HAL
open science

Vérification de propriétés logico-temporelles de spécifications SystemC TLM

Luca Ferro

To cite this version:

Luca Ferro. Vérification de propriétés logico-temporelles de spécifications SystemC TLM. Autre. Université de Grenoble, 2011. Français. NNT : 2011GRENT032 . tel-00633069v2

HAL Id: tel-00633069

https://theses.hal.science/tel-00633069v2

Submitted on 24 Oct 2011

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

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

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Micro & Nano Électronique**

Arrêté ministériel : 7 août 2006

Présentée par

Luca FERRO

Thèse dirigée par **Laurence PIERRE**

préparée au sein du laboratoire **TIMA**
et de l'École Doctorale **Électronique, Électrotechnique, Automatique et Traitement du Signal**

Vérification de propriétés logico-temporelles de spécifications SystemC TLM

Thèse soutenue publiquement le **11 juillet 2011**,
devant le jury composé de :

M. El Mostapha ABOULHAMID
Professeur à l'Université de Montréal, Rapporteur
M. Philippe COUSSY
Maître de conférences à l'Université de Bretagne Sud, Examinateur
M. Laurent MAILLET-CONTOZ
CAD Manager à STMicroelectronics, Examinateur
M. Renaud PACALET
Directeur d'études à Télécom ParisTech, Président
Mme. Laurence PIERRE
Professeur à l'Université Joseph Fourier, Directrice de thèse
M. Lionel TORRES
Professeur à l'Université de Montpellier 2, Rapporteur

Remerciements

Le temps est venu de “rendre à César ce qui est à César”. Et c’est avec immense plaisir que je le fais, car la thèse est aussi un travail d’équipe, bien plus que nous ne l’imaginons.

Mon premier et plus grand remerciement s’adresse à Laurence Pierre, ma directrice de thèse. C’est la personne qui a réellement rendu possible le travail dont les résultats sont synthétisés dans le présent document. Outre sa présence durant les années de thèse, outre ses conseils (pas toujours faciles à suivre ! ☺) et son inébranlable enthousiasme, je remercie sa détermination à faire franchir le stade de “brouillon” à ce document.

Je remercie El Mostapha Aboulhamid et Lionel Torres d’avoir accepté de lire et d’évaluer ma thèse. Je leur suis également reconnaissant pour les précieuses remarques soulevées. Je remercie les trois autres membres du jury qui ont rendu possible et profitable la soutenance de cette thèse : merci à Renaud Pacalet d’avoir accepté de présider le jury, merci à Philippe Coussy et à Laurent Maillet-Contoz d’avoir accepté d’examiner mes travaux, ainsi que pour les fructueux échanges dans le cadre du projet SoCKET.

Un grand merci à toute l’équipe VDS, passée et présente : je remercie Dominique Borrione pour avoir toujours su faire du laboratoire un environnement stimulant, enrichissant et agréable ; merci à Katell Morin-Allory, en particulier pour son initiation à PSL bien avant le début de cette thèse ; merci à Eric Gascard surtout pour le précieux pointeur lors de la rédaction de ce document ; un grand merci à Yann Oddos et à Amr Helmy, deux bons amis et pour moi deux icones de VDS ; je remercie Renaud Clavel car en sa compagnie on a l’impression de ne jamais arrêter d’apprendre ; merci à Florent Ouchet (en particulier pour l’indispensable SVN !), à Zeineb Bel Hadj Amor, à Laila Damri, à Georgios Tsiligiannis, à Jérôme Sester. J’aurais presque envie de dire “merci à la famille VDS” ☺. Un merci spécial à Alexandre Porcher et à Franck Paugnat pour leur aide lors de la préparation de la soutenance.

Je tiens à remercier Yves Ledru et Lydie du Bousquet pour la profitable collaboration en matière de campagnes de tests. Merci à Cédric Koch-Hofer pour son initiation à SystemC et pour toutes les ressources mises à ma disposition. Je remercie Patrice Gerin de m’avoir fourni un cas d’étude intéressant. Je remercie Vincent Lefftz, Jérôme Lachaize et Ahmad Berjaoui pour l’active collaboration, pour tous leurs conseils et pour le dernier cas d’étude traité. Merci à Alejandro Chagoya pour son aide avec les plates-formes du CIME Nanotech

et pour son immuable gentillesse. Merci à Jean-François Méhaut pour son assistance en matière de parallélisation.

Un merci à tout le personnel technique et administratif de TIMA pour son amabilité et efficacité.

Je remercie Anthony Coadou et Nader Salman. Cette thèse s'est aussi nourrie de leur soutien et de leur extraordinaire amitié.

Je veux remercier Davide Deghelli, auquel je me sens lié comme un frère.

Merci à la fée Morgane pour m'avoir montré que la magie existe encore.

Enfin, une profonde gratitude à mes parents, Dario Ferro et Antonella Bianchi, pour tout leur inestimable amour, et pour avoir été toujours présents comme seulement un père et une mère savent faire. Je remercie la totalité de ma famille, en particulier mes grands-parents Maria, Mario, Carla et Cicin, pour m'avoir toujours tant donné.

...Ah, j'oubliais presque : choucroute !

Table des matières

Glossaire **xi**

1 Introduction **1**
Préambule . . . 1
1.1 Les limitations des démarches de conception classiques . . . 2
1.2 Un besoin d'abstraction : vers SystemC et TLM . . . 3
1.3 Vérification de SoCs . . . 4
1.3.1 Approches de vérification . . . 5
1.3.2 Motivations et contribution . . . 5
1.3.2.1 Surveillance d'assertions . . . 5
1.3.2.2 Organisation du document . . . 6

2 Contexte **9**
Introduction . . . 9
2.1 Modélisation de circuits avec SystemC TLM . . . 9
2.1.1 Principaux composants SystemC . . . 10
2.1.1.1 Modules . . . 10
2.1.1.2 Ports . . . 10
2.1.1.3 Interfaces . . . 11
2.1.1.4 Canaux . . . 11
2.1.1.5 Exemple . . . 12
2.1.2 La simulation . . . 13
2.1.2.1 Processus . . . 14
2.1.2.2 Événements et synchronisation . . . 14
2.1.2.3 Sémantique de simulation . . . 15
2.1.3 La bibliothèque TLM . . . 16
2.1.3.1 Niveaux d'abstraction . . . 17
2.1.3.2 Architecture TLM typique . . . 18
2.1.3.3 Styles de modélisation . . . 20
2.1.3.4 Communications bloquantes et non-bloquantes . . . 22
2.1.3.5 Données des transactions . . . 22
2.1.3.6 Découplage temporel . . . 23
2.1.3.7 Autres interfaces . . . 23
2.2 Vérification par assertions . . . 23
2.2.1 Spécification et vérification du comportement attendu . . . 23

2.2.1.1 Vérification formelle 24
2.2.1.2 Vérification durant la simulation 25
2.2.2 Tour d'horizon des formalismes existants 25
2.2.2.1 Logiques temporelles 26
2.2.2.2 SVA 29
2.2.2.3 OVL 32
2.2.2.4 Autres langages et méthodes 33
2.2.3 PSL : *Property Specification Language* 34
2.2.3.1 Un aperçu du langage 34
2.2.3.2 SEREs 36
2.2.3.3 Sémantique 37
2.2.3.4 Niveaux de satisfaction d'une formule 40
2.3 Bilan 41

3 Travaux sur la vérification de descriptions SystemC 43
Introduction 43
3.1 Approches statiques 43
3.2 Approches dynamiques 48
3.2.1 Premières contributions 48
3.2.2 Travaux connexes aux nôtres 51
3.2.3 Outils commerciaux 56
3.2.3.1 Vers la validation par ABV 56
3.2.3.2 Retour d'expériences 57
3.3 Bilan 58

4 Surveillance de propriétés au niveau transactionnel 61
Introduction 61
4.1 Construction de moniteurs de surveillance 62
4.1.1 Une approche modulaire au niveau RTL prouvée correcte 62
4.1.2 Construction des moniteurs orientés TLM 65
4.1.2.1 Implémentation de la bibliothèque de moniteurs C++ . . 65
4.1.2.2 Interconnexion des composants C++ élémentaires 66
4.2 ABV au niveau TLM 68
4.2.1 Définition de trace 68
4.2.1.1 Le problème de l'échantillonnage 68
4.2.1.2 À propos des SEREs et de l'opérateur *next* 70
4.2.2 Modèle de surveillance 71
4.2.3 Couche booléenne et choix syntaxiques 73
4.2.4 Exemple : FIFO avec arbitre 75
4.2.5 Approches alternatives 77

5 Propriétés avec variables auxiliaires 81
Introduction 81
5.1 Couche modélisation et variables globales 82
5.1.1 Une sémantique opérationnelle pour PSL 83
5.1.1.1 *Weak* PSL 83
5.1.2 Proposition d'une extension pour la sémantique opérationnelle . . . 85
5.1.2.1 Notations 85

5.1.2.2 Règles sémantiques et satisfaction d'une formule 86
5.1.2.3 Sémantique pour les états d'une formule 88
5.1.2.4 Adaptation au niveau transactionnel 89
5.1.3 Mise en œuvre pour les variables globales 90
5.2 Propriétés réentrantes et variables locales 94
5.2.1 Réentrance avec simple mémorisation 95
5.2.2 Réentrance avec mise à jour des variables locales 97
5.2.3 Mise en œuvre pour les variables locales 100
5.2.3.1 Principes d'implémentation 100
5.2.3.2 Application à l'exemple du *Packet Switch* 103
5.2.4 Spécifications temporelles avec variables 104
5.2.4.1 Vérification avec instances multiples 105
5.2.4.2 Sémantiques avec variables locales 106
Bilan 109

6 Mise en oeuvre 111
Introduction 111
6.1 ISIS 112
6.1.1 Flot d'instrumentation de designs SystemC pour l'ABV dynamique 112
6.1.2 Étapes fondamentales de l'instrumentation 114
6.2 Travaux connexes 118
6.2.1 Parallélisation 118
6.2.2 Campagnes de test 120
6.2.2.1 Génération de tests pour des spécifications de haut niveau 120
6.2.2.2 Test combinatoire et ABV 121
6.2.3 Application à l'exemple du DMA 122

7 Expérimentations 127
Introduction 127
7.1 Études de cas 127
7.1.1 Communication par FIFO 127
7.1.1.1 Échanges simples 127
7.1.1.2 Échanges avec arbitre 128
7.1.2 Communication par canal défectueux 130
7.1.3 Contrôleur DMA 131
7.1.4 Packet Switch 133
7.1.5 Plate-forme de décodage Motion-JPEG 136
7.1.6 Modèle transactionnel d'un SoC de traitement de données bord pour charge utile spatiale 138
7.2 Analyse des performances 144
7.2.1 Temps CPU et synthèse 145
7.2.2 Flexibilité du modèle 154

8 Conclusion et perspectives 157
8.1 Contributions 157
8.2 Travaux futurs 159

A *Makefile* type pour les designs instrumentés **163**
A.1 *Makefile* original 163
A.2 *Makefile* pour le DUV instrumenté 164

B Packet Switch : extrait d'une trace de simulation avec P_{12} **167**

C Modules d'ISIS **171**

Bibliographie **173**

Table des figures

1.1 Flot classique de conception d'un SoC (source : [GCMC+05]) 2
1.2 Coûts totaux d'un IC pour les différentes activités à différents niveaux technologiques (source : [Avi10]) 3
1.3 Moniteur de surveillance 6

2.1 Visibilite des communications par l'utilisateur 10
2.2 Ports de communication 11
2.3 Exemple élémentaire d'architecture en SystemC 12
2.4 Port et surcharge de l'operateur *flèche* 14
2.5 Noyau de simulation SystemC (source : [BD04]) 16
2.6 Éléments de la bibliothèque SystemC TLM (source : [OSC09]) 17
2.7 Modèle d'architecture TLM en trois couches 18
2.8 Contrôleur DMA 19
2.9 Exemple en style LT 21
2.10 Exemple en style AT 22
2.11 Principe de vérification formelle 24
2.12 Principe de vérification dynamique 25
2.13 Exemple de structure de Kripke et arbre de calcul associé 27
2.14 Exemple d'arbre qui satisfait la propriété 1 29
2.15 Trace de simulation satisfaisant la séquence S_1 31
2.16 Exemple de coopération des quatre couches 35
2.17 Exemple de trace qui satisfait P_1 36
2.18 Exemple de trace qui satisfait P_2 36
2.19 Deux traces satisfaisant la propriété P_5 37
2.20 Une trace qui viole la propriété P_5 37
2.21 Exemple de trace satisfaisant la propriété P_3 39
2.22 Exemple de trace satisfaisant la propriété P_4 40

3.1 Réseau PRES+ pour le programme "xy" (source : [CEP00]) 46
3.2 Machine à états finie associée à la propriété P_1 49
3.3 Exemples d'automates AR (source : [RHKR01]) 50
3.4 Flot de génération proposé dans [Lah06] 53
3.5 Relations entre les transactions (source : [EEH+06b]) 54
3.6 Exemple de systéme avec *proxies* 55
3.7 Structure interne d'un moniteur (source : [EEH+07]) 56

4.1 Interface et structure d'un moniteur élémentaire 63
4.2 Arbre syntaxique pour la formule PSL P_1 64
4.3 Moniteur RTL composite pour la formule P_1 64
4.4 Mécanisme d'évaluation des moniteurs élémentaires pour P_1 67
4.5 Cas particulier de l'opérateur *et* 67
4.6 Exemple de comportement d'un système séquentiel synchrone 68
4.7 Extrait de simulation du DMA sous la forme de diagramme de séquence UML simplifié 69
4.8 Modèle d'observation - Diagramme de classes UML 71
4.9 Mécanisme d'observation 73
4.10 Modèle de communication par FIFO avec arbitre 75

5.1 Exemple de trace de simulation pour $A1$ et $A2$ 91
5.2 Structure du *wrapper* et évaluation des moniteurs 92
5.3 *Packet Switch* de la bibliothèque SystemC 94
5.4 Opérateur *new* et instances d'une sous-propriété 100
5.5 Transmission des données aux instances d'une même sous-formule 102
5.6 Mécanisme global de surveillance 103
5.7 Exemple de trace de simulation pour le *Packet Switch* 104

6.1 Flot d'ABV avec instrumentation du DUV à l'aide d'ISIS 112
6.2 Choix de la méthode observée `write` du port initiateur via l'interface graphique d'ISIS 116
6.3 Saisie de la propriété à l'aide de l'interface graphique d'ISIS 116
6.4 Liaison du moniteur au DUV via l'interface graphique d'ISIS 117
6.5 Infrastructure avec plusieurs processus 119
6.6 Test combinatoire avec TOBIAS 122

7.1 Obtention de la trace d'évaluation de la propriété P_4 129
7.2 Modèle de communication par canal défectueux 130
7.3 Plate-forme avec contrôleur DMA 132
7.4 Utilisations imbriquées du *new* et sous-instances multiples 136
7.5 Plate-forme de décodage Motion-JPEG 137
7.6 Simulation instrumentée de la plate-forme MJPEG comportant une version défectueuse du Crossbar 138
7.7 Diagramme de l'architecture du SoC 139
7.8 Plate-forme SystemC TLM fonctionnelle avec modélisation des contraintes temporelles 139
7.9 FIFO, échanges simples : temps CPU pour les propriétés P_1 et P_2 145
7.10 FIFO, échanges simples : nombre d'activations des moniteurs pour les propriétés P_1 et P_2 145
7.11 FIFO, échanges avec arbitre : temps CPU pour les propriétés P_3 et P_4 . . . 146
7.12 FIFO, échanges avec arbitre : nombre d'activations des moniteurs pour les propriétés P_3 et P_4 147
7.13 Canal défectueux : temps CPU pour les propriétés P_5, P_6 et P_f 147
7.14 Canal défectueux : nombre d'activations des moniteurs pour les propriétés P_5, P_6 et P_f 148
7.15 DMA : temps CPU pour les propriétés P_7, P_8 et P_9 148

7.16 DMA : temps CPU pour les trois propriétés P_7, P_8 et P_9 simultanément . . 149
7.17 DMA : nombre d'activations des moniteurs pour les propriétés P_7, P_8 et P_9 149
7.18 Packet Switch : temps CPU pour les propriétés P_{10} à P_{13} 150
7.19 Packet Switch : nombre d'activations des moniteurs pour les propriétés P_{10} à P_{13} 151
7.20 Plate-forme MJPEG : temps CPU pour les propriétés P_{14} et P_{15} 151
7.21 Plate-forme MJPEG : nombre d'activations des moniteurs pour les propriétés P_{14} et P_{15} 152
7.22 Plate-forme de traitement vidéo pour la télémesure : temps CPU pour les propriétés P_{16} et P_{17} 152
7.23 Plate-forme de traitement vidéo pour la télémesure : temps CPU pour les propriétés P_{18} v1 et P_{18} v2 153
7.24 Plate-forme de traitement vidéo pour la télémesure : nombre d'activations des moniteurs pour les propriétés P_{16} et P_{17} 153
7.25 Plate-forme de traitement vidéo pour la télémesure : nombre d'activations des moniteurs pour les propriétés P_{18} v1 et P_{18} v2 153

8.1 Trace de simulation avec communications "simultanées" 161

C.1 Modules du flot d'ISIS 172

Glossaire

A

ABV *Assertion-Based Verification*
API *Application Programming Interface*
ASIP *Application-Specific Instruction-set Processor*
ASM *Abstract State Machine*
AT *Approximately Timed*
BDD *Binary-Decision Diagram*

C

CA *Cycle Accurate*
CTL *Computational Tree Logic*

D

DBI *Dynamic Binary Intrumentation*
DMA *Direct Memory Access*
DUV *Design Under Verification*

E

EU *Execution Unit*

F

FPGA *Field-Programmable Gate Array*
FSM *Finite State Machine*

H

HDL *Hardware Description Language*

I

IC *Integrated Circuit*
IEEE *Institute of Electrical and Electronics Engineers*
IMC *Interface Method Call*
IP *Intellectual Property*
IPC *Inter-Process Communication*

L

LIG *Laboratoire d'Informatique de Grenoble*
LT *Loosely Timed*
LTL *Linear Temporal Logic*

LUT *Look-Up Table*
MPI *Message Passing Interface*
MPICH *Message Passing Interface Chameleon*

N
NSCa *Native SystemC assertion (mechanism)*

O
OCB *On-Chip Bus*
OSCI *Open SystemC Initiative*
OVL *Open Verification Library*
OVM *Open Verification Methodology*

P
PSL *Property Specification Language*
PV *Programmer's View*
PVS *Prototype Verification System*
PVT *Programmer's View plus Timing*

R
RTL *Register Transfer Level*

S
SAT *SATisfiability problem*
SCV *SystemC Verification library*
SERE *Sequential Extended Regular Expression*
SMV *Symbolic Model Verifier*
SoC *System-on-Chip*
SVA *SystemVerilog Assertions*

T
TA *Transaction Accurate*
TIMA *Techniques de l'Informatique et de la Microéléctronique pour l'Architecture des systèmes intégrés*
TLM *Transaction Level Modeling*
TTP *Timed TLM Protocol*

U
UART *Universal Asynchronous Receiver/Transmitter*
UML *Unified Modeling Language*
UVM *Universal Verification Methodology*
VCD *Value Change Dump*
VHDL *Very high speed integrated circuit Hardware Description Language*
vunit *Verification Unit*

W
WPL *Weak Property Language*

X
XML *eXtensible Markup Languag*

Chapitre 1

Introduction

Electronics Magazine, 1965 [Moo65] : Gordon E. Moore[1] constate que la complexité des semi-conducteurs double tous les ans à coût constant depuis 1959 et il prédit la poursuite de cette croissance. Il affirme que le nombre de transistors incorporés dans un IC (*Integrated Circuit*) doublera tous les ans. En 1975, à la lumière des données plus récentes et en incluant un plus vaste assortiment de designs de microprocesseurs, il réévalue sa prédiction : il déclare que le nombre de transistors des microprocesseurs (et non plus de simples circuits intégrés moins complexes) sur une puce de silicium doublera tous les deux ans. Au cours des dernières années cette croissance a connu seulement un léger ralentissement.

Au-delà de la formidable évolution en termes de complexité du circuit électronique en soi, son adoption et sa diffusion ont connu, au fil des dernières années, une explosion dans un très grand nombre de domaines distincts. En particulier, la forte miniaturisation de ces circuits a favorisé considérablement leur emploi dans des domaines tels que la télécommunication, le médical ou encore les transports [Hel04]. La notion de SoC (*System-on-Chip*), concept selon lequel plusieurs composants électroniques seraient rassemblés sur une seule puce afin de constituer un système complet, est devenue un élément incontournable dans de nombreux produits. Rendu possible, entre autres, par la capacité d'intégrer de plus en plus de transistors sur une seule puce, ce concept très prometteur est désormais au cœur d'articles tels que les appareils photo ou les téléphones portables.

Dans ce contexte, la complexité possède, dans le meilleur des cas, deux facettes. En effet, aux difficultés liées au simple micro-composant s'ajoutent les problématiques liées à l'hétérogénéité et à l'assemblage du système complet. Un système sur puce peut incorporer une combinaison de composants aux fonctionnalités très différentes, comme microprocesseurs et mémoires, contrôleurs pour l'audiovisuel, modems, contrôleurs graphiques 2D et 3D, unités de DSP dédiées, etc. La complexité d'un *smartphone* de dernière génération dépasse amplement celle d'un ordinateur complet d'autrefois. S'assurer du bon fonctionnement de chaque composant, et du système complet, est une tâche primordiale, à la fois pour la sécurité des utilisateurs (transports, avionique, médical...) et pour la bonne réussite commerciale du produit final (multimédia, téléphonie...). Malheureusement, il s'agit d'une tâche épineuse.

[1] Gordon E. Moore était directeur de la section R&D de Fairchild Semiconductor's en 1965.

Le flot de conception d'un design complet doit être rigoureux mais surtout adapté au design visé. Pour l'élaboration d'un SoC complet, les entreprises ne peuvent pas se contenter des démarches adoptées lors de la conception d'un seul composant.

1.1 Les limitations des démarches de conception classiques

Le flot de conception d'un design comporte un processus long et rigoureux [GCMC+05]. Ce processus annexe des activités telles que la spécification et la vérification, et s'articule selon différentes étapes. Par le passé, une approche de conception classique comportait deux activités séparées et malencontreusement indépendantes : le développement du matériel et celui du logiciel. Le principe global de ce flot est résumé par la figure 1.1.

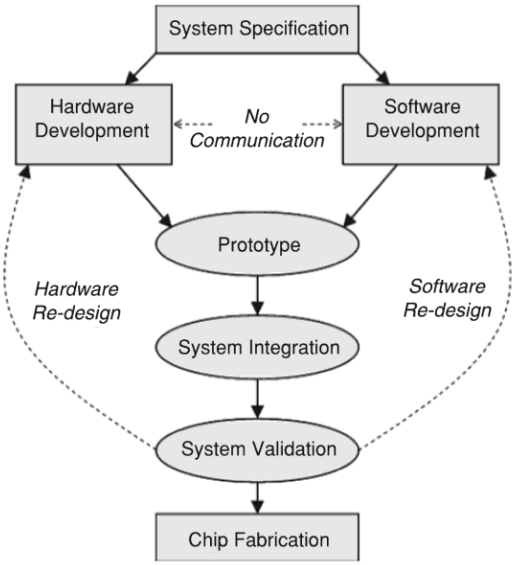

FIGURE 1.1 – Flot classique de conception d'un SoC (source : [GCMC+05])

Dans ce flot, les modèles du matériel, décrits à l'aide de langages tels que VHDL [VHD02] ou Verilog [VER01] au niveau transfert de registres, sont vérifiés et synthétisés sans la partie logicielle. Cette dernière est conçue de façon complètement indépendante. Pour pouvoir tester le logiciel il faut disposer, dans le meilleur des cas, du code RTL (*Register Transfer Level*) et d'un émulateur ou d'un système de prototypage basé sur FPGA (*Field-Programmable Gate Array*). Ceci comporte des coûts additionnels, un retard conséquent dans la validation du logiciel et une découverte tardive des problèmes éventuels. Durant les années 90, des techniques de "co-design" et "co-vérification" étaient employées pour simuler de façon concurrente les description matérielles RTL et le logiciel embarqué [Mar97]. Elles ne permettaient toutefois qu'une analyse limitée à cause de la lenteur de la simulation au niveau transfert de registres.

