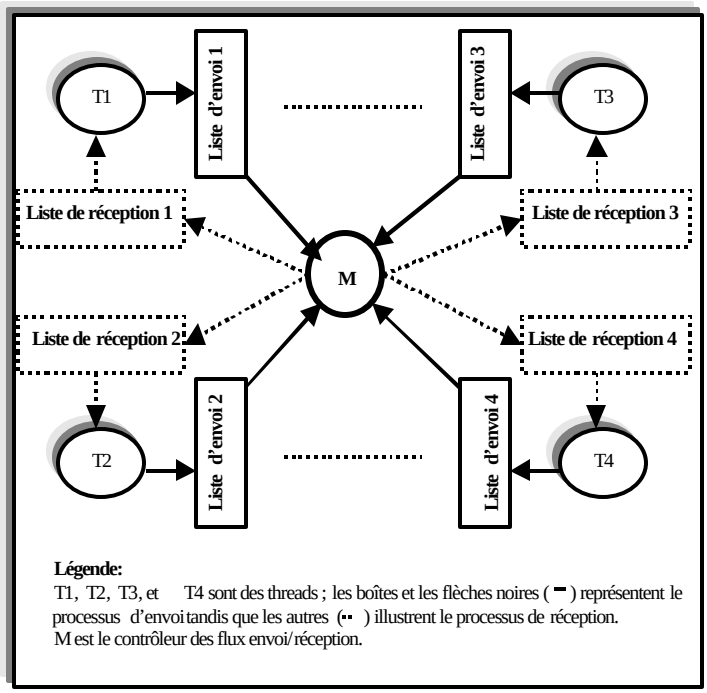


Figure 5.2 : Vue globale de l'implémentation

5.5.2. Gestion des horloges

Dans le cas de la simulation séquentielle, nous considérons juste une horloge pour cadencer tout le processus, c'est l'horloge globale *Clock*. Les événements ont donc lieu à leurs instants prévus à l'échelle de cette horloge. Le message dont l'instant d'envoi est le plus petit de tous les instants des messages présents dans les listes, il est émis en premier et selon la même règle, sont envoyés les autres. Quant à la réception des messages, supposons que le délai de voyage d'un message migrant est réduit à 0, alors les instants d'émission et de réception seront pareils. L'horloge globale évolue en passant par tous les instants d'occurrence des messages destinés à la communication inter threads (tableau 5.1).

Dans le cas de la simulation distribuée, nous considérons pour chaque thread sa propre horloge. Un thread émetteur fait ses tâches locales et envoie éventuellement des messages aux autres threads. La règle d'envoi ici est moins restreinte, il faut juste respecter le principe de cause-effet, c'est-à-dire un message *effet* ne doit pas être envoyé avant un message *cause* (tableau 5.2). Par incrémentation, l'horloge de chaque thread passe par tous les instants d'occurrence des messages de celui-ci.

5.5.3. Exemple d'exécution

L'exemple (figure 5.4) dont l'exécution est illustrée par les figures 5.5 et 5.6, est décrit comme suit : Les threads mis en jeu sont T_1, T_2 et T_3 et leurs listes d'événements respectives sont L_1, L_2 et L_3.

L_1 = {(1, l, T3), (10, m, T2), (69, k, T3)}
L_2 = {(5, n, T1), (16, l, T3), (35, n, T1)}
L_3 = {(3, d, T2), (6, s, T1), (45, k, T2)}

Le thread T_1 envoie des messages aux threads T_3, T_2 et encore à T_3 aux instants respectifs 1, 10 et 69. T_2 émet des messages aux threads T_1, T_3 et encore à T_1 aux instants respectifs 5, 16 et 35. T_3 émet des messages aux threads T_2, T_1 et encore à T_2 aux instants respectifs 3, 6 et 45. La figure 5.5 montre la chronologie des échanges de messages entre les trois threads dans le cas de la simulation séquentielle. Le temps est décrit juste par une seule horloge et l'occurrence des événements suit l'ordre croissant de cette horloge. Par ailleurs, les résultats de la simulation distribuée sont présentés par la figure 5.6. $Clock_1$, $Clock_2$ et $Clock_3$ sont respectivement les horloges de T_1, T_2 et T_3.

5.6. Discussion

5.6.1. SS ou DS?

L'application des deux algorithmes sur le même exemple d'une ligne d'assemblage montre que (dans ce cas) la simulation distribuée l'emporte sur la simulation séquentielle. Tel n'est pas toujours le cas, le succès de la simulation distribuée dépend étroitement de la nature du système étudié. Par exemple, dans le cas du système présenté par la figure 5.3, les performances d'une simulation distribuée par rapport à une simulation séquentielle seront similaires. En effet, à cause de la retroaction du circuit, la sortie actuelle du circuit S_1 est aussi l'entrée suivante à côté de l'autre entrée S_0 ($S_1^{(n)}$ = [Circuit]*[$S_1^{(n-1)}$ S_0]). Donc, pour produire $S_1^{(n+1)}$, nous aurons tout d'abord besoin de produire $S_1^{(n)}$. Cette dépendance de causalité entrave l'accélération de la simulation distribuée. Pratiquement, nous aurons les mêmes résultats qu'avec SS mais avec plus de dépenses. Généralement, l'implémentation de la simulation distribuée est coûteuse en termes de produits matériels/logiciels requis (une architecture matérielle et un environnement logiciel appropriés) et de code à développer; si ses performances sont comparables à (ou moindres que) celles de la simulation séquentielle, alors il est sage d'opter pour cette dernière. En fait, nous tenons ici à attirer l'attention du lecteur sur ce point, sans toutefois s'aventurer dans les détails. Une comparaison approfondie entre les deux techniques se base essentiellement sur l'étude de la « distribuabilité » du système à simuler sachant que le matériel/logiciel nécessaire est disponible; cependant, une telle comparaison n'a pas été abordée dans cette thèse.

5.6.2. Difficultés

- *Problème d'exclusion mutuelle* :

Étant donné que notre implémentation est lieu de concurrence de plusieurs threads, il se pose le problème d'accès simultané aux même entités. Ceci est résolu

en Java par l'utilisation du mot clé *synchronized*. Ce dernier est associé aux objets et aux méthodes susceptibles d'être partagés lors de leurs déclarations (méthodes) ou aux endroits de leurs usages (blocs).

À maintes fois, les threads mis en jeu partagent des objets et des méthodes d'une ou de plusieurs classes. Alors, il se peut que deux ou plusieurs threads veulent simultanément manipuler un objet ou utiliser une méthode d'une classe donnée. Donc, il va se poser un problème d'accès à ces éléments de manière légale et correcte. Pour rendre exclusif l'accès en Java, on associe à l'élément partagé l'attribut *synchronized*. L'instance de la classe ayant un élément *synchronized* se voit attribuer un jeton. Alors, un thread voulant manipuler un élément *synchronized*, doit acquérir le jeton de cet élément. Si un jeton est déjà utilisé par un thread, un autre thread doit se mettre en attente jusqu'à ce que le jeton soit libéré afin qu'il puisse l'utiliser. Par exemple, dans le cas de la simulation séquentielle, la méthode d'actualisation de la classe Clock est beaucoup utilisée par les threads afin de mettre à jour l'horloge, alors deux threads ne peuvent pas modifier simultanément la variable d'horloge mais ils le font un par un.

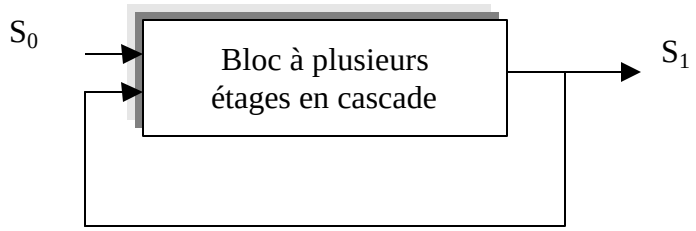


Figure 5.3 : Un circuit à «feed-back»

- *Problème d'arrêt des horloges* :

Ce problème se voit clairement sur le tableau 5.2. En effet, initialement, dès qu'un des threads a vidé sa liste d'événements, son horloge reste bloquée à l'instant du dernier événement et elle ne progresse plus; ceci pourrait entraîner un blocage (ou une attente) de tous les autres processus même s'ils n'ont pas encore achevé leurs tâches. Une telle situation est une conséquence de la règle de gestion de l'envoi des

événements : l'événement dont l'instant est le plus petit est émis en premier. Si un thread a terminé toutes les communications avec les autres threads, il voit son horloge s'arrêter à l'instant de son dernier événement consommé (émis ou reçu). Si d'autres threads ont encore des messages à transmettre, aucun de ces derniers (messages) ne va pas être perçu comme un candidat à l'émission suivante du fait que son instant est en avance sur l'horloge du thread mourant (état fin). Ce problème a été résolu en envoyant des messages vides juste pour faire avancer les horloges qui se sont arrêtées avant d'atteindre le temps maximal de simulation. Par exemple dans les exécutions présentées sur les figures 5.5 et 5.6, le dernier message du thread T_1 a eu lieu à l'instant 69, celui de T_2 à 45 et celui de T_3 à 69 mais la simulation continue jusqu'à l'instant 101.

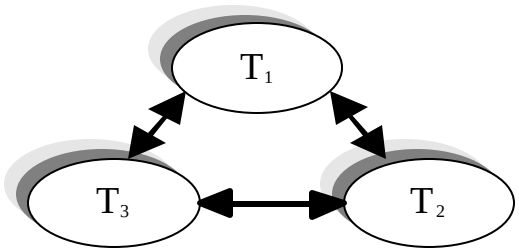


Figure 5.4 : Exemple de trois threads en communication

```
import coverification.*;
public class Class1 //Sequential Simulation
{
public static void main (String[] args)
      {
            Clock Clk=new Clock();
            Sim_Thread.Number_clocks(1);
            Event_List ls1=new Event_List();
            Event_List ls2=new Event_List();
            ...
            Message m=new Message();
            …
            software T1=new software("T1",1,ls1,lr1);
            software T2=new software("T2",1,ls2,lr2);
            software T3=new software("T3",1,ls3,lr3);
            ...
            m.setMessage(1, "l", "T3");
            ls1.addMessage(m);
            …

            T1.start();
            T2.start();
            T3.start();
      }
}
```

(a) Code

```
T1 starts ...
T2 starts ...
T3 starts ...
Clock( 1 ):- T1 is sending: M( 1, l, T3 )
Clock( 1 ):- T3 is receiving: M( 1, l, T3 )
Clock( 3 ):- T3 is sending: M( 3, d, T2 )
Clock( 5 ):- T2 is sending: M( 5, n, T1 )
Clock( 3 ):- T2 is receiving: M( 3, d, T2 )
Clock( 5 ):- T1 is receiving: M( 5, n, T1 )
Clock( 6 ):- T3 is sending: M( 6, s, T1 )
Clock( 6 ):- T1 is receiving: M( 6, s, T1 )
Clock( 10 ):- T1 is sending: M( 10, m, T2 )
Clock( 10 ):- T2 is receiving: M( 10, m, T2 )
Clock( 16 ):- T2 is sending: M( 16, l, T3 )
Clock( 16 ):- T3 is receiving: M( 16, l, T3 )
Clock( 35 ):- T2 is sending: M( 35, n, T1 )
Clock( 35 ):- T1 is receiving: M( 35, n, T1 )
Clock( 45 ):- T3 is sending: M( 45, k, T2 )
Clock( 45 ):- T2 is receiving: M( 45, k, T2 )
Clock( 69 ):- T1 is sending: M( 69, k, T3 )
Clock( 69 ):- T3 is receiving: M( 69, k, T3 )
T2: End of simulation ...101
T3: End of simulation ...101
T1: End of simulation ...101
```

(b) Exécution

Figure 5.5 : Code (a) et Exécution (b) d'un exemple de la simulation séquentielle

```
import coverification.*;
class Class2 // Distributed Simulation{
public static void main (String[] args)
        {
                Clock Clk1=new Clock();
                Clock Clk2=new Clock();
                Clock Clk3=new Clock();
                Sim_Thread.Number_clocks(3);
                Event_List ls1=new Event_List();
                Event_List lr1=new Event_List();
                ...
                Message m=new Message();
                …
                software T1=new software("T1",1,ls1,lr1);
                software T2=new software("T2",2,ls2,lr2);
                software T3=new software("T3",3,ls3,lr3);
                ...
                m.setMessage(1, "l", "T3");
                ls1.addMessage(m1);
                …
                T1.start();
                T2.start();
                T3.start();
        }
}
```

(a) Code

```
T2 starts ...
T3 starts ...
T1 starts ...
Clock1( 1 ):- T1 is sending: M( 1, l, T3 )
Clock3( 1 ):- T3 is receiving: M( 1, l, T3 )
Clock3( 3 ):- T3 is sending: M( 3, d, T2 )
Clock2( 3 ):- T2 is receiving: M( 3, d, T2 )
Clock2( 5 ):- T2 is sending: M( 5, n, T1 )
Clock3( 6 ):- T3 is sending: M( 6, s, T1 )
Clock1( 5 ):- T1 is receiving: M( 5, n, T1 )
Clock1( 6 ):- T1 is receiving: M( 6, s, T1 )
Clock1( 10 ):- T1 is sending: M( 10, m, T2 )
Clock2( 10 ):- T2 is receiving: M( 10, m, T2 )
Clock2( 16 ):- T2 is sending: M( 16, l, T3 )
Clock3( 16 ):- T3 is receiving: M( 16, l, T3 )
Clock2( 35 ):- T2 is sending: M( 35, n, T1 )
Clock1( 35 ):- T1 is receiving: M( 35, n, T1 )
Clock3( 45 ):- T3 is sending: M( 45, k, T2 )
Clock2( 45 ):- T2 is receiving: M( 45, k, T2 )
Clock1( 69 ):- T1 is sending: M( 69, k, T3 )
Clock3( 69 ):- T3 is receiving: M( 69, k, T3 )
T1: End of simulation ...101
T3: End of simulation ...101
T2: End of simulation ...101
```

(b) Exécution

Figure 5.6 : Code (a) et Exécution (b) d'un exemple de la simulation distribuée

5.7. Conclusion

Nous avons présenté une implémentation basée sur les threads de Java pour une simulation séquentielle et distribuée. Nous avons réécrit leurs algorithmes classiques en affectant les threads aux processus et en bénéficiant des avantages de la programmation orientée objet avec Java. Le déroulement de la simulation est guidé par des échanges d'événements.

Nous avons donc montré dans ce chapitre la faisabilité de ces deux cas de simulation à l'aide des threads de Java. Les résultats obtenus sont prometteurs dans le sens que cette implémentation pourrait servir comme utilitaire pour le processus de la covérification L/M. En particulier, la gestion des horloges des threads et celle de leurs événements de communication en sont d'un grand intérêt.

Dans le chapitre suivant, nous allons discuter la spécification des propriétés en CPL et leur covérification avec JACOV. CPL est un petit langage que nous avons développé pour servir d'entrée à l'outil JACOV et ainsi faciliter à l'utilisateur la description et la saisie de la spécification de son système.

Chapitre 6

JACOV : Spécification et covérification des propriétés

6.1. Introduction

Pour éviter la convergence éventuelle vers un processus itératif de codesign d'un système L/M donné, tel est souvent le cas d'une intégration non réussie des deux parties logicielle et matérielle, il est indispensable qu'une stratégie de covérification soit mise en exécution le plus tôt possible, comme nous l'avons expliqué dans les chapitres précédents. Les techniques de coverification actuellement utilisées dans l'industrie de la microélectronique sont basées sur des environnements de cosimulation où les modules logiciels et matériels sont décrits en HLL et HDL respectivement. Les propriétés du système L/M à covérifier sont incluses dans les codes d'implémentation de manière homogène et ne sont pas exprimées séparément. Ceci rend leur manipulation fastidieuse et coûteuse, surtout pour faire leur revue, leur édition, leur correction, etc. En conséquence, une spécification réalisée

séparément du code d'implémentation est fortement recommandée pour améliorer la qualité et la convivialité de notre « framework ».

Nous présentons dans ce chapitre un outil dédié à la covérification des systèmes L/M. Cet outil est orienté thread; actuellement, il supporte un modèle unifié d'un système L/M, quant au modèle distribué, nous avons projeté de faire son implémentation dans les travaux futurs. La partie logicielle du système L/M et son interaction avec la partie matérielle sont décrites comme étant un ensemble de threads inter-communicants. Les propriétés du système sont écrites en CPL, un petit langage que nous avons conçu pour exprimer, séparément de l'implémentation, les propriétés à covérifier.

Le reste de ce chapitre est organisé de la manière suivante. La section 6.2 met l'accent sur le cadre et la nécessité de cet outil. La section 6.3 présente notre outil de covérification orientée thread. La section 6.4 explique comment exprimer en CPL les propriétés à covérifier ainsi que leur covérification. Avant de conclure, la section 6.5 discute la stratégie de génération des tests à la lumière de la définition des propriétés.

6.2. Motivations

Comme nous l'avons vu au chapitre 4, l'idée originale de notre technique de covérification consiste à mettre en harmonie sur une même assise le processus de cosimulation avec un ensemble de propriétés à tester en concurrence. Cette technique est résumée en quatre étapes (chapitre 4) : *(i) Organisation de la partie logicielle en threads, (ii) Mise en transparence des signaux clés de la partie matérielle, (iii) Spécification des propriétés en CPL* et *(iv) Cosimulation du système global*. Le flux d'exécution de ces étapes est illustré par la figure 6.1.

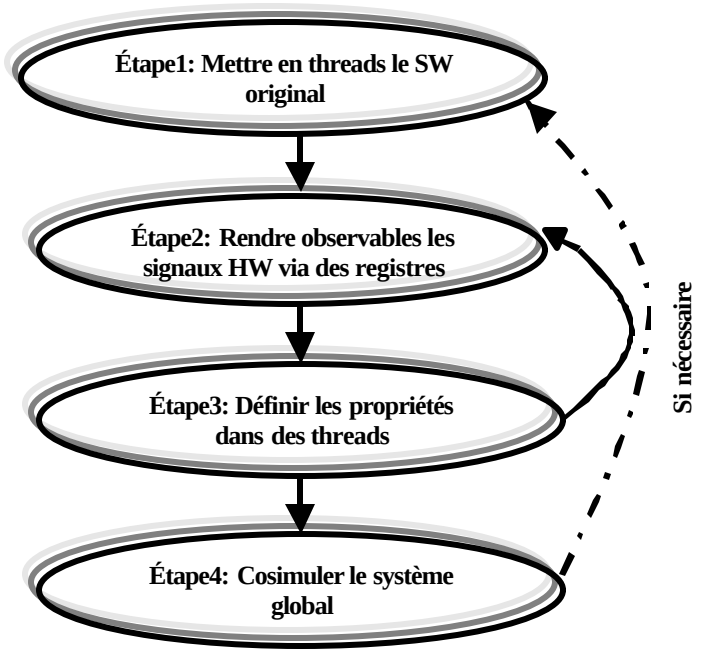


Figure 6.1 : Flux d'exécution de notre technique de covérification

6.3. CPL : Langage de spécification des propriétés à covérifier par JACOV

6.3.1. Définitions

Définition 1

D'un point de vue syntaxique, une propriété en CPL est une composition de conditions sous contrôle. D'un point de vue sémantique, une propriété traduit une ou plusieurs caractéristiques de la spécification du système à covérifier.

Définition 2

Deux propriétés sont dites concurrentes entre elles si elles sont contenues dans deux threads différents. En CPL, nous représentons deux propriétés concurrentes, par exemple p_1 et p_2, par l'instruction : [*concurrent*(p_1, p_2);].

Définition 3

Deux propriétés sont dites séquentielles entre elles si elles appartiennent à un même thread. En CPL, nous représentons deux propriétés séquentielles, par exemple p_1 et p_2, par l'instruction : [*sequential(p_1, p_2);*].

6.3.2. Syntaxe de CPL

La syntaxe des propriétés décrites en CPL est donnée par le format suivant :

Property_name **property** *property_body* **end**

où **property** et **end** sont des mots clés réservés de CPL. *property_name* est un identificateur de la propriété. Cet identificateur doit respecter les règles de définition des identificateurs en HLL. *property_body* est le corps définissant la propriété. Cette définition peut être exprimée en utilisant un des formats listés ci-dessous :

- **(** *condition* **) ;**
- **not (** *condition* **) ;**
- **if (** *condition1* **) (** *condition2* **) ;**
- **if (** *condition1* **) not (** *condition2* **) ;**
- **switch (** *variable_name* **)**

```
{   case val1 : ( condition1 ) ;
    case val2 : not ( condition2 ) ;
                        …
    default : not ( condition ) ;
}
```

if, **not**, **case**, **switch** et **default** sont des mots réservés de CPL. Les commentaires en CPL sont écrits entre (/**) et (**/) ou après (//) jusqu'à la fin de la ligne. CPL permet aussi d'organiser les propriétés déclarées à l'aide des instructions dont la syntaxe est la suivante :

concurrent (*property11_name, property12_name, …* **) ;**

sequential (*property21_name, property22_name, …* **) ;**

6.3.3. Sémantique de CPL

Prototype	Sémantique
Property_name **property (condition); end**	condition est l'expression booléenne de la propriété *property_name*. Cette condition doit être *vraie* durant la simulation.
Property_name **property not (condition); end**	Le même commentaire comme en haut sauf que condition doit être *fausse* durant la simulation.
Property_name **property if (condition1) (condition2) ; end**	La propriété *property_name* décrit le fait que condition2 doit être *vraie* tant que condition1 est *vraie*.
Property_name **property** **switch (** *variable_name* **)** **{case** val1 **: (** condition1 **) ;** **case** val2 **: not (** condition2 **) ;** **…** **default : not (** condition **) ; }** **end**	La propriété *property_name* est contrôlée par une variable clé, *variable_name*. Par exemple, quand cette dernière est à la valeur val1, condition1 doit être *vraie*, et ainsi de suite. Si *variable_name* est différente de toutes les valeurs de la liste des cas (val1, val2, …), condition doit être *fausse*.
Concurrent(*property11_name*, *property12_name*, … **) ;**	Cette instruction dit que les propriétés de la liste (*property11_name*, *property12_name*, …) sont concurrentes et doivent être mises dans des threads distincts.
Sequential(*property21_name*, *property22_name*, … **) ;**	Cette instruction exprime le fait que les propriétés de la liste (*property21_name*, *property22_ name*, …) sont non concurrentes "deux à deux" et peuvent être insérées ensemble dans un seul thread.