Avec la croissance incontrôlable de la complexité des SoC, en particulier de celle liée aux applications logicielles, les techniques classiques de co-vérification ont atteint leurs limites depuis désormais plusieurs années [Mar98]. Un SoC moderne intègre un grand nombre de blocs logiciels et matériels, incluant plusieurs processeurs de nature différente, des bus et d'autres éléments d'interconnexion complexes, mémoires et périphériques aux

fonctions hétérogènes. Le système final accomplit des tâches de plus en plus sophistiquées. Un flot classique comme celui de la figure 1.1 n'est plus en mesure de faire face à une telle complexité.

Par ailleurs, la pression due au temps de mise sur le marché pour certaines catégories de produits[2] n'admet aucun ralentissement dans le flot de conception. En même temps, les coûts liés au développement du logiciel et à son intégration et vérification avec le matériel occupent de loin la plus grande portion du coût total de conception du SoC [Avi10]. Cet aspect devient de plus en plus significatif avec la progression de la technologie de production, comme schématisé par le graphe de la figure 1.2.

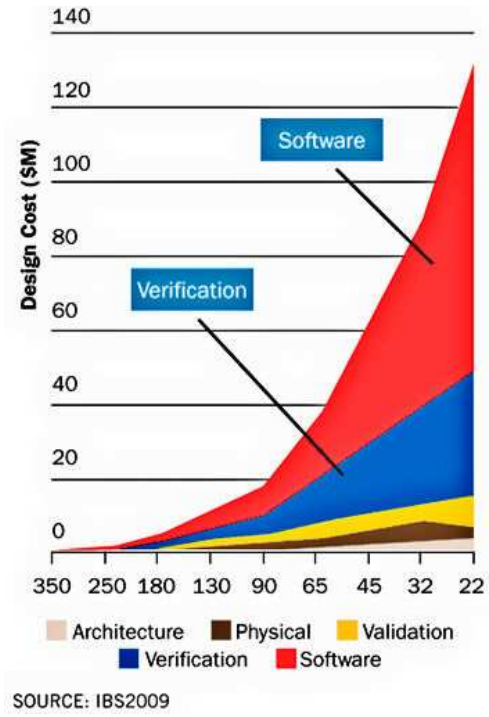

FIGURE 1.2 – Coûts totaux d'un IC pour les différentes activités à différents niveaux technologiques (source : [Avi10])

1.2 Un besoin d'abstraction : vers SystemC et TLM

Depuis une dizaine d'années environ, des plates-formes telles que la catégorie OMAP (Texas Instruments) et la série Nexperia (NXP Semiconductors, originairement Philips) ont émergé dans le monde des SoC [Avi10]. L'une des idées de base était de pouvoir réexploiter le même squelette d'architecture et les mêmes blocs matériels propriétaires, ou IPs (*Intellectual Properties*), dans plusieurs produits et applications. Si la réutilisation des IPs est un élément fondamental pour pouvoir assembler rapidement un SoC complet, il ne fournit toutefois pas une solution suffisante : les efforts pour pouvoir choisir convenablement et intégrer correctement les blocs du système ne sont pas négligeables [Smi05]. Si les premières descriptions du SoC sont RTL, le système complet s'avère toujours très complexe à concevoir et à analyser. Pour toutes les raisons mentionnées auparavant, le

[2] Un produit commercialisé plus tôt pourra bénéficier d'une plus grande part de marché.

niveau transfert de registres ne constitue donc plus un point de départ satisfaisant dans le flot.

Le *system level design*, ou conception au niveau système, commence alors à émerger comme possible extension du point d'entrée RTL dans le flot du SoC. Chez ST Microelectronics, les premiers efforts dans cette direction visaient des modèles décrits en C ou C++ précis au niveau cycle d'horloge [GCMC+05]. Cependant, la complexité de ces modèles était encore trop proche de celle du code RTL, et leur vitesse de simulation très détériorée. Il fallait à tout prix pouvoir atteindre un niveau d'abstraction suffisant pour pouvoir décrire et simuler le design bien plus rapidement, sans toutefois perdre trop de précision lors de la caractérisation du matériel.

La conception au niveau système paraît constituer la bonne direction pour permettre une meilleure et plus rapide interaction entre les ingénieurs du logiciel et du matériel, entre autres avec la perspective de réduire le temps de mise sur le marché. De plus, cette approche de conception permet une analyse d'architecture plus aisée et une première étude précoce du comportement du SoC [EEH+07].

Dans cette optique, SystemC, bibliothèque étendant le langage C++, revêt un rôle prépondérant. Née d'une première proposition d'un ensemble de classes pour la description de matériel et pourvue d'un noyau de simulation *open source*, la bibliothèque connaît ses premières versions à la fin des années 90. Dès sa première proposition par Coware et Synopsys, elle n'a pas cessé d'évoluer vers un langage[3] de plus en plus riche, particulièrement prisé pour sa possibilité de décrire le design à un niveau bien plus abstrait que RTL. Depuis ses premières versions, l'un des aspects révélateurs de cette tendance à l'abstraction était la présence de canaux de communication ouvrant le chemin vers une abstraction des communications. Le niveau de modélisation transactionnel (TLM, pour *Transaction Level Modeling*) s'appuie sur des langages tels que SystemC. Il est même étroitement associé à ce dernier. Avec comme concept central la séparation de l'aspect communication de l'aspect calcul à l'intérieur du système, TLM permet des modélisations à plus haut niveau par rapport à RTL. Son adoption comme point de départ, en amont du niveau transfert de registres dans le flot, semble présenter les caractéristiques nécessaires pour pallier, au moins en partie, les faiblesses typiques du processus de conception d'un SoC.

1.3 Vérification de SoCs

Plus rapide à concevoir et à simuler, un modèle transactionnel ne permet pas uniquement un développement conjoint du logiciel et du matériel, ni se limite à l'étude architecturale. Il offre aussi un atout à la vérification du design. Cette étape recouvre généralement une très grande portion du flot de conception du SoC : elle seule peut facilement atteindre, et souvent dépasser amplement, 50% des efforts totaux [Dub05, DG05, Bal06, Fos09]. En même temps, si le modèle TLM constitue le nouveau point d'entrée du sous-flot de développement, il représentera certainement un modèle de référence pour la suite [EEH+06b]. Par conséquent, s'assurer du bon fonctionnement de ce modèle est un impératif.

Le 31 janvier 2011, dans un communiqué de presse en ligne [Cor11], Intel Corporation relate la découverte d'une anomalie de conception sur le chipset Intel série 6 (nom de

[3] SystemC n'est pas un langage en soi, mais il est couramment qualifié comme tel.

code *Cougar Point*), récemment sorti. Le coût total de réparation et de remplacement du matériel et des configurations affectés qui sont sur le marché est estimé à 700 millions USD. Malgré les engagements en matière de vérification, même dans une compagnie telle qu'Intel, aucune solution capable d'assurer une identification intégrale des erreurs n'existe à l'heure actuelle. Si les outils de conception ont connu des progrès significatifs dans les années passées, les outils de vérification n'ont malheureusement pas vécu la même développement [Var07]. Ce besoin constant pour des approches de vérification nouvelles, capables de s'adapter à l'évolution pressante dans le monde de l'électronique, est à l'origine des motivations qui ont inspiré cette thèse.

1.3.1 Approches de vérification

Le terme "vérification" peut être interprété de manière différente selon le milieu, le contexte, ou l'état dans le flot de conception du SoC [Dub05, Das06, Var07]. La vérification s'effectue sur une description plus ou moins abstraite du circuit, mais avant l'étape de fabrication. Dans le cadre de cette thèse, nous nous intéressons à la vérification qualifiée de fonctionnelle dans [Dub05], et qui consiste à s'assurer qu'un circuit, ou modèle de système, se comporte en accord avec un ensemble de spécifications [FKL03]. Il existe plusieurs approches pour vérifier les fonctionnalités d'un design, mais elles peuvent être groupées selon deux catégories principales : statiques et dynamiques [Das06]. Dans le premier cas, l'objectif est de vérifier tous les comportements possibles du design. Cela est dispendieux mais exhaustif. La deuxième catégorie s'appuie essentiellement sur la simulation. Elle ne couvre pas tous les cas possibles, mais elle est beaucoup plus applicable, en particulier pour des designs complexes. Nous approfondirons cette comparaison dans la section 2.2.1.

Dans les deux cas, il est indispensable de disposer d'une spécification du *comportement attendu* du design. Dans les deux cas, cette spécification doit être *précise* afin de pouvoir être vérifiée. L'ambiguïté liée aux langages naturels a été source de plusieurs erreurs dans les designs.

Une *assertion* est la description *précise* d'un *comportement attendu* du design [FKL03], et s'exprime à l'aide de langages spécifiques. L'ABV (*Assertion-Based Verification*) est une démarche de vérification qui consiste à utiliser un ensemble d'assertions lors de l'analyse du comportement du design. Il s'agit d'une approche dont la valeur est désormais reconnue dans l'industrie depuis plusieurs années [Mal02, Das06], en particulier au niveau transfert de registres. Toutefois, bien que des travaux aient été entrepris concernant l'ABV à des niveaux plus abstraits, notamment TLM, aucune solution utilisable industriellement n'a encore été complètement retenue.

1.3.2 Motivations et contribution

Les travaux menés dans le cadre de cette thèse se focalisent sur l'ABV dynamique pour des descriptions SystemC au niveau transactionnel : l'objectif est celui d'apporter une solution utile, tangible et rapidement applicable.

1.3.2.1 Surveillance d'assertions

Ces travaux s'inscrivent dans l'un des axes de recherche des dernières années au sein du laboratoire TIMA. Ils trouvent appui sur le concept de moniteur de surveillance, un composant matériel synthétisé à partir d'une assertion, destiné à être connecté au système

à vérifier [GBA+99, ABG+00]. Le rôle d'un tel composant, décrit dans le même langage de description de matériel que le design, est de contrôler certains comportements de ce dernier, cela tout au long de la simulation. Par exemple, dans le cas d'une architecture décrite en VHDL au niveau transfert de registres, un moniteur peut surveiller constamment des relations entre certains signaux utilisés pour les échanges entre deux composants, cf. figure 1.3

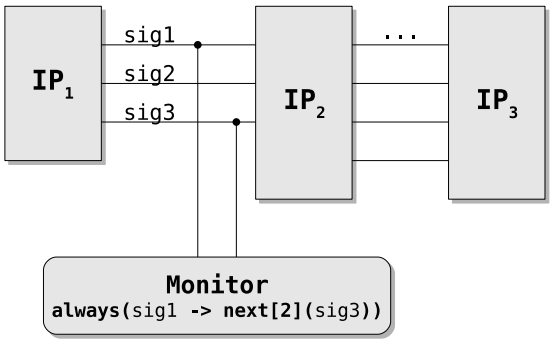

FIGURE 1.3 – Moniteur de surveillance

Cette activité s'inscrit parfaitement dans le cadre de l'ABV : typiquement, un comportement est décrit de façon concise et formelle via une assertion, à l'aide d'un langage de spécification adéquat, comme mentionné ci-dessus.

Parmi les techniques existantes pour la construction de moniteurs de surveillance à partir d'assertions, celle développée initialement au sein de TIMA est originale, à la fois modulaire et efficace. A partir d'une assertion exprimée dans la langage PSL (*Property Specification Language*, Standard IEEE 1850 [PSL05]), elle permet de générer automatiquement un composant de surveillance VHDL, décrit au niveau transfert de registres et parfaitement "synthétisable" [MAB06b, OMAB08].

Cette technique constitue le point de départ pour les travaux de recherche menés dans le cadre de cette thèse. En effet, puisque la solution originelle s'est avérée valide dans un contexte RTL, nous avons étudié son intérêt et son applicabilité dans le contexte TLM.

1.3.2.2 Organisation du document

Avant de présenter les résultats obtenus, nous nous focaliserons d'abord sur le contexte. Le chapitre 2 offre une introduction à la bibliothèque SystemC, extension du langage C++ pour la modélisation de circuits, ainsi qu'à son niveau de modélisation transactionnel. La première partie de ce chapitre adresse donc les aspects clé de SystemC et de TLM, aspects qui jouent un rôle important dans les choix et dans les solutions que nous proposons. La deuxième partie du chapitre 2 introduit les logiques temporelles et les langages de spécification d'assertions. Après un survol des moyens de spécification passés et présents, le chapitre se conclut par l'étude de PSL, le langage que nous avons choisi et adopté dans nos travaux.

Parmi les efforts existants en matière de vérification de descriptions matérielles, un certain nombre comporte des idées ou des suggestions intéressantes, qui méritent d'être approfondies ou améliorées. D'autres efforts exhibent des écueils à éviter. Avant de nous pencher sur le développement de notre solution, nous avons donc effectué une étude des travaux menés dans la vérification de descriptions SystemC, dont seul un sous-ensemble restreint vise réellement TLM. Cette étude est relatée dans le chapitre 3.

Notre contribution, détaillée dans les trois chapitres suivants, s'efforce de résoudre un certain nombre de limitations liées aux travaux existants et vise ainsi une solution plus mature. Le chapitre 4 se concentre sur les éléments à la base de notre solution d'ABV. D'une part, une technique efficace de construction de moniteurs TLM à partir de propriétés PSL est présentée ; cette technique est inspirée de l'approche originale mentionnée dans la section 1.3.2.1. D'autre part, une méthode spécifique de surveillance des actions de communication à haut niveau d'abstraction (échanges entre les composants du SoC) est détaillée. Le chapitre 5 étend significativement les possibilités offertes par notre approche, en proposant, pour les assertions PSL, à la fois un support formel et une mise en œuvre pratique pour des variables auxiliaires globales et locales. Ces variables auxiliaires constituent un élément essentiel lors de la spécification d'assertions à haut niveau d'abstraction, comme le niveau transactionnel de SystemC. Le chapitre 6 détaille la mise en œuvre de tous ces concepts dans l'outil prototype ISIS, ainsi que quelques apports concernant cette solution, en particulier en matière de campagnes de tests.

Afin d'illustrer l'intérêt de la solution proposée, plusieurs expérimentations sont regroupées dans le chapitre 7. Ces expérimentations portent sur des designs aux dimensions et complexités différentes. Les résultats obtenus permettent de souligner le fait que la méthode de vérification suggérée reste applicable quelle que soit la taille du design à vérifier.

Enfin, au travers d'une synthèse et d'un bilan des travaux menés, le chapitre 8 conclut cette thèse en offrant à la fois une analyse de la solution proposée et une ouverture vers des travaux futurs, passibles de l'améliorer et de la compléter.

Chapitre 2

Contexte

Introduction

Dans le passé, plusieurs sociétés et groupes de recherche académiques se sont intéressés aux langages C et C++ pour la modélisation de systèmes matériels [BM10]. Certains ont développé des bibliothèques et inventé des techniques propriétaires afin d'exprimer des concepts liés au monde du matériel, tels que la simultanéité et la communication entre blocs, les types de donnés et le temps. Toutefois, une standardisation des approches et des techniques était nécessaire afin de rendre possible l'intégration des modèles et des flots de conception. Cet effort de standardisation a donné naissance à des langages tels que SystemC.

2.1 Modélisation de circuits avec SystemC TLM

SystemC est une bibliothèque du langage C++ [CPP03]. Elle permet la modélisation de systèmes complexes, impliquant un grand nombre de composants matériels et logiciels, à différents niveaux d'abstraction [SC005]. Cette bibliothèque présente un ensemble de classes et méthodes pour la modélisation de matériel et la description de comportements au niveau système, et inclut un noyau de simulation. SystemC est actuellement défini par le standard IEEE 1666-2005 [SC005]. Avant la publication du standard, il s'agissait d'une simple bibliothèque conçue comme une démonstration de faisabilité *open source*, mise à disposition par l'OSCI[1]. Une version officielle est toujours spécifiée et diffusée par l'OSCI.

La bibliothèque TLM (*Transaction Level Modeling*) de SystemC réunit un ensemble d'interfaces, classes et styles de modélisation pour une conception efficace à haut niveau d'abstraction et pour une simulation rapide [OSC09]. Dans un modèle transactionnel, les détails des communications entre les différents blocs de calcul sont dissociés des détails concernant l'implémentation de ces blocs. Les classes de la bibliothèque transactionnelle sont une extension de la bibliothèque SystemC et constituent donc un complément au standard IEEE.

[1] *Open SystemC Initiative*, une association à but non lucratif dédiée à définir et promouvoir SystemC (`http://www.systemc.org`).

2.1.1 Principaux composants SystemC

La simultanéité en SystemC existe à deux niveaux [BM10]. En premier lieu, un système peut comporter plusieurs modules ou IPs (*Intellectual Properties*) qui fonctionnent en parallèle. En deuxième lieu, un même module peut contenir plusieurs processus qui s'exécutent en parallèle. Les modules représentent la structure du système et peuvent être assemblés de façon hiérarchique. SystemC ne définit aucune règle précise quant au niveau de détail d'un module. Par exemple, un processeur, décrit par un module et comportant une mémoire cache de premier niveau, pourrait contenir directement la fonctionnalité du cache dans sa description, ou communiquer avec un deuxième module distinct modélisant la mémoire. Dans le premier cas (figure 2.1 a), les interfaces entre le processeur et sa mémoire ne sont pas exposées : l'échange d'information reste caché à l'intérieur du module. Dans le deuxième cas (figure 2.1 b), la communication est exposée à l'utilisateur.

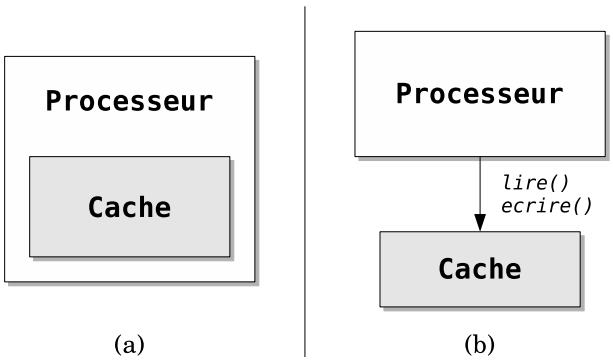

FIGURE 2.1 – Visibilite des communications par l'utilisateur

2.1.1.1 Modules

Le module est l'élément de base pour partitionner tout design SystemC [GLMS02] et s'obtient par dérivation de la classe `sc_module`. Il peut contenir des structures de données C++ aussi bien que d'autres modules SystemC. La communication entre modules se fait essentiellement à l'aide de *ports*, *exports* et *canaux*. La fonctionnalité d'un module est caractérisée par son ensemble de processus, exécutés de façon concurrente. Toute classe qui implémente un module doit avoir au moins un constructeur.

Dans la modélisation de matériel, un *signal* matériel est le moyen classique de communication et de synchronisation. Au niveau système, il s'agit d'un mécanisme de trop bas niveau [GLMS02]. SystemC utilise les *ports*, les *interfaces* et les *canaux* pour fournir la flexibilité et le niveau d'abstraction nécessaires.

2.1.1.2 Ports

Un port (classe `sc_port`), élément central dans l'interconnexion, est un point d'entrée et de sortie d'un module vers un canal. Ce dernier, primaire ou hiérarchique, permet la communication entre modules via l'implémentation de primitives de communication. L'accès à ces fonctions se fait de façon indirecte, depuis un port, en exploitant la notion d'interface. Les ports offrent un moyen de décrire un module qui rend ce dernier indépendant du contexte dans lequel il sera créé : grâce à la surcharge de l'opérateur "`->`", un port fait suivre les appels aux méthodes de communication vers le canal auquel il est lié. Il

définit donc un ensemble de services (méthodes de communication) requis par le module qui le contient.

Un export (classe `sc_export`) rend visible un canal (ou module) et ses fonctionnalités depuis l'extérieur. Il permet à un port de se connecter à un canal implémenté dans un autre module et autorise la communication entre niveaux hiérarchiques différents.

Dans l'exemple montré par la figure 2.2, le port P1 permet au module de faire appel à la méthode de communication du canal. Si le canal contient d'autres sous-canaux ou sous-modules, outre la transmission de l'appel de méthode, le export P2 permettrait l'accès aux méthodes de communication de ces sous-éléments.

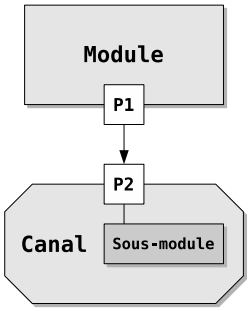

FIGURE 2.2 – Ports de communication

2.1.1.3 Interfaces

Une interface spécifie un ensemble de fonctionnalités, sans en donner l'implémentation, en n'indiquant que la signature de chaque opération. Plus précisément, elle spécifie les primitives de communication d'un canal (ou d'un autre module) accessibles par un port. Un canal peut satisfaire plusieurs fonctionnalités de communication en implémentant les primitives de plusieurs interfaces. Au contraire, un port n'est lié qu'à une seule interface : le module qui contient ce port pourra donc communiquer suivant les primitives (de l'interface) du port. En SystemC, toutes les interfaces doivent hériter, directement ou indirectement, de la classe abstraite `sc_interface`. Les interfaces permettent d'appliquer le patron de conception IMC (*Interface Method Call*, pour "appel de méthode d'interface") qui rend possible la liaison d'un port à un canal. Puisque le port se limite à transférer les appels de ses méthodes vers le canal auquel il est lié, l'utilisateur peut facilement remplacer le canal par un autre, respectant la même interface mais offrant une implémentation différente. Ainsi, les parties fonctionnelle et de communication sont rendues indépendantes [KH09].

2.1.1.4 Canaux

Les ports et les interfaces définissent les fonctionnalités disponibles lors de la communication entre modules. Les canaux définissent comment ces communications sont effectuées : un canal implémente une interface lorsqu'il fournit une implémentation concrète pour toutes les opérations de l'interface. Par ailleurs, un canal peut implémenter plusieurs interfaces. En SystemC les canaux peuvent être très hétérogènes en termes de complexité, et incluent aussi bien des simples signaux que des modules hiérarchiques modélisant des protocoles complexes. De même, les restrictions relatives au nombre de connexions et aux

accès dépendent entièrement du type de canal. La bibliothèque définit un ensemble de canaux de communication de base, comme le `sc_signal` et la `sc_fifo`.

Canaux primitifs Un canal primitif (classe de base SystemC `sc_prim_channel`) est atomique dans la mesure où il ne contient pas d'autres structures SystemC. Il supporte la technique *request-update* pour l'accès et la mise à jour. Cette technique, destinée à reproduire la concurrence matérielle, est fortement liée à la sémantique de simulation SystemC présentée en section 2.1.2. Quand plusieurs actions simultanées sont simulées les unes à la suite des autres, le changement de l'état interne du canal doit être retardé afin d'éviter le non déterminisme. Toutes les opérations passibles de changer l'état d'un canal primitif (par exemple une écriture qui en modifierait la valeur) n'auront d'effet qu'après synchronisation ou terminaison de tous les processus actuellement actifs. Deux méthodes sont à la base de ce mécanisme :

- la methode `request_update()` demande la mise à jour en indiquant à l'ordonnanceur de positionner le canal dans une file d'attente,
- la methode `update()`, exécutée dans une deuxième phase, effectue la mise à jour.