6.3.4. À propos de CPL

Nous avons réalisé l'analyseur syntaxique (ou le « parser ») de CPL à l'aide de Javacc (« Java compiler compiler ») version 1.8, un produit de Sun Microsystems. Le fichier source de l'analyseur, cpl_parser.jj, contient plus de 200 lignes de code (annexe 1). En appliquant javacc sur ce dernier (/u/user>javacc cpl_parser.jj), nous générons ainsi l'analyseur de CPL en plus de 1100 lignes de code Java (cpl_parser.java et autres fichiers de constantes, « tokens », exceptions, etc.). Ensuite, ces fichiers Java sont compilés avec javac (« Java compiler ») afin de générer le fichier .class (« byte code ») de notre analyseur. Pour soumettre un fichier, par exemple file_name.cpl, à l'analyseur et voir si la syntaxe de son code est conforme aux règles de CPL, nous faisons appel à l'interpréteur de Java et en voici la commande : /u/user>java cpl_parser file_name.cpl. S'il y a des erreurs dans le code, elles seront détectées, localisées et retournées à l'utilisateur pour les corriger. CPL en est à sa première version, CPL v1.0.

6.4. Description et covérification des propriétés L/M avec JACOV

6.4.1. Spécification des propriétés en CPL

La spécification d'un système L/M est traduite en un ensemble de propriétés décrites en CPL. CPL est un langage simple que nous avons développé spécifiquement pour exprimer ce type de propriétés auxquelles nous avons affaire. Il se prête très bien à la description des propriétés aussi bien au niveau comportemental qu'au niveau RTL. Les paramètres de chacune des propriétés peuvent appartenir aux deux parties logicielle et matérielle ou juste à l'une d'elles. En CPL, le programmeur doit spécifier lesquelles des propriétés sont exécutées en concurrence et celles en série; toutefois, les propriétés non affectées à un mode d'exécution sont par défaut exécutées séquentiellement. Par exemple, dans la figure 6.2, p_1 et p_2 sont concurrentes entre elles, tandis que p_2 et p_3 ne le sont pas.

```
p1 property ( flow_input - flow_output * sqr(2) <= 0.5 ) end

p2 property if ( temperature == 199) ( current <= 0.2 ) end

p3 property if ( signal_1 == 199) not( signal_2 == 134 ) end

p4 property switch ( state )
{
case S1 : not(signal_ack & signal_start);
case S2 : (red_light & not(green_light & orange_light));
default : (reg_R = 1);
}
end
concurrent ( p1, p2);
sequential (p2, p3);
```

Figure 6.2 : Exemple de propriétés décrites en CPL

Cependant, le champ d'application de CPL n'est pas restreint aux systèmes L/M et à leur covérification mais il peut englober d'autres applications similaires. CPL peut être enrichi et étendu pour répondre avec satisfaction à toutes les exigences des utilisateurs. En outre, CPL pourrait être traduit non seulement dans un code Java mais aussi dans d'autres langages possédant des objets de concurrence, tels que C++ ou Ada. D'ici, nous pourrons songer à l'élaboration d'un langage universel de description de propriétés pour toute forme de simulation.

6.4.2. Covérification des propriétés avec JACOV

Contenu dans un fichier d'extension (**.cpl**), le code CPL décrivant les propriétés du système est traduit en un sous-ensemble de code Java orienté thread. Les propriétés déclarées comme *concurrent* sont manipulées chacune par un thread et celles identifiées comme *sequential* sont ramassées dans un thread. Par défaut, celles qui n'ont pas été mentionnées ni comme *concurrent* ni comme *sequential*, elles seront traitées ensemble dans un même thread. Tous les threads porteurs de propriétés sont ajoutés à l'implémentation du système pour ensuite mener une

simulation fonctionnelle du code global. Le synoptique de l'outil de covérification, nommé JACOV, est illustré par la figure 6.3. La fonction essentielle du bloc « Intégrateur » sur cette figure est d'insérer les propriétés rangées dans des threads (properties.java) au sein du code du logiciel (implementation.java) pour générer ensuite un fichier global à soumettre au moteur de covérification.

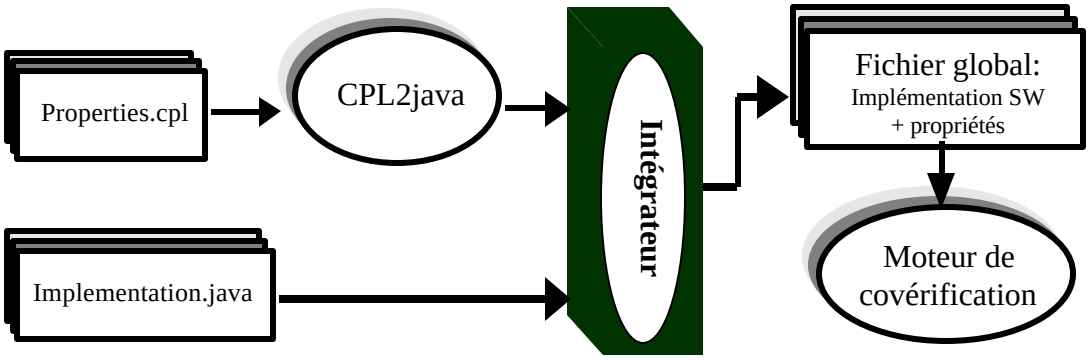


Figure 6.3 : Vue globale de l'outil semi-automatique de covérification JACOV

6.4.3. À propos de JACOV

JACOV est un outil dédié à la covérification des propriétés, que nous avons conçu et implémenté en Java, il en est à sa version primaire, JACOV v1.0. Cette dernière est semi-automatique et dépourvue d'interface graphique, mais JACOV pourrait paraître dans des versions subséquentes avec des caractéristiques aussi bien attirantes que qualitatives. Nous étions intéressés en élaborant JACOV v1.0 de montrer sa faisabilité par compléter les tâches qui composent l'outil sans trop se préoccuper de l'automatisation et de l'interface utilisateur. Cependant, cet aspect pourrait être abordé (dans de futures versions) dans l'optique de rendre l'outil plus professionnel et plus compétitif.

JACOV v1.0 accepte à son entrée deux fichiers principaux : le fichier d'implémentation primaire (implementation.java) et celui des propriétés (properties.cpl) et il génère à sa sortie deux fichiers : le fichier d'implémentation secondaire (impl_prop.java) et celui des résultats (figures 6.3 et 6.4). Le noyau de JACOV v1.0 est développé autour de la méthodologie de covérification à base de « multithreading » (chapitre 4).

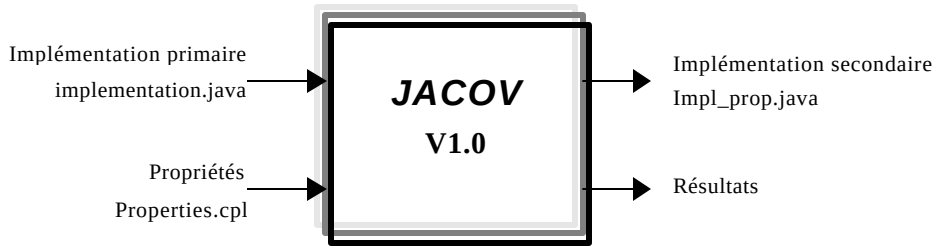


Figure 6.4 : Entrées-Sorties de JACOV

6.4.4. Activation des propriétés

Après la traduction du CPL en Java par *CPL2Java*[9], les propriétés apparaissent comme des fragments de code Java. L'utilisateur peut directement les exprimer sous réserve de respecter la syntaxe de Java et certaines règles particulières ou, comme nous l'avons expliqué plus haut, il peut les décrire séparément à l'aide de CPL. En appliquant un vecteur de test à l'entrée du système, ces propriétés peuvent être activées ou non. Nous poursuivons l'activation de chacune des propriétés durant la simulation à l'aide de compteurs spécifiques. La fonction d'évaluation est basée essentiellement sur les valeurs retournées par ces compteurs. L'exemple ci-dessous montre le corps d'une propriété dont le nom est Property_name en CPL et son équivalent en Java. L'activation de Property_name est contrôlée par les quatre compteurs : precounter_T, precounter_F, postcounter_T et postcounter_F.

[9] CPL2Java n'est pas encore automatisé.

CPL :

Property_name **property if (** *Precondition()* **)** *Postcondition()* **end**

Java :

```
Id = Property_name ;
If      ( Precondition() )
        {
                precounter_T = precounter_T + 1 ;
                If ( Postcondition() ) postcounter_T = postcounter_T + 1 ;
                Else postcounter_F = postcounter_F + 1 ;
        }
Else    precounter_F = precounter_F + 1;
```

Légende :

Precondition()	:	est la condition de la propriété *Property_name.*
Postcondition()	:	est l'expression de la propriété *Property_name* qui doit être satisfaite si la *Precondition()* est vraie.
Precounter_T	:	compte combien de fois *Precondition()* est vraie.
Precounter_F	:	compte combien de fois *Precondition()* est fausse.
Postcounter_T	:	compte combien de fois *Postcondition()* est vraie.
Postcounter_F	:	compte combien de fois *Postcondition()* est fausse.

Les séquences des vecteurs de test sont aléatoires (figure 6.5) ou construites (figure 6.6). Dans le premier cas, ces séquences sont générées à l'aide d'un processus

de génération aléatoire basé sur la méthode prédéfinie *Random()*. Quant au second cas, un module est chargé de la génération d'une séquence de vecteurs de test selon certains algorithmes ou fonctions de transformation, tout en tenant compte de l'évolution de l'activation des propriétés. Suite à l'application de ces algorithmes sur une séquence de vecteurs de test initiale, une nouvelle séquence fille est construite dans l'espoir d'obtenir quelques vecteurs de test. La fonction d'évaluation calcule la couverture des propriétés. Elle retourne pour chaque vecteur de test TV_i le nombre d'activations PAN_i de toutes les propriétés P_j à la fin du processus de simulation :

$$\mathrm{PAN}_i = \sum_{j=1}^{N} \mathrm{PAN}(\mathrm{P}_j), \text{ N est le nombre de propriétés considérées}$$

La fonction $PAN(P_j)$ donne le nombre d'activations de la propriété P_j. Elle est exprimée en fonction des quatre compteurs : *precounter_T*, *precounter_F*, *postcounter_T* et *postcounter_F* selon les objectifs de l'opérateur. Les vecteurs de test de la séquence appliquée à la fin de la simulation sont classés selon l'ordre croissant de leurs nombres d'activations. Un vecteur de test est jugé comme étant un bon vecteur de test (parmi tous les vecteurs de la séquence) si son nombre d'activations est plus grand en comparaison avec les nombres d'activations des autres vecteurs de test.

Dans le cas de la génération aléatoire des vecteurs de test (figure 6.5), les vecteurs de test qui sont sélectionnés peuvent servir comme vecteurs d'amorçage pour le cas du test construit (figure 6.6). Dans ce dernier cas, le vecteur de test subit des transformations au niveau de ses composants dans l'espoir d'améliorer son nombre d'activations. Ces transformations peuvent être aléatoires, déterministes ou mixtes. Cependant, il n'y a aucune garantie de trouver ou de construire de bons vecteurs de test. Ce sont juste des tentatives de vouloir mieux comprendre le comportement d'un système donné vis-à-vis à l'application de ces vecteurs de test à son entrée. Pour un système simple, les conclusions du test sont évidentes, cependant la tâche est souvent très fastidieuse voire impossible pour un système complexe.

Nous pensons que ces propositions peuvent être perfectionnées davantage dans un travail futur en menant une compagne de test rigoureuse sur un design réel.

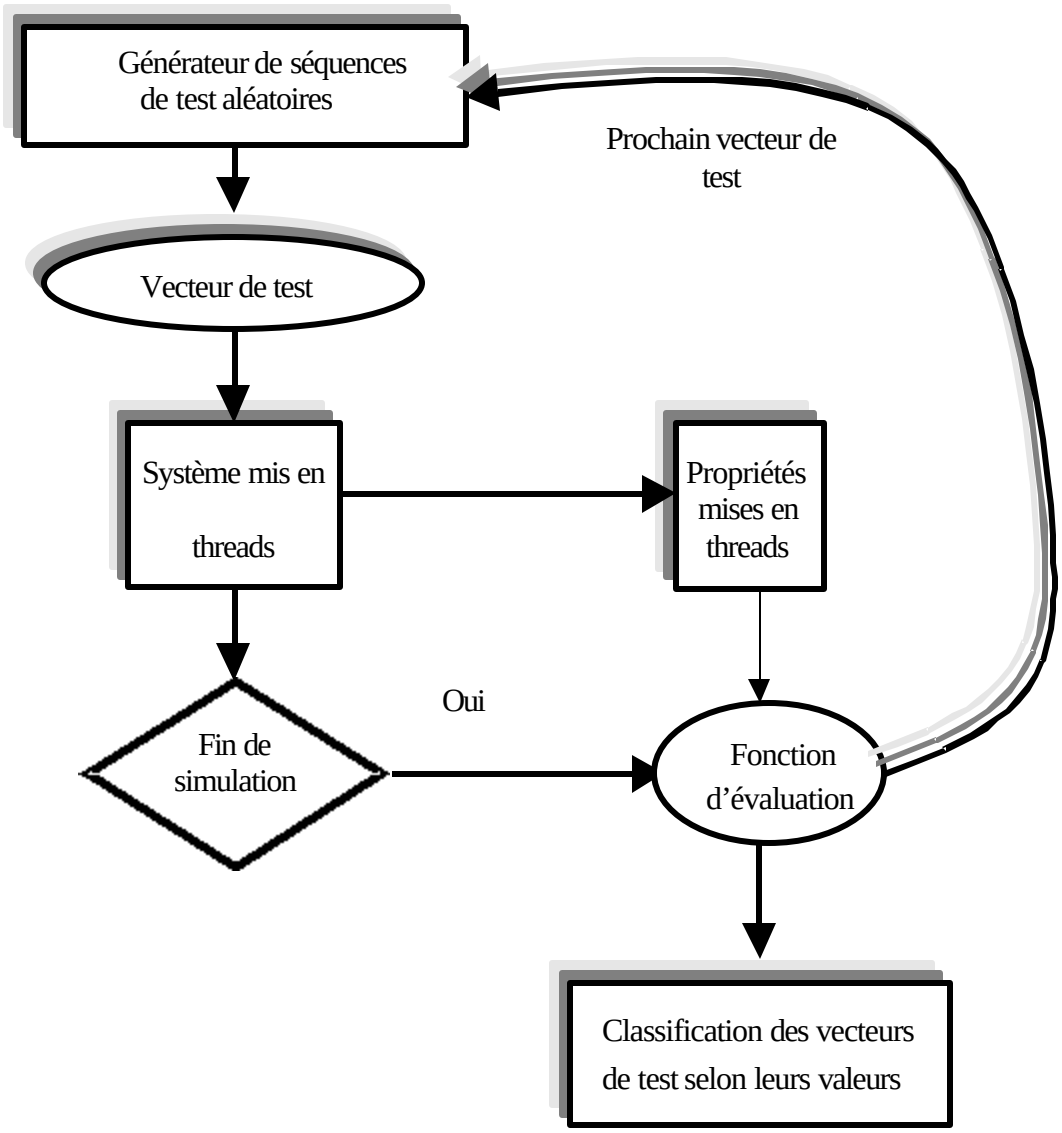


Figure 6.5 : Cas de test aléatoire des propriétés

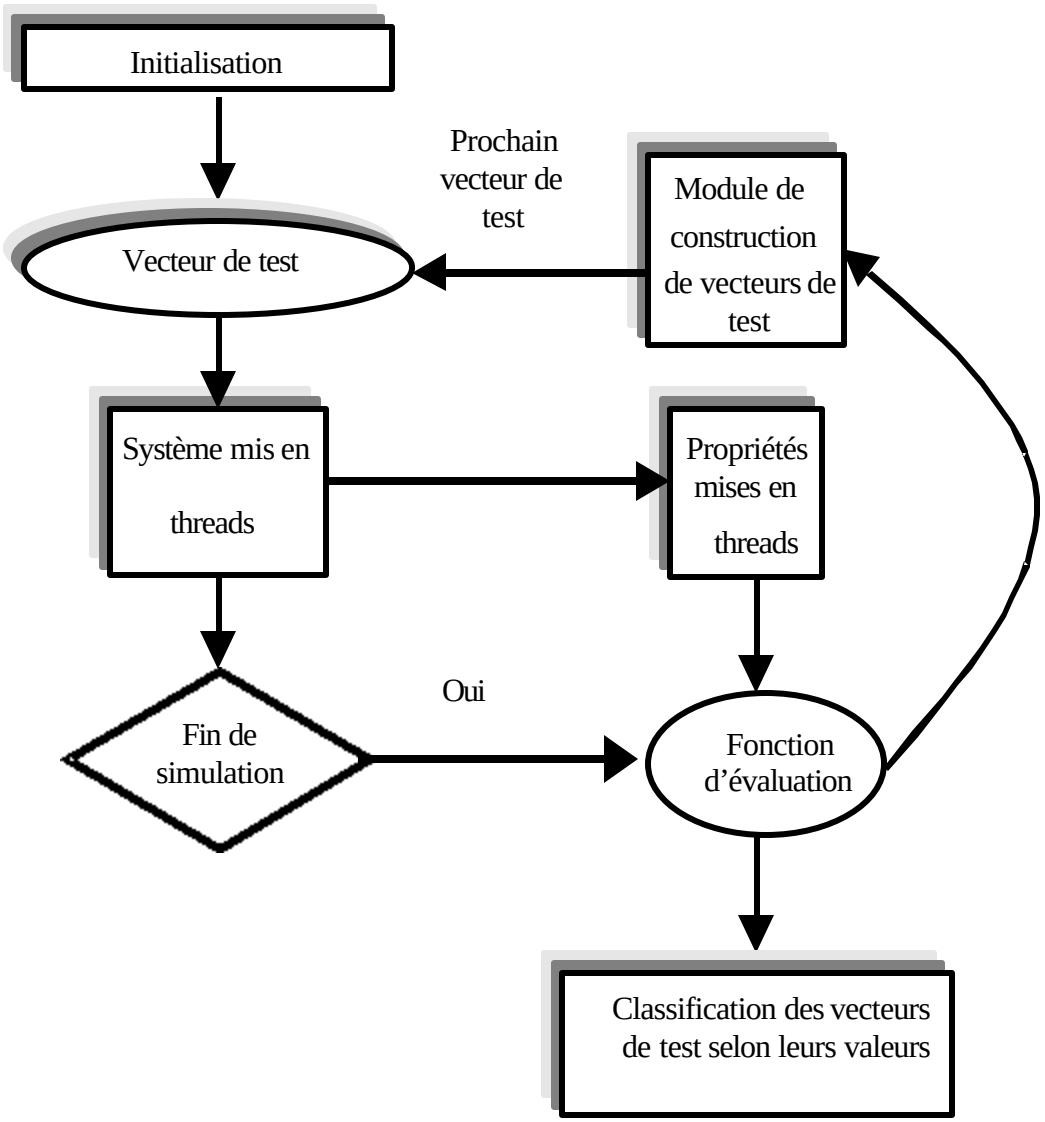


Figure 6.6 : Cas de test guidé par des propriétés

6.5. Conclusion

Dans ce chapitre, nous avons présenté notre outil de covérification des systèmes L/M, JACOV, implémentant la méthodologie discutée au sein du chapitre 4. Cet outil est basé sur la technologie de cosimulation et le concept du « multithreading ». Un petit langage, baptisé CPL, a été spécialement développé pour décrire les propriétés à covérifier. L'utilisation de CPL est très facile et proche de celle d'un langage de programmation tel que Java. JACOV et CPL en sont à leurs premières versions (v1.0) et ils se prêtent très favorablement à une évolution ultérieure. Nous discuterons dans le dernier chapitre les extensions futures de ces deux produits. Dans le chapitre suivant, nous allons démontrer le rôle complémentaire de la covérification, selon notre méthodologie, à la vérification d'ordre général. Une telle démonstration est menée sur un exemple de protocole de cohérence des caches.

Chapitre 7

Vérification et Covérification d'un protocole de cohérence des caches

7.1. Introduction

Les approches de vérification traditionnelles telles que la simulation et le test ne sont pas les seuls moyens existants pour valider le design d'un système donné. Leurs inconvénients se résument dans le fait que lorsqu'il s'agit d'un système complexe, la considération de tous les vecteurs de test possibles est une tâche très laborieuse et fastidieuse, voire impossible, en termes de temps d'exécution et de coût de réalisation. Leur application pourrait avoir de l'importance dans le cas où un nombre limité de vecteurs de test serait considéré afin de vérifier un ou plusieurs comportements bien définis du système. Cependant, lorsque nous voulons démontrer la conformité d'un système complexe pour tous les cas d'entrée, ces approches sont pratiquement inaptes à réaliser une telle tâche. Les méthodes de vérification formelle ont attiré au cours de la décennie passée l'attention de beaucoup de chercheurs [82, 105-108], ceci est dû grâce à leurs preuves mathématiques de la validité du design d'un système sans faire appel à la simulation. Cependant, à cause de la réticence des industriels à son égard, de sa limitation dans le cas des systèmes complexes, de son

abstraction assez élevée, et quoique soient très abondants les articles la traitant, l'utilisation de la vérification formelle reste très restreinte et est souvent prise dans un sens de consultation. Des propositions de combiner des techniques de simulation avec la vérification formelle ont été récemment faites par des spécialistes de cette dernière dont David Dill [109], pour élaborer une vérification dite semi-formelle.

Nous considérons dans ce travail un système composé de plusieurs processeurs auquel nous appliquons la technique de « model checking » pour prouver la conformité du design du système par rapport à sa spécification en terme de la cohérence des caches. Bien que la vérification du bon fonctionnement des caches d'un système multiprocesseur ait fait l'objet de plusieurs travaux [5, 51, 105, 111-116, 118, 119, 122-128], elle faisait souvent cas de la vérification du flux de contrôle et omettait le flux de données.

Dans ce chapitre, nous effectuons la vérification du protocole de la cohérence des caches sous plusieurs facettes afin de montrer les difficultés rencontrées et comment y remédier. Pour la vérification formelle, nous utilisons l'outil du domaine publique VIS [98, 110]. Nous illustrons les effets respectifs de la longueur des données et du nombre de processeurs sur l'analyse d'accessibilité. Le système considéré est un ensemble de processeurs à mémoire partagée où ces processeurs communiquent entre eux via un bus partagé et ils devront maintenir la cohérence de leurs caches selon le protocole « snoopy » [117]. L'implémentation est décrite en un sous-ensemble de Verilog et les propriétés sont exprimées en CTL («Computation Tree Logic») [98, 110]. Quant à la covérification, nous avons partiellement décrit en Java le même système avec un protocole de cohérence des caches, et nous avons appliqué quelques vecteurs de test pour tester la conformance du protocole. Nous avons considéré les cas «sans et avec protocole» pour des raisons de comparaison. La covérification est introduite dans ce cas d'étude pour contrôler le flux des données du système. Son apport vient compléter celui de la vérification formelle afin de mieux comprendre et analyser le système étudié.

7.2. Architecture du système

L'architecture du système (figure 7.1) comprend trois ou plusieurs processeurs symétriques munis chacun d'une cache et partageant tous la mémoire principale et le bus. Ce dernier est contrôlé par un arbitre. Le processeur voulant utiliser le bus envoie une requête à l'arbitre pour en obtenir la permission. Si le bus est libre, elle lui sera accordée, autrement l'arbitre range la requête dans une file d'attente. Le processeur demandeur reçoit une réponse concernant le traitement de sa requête. Chaque processeur peut sans crainte traiter les données contenues dans sa cache si ces dernières (copies) sont conformes à leurs originales qui se trouvent dans la mémoire partagée. Si cette condition est invalide, alors les données de la cache doivent être mises hors usage et une mise à jour est obligatoire. Seul un processeur maître peut lire/écrire des données de/dans la mémoire, les autres processeurs esclaves qui demandent un accès mémoire à ce moment rentrent dans un état d'attente.

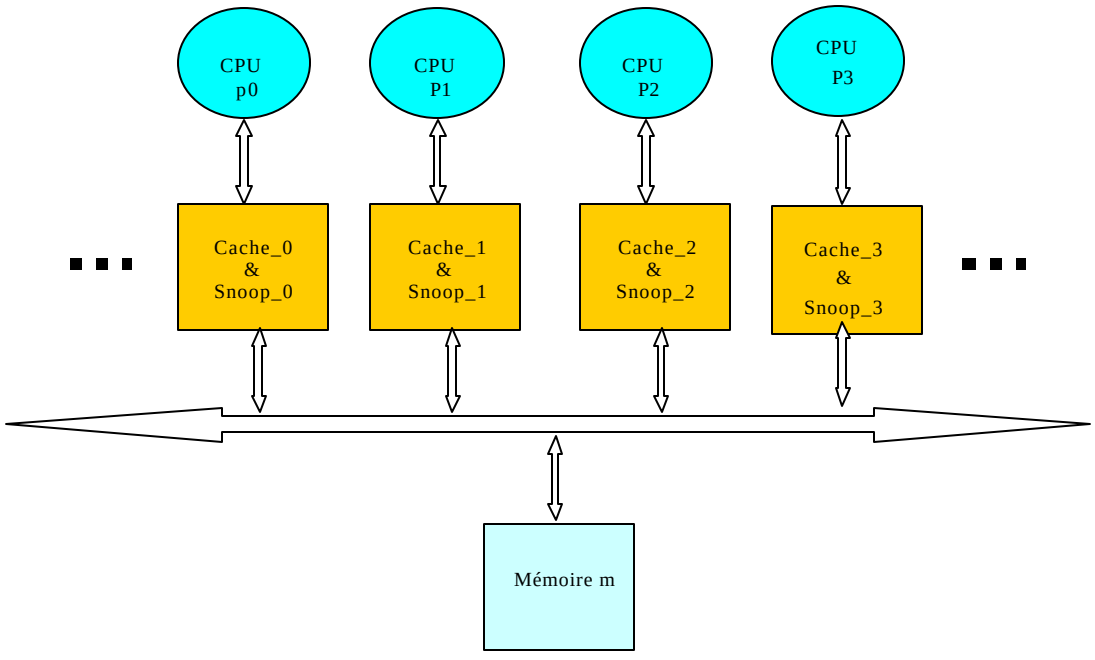


Figure 7.1 : Architecture du système étudié

Dans un premier temps, nous considérons seulement trois processeurs dans le système global. Ensuite, nous augmentons progressivement le nombre des processeurs pour manifester l'effet de ce dernier sur les paramètres de l'analyse d'accessibilité tels que la taille des BDD (« Binary Decision Diagrams ») et le temps d'analyse. Nous appliquons la même stratégie pour montrer l'effet de la taille des données (nombre de bits) des caches sur l'analyse d'accessibilité.

7.3. Modélisation du système

Sans perte de généralité, toutes les machines à états finis sont établies juste pour un seul bloc de mémoire. En outre, elles sont toutes non déterministes. Les transitions sont désignées par les symboles T_i (i=0, 1, 2, …). Tous les états sont déclarés en HDL comme des registres; quant aux transitions, elles sont représentées par des conditions entraînant l'évolution du système vers des états spécifiques si elles sont validées. Des transitions sont multi-évaluées à cause du non-déterminisme appliqué à certaines variables. Chacune d'elles représente en fait une classe de transitions qui possèdent toutes la même formule mais de différentes valeurs. Dans ce qui suit, nous présentons les machines FSM (« Finite State Machines ») de la mémoire partagée, du bus partagé, des caches privées, des processeurs et du système global.

7.3.1. La mémoire partagée

La mémoire partagée fonctionne globalement en deux états : "*busy=0*" et "*busy=1*" (figures 7.2 et 7.3). Dans le premier état, la mémoire est libre et disponible à toute requête d'accès; dans le second état, elle est occupée à servir le processeur maître et toute autre requête reçue en ce moment est mise en attente. Par exemple, la transition T_6 bascule l'état de la mémoire partagée de "busy=1" à busy=0" lorsque celle-ci a fini de servir un accès de lecture.

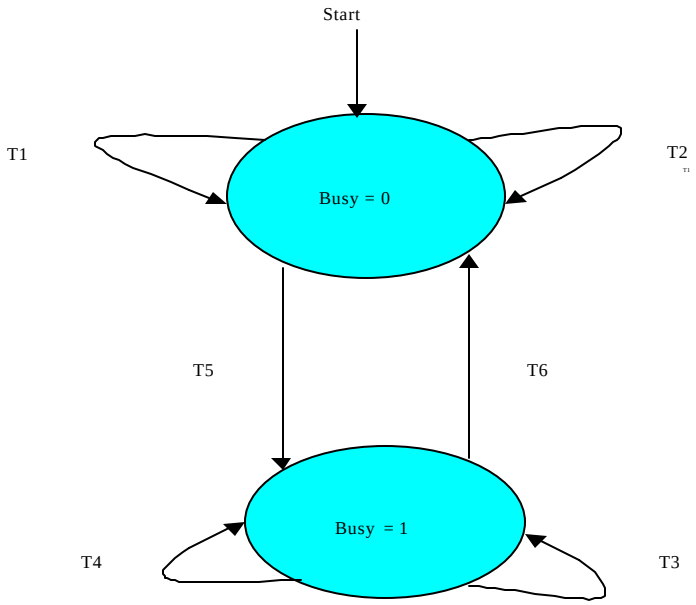


Figure 7.2 : FSM de la mémoire partagée

```
module main_shared_memory(clk, COMD, master, R_STALLED,
R_OWNED, REPLY_WAITING, comd, r_owned, r_waiting,
r_stalled, datam[0:7]);
...
reg busy; //memory state
...
initial busy = 0;
always @(posedge clk)
begin
if (abort) begin busy = busy; end
else if (master && COMD == read_response) begin busy = 0;
end
else if (!master && COMD == read_owned || CMD==read_shared)
begin busy=1; end
end
...
endmodule
```

Figure 7.3 : Module de la mémoire partagée en Verilog

7.3.2. Bus partagé

Le bus, tout comme la mémoire partagée, possède deux états (figures 7.4 et 7.5). Lorsque celui-ci est manipulé par un processeur maître, il se trouve dans l'état « waiting = 1», autrement il est à l'état libre «waiting = 0». Par exemple, la transition T_6 fait passer l'état du bus de "waiting=0" à "waiting=1" lorsque le processeur maître fait une requête d'accès de lecture à une donnée non partagée; dans la même circonstance, la transition T_{10} maintient l'état du bus "waiting=1".

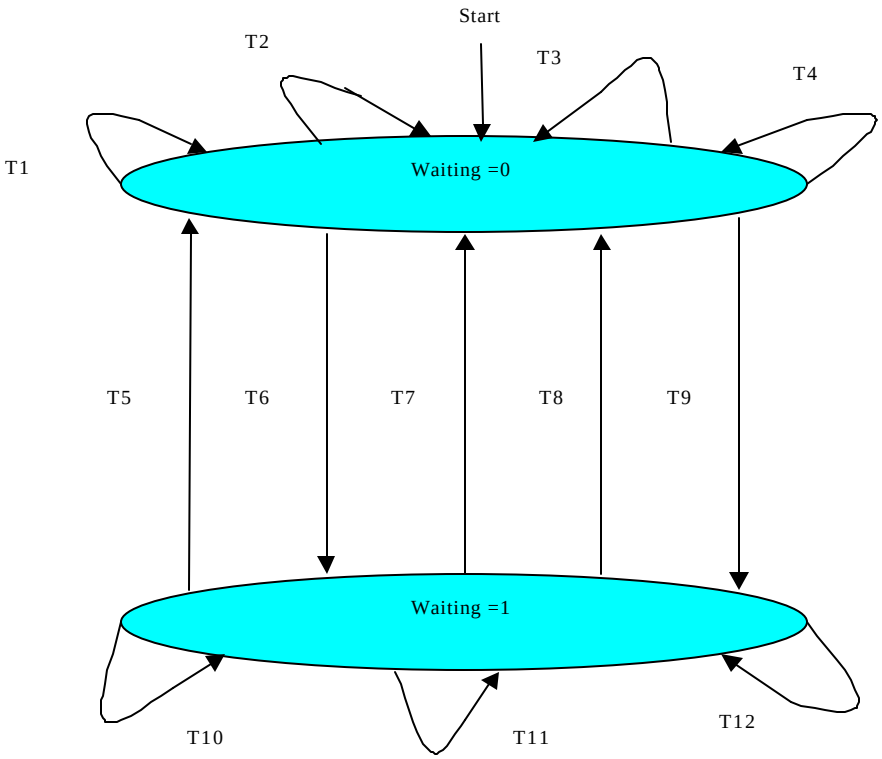


Figure 7.4 : FSM du bus

```
module shared_bus_device(clk, COMD, master, R_STALLED,
R_WAITING, waiting, r_waiting, abort);
...
reg   waiting; //Bus state
...
initial waiting = 0;
always @(posedge clk)
begin
if (abort) begin waiting = waiting; end
else if (master && COMD==read_shared) begin  waiting = 1;
end
else if (master && COMD==read_owned) begin  waiting = 1;
end
else if (!master && COMD==response) begin  waiting = 0; end
...
end

endmodule
```

Figure 7.5 : Module du bus en Verilog

7.3.3. Mémoire cache

Deux FSMs décrivent le contrôle de la mémoire cache; elles retournent les états respectivement de la cache : “state” (figures 7.6 et 7.8), et du module d’écoute : “snoop” (figures 7.7 et 7.8). Le rôle principal de cette dernière est de maintenir la cohérence des caches. Ce module d’écoute met à la disposition des autres modules rivaux l’état actuel du bloc de la cache (« invalid », « shared » ou « owned »). L’état d’un bloc de la cache se trouvant à “state=owned”, passe à “state=shared” au moment où deux ou plusieurs processeurs viennent le manipuler simultanément (transitions $T_{20} + T_{19} + T_{18}$).

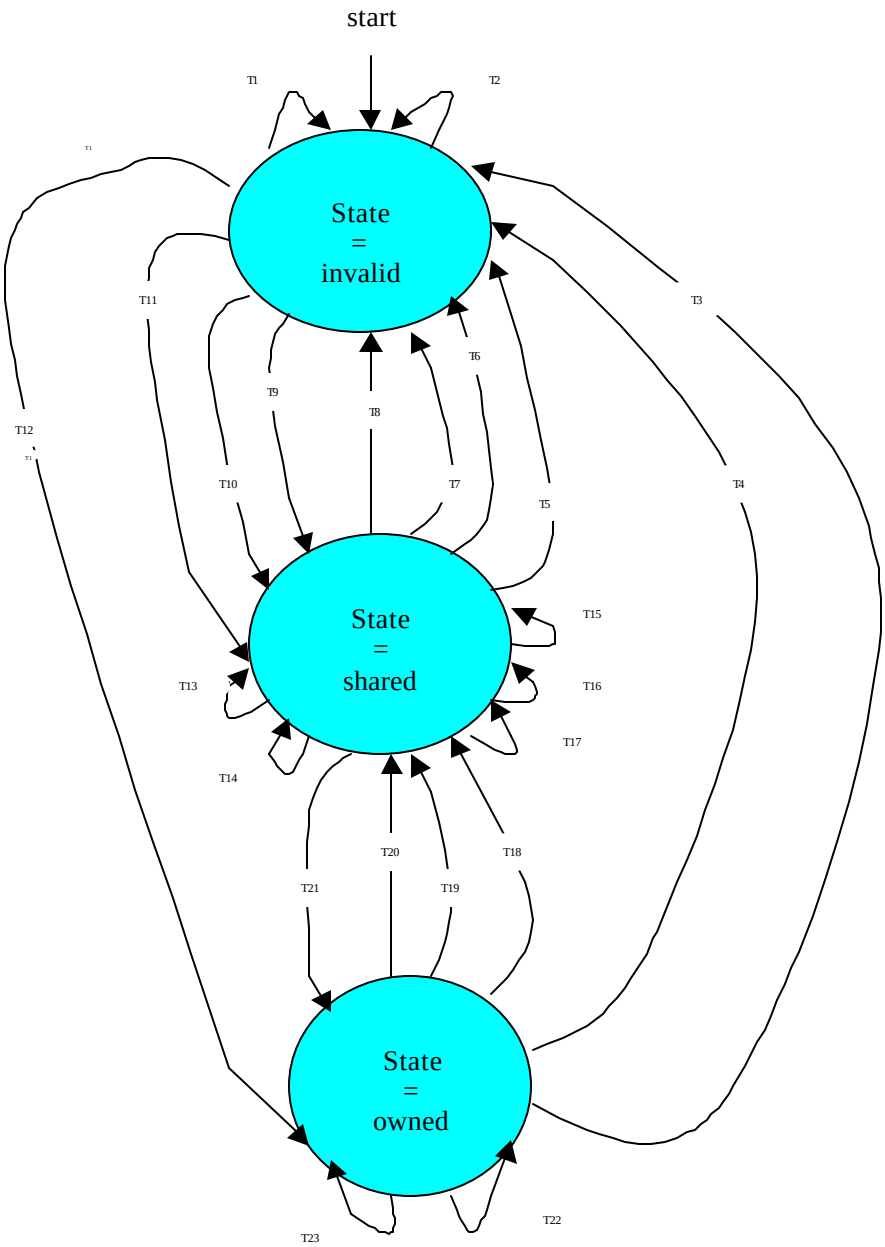


Figure 7.6 : FSM d'un bloc de la cache

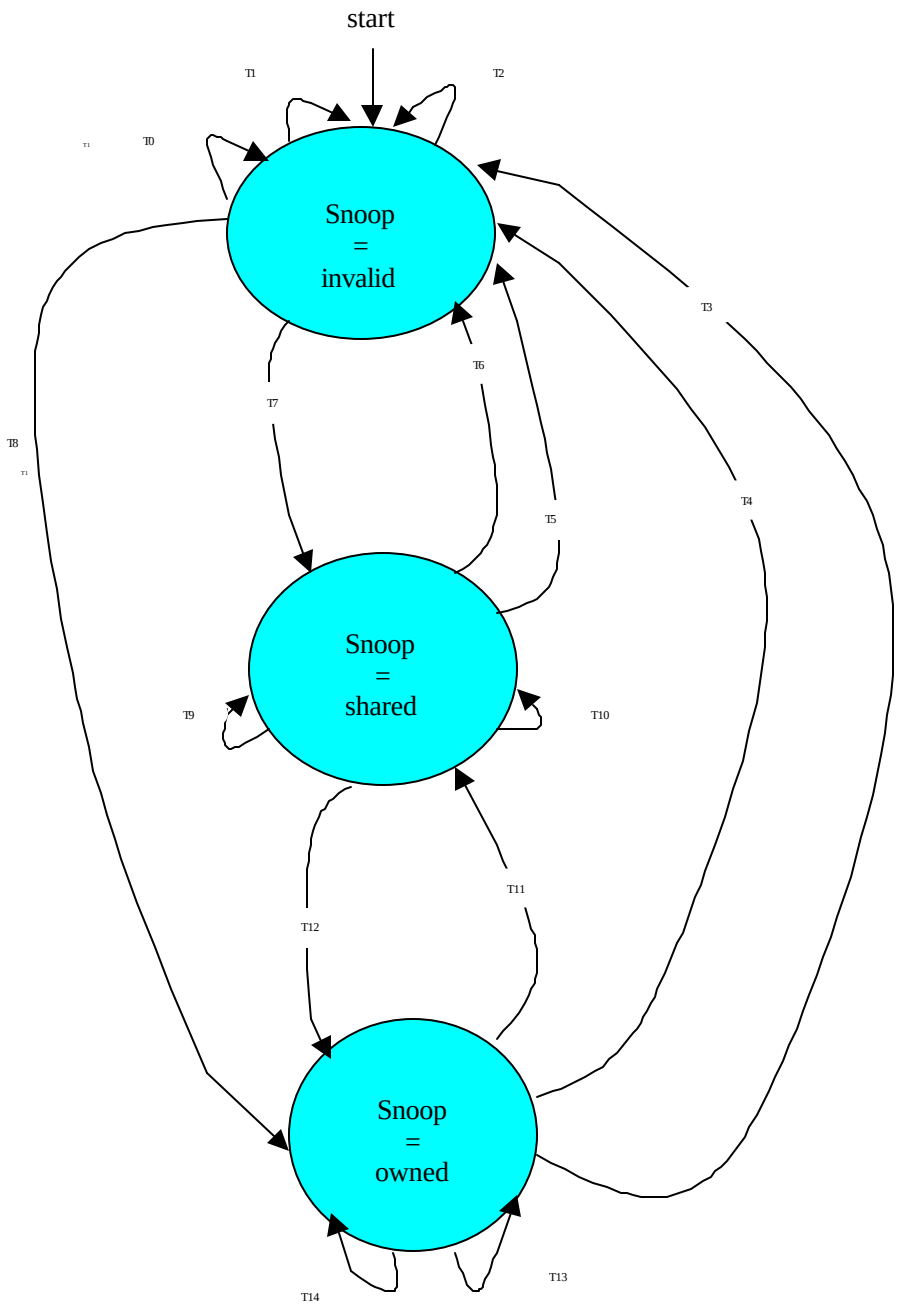


Figure 7.7 : FSM du protocole « snoopy »

```
module cache_memory (mem, datam[0:7], clk, COMD, master,
abort, waiting, state, snoop, r_owned, readable,
writable,  data);
...
reg[0:2]    state, snoop; //cache and snnopy states
...
initial begin state = invalid; snoop = invalid; end
always @(posedge clk)
begin
if(abort) begin snoop = snoop; end
else if (!master && state==owned && COMD==read_shared)
begin snoop = non_determ_snoop; end
...
if (abort) begin state = state; end
else if (master ==1)
begin
      case(COMD)
      read_shared : state = shared;
      read_owned: state = owned;
      write_invalid: state = invalid;
      ...
      endcase
end
else  if  (!master  &&  state  ==  shared  &&  (COMD  ==
read_owned || COMD ==invalidate)) begin state = invalid;
end
else if (state==shared) begin state=non_determ_state; end
...
end
...
endmodule
```