Puisque ce type de comportement est identique pour tous les canaux primitifs, seulement la deuxième méthode est déclarée comme *virtuelle*[2] et chaque implémentation en spécifie le fonctionnement. Comme nous le verrons par la suite (cf. chapitre 4 section 4.2.4), ce mécanisme s'avère particulièrement utile lors de l'observation de propriétés fonctionnelles dans des designs SystemC hybrides, où les canaux primitifs coexistent avec des canaux hiérarchiques et d'autres composants plus abstraits.

Canaux hiérarchiques Un canal hiérarchique offre un moyen plus puissant pour décrire des structures de communication complexes. À la différence du canal primitif, le canal hiérarchique n'a pas accès à la technique *request-update* mais il peut contenir d'autres sous-blocs et structures SystemC, de même manière qu'un module classique. Par exemple, le *On-Chip Bus* (OCB) comporte plusieurs unités intelligentes, comme un arbitre ou une unité de décodage [GLMS02]. Un canal primitf n'est pas adapté à ce type de modèles.

2.1.1.5 Exemple

Les concepts et les éléments décrits ci-dessus sont résumés dans l'exemple illustré[3] par la figure 2.3. Cet exemple est issu de la distribution 2.2.0 de la bibliothèque SystemC.

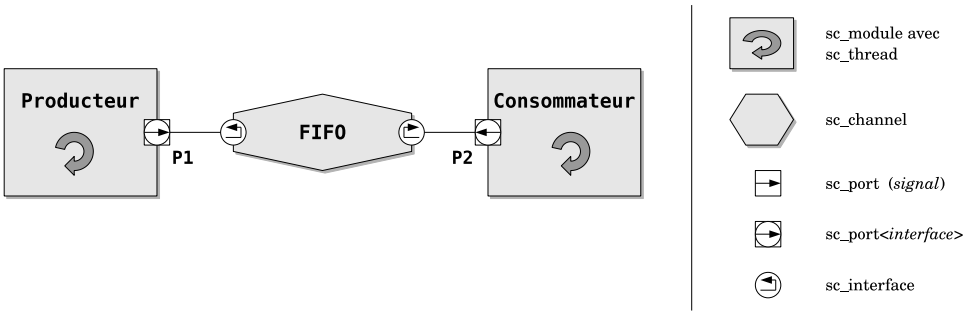

FIGURE 2.3 – Exemple élémentaire d'architecture en SystemC

[2] Voir [CPP03] pour la notion de méthode virtuelle.
[3] Les notations utilisées sont inspirées de celles de [BD04].

Le modèle SystemC de l'exemple comporte trois instances de modules décrivant les composants principaux du système :

- le Producteur est un module émetteur qui envoie des messages à destination d'un module récepteur via une FIFO, en utilisant son port P1 ;
- le Consommateur est le module récepteur, il lit les messages écrit dans la FIFO en utilisant son port P2 ;
- la FIFO est un canal de communication hiérarchique simple, utilisé pour les échanges de messages entre les deux modules précédents.

Ces trois composants principaux héritent de la classe `sc_module`. La FIFO est un canal dit hiérarchique, même si elle ne comporte pas de sous-modules. Elle implémente deux interfaces SystemC : `write_if`, qui définit les fonctionnalités d'écriture, et `read_if`, qui définit celles de lecture. Par exemple, l'interface d'écriture est spécifiée comme suit :

```
class write_if : virtual public sc_interface
{
  public:
    virtual void write(char) = 0;
    virtual void reset() = 0;
};
```

De même, l'interface `read_if` spécifie l'existence d'une fonction `void read(char &)`. La FIFO est liée à la fois au port P1 du Producteur et au port P2 du Consommateur. La classe `fifo` fournit des implémentations pour les méthodes virtuelles pures des classes `write_if` et `read_if`, permettant ainsi l'utilisation des fonctions correspondantes par le producteur et par le consommateur auxquels elle va être liée :

```
class fifo : public sc_module, public write_if, public read_if
{
  public:
    fifo(sc_module_name name) :
      sc_module(name), num_elements(0), first(0) { }
    void write(char c) {
      // definition de la fonction d'ecriture...
    }
    void read(char &c){
      // definition de la fonction de lecture...
    }
    ...
  private:
    enum e { max = 10 };
    char data[max];
    int num_elements, first;
    sc_event write_event, read_event;
};
```

Par exemple, le Producteur fait appel à la méthode `write` par son port P1. Grâce à la surcharge de l'opérateur "`->`" pour les ports, P1 fait suivre cet appel à la FIFO. La figure 2.4 résume ce mécanisme. Dans des designs plus élaborés, plusieurs appels de fonctions peuvent se succéder pour réaliser une seule opération.

2.1.2 La simulation

Un modèle SystemC peut être compilé avec un compilateur C++ pour obtenir un programme exécutable réalisant une simulation du système décrit. L'exécution d'une application SystemC comporte une phase d'*élaboration*, suivie par la *simulation* proprement

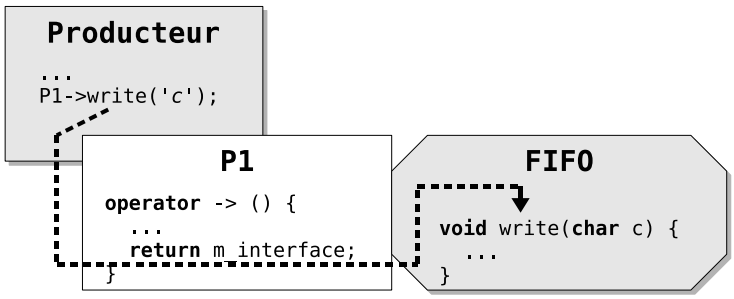

FIGURE 2.4 – Port et surcharge de l'operateur *flèche*

dite. La première est caractérisée par l'initialisation des structures de données et par l'établissement des connexions [BD04]. La deuxième orchestre l'exécution des processus afin de décrire le comportement du système, sur la base du noyau de simulation SystemC.

2.1.2.1 Processus

Tout processus doit être inclus dans un module : il est défini comme une fonction membre et doit être qualifié comme processus SystemC à l'intérieur du constructeur. Le processus constitue l'unité fonctionnelle de base d'un module. Chaque module peut avoir plusieurs processus, enregistrés auprès du noyau de simulation lors de leur déclaration dans le constructeur, et simulés de façon concurrente. SystemC définit deux types de processus, identifiés par les macros `SC_METHOD` et `SC_THREAD`.

Le premier, une fois appelé, s'exécute entièrement, sans interruption du début à la fin. Il ne garde aucun état d'exécution implicite : son déroulement ne peut pas être arrêté par un appel à la fonction `wait()`. Pour cette raison, le noyau SystemC n'a aucune nécessité de garder en mémoire son contexte d'exécution et un processus de ce type simule plus rapidement qu'un processus avec contexte.

Le deuxième peut se suspendre par un appel à la fonction `wait()` : son état est alors sauvegardé. Chaque `SC_THREAD` possède un contexte d'exécution qui lui est propre. Une fois rétabli, il reprendra son exécution depuis le dernier point d'arrêt plutôt que du début. Cette caractéristique le rend plus souple d'usage, mais elle en dégrade la vitesse d'exécution.

L'activation des processus est contrôlée par le noyau de simulation SystemC. Ce dernier effectue automatiquement un appel à tous les processus durant l'initialisation du système, lorsque le temps de simulation est encore à 0. L'utilisateur doit indiquer explicitement s'il ne souhaite pas initialiser un processus, par moyen de la fonction `dont_initialize()`. Le simulateur est dirigé par événements : les processus sont exécutés en réponse à l'occurrence d'événements.

2.1.2.2 Événements et synchronisation

Un événement, représenté par la classe `sc_event`, est un objet qui détermine quand l'exécution d'un processus doit être déclenchée (ou rétablie). Il est donc à la base de la synchronisation des processus. Il n'a ni valeur ni durée : il se produit à un instant précis dans le temps. Plus concrètement, un événement est utilisé pour représenter une condition qui peut se vérifier au cours de la simulation, et à laquelle un processus donné est dit sensible. Cette sensibilité peut être indiquée

- Dans une liste, dite de “sensibilité statique”, lors de la déclaration du processus dans le constructeur du module qui le contient.
- À l'aide de la fonction `next_trigger()` pour un processus sans contexte. Cette fonction indique sous quelle condition la méthode sera appelée de nouveau. Elle peut accepter aussi des retards temporels.
- À l'aide de la fonction `wait()` pour un processus avec contexte. Cette fonction indique sous quelle condition le processus sera réveillé. Elle peut accepter aussi des retards temporels.

Le fait de signaler qu'un événement a lieu est nommé *notification*, et se traduit par un appel à la fonction `notify()` de la classe `sc_event`. Comme détaillé au paragraphe suivant, ces demandes de notification peuvent être immédiates, retardées d'une durée infinitésimale ou planifiées après un certain temps.

2.1.2.3 Sémantique de simulation

La simulation en SystemC s'appuie sur un modèle de temps absolu. La bibliothèque utilise un nombre entier non signé sur 64 bits pour représenter les valeurs temporelles. La résolution (secondes, microsecondes...) peut être définie par l'utilisateur. Comme pour VHDL [VHD02] et Verilog [VER01], en SystemC chaque instant peut être décomposé en plusieurs cycles delta. Un *cycle delta* peut être défini comme un intervalle de durée infinitésimale, toujours trop petit pour faire avancer le temps de simulation, mais utilisé pour imposer un ordre partiel entre actions simultanées. Au cours d'un cycle delta, l'ordonnanceur exécute les actions planifiées pour l'instant courant suivant deux phases principales : *évaluation*, durant laquelle tous les processus dans l'état éligible sont exécutés ; *mise-à-jour*, pendant laquelle sont effectués tous les appels en attente à la méthode `update()` (cf. section 2.1.1, canaux primitifs). Le principe du noyau de simulation SystemC, schématisé en figure 2.5, peut être résumé comme suit :

1. *Initialisation.* Chaque processus sans contexte est exécuté une fois et chaque processus avec contexte est exécuté jusqu'à son premier point de synchronisation (appel à `wait()`). L'ordre dans lequel les processus seront choisis n'est pas spécifié. Cette étape est effectuée une seule fois.

2. *Évaluation.* Tous les processus dans l'état éligible (prêts à être exécutés à l'instant courant) sont réveillés et exécutés un par un. L'ordre n'est pas spécifié. La notification immédiate d'événements est possible au cours de cette étape. Il faut souligner le fait qu'un seul processus à la fois est réellement actif et en train de s'exécuter, bien que tous les processus ainsi exécutés soient présentés comme concurrents à l'instant de simulation courant.

3. *Mise-à-jour.* Tous les appels à la méthode `update()` en attente sont effectués. Ces appels en attente sont issus des requêtes menées via la méthode `request_update()` au cours de la phase précédente.

4. *Notification retardée.* S'il y a en attente des notifications d'événements retardées d'un cycle delta, tous les processus qui en dépendent sont marqués comme éligibles et la simulation reprend de l'étape 2.

5. *Avancement temporel.* S'il n'y a aucune notification d'événement planifiée après un certain temps, la simulation s'arrête. Sinon, le temps de simulation est avancé jusqu'à la notification retardée la plus proche (il peut y en avoir plusieurs au même

instant) : tous les processus qui en dépendent sont marqués comme éligibles et la simulation reprend de l'étape 2.

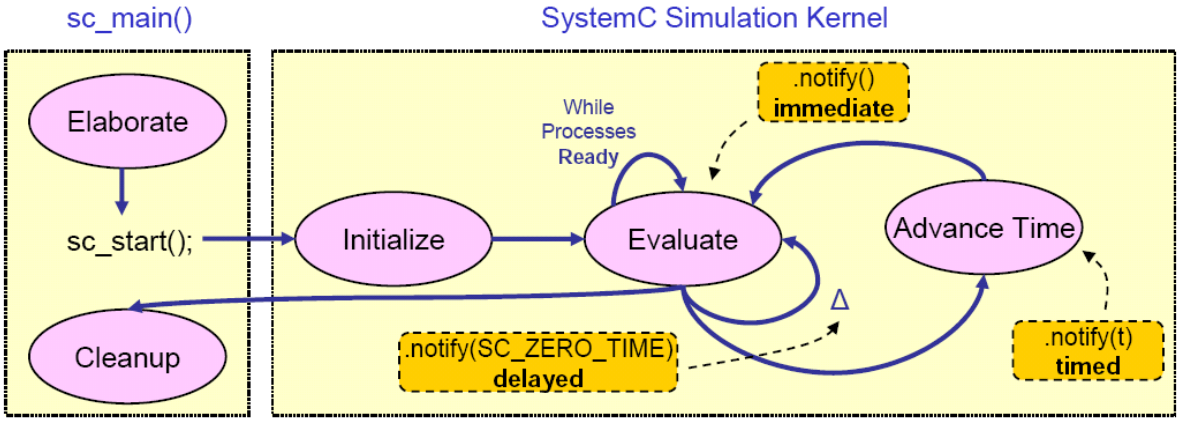

FIGURE 2.5 – Noyau de simulation SystemC (source : [BD04])

Par ailleurs, les processus en SystemC ne sont pas préemptifs, c'est-à-dire que le déroulement d'un processus ne peut être interrompu que par lui-même. Pour un processus avec contexte, le code compris entre deux appels à la méthode `wait()` sera exécuté sans aucune interruption de la part des autres processus.

En résumant, la simulation proprement dite peut être vue comme une boucle. Celle-ci se répète jusqu'à écoulement du temps limite, épuisement de toutes les activités en attente ou appel à la fonction `sc_stop()`.

La figure 2.5 montre aussi la phase d'élaboration, qui précède le premier appel à la fonction `sc_start()` et qui est accomplie en amont de la simulation. Toutes les instances de composants matériels (modules, ports, etc.) peuvent être créées uniquement au cours de cette phase. De même, toute connexion matérielle (ports et signaux) n'est possible que durant la phase d'élaboration. Il s'agit d'un détail important, puisque toute instrumentation d'un design existant par des composants additionnels, telle que nous la discuterons dans le chapitre 4, ne peut être effectuée qu'une fois pour toutes avant le début de la simulation.

2.1.3 La bibliothèque TLM

TLM, acronyme pour *Transaction Level Modeling*, est devenu un terme très en vogue ces dernières années, presque pléthorique, utilisé pour désigner un très grand nombre de techniques de modélisation et de niveaux d'abstraction [GCMC+05]. Le but de cette section n'est pas d'opérer une taxonomie [BAGK07], mais plutôt de présenter un certain nombre de concepts liés à TLM et intéressants dans le cadre de cette thèse.

Une distinction qui peut prêter à confusion est celle entre

- le *niveau de modélisation transactionnel*, qui est à priori indépendant du langage de description utilisé et
- la *bibliothèque TLM de SystemC*, qui est une implémentation proposée par OSCI de certains concepts liés à ce niveau de modélisation.

La suite se focalise principalement sur le deuxième point.

Une transaction peut être assimilée au concept de communication, ou à celui d'ensemble d'interactions, entre certains blocs du système. En SystemC une opération de commu-

nication est effectuée par un appel de fonction. SystemC TLM met en avant cet aspect. Les communications sont modélisées par les canaux, alors que les requêtes de transactions ont lieu en appelant les méthodes des interfaces implémentées par ces canaux [CG03]. Les détails non indispensables liés à la communication et au calcul sont cachés dans un tel modèle. Toutefois, si le format des données, des paramètres et la syntaxe des fonctions ne sont jamais identiques selon les sources des modèles, les composants ne sont pas réutilisables et interchangeables. La bibliothèque TLM 2.0 de OSCI offre une réponse à ce besoin de standardisation. Elle comporte un ensemble d'interfaces de base, de protocoles et de classes pour les données, les ports de communication et d'autres services. Les interfaces et les classes proposées sont particulièrement adaptées à la conception de modèles de bus *memory-mapped*.

Depuis la sortie officielle de SystemC 2.0, vers la moitié de 2001, STMicroelectronics avait commencé à évaluer les aspects de plus haut niveau du langage, comme les canaux et les événements, en créant ainsi des premiers modèles transactionnels. En 2005 l'OSCI a approuvé la première version du standard TLM. Au cours des années suivantes la bibliothèque a connu un certain nombre d'évolutions, conduisant à la version 2.0.1 diffusée en juillet 2009.

À l'heure actuelle, la bibliothèque inclut aussi les éléments de la version 1.0, qui sont toutefois regroupés séparément de la structure principale de la version 2.0. Tous les éléments des bibliothèques transactionnelles constituent des couches supplémentaires au standard, organisées comme le montre la figure 2.6.

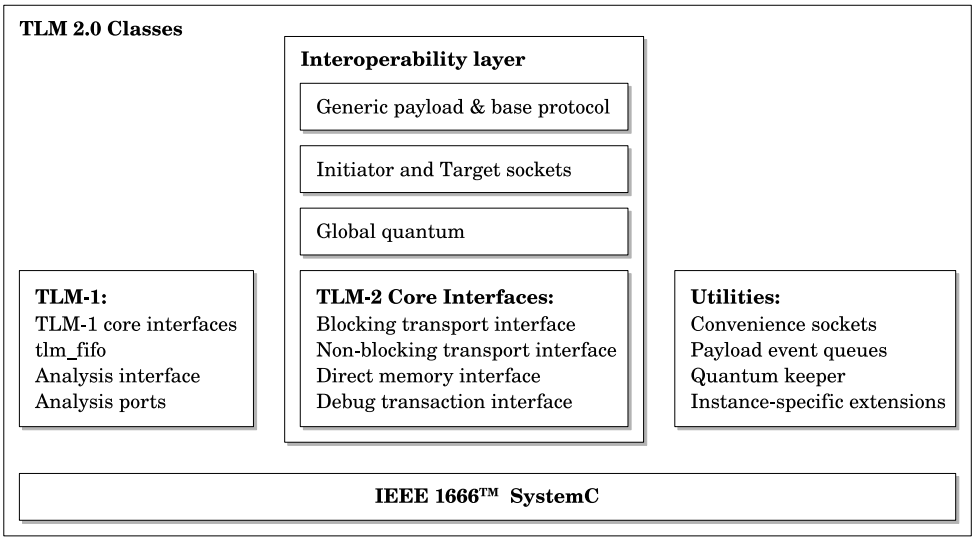

FIGURE 2.6 – Éléments de la bibliothèque SystemC TLM (source : [OSC09])

2.1.3.1 Niveaux d'abstraction

Avant l'arrivée du standard OSCI 2.0 de la TLM, on identifiait typiquement trois sous-niveaux d'abstraction [GCMC+05], encore compris et récurrents à l'heure actuelle :

- PV (*Programmer's View*), qui est le niveau le plus élevé, ne comportant aucune information sur le temps. La synchronisation se fait uniquement grâce aux événements.

– PVT (*Programmer's View with Time*), qui ajoute la notion de temps : toutes les actions et les événements correspondent à une estampille précise, à un point sur l'axe temporel. Il peut y avoir des attentes.
– CA (*Cycle Accurate*), qui se rapproche de RTL, mais pour lequel la communication en général se fait toujours via transactions.

Nous verrons dans la section 2.1.3.3 les nouvelles définitions des styles de modélisation.

2.1.3.2 Architecture TLM typique

Les protocoles de communication s'appuyant sur la bibliothèque transactionnelle impliquent des systèmes avec maîtres et esclaves. Un *maître* est un module source qui effectue les requêtes et génère les transactions. Il démarre donc la communication. Un *esclave* est un module cible qui reçoit et dessert les requêtes. Il traite ainsi les transactions. Dans la plupart de ces systèmes, un chemin de communication comporte plus de deux composants. Un design typique utilise au moins un canal pour l'acheminement de la transaction entre maître et esclave. Ce canal est souvent assez complexe (cf. canaux hiérarchiques en section 2.1.1.4) et fait office à la fois de cible, vis-à-vis du module maître, et de module démarrant la communication, vis-à-vis de l'esclave. Parfois plusieurs canaux ou modules d'interconnexion sont utilisés le long d'un chemin de communication.

La dernière version de la TLM définit la notion de *initiator socket* et celle de *target socket*, qui regroupent les différentes interfaces de transport en un seul objet. Les designs basés sur les versions précédentes utilisent souvent des composants de type *initiator port* et *target port*. Le patron de conception *initiator port* convertit les appels aux méthodes des modules cibles en appels aux méthodes des interfaces de communication de la bibliothèque TLM [KH09]. Du côté du module cible, un concept dual au précédent est souvent utilisé : les appels aux méthodes de communication sont traduits en appels aux méthodes de l'esclave. Ceci s'exprime via le patron de conception *slave base*, qui offre une classe de base abstraite pour les modules cibles. Le *target port* est un export (cf. section 2.1.1.2).

La méthodologie de modélisation utilisant ces composants s'appuie souvent sur un modèle d'architecture fractionnée en trois couches, montrées en figure 2.7.

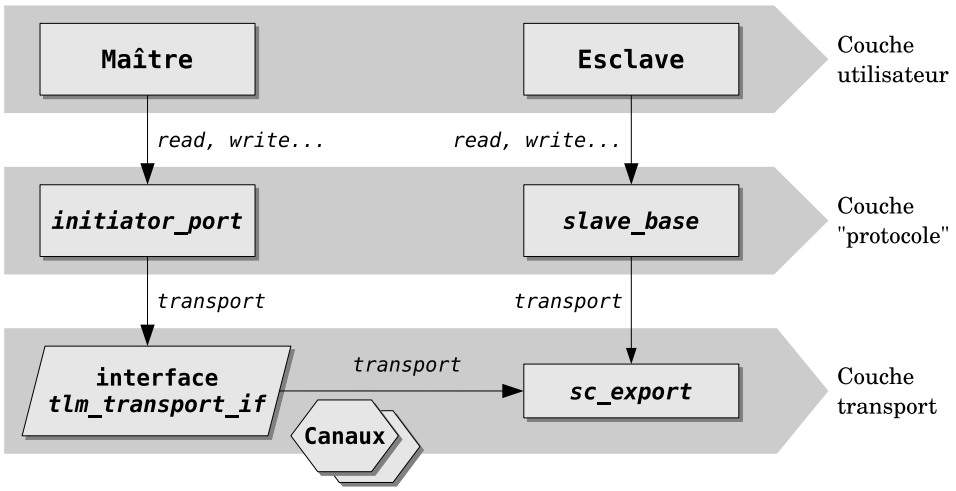

FIGURE 2.7 – Modèle d'architecture TLM en trois couches

Dans ce modèle, le module maître fait appel aux primitives de communication dites "de commodité" et définies dans son `initiator_port` (par exemple, `read` et `write`). Ce

dernier les traduit en appels à la méthode `transport` du canal de communication utilisé. A son tour, le canal appelle la méthode `transport` implémentée au niveau de l'esclave, qui hérite de la classe `slave_base` et effectue donc la traduction inverse (appel aux méthodes de commodité). Ces motifs de programmation sont utilisés pour masquer les détails de réalisation de la communication. Ils permettent par exemple de substituer le canal de communication par un autre, ou plusieurs autres, sans devoir modifier le reste du design.

Exemple d'implémentation L'exemple de contrôleur DMA (*Direct Memory Access*, pour contrôleur d'accès direct à la mémoire) issu de la première version d'évaluation de la bibliothèque transactionnelle[4] est une implémentation typique du modèle décrit. Il est illustré par la figure 2.8.