Figure 7.8 : Module de la cache en Verilog

7.3.4. Processeur

L'état de chaque processeur dépend des états de sa cache et du bus (figure 7.9). Ainsi, sa machine FSM est construite en combinant les machines FSM de la cache et du bus; il faut éventuellement compléter la machine FSM globale obtenue par des entrées/sorties qui ne sont pas nécessairement consommées par les machines de la cache et du bus. Nous avons alors :

$$\text{FSM (processeur } p_i) = (\text{ FSM(cache } c_i), \text{FSM(bus)})$$

$$\text{pour tout } i \in \{0, 1, 2, ...\}$$

```
module processor(mem, datam[0:7], clk, COMD, master,
R_WAITING, R_STALLED, comd, r_owned, r_waiting, r_stalled,
data1);
...
typemem wire mem;
...
shared_bus_device Bus(clk, COMD, master, R_STALLED,
R_WAITING, waiting, r_waiting, abort);
cache_memory Cache(mem,datam[0:7], clk, COMD, master,
abort, waiting, state, snoop, r_owned, readable, writable,
data1);
...
endmodule
```

Figure 7.9 : Module du processeur en Verilog

7.3.5. FSM du système global

Le système global, comme nous l'avons défini dans le module principal (figure 7.10 et 7.12), regroupe toutes les instantiations des modules des processeurs (p_0, p_1, …) et de la mémoire partagée (m). Sa machine FSM est donc une composition de leurs machines FSM (figure 7.11) :

$$FSM = (FSM(m), FSM (p_0), FSM (p_1), \ldots)$$

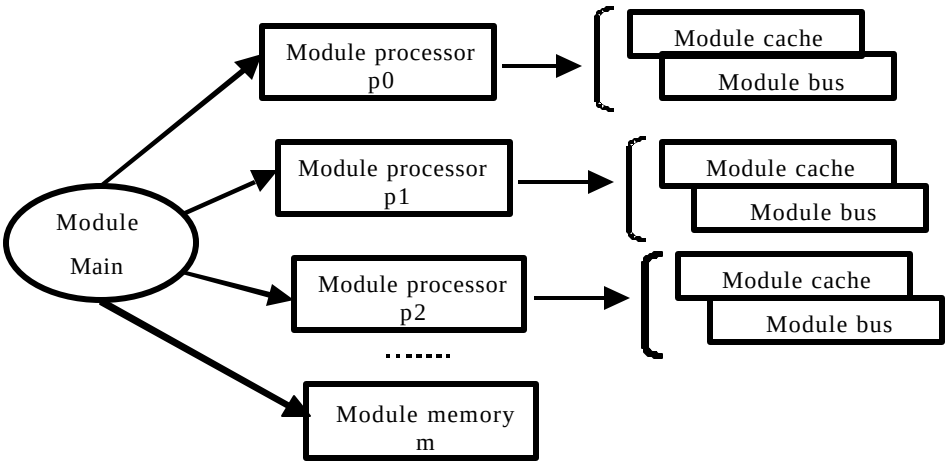


Figure 7.10 : Vue globale du code du système

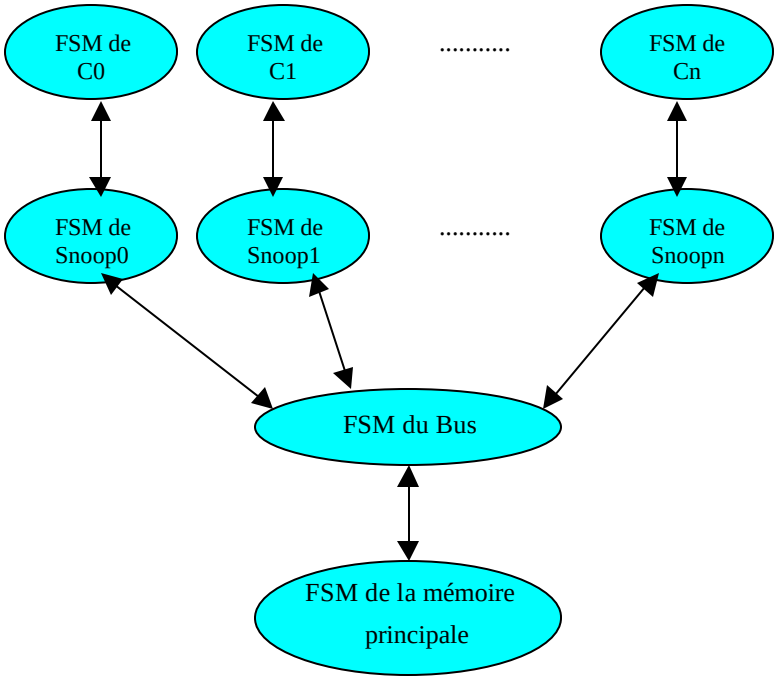


Figure 7.11 : FSM du système global

```
module main(clk, Time_max);
...
processor   p0(c0,datam[0:7],   clk,   COMD,   p0_master,
R_WAITING, R_STALLED, p0_comd, p0_r_owned, p0_r_waiting,
p0_reply_stalled, data0[0:7]);
processor   p1(c1,   datam[0:7],   clk,   COMD,   p1_master,
R_WAITING, R_STALLED, p1_comd, p1_r_owned, p1_r_waiting,
p1_reply_stalled, data1[0:7]);
processor   p2(c2,datam[0:7],   clk,   COMD,   p2_master,
R_WAITING, R_STALLED, p2_comd, p2_r_owned, p2_r_waiting,
p2_reply_stalled, data2[0:7]);
processor   p3(c3,datam[0:7],   clk,   COMD,   p3_master,
R_WAITING, R_STALLED, p3_comd, p3_r_owned, p3_r_waiting,
p3_r_stalled, data3[0:7]);
processor   p4(c4,datam[0:7],   clk,   COMD,   p4_master,
R_WAITING, R_STALLED, p4_comd, p4_r_owned, p4_r_waiting,
p4_reply_stalled, data4[0:7]);
main_shared_memory       m(clk, COMD, m_master, R_OWNED,
R_WAITING,  R_STALLED,  m_comd,  m_r_owned,  m_r_waiting,
m_r_stalled, datam[0:7]);
...
endmodule
```

Figure 7.12 : Module principal en Verilog

7.4. Vérification des propriétés par «model checking»

Dans ce qui suit, nous donnons une liste non exhaustive des différentes propriétés que nous avons examinées au cours de ce travail. La technique de vérification que nous avons employée est « VIS- model checking ».

7.4.1. Protocole de cohérence des caches

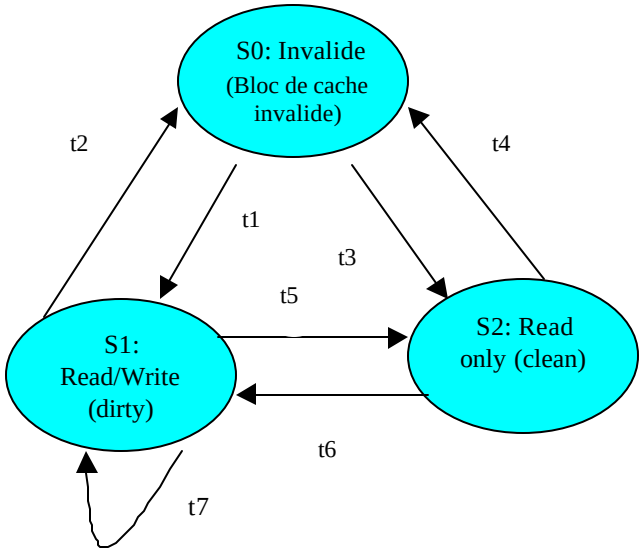
Le principe de ce protocole [117] est illustré par le diagramme de la figure 7.13. Toute cache, autre que celle du processeur maître qui entreprend un accès mémoire d'écriture, doit être mise à jour après l'achèvement de ce dernier si elle partage des blocs avec la cache du maître. En d'autres mots, dès qu'un processeur a effectué un accès d'écriture à la mémoire partagée, un signal d'avertissement est adressé aux autres processeurs afin de faire une mise à jour de leurs caches avant tout usage. Des propriétés de ce protocole { P_1, P_2, P_3, P_4 } sont exprimées en CTL dans le tableau 7.1. P_1 spécifie le fait qu'après tout accès d'écriture, les caches impliquées (partageant des données réécrites) doivent être mises à jour; P_2 traduit le fait que lors du chargement d'une cache, les autres caches puissent être lues sans aucune crainte de problèmes d'inconsistance de données; P_3 signifie que si deux caches contiennent des blocs copiés des mêmes adresses de la mémoire partagée, leurs données doivent être identiques; P_4 dit que si une cache est lisible (c'est-à-dire ses données sont valides selon le protocole), alors ses données doivent être similaires à leurs originales dans la mémoire partagée.

P_1 : $\forall$ *i, j* $\in$ *{0, 1, 2, …}, if { processor* p_i *executes a write command of a shared block }, then { all caches* c_j $(j \neq i)$ *sharing that block, will be updated}.*

P_2 : $\forall$ *i, j* $\in$ *{0, 1, 2, …}, if { the cache* c_i *is writable }, then {any other cache* $c_j (j \neq i)$ *can be readable}.*

P_3 : $\forall$ *i, j* $\in$ *{0, 1, 2, …}, if {*c_i *and* c_j *are readable (*c_i *and* c_j *are the caches respectively of processors* p_i *and* p_j*)}, then {the data of the shared block in* c_i *and* c_j *have the same values}.*

P_4 : $\forall$ *i* $\in$ *{0, 1, 2, …}, if { *c_i *is readable and shared memory has not been modified}, then {the data in* c_i *must be the same in the shared memory}.*



États: S0 (Les données du bloc de la cache sont non valides), S1 (le bloc de la cache peut être lu par le processeur local et partagé avec les autres), S3 (le bloc de la cache est "dirty" et non partagé).
Transitions: t1 ("processor write miss"), t2 ("read or write miss" d'un autre processeur pour ce bloc + "write back" l'ancien bloc), t3 ("processor read miss"), t4 ("invalidate or write miss" d'un autre processeur pour ce bloc), t5 ("processor read miss"), t6 ("processor write"), t7 ("processor write")

Figure 7.13 : Protocole de cohérence des caches

7.4.2. Bus

Le bus ne peut être utilisé simultanément par plus d'un processeur maître. Les autres demandeurs de bus sont mis en file d'attente jusqu'à l'arrivée de leur tour. La propriété décrivant ce fait, est donnée ci-dessous :

P_5 : *if { the bus is owned },*

then { (p_0 master) ⊕ (p_1 master) ⊕ (p_2 master) ⊕… ⊕ (m master) }.

7.4.3. Mémoire partagée

La mémoire partagée ne permet qu'un seul accès à la fois. Cette contrainte est exprimée par la propriété suivante :

P_6 : *if { the shared memory is owned }, then { (p_0 master) ⊕ (p_1 master) ⊕ (p_2 master) ⊕… }.*

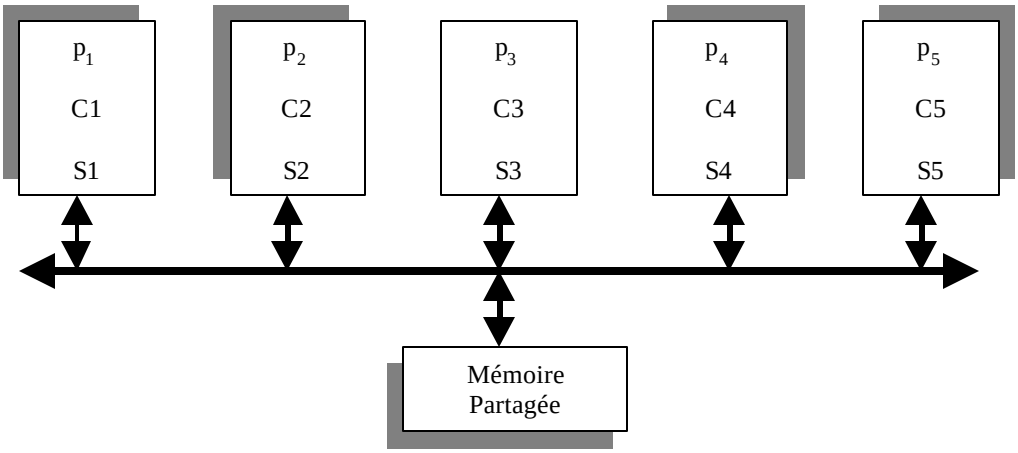
7.4.4. Des propriétés de vivacité

- *"There exists a state in which all the units are idle"*.
- **"***There exists a state in which all the units are working"*.
- *"After a memory access, the bus is free"*.
- **"***It may happen that only one processor accesses the shared memory"*.
- *"It may happen that only one processor is working"*.
- *"It may happen that only two processors among all are working"*.

7.5. Covérification des propriétés par simulation

Le système mis en jeu est composé de cinq processeurs p_i, i=1…5 (figure 7.14) dont chacun possède sa propre cache C_i. Ces processeurs partagent équitablement la mémoire principale. Un seul processeur à la fois peut avoir accès à la mémoire; les autres, en cas d'une requête d'accès, se mettent en file d'attente. Il se peut qu'à un certain moment, une partie de la mémoire partagée par deux ou plusieurs processeurs soit réécrite par l'un des processeurs sans que les autres se rendent compte; il y a donc une incohérence des données des caches. D'où, l'intérêt de l'intégration d'un protocole de cohérence des caches au système original. Dans notre cas, nous avons implémenté le protocole de cohérence des caches dit de diffusion (« broadcasting ») : chaque fois qu'une modification de la mémoire partagée a eu lieu, un message est envoyé à tous les processeurs dont les caches

abritent les blocs modifiés, pour les informer de faire impérativement une mise à jour de leurs caches. La gestion du protocole est faite par les modules d'écoute S_i, i=1…5. Chaque processeur, y compris sa cache et son module d'écoute, est décrit par un thread.



p_i (i=1…5) : Processeurs, C_i (i=1…5) : Caches, S_i (i=1…5) : Modules d'écoute

Figure 7.14 : Système multiprocesseur à mémoire partagée

Un exemple de spécification du fonctionnement pour les cinq processeurs (c'est à dire une description d'un cas de fonctionnement) est donné ci-dessous :

p_1	*p_1 lit le bloc 0 de la mémoire partagée à chaque cycle d'horloge durant toute la simulation.*
p_2	*idem que p_1*
p_3	*Au cycle 1 (horloge locale), p_3 réécrit le bloc 0 de la mémoire partagée par des entiers tous égaux à 33.*
p_4	*p_4 réécrit, au cycle 20, (horloge locale) le bloc 0 de la mémoire partagée par des entiers tous égaux à 44.*
p_5	*p_5, à son tour, réécrit toutes les cases du bloc 0 de la mémoire partagée par le nombre 55, au cycle 40 (horloge locale).*

Chaque processeur doit accomplir, durant toute la durée de la simulation, la tâche qui lui est affectée dans la description susmentionnée. Il s'agit en fait de lire ou

écrire les cases de la mémoire partagée. Cela pourrait causer une incohérence des caches. Pour mettre ceci en évidence, nous avons observé le comportement des différentes caches durant toute la simulation à l'aide des propriétés d'écoute. Ces dernières lit en permanence les états toutes les caches et les affichent. Les résultats obtenus dans les deux cas, avec ou sans protocole de cohérence des caches, sont illustrés à la section 7.6.2.

Propriété	**Code en CTL**

P_1	`AF(((m.busy=1) * (p0.Cdevice.state=owned)) -> (!(p1.Cdevice.state=invalid) * !(p2.Cdevice.state=invalid)));`
	`AF(((m.busy=1) * (p1.Cdevice.state=owned)) -> (!(p0.Cdevice.state=invalid)* !(p2.Cdevice.state=invalid)));`
	`AF(((m.busy=1) * (p2.Cdevice.state=owned)) -> (!(p0.Cdevice.state=invalid) * !(p1.Cdevice.state=invalid)));`
	…

P_2	`AG(p0.Cdevice.writable=1->!(p2.Cdevice.readable=1));`
	`AG(p1.Cdevice.writable=1->!(p0.Cdevice.readable=1));`
	`AG(p1.Cdevice.writable=1->!(p2.Cdevice.readable=1));`
	`AG(p2.Cdevice.writable=1->!(p0.Cdevice.readable=1));`
	`AG(p2.Cdevice.writable=1->!(p1.Cdevice.readable=1));`
	…

P_3	`AG((p0.Cdevice.readable=1)*(p1.Cdevice.readable=1) -> (((p0.Cdevice.data=1) * (p1.Cdevice.data=1)) + ((p0.Cdevice.data=0) * (p1.Cdevice.data=0))));` …

P_4	`AG((m.datam=1)*(p0.Cdevice.readable=1)> (p0.Cdevice.data=1));`
	`AG(m.datam=1)*(p1.Cdevice.readable=1)-> (p1.Cdevice.data=1));`
	`AG(m.datam=1)*(p2.Cdevice.readable=1)-> (p2.Cdevice.data=1));`
	…

P_5	`AG(((p0.Bdevice.waiting=1) + (p1.Bdevice.waiting=1) + (p2.Bdevice.waiting=1) + … ) -> EF(m.busy=1));` …
P_6	`AG(m.busy=1 -> ((p0.Bdevice.waiting=0) + (p1.Bdevice.waiting=0) + (p2.Bdevice.waiting=0)+ … );` …

Tableau 7.1 : Version CTL des propriétés

7.6. Implémentation et résultats

7.6.1. Cas de la vérification formelle

Le système est décrit par un sous-ensemble de Verilog augmenté d'une fonction de non-déterminisme et de variables symboliques [98, 110]. Les propriétés à vérifier sont décrites en CTL («Computation Tree Logic»). Pour accomplir la tâche de vérification, nous avons utilisé l'outil VIS sur une machine ULTRASPARC station-1 dotée d'une mémoire principale de 252 Mo.

Toutes les propriétés que nous avons considérées, à titre d'exemple, pour la spécification du protocole de la cohérence des caches, ont été évaluées positives (tableau 7.2). Des contraintes d'équité peuvent être ajoutées afin d'éviter que le système tourne en rond dans le même état et ainsi le libérer des points morts («deadlocks») s'il y a lieu. Nous avons aussi étudié l'effet de la longueur des données des caches et celui du nombre de processeurs sur les paramètres de l'analyse d'accessibilité tels que la taille des BDDs et le temps d'analyse. Les résultats obtenus (figure 7.15) démontrent que la taille des BDDs évolue exponentiellement avec l'accroissement du nombre des processeurs et de la longueur des données des caches. Le même effet est observé sur le temps de vérification.

Propriété (n_p=3, dw=1)	**Temps de vérification (s)**	**Mémoire utilisée (MB)**	**Valide?**
P_1	9.3	4.5	Ok
P_2	0.5	6.7	Ok
P_3	4.66	5.4	Ok
P_4	0.66	7.4	Ok
P_5	0.27	2.3	Ok
P_6	0.03	2.6	Ok

Tableau 7.2 : Résultats de la vérification des propriétés

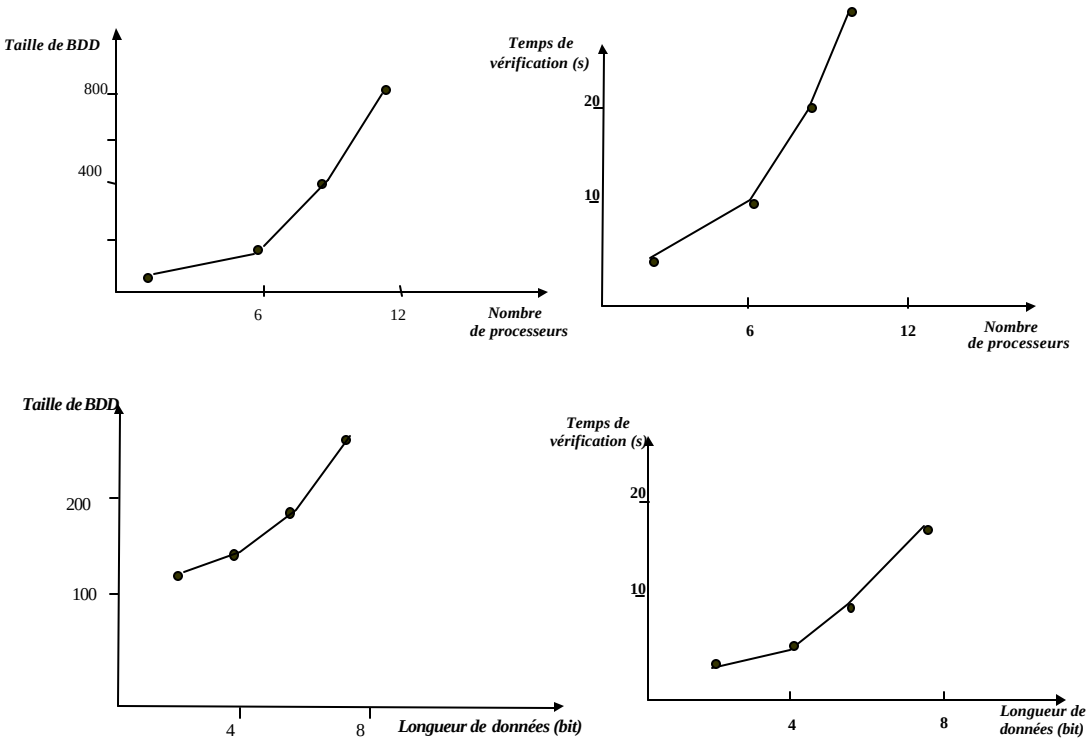


Figure 7.15 : Effets de la longueur des données et du nombre de processeurs sur la taille des BDDs et le temps de vérification

7.6.2. Cas de la covérification

Sur le tableau 7.3, nous exposons une trace d'exécution de la simulation du fonctionnement du système global selon la définition donnée à la section 7.5. Nous identifions en particulier l'état de chaque processeur à chaque cycle d'horloge : maître ou esclave. Nous illustrons dans la figure 7.16 les proportions de cycles consommés par chaque processeur pour une simulation faite sur un nombre de 278 cycles. Ces quotas paraissent différents mais si le nombre total de cycles est suffisamment grand, elles seront en hauteur très similaires du fait que le processus d'arbitrage est aléatoire (dans ce cas).