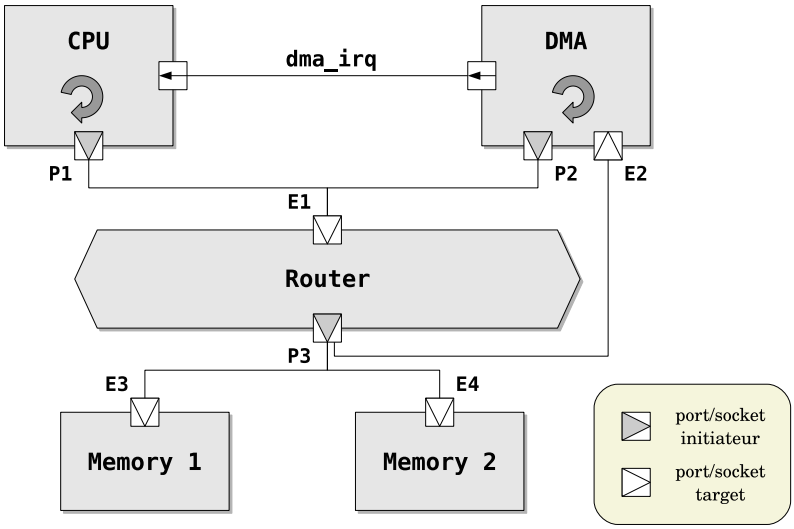

FIGURE 2.8 – Contrôleur DMA

Dans l'exemple

- le CPU est un module maître qui programme le DMA via le Router, en utilisant son port P1, afin d'opérer des transferts en mémoire ;
- le DMA est à la fois esclave et maître : il dessert les requêtes du CPU transmises via le Router et gère les transferts entre les deux mémoires via ce même Router, en utilisant le port P2 ;
- les deux mémoires sont deux instances distinctes d'un même module décrivant le type de composant Mémoire : il s'agit de deux esclaves qui desservent les requêtes du DMA ;
- le Router est un canal hiérarchique, de type *memory-mapped* ;
- le signal IRQ est un canal primitif, utilisé par le DMA pour effectuer des interruptions matérielles à l'attention du CPU, afin de signaler l'accomplissement d'un transfert ;
- les ports P1, P2 et P3 utilisent le patron de conception *initiator port* et déclarent quatre méthodes de commodité : `read`, `write`, `read_block` et `write_block` ;
- les ports E1 à E4 sont des *target ports* ;

[4] *TLM 2.0 Draft 1 for Public Review*, dont une courte introduction est disponible à l'adresse `http://www-ti.informatik.uni-tuebingen.de/~systemc/Documents/Presentation-15-OSCI2_aynsley.pdf`

– les trois modules esclaves, c.-à-d. le DMA et les deux mémoires, sont basés sur le patron de conception *slave base* et implémentent donc la classe abstraite correspondante, où les appels à la méthode de transport sont retraduits en appels aux méthodes de commodité.

Dans ce système, quand le CPU programme le contrôleur DMA, il fait appel à la méthode `write` de son port P1, en indiquant, entre autres, l'adresse de l'un des registres du contrôleur et la donnée à écrire. Ceci se passe donc au niveau de la couche utilisateur de la figure 2.7. Le port P1 traduit l'appel à `write` en un appel à la méthode `transport` et fait suivre ce dernier appel au Router. Cette tâche relève de la couche protocole. Le Router, élément clé de la couche transport, hérite de l'une des interfaces TLM de transport. Dans son implémentation, il appelle la méthode `transport` de l'esclave (le DMA dans ce cas). La classe `slave_base`, parente de tout esclave, traduit l'appel à `transport` en un appel à `write`. Enfin, le DMA implémente concrètement la méthode `write` : la donnée reçue est écrite dans le registre demandé. Par ailleurs, le Router est un exemple de canal *memory-mapped* : toutes les communications en entrée du canal comportent l'adresse destination de l'esclave, connue par le maître. Le canal utilise un fichier de configuration qui lui permet d'identifier l'esclave destinataire en fonction de l'adresse spécifiée.

Les *sockets* du dernier standard TLM regroupent toutes les fonctionnalités spécifiées par les interfaces de transport (cf. sections 2.1.3.4 et 2.1.3.7) en un seul objet. De plus, la couche transport est enrichie et améliorée dans ce dernier standard, comme expliqué dans les sections qui suivent. Toutefois, le principe de base est toujours le même :

1. le module maître utilise son port de communication (*initiator port* ou *initiator socket*) pour démarrer une transaction ;
2. le port fait suivre l'appel à une méthode de transport du canal (appel qui traverse le *target port* ou *target socket* du canal) ;
3. l'implémentation de la communication est faite par le canal et cachée par celui-ci ;
4. le canal utilise un port de communication (encore une fois de type *initiator port* ou *initiator socket*) pour faire suivre la requête à l'esclave, ou au prochain module d'interconnexion.

Dans les systèmes transactionnels, plusieurs appels de fonctions se succèdent donc pour effectuer une transaction.

2.1.3.3 Styles de modélisation

Le standard OSCI TLM 2.0 n'adopte plus les niveaux d'abstraction tels qu'ils ont été définis dans la section 2.1.3.1, mais présente ces concepts de façon un peu différente. Deux styles de modélisation, LT pour *Loosely Timed* ("vaguement") et AT pour *Approximately Timed* ("à peu près"), sont définis, chacun étant plus ou moins adapté à des cas d'utilisation donnés (développement du logiciel, exploration d'architecture, vérification matérielle...). Plusieurs mécanismes sont disponibles et peuvent être combinés selon le style choisi. Parmi ces mécanismes on trouve les interfaces bloquantes et non-bloquantes, l'identification des phases durant une transaction, la notion de quantum temporel, les *sockets* (prises/coupleurs, assimilables au concept de port). Ces styles sont définis indépendamment des interfaces standard de la bibliothèque : ils ne représentent pas vraiment deux niveaux d'abstraction, mais plutôt deux ensembles de techniques et d'idiomes.

Bien évidemment, certaines classes et interfaces de la bibliothèque sont plus adaptées que d'autres à un style donné.

Le style de modélisation LT est préconisé principalement pour les plates-formes décrites au niveau système et pour les modèles précis au niveau instructions, où la précision au niveau cycles n'est pas indispensable et la vérification fonctionnelle impose simplement un ordonnancement correct des transactions [BM10].

Ce style emploie principalement l'interface de communication bloquante (méthode `b_transport`), qui ne considère que deux instants dans le temps : le début et la fin d'une transaction, associés respectivement à l'appel et au retour de la fonction de transport bloquante, comme montré par le diagramme de séquence de la figure 2.9. Ici le maître démarre une première fois la transaction : le premier instant dans la communication est identifié au point *a*. L'esclave répond tout de suite et le retour de la fonction de transport, identifié par le point *b*, s'effectue au même temps de simulation. Peu après le maître démarre une deuxième transaction (point *c*), mais cette fois l'esclave effectue un traitement plus long et ne répond que 40ns plus tard (point *d*).

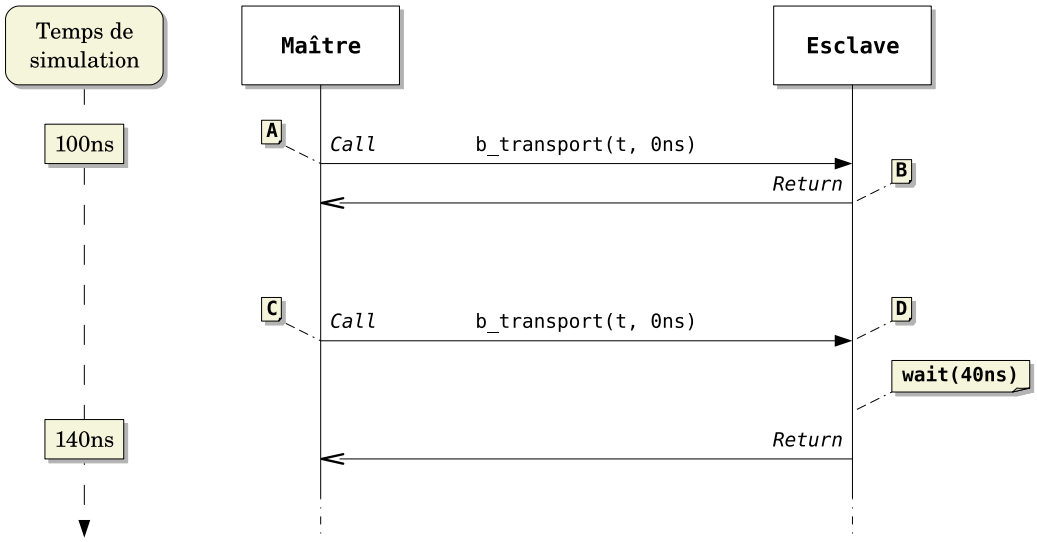

FIGURE 2.9 – Exemple en style LT

Avec le style AT, les dépendances temporelles entre composants peuvent être représentées avec plus de précision, ceci afin de modéliser fidèlement les conflits d'accès aux ressources et l'arbitrage. Ce style est plus adapté à l'analyse de performances et à la description de tout système dont le bon fonctionnement dépend fortement du temps. Lors d'une transaction, ce style identifie par défaut quatre phases : le début de la requête, la fin de la requête, le début de la réponse, la fin de la réponse. Le diagramme de séquence de la figure 2.10 montre un exemple. Dans ce cas, le maître marque le début de la communication au point *a*, à 100ns, et la fonction retourne avec l'acceptation de l'opération par l'esclave. À 110ns l'esclave indique la fin de la phase de requête et donc le début de la préparation de la réponse (point *b*) ; 10ns plus tard il marque le début de sa réponse à la requête (point *c*). Enfin, à 130ns, le maître acquitte, en indiquant la fin de la phase de réponse (point *d*). Il est important de remarquer que la fonction de communication ici utilisée retourne toujours immédiatement : même si l'esclave comporte un temps de traitement de l'opération, ce temps sera consommé entre les différentes phases, et jamais par une attente temporelle entre l'appel et le retour de la fonction.

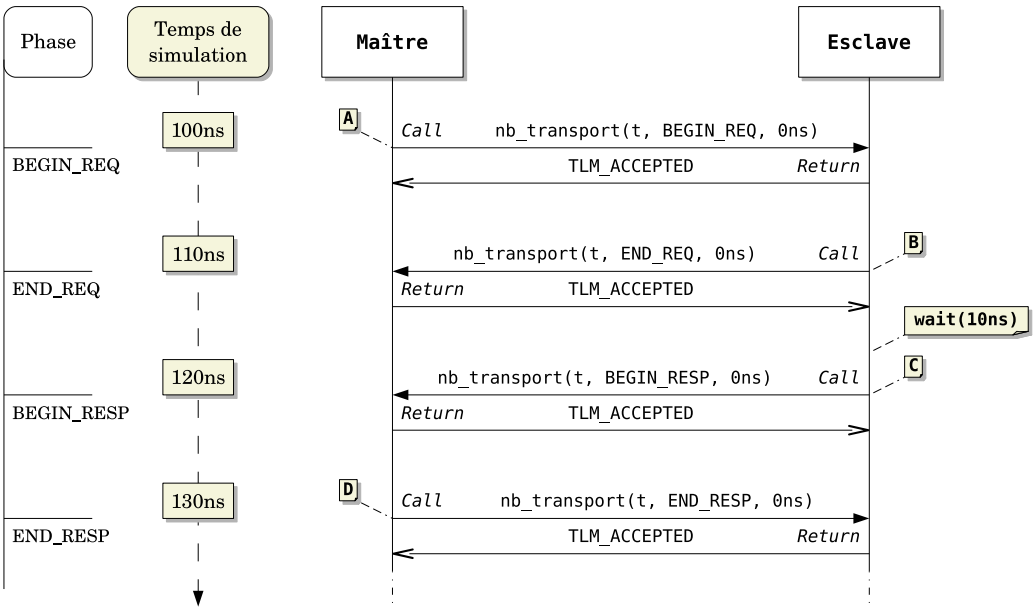

FIGURE 2.10 – Exemple en style AT

L'interface adaptée à ce style est donc l'interface de communication non-bloquante (méthodes `nb_transport_fw` et `nb_transport_bw`), qui permet de décomposer chaque transaction en plusieurs phases.

2.1.3.4 Communications bloquantes et non-bloquantes

L'interface de communication bloquante définit uniquement la méthode virtuelle pure `b_transport`. Cette méthode est toujours utilisée pour établir une communication du module source vers le module cible. Avec un paramètre dédié, chaque appel permet de spécifier une annotation temporelle, qui sera ensuite traduite en attente (appel à `wait`). Cette interface est appropriée quand le module source souhaite réaliser la transaction via un seul appel de fonction : comme son nom l'indique, l'interface entraîne le blocage du processus jusqu'au retour de la fonction `b_transport`.

L'interface de communication non-bloquante est définie par deux classes qui déclarent respectivement les méthodes `nb_transport_fw` et `nb_transport_bw` pour les acheminements du module source au module cible et vice-versa. Comme l'interface bloquante, la non-bloquante permet de spécifier des annotations temporelles. Grâce à un troisième paramètre, elle permet aussi d'indiquer la phase de la transaction en cours, comme montré par la figure 2.10. L'interface non-bloquante est appropriée quand l'utilisateur souhaite décrire de façon détaillée la séquence des interactions entre modules source et cible au cours d'une transaction : chaque appel à (et retour de) la méthode de transport peut correspondre à une transition entre deux phases de la communication.

Enfin, le premier paramètre de toute méthode de transport bloquante ou non-bloquante est la référence à un objet représentant les données de la transaction.

2.1.3.5 Données des transactions

La bibliothèque définit la classe `tlm_generic_payload`, décrivant le type de base d'une transaction. Cette classe regroupe un ensemble d'attributs, comme l'adresse et la

donnée d'une communication, ainsi qu'un ensemble de fonctions d'accès et de modification pour ces attributs. Ceci améliore l'interchangeabilité des composants, et rend possible la réutilisation de modèles où les détails précis concernant les attributs d'une transaction sont peu importants.

2.1.3.6 Découplage temporel

Le style LT permet aussi le découplage temporel, où les différents processus sont autorisés à se détacher du temps de simulation du noyau SystemC : ils peuvent avancer chacun dans une trame temporelle locale, sans faire réellement avancer le temps de simulation global, jusqu'à ce qu'ils atteignent un point où ils ont besoin de se synchroniser avec le reste du système. Ceci permet de minimiser les changements de contexte et donc d'améliorer la vitesse de simulation. La bibliothèque TLM 2.0 définit également la notion de global quantum, qui représente la quantité maximale de temps durant lequel un processus peut avancer dans sa trame temporelle locale sans être synchronisé avec le noyau.

2.1.3.7 Autres interfaces

Outre les interfaces de type transport, la bibliothèque définit l'interface d'accès direct à la mémoire (DMI pour *Direct Memory Interface*) et l'interface *debug*. La première est prévue pour rendre plus rapides les accès à la mémoire : utilisée avec le style LT, cette interface évite les *sockets* et les canaux et permet un accès direct au module cible. La deuxième est conçue pour extraire des informations en cas de modèles sans attentes temporelles et sans effets de bord. Cette interface prévoit une seule direction, depuis le module source vers la cible : la méthode `transport_dbg` retourne immédiatement et n'entraîne aucun effet de bord sur la simulation. Par exemple, elle permettrait à un module maître de connaître l'état d'une mémoire du système au cours de la simulation, uniquement pour des raisons de diagnostic et sans effectuer une véritable opération en mémoire.

Les concepts présentés jusqu'ici relèvent de la conception de systèmes matériels. Les bibliothèques SystemC et TLM offrent un standard pour la *modélisation* efficace de circuits et de plates-formes complexes à différents niveaux d'abstraction. Dans la section et dans les chapitres suivants, nous allons voir comment de tels systèmes peuvent être *vérifiés*.

2.2 Vérification par assertions

2.2.1 Spécification et vérification du comportement attendu

Dès les premières phases du flot de conception, les développeurs sont amenés à vérifier les fonctionnalités du modèle implémenté. Cette tâche est qualifiée de vérification fonctionnelle dans [Dub05] et consiste à s'assurer qu'un circuit, ou modèle de système, se comporte en accord avec un ensemble de spécifications [FKL03].

Historiquement, les ingénieurs vérifiaient l'implémentation d'un design suivant une approche de test à boîte noire (*black box*). Pour cela, une collection de jeux de tests était délivrée en entrée au modèle, décrit au niveau transfert de registres et représenté à l'aide d'un langage de description de matériel comme VHDL ou Verilog. Les sorties produites par le DUV (*Design Under Verification*) étaient ensuite comparées avec un modèle de

référence. La taille et la complexité des designs actuels rendent désormais particulièrement onéreuse l'analyse des sorties. Par ailleurs, un comportement erroné du circuit n'est pas toujours visible sur la sortie. Les modèles transactionnels permettent d'assembler des systèmes complexes à un niveau plus abstrait. Plus rapides à implémenter, ces modèles sont souvent utilisés pour l'exploration d'architecture, pour le développement anticipé du logiciel et comme modèles de référence pour la vérification des prototypes RTL. Pour cette raison, vérifier la qualité des modèles transactionnels est une tâche primordiale [EEH+07]. En accédant à la structure interne du DUV et à la communication entre ses blocs (approche à boîte blanche, ou *white box*), il est possible d'introduire des assertions afin de s'assurer du bon comportement du circuit [FKL03, CVK04].

L'ABV (*Assertion-Based Verification*, vérification d'assertions logico-temporelles) est une méthodologie s'appuyant, comme son nom l'indique, sur une spécification en termes de propriétés décrivant le comportement attendu (assertions) du design [FKL03]. Bien que souvent associée à la *vérification dynamique*, où l'observation du respect des propriétés se fait à la simulation, le terme ABV peut aussi bien être employé dans un contexte de *vérification formelle* [BFM+06]. Cette dernière, nommée aussi vérification statique, est une approche reposant sur des algorithmes et outils travaillant sur des données symboliques. Cette thèse se focalise sur la vérification durant la simulation, cependant les deux approches méritent une courte introduction.

2.2.1.1 Vérification formelle

Dans [McM93], K. L. McMillan définit la vérification formelle comme la coopération entre le modèle mathématique d'un système, un langage de spécification à la fois concis et non ambigu, et une méthode de preuve pour vérifier le respect des propriétés. La vérification formelle suit principalement deux orientations : soit une démarche déductive via des systèmes de démonstration automatisée, soit une approche algorithmique, par *model checking* (vérification du modèle) [CGP00], qui vise à prouver qu'une condition est satisfaite (ou pas) pour la totalité des entrées du système. La vérification formelle fournit donc une preuve sûre. Le principe, schématisé par la figure 2.11, comporte d'une part la traduction du design en une représentation formelle (le modèle mathématique du système), d'autre part l'écriture de sa spécification formelle (le langage non ambigu). L'outil de *model checking*, par exemple, utilise ces deux éléments pour prouver que le modèle est conforme à sa spécification, essentiellement via la mise en œuvre d'algorithmes de point fixe [EC80].

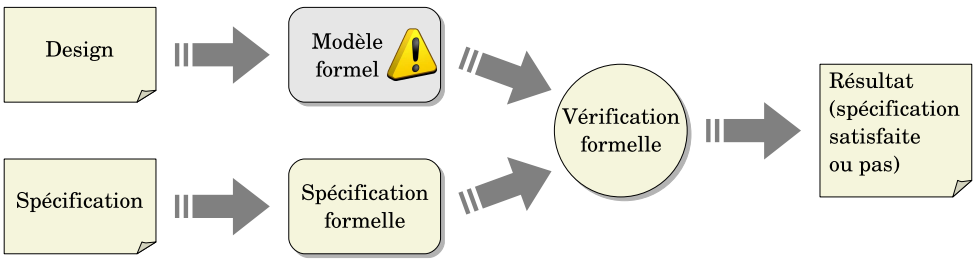

FIGURE 2.11 – Principe de vérification formelle

Dans cette approche, l'élément critique est l'extraction du modèle formel. En effet, la difficulté majeure rencontrée est l'explosion en nombre d'états à calculer : une preuve complète des assertions sur des vrais modèles RTL industriels n'est pas toujours possible.

Dans [ABF+06] on trouve une panoplie de cas d'étude, presque tous décrits au niveau transfert de registres, pour lesquels les auteurs ont étudié l'application de la spécification fonctionnelle basée sur assertions. La plupart de ces cas d'étude est reprise dans [BFM+06], où l'objectif visé devient la vérification, statique et dynamique, des propriétés énoncées. Dans les cas les plus complexes, l'application du *model checking* était limitée à des sous-blocs du design, et parfois même impossible.

2.2.1.2 Vérification durant la simulation

Comme il a été mentionné dans la section 2.1.2, la description d'un design peut-être simulée afin de reproduire et étudier son comportement. Même si la simulation ne constitue pas une représentation exhaustive du fonctionnement du design, l'analyse des transactions et des sorties produites devient rapidement très difficile, voire impossible. Dans ce contexte, l'approche de vérification par assertions consiste à équiper le DUV de propriétés décrivant le comportement attendu : utilisées en entrée des outils de vérification dynamique, ces propriétés correspondent à des contrôles tout au long de la simulation, ou menés après cette dernière, sur la trace d'exécution. Le principe de cette approche, résumé par la figure 2.12, comporte toujours l'écriture de la spécification formelle du design, ici sous la forme de propriétés logiques et temporelles, comme pour la vérification par *model checking*. Aucun modèle formel n'est requis pour le design, toutefois l'approche par simulation requiert un ensemble de jeux de test suffisamment riche et pertinent.

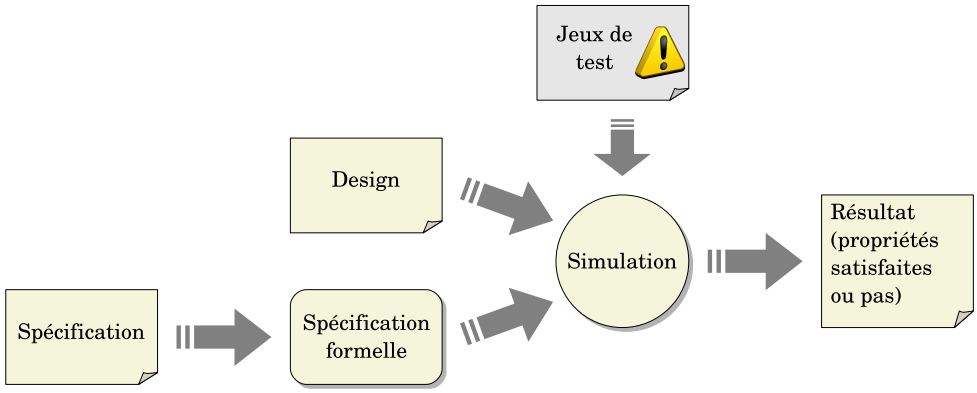

FIGURE 2.12 – Principe de vérification dynamique

La problème principal dans ce cas est la réalisation d'une simulation exhaustive, couvrant tous les comportements possibles du design. Ceci est généralement impossible. En contrepartie, la taille des systèmes ne présente pas une limitation. L'emploi d'assertions durant la simulation offre des informations précises et automatiques sur la violation ou pas de la spécification. La plupart des outils commerciaux pour la simulation au niveau transfert de registres acceptent un ou plusieurs langages pour la spécification de propriétés, comme SVA (*SystemVerilog Assertions*, cf. section 2.2.2.2) ou PSL (*Property Specification Language*, cf. section 2.2.3).

2.2.2 Tour d'horizon des formalismes existants

L'ensemble d'assertions choisi doit viser à caractériser le plus rigoureusement et intégralement le comportement attendu du modèle. Différents langages de spécification exis-

tent à l'heure actuelle, et ils s'appuient tous principalement sur des variantes ou adaptations des logiques temporelles. L'intérêt d'utiliser des langages formels à la place du langage naturel est double : d'une part, les propriétés énoncées possèdent une sémantique formelle évitant toute ambiguïté d'interprétation ; d'autre part, ces propriétés suivent une syntaxe précise et peuvent donc être acceptées en entrée par les outils de vérification.