Cycles de l'horloge globale	Processeur maître	Processeurs en attente
0 – 4	p_3	p_1, p_2, p_4, p_5
5 – 6	p_5	p_1, p_2, p_3, p_4
7 – 28	p_1	p_2, p_3, p_4, p_5
29 – 54	p_2	p_1, p_3, p_4, p_5
55 – 78	p_4	p_1, p_2, p_3, p_5
79 – 104	p_5	p_1, p_2, p_3, p_4
105 – 123	p_1	p_2, p_3, p_4, p_5
124 – 140	p_3	p_1, p_2, p_4, p_5
141 – 164	p_2	p_1, P3, p_4, p_5
165 – 177	p_5	p_1, p_2, p_3, p_4
178 – 196	p_2	p_1, p_3, p_4, p_5
197 – 221	p_1	p_2, p_3, p_4, p_5
222 – 228	p_3	p_1, p_2, p_4, p_5
229 – 235	p_4	p_1, p_2, p_3, p_5
236 – 250	p_5	p_1, p_2, p_3, p_4
251 – 277	p_3	p_1, p_2, p_4, p_5
...	...	...

Tableau 7.3 : Simulation du fonctionnement du système

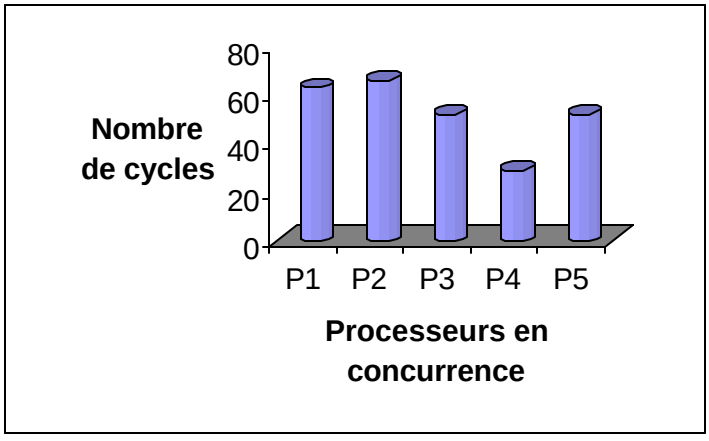


Figure 7.16 : Nombre de cycles consommés par chaque processeur (dans un total de cycles =278)

- Sans protocole :

Dans un premier temps, nous avons fait la simulation de l'ensemble des processeurs sans faire appel aux services du module de protocole de cohérence qui veille sur la bonne conduite du transfert de données entre la mémoire partagée et les mémoires caches. En exécutant la spécification faite ci-haut, nous avons suivi l'évolution des contenus de toutes les mémoires (figures 7.17 -21).

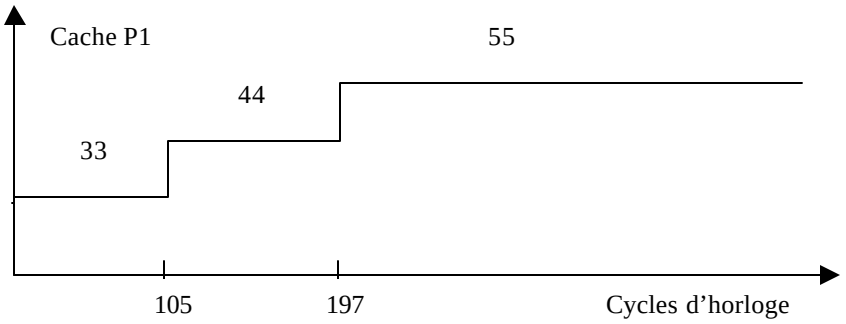


Figure 7.17 : Répercussion de la réécriture de la mémoire partagée sur la cache de p_1

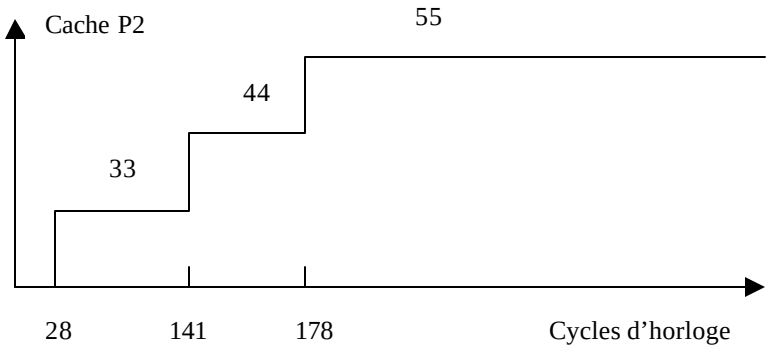


Figure 7.18 : Répercussion de la réécriture de la mémoire partagée sur la cache de p_2

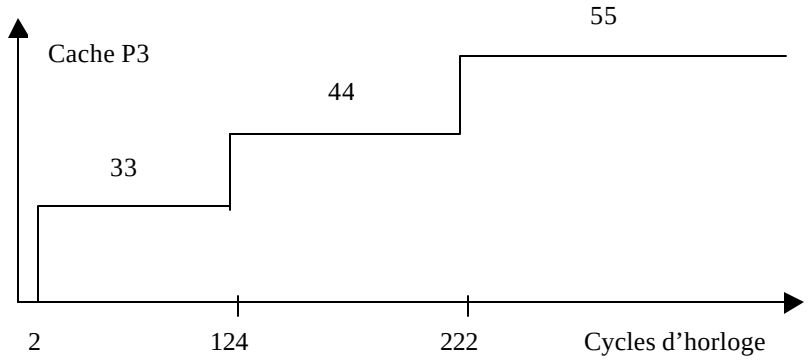


Figure 7.19 : Répercussion de la réécriture de la mémoire partagée sur la cache de p_3

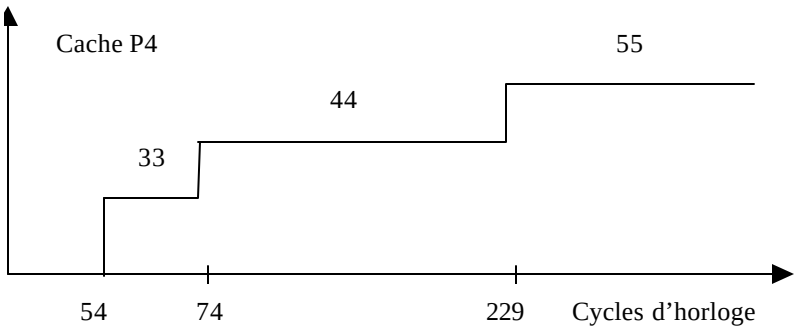


Figure 7.20 : Répercussion de la réécriture de la mémoire partagée sur la cache de p_4

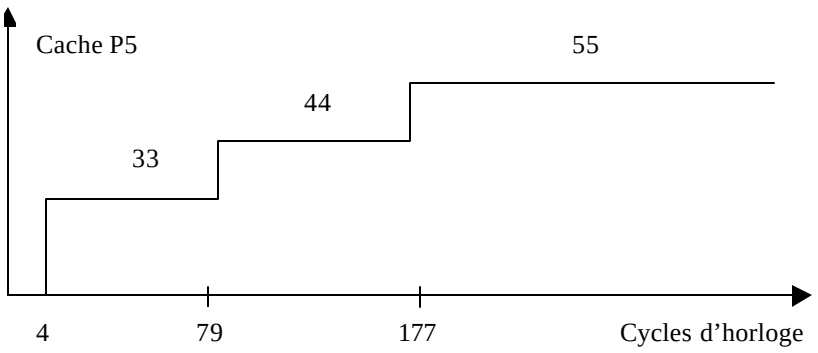


Figure 7.21 : Répercussion de la réécriture de la mémoire partagée sur la cache de p_5

En examinant ces figures, nous constatons rapidement que les gabarits des différentes courbes sont très distincts. Il y a donc une incohérence des données des cinq caches qui partagent toutes le même bloc dans la mémoire partagée (voir spécification ci-avant). Les cinq processeurs sont en fonctionnement continu et risquent de manipuler des données invalides du fait qu'ils ne sont pas avertis des changements ayant lieu au niveau de la mémoire partagée.

- **Avec protocole :**

Le protocole de cohérence des caches ainsi que tous les modules du système sont implémentés à l'aide des threads de Java. Dans ce cas, le module du protocole est toujours en veille et détecte toute modification de la mémoire partagée; c'est lui qui ordonne aux processeurs de faire la mise à jour de leurs caches si c'est nécessaire. Avec l'application de ce protocole, les incohérences notées dans le cas « sans protocole » sont corrigées et les caches, d'après les figures 7.22-26, semblent avoir toutes le même gabarit de courbes. Certaines différences légères entre les instants de casse des courbes sont conséquence de l'exécution des transferts de contrôle entre les processeurs.

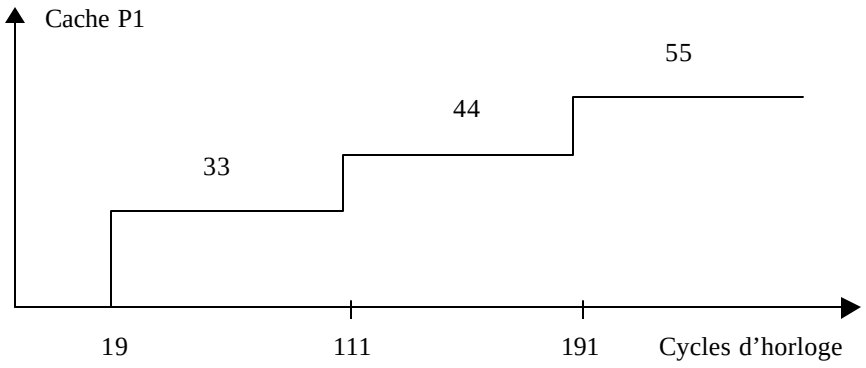


Figure 7.22 : Répercussion de la réécriture de la mémoire partagée sur la cache de p_1

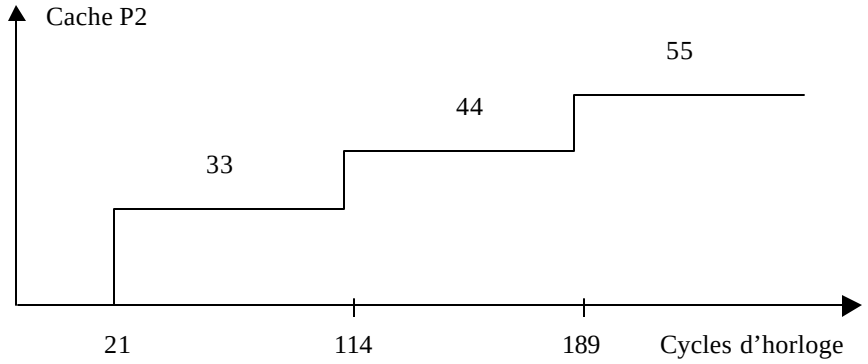


Figure 7.23 : Répercussion de la réécriture de la mémoire partagée sur la cache de p_2

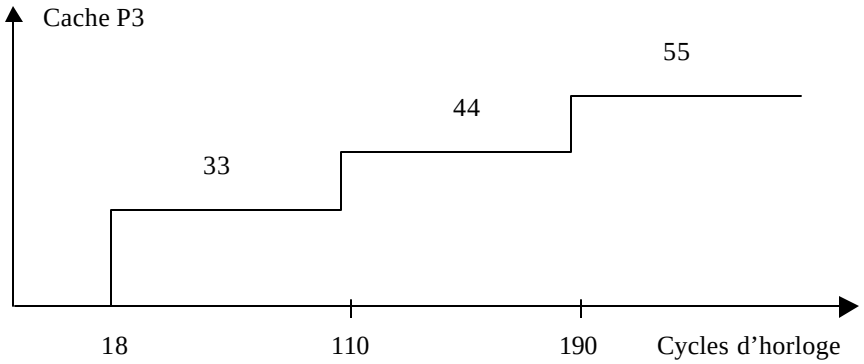


Figure 7.24 : Répercussion de la réécriture de la mémoire partagée sur la cache de p_3

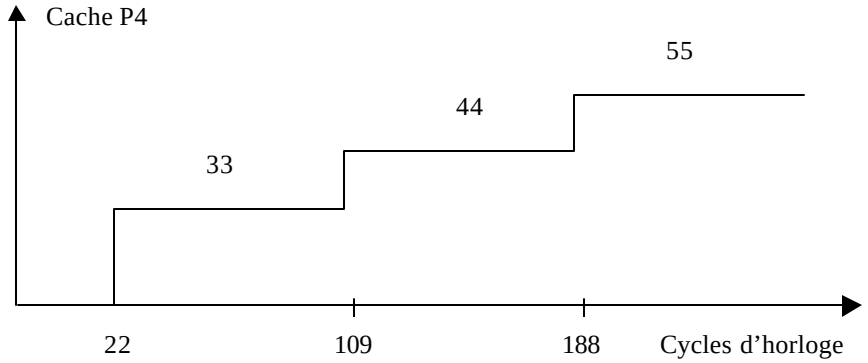


Figure 7.25 : Répercussion de la réécriture de la mémoire partagée sur la cache de p_4

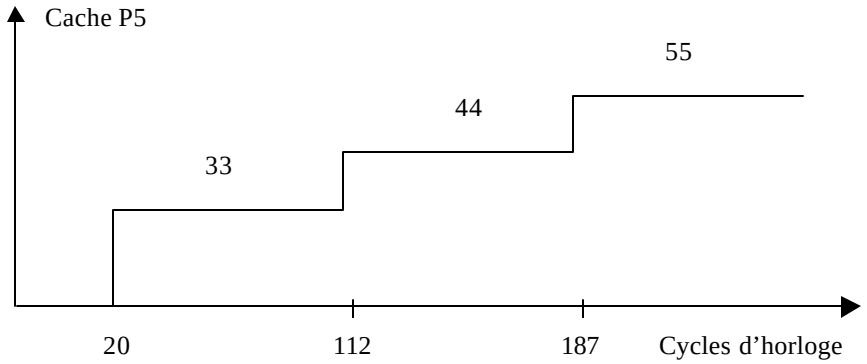


Figure 7.26 : Répercussion de la réécriture de la mémoire partagée sur la cache de p_5

7.7. Conclusion

Dans ce chapitre, nous avons expérimenté la vérification formelle et la covérification d'un protocole de cohérence des caches pour un système multiprocesseur à mémoire partagée. En effet, nous avons utilisé principalement l'outil VIS pour faire de la vérification formelle du système étudié en faisant appel à la méthode « model checking ». Nous avons vérifié avec succès des propriétés décrites en CTL contre l'implémentation du système décrite en Verilog. En plus, nous avons étudié l'effet du nombre des processeurs et celui de la taille des données des caches sur les paramètres de la vérification formelle : la taille des BDDs et le temps d'exécution. Ces derniers évoluent exponentiellement en réponse aux accroissements du nombre des processeurs et de la taille des données des caches. Ceci montre que cette méthode de vérification formelle commence à souffrir et à ne plus garantir ses performances dès que la complexité du système étudié deviendra assez importante (tel est souvent le cas en industrie). Par ailleurs, nous avons mené une covérification du système en exploitant nos propositions dans les chapitres précédents. Ceci nous a permis de suivre l'évolution des données de toutes les mémoires considérés dans le système à cinq processeurs et à mémoire partagée. Une simulation d'un tel système sans tenir compte du protocole de cohérence montre

clairement une incohérence des caches. Cette situation illégale est redressée par la considération du protocole.

Par ce cas d'étude, nous avons montré l'apport complémentaire de la covérification à celui de la vérification formelle surtout dans le cas des systèmes qui sont orientés flux de données. La manipulation de ces derniers est fastidieuse dans le cas de la vérification formelle; cependant, une covérification par cosimulation pourrait bien être d'une grande importance pour ce type de systèmes ainsi que pour les autres tout en investissant un minimum d'effort et de coût.

Chapitre 8

Conclusion et perspectives

Conclusion

Dans cette thèse intitulée « Covérification des systèmes intégrés », nous avons fait le tour des techniques et des technologies de conception, de vérification, de covérification et d'autres domaines connexes, auxquelles s'ajoutent nos contributions qui s'insèrent dans le champ de la covérification des systèmes mixtes à deux composantes logicielle et matérielle.

Ce travail est implicitement composé de deux parties : l'une à caractère introductif et l'autre à caractère contributif. En effet, dans le chapitre 2, nous survolons les méthodologies de conception et de vérification avec leurs outils; puis nous introduisons les notions de codesign et de covérification L/M dans le chapitre 3, tout en discutant la problématique de cette thèse. Certes, ceci permet au lecteur de se situer et de comprendre facilement nos propos dans le reste des chapitres. Ces derniers font le point sur notre méthodologie de covérification basée sur le principe

du « multithreading », sur la technique de spécification des propriétés à covérifier avec CPL et sur l'outil semi-automatique dédié à ce genre de covérification, JACOV.

Nous avons classifié la covérification en trois catégories : par « bread-boarding », par cosimulation et par des méthodes formelles. La première s'applique après la phase du prototypage à l'instar d'un test matériel des entrées-sorties d'un système. Celle-ci est jugée très primitive et à risque. En effet, s'il s'est avéré, après avoir testé le prototype, que ce dernier est porteur d'une anomalie, alors il faut absolument réviser le design pour localiser la cause des erreurs, l'anéantir et ensuite générer un autre prototype. Cette procédure pourrait être réitérée jusqu'à ce que le test ne détecte aucun mauvais comportement. La seconde catégorie fait appel aux services des outils de la cosimulation. Cette dernière permet de mettre en communication plusieurs simulateurs. Notre travail s'inscrit dans cette deuxième catégorie. L'usage de la cosimulation commence récemment à gagner de l'ampleur dans le domaine des systèmes L/M. La troisième catégorie s'apparente en fait avec la vérification formelle du matériel; le caractère bidimensionnel des systèmes L/M se trouve donc réduit à une seule dimension, ce qui limite la contrôlabilité et l'observabilité du processus «covérificationnel» de ce genre de systèmes.

Nous avons proposé une méthodologie de covérification basée sur le « multithreading ». Celle-ci s'effectue en quatre procédures successives : (1) la mise en threads de la partie logicielle, (2) la sauvegarde des paramètres révélateurs de la partie matérielle dans des registres accessibles à partir de la partie logicielle, (3) la description des propriétés du système et (4) la cosimulation de l'ensemble « système et propriétés ». Nous avons implémenté une assise à l'aide du langage Java pour servir d'ingrédients à des applications ultérieures; il s'agit du « *package* » nommé *coverification*. Celui-ci doit figurer sur la liste des modules importés au sein d'un code utilisateur. Le « package » *coverification* est ouvert pour ajouter d'autres objets nécessaires à des applications particulières ou pour enrichir la spécification des objets existants. Actuellement, le processus de covérification s'effectue à l'aide de l'outil JACOV.

Dédié à la covérification, JACOV est un outil semi-automatique que nous avons développé afin de concrétiser nos propos. Son noyau est réalisé à la lumière de notre méthodologie de covérification. Les entrées de JACOV sont au nombre de trois : un fichier contenant l'implémentation écrite en Java, un fichier de propriétés décrites en CPL et une pile de vecteurs de test. À sa sortie, JACOV génère les résultats de validation des différentes propriétés. CPL a été considéré pour pouvoir exprimer les propriétés facilement et séparément du code de l'implémentation. L'utilisateur n'aura pas à se préoccuper de l'emplacement de chacune des propriétés au sein du code de l'implémentation. Les propriétés écrites en CPL seront traduites en threads de Java et ensuite rajoutées aux autres threads décrivant l'implémentation du système. Actuellement, JACOV manipule juste un modèle homogène, c'est-à-dire le système étudié est modélisé avec un seul langage (dans notre cas, c'est Java) en réduisant la partie matérielle à un tableau de registres accessibles.

Les résultats obtenus démontrent la faisabilité d'une telle méthodologie. Cependant, il faut payer le coût de représenter le système à examiner en threads de Java et les propriétés en CPL, d'autant plus si le code était initialement écrit dans un autre langage que Java. Par ailleurs, si les propriétés répondent favorablement aux vecteurs de test appliqués à l'entrée du système, ce résultat reste valable pour ces vecteurs et ne peut être étendu systématiquement à tous les vecteurs possibles. En pratique, on ne s'intéresse pas à tous ces vecteurs possibles mais juste à certains d'entre eux qui satisfont aux contraintes de fonctionnement (qui jouent le rôle d'un filtre). Avec cette considération, la covérification aura plus de crédibilité si des vecteurs représentatifs sont appliqués.

Utilité de cette thèse

Cette thèse est d'une grande utilité tant pour un lecteur non-spécialiste voulant se faire une idée sur la covérification des systèmes intégrés que pour un chercheur désireux de poursuivre ce travail ou de l'améliorer. Par ailleurs, jusqu'à présent, les environnements professionnels connus de cosimulation tels que *Eaglei*

de *iViewLogic-Synopsys* et *CVE-Seamless* de *Menthor Graphics* ne supportent pas les threads de Java et n'ont aucune interface entre Java et Verilog/VHDL; alors il serait très bénéfique à ces outils d'intégrer notre méthodologie afin de permettre à l'opérateur d'identifier le processus de covérification et de l'utiliser sans se faire de complications. Nous formulons ces suggestions après avoir utilisé et examiné ces outils au cours d'un séjour industriel à la compagnie canadienne de télécommunication *NORTEL*. En effet, nous avons réalisé un pont de communication entre les parties matérielle (en Verilog) et logicielle (en C) du design d'une puce de télécommunication (appelé *LUNAR*) en utilisant l'environnement Eaglei. Nous ne pouvons donner plus de détails concernant ce projet puisqu'il s'agit d'une propriété privée de *NORTEL*.

Travaux futurs

Comme travaux futurs, nous suggérons de poursuivre ce travail dans les orientations suivantes : CPL2Java, API Java-HDL, Interface graphique de JACOV et Perfectionnement du test.

CPL2Java est l'outil responsable de la tâche de traduction (« translator ») du code CPL des propriétés en threads de Java. Jusqu'à présent, cette opération se fait de manière manuelle. Alors, son automatisation est fortement recommandée pour permettre plus d'autonomie à JACOV.

API Java-HDL est l'interface assurant une communication transparente entre les threads Java et le simulateur du code HDL de manière à ce qu'une cosimulation soit possible avec Eaglei ou Seamless.

Actuellement, JACOV est sans interface graphique. Il lui faut un éditeur de code et d'affichage de résultats plus performant. Nous projetons de rendre ses versions futures plus professionnelles.

Java avec ses threads a été l'objet de notre choix parmi les langages à threads grâce surtout à sa portabilité qui permet à ses applications d'être opérationnelles sur différentes plates-formes. Cependant, nous sommes persuadé que des résultats de covérification pareils à ceux de JACOV pourront être obtenus avec des langages tels que C++ ou Ada. Si la portabilité n'est pas requise, a priori, l'utilisation de C++ et d'Ada faciliteront davantage la communication respectivement avec Verilog et VHDL.

En prenant en considération les extensions susmentionnées, JACOV pourrait bien être un outil de covérification pertinent et capable d'effectuer des tests guidés par la couverture des propriétés. Comme nous l'avons discuté dans le chapitre 7, le compte rendu de l'activation des propriétés du système étudié sert de visée pour rectifier le prochain tir c'est-à-dire pour contrôler la génération de test et ainsi composer les vecteurs de test désirés.

Annexe 1

Un aperçu sur le code de l'analyseur syntaxique de CPL

Code en JavaCC

```
options {
  LOOKAHEAD = 10;
  CHOICE_AMBIGUITY_CHECK = 12;
  OTHER_AMBIGUITY_CHECK = 1;
  STATIC = true;
  DEBUG_PARSER = false;
  DEBUG_LOOKAHEAD = false;
  DEBUG_TOKEN_MANAGER = false;
  ERROR_REPORTING = true;
  JAVA_UNICODE_ESCAPE = false;
  UNICODE_INPUT = false;
  IGNORE_CASE = true;
  USER_TOKEN_MANAGER = false;
  USER_CHAR_STREAM = false;
  BUILD_PARSER = true;
  BUILD_TOKEN_MANAGER = true;
  SANITY_CHECK = true;
  FORCE_LA_CHECK = true;
  // OUTPUT_DIRECTORY=true;
}

PARSER_BEGIN(azizi_parser5)
import java.io.*;
public class azizi_parser5
  {
  public static void main(String[] args) throws ParseException
  {
    System.out.println("\n Parser for Coverification Properties Description
Language (CPL)");
    System.out.println("\n Elaborated by Coverification Group: ");
    System.out.println("\n Mostafa Azizi, Université de Montréal, Canada");
    System.out.println(" E.-M. Aboulhamid, Université de Montréal,
Canada");
    System.out.println(" S. Tahar, Concordia University, Canada");
    System.out.println("\n Copyright@1999 by Coverification Group");
    System.out.println(" \n\n\n\n Looking for the source code ...\n\n");

    InputStream in = null;
    if (args.length == 0) in = System.in;
    else
    try
    {
    in = new FileInputStream(args[0]);
    }
    catch (IOException e)
```

```
    {
    System.out.println("File not found");
    }
    azizi_parser5 parser = new azizi_parser5(in);
    // parser.Input();
    while (true) // (!(args[0].EOF))
                   { parser.Input(); }
  }
}
PARSER_END(azizi_parser5)

SKIP :
{
  " "
| "\t"
| "\n"
| "\r"
| <"//" (~["\n", "\r"])* ("\n" | "\r" | "\r\n")>
| <"/*" (~["*"])* "*" (~["/"] (~["*"])* "*")* "/">
}

TOKEN:
{
 <pr : "property">
|<pr_e : "end">
|<fi : "if">
|<sw : "switch">
|<ca : "case">
|<de : "default">
|<co : "concurrent">
|<cs : "sequential">
| <n: "not">
| <T: "true">
| <F: "false">
| <PLUS: "+">
| <MINUS: "-">
| <DIVIDE: "/">
| <MULTIPLY: "*">
| <EQUAL_TEST: "==">
| <EQUAL: "=?">
| <GE: ">=">
| <LE: "<=">
| <G: ">">
| <L: "<">
| <DIFFERENT: "!=?">
| <INTEGER: (<DIGIT>)+>
| <DIGIT: ["0" - "9"]>
| <IDENTIFIER: ["a"-"z","A"-"Z","_"] ( ["a"-"z","A"-"Z","_","0"-"9"] )*>
| <other: ["a"-"z","A"-"Z","_"] ( ["a"-"z","A"-"Z","_","0"-"9"] )*>
| <LETTER: ["a" - "z", "A" - "Z"]>

}
TOKEN:
{
<FLOAT:
       (["0"-"9"])+ "." (["0"-"9"])* (<EXPONENT>)?
       |"." (["0"-"9"])+ (<EXPONENT>)?
       |(["0"-"9"])+ <EXPONENT>
       |(["0"-"9"])+ (<EXPONENT>)?
>
| <EXPONENT: ["e", "F"] (["+", "-"])? (["0"-"9"])+ >
}
```

```
void Input() :
{}
{
MatchedBraces()
}

void MatchedBraces() :
{}
{
<IDENTIFIER> <pr> expression_super() <pr_e>
|
(<co>|<cs>) "(" (<IDENTIFIER> ",")+  <other> ")" <pr_e>

}

void expression_super():
{}
{
expression_inf()
|
<fi> "(" condition() ")" expression_inf()
|
<sw> "(" <IDENTIFIER> ")" "{" (<ca> <INTEGER> ":" expression_inf() )+ <de>
":" expression_inf() "}"
}

void expression_inf():
{}
{
"(" condition() ")" ";"
|
<n> "(" condition() ")" ";"
}
void condition():
{}
{
term() ((<PLUS>|<MINUS>|<DIVIDE>|<MULTIPLY>) term())*
(<EQUAL_TEST>|<GE>|<LE>|<DIFFERENT>|<G>|<L>) term()
((<PLUS>|<MINUS>|<DIVIDE>|<MULTIPLY>) term())*
}

void term():
{}
{
number()
|
<IDENTIFIER>
}

void number():
{}
{
<INTEGER>
|
<FLOAT>
}
```