2.2.2.1 Logiques temporelles

La logique temporelle [Pnu77, MP92] se caractérise par ces propositions dont la valeur logique dépend, entre autres, du temps. L'aspect temporel peut être vu comme une couche permettant de spécifier l'évolution, et non seulement la condition dans l'absolu ou à un moment donné. Généralement, cette couche se traduit par l'introduction de quantificateurs temporels, qui peuvent se combiner aux quantificateurs existentiel et universel, et aux connecteurs booléens. La logique temporelle constitue donc un formalisme pour décrire des séquences de transitions entre états dans un système réactif [CGP00]. Dans les logiques temporelles étudiées par la suite, le temps n'est pas évoqué explicitement : ainsi, une formule peut exprimer le fait qu'une condition se vérifiera *inévitablement* ou qu'une erreur ne se produira *jamais*.

Bien que les logiques telles que LTL (*Linear Temporal Logic*) [Pnu77] et CTL (*Computation Tree Logic*) [BAPM83] soient à la base de tout langage de spécification de propriétés moderne, leur emploi direct a été généralement abandonné en faveur de langages plus évolués (par exemple PSL et SVA), à la fois d'usage plus aisé et mieux adaptés aux besoins de la micro-électronique. Avant de présenter ces langages, il est toutefois opportun de donner un bref aperçu des logiques temporelles classiques.

CTL* CTL* [ES84], acronyme de *Computation Tree Logic* (l'étoile signifie *full*, i.e. complète), est une logique dite arborescente, dont les formules décrivent des propriétés sur des "arbres calculatoires" (*computation trees*). Elle s'appuie sur le concept de structure de Kripke[5] [CGP00], définie comme un quadruplet $M = (S, S_0, R, L)$ pour un ensemble AP de propositions atomiques, où

1. S est un ensemble fini d'états ;
2. $S_0 \subseteq S$ est l'ensemble d'états initiaux ;
3. $R \subseteq S \times S$ est une relation de transition qui doit être totale, i.e. $\forall s \in S$, $\exists s' \in S$ tel que $(s, s') \in R$
4. $L : S \mapsto 2^{AP}$ est une fonction d'étiquetage, qui associe à chaque état un ensemble de propositions atomiques de AP vraies dans cet état.

Un arbre calculatoire s'obtient en partant d'un état initial, puis en déroulant la structure à l'infini en appliquant la relation de transition, avec l'état choisi comme racine. Chaque nœud dans l'arbre correspond à un état dans la structure. Pour un nœud donné, ses fils sont obtenus par tous les états atteignables avec une transition dans la structure. La figure 2.13 montre un exemple de structure de Kripke (partie *a*) et de l'arbre qui lui est associé (partie *b*).

Un *chemin* dans une structure M, à partir d'un état s, est une séquence infinie d'états $(s_0, s_1, s_2 \ldots)$ telle que

[5] Un graphe représentant les transitions entre états, proposé par Saul Kripke dans les années 60.

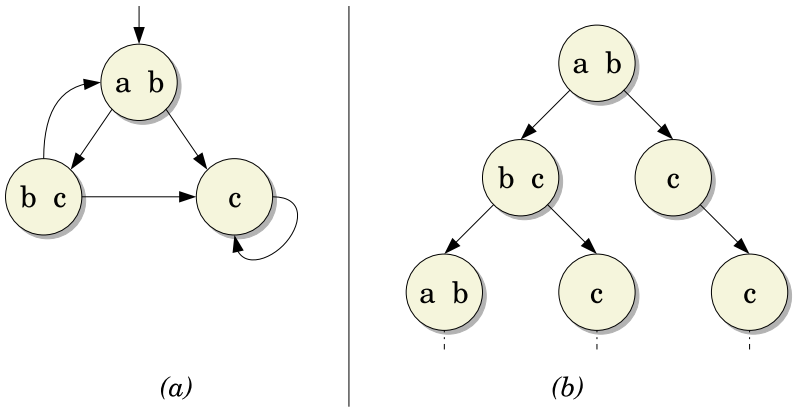

FIGURE 2.13 – Exemple de structure de Kripke et arbre de calcul associé

1. $s = s_0$;
2. $\forall i \geq 0$, $(s_i, s_{i+1}) \in R$.

Dans CTL*, les formules utilisent des quantificateurs de chemin et des opérateurs temporels. Ces formules combinent des prédicats atomiques issus de AP et emploient les connecteurs logiques classiques de la logique du premier ordre.

Les deux quantificateurs de chemin, **A** (universel) et **E** (existentiel), sont utilisés pour indiquer respectivement que, à partir d'un état donné, *tous les chemins* ou *au moins un chemin* respectent une propriété donnée. Les opérateurs temporels caractérisent une propriété le long d'un chemin donné.

Dans CTL* existent deux types de formules : les formules d'état, qui sont vraies dans un état spécifique, et les formules de chemin, qui sont vraies le long d'un chemin spécifique. La syntaxe des formules d'état est définie comme suit [CGP00] :

- si $p \in AP$, alors p est une formule d'état ;
- si f et g sont deux formules d'état, alors $\neg f$, $f \vee g$ et $f \wedge g$ sont aussi des formules d'état ;
- si f est une formule de chemin, alors $\mathbf{E}\, f$ et $\mathbf{A}\, f$ sont des formules d'état.

Deux règles additionnelles sont nécessaires pour définir la syntaxe des formules de chemin :

- si f est une formule d'état, alors f est aussi une formule de chemin ;
- si f et g sont deux formules de chemin, alors $\neg f$, $f \vee g$, $f \wedge g$, $\mathbf{X}\, f$, $\mathbf{F}\, f$, $\mathbf{G}\, f$ et $f\, \mathbf{U}\, g$ sont aussi des formules de chemin.

Toute formule d'état ou de chemin est une formule CTL*. Les symboles **X**, **F**, **G** et **U** sont des opérateurs temporels dont la sémantique est présentée par la suite. De façon informelle, on reconnaît

- **X** ("*next*" en anglais) : la propriété doit être vraie dans le deuxième état du chemin ;
- **F** ("*eventually*" en anglais) : la propriété devra être vraie dans au moins un état du chemin ;
- **G** ("*always*" en anglais) : la propriété doit être vraie dans tout état du chemin ;
- **U** ("*until*" en anglais) : opérateur binaire indiquant que la propriété droite doit être vraie dans un état s_i du chemin et que la propriété gauche doit être vraie jusqu'à s_i.

La satisfaction d'une formule d'état f dans un état s de la structure de Kripke M est notée $M, s \models f$. De même, la satisfaction d'une formule de chemin g le long du chemin π de la structure de Kripke M est notée $M, \pi \models g$. En considérant les formules d'état f_1 et

f_2, les formules de chemin g_1 et g_2 et en notant le suffixe d'un chemin π depuis sont état s_i par π^i, la relation $\models$ est définie de façon inductive comme suit [CGP00] :

1. $M, s \models p \quad \Leftrightarrow \quad p \in L(s)$
2. $M, s \models \neg f_1 \quad \Leftrightarrow \quad M, s \not\models f_1$
3. $M, s \models f_1 \vee f_2 \quad \Leftrightarrow \quad M, s \models f_1$ ou $M, s \models f_2$
4. $M, s \models f_1 \wedge f_2 \quad \Leftrightarrow \quad M, s \models f_1$ et $M, s \models f_2$
5. $M, s \models \mathbf{E}\, g_1 \quad \Leftrightarrow \quad \exists \pi = (s, s' \ldots)$ tel que $M, \pi \models g_1$
6. $M, s \models \mathbf{A}\, g_1 \quad \Leftrightarrow \quad \forall \pi = (s, s' \ldots),\ M, \pi \models g_1$
7. $M, \pi \models f_1 \quad \Leftrightarrow \quad \pi = (s_{0,}, s_1 \ldots)$ et $s = s_0$ et $M, s \models f_1$
8. $M, \pi \models \neg g_1 \quad \Leftrightarrow \quad M, \pi \not\models g_1$
9. $M, \pi \models g_1 \vee g_2 \quad \Leftrightarrow \quad M, \pi \models g_1$ ou $M, \pi \models g_2$
10. $M, \pi \models g_1 \wedge g_2 \quad \Leftrightarrow \quad M, \pi \models g_1$ et $M, \pi \models g_2$
11. $M, \pi \models \mathbf{X}\, g_1 \quad \Leftrightarrow \quad M, \pi^1 \models g_1$
12. $M, \pi \models \mathbf{F}\, g_1 \quad \Leftrightarrow \quad \exists k \geq 0$ tel que $M, \pi^k \models g_1$
13. $M, \pi \models \mathbf{G}\, g_1 \quad \Leftrightarrow \quad \forall i \geq 0,\ M, \pi^i \models g_1$
14. $M, \pi \models g_1\, \mathbf{U}\, g_2 \quad \Leftrightarrow \quad \exists k \geq 0$ tel que $M, \pi^k \models g_2$ et $\forall j,\ 0 \leq j < k,\ M, \pi^j \models g_1$

En réalité, les opérateurs $\neg$, $\vee$, $\mathbf{X}$, $\mathbf{U}$ et $\mathbf{E}$ suffisent pour exprimer toute autre formule CTL* :

- $f \wedge g \equiv \neg(\neg f \vee \neg g)$
- $f \rightarrow g \equiv \neg f \vee g$
- $\mathbf{F}\, f \equiv true\, \mathbf{U}\, f$
- $\mathbf{G}\, f \equiv \neg \mathbf{F}\, \neg f$
- $\mathbf{A}\, f \equiv \neg \mathbf{E}\, \neg f$

Dans le modèle de la figure 2.13, depuis l'état racine, les deux propriétés $\mathbf{EX}\, b$ et $\mathbf{A}\,(a\, \mathbf{U}\, c)$ sont vraies, tandis que la propriété $\mathbf{AG}\, c$ ne l'est pas.

Il existe deux sous-logiques de CTL* communément étudiées et souvent utilisées par le passé : l'une est la logique temporelle de branchement CTL et l'autre est la logique temporelle linéaire LTL. La différence entre ces deux logiques réside dans leur façon de considérer et traiter les branches de l'arbre de calcul sous-jacent [EH86].

CTL Dans la logique temporelle de branchement, les formules sont des formules d'état et les quantificateurs dénotent les chemins possibles depuis un état donné. En CTL, chacun des opérateurs temporels ($\mathbf{X}$, $\mathbf{F}$, $\mathbf{G}$ et $\mathbf{U}$) doit être précédé directement par un quantificateur de chemin. Plus précisément, CTL est un sous-ensemble de CTL* qui s'obtient en restreignant la syntaxe des formules de chemin selon la règle suivante : si f et g sont des formules d'état, alors $\mathbf{X}\, f$, $\mathbf{F}\, f$, $\mathbf{G}\, f$ et $f\, \mathbf{U}\, g$ sont des formules de chemin.

Voici trois exemples de propriétés CTL typiques pour un système concurrent [CGP00] :

1. $\mathbf{EF}\,(\textit{démarré} \wedge \neg \textit{prêt})$: il est possible d'atteindre un état où le système a *démarré* mais n'était pas *prêt*. Cette propriété indique probablement un comportement défaillant, par conséquent le concepteur serait plutôt amené à s'assurer qu'une telle condition ne se produise jamais ;
2. $\mathbf{AG}\,(\textit{requête} \rightarrow \mathbf{AF}\, confirmation)$: il est toujours vrai que si une *requête* a lieu, alors il y aura inévitablement une *confirmation*, dans tous les futurs possibles ;
3. $\mathbf{AG}\,(\mathbf{EF}\, \textit{redémarrage})$: d'un état quelconque il est toujours possible d'atteindre un état de *redémarrage* dans au moins un futur possible.

Par exemple, dans un design erroné, dont certains comportements possibles sont montrés par l'arbre de calcul de la figure 2.14, un état d'erreur correspondant à la première propriété serait effectivement atteignable.

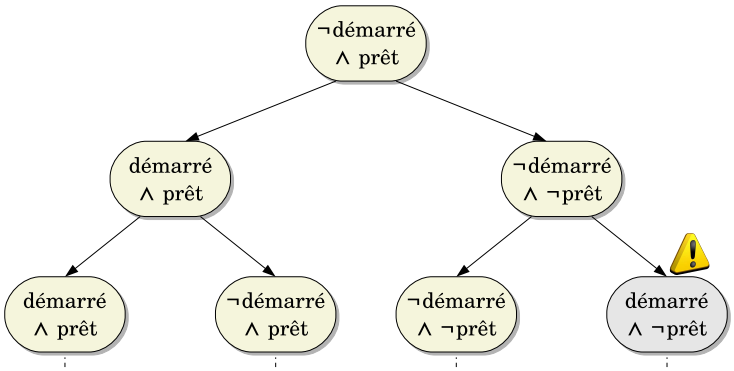

FIGURE 2.14 – Exemple d'arbre qui satisfait la propriété 1

CTL est plus communément utilisée pour exprimer des propriétés à vérifier dans un contexte statique, par *model checking* [McM93]. Parmi les nombreux outils de *model checking* existants, Bandera[6] [CDH+00], Cadence SMV[7] [McM00] et NuSMV[8] [CCG+02] acceptent CTL en entrée.

LTL Les opérateurs de la logique temporelle linéaire décrivent des événements le long d'un unique chemin. Les formules LTL sont de la forme $\mathbf{A}\, f$, où f est une formule de chemin dans laquelle les seules sous-formules d'état autorisées sont des propositions atomiques. Intuitivement, la présence systématique du quantificateur $\mathbf{A}$ peut être considérée comme une exploration identique de tous les chemins : le raisonnement se fait donc sur un seul chemin. Ainsi, dans l'écriture des propriétés, ce quantificateur est fréquemment omis.

Formellement, une formule LTL se caractérise comme suit :
- si $p \in AP$, alors p est une formule de chemin ;
- si f et g sont deux formules de chemin, alors $\neg f$, $f \vee g$, $f \wedge g$, $\mathbf{X}\, f$, $\mathbf{F}\, f$, $\mathbf{G}\, f$ et $f\, \mathbf{U}\, g$ sont aussi des formules de chemin.

Deux exemples de formules LTL sont

1. $\mathbf{G}\,(\mathit{requête} \rightarrow \mathbf{F}\, \mathit{confirmation})$: il est toujours vrai que si une *requête* a lieu, alors il y aura inévitablement une *confirmation*.
2. $\mathit{demande}\, \mathbf{U}\, \mathit{réponse}$: la *demande* continue (au moins) jusqu'à la *réponse*.

Les formules de la logique LTL sont plus communément associées à la vérification dynamique, c'est-à-dire en cours de simulation, où, typiquement, un seul chemin est en cours d'évaluation. Le pouvoir d'expression des deux logiques, CTL et LTL, n'est pas équivalent [EH86]. Toutefois, celui de LTL paraît approprié et suffisant dans un contexte de vérification dynamique.

[6] Un *model checker* pour logiciels concurrents en Java (`http://bandera.projects.cis.ksu.edu`).
[7] Un *model checker* symbolique qui étend celui de la Carnegie Mellon university (`http://www.kenmcmil.com/smv.html`).
[8] Un *model checker* symbolique *open source* (`http://nusmv.fbk.eu`).

2.2.2.2 SVA

Les *SystemVerilog Assertions* (SVA) sont partie intégrante de SystemVerilog [SV005, SV009], langage unifié de description, spécification et vérification de matériel, conçu comme un ensemble d'extensions au standard IEEE 1364 Verilog [VER01]. SVA a été développé pour permettre de décrire et spécifier les comportements complexes d'un design, cela de façon claire et concise, en gardant la même syntaxe que SystemVerilog. Les assertions peuvent être définies soit comme unités de vérification externes liées au DUV, soit comme spécifications embarquées directement dans le code du design. Le standard définit de façon formelle la syntaxe et la sémantique du langage d'assertions.

Une propriété SVA comporte un ensemble de propositions atomiques qui caractérisent des conditions pouvant se vérifier à des moments précis, dans certains cycles d'horloge. Ces conditions peuvent être combinées à l'aide des opérateurs booléens classiques, comme la conjonction et la négation. Ensuite, elles sont englobées dans des séquences qui en décrivent l'évolution dans le temps : il s'agit d'une forme plus développée des expressions rationnelles [Kle56, Aho90], suivant un concept similaire à celui des SEREs (*Sequential Extended Regular Expressions*) dans PSL (cf. section 2.2.3).

Parmi les langages de spécification de propriétés modernes, SVA constitue la principale alternative à PSL, langage présenté en section 2.2.3 et adopté dans le cadre de cette thèse. Comme nous le verrons par la suite, SVA inclut la notion de variable locale [HLMS02, OH02], qui s'avère extrêmement précieuse dans certaines circonstances. Malheureusement cette notion est absente dans PSL[9].

SVA comporte deux types d'assertions : immédiates et concurrentes. Une assertion immédiate est une instruction SystemVerilog contenant le test d'une expression booléenne. Il s'agit de la commande `assert(bool)`, également présente dans VHDL et SystemC.

Assertions concurrentes Les assertions concurrentes décrivent des comportements qui se prolongent dans le temps. Elles constituent une alternative aux logiques temporelles présentées en section 2.2.2.1.

À la différence des assertions immédiates, le modèle d'évaluation s'appuie sur une horloge : l'assertion concurrente est calculée uniquement en présence d'un top d'horloge (*clock tick*). Les valeurs des variables utilisées sont des valeurs échantillonnées. Cette suite de valeurs forme ce qu'on appelle une trace. Ainsi, le résultat de l'évaluation est prévisible et indépendant du mécanisme interne du simulateur ou de l'outil de vérification. Tout comportement entre fronts d'horloge, lié au temps ou aux événements, est ignoré. Ce type de sémantique basée sur l'horloge est incorporée dans celle des assertions concurrentes.

La *séquence* est l'épine dorsale de la plupart des assertions SystemVerilog. Sa forme la plus simple décrit des comportements linéaires : il s'agit d'une liste finie d'expressions booléennes dans un ordre temporel croissant. La première expression booléenne est évaluée au premier top d'horloge de la trace considérée, la deuxième est évaluée au deuxième top d'horloge et ainsi de suite, jusqu'à la dernière (si les expressions précédentes étaient vraies). L'opérateur de base pour exprimer ce genre de comportements est la concaténation, noté `##`, éventuellement suivi par un entier qui indique le décalage en nombre de cycle. La figure 2.15 montre une trace de simulation qui satisfait la séquence $S_1 =$ `req ##2 gnt` sur la trace $s_0 s_1 s_2$.

[9] Les variables locales ont été introduites dans la nouvelle norme PSL, parue en 2010. Un discussion à ce propos sera proposée dans le chapitre 5.

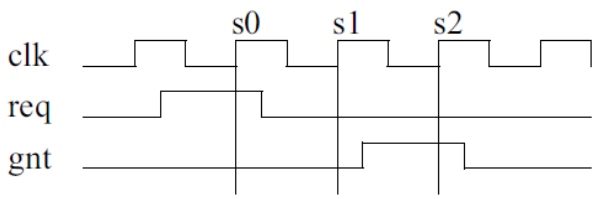

FIGURE 2.15 – Trace de simulation satisfaisant la séquence S_1

Bien évidemment, des comportement séquentiels plus complexes peuvent être décrits, ceci à l'aide des opérateurs additionnels suivants :

- `[* ]` : répétition consécutive d'une séquence ;
- `[= ]` et `[-> ]` : répétitions non consécutives ;
- `and` : conjonction de séquences ;
- `or` : disjonction de séquences ;
- `throughout` : condition le long d'une séquence ;
- `within` : séquence contenue dans une autre ;
- `intersect` : intersection entre séquences, c.-à-d. conjonction avec longueur égale des deux opérandes ;

Une propriété SVA est constituée d'une ou plusieurs séquences. Pour lier les séquences, en plus de la conjonction et de la disjonction, il est possible d'utiliser la négation et l'implication, recouvrante ou pas (`|->` et `|=>` respectivement).

SVA s'appuie donc grandement sur le concept d'expression rationnelle. Si cette syntaxe est considérée intuitive par certains utilisateurs, elle peut également constituer un point négatif. En effet, l'écriture de séquences et d'expressions rationnelles, plus le fait que ces expressions sont liées à l'horloge, rend SVA particulièrement adapté à un contexte RTL, mais visiblement moins approprié dans un contexte TLM, où les notions de temps et d'horloge peuvent être absentes. Par ailleurs, en dépit de sa richesse de constructions, et bien que son emploi puisse être adapté à d'autres langages de description de matériel, le fonctionnement de SVA reste fortement lié à celui de SystemVerilog [HW05].

Variables locales Dans certaines situations, les valeurs des expressions dans une assertion doivent être mémorisées de façon indépendante alors que plusieurs tentatives d'évaluation de l'assertion coexistent. Par exemple, lors de la vérification de designs en pipeline, chaque nouvelle donnée qui entre dans le flot nécessite d'être mémorisée dans une variable indépendante afin de pouvoir être comparée avec le résultat en sortie du flot [Dou].

Le standard IEEE de 2005 [SV005] introduit la notion de variable locale à une propriété ou à une séquence. Par exemple, la séquence suivante

```
int x;
(valid_in, x = data_in) |=> (data_out == (x*2));
```

permet de vérifier que toute donnée en sortie d'un bloc de calcul est égale à deux fois la valeur en entrée. Si cette séquence est vérifiée à chaque top d'horloge (par exemple incluse dans un block *always* synchrone de SystemVerilog), alors chaque fois que `valid_in` est vrai, la valeur en entrée est mémorisée dans une nouvelle copie de la variable locale `x`. Au top d'horloge suivant, la donnée en sortie (`data_out`) doit être égale à deux fois la valeur mémorisée.

Cependant, la sémantique des variables locales dans [SV005] est dépourvue d'une définition précise de portée. En effet, la déclaration elle-même des variables n'est pas

prise en compte dans les règles sémantiques, et une variable "déclarée" dans une séquence peut en réalité en sortir (cf. fonctions *sample*, *block* et *flow* de [SV005]). Dans le standard de 2009 [SV009], la sémantique a été complétée : une règle associée à la déclaration de variables locales a été ajoutée. Nous reviendrons sur ces notions dans le chapitre 5.

Avec la notion de variable locale, celle de *thread* contribue à reproduire le phénomène de réentrance en SVA, où plusieurs évaluations concurrentes d'une même assertion peuvent se superposer et coexister avec des valeurs différentes dans ses paramètres.

En pratique, quand une même séquence admet plusieurs traces qui en modélisent le comportement, plusieurs *threads* SystemVerilog peuvent être démarrés, chacun associé à une trace possible. Chaque *thread* qui reconnaît la séquence effectue alors une notification distincte. L'opérateur `first_match`, appliqué à une séquence, est utilisé pour ne reporter que la première reconnaissance et pour ignorer toutes celles suivantes.

Examples Les trois exemples qui suivent offrent un aperçu des propriétés qui peuvent s'exprimer en SVA.

```
property prop1;
  @(posedge clk) a |-> (b ##1 c ##1 d);
endproperty
```

La propriété élémentaire `prop1` indique le fait que, si `a` est vrai au début de la trace, alors
- `b` doit être vrai au même cycle d'horloge puis
- `c` devra être vrai au cycle suivant enfin
- `d` à celui d'après.