Code en Java

```
import java.io.*;
public class azizi_parser5 implements azizi_parser5Constants {
  public static void main(String[] args) throws ParseException
  {
    System.out.println("\n Parser for Coverification Properties Description
Language (CPL)");
    System.out.println("\n Elaborated by Coverification Group: ");
    System.out.println("\n Mostafa Azizi, Université de Montréal, Canada");
    System.out.println(" E.-M. Aboulhamid, Université de Montréal,
Canada");
    System.out.println(" S. Tahar, Concordia University, Canada");
    System.out.println("\n Copyright@1999 by Coverification Group");
    System.out.println(" \n\n\n\n Looking for the source code ...\n\n");
    InputStream in = null;
    if (args.length == 0) in = System.in;
    else
    try
    {
    in = new FileInputStream(args[0]);
    }
    catch (IOException e)
    {
    System.out.println("File not found");
    }
    azizi_parser5 parser = new azizi_parser5(in);
    // parser.Input();
    while (true) // (!(args[0].EOF))
                        { parser.Input(); }
  }

  static final public void Input() throws ParseException {
    MatchedBraces();
  }

  static final public void MatchedBraces() throws ParseException {
    if (jj_2_4(10)) {
      jj_consume_token(IDENTIFIER);
      jj_consume_token(pr);
      expression_super();
      jj_consume_token(pr_e);
    } else if (jj_2_5(10)) {
      if (jj_2_1(10)) {
        jj_consume_token(co);
      } else if (jj_2_2(10)) {
        jj_consume_token(cs);
      } else {
        jj_consume_token(-1);
        throw new ParseException();
      }
      jj_consume_token(36);
      label_1:
      while (true) {
        jj_consume_token(IDENTIFIER);
        jj_consume_token(37);
        if (jj_2_3(10)) {
          ;
        } else {
          break label_1;
        }
```

```
    }
    jj_consume_token(other);
    jj_consume_token(38);
    jj_consume_token(pr_e);
  } else {
    jj_consume_token(-1);
    throw new ParseException();
  }
}

static final public void expression_super() throws ParseException {
  if (jj_2_7(10)) {
    expression_inf();
  } else if (jj_2_8(10)) {
    jj_consume_token(fi);
    jj_consume_token(36);
    condition();
    jj_consume_token(38);
    expression_inf();
  } else if (jj_2_9(10)) {
    jj_consume_token(sw);
    jj_consume_token(36);
    jj_consume_token(IDENTIFIER);
    jj_consume_token(38);
    jj_consume_token(39);
    label_2:
    while (true) {
      jj_consume_token(ca);
      jj_consume_token(INTEGER);
      jj_consume_token(40);
      expression_inf();
      if (jj_2_6(10)) {
        ;
      } else {
        break label_2;
      }
    }
    jj_consume_token(de);
    jj_consume_token(40);
    expression_inf();
    jj_consume_token(41);
  } else {
    jj_consume_token(-1);
    throw new ParseException();
  }
}

static final public void expression_inf() throws ParseException {
  if (jj_2_10(10)) {
    jj_consume_token(36);
    condition();
    jj_consume_token(38);
    jj_consume_token(42);
  } else if (jj_2_11(10)) {
    jj_consume_token(n);
    jj_consume_token(36);
    condition();
    jj_consume_token(38);
    jj_consume_token(42);
  } else {
    jj_consume_token(-1);
    throw new ParseException();
  }
```

```
}

static final public void condition() throws ParseException {
  term();
  label_3:
  while (true) {
    if (jj_2_12(10)) {
      ;
    } else {
      break label_3;
    }
    if (jj_2_13(10)) {
      jj_consume_token(PLUS);
    } else if (jj_2_14(10)) {
      jj_consume_token(MINUS);
    } else if (jj_2_15(10)) {
      jj_consume_token(DIVIDE);
    } else if (jj_2_16(10)) {
      jj_consume_token(MULTIPLY);
    } else {
      jj_consume_token(-1);
      throw new ParseException();
    }
    term();
  }
  if (jj_2_17(10)) {
    jj_consume_token(EQUAL_TEST);
  } else if (jj_2_18(10)) {
    jj_consume_token(GE);
  } else if (jj_2_19(10)) {
    jj_consume_token(LE);
  } else if (jj_2_20(10)) {
    jj_consume_token(DIFFERENT);
  } else if (jj_2_21(10)) {
    jj_consume_token(G);
  } else if (jj_2_22(10)) {
    jj_consume_token(L);
  } else {
    jj_consume_token(-1);
    throw new ParseException();
  }
  term();
  label_4:
  while (true) {
    if (jj_2_23(10)) {
      ;
    } else {
      break label_4;
    }
    if (jj_2_24(10)) {
      jj_consume_token(PLUS);
    } else if (jj_2_25(10)) {
      jj_consume_token(MINUS);
    } else if (jj_2_26(10)) {
      jj_consume_token(DIVIDE);
    } else if (jj_2_27(10)) {
      jj_consume_token(MULTIPLY);
    } else {
      jj_consume_token(-1);
      throw new ParseException();
    }
    term();
  }
```

```
}

static final public void term() throws ParseException {
  if (jj_2_28(10)) {
    number();
  } else if (jj_2_29(10)) {
    jj_consume_token(IDENTIFIER);
  } else {
    jj_consume_token(-1);
    throw new ParseException();
  }
}

static final public void number() throws ParseException {
  if (jj_2_30(10)) {
    jj_consume_token(INTEGER);
  } else if (jj_2_31(10)) {
    jj_consume_token(FLOAT);
  } else {
    jj_consume_token(-1);
    throw new ParseException();
  }
}

static final private boolean jj_2_1(int xla) {
  jj_la = xla; jj_lastpos = jj_scanpos = token;
  boolean retval = !jj_3_1();
  jj_save(0, xla);
  return retval;
}

static final private boolean jj_2_2(int xla) {
  jj_la = xla; jj_lastpos = jj_scanpos = token;
  boolean retval = !jj_3_2();
  jj_save(1, xla);
  return retval;
}

static final private boolean jj_2_3(int xla) {
  jj_la = xla; jj_lastpos = jj_scanpos = token;
  boolean retval = !jj_3_3();
  jj_save(2, xla);
  return retval;
}

static final private boolean jj_2_4(int xla) {
  jj_la = xla; jj_lastpos = jj_scanpos = token;
  boolean retval = !jj_3_4();
  jj_save(3, xla);
  return retval;
}

static final private boolean jj_2_5(int xla) {
  jj_la = xla; jj_lastpos = jj_scanpos = token;
  boolean retval = !jj_3_5();
  jj_save(4, xla);
  return retval;
}

static final private boolean jj_2_6(int xla) {
  jj_la = xla; jj_lastpos = jj_scanpos = token;
  boolean retval = !jj_3_6();
  jj_save(5, xla);
```

```
    return retval;
  }

  static final private boolean jj_2_7(int xla) {
    jj_la = xla; jj_lastpos = jj_scanpos = token;
    boolean retval = !jj_3_7();
    jj_save(6, xla);
    return retval;
  }

  static final private boolean jj_2_8(int xla) {
    jj_la = xla; jj_lastpos = jj_scanpos = token;
    boolean retval = !jj_3_8();
    jj_save(7, xla);
    return retval;
  }

  static final private boolean jj_2_9(int xla) {
    jj_la = xla; jj_lastpos = jj_scanpos = token;
    boolean retval = !jj_3_9();
    jj_save(8, xla);
    return retval;
  }

  static final private boolean jj_2_10(int xla) {
    jj_la = xla; jj_lastpos = jj_scanpos = token;
    boolean retval = !jj_3_10();
    jj_save(9, xla);
    return retval;
  }

  static final private boolean jj_2_11(int xla) {
    jj_la = xla; jj_lastpos = jj_scanpos = token;
    boolean retval = !jj_3_11();
    jj_save(10, xla);
    return retval;
  }

  static final private boolean jj_2_12(int xla) {
    jj_la = xla; jj_lastpos = jj_scanpos = token;
    boolean retval = !jj_3_12();
    jj_save(11, xla);
    return retval;
  }

  static final private boolean jj_2_13(int xla) {
    jj_la = xla; jj_lastpos = jj_scanpos = token;
    boolean retval = !jj_3_13();
    jj_save(12, xla);
    return retval;
  }

  static final private boolean jj_2_14(int xla) {
    jj_la = xla; jj_lastpos = jj_scanpos = token;
    boolean retval = !jj_3_14();
    jj_save(13, xla);
    return retval;
  }

  static final private boolean jj_2_15(int xla) {
    jj_la = xla; jj_lastpos = jj_scanpos = token;
    boolean retval = !jj_3_15();
    jj_save(14, xla);
```

```
    return retval;
  }

  static final private boolean jj_2_16(int xla) {
    jj_la = xla; jj_lastpos = jj_scanpos = token;
    boolean retval = !jj_3_16();
    jj_save(15, xla);
    return retval;
  }

  static final private boolean jj_2_17(int xla) {
    jj_la = xla; jj_lastpos = jj_scanpos = token;
    boolean retval = !jj_3_17();
    jj_save(16, xla);
    return retval;
  }

  static final private boolean jj_2_18(int xla) {
    jj_la = xla; jj_lastpos = jj_scanpos = token;
    boolean retval = !jj_3_18();
    jj_save(17, xla);
    return retval;
  }

  static final private boolean jj_2_19(int xla) {
    jj_la = xla; jj_lastpos = jj_scanpos = token;
    boolean retval = !jj_3_19();
    jj_save(18, xla);
    return retval;
  }

  static final private boolean jj_2_20(int xla) {
    jj_la = xla; jj_lastpos = jj_scanpos = token;
    boolean retval = !jj_3_20();
    jj_save(19, xla);
    return retval;
  }

  static final private boolean jj_2_21(int xla) {
    jj_la = xla; jj_lastpos = jj_scanpos = token;
    boolean retval = !jj_3_21();
    jj_save(20, xla);
    return retval;
  }

  static final private boolean jj_2_22(int xla) {
    jj_la = xla; jj_lastpos = jj_scanpos = token;
    boolean retval = !jj_3_22();
    jj_save(21, xla);
    return retval;
  }

  static final private boolean jj_2_23(int xla) {
    jj_la = xla; jj_lastpos = jj_scanpos = token;
    boolean retval = !jj_3_23();
    jj_save(22, xla);
    return retval;
  }

  static final private boolean jj_2_24(int xla) {
    jj_la = xla; jj_lastpos = jj_scanpos = token;
    boolean retval = !jj_3_24();
    jj_save(23, xla);
```

```
    return retval;
  }

  static final private boolean jj_2_25(int xla) {
    jj_la = xla; jj_lastpos = jj_scanpos = token;
    boolean retval = !jj_3_25();
    jj_save(24, xla);
    return retval;
  }

  static final private boolean jj_2_26(int xla) {
    jj_la = xla; jj_lastpos = jj_scanpos = token;
    boolean retval = !jj_3_26();
    jj_save(25, xla);
    return retval;
  }

  static final private boolean jj_2_27(int xla) {
    jj_la = xla; jj_lastpos = jj_scanpos = token;
    boolean retval = !jj_3_27();
    jj_save(26, xla);
    return retval;
  }

  static final private boolean jj_2_28(int xla) {
    jj_la = xla; jj_lastpos = jj_scanpos = token;
    boolean retval = !jj_3_28();
    jj_save(27, xla);
    return retval;
  }

  static final private boolean jj_2_29(int xla) {
    jj_la = xla; jj_lastpos = jj_scanpos = token;
    boolean retval = !jj_3_29();
    jj_save(28, xla);
    return retval;
  }

  static final private boolean jj_2_30(int xla) {
    jj_la = xla; jj_lastpos = jj_scanpos = token;
    boolean retval = !jj_3_30();
    jj_save(29, xla);
    return retval;
  }

  static final private boolean jj_2_31(int xla) {
    jj_la = xla; jj_lastpos = jj_scanpos = token;
    boolean retval = !jj_3_31();
    jj_save(30, xla);
    return retval;
  }

  static final private boolean jj_3_17() {
    if (jj_scan_token(EQUAL_TEST)) return true;
    if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
    return false;
  }

  static final private boolean jj_3_16() {
    if (jj_scan_token(MULTIPLY)) return true;
    if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
    return false;
  }
```

```
static final private boolean jj_3_27() {
  if (jj_scan_token(MULTIPLY)) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  return false;
}

static final private boolean jj_3_15() {
  if (jj_scan_token(DIVIDE)) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  return false;
}

static final private boolean jj_3_26() {
  if (jj_scan_token(DIVIDE)) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  return false;
}

static final private boolean jj_3_6() {
  if (jj_scan_token(ca)) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  if (jj_scan_token(INTEGER)) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  if (jj_scan_token(40)) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  if (jj_3R_6()) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  return false;
}

static final private boolean jj_3_31() {
  if (jj_scan_token(FLOAT)) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  return false;
}

static final private boolean jj_3_14() {
  if (jj_scan_token(MINUS)) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  return false;
}

static final private boolean jj_3R_9() {
  Token xsp;
  xsp = jj_scanpos;
  if (jj_3_30()) {
  jj_scanpos = xsp;
  if (jj_3_31()) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  } else if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  return false;
}

static final private boolean jj_3_30() {
  if (jj_scan_token(INTEGER)) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  return false;
}

static final private boolean jj_3_25() {
  if (jj_scan_token(MINUS)) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
```

```
  return false;
}

static final private boolean jj_3_13() {
  if (jj_scan_token(PLUS)) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  return false;
}

static final private boolean jj_3_29() {
  if (jj_scan_token(IDENTIFIER)) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  return false;
}

static final private boolean jj_3_12() {
  Token xsp;
  xsp = jj_scanpos;
  if (jj_3_13()) {
  jj_scanpos = xsp;
  if (jj_3_14()) {
  jj_scanpos = xsp;
  if (jj_3_15()) {
  jj_scanpos = xsp;
  if (jj_3_16()) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  } else if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  } else if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  } else if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  if (jj_3R_8()) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  return false;
}

static final private boolean jj_3_24() {
  if (jj_scan_token(PLUS)) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  return false;
}

static final private boolean jj_3R_8() {
  Token xsp;
  xsp = jj_scanpos;
  if (jj_3_28()) {
  jj_scanpos = xsp;
  if (jj_3_29()) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  } else if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  return false;
}

static final private boolean jj_3_28() {
  if (jj_3R_9()) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  return false;
}

static final private boolean jj_3_23() {
  Token xsp;
  xsp = jj_scanpos;
  if (jj_3_24()) {
  jj_scanpos = xsp;
  if (jj_3_25()) {
```

```
    jj_scanpos = xsp;
    if (jj_3_26()) {
    jj_scanpos = xsp;
    if (jj_3_27()) return true;
    if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
    } else if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
    } else if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
    } else if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
    if (jj_3R_8()) return true;
    if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
    return false;
  }

  static final private boolean jj_3R_7() {
    if (jj_3R_8()) return true;
    if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
    Token xsp;
    while (true) {
      xsp = jj_scanpos;
      if (jj_3_12()) { jj_scanpos = xsp; break; }
      if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
    }
    xsp = jj_scanpos;
    if (jj_3_17()) {
    jj_scanpos = xsp;
    if (jj_3_18()) {
    jj_scanpos = xsp;
    if (jj_3_19()) {
    jj_scanpos = xsp;
    if (jj_3_20()) {
    jj_scanpos = xsp;
    if (jj_3_21()) {
    jj_scanpos = xsp;
    if (jj_3_22()) return true;
    if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
    } else if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
    } else if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
    } else if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
    } else if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
    } else if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
    if (jj_3R_8()) return true;
    if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
    while (true) {
      xsp = jj_scanpos;
      if (jj_3_23()) { jj_scanpos = xsp; break; }
      if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
    }
    return false;
  }

  static final private boolean jj_3_11() {
    if (jj_scan_token(n)) return true;
    if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
    if (jj_scan_token(36)) return true;
    if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
    if (jj_3R_7()) return true;
    if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
    if (jj_scan_token(38)) return true;
    if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
    if (jj_scan_token(42)) return true;
    if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
    return false;
  }
```

```
static final private boolean jj_3R_6() {
  Token xsp;
  xsp = jj_scanpos;
  if (jj_3_10()) {
  jj_scanpos = xsp;
  if (jj_3_11()) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  } else if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  return false;
}

static final private boolean jj_3_22() {
  if (jj_scan_token(L)) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  return false;
}

static final private boolean jj_3_10() {
  if (jj_scan_token(36)) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  if (jj_3R_7()) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  if (jj_scan_token(38)) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  if (jj_scan_token(42)) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  return false;
}

static final private boolean jj_3_3() {
  if (jj_scan_token(IDENTIFIER)) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  if (jj_scan_token(37)) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  return false;
}

static final private boolean jj_3_21() {
  if (jj_scan_token(G)) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  return false;
}

static final private boolean jj_3_9() {
  if (jj_scan_token(sw)) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  if (jj_scan_token(36)) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  if (jj_scan_token(IDENTIFIER)) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  if (jj_scan_token(38)) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  if (jj_scan_token(39)) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  Token xsp;
  if (jj_3_6()) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  while (true) {
    xsp = jj_scanpos;
    if (jj_3_6()) { jj_scanpos = xsp; break; }
    if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  }
```

```
  return false;
}

static final private boolean jj_3_8() {
  if (jj_scan_token(fi)) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  if (jj_scan_token(36)) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  if (jj_3R_7()) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  if (jj_scan_token(38)) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  if (jj_3R_6()) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  return false;
}

static final private boolean jj_3R_5() {
  Token xsp;
  xsp = jj_scanpos;
  if (jj_3_7()) {
  jj_scanpos = xsp;
  if (jj_3_8()) {
  jj_scanpos = xsp;
  if (jj_3_9()) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  } else if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  } else if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  return false;
}

static final private boolean jj_3_7() {
  if (jj_3R_6()) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  return false;
}

static final private boolean jj_3_2() {
  if (jj_scan_token(cs)) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  return false;
}

static final private boolean jj_3_20() {
  if (jj_scan_token(DIFFERENT)) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  return false;
}

static final private boolean jj_3_1() {
  if (jj_scan_token(co)) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  return false;
}

static final private boolean jj_3_5() {
  Token xsp;
  xsp = jj_scanpos;
  if (jj_3_1()) {
  jj_scanpos = xsp;
  if (jj_3_2()) return true;
  if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
  } else if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
```

```
    if (jj_scan_token(36)) return true;
    if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
    if (jj_3_3()) return true;
    if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
    while (true) {
      xsp = jj_scanpos;
      if (jj_3_3()) { jj_scanpos = xsp; break; }
      if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
    }
    if (jj_scan_token(other)) return true;
    if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
    if (jj_scan_token(38)) return true;
    if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
    if (jj_scan_token(pr_e)) return true;
    if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
    return false;
  }

  static final private boolean jj_3_4() {
    if (jj_scan_token(IDENTIFIER)) return true;
    if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
    if (jj_scan_token(pr)) return true;
    if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
    if (jj_3R_5()) return true;
    if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
    if (jj_scan_token(pr_e)) return true;
    if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
    return false;
  }

  static final private boolean jj_3_19() {
    if (jj_scan_token(LE)) return true;
    if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
    return false;
  }

  static final private boolean jj_3_18() {
    if (jj_scan_token(GE)) return true;
    if (jj_la == 0 && jj_scanpos == jj_lastpos) return false;
    return false;
  }

  static private boolean jj_initialized_once = false;
  static public azizi_parser5TokenManager token_source;
  static ASCII_CharStream jj_input_stream;
  static public Token token, jj_nt;
  static private int jj_ntk;
  static private Token jj_scanpos, jj_lastpos;
  static private int jj_la;
  static public boolean lookingAhead = false;
  static private boolean jj_semLA;
  static private int jj_gen;
  static final private int[] jj_la1 = new int[0];
  static final private int[] jj_la1_0 = {};
  static final private int[] jj_la1_1 = {};
  static final private JJCalls[] jj_2_rtns = new JJCalls[31];
  static private boolean jj_rescan = false;
  static private int jj_gc = 0;

  public azizi_parser5(java.io.InputStream stream) {
    if (jj_initialized_once) {
      System.out.println("ERROR: Second call to constructor of static
parser.  You must");
```

```
      System.out.println("       either use ReInit() or set the JavaCC
option STATIC to false");
      System.out.println("       during parser generation.");
      throw new Error();
    }
    jj_initialized_once = true;
    jj_input_stream = new ASCII_CharStream(stream, 1, 1);
    token_source = new azizi_parser5TokenManager(jj_input_stream);
    token = new Token();
    jj_ntk = -1;
    jj_gen = 0;
    for (int i = 0; i < 0; i++) jj_la1[i] = -1;
    for (int i = 0; i < jj_2_rtns.length; i++) jj_2_rtns[i] = new
JJCalls();
  }

  static public void ReInit(java.io.InputStream stream) {
    jj_input_stream.ReInit(stream, 1, 1);
    token_source.ReInit(jj_input_stream);
    token = new Token();
    jj_ntk = -1;
    jj_gen = 0;
    for (int i = 0; i < 0; i++) jj_la1[i] = -1;
    for (int i = 0; i < jj_2_rtns.length; i++) jj_2_rtns[i] = new
JJCalls();
  }

  public azizi_parser5(java.io.Reader stream) {
    if (jj_initialized_once) {
      System.out.println("ERROR: Second call to constructor of static
parser.  You must");
      System.out.println("       either use ReInit() or set the JavaCC
option STATIC to false");
      System.out.println("       during parser generation.");
      throw new Error();
    }
    jj_initialized_once = true;
    jj_input_stream = new ASCII_CharStream(stream, 1, 1);
    token_source = new azizi_parser5TokenManager(jj_input_stream);
    token = new Token();
    jj_ntk = -1;
    jj_gen = 0;
    for (int i = 0; i < 0; i++) jj_la1[i] = -1;
    for (int i = 0; i < jj_2_rtns.length; i++) jj_2_rtns[i] = new
JJCalls();
  }

  static public void ReInit(java.io.Reader stream) {
    jj_input_stream.ReInit(stream, 1, 1);
    token_source.ReInit(jj_input_stream);
    token = new Token();
    jj_ntk = -1;
    jj_gen = 0;
    for (int i = 0; i < 0; i++) jj_la1[i] = -1;
    for (int i = 0; i < jj_2_rtns.length; i++) jj_2_rtns[i] = new
JJCalls();
  }

  public azizi_parser5(azizi_parser5TokenManager tm) {
    if (jj_initialized_once) {
      System.out.println("ERROR: Second call to constructor of static
parser.  You must");
```

```
      System.out.println("       either use ReInit() or set the JavaCC
option STATIC to false");
      System.out.println("       during parser generation.");
      throw new Error();
    }
    jj_initialized_once = true;
    token_source = tm;
    token = new Token();
    jj_ntk = -1;
    jj_gen = 0;
    for (int i = 0; i < 0; i++) jj_la1[i] = -1;
    for (int i = 0; i < jj_2_rtns.length; i++) jj_2_rtns[i] = new
JJCalls();
  }

  public void ReInit(azizi_parser5TokenManager tm) {
    token_source = tm;
    token = new Token();
    jj_ntk = -1;
    jj_gen = 0;
    for (int i = 0; i < 0; i++) jj_la1[i] = -1;
    for (int i = 0; i < jj_2_rtns.length; i++) jj_2_rtns[i] = new
JJCalls();
  }

  static final private Token jj_consume_token(int kind) throws
ParseException {
    Token oldToken;
    if ((oldToken = token).next != null) token = token.next;
    else token = token.next = token_source.getNextToken();
    jj_ntk = -1;
    if (token.kind == kind) {
      jj_gen++;
      if (++jj_gc > 100) {
        jj_gc = 0;
        for (int i = 0; i < jj_2_rtns.length; i++) {
          JJCalls c = jj_2_rtns[i];
          while (c != null) {
            if (c.gen < jj_gen) c.first = null;
            c = c.next;
          }
        }
      }
      return token;
    }
    token = oldToken;
    jj_kind = kind;
    throw generateParseException();
  }

  static final private boolean jj_scan_token(int kind) {
    if (jj_scanpos == jj_lastpos) {
      jj_la--;
      if (jj_scanpos.next == null) {
        jj_lastpos = jj_scanpos = jj_scanpos.next =
token_source.getNextToken();
      } else {
        jj_lastpos = jj_scanpos = jj_scanpos.next;
      }
    } else {
      jj_scanpos = jj_scanpos.next;
    }
    if (jj_rescan) {
```

```
      int i = 0; Token tok = token;
      while (tok != null && tok != jj_scanpos) { i++; tok = tok.next; }
      if (tok != null) jj_add_error_token(kind, i);
    }
    return (jj_scanpos.kind != kind);
  }

  static final public Token getNextToken() {
    if (token.next != null) token = token.next;
    else token = token.next = token_source.getNextToken();
    jj_ntk = -1;
    jj_gen++;
    return token;
  }

  static final public Token getToken(int index) {
    Token t = lookingAhead ? jj_scanpos : token;
    for (int i = 0; i < index; i++) {
      if (t.next != null) t = t.next;
      else t = t.next = token_source.getNextToken();
    }
    return t;
  }

  static final private int jj_ntk() {
    if ((jj_nt=token.next) == null)
      return (jj_ntk = (token.next=token_source.getNextToken()).kind);
    else
      return (jj_ntk = jj_nt.kind);
  }

  static private java.util.Vector jj_expentries = new java.util.Vector();
  static private int[] jj_expentry;
  static private int jj_kind = -1;
  static private int[] jj_lasttokens = new int[100];
  static private int jj_endpos;

  static private void jj_add_error_token(int kind, int pos) {
    if (pos >= 100) return;
    if (pos == jj_endpos + 1) {
      jj_lasttokens[jj_endpos++] = kind;
    } else if (jj_endpos != 0) {
      jj_expentry = new int[jj_endpos];
      for (int i = 0; i < jj_endpos; i++) {
        jj_expentry[i] = jj_lasttokens[i];
      }
      boolean exists = false;
      for (java.util.Enumeration enum = jj_expentries.elements();
enum.hasMoreElements();) {
        int[] oldentry = (int[])(enum.nextElement());
        if (oldentry.length == jj_expentry.length) {
          exists = true;
          for (int i = 0; i < jj_expentry.length; i++) {
            if (oldentry[i] != jj_expentry[i]) {
              exists = false;
              break;
            }
          }
          if (exists) break;
        }
      }
      if (!exists) jj_expentries.addElement(jj_expentry);
      if (pos != 0) jj_lasttokens[(jj_endpos = pos) - 1] = kind;
```

```
    }
  }

  static final public ParseException generateParseException() {
    jj_expentries.removeAllElements();
    boolean[] la1tokens = new boolean[43];
    for (int i = 0; i < 43; i++) {
      la1tokens[i] = false;
    }
    if (jj_kind >= 0) {
      la1tokens[jj_kind] = true;
      jj_kind = -1;
    }
    for (int i = 0; i < 0; i++) {
      if (jj_la1[i] == jj_gen) {
        for (int j = 0; j < 32; j++) {
          if ((jj_la1_0[i] & (1<<j)) != 0) {
            la1tokens[j] = true;
          }
          if ((jj_la1_1[i] & (1<<j)) != 0) {
            la1tokens[32+j] = true;
          }
        }
      }
    }
    for (int i = 0; i < 43; i++) {
      if (la1tokens[i]) {
        jj_expentry = new int[1];
        jj_expentry[0] = i;
        jj_expentries.addElement(jj_expentry);
      }
    }
    jj_endpos = 0;
    jj_rescan_token();
    jj_add_error_token(0, 0);
    int[][] exptokseq = new int[jj_expentries.size()][];
    for (int i = 0; i < jj_expentries.size(); i++) {
      exptokseq[i] = (int[])jj_expentries.elementAt(i);
    }
    return new ParseException(token, exptokseq, tokenImage);
  }

  static final public void enable_tracing() {
  }

  static final public void disable_tracing() {
  }

  static final private void jj_rescan_token() {
    jj_rescan = true;
    for (int i = 0; i < 31; i++) {
      JJCalls p = jj_2_rtns[i];
      do {
        if (p.gen > jj_gen) {
          jj_la = p.arg; jj_lastpos = jj_scanpos = p.first;
          switch (i) {
            case 0: jj_3_1(); break;
            case 1: jj_3_2(); break;
            case 2: jj_3_3(); break;
            case 3: jj_3_4(); break;
            case 4: jj_3_5(); break;
            case 5: jj_3_6(); break;
            case 6: jj_3_7(); break;
```

```
            case 7: jj_3_8(); break;
            case 8: jj_3_9(); break;
            case 9: jj_3_10(); break;
            case 10: jj_3_11(); break;
            case 11: jj_3_12(); break;
            case 12: jj_3_13(); break;
            case 13: jj_3_14(); break;
            case 14: jj_3_15(); break;
            case 15: jj_3_16(); break;
            case 16: jj_3_17(); break;
            case 17: jj_3_18(); break;
            case 18: jj_3_19(); break;
            case 19: jj_3_20(); break;
            case 20: jj_3_21(); break;
            case 21: jj_3_22(); break;
            case 22: jj_3_23(); break;
            case 23: jj_3_24(); break;
            case 24: jj_3_25(); break;
            case 25: jj_3_26(); break;
            case 26: jj_3_27(); break;
            case 27: jj_3_28(); break;
            case 28: jj_3_29(); break;
            case 29: jj_3_30(); break;
            case 30: jj_3_31(); break;
          }
        }
        p = p.next;
      } while (p != null);
    }
    jj_rescan = false;
  }

  static final private void jj_save(int index, int xla) {
    JJCalls p = jj_2_rtns[index];
    while (p.gen > jj_gen) {
      if (p.next == null) { p = p.next = new JJCalls(); break; }
      p = p.next;
    }
    p.gen = jj_gen + xla - jj_la; p.first = token; p.arg = xla;
  }

  static final class JJCalls {
    int gen;
    Token first;
    int arg;
    JJCalls next;
  }

}
```

Annexe 2
Exemple simple de simulation à l'aide des threads

```
//    Title : Coverification Project
//    Author      : Mostafa Azizi
//    Language: Java version "1.1.3"
//    Date  : Sept. 03, 97


import java.awt.*;

class SW_system extends Thread
{//start of SW_system
Object comware;
float output1, output2;

SW_system(Object co)
{
output1=0;
output2=0;
comware=co;
}

public void run()
{ //start run
while(true)
{
int i;
float o1=0, o2=0;
for(i=0; i<1000000; i++)
{
if ((i%2)==0) o1=o1 -2*((float) Math.random());
else o1++;
}
output1=o1;
synchronized(comware)
{comware.notify();}
synchronized(comware)
{
try
{
comware.wait();
}
catch(InterruptedException e) {System.out.println(e);}
}

for(i=0; i<1000000; i++)
{
if ((i%2)==0) o2=o1-1;
```

```
else o2=o1 + 1;
}
output2=o2;
synchronized(comware)
{comware.notify();}
synchronized(comware)
{
try
{
comware.wait();
}
catch(InterruptedException e) {System.out.println(e);}
}
}
} //end of run

public float result_output1()
{
return output1;
}

public float result_output2()
{
return output2;
}
}

class Properties extends Thread
{//start of Properties
Object comware;
SW_system design;
Properties(SW_system ss, Object co)
{
design=ss;
comware=co;
}
public void run()
{
while(true)
{
float o1, o2;
int i;
{
synchronized(comware)
{
try
{
comware.wait();
}
catch(InterruptedException e)
{
System.out.println(e);
}}
o1=design.result_output1();
if (o1>100)
System.out.print("Output1 is:"+o1+". It's in range.");
else System.out.print("**************************");
```

```
}
synchronized(comware)
{
comware.notify();
}
synchronized(comware)
{
try
{
comware.wait();
}
catch(InterruptedException e)
{
System.out.println(e);
}}
o2=design.result_output2();
if (o2<100)
System.out.print("  Output2 is:"+o2+".It's in range.");
else System.out.print("  **************************");
System.out.println("");
synchronized(comware)
{
comware.notify();
}
}
}
}//end of Properties

public class simulation
{ //start of simulation
public static void main(String[] arg)
{
Object comware=new Object();
SW_system design=new SW_system(comware);
Properties properties=new Properties(design, comware);
properties.start();
design.start();
}
}//end of simulation
```

Références

[1] M. Azizi, E. -M. Aboulhamid, and S. Tahar, "Sequential and Distributed Simulations using Java Threads", Proceedings of the International Conference on Parallel Computing in Electrical Engineering (PARELEC'2000), IEEE Computer Society, pp. 237-241, Trois-Rivières, Quebec, Canada, August 27-30, 2000 (ISBN 0-7695-0759-X)

[2] E. -M. Aboulhamid, M. Azizi, and X. Song, "Hardware Design and Verification Methods", Book chapter in the Encyclopedia of Computer Science and Technology, Vol. 42, pp. 67-91, A. Kent and J. G. Williams Eds., Marcel Dekker, Inc., New York, USA, 2000 (ISBN 0-8247-2295-7)

[3] M. Azizi, E. -M. Aboulhamid, and S. Tahar, "Properties Coverification in HW/SW Systems", Proceedings of the 2nd International Conference of Electronic Circuits and Systems (ECS'99), pp. 80-83, Bratislava, Slovakia, September 6-8, 1999 (http://www.ieee.org)

[4] M. Azizi, E. -M. Aboulhamid, and S. Tahar, "Multithreading-Based Coverification Technique of HW/SW Systems", Proceedings of the International Conference of Parallel and Distributed Processing Techniques and Applications (PDPTA'99), CSREA Press, Vol. IV, pp. 1999-2005, Las Vegas (Nevada), USA, June 27-30, 1999 (ISBN 1-892512-14-9)

[5] M. Azizi, O. Ait-Mohamed, and X. Song, "Cache Coherence Protocol Verification of a Multiprocessor System with Shared Memory", Proceedings of the 10th International Conference of Microelectronics (ICM'98), pp. 99-102, Mounastir, Tunisia, December 17-19, 1998 (http://www.ieee.org)

[6] M. Azizi, "Cosimulation using Eaglei Environment", Internal Report, NORTEL Corkstown, Nepean, Canada, March 1998 (team of Cosimulation & Emulation)

[7] M. Azizi, E. -M. Aboulhamid et I. Bennour, "Covérification des systèmes intégrés", Actes du Colloque International du Traitement d'Image et des Systèmes de Vision Artificielle (TISVA'98), pp. 234-240, Oujda, Maroc, avril 1998

[8] P. Chartier, « Des ordinateurs presque vivants », Québec Science, Vol. 38, No.7, pp. 26-31, Montréal, Canada, Avril 2000

[9] G. De Micheli, R. Gupta, "Hardware/Software Co-Design," Proc. IEEE, Mar. 1997, pp. 349-365

[10] R. W. Hartenstein, " Hardware Description Languages", Advances in CAD for VLSI, Volume 7, Elsevier Science Publishers B. V. (North-Holland), 1987

[11] E.D. Thomas, P.R. Moorby, The Verilog Hardware Description Language, second edition, Kluwer Academic Publishers, Norwell MA, 1994

[12] P.J. Ashenden, The Student's Guide to VHDL , Morgan Kauffmann, Publishers, January 1998

[13] T. Ben Ismail, A. Jerraya, "Synthesis Steps and Design Models for Codesign", Computer Vol. 28, No. 2, pp. 44-52, February 1995

[14] J. Bhasker, A VHDL Primer, Third Edition , Prentice Hall, Sept 1998
[15] J. Bhasker, A VHDL Synthesis Primer, Second Edition, Star Galaxy Publishing, August 1998
[16] K.C.C. Boeing, Digital Design and Modeling with VHDL and Synthesis, IEEE Computer Society Press, January 1997
[17] G. De Micheli, Synthesis and Optimization of Digital Circuits, McGraw-Hill Inc., 1994
[18] A.A. Jerraya, H. Ding, P. Kission, M. Rahmouni, Behavioral Synthesis and components Reuse with VHDL, Kluwer Publisher, November 1996
[19] J.P. Mermet, VHDL for Simulation, Synthesis and Formal Proofs of Hardware, Kluwer Publisher, May 1992
[20] D. Pellerin, D. Taylor, VHDL Made Easy, Prentice hall, 1996
[21] D. Perry, VHDL 3rd Edition , MacGraw-Hill, June 1998, 3nd edition
[22] A. Rushton, VHDL for Logic Synthesis, 2nd Edition, John Wiley & Sons, May 1998
[23] S. Sjoholm, L. Lindh, VHDL for Designers, Prentice Hall, 1996
[24] R. Sundar, Essential VHDL : RTL Synthesis Done Right , S & G Publishing, June 1998
[25] D.J. Smith, HDL Chip Design – A Practical Guide for Designing, Synthesizing and simulating ASICs and FPGAs using VHDL or Verilog, Doone Publications, 1997
[26] C.H. Roth, Jr, Digital Systems Design Using VHDL, PWS Publishing Company, 1998
[27] C. Liem, Retargetable Compilers for Embedded Core Processors : Methods and Experience in Industrial Applications, Kluwer Publisher, June 1997
[28] J. Rozenblit and K. Buchenrieder, Codesign computer SW/HW engineering, IEEE Press, 1994
[29] POLIS, " A framework for HW/SW codesign of embedded systems ", 1997
[30] O. Coudert, I.C. Madre, and C. Berthet, " Verifying temporal properties of sequential machines without building their date diagrams ", In Proceedings of the Workshop on Computer-Aided Verification (CAV 90)
[31] C. E. Leiserson and J. B. Saxe. " Retiming Synchronous Circuitry ", Algorithmica, Vol. 6, 1991.
[32] F. R. Boyer, E. M. Aboulhamid, Y. Savaria and I. E. Bennour, " Optimal design of synchronous circuits using software pipelining techniques ", IEEE International Conference on Computer Design, Austin, Texas, USA, 1998
[33] M. Abramovici, M. A. Breuer and A. D. Friedman, Digital Systems Testing and Testable Design, Computer Science Press, 1990
[34] H. Fujiwara, S. Toida, " The Complexity of Fault Detection Problems for Combinational Logic Circuits ", IEEE Trans. On Computers, vol. C-31, no. 6, pp. 555-560, 1982
[35] O.H. Ibarra, S.K. Sahni, " Polynomially Complete Fault Detection Problems", IEEE Trans. On Computers, vol. C-24, n0. 3, pp. 242-249, 1975
[36] T. Kropf, Formal Hardware Verification : Methods and Systems in Comparison, Springer, 1997

[37] J.P. Roth, W.G. Bouricius, P.R. Shneider, " Programmed Algorithms to Compute Tests to Detect and Distinguish Between Failures in Logic Circuits ", IEEE Trasn. On Electronic Computers, Vol. EC-16, No. 10, pp. 567-579, October, 1967

[38] P. Goel, " An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits ", IEEE Trans. On Computers, Vol C-30, No. 3, pp. 215-222, March, 1981

[39] H. Fujiwara, T. Shimono, " On the acceleration of Test Generation Algorithms ", IEEE Trans. On Computers, Vol. C-32, No. 12, pp. 1137-1144, December 1983.

[40] E.B. Eichelberger, T.W. Williams, " A Logic Design Sturcture for LSI Testing ", Proc. 14th Design Automation Conf., pp. 462-468, June, 1977

[41] JTAG, Technical Subcommittee of Joint Test Action Group, " Boundary Scan Architecture Standard Proposal ", version 2.0, March, 1988

[42] " Standard Test Access Port and Boundary-Scan Architecture " Sponsored by Test Technology Technical Committee of the IEEE Computer society, Document P1149.1/(D5), 1989

[43] M. Yoeli, " Formal Verification of Hardware Design ", IEEE Computer Society Press Tutorial, 1990

[44] J. Joyce, Multi-Level Verification of Microprocessor-Based Systems, Ph.D. Thesis, Computer Laboratory, Cambridge University, December 1989

[45] K. L. McMillan. Symbolic model checking, An approach to the state explosion problem. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA,1992

[46] R. Bryant, " Graph-Based Algorithms for Boolean Function Manipulation ", IEEE Transactions on Computers, Vol. C-35, No. 8, pp. 677-691, August 1986

[47] Z. Zhou, X. Song, F. Corella, E. Cerny and M. Langevin, " Description and verification of RTL designs using Multiway Decision Graphs ", In Proceedings of the Conference on Computer Hardware Description Languages and their applications (CHDL'95), Chiba-Japan, 1995

[48] Z. Zhou, Multiway Decision Graphs and Their Applications in Automatic Formal Verification of RTL Designs, Ph. D. Thesis, Université de Montréal, 1996

[49] H. Touati, H. Savoj, B. Lin, R. K. Brayton and A. Sangiovanni-Vincentelli, " Implicit State Enumeration of Finite State Machines using BDDs ", International Conference on Computer Aided Design, pp.130-133, 1990

[50] O. Coudert, I.C. Madre, and C. Berthet, " Verifying temporal properties of sequential machines without building their date diagrams ", In Proceedings of the Workshop on Computer-Aided Verification (CAV 90)

[51] F. Pong and M. Dubois, " Verification techniques for cache coherence protocols ", ACM Computing Surveys, Vol. 29, No. 1, March 1997.

[52] M. J. C. Gordon and T. F. Melham, Introduction to HOL : A theorem proving environment for higher order logic, Cambridge University Press, 1993.

[53] W. A. Hunt Jr., FM8501 : A verified microprocessor, Ph.D thesis, Technical Report ICSCA-CMP-47, University of Texas at Austin, 1985

[54] A. Gupta, " Formal Hardware Verification Methods : A survey ", Formal Methods in System Design, Vol. 1, pp. 151-238, 1992

[55] A. Cohn, " A proof of the Viper microprocessor : the first level ", VLSI Specification, Verification and Synthesis, Kluwer Academic Publishers, 1988

[56] W. Hunt, The Mechanical Verification of a Microprocessor Design, From HDL Description to Guaranteed Correct Circuit Designs, pp. 89-129, Elsevier Science Publishers B.V. (North-Holland), 1987

[57] P. Windley, The formal verification of generic interpreters, Ph. D. thesis, Division of Computer Science, University of California, Davis, July 1990

[58] M. Aagaard, M. Leeser, " Reasoning about Pipelines with Structural Hazards", Proceedings of Theorem Provers in Circuit Design, pp. 15-34, Bad Herrenalb, Germany, September 1994

[59] D. Cyrluk, " Microprocessor Verification in PVS : A Methodology and Simple Example ", Technical Report SRI-CSL-92-12, SRI Computer Science Laboratory, December 1993

[60] A. Roscoe, " OCCAM in the Specification and Verification of Microprocessors ", Philosophical Transactions of the Royal Society of London, Series A : Physical Sciences and Engineering, Vol.339, No. 1652, pp. 137-151, April 1992

[61] S. Bennett, " Enabling technologies for embedded system design tools ", EDA 1996.

[62] F. Balarin, H. Hsieh, A. Jurecska, L. Lavagno and A. Sangiovanni-Vincentelli, " Formal Verification of Embedded Systems based on CFSM Networks ", 33rd Design Automation Conference, ACM 1996.

[63] P. Chou and G. Borriello, " Software Scheduling in the cosynthesis of reactive real-time systems ", DAC 1994.

[64] P. Godefroid, " Model checking for programming languages using Verisoft ", Proceedings of the 24th ACM Symposium on Principles of Programming Languages, Paris, 1997.

[65] K. Varpaaniemi, " Efficient detection of deadlocks in Petri Nets ", Helsinki University of Technology, Digital Systems Laboratory, research reports, series A, No. 26, October 1993.

[66] M. Rauhamaa, " A comparative study of methods for efficient reachability analysis ", Helsinki University of Technology, Digital Systems Laboratory, research reports, series A, No. 14, September 1990.

[67] J. Rowson, " HW/SW cosimulation ", 31st Design Automation Conference (DAC'94), San Diego, June 1994

[68] Q. M. Tan, A. Petrenko and G. v. Bochmann, " deriving tests with fault coverage for specifications in the form of labeled transition systems ", Université de Montréal, Département d'Informatique et de Recherche Opérationnelle, publication 1073, June 1997.

[69] K. Varpaaniemi, " The sleep set method revised ", Helsinki University of Technology, Digital Systems Laboratory, 1994.

[70] S. E. Schulz, " Modeling issues for coverification ", Integrated System Design Magazine, 1996.

[71] B. Schnaider and E. Yogev, " SW development in a HW simulation environment ", 33rd Design Automation Conference, ACM 1996.

[72] E. Stoy and Z. Peng, " A design representation for HW/SW cosynthesis ", proceeding of Euromicro 94, Liverpool, September 1994.

[73] D. Becker, R. K. Singh and S. G. Tell, " An engineering environment for HW/SW cosimulation ", 29th ACM/IEEE Design Automation Conference (DAC'92), June 1992

[74] C. A. Valderrama, F. Nacabal, P. Paulain and A. A. Jerraya, " Automatic generation of interface for distributed G-VHDL cosimulation of embedded systems : an industrial experience ", proceedings of the 7th IEEE International Workshop on Rapid systems prototyping, June 1996

[75] K. Buchenrieder and C. Veith, " A prototyping Environment for control-oriented HW/SW systems using State-Charts, Activity-Charts and FPGA's ", Design Automation Conference, ACM 1994.

[76] S. Vercauteren, B. Lin and H. De Man, " Constructing application-specific heterogeneous embedded architectures from custom HW/SW applications ", Design Automation, ACM 1996.

[77] C. Bourhfir, Génération automatique de cas de test pour les systèmes modélisés par des machines à états finis communicantes, Thèse de doctorat, Université de Montréal, 1999

[78] R. Kurshan, V. Levin, M. Minea, D. Peled and H. Yenigun, " Verifying HW in its SW context ", proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD'97), San Jose, November 1997

[79] G. Berthelot, Transformations et analyse de réseaux de Petri, application aux protocoles, Thèse d'état, Université Pierre et Marie Curie, Paris 1983.

[80] S. Haddad, "Generalization of reduction theory to colored nets", Proceedings of the Ninth European Workshop on Application and Theory of Petri Nets, Vol. II, Venice 1988.

[81] W. D. Bishop and W. M. Loucks, " A heterogeneous environment for HW/SW cosimulation", Proceedings of the 30th Annual Simulation Symposium, p. 14-22, Atlanta, April 1997

[82] R. K. Gupta, C.N. Coelho Jr. and G. De Mecheli, " Synthesis and Simulation of Digital Systems containing interacting hardware and software components", 29th Design Automation Conference (DAC'92), June 1992

[83] J. Wilson, "HW/SW selected cycle solution ", International Workshop on HW/SW Codesign, 1994

[84] D.E. Thomas, J.K. Adams and H. Schmit, " A model and methodology for HW/SW codesign ", IEEE Design and Test of Computers, September 1993

[85] K. Ten Hagen and H. Meyer, "Timed and untimed HW/SW cosimulation : application and efficient implementation", International Workshop on HW/SW Codesign, October 1993

[86] Eagle Design Automation : VSP/LINK's Manual, 1995

[87] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki and B. Tabbara, " Hardware-Software Co-Design of Embedded Systems, the POLIS Approach ", Kluwer Academic Publishers, 1997

[88] Mentor Graphics Corporation, " CVE-Seamless ", http://www. meng.com/, 1998
[89] View Logic, " Eaglei ", http://www.Viewlogic .com/, 1998
[90] K.-C. Shumate, Understanding concurrency in Ada, McGraw-Hill Book Co., 1988
[91] D. Lea, Concurrent programming in Java, Addison-Wesley, 1996
[92] S. Oaks & H. Wong, Java Threads, O'Reilly & Associates, Inc., 1999
[93] J. Lowis & W. Loftus, Java : Software solutions, Addison-Wesley, 1998
[94] K. Arnold & J. Gosling, The Java™ programming language, Addison-Wesley, 1996
[95] T. Kuhn, W. Rosenstiel, and U. Kebschull, " Description and simulation of hardware/software systems with Java ", DAC'99, New Orleans, Louisiana, 1999
[96] R. Helaihel & K. Olukotun, "Java as a Specification Language for Hardware-Software Systems", Proceedings of the International Conference on Computer-Aided Design (ICCAD), pp. 690-697, November 1997.
[97] Craig A. Lindley, Digital Audio with Java, Prentice Hall Inc., 2000
[98] R. Brayton et al., " VIS : A system for verification and synthesis ", Technical report UCB/ERL M95, Electronics Research Laboratory, University of California, Berkeley, December 1995
[99] A. Burns and G. Davies, Concurrent Programming, International Computer Science Series, Addison-Wesley, 1993
[100] L. Lavagno, A. Sangiovanni-Vincentelli and H. Hsieh, Emdedded Systems Co-Design : Synthesis and Verification, Kluwer Academic Publishers, 1997
[101] E. Hunnell and M. Lyons, " Coverification Goes From Cutting Edge to Mainstream : DPLL Design Demonstrates the Viability of Today's Tools ", Electronic Design, June 22, 1998
[102] J. Misra, "Distributed Discrete-Event Simulation", ACM/Computing Surveys, Vol. 18, No. 1, 1986
[103] T. W. Albrecht, J. Notbauer and S. Rohringer, "HW/SW CoVerification Performance Estimation & Benchmark for a 24 Embedded RISC Core Design ", Proceedings of Design Automation Conference (DAC'98), pp. 808-811, San Francisco (California), 1998
[104] D. Lungeanu, and C.-J. R. Shi, "Distributed Simulation of VLSI Systems via Lookahead-Free Self-Adaptative Optimistic and Conservative Synchronization ", ICCAD'99, pp. 500-504, 1999
[105] P. Loewenstein and D. L. Dill, "Verification of a multiprocessor cache protocol using simulation relations and High-Order Logic", Proceedings of the 2nd International Workshop on Computer Aided Verification, pp. 302-311, June 1990
[106] M. Gordon and T. Melham, " Introduction to HOL : A theorem proving environment for higher order logic ", Cambridge University Press, 1993
[107] M. McMillan, Symbolic model checking, Kluwer Academic Publishers, Boston, Massachusetts, 1993
[108] E. M. Clarke and R. P. Kurshan, " Computer-aided verification ", IEEE Spectrum, January 1996

[109] D.L. Dill, " What's between simulation and formal verification? ", Design Automation Conference (DAC'98), San Francisco, June 1998
[110] VIS Home Page : http://www.berkely.edu/vis/
[111] R. Nalumaru, R. Ghughal, A. Mokkedem and G. Gopalakrishnan, " The 'test model checking' approach to the verification of formal memory models of multiprocessors ", Technical report UUCS-98-008, 1998
[112] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan and L. A. Ness, " Verification of the Futurebus+cache coherence protocol ", Proceedings of the 11th International Symposium on Computer Hardware Description Languages and their Applications (CHDL'93), Ottawa, April 1993.
[113] K. L. McMillan, Symbolic model checking : an approach to the state explosion problem, Ph. D. thesis, Carnegie Mellon University, 1992.
[114] K. L. McMillan and J. Schwalbe, " Formal verification of the gigamax cache consistency protocol ", In proceedings of the International Symposium on Shared Memory Multiprocessors, April 1991
[115] Xiao Shan Li and al, " Proving the correctness of the Interlock Mechanism in Processor Design ", CHARME'97, October 1997.
[116] N. Narasimhan, R. Kalyanaraman and R. Vemuri, " Validation of synthesized RTL designs using simulation and formal verification ", HLDVT Workshop, November 1996.
[117] D. A. Patterson and J. L. Hennessy, Computer Organization & Design, the Hardware/Software Interface, Morgan Kaufmann Publishers, Inc. 1994.
[118] F. Pong and M. Dubois, "A new approach for the verification of cache coherence protocols", IEEE Transactions on Parallel and Distributed Systems, Vol. 6, No. 8, pp. 773-787, August 1995
[119] F. Pong, Symbolic State Model; A new approach for the verification of cache coherence protocols, Ph. D. Dissertation, Dept. of Electrical Engineering-Systems, University of Southern California, August 1995
[120] A. J. Hu, D. L. Dill, A. J. Drexler and C. H. Yang, "Higher-level specification and verification with BDDs", Proceedings of the 4th International Workshop on Computer Aided Verification, pp. 82-95, July 1992
[121] FormalCheck v2.1, Lucent Technologies, 1998
[122] J. Archibald and J. L. Baer, " Cache coherence protocols : Evaluation using a multiprocessor simulation model", ACM Transactions on Computer Systems, Vol. 4, No. 4, pp. 273-298, November 1986
[123] G. V. Bochmann and C. A. Sunshine, " Formal Methods in communication protocol design ", IEEE Transactions on Communications, Vol. COM-28, No. 4, pp. 624-631, April 1980
[124] L. M. Censier and P. Feautrier, " A new solution to coherence problems in multicache systems ", IEEE Transactions on Computers, C-27(12), pp. 1112-1118, December 1978
[125] P. Stenstrom, " A survey of cache coherence schemes for multiprocessors ", IEEE Computer, Vol. 23, No. 6, pp. 12-24 , June 1990

[126] S. Graf, " Verification of a distributed cache memory by using abstractions ", Proceedings of the 6th International Workshop on Computer-Aided Verification, pp.207-219, 1994

[127] W. C. Yen, W. L. Yen and K. S. Fu, "Data coherence problem in a multicache system", IEEE Transactions on Computers, Vol. G-34, No. 1, January 1985

[128] M. C. Yuang, " Survey of protocol verification techniques based on finite state machine models", Proceedings of the Computer Networking Symposium, pp. 164-172, April 1988

[129] L. Alejandro, P. Elus, and Z. Peng, " Formal coverification of embedded systems using model checking ", Proceedings of EUROMICRO'2000, pp.106-113

Bibliographie

1. A. Dewey, Analysis and Design of Digital Systems with VHDL, PWS Publishing Company, August 1996
2. A. K. Nanda and L. N. Bhuyan, "A formal specification and verification technique for cache coherence protocols", Proceedings of the International Conference on Parallel Processing, pp. I 22-I 26, 1992
3. A. Pnueli, " The temporal logic of programs ", In Proceedings of the Eighth Annual Symposium on Foundations of Computer Science, IEEE, pp. 123-144, New York, 1994
4. C. M. Angelo, D. Verkest, L. Claesen and H. De Man, " On the comparison of HOL and Boyer-Moore for formal hardware verification ", In Journal Formal Methods in System Design, Vol. 2, pp. 45-72, 1992
5. D. Harel and B. Krishnamurthy, " Is There Hope for Linear-Time Fault Simulation? " Proc. Of the 17th Fault-Tolerant Computing Symp., pp. 28-33, 1987
6. E. Sternheim, R. Singh, Y. Trivedi, R. Madhaven and W. Stapleton, Digital Design and Synthesis with Verilog HDL, Automata Publishing Co., Cupertino, CA, 1993
7. E. Villar, and M. Veiga, " Embedded System Specification ", J.C. Lopez, R. Hermida, and W. Geisselhardt (eds.), Advanced Techniques for Embedded Systems Design and Test, pp. 1-30, Kluwer Academic Publishers, 1998
8. I.-E. Bennour, Estimation de la performance et Méthodes d'Allocation dans la Synthèse de systèmes numériques, Thèse de doctorat, Université de Montréal, 1996
9. IMEC, " OCAPI/RT, User Manual v0.81 ", http://www.imec.be, 1999
10. J. A. Hamilton Jr., D. A. Nash and U. W. Pooch, " Distributed Simulation ", CRC Press, Computer Engineering Series, 1997
11. J. Magee and J. Kramer, Concurrency : state models & Java programs, Wiley, Worldwide series in computer science, 1999
12. J. Saxe, S. Garland, J. Guttag, and J. Horning, " Using Transformations and Verification in Circuit Design ", Proceedings of 2nd Workshop on Designing Correct Circuits, Lyngby, Danmark, January 1992
13. J. Staunstrup and W. Wolf, Hardware/Software Codesign : Principles and Practice, Kluwer Publisher, October 1997
14. K. Skahill, J. Legenhausen, R. Wade, C. Wilner, B. Wilson, VHDL for Programmable Logic, Addison-Wesley, May 1996
15. M. S. Ben Romdhane, V. K. Madisetti and J. W. Hines, Quick-Turnaround ASIC Design in VHDL : Core-Based Behavioral Synthesis, Kluwer Publisher, June 1996
16. M. Srivas and M. Bickford, "Formal Verification of a Pipelined Microprocessor", IEEE Software, Vol. 7, No. 5, pp. 52-64, September 1990

17. Open Verilog International (OVI), Verilog HDL Language Reference Manual (LRM), 15466 Los Gatos Boulevard, Suite 109-071, Los Gatos, CA
18. S. Tahar and R. Kumar, “Implementational Issues for Verifying RISC- Pipeline Conflicts in HOL”, Higher Order Logic Theorem Proving and Its Applications, Lecture Notes in Computer Science 854, pp. 424-439, Springer Verlag, 1994
19. S. Tahar and R. Kumar, “ Implementing a Methodology for Formally Verifying RISC Processors in HOL”, Higher Order Logic Theorem Proving and Its Applications, Lecture Notes in Computer Science 780, pp. 281-294, Springer Verlag, 1994
20. Sun Microsystems Inc., The SPARC Architecture Manual, Sun Microsystems Inc., USA, Version 8, Part No. 800-1399-09, August 1989
21. University of California at Berkeley, “Ptolemy project”, http://ptolemy.eecs.berkeley.edu/index.htm, 1999
22. Y. Karkouri et E.-M. Aboulhamid, “ Complexité de test des circuits logiques ”, Technique et Science Informatiques, Vol. 9, no. 4, pp. 274-287, 1990