La variante `always ( a |-> (b ##1 c ##1 d) )` déclenche la vérification de la séquence à droite de l'implication chaque fois que `a` est vrai.

```
property prop2;
  @(posedge clk) a[*2] |-> ((##[1:3] c) and (d |=> e));
endproperty
```

La propriété `prop2` peut se traduire comme suit : le fait que `a` soit vrai une deuxième fois implique que
- `c` le soit (au moins une fois) dans les trois cycles qui suivent et que
- si `d` est vrai, alors `e` le soit au cycle suivant.

```
property prop3;
  int x;
  (valid_in, x = pipe_in) |-> ##5 (pipe_out1 == (x+1));
endproperty
```

La propriété `prop3` montre un exemple d'utilisation d'une variable locale : la valeur qui entre dans le pipeline (`pipe_in`) quand `valid_in` est vrai est mémorisée dans la variable `x` de façon à ce que la valeur calculée (ici `x+1`) soit comparée à la valeur en sortie du pipeline (`pipe_out1`) cinq cycles plus tard (`##5`). Puisque la propriété ne comporte pas directement d'expression avec horloge, elle devra être utilisée dans un bloc SystemVerilog synchronisé [SV005].

2.2.2.3 OVL

La *Open Verification Library* (OVL) [FLT06] offre une interface pour la validation des designs indépendante du langage de modélisation utilisé. Il s'agit d'une bibliothèque d'assertions prédéfinies, prévues pour la vérification de composants caractérisés par des

comportements spécifiques et pour un certain nombre de cas de figure récurrents. Elle n'est pas équivalente à une logique temporelle et ne constitue pas un langage d'assertions, comme SVA ou PSL. Introduite pour la première fois en 2001 [FC01], elle est actuellement un standard Accellera[10] [OVL10].

Chaque assertion OVL est implémentée par un module dédié à sa vérification, et comporte plusieurs versions décrites avec des langages différents. Les assertions OVL sont partitionnées selon cinq classes :

1. assertions caractérisant des comportements à l'aide de la logique combinatoire ;
2. assertions portant sur le cycle d'horloge courant ;
3. assertions portant sur une transition entre le cycle courant et le prochain cycle d'horloge ;
4. assertions caractérisant des comportements avec une longueur donnée de cycles d'horloge ;
5. assertions caractérisant des comportements entre deux événements.

Par exemple, le module `ovl_one_hot` vérifie que la valeur d'une expression comporte un seul bit avec une valeur positive. Elle fait partie de la deuxième classe.

2.2.2.4 Autres langages et méthodes

Langage *e* Le langage de vérification fonctionnelle *e* est un langage de programmation à application spécifique, destiné à automatiser la tâche de vérification matérielle (et logicielle) d'un design vis-à-vis de sa spécification. Son emploi principal est lié à la modélisation d'environnements de vérification, comportant la génération de jeux de tests et de scénarios complexes [Kir03]. Afin d'évaluer la qualité du design et celle de l'environnement de vérification lui-même, des métriques de couverture sont disponibles.

À présent, *e* réunit un certain nombre de concepts appartenant à des domaines distincts. Parmi ces concepts figurent :

- la possibilité de décrire des blocs concurrents et hiérarchiques, à l'instar de VHDL et Verilog ;
- la présence d'un langage temporel inspiré de LTL (cf. section 2.2.2.1) ;
- un paradigme orienté objet, avec support pour la programmation par aspects[11].

Produit par Verisity en 1996, le langage *e* faisait partie d'une chaîne d'outils pour la vérification fonctionnelle. Il constitue actuellement un standard IEEE [VLE08].

OVM La *Open Verification Methodology* (OVM) est une initiative industrielle issue de Mentor Graphics et Cadence Design Systems [OVM]. Il s'agit d'une bibliothèque de classes pour la construction modulaire d'environnements de vérification, où les composants peuvent communiquer via des interfaces transactionnelles, dans une infrastructure TLM [SC07]. Les classes de la bibliothèque permettent la création de bancs de test, l'enregistrement et l'analyse des résultats de simulation. OVM naît de la fusion de Mentor Graphics AVM 3.0 et Cadence URM 6.2, disponibles au cours de l'année 2006.

[10] *Accellera Organization*, un organisme de normalisation à but non lucratif (`http://www.accellera.org`).

[11] Paradigme de programmation qui permet de séparer les considérations techniques (*aspects* en anglais) des descriptions métier dans une application. Voir [KLM+97] pour plus de détails.

UVM La *Universal Verification Methodology* (UVM) est une bibliothèque proposée par Accellera et inspirée d'OVM. Comme celle-ci, UVM comporte un ensemble de classes pour la génération automatique de jeux de tests, l'analyse des résultats et de la qualité de l'environnement de vérification [Acc10].

2.2.3 PSL : *Property Specification Language*

PSL, acronyme de *Property Specification Language*, trouve ses racines dans Sugar [FMW05], un langage développé depuis 1994 par le laboratoire de recherche de IBM Haifa. Sugar avait été créé comme "sucre syntaxique" pour la logique temporelle CTL, afin de permettre l'expression de propriétés complexes de manière plus concise et aisée. La version 1.0 du langage fut ensuite donnée à Accellera comme candidate pour le choix d'un langage de spécification standard, visant la vérification basée sur assertions. Trois autres langages concouraient à la définition du standard : CBV de Motorola [FHLM02], le langage *e* de Versity (cf. section 2.2.2.4) et ForSpec d'Intel [AFF+02]. Suite à un long procédé de sélection, qui retint dans un premier temps CBV et Sugar, la version 2.0 de ce dernier fut choisie. Cette version comportait à la fois le sous-ensemble doté d'une sémantique LTL et une extension pour CTL (cf. section 2.2.2.1). PSL est un standard IEEE depuis 2005 [PSL05].

2.2.3.1 Un aperçu du langage

De même que SVA (cf. section 2.2.2.2), PSL vise à fournir les moyens pour spécifier les propriétés d'un design, en utilisant une syntaxe concise, rigoureuse et dotée d'une sémantique clairement définie. Le langage permet ainsi d'énoncer des propriétés de spécification matérielle à la fois relativement simples à lire et précises du point de vue logique/mathématique. Un collection d'opérateurs et de mots-clés issus du langage courant (anglais) favorisent l'écriture des propriétés moins élaborées. Par exemple, la traduction PSL de "chaque fois que *condition* est vraie, alors, à l'instant suivant, *expression* doit se produire" est particulièrement intuitive : `always ( condition -> next expression)`.

Structure La structure d'une formule PSL est organisée selon quatre couches, chacune visant un aspect dans la spécification de la propriété. La collaboration des quatre couches permet d'énoncer des propriétés particulièrement riches et complexes.

La couche **booléenne** est à la base de toute propriété. Elle est utilisée pour obtenir les expressions exploitées par les autres couches. Toute expression appartenant à cette couche est vérifiée en un seul cycle d'évaluation, et utilise les opérateurs booléens classiques, comme la conjonction et la négation.

La couche **temporelle** utilise les expressions de la couche booléenne comme opérandes. Elle est au cœur du langage et permet de décrire des relations temporelles complexes, vérifiées sur un ou plusieurs cycles d'évaluation. Par exemple, elle permet d'exprimer le fait qu'une condition se vérifiera *avant* une autre ou qu'une erreur ne se produira *jamais*.

La couche de **vérification** fournit les directives qui indiquent comment utiliser la propriété, en exigeant par exemple le respect de cette propriété tout au long de la simulation (mot-clé `assert`) ou par les vecteurs de test en entrée au simulateur (mot-clé `assume`). Ces directives sont généralement interprétés par l'outil de vérification.

La couche de **modélisation** peut être employée pour contraindre les entrées du design ou pour décrire le comportement de variables et composants auxiliaires, n'appartenant pas forcément à l'architecture mais également utiles à l'expression de la propriété. La gestion d'un compteur pour mémoriser le nombre d'occurrences de la satisfaction d'une condition en est un exemple.

Les propriétés sont organisées à l'intérieur d'unités de vérification nommées *vunit*. Chaque *vunit* peut contenir plusieurs propriétés, ainsi que des instructions de la couche de modélisation. Ces dernières sont communes à toutes les propriétés de l'unité. Par exemple, un compteur déclaré dans une *vunit* sera partagé par toutes ses propriétés. La figure 2.16 montre un exemple de coopération des quatre couches.

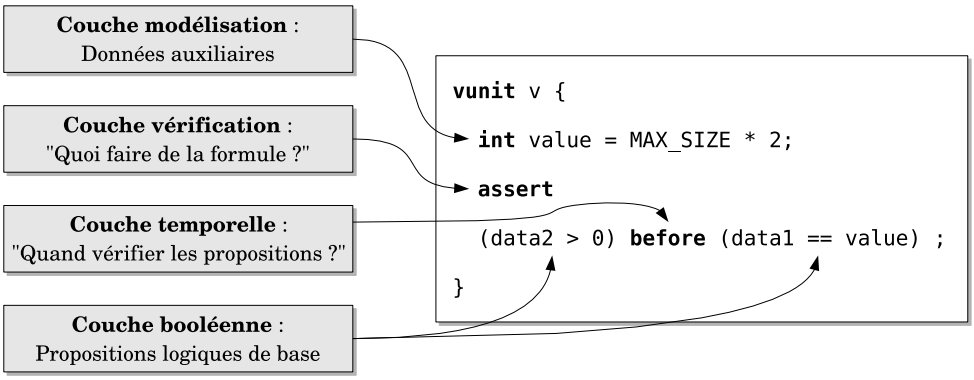

FIGURE 2.16 – Exemple de coopération des quatre couches

En PSL, les couches booléenne et modélisation empruntent la syntaxe et le potentiel d'expression du langage de description de matériel sous-jacent. Plus précisément, cinq formes, appelées *flavors*, sont actuellement prévues et elles sont associées respectivement à SystemVerilog, Verilog, VHDL, SystemC, et GDL. PSL n'est donc pas lié à un seul langage de modélisation. La figure 2.16 est un exemple de spécification avec la syntaxe de SystemC.

Classes d'opérateurs Si les quatre couches définissent la structure syntaxique du langage, deux classes d'opérateurs permettent d'organiser la sémantique de PSL en fonction des besoins de vérification. La classe FL, acronyme de *Foundation Language*, comporte tous les opérateurs avec une sémantique de trace, comparable à celle de la logique LTL présentée dans la section 2.2.2.1. Les propriétés de la logique linéaire étant adaptées à la vérification dynamique, nous nous focaliserons uniquement sur la classe FL par la suite. Cette classe inclut aussi la sous-classe d'opérateurs SERE, acronyme de *Sequential Extended Regular Expression*, analogue aux séquences de SVA.

La classe OBE, pour *Optional Branching Extension*, ajoute les opérateurs de la logique de branchement CTL, avec les quantificateurs indispensables pour raisonner sur les différents chemins possibles dans un arbre de calcul. Par exemple, cette classe d'opérateurs est parfois nécessaire pour exprimer des propriétés qui portent sur les inter-blocages (*deadlocks*).

Sous-ensemble simple Même en se restreignant à la classe FL, il est possible d'exprimer des propriétés qui ne peuvent pas être évaluées facilement (voire pas du tout) en cours de simulation. À l'aide d'un ensemble de contraintes syntaxiques, PSL définit

aussi un sous-ensemble de propriétés qui est conforme à la notion d'avancement monotone du temps et qui est donc proche du concept de simulation. Ce ensemble est nommé *sous-ensemble simple* et ses contraintes excluent la nécessité d'un "retour en arrière" dans l'évaluation de la formule.

Par exemple, considérons la trace de simulation de la figure 2.17 et rappelons la propriété P_1 énoncée en début de section : `always ( condition -> next expression)`.

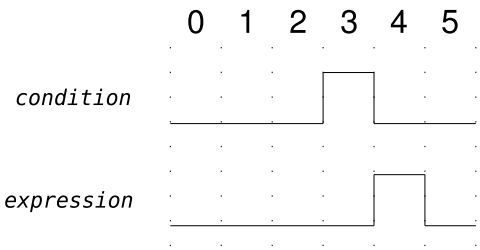

FIGURE 2.17 – Exemple de trace qui satisfait P_1

Au fur et à mesure que le temps avance, il est possible de vérifier si *condition* est vraie. Puisque c'est le cas à l'instant 3, l'outil de vérification dynamique devra contrôler la présence de *expression* à l'instant suivant. Ainsi, la propriété est satisfaite sur la trace de la figure.

Considérons maintenant la propriété P_2 : `(eventually! condition) -> next expression` ainsi que la trace de simulation de la figure 2.18.

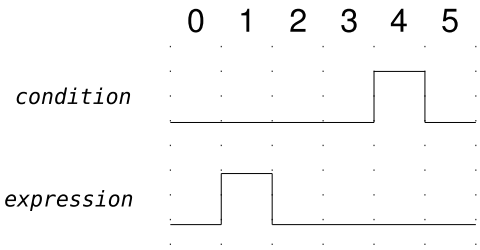

FIGURE 2.18 – Exemple de trace qui satisfait P_2

Dans ce cas, l'identification du moment précis où l'opérande gauche de l'implication est vrai rend plus problématique l'évaluation de la propriété : la sous-formule `(eventually! condition)` est effectivement satisfaite sur la trace de la figure 2.18, depuis son premier instant, mais cette information n'est connue qu'à l'instant 4. Par conséquent, un "retour en arrière" dans la simulation serait nécessaire afin de vérifier que *expression* était vraie au deuxième instant (instant 1) de la trace, ce qui est exact. Quelques règles du sous-ensemble simple sont définies dans la section 2.2.3.3. La liste complète est disponible dans [PSL05].

2.2.3.2 SEREs

Comme indiqué dans la section précédente, PSL contient aussi la sous-classe d'opérateurs SERE pour décrire des comportements séquentiels à l'instar de SVA. Cette classe comporte tous les opérateurs habituels des expressions rationnelles, comme la concaténation et la répétition. Nous ne détaillerons pas la sémantique de ces opérateurs, puisque

aucun opérateur SERE n'a été employé dans le cadre de cette thèse. En effet, leur usage s'est avéré inadapté au contexte TLM. Les détails de ce choix seront exposés au cours des deux chapitres suivants.

La propriété P_5 = `{a; b; c} |=> {d[+]; e}` est un exemple d'utilisation des SEREs en PSL. Elle exprime le fait que "si a est vrai à l'instant présent et b à l'instant suivant et c à celui d'après, alors on devra observer une répétition non vide de d suivie par e". La figure 2.19 montre deux exemples de traces qui satisfont P_5, tandis que la figure 2.20 montre une trace qui viole P_5.

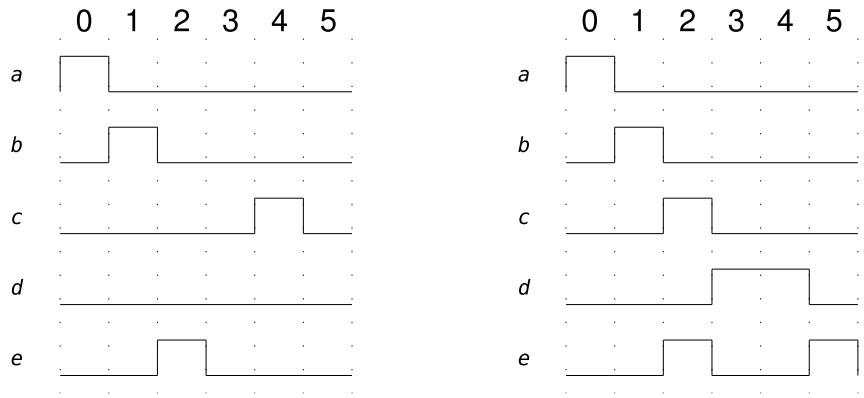

FIGURE 2.19 – Deux traces satisfaisant la propriété P_5

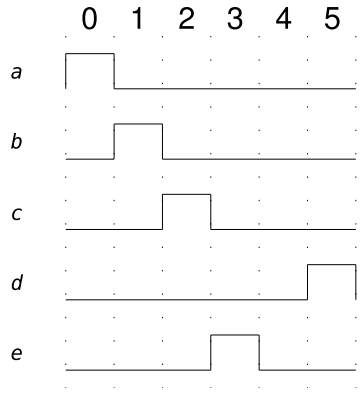

FIGURE 2.20 – Une trace qui viole la propriété P_5

2.2.3.3 Sémantique

La sémantique de PSL définit précisément quand une propriété est satisfaite par une *trace* donnée, aussi appelée *mot* ou *comportement* [PSL05]. Dans la vérification dynamique sont considérées uniquement les traces de longueur finie.

Même si la vérification dynamique considère les traces de longueur finie, la sémantique de PSL est définie par rapport au concept universel de trace d'exécution, soit-elle finie ou infinie. Une trace est un mot sur un alphabet $\Sigma = 2^P \cup \{\top, \bot\}$, où

- P est un ensemble de propositions atomiques ;
- $\top$ est un élément tel que, pour toute expression booléenne b, $\top$ satisfait toujours b ;
- $\bot$ est l'élément opposé à $\top$, i.e. pour toute expression booléenne b, $\bot$ ne satisfait jamais b.

Un mot se compose de lettres, chacune identifiant un cycle ou instant d'évaluation précis. Le mot $v = (\ell_0\ell_1\ell_2...\ell_n)$ est de longueur $n+1$, dénotée par $|v|$. Le $i^{ème}$ instant d'évaluation sur le mot v est indiqué par v^{i-1}, et $v^{i..}$ indique le suffixe de la trace v commençant à v^i. Dans l'exemple montré en figure 2.17, la trace finie se compose de 6 lettres, associées aux 6 instants d'évaluation. La progression des lettres se juxtapose à la progression temporelle : chaque lettre identifie une combinaison de propositions atomiques. Par exemple, la première lettre (ℓ_0) contient les propositions ($\neg condition, \neg expression$), tandis que la quantrième (ℓ_3) contient ($condition, \neg expression$). Dans la trace d'exécution d'un système décrit au niveau transfert de registres et synchronisé sur les fronts montants d'une horloge c, chaque lettre correspondrait à l'échantillonnage lors d'un front montant de c. Toujours dans l'exemple de la figure 2.17, la quatrième lettre, ℓ_3, satisfait *condition* mais pas *expression*. On note alors $\ell_3 \vDash_B condition$ et $\ell_3 \nvDash_B expression$, où $\vDash_B$ est la relation de satisfaction pour les booléens.

Plus précisément, pour chaque lettre $\ell \in 2^P$, pour toute proposition atomique $p \in P$ et pour toutes expressions booléennes b_1 et b_2 [PSL05] :

1. $\ell \vDash_B p \Leftrightarrow p \in \ell$
2. $\ell \vDash_B \neg b_1 \Leftrightarrow \ell \nvDash_B b_1$
3. $\ell \vDash_B true$ et $\ell \nvDash_B false$
4. $\ell \vDash_B b_1 \wedge b_2 \Leftrightarrow \ell \vDash_B b_1$ et $\ell \vDash_B b_2$

La sémantique des formules FL sur des traces est définie de manière inductive, en utilisant comme cas de base la sémantique des expressions booléennes sur les lettres de Σ, donnée ci-dessus. Parmi tous les opérateurs et les constructions FL, seulement ceux utilisés dans les exemples et dans les cas d'études proposés par la suite seront détaillés. Pour plus de détails, se référer à [PSL05]. Dans les règles qui vont suivre, les notations suivantes sont adoptées :

- v dénote un mot (trace) de longueur quelconque, finie ou infinie ;
- b est une expression booléenne, obtenue par un prédicat atomique ou par combinaison de prédicats atomiques et expressions booléennes, en utilisant les opérateurs booléens usuels ;
- φ et ψ sont des formules FL ;
- le symbole $\vDash_F$ indique la relation de satisfaction pour les formules FL : l'écriture $v \vDash_F \varphi$ signifie que v satisfait (ou modélise) φ ;
- $k, j \in \mathbb{N}$.

Un premier opérateur clé est le *until* fort, noté **until!** ou **U**. Il requiert que l'opérande gauche soit vrai jusqu'à production impérative de l'opérande droit. Sa sémantique s'exprime comme suit (formule 2.1) :

$$v \vDash_F \varphi \textbf{ until! } \psi \Leftrightarrow v \vDash_F \varphi \,\mathbf{U}\, \psi \Leftrightarrow \exists k < |v| \text{ tel que } v^{k..} \vDash_F \psi \text{ et } \forall j < k,\ v^{j..} \vDash_F \varphi \quad (2.1)$$

Dans le sous-ensemble simple, φ peut être de nature FL ou booléenne, mais ψ est restreint au type booléen uniquement. La version faible de l'opérateur *until* n'impose pas que l'opérande droit devienne satisfait. Dans ce cas on devra observer indéfiniment l'opérande gauche (formule 2.2).

$$\varphi \textbf{ until } \psi \overset{def}{=} \varphi \,\mathbf{W}\, \psi \overset{def}{=} (\varphi \,\mathbf{U}\, \psi) \vee \mathbf{G}\, \varphi \quad (2.2)$$

Pour certains opérateurs, la distinction entre version forte et faible n'a aucun sens. Un exemple est fourni par l'opérateur *always*, dont l'unique version est définie comme suit (formule 2.3) :

$$\texttt{always}\ \varphi \stackrel{def}{=} \mathbf{G}\,\varphi \stackrel{def}{=} \neg(true\,\mathbf{U}\,\neg\varphi) \tag{2.3}$$

La figure 2.21 fournit un exemple de trace qui modélise la propriété P_3.

$P_3 =$ `always(a -> (b until! c))`

L'opérateur *always* déclenche la vérification de l'implication à chaque instant de la trace. Par conséquent, la présence de a au deuxième instant demande à ce que b soit vrai jusqu'à l'arrivée de c, ce qui est respecté. L'implication est déclenchée deux autres fois aux instants 7 et 8. La présence de b jusqu'à l'arrivée de c à l'instant 8 permet de valider l'implication démarrée à l'instant 7, et la présence de c à l'instant 8 valide celle démarrée à 8. D'après la sémantique de l'opérateur *until*, b n'est pas nécessaire à l'instant 8.

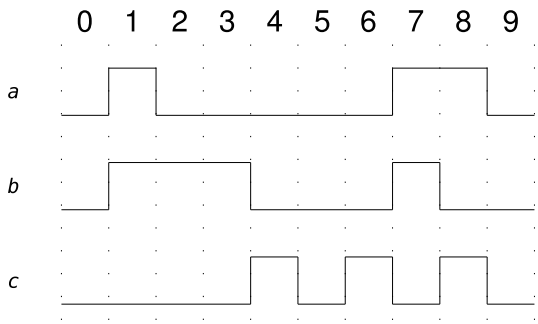

FIGURE 2.21 – Exemple de trace satisfaisant la propriété P_3

L'opérateur *next* fort requiert que son opérande soit satisfait au prochain instant d'évaluation de la trace d'exécution, instant qui doit exister dans la trace (formule 2.4).

$$v \models_F \texttt{next!}\ \varphi \Leftrightarrow |v| > 1 \text{ et } v^{1..} \models_F \varphi \tag{2.4}$$

La présence d'un prochain instant d'évaluation n'est pas obligatoire dans la version faible, qui s'exprime comme $\neg$ `next!` $\neg\varphi$.

L'opérateur *eventually* n'a qu'une variante forte et demande à ce que son opérande soit satisfait au moins une fois, à un instant quelconque de la trace (formule 2.5). Dans le sous-ensemble simple, son opérande doit être un booléen.

$$\texttt{eventually!}\ \varphi \stackrel{def}{=} \mathbf{F}\,\varphi \stackrel{def}{=} true\,\mathbf{U}\,\varphi \tag{2.5}$$

L'opérateur *next_event* requiert la satisfaction de son deuxième opérande la prochaine fois que la condition booléenne dénotée par le premier opérande sera vraie (formules 2.6 et 2.7).

$$\texttt{next_event!}(b)(\varphi) \stackrel{def}{=} \neg b\,\mathbf{U}\,(b \wedge \varphi) \tag{2.6}$$

$$\texttt{next_event}(b)(\varphi) \stackrel{def}{=} \neg b\,\mathbf{W}\,(b \wedge \varphi) \tag{2.7}$$

$$\text{avec } v \models_F b \wedge \varphi \Leftrightarrow v \models_F b \text{ et } v \models_F \varphi \tag{2.8}$$

Un exemple de trace qui satisfait la propriété P_4 est montrée par la figure 2.22.

$P_4 =$ `a -> (next! next_event!(a)(b))`

L'opérateur *always* n'étant pas utilisé, la vérification de l'opérande droit de l'implication n'est activée que par la présence de a au premier instant de la trace. Par ailleurs, l'emploi de l'opérateur *next* permet d'exclure l'instant présent dans la vérification de la sous-formule `next_event!(a)(b)`, car la vérification de l'opérande droit de `->` est déclenchée dès que l'opérande gauche est satisfait.

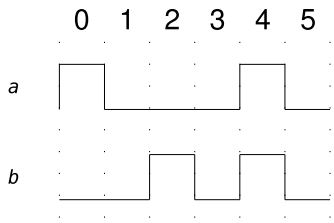

FIGURE 2.22 – Exemple de trace satisfaisant la propriété P_4

La sémantique de l'opérateur *before*, encore une fois exprimée en fonction du *until*, impose la vérification de l'opérande gauche avant celle de l'opérande droit (formules 2.9 et 2.10). Dans le sous-ensemble simple, ses deux opérandes sont restreints au type booléen.

$$\varphi \ \textbf{before!} \ \psi \stackrel{def}{=} \neg\psi \, \mathbf{U} \, (\varphi \wedge \neg\psi) \tag{2.9}$$

$$\varphi \ \textbf{before} \ \psi \stackrel{def}{=} \neg\psi \, \mathbf{W} \, (\varphi \wedge \neg\psi) \tag{2.10}$$

Enfin, la négation d'une formule FL (formule 2.11) utilise la notion de dual $\bar{v}$ d'une trace v, où toutes les occurrences de l'élément $\top$ sont remplacées par $\bot$ et vice versa. Toutefois, le sous-ensemble simple n'autorise que la négation d'expressions booléennes.

$$v \models_F \neg\varphi \Leftrightarrow \bar{v} \not\models_F \varphi \tag{2.11}$$

Les règles énoncées ci-dessus suggèrent beaucoup d'éléments communs avec les logiques temporelles décrites dans la section 2.2.2.1. D'autres opérateurs temporels existent en PSL : **abort**, **never**, **next_a**, **next_e**, **next_event_a**, **next_event_e**. La conjonction, la disjonction et l'implication entre formules FL ont la sémantique habituelle des opérateurs logiques, bien que la valeur logique des opérandes soit obtenue en appliquant la relation de satisfaction $\models_F$. Dans le sous-ensemble simple, l'opérande gauche de l'implication doit être un booléen, et la disjonction n'admet qu'un seul opérande FL à la fois. Aucune limitation ne s'applique à la conjonction.

2.2.3.4 Niveaux de satisfaction d'une formule

PSL définit quatre niveaux de satisfaction qui caractérisent les quatre états possibles de toute propriété durant la simulation :

- *Satisfaite fortement* (*Holds strongly*) : aucun état d'erreur n'a été rencontré, toutes les obligations futures pour la satisfaction de la propriété ont été rencontrées, la propriété sera satisfaite sur toutes les extensions du chemin d'exécution. Il s'agit d'un état définitif, car toute propriété satisfaite fortement le restera et ne pourra plus être violée.

- *Satisfaite* (*Holds*) : comme la condition précédente, sauf que la propriété peut être satisfaite ou pas sur toutes les extensions de la trace.
- *En attente* (*Pending*) : aucun état d'erreur n'a été rencontré, mais les obligations futures n'ont pas encore été rencontrées et la propriété peut être satisfaite ou pas sur les extensions du chemin d'exécution. Cela signifie que la propriété n'a pas été explicitement violée, néanmoins si la trace de simulation s'arrête en ce moment, alors la propriété ne pourra pas être considérée comme satisfaite.
- *Échouée* (*Fails*) : un état d'erreur a été rencontré et la propriété ne sera satisfaite sur aucune extension du chemin d'exécution.

Durant la simulation, ces quatre niveaux fournissent des informations précieuses au-delà de la simple alerte de violation. Dans le chapitre 5 nous verrons comment ces informations sont exploitées par l'extension de la sémantique proposée.

En PSL ils existent des versions fortes et faibles de certains opérateurs, et cette distinction est liée au concept de *satisfaction forte* [Sis94, Eis07]. Typiquement, la version forte d'un opérateur impose que la condition de terminaison soit inévitablement atteinte dans la trace ; la contrepartie faible ne possède pas cette obligation. PSL permet d'indiquer syntaxiquement qu'un opérateur est dans sa version forte via l'ajout du symbole " **!** " après le mot-clé. Par exemple, l'opérateur *until* des logiques temporelles vues en section 2.2.2.1 existe dans ces deux formes : version forte `until!` et version faible `until`. Ainsi, la propriété `expression until! condition` exige que *condition* se produise inévitablement et que *expression* soit vraie jusqu'à la. La propriété `expression until condition` requiert que *expression* soit vraie jusqu'au moment où *condition* le sera et que, si la dernière ne se produit jamais, alors la première soit toujours vraie.

Avec la sémantique donnée dans la section 2.2.3.3, la relation entre opérateurs forts et satisfaction forte peut être énoncée comme suit : pour toute formule FL φ, si φ_{strong} est la transformation de φ où tout opérateur a été substitué par sa version forte, alors la propriété φ est satisfaite *fortement* par la trace finie v si et seulement si φ_{strong} est satisfaite par v. Par exemple, `next_event(cond)(expr1 until expr2)` est satisfaite fortement par une trace v si et seulement si $v \models$ `next_event!(cond)(expr1 until! expr2)`. Aucune trace d'exécution finie ne peut satisfaire fortement une propriété du type `always` φ (une extension de la trace pourrait toujours violer la propriété). Sur une trace infinie, les deux interprétations de satisfaction forte et faible coïncident [Eis07].

2.3 Bilan

Nous pouvons remarquer que la définition de PSL, tout comme celle de SVA, est particulièrement adaptée à des descriptions de niveaux portes ou RTL, où l'on considère, sur les fronts d'horloge, les valeurs des signaux de la description.

Dans le cas de systèmes complexes décrits au niveau TLM, la vérification va se concentrer sur les communications entre les blocs du système [EEH+06a, HT06], vues de façon très abstraite. Dans le chapitre suivant nous étudierons un certain nombre d'approches de vérification de descriptions SystemC. Nous adopterons ensuite un point de vue selon lequel, avec ses opérateurs temporels intuitifs et avec l'abstraction qu'opère sa sémantique vis-à-vis du temps de simulation, PSL offre un moyen efficace pour spécifier les propriétés au niveau transactionnel.

Les travaux menés dans cette thèse proposent des solutions basées sur ce concept : adapter et étendre PSL pour offrir un moyen simple et efficace de réaliser des vérifications sur des plates-formes TLM.

Chapitre 3

Travaux sur la vérification de descriptions SystemC

Introduction

L'objectif du chapitre 2 était d'offrir au lecteur une introduction à la fois à SystemC TLM et à la vérification par assertions. Les deux concepts ont été présentés de manière relativement indépendante. Dans ce chapitre nous allons étudier un certain nombre d'approches de vérification où l'ABV est appliquée à SystemC.

Comme expliqué dans les sections 2.2.2.1, 2.2.2.2, 2.2.3.3, la vérification de propriétés de la logique temporelle se réalise sur des représentations des exécutions du système, symboliques, ou traces effectives. En RTL, les lettres qui composent une trace peuvent facilement être obtenues grâce à l'horloge : chaque front identifie un instant d'échantillonnage, et donc une lettre de la trace (cf. sections 2.2.2.2 et 2.2.3.3 par exemple). Puisqu'en TLM la notion d'horloge, ainsi que celle de temps de simulation, peuvent être absentes, l'échantillonnage doit être considéré différemment. En premier lieu, si la vérification visée est statique, il faut parvenir à un modèle formel pertinent, comme introduit dans la section 2.2.1.1. Si le mode de vérification est dynamique, il faut parvenir à des traces de simulation pertinentes pour les assertions énoncées, cela même quand la notion de temps de simulation est complètement absente. En second lieu, puisque les modèles transactionnels se veulent moins précis mais plus rapides à développer et à simuler, la solution de vérification doit être réellement applicable, pas trop pénalisante en temps de calcul/simulation et en mémoire. Enfin, la question qui se veut primordiale est la caractérisation, et donc l'énoncé au niveau transactionnel, des propriétés de la logique temporelle.

Bien que le contexte de la solution de vérification que nous proposons par la suite soit dynamique, nous allons d'abord étudier quelques approches formelles intéressantes avant de nous concentrer sur les méthodes appliquées à la simulation.

3.1 Approches statiques

Große et Drechsler L'une des premières approches formelles pour la vérification fonctionnelle de descriptions SystemC est présentée dans [GD03]. La méthode vise à prouver le respect de propriétés de la logique temporelle LTL dans les circuits séquentiels synchrones, décrits en SystemC au niveau portes logiques. Après calcul des fonctions de

sortie et de transition de la machine à états finie représentant le circuit, une analyse des états atteignables[1] est réalisée. Puis, la vérification d'une formule LTL est traduite en un problème décisionnel (SAT), en utilisant des diagrammes de décision binaire (BDD) pour les fonctions de sortie et de transition, ainsi que pour l'ensemble des états atteignables. Cette approche considère uniquement les descriptions au niveau portes logiques, très loin du niveau transactionnel. La preuve de certaines propriétés, comme celle de vivacité (*liveness*) proposée, atteint rapidement des temps de calcul prohibitifs. Un détail intéressant dans cette approche est la surcharge de la méthode `end_of_elaboration()` par tous les composants de base (porte `AND`, `NOT`...) : les auteurs exploitent (la fin de) l'étape d'élaboration SystemC, présentée en section 2.1.2.3, pour identifier toutes les créations et les interconnexions de ces composants, et pour réunir donc les éléments du modèle.

Habibi et Tahar La solution étudiée dans [HT05] s'intéresse à la vérification de formules PSL au niveau transactionnel de SystemC. La méthode comporte deux flots en parallèle, l'un pour le design et l'autre associé aux propriétés PSL. La conception des modèles SystemC, ainsi que celle des propriétés fonctionnelles qui les caractérisent, commence par des spécifications UML [OMG09]. Dans le flot dédié au design, les diagrammes de cas d'utilisation, de classe et de séquence sont utilisés pour décrire le système (son comportement et sa structure) durant les premières étapes de conception. Le flot de spécification des propriétés s'appuie sur une version modifiée des diagrammes de séquence, qui met en jeu plusieurs éléments, parmi lesquels

- les opérateurs temporels tels que *always*, *eventually* et *until* ;
- des opérateurs de séquence comme *next* ou *prev* ;
- l'horloge ;
- le retard, en nombre de cycles, d'activation des méthodes.

La modélisation graphique, ici envisagée grâce à UML, offre une aide précieuse à la conception, en particulier durant ses premières phases. Les diagrammes de séquence se focalisent sur les échanges de messages entre les objets du système décrit et, par conséquent, s'avèrent adaptés au niveau TLM. La volonté de représenter les propriétés PSL par ce même formalisme présente un double intérêt : d'une part, elle permet d'avoir une description uniforme pour le système et pour ses propriétés ; d'autre part, elle vise une expression plus aisée des formules de la logique temporelle. Cependant, le standard UML [OMG09] ne possède pas les constructions adaptées pour représenter, de façon directe, les opérateur de la couche temporelle de PSL [BHA10]. Pour cette raison, les auteurs sont amenés à enrichir les diagrammes des éléments mentionnés ci-dessus. L'inconvénient principal de cette solution est l'utilisation d'un langage de modélisation non standard, où la sémantique formelle de PSL n'est pas garantie. Dans [HT05] et [HT06] les auteurs ne détaillent pas la sémantique des annotations introduites, mais soulignent le fait que les diagrammes UML ne permettent pas d'exprimer intégralement et précisément toutes les propriétés, ni d'opérer une traduction automatique vers PSL.

Les descriptions UML du système et des propriétés sont transformées en machines à états abstraites ASM (*Abstract State Machine*), décrites en AsmL[2]. Ce modèle est alors

[1] *Reachability analysis* : une analyse de chaque état dans le circuit pour vérifier s'il est atteignable depuis l'état initial (ou le *reset*). Plus de détails concernant le travail de Drechsler et Große à ce sujet sont présentés dans [DG02]

[2] *Abstract State Machine Language*, un langage de spécification exécutable de Microsoft (`http://research.microsoft.com/en-us/projects/asml`)

utilisé pour générer une machine à états finie du système. Durant la phase de transformation du modèle AsmL vers la FSM (*Finite State Machine*), l'ensemble des propriétés PSL est vérifié par *model checking*. Il s'agit d'une étape caractérisée par la limitation récurrente des approches statiques, comme introduit dans la section 2.2.1.1. En général, la construction d'un automate n'est pas réalisable pour l'intégralité du système : le modèle obtenu est une sous-approximation. La description SystemC du système est produite à partir du modèle ASM, en appliquant un ensemble de règles afin de préserver les propriétés du code ASM d'origine. De même, les propriétés PSL sont traduites en modules C# depuis leur contrepartie ASM. Cela permet de réaliser de l'ABV dynamique par simulation conjointe du code SystemC et des modules C#. Il s'agit d'un étape avec un intérêt prédominant à nos yeux, car elle est proche de la démarche que nous proposons dans le chapitre 4. Toutefois l'absence de détails concernant la construction des modules C# ne nous permet pas d'étudier leur fonctionnement et la façon dont ils sont connectés au design SystemC. En outre, les résultats reportés sont restreints à des descriptions synchronisées par horloge (ceci est également valable pour l'exemple de propriété proposé), les communications se font au moyen de signaux. Cet aspect est caractéristique des niveaux de modélisation moins abstraits, typiquement RTL ou TLM *cycle accurate* (cf. section 2.1.3.1).

Karlsson *et al.* L'approche proposée par les auteurs de [KEP06] consiste à traduire le modèle SystemC en une représentation basée sur les réseaux de Petri [Mur89]. Les propriétés, exprimées en CTL, sont vérifiées par *model checking* sur la nouvelle représentation. Comme pour les travaux précédemment cités, ce type d'approche entraîne des limitations dues à la complexité du modèle à vérifier. La représentation employée, nommée PRES+, est une extension de la notion de réseau de Petri classique et présente les caractéristiques suivantes :

1. les jetons propagés ont une valeur et une estampille temporelle ;
2. les transitions ont une fonction associée qui permet de calculer la nouvelle valeur du jeton propagé ;
3. les transitions sont dotées d'un délai temporel variable et qui peut être nul ;
4. chaque place peut contenir au plus un jeton, le modèle est ainsi défini sûr (la présence d'un jeton dans une place de sortie d'une transition désactive la transition) ;
5. les transitions peuvent avoir des conditions de garde.

La représentation PRES+ obtenue depuis la description SystemC initiale est soumise, avec les propriétés temporelles souhaitées, à l'outil de *model checking* UPPAAL[3] pour sa vérification [CEP00]. Pour obtenir le modèle PRES+, tout élément du système doit être traduit, y compris l'ordonnanceur SystemC. A titre d'exemple, un ordonnanceur capable de gérer deux processus et deux signaux comporte 21 places (dont 10 internes et 11 en interface), 16 transitions et plusieurs dizaines de flèches liant les places aux transitions. Chaque instruction de base est représentée par une place et une transition : la transition effectue l'instruction associée et la place en permet simplement l'exécution lorsqu'elle contient un jeton. L'exemple de la figure 3.1 montre la traduction PRES+ du programme élémentaire "`xy`".

Les transitions t_1 et t_2 correspondent respectivement aux instructions des lignes 1 et 2. Par exemple, la transition t_1 propage un jeton avec la valeur 3 dans la place associée

[3] Un environnement pour la modélisation et la vérification de systèmes temps-réel décrits par des réseaux d'automates temporels (`http://www.uppaal.org`)

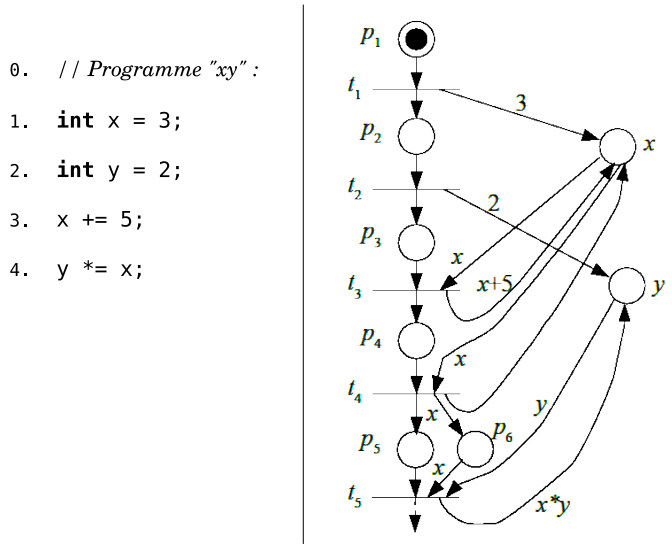

FIGURE 3.1 – Réseau PRES+ pour le programme “xy” (source : [CEP00])

à la variable x. Elle propage également un jeton dans p_2 afin de permettre l'exécution de l'instruction suivante. Quand une instruction utilise la valeur d'une variable précédemment introduite, le modèle doit au préalable récupérer cette valeur : c'est le rôle de la transition t_4 par exemple. Toute place correspondant à une variable active doit toujours contenir un jeton. Toutes les transitions qui lisent la valeur d'une variable doivent comporter une flèche qui retourne à cette variable, si la variable doit rester utilisable par la suite. Le niveau de détail d'un tel modèle est donc très élevé : outre la surveillance des communications entre modules, il permet de raisonner à propos des instructions et des variables à l'intérieur des méthodes. Il est par exemple possible de s'assurer qu'une valeur interdite n'est jamais affectée à une variable donnée. Bien que soumise aux limitations des techniques de vérification statique, l'approche décrite dans [KEP06] est séduisante : la sémantique du réseau de Pétri étant originellement indépendante de la notion de temps et d'horloge, l'approche permet de traiter réellement les modèles transactionnels les plus abstraits. Malheureusement, la taille des réseaux de Petri obtenus est relativement rédhibitoire. Par exemple, chaque signal engendre un sous-système d'environ 20 places et 12 transitions. Dans le cas d'étude du *Packet Switch* délivré avec la distribution SystemC de OSCI, la vérification par *model checking* de certaines propriétés requiert plusieurs heures en présence de 2 maîtres et 2 esclaves. Parmi les différents cas d'étude que nous avons menés, ce même *Packet Switch* a également été traité. Dans la section 7.1.4 du chapitre 7, nous verrons que la vérification de la propriété “tous les messages envoyés par un maître seront inévitablement reçus par l'esclave concerné”, est bien violée, comme le constatent les auteurs de [KEP06], mais la raison n'est pas uniquement celle qu'ils suggèrent.

Moy *et al.* Dans [MMMC06], les auteurs présentent un ensemble d'outils, appelé LusSy, pour l'extraction d'informations sur l'architecture et sur le comportement d'un design SystemC transactionnel, et pour la réalisation d'abstractions sur ce design et la construction d'un modèle intermédiaire, nommé HPIOM (*Heterogeneous Parallel Input/Output Machines*). Cette représentation intermédiaire peut être soumise à différents outils de *model*

checking, comme SMV, Lesar[4] ou NBac[5] . Plus précisément, une première étape consiste à extraire toutes les informations nécessaires du programme SystemC, en utilisant le *front-end* PINAPA[6]. Ce dernier réalise l'extraction de l'arbre syntaxique abstrait du programme, l'extraction des informations architecturales et la liaison entre syntaxe C++ et architecture SystemC. Le module BISE (*Back-end Independent Semantic Extractor*) se charge ensuite de transformer la sortie produite par PINAPA vers le formalisme d'automates synchrones communicants HPIOM. L'idée de base est d'avoir un automate par processus, plus un automate pour chaque élément SystemC TLM. Des techniques d'abstraction et d'optimisation du modèle HPIOM obtenu sont nécessaires pour tenter de lutter contre le problème de l'explosion en nombre d'états du système. Le flot global de LusSy est élaboré : plusieurs modules interviennent afin d'analyser et optimiser le modèle SystemC initial. Il est cependant nécessaire de remarquer que les assertions traitées ne sont pas exprimées avec un langage de spécification comme PSL ou SVA : dans [MMMC06], uniquement un ensemble restreint de propriétés est évoqué, comme la recherche d'interblocages (*deadlocks*) globaux ou la vérification de non terminaison d'un processus.

Blanc *et al.* Les efforts principaux dans la plupart des solutions de vérification statique portent sur l'obtention et l'optimisation du modèle formel. L'outil SCOOT [BKS08] met également l'accent sur l'extraction d'un tel modèle intermédiaire à partir de descriptions SystemC. Les auteurs soulignent la possibilité de re-synthétiser le code C++ à partir de la représentation intermédiaire de SCOOT, en atteignant ainsi des meilleures performances à la simulation. La technique d'extraction du modèle, vérifié ensuite par *model checking*, est introduite dans [KS05]. Afin d'optimiser l'étape de vérification, les auteurs préconisent un partitionnement automatique entre les parties logicielles et les parties matérielles du design, cela en analysant la nature des processus SystemC. Trois types de processus sont ainsi identifiés :

1. les *processus combinatoires* sont sensibles à toutes les entrées mais pas à l'horloge, ils ne comportent pas d'appels à la méthode `wait` et ne contiennent pas de boucles non bornées ;
2. les *processus avec horloge* sont sensibles à un front d'horloge, ils ne comportent pas d'appels avec paramètres à la méthode `wait` et, à l'intérieur des boucles non bornées, ils doivent contenir un appel à `wait` ;
3. les *processus sans restrictions* sont identifiés comme des parties logicielles.

Les processus combinatoires sont transformés en formules mathématiques et effacés du modèle. Les processus sans restrictions sont gardés inaltérés. Seul les processus avec horloge, considérés comme étant la vraie partie matérielle, sont transformés en machines à états, en utilisant la notion de structure de Kripke (cf. section 2.2.2.1) étiquetée. Puisque seulement une machine à états par processus matériel est extraite, le modèle ainsi obtenu est considérablement réduit et optimisé pour l'analyse formelle. La fonction de transition de chaque structure de Kripke est définie en fonction de l'état du processus (cf. sections 2.1.2.1 et 2.1.2.3). Cette approche se limite donc aux niveaux d'abstraction les plus bas, elle n'est pas adaptée au contexte TLM sans horloge et temps de simulation. Le cas d'étude considéré dans [KS05] comporte un module UART (*Universal Asynchronous*

[4] `http://www-verimag.imag.fr/The-Lustre-Toolbox.html`
[5] `http://pop-art.inrialpes.fr/people/bjeannet/nbac/index.html`
[6] `http://greensocs.sourceforge.net/pinapa`

Receiver/Transmitter) et un deuxième module représentant un système logiciel. Afin de vérifier l'absence de perte de données lors des échanges avec l'UART, les auteurs doivent ajouter des compteurs aux processus du design et les comparer. Il s'agit d'une solution qui ne considère pas réellement les communications entre les modules, qui n'analyse pas les échanges d'information via leurs interfaces.

3.2 Approches dynamiques

Les méthodes de vérification appliquées à la simulation portent parfois le qualificatif de méthodes *semi-formelles*, puisqu'elles s'appuient toujours sur des bases mathématiques mais n'offrent pas de réponses exhaustives. Dans [Var07], l'ABV dynamique est positionnée parmi les techniques de vérification formelles. En effet, dans ce contexte, la spécification des assertions se fait toujours à l'aide d'une logique temporelle, et la surveillance des assertions peut également suivre des démarches prouvées sûres. Toutefois, le verdict pour chaque assertion n'est pas exhaustif, à moins que la simulation elle-même ne le soit, ou que l'assertion ne soit violée.

3.2.1 Premières contributions

Niemann et Haubelt Dans [NH06], les auteurs proposent une approche d'ABV dynamique pour SystemC TLM. Au lieu de considérer les transactions comme des communications incorporant des structures de données complexes, les auteurs associent un signal booléen à chaque transaction. Ce signal est positionné à vrai durant la transaction et repositionné à faux en tout autre moment. Suivant cette démarche, le code SystemC est simulé une première fois et l'évolution de tous les signaux est écrite dans un fichier en format VCD (*Value Change Dump*). Ce fichier est ensuite traduit en une description Verilog qui est enrichie d'un ensemble d'assertions SVA et simulée une deuxième fois, à l'aide d'un simulateur qui supporte les assertions concurrentes de SystemVerilog. Malheureusement, il s'agit d'une solution limitée, ne s'adaptant qu'à une catégorie restreinte de descriptions TLM. Le type de propriétés exprimables est assez exigu, car il ne considère que les relations entre les transactions mais exclut tout autre critère, comme les paramètres et les valeurs transmises. Le cas d'étude proposé fait apparaître les trois fonctions distinctes `put_red()`, `put_yellow()` et `put_green()`, gouvernant l'affichage d'un feu tricolore. Des propriétés comme l'exclusion mutuelle sont vérifiées sur les trois transactions. Toutefois, il suffirait que les fonctions soient substituées par une seule procédure (i.e. une "macro-transaction") `change_light(int color)` pour que la méthode décrite ne soit plus applicable. Par ailleurs, l'approche nécessite une double simulation : la première est utilisée pour enregistrer l'état des transactions, la deuxième pour vérifier les assertions. Ceci n'est pas économique en temps CPU. Cette solution compte néanmoins deux points intéressants : les auteurs exploitent la programmation par aspects afin d'enregistrer les transactions sans modifier le code source (cependant cette tâche est très peu détaillée) et considèrent l'ordre entre les communications dans un modèle transactionnel abstrait.

Cheung et Forin Les auteurs de [CF06] préconisent l'application de PSL à la vérification par assertions de logiciel pour systèmes embarqués. Un sous-ensemble de PSL, nommé sPSL, est défini et implémenté dans Giano, un simulateur de logiciel et applications temps-réel embarquées, dédié au co-développement matériel et logiciel [FNL06].

Un moteur d'évaluation additionnel permet d'obtenir les informations relatives à la simulation (par exemple les adresses mémoire ou le contenu des registres) ainsi que celles présentes statiquement dans le code source. L'évaluation des assertions décrites en sPSL est effectuée par ce moteur au lieu de celui dédié à la simulation du logiciel. Comme pour la solution précédente, l'aspect intéressant de cette approche est son indépendance de l'horloge de simulation : dans sPSL, la notion de "prochain top d'horloge" est remplacée par celle de "prochain événement" (un événement pourrait être l'exécution d'une instruction ou la modification d'un registre). Cependant, il s'agit d'une solution liée au simulateur de [FNL06]. L'outil accepte un certain nombre d'opérateurs issus de la couche temporelle de PSL, mais aucun élément de la couche modélisation (cf. section 2.2.3). L'approche ne se conforme pas toujours à la syntaxe ou à la sémantique de PSL.

Afin de vérifier des propriétés de la logique temporelle durant la simulation, une alternative à l'utilisation d'un simulateur dédié consiste à traduire chaque assertion en un composant appelé *moniteur*, ou *checker* en anglais. Comme indiqué dans l'introduction du présent document (cf. section 1.3.2.1), un tel composant, décrit dans le même langage que celui du design, est ensuite ajouté au code du DUV et utilisé pour surveiller l'état de l'assertion pendant la simulation. Typiquement, un moniteur est représenté par un automate.

Plusieurs techniques de vérification, statiques et dynamiques, s'appuient sur la construction d'un automate à partir d'une formule de logique temporelle [ES84, VW86]. Par exemple, la figure 3.2 montre une machine à états finie capable de reconnaître toute violation de la formule PSL énoncée en début de section 2.2.3.1 :

$P_1 =$ `always ( condition -> next expression)`

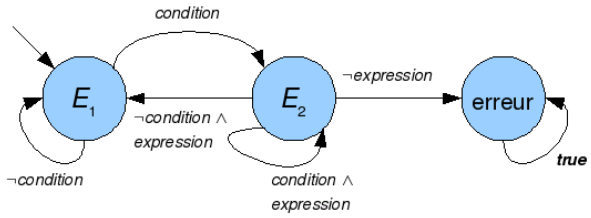

FIGURE 3.2 – Machine à états finie associée à la propriété P_1

Dans l'automate de la figure 3.2, nous pouvons observer un aperçu informel de la sémantique de l'opérateur **always**. Depuis l'état E_2, le cas $condition \wedge expression$ est explicitement séparé de $\neg condition \wedge expression$. Puisque les deux opérandes ne sont pas mutuellement exclusifs à priori, leur présence simultanée a pour double effet de maintenir la propriété vraie à l'instant présent et de demander une nouvelle évaluation de *expression* à l'instant suivant (**next**). C'est pour cela que l'état E_2 comporte une boucle sur lui-même. Nous pouvons donc affirmer que *condition* réactive toujours (**always**) la propriété.

Ruf *et al.* Une solution à base d'automates est adoptée par les auteurs de [RHKR01], où la vérification de propriétés logico-temporelles durant la simulation est effectuée à l'aide d'automates finis traduits en descriptions SystemC. La logique temporelle employée, FLTL, est une variante de LTL limitée aux intervalles de temps finis. Les automates

utilisés sont des machines à états AR (*Accepting-Rejecting*), définies par le quintuplet $(S, \rightarrow, A, R, s_0)$, où S est l'ensemble d'états avec état initial s_0, $\rightarrow$ est la fonction de transition et A et R sont respectivement les ensembles d'états qui acceptent et qui refusent la validation ($A \subset S$ et $R \subset S$). Par exemple, la partie (a) de la figure 3.3 montre l'automate AR pour la formule élémentaire a, tandis que la partie (b) de la figure montre la traduction de la formule $F_{[2,4]}a$ ([2,4] est l'intervalle pour l'opérateur *eventually*).

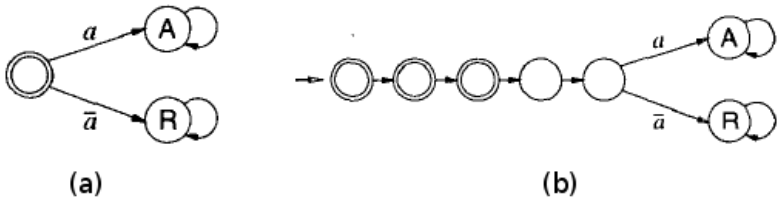

FIGURE 3.3 – Exemples d'automates AR (source : [RHKR01])

Cette approche étant l'une des premières solutions proposées pour SystemC, son application est limitée à des designs avec horloge et nombre de cycles explicites. Par ailleurs, l'implémentation des moniteurs est réalisée en exploitant des processus SystemC standard ; ces derniers encombrent le noyau de simulation et sont sujets à la même politique d'ordonnancement que les processus du design surveillé. L'ajout des propriétés se fait à l'aide de la commande `sc_assert` (il s'agit sans doute d'une version modifiée, puisque la commande originale définie dans [SC005] n'accepte que des expressions booléennes). L'instrumentation du code n'est donc pas automatisée, dans la mesure où l'utilisateur doit identifier la position exacte où insérer l'assertion dans la description SystemC. Par exemple, la propriété de réactivité "chaque fois que le signal *req* est actif, alors le signal *ack* sera inévitablement activé", mentionnée dans [RHKR01], nécessite l'identification de l'instruction où *req* reçoit la valeur *true*, instruction qui sera suivie par la commande `sc_assert(F[intervalle] ack);`. Si la condition d'activation de l'implication (ici *req*) est remplacée par un ensemble de signaux modifiés à des endroits différents dans le code source, la solution proposée par les auteurs apparaît difficilement applicable. Cependant, cette approche comporte un aspect modulaire intéressant : l'implémentation des moniteurs utilise une bibliothèque d'automates AR enrichie après synthèse de chaque nouvelle propriété. Les moniteurs sont donc compilés comme étant une bibliothèque indépendante, liée au code du DUV.

Große et Drechsler Une approche visant la traduction de propriétés logico-temporelles en SystemC est également présentée dans [GD04]. Un aspect intéressant de ces travaux est la possibilité de réutiliser les moniteurs produits même après simulation : il s'agit de modules matériels qui peuvent être synthétisés et qui sont aptes à l'instrumentation du circuit final après fabrication, offrant ainsi les mêmes avantages que la solution apportée dans [BLMA+05, BZ05, PLBN05, BLOF06]. Si la méthode de [GD04] s'applique à SystemC, elle est toutefois limitée aux descriptions de matériel les moins abstraites : les moniteurs sont exprimés en termes de registres et portes logiques et ils fonctionnent suivant les cycles d'horloge. Par ailleurs, la logique temporelle utilisée est moins élaborée que celle de PSL ou SVA. Les propriétés se composent uniquement de deux parties, une liste d'hypothèses (*assume*) et une liste d'obligations (*prove*), dont la syntaxe est toujours "`at t+i: expression;`". Par exemple, la propriété P, exprimant que "chaque fois que le

signal x a la valeur 1, alors, deux cycles d'horloges après, le signal y aura la valeur 2", se traduit par

```
theorem P is
  assume:
    at t: x = 1;
  prove:
    at t+2: y = 2;
end theorem;
```

Le temps doit toujours apparaître explicitement dans ces formules.

Hessabi *et al.* Dans [GYHG07] les auteurs proposent l'intégration d'une méthode de vérification basée sur assertions dans leur méthodologie de synthèse au niveau système, appelée ODYSSEY [GHMZ08]. Le contexte de ces travaux est relativement disjoint du nôtre puisque plus spécifique : le projet ODYSSEY s'intéresse au partitionnement matériel/-logiciel au niveau application, dans une approche basée sur ASIP (*Application-Specific Instruction-set Processor*). Cependant la nécessité de synthèse automatique des propriétés à surveiller et le caractère orienté-objet (il s'agit ici de modèles décrits en langage C++) constituent deux aspects intéressants. Dans ces travaux, les auteurs utilisent les méthodes des objets pour représenter les instructions à exécuter et ils définissent un nouveau langage d'assertions. Ce langage comporte trois couches, suivant vaguement le principe de PSL (cf. section 2.2.3). Toutefois, le langage défini est nettement plus limité que PSL :

- la couche *combinatoire* est comparable à la couche booléenne de PSL et fournit la possibilité d'exprimer les appels de méthodes ; nous avons également prévu cette possibilité, qui offre un moyen syntaxique clair pour indiquer, dans notre cas, non pas l'exécution d'une instruction mais le fait qu'une opération de communication entre les composants du SoC a lieu ;
- la couche *temporelle* comporte essentiellement des séquences d'appels de méthodes (suite d'éléments séparés par des virgules), ainsi que leur négation, conjonction et disjonction ;
- la couche *vérification* introduit une "polarité", qui peut être *always* ou *never*.

Pour identifier les séquences d'instructions correctes, la synthèse des assertions est effectuée au moyen de machines à états finies, où chaque transition vers l'état successif est empruntée si l'appel de méthode demandé a lieu.

Outre l'absence d'opérateurs temporels complexes, cette solution comporte deux lacunes : les auteurs ne définissent aucune sémantique formelle, qui constitue un élément fondamental pour l'automatisation et pour la preuve, et le langage s'appuie uniquement sur le concept de séquence. Nous estimons qu'en SystemC TLM la séquence permet difficilement d'exprimer les relations entre les transactions du système : nous reviendrons sur cette question dans la section 4.2.1.2 du chapitre suivant.

3.2.2 Travaux connexes aux nôtres

Abarbanel *et al.* L'un des premiers outils conçus pour générer des moniteurs à partir de propriétés de la logique temporelle est FoCs [ABG+00], produit par le laboratoire de recherche d'IBM à Haifa. Toutefois, dans un premier temps, la sortie produite par l'outil était limitée à VHDL. Une extension de l'outil FoCs à la génération de moniteurs en C++, visant ainsi les descriptions SystemC, est présentée dans [DGG+05]. La méthode de synthèse des propriétés utilise toujours la représentation des propriétés sous la forme

d'automates. Les moniteurs produits par FoCs comportent une méthode `transition` qui représente la fonction de transition de l'automate (cf. structure de Kripke en section 2.2.2.1), et qui effectue donc le calcul de l'état suivant. La solution proposée prévoit encore une fois la création de moniteurs synchronisés avec l'horloge, donc utilisables au niveau transfert de registres mais inadaptés à un contexte transactionnel. Le principe de synchronisation est schématisé par le squelette de code qui suit (nous rappelons que la version de SystemC est celle de 2005) :

```
// declaration d'une classe qui englobe le moniteur et qui en
//  declenche la fonction de transition sur les fronts d'horloge :
class myChecker_wrapper : public sc_module {
  public:
    // horloge du systeme :
    sc_in_clk clk;
    // instance du moniteur FoCs pour l'assertion PSL :
    myChecker * checker_inst;
    ...

    myChecker_wrapper(sc_module_name name) : sc_module(name) {
      SC_METHOD(transition);
      // la methode transition est declaree sensible a l'horloge
      //  du circuit :
      sensitive_pos << clk;
      ...
    }
    void transition() {
      // calcul du prochain etat de l'automate associe a la propriete
      //  par appel a la fonction de transition du moniteur FoCs :
      checker_inst->transition(...);
      if( ! checker_inst->getStatus() ) {
        // si la propriete n'est plus satisfaite, la violation
        //  est signalee ici...
      }
    }
    ...
};
```

Lahbib Dans sa thèse, Lahbib [Lah06] utilise l'outil FoCs pour la traduction des propriétés PSL en C++. Dans l'optique d'étendre les résultats décrits dans [DGG+05] à SystemC TLM, Lahbib redéfinit l'interface de communication TLM et utilise des classes spécifiques pour les canaux de communication. Les moniteurs concernés sont notifiés lors de chaque nouvelle transaction. Il s'agit de la première méthode réellement idoine aux niveaux les plus abstraits de TLM.

Selon Lahbib, PSL serait le langage d'assertions le mieux adapté à SystemC TLM. Cependant, il mentionne que les éléments utilisés par la bibliothèque TLM pour représenter l'abstraction des données et des communication (par exemple, le *payload* mentionné en section 2.1.3.5) sont absents dans PSL. Le langage d'assertions n'étant pas considéré comme directement applicable, le flot d'ABV dynamique doit être adapté selon les trois points suivants :

1. il y a nécessité de représenter en PSL les types propres à TLM (comme une adresse ou la transaction elle-même) avant la traduction des propriétés ;
2. le langage C++ seul ne fournit pas les éléments nécessaires à identifier les instants

d'évaluation d'une propriété PSL, par conséquent il faut déployer une nouvelle couche temporelle qui suive la politique de synchronisation de TLM ;

3. les moniteurs générés doivent pouvoir être liés aux designs TLM.

Les moniteurs TLM sont obtenus suivant le flot de génération de la figure 3.4.

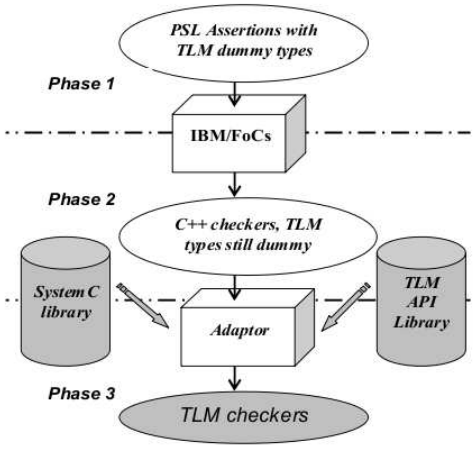

FIGURE 3.4 – Flot de génération proposé dans [Lah06]

Afin de pouvoir utiliser l'outil d'IBM, qui n'accepte que des expressions élémentaires dans la couche booléenne de PSL, les types propres à TLM sont remplacés par des types fictifs (point 1 ci-dessus) dans l'énoncé de la propriété, comme indiqué dans la première phase du flot proposé. La substitution des types temporaires par les vrais types est effectuée par un adaptateur durant la troisième phase. En réalité, puisque PSL n'est pas lié à un langage ou à un contexte précis, il n'existe pas vraiment de notion de "type supporté" dans le langage : si le type final des expressions de la couche booléenne est préservé, les éléments utilisés à l'intérieur des expressions peuvent provenir de toute autre bibliothèque, notamment SystemC et son extension TLM. À la différence du premier point, les points 2 et 3 évoqués ci-dessus constituent la véritable innovation des travaux de Lahbib. L'échantillonnage de la trace de simulation au niveau transactionnel est effectué par les transactions plutôt que par l'horloge. Toute assertion PSL φ, évaluée comme φ @*clock* en RTL, en TLM sera évaluée comme φ @*transaction* [LGH+06]. D'un point de vue technique, l'évaluation du moniteur par appel à sa fonction `transition` n'est plus effectuée lors de chaque front d'horloge, mais depuis la fonction `transport` de l'interface TLM, comme schématisé par le code suivant :

```
tlm_status TLM_IF::transport(tlm_transaction & transaction) {
  ...
  // les entrees du moniteur (elements utilises dans la propriete)
  //  sont obtenues depuis l'objet transaction :
  transaction_data & trans =
      *( (transaction_data *)(transaction.get_data()) );
  ADDRESS_TYPE addr =
      *( reinterpret_cast<ADDRESS_TYPE*>(trans.get_address()) );
  ...
  // le moniteur est evalue par appel a sa fonction de transition :
  checker.transition(addr, ...);
  if( ! checker.getStatus() ) {
    // le cas echeant, une violation est signalee
  }
```

```
}
```

Comme notre approche, cette solution englobe le moniteur dans une structure qui permet de contrôler son déclenchement, et donc l'évaluation de la propriété PSL. Toutefois, avec la méthode que nous proposons dans le chapitre 4, nous parvenons à résoudre un certain nombre de limitations. En premier lieu, la solution de [Lah06] est liée uniquement aux interfaces de la couche transport et aux éléments contenus dans le *payload* de la transaction. En deuxième lieu, si la technique est bien adaptée au niveau transactionnel le plus abstrait, elle ne convient pas aux designs qui mélangent les aspects de plusieurs niveaux d'abstraction. En particulier, elle ne convient pas aux designs hybrides, où des canaux primitifs (par exemple un `sc_signal`) et hiérarchiques coexistent. Ensuite, nos résultats expérimentaux montrent un bien meilleur impact sur le temps de simulation, cela à cause de la nature optimisée de nos moniteurs par rapport à ceux générés par FoCs. Il s'agit d'un aspect non négligeable pour une solution d'ABV dynamique qui se veut pleinement applicable aux modèles transactionnels. Enfin, la différence la plus importante de notre contribution réside dans la prise en compte des variables globales (couche modélisation de PSL) et du phénomène de réentrance, via une implémentation des variables locales. Nous approfondirons cet aspect dans le chapitre 5.

Ecker *et al.* Dans [EEH+06a, EEH+06b], Ecker *et al.* définissent un langage de spécification d'assertions visant les modèles SystemC TLM. Ce langage étend SVA (cf. section 2.2.2.2) afin d'exprimer explicitement les événements liés au contexte transactionnel. Les auteurs consacrent une attention particulière à la nature et aux relations entre les transactions. Ces dernières, souvent non atomiques, comportent un début, une fin et une durée dont les combinaisons peuvent être résumées comme montré par la figure 3.5.

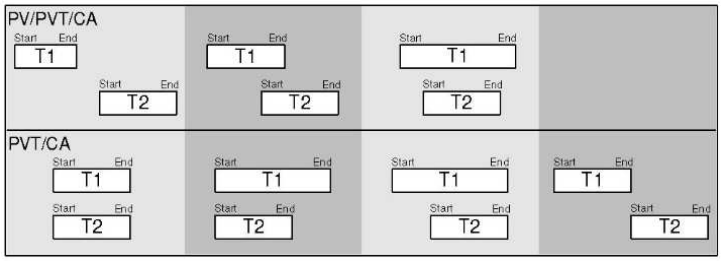

FIGURE 3.5 – Relations entre les transactions (source : [EEH+06b])

Selon les auteurs, au niveau TLM PV (cf. section 2.1.3.1), il n'est pas possible de raisonner à propos de transactions qui se vérifient simultanément, puisque chaque transaction démarre à un instant précis et unique. Le fait qu'un événement (par exemple, le début d'une transaction dans un canal) se produise à un point donné peut être spécifié uniquement par rapport à d'autres événements, plutôt que de façon absolue. Il en est de même pour la distance entre deux événements distincts. Comme pour la solution proposée dans [Lah06], Ecker *et al.* expriment le besoin d'un mécanisme de temporisation et de notification. En outre, au niveau PVT il faut pouvoir raisonner à propos de la simultanéité, en termes de temps de simulation, de deux événements. Le niveau CA, très proche de RTL, doit considérer aussi les signaux et la notion d'horloge.

Le nouveau formalisme proposé est assez riche et comporte une structure à cinq couches : événementielle, booléenne, séquentielle, propriété, assertion. Les quatre dernières couches s'inspirent amplement des standards existants en matière de langages d'assertions.

Avec la couche événementielle, les auteurs visent à pallier le problème de synchronisation, aussi évoqué dans [Lah06]. Dans ce cadre, une nouvelle notion de machine à états transactionnelle est définie, où apparaît l'ensemble E de tous les événements du design *plus* les événements de début et de fin d'une transaction. Ainsi, l'échantillonnage de la trace de simulation sur laquelle une propriété donnée est vérifiée se fait par rapport aux éléments de E. Conjonction et disjonction d'événements font partie de la couche événementielle. Toute évaluation d'une propriété est donc forcément déclenchée par un (ensemble d') événement(s). Cette démarche conduit toutefois à un suréchantillonnage qui peut pénaliser considérablement le temps de simulation. En réalité, ce suréchantillonnage n'est pas indispensable pour exprimer les relations entre transactions : la solution proposée dans [Lah06] en est un exemple.

Puisque le langage de spécification de Ecker *et al.* est inspiré de SVA, l'opérateur de séquence est encore une fois à la base de l'expression des propriétés. Selon les auteurs, son fonctionnement ne dépend pas du niveau d'abstraction, puisque son activation dépend de la définition des événements qui composent la trace, mais celle-ci est suréchantillonnée. Par ailleurs, la séquence seule ne permet pas de représenter la complexité des comportements du design, mais doit être combinée avec les modes d'évaluation d'une propriété (par exemple *Restart* ou *Overlap*). En PSL ces concepts sont naturellement inclus dans la sémantique de chaque opérateur. Par exemple, la vérification de l'opérande gauche d'une implication est continuellement itérée par le simple fait d'être sous la portée d'un *always* (cf. section 2.2.3.3).

Le prototype d'implémentation détaillé dans [EEH+07] prévoit l'emploi de modules mandataires (*proxies*) pour engendrer la réaction aux événements. Dans l'exemple montré en figure 3.6, un module maître communique par diffusion (*broadcast*) avec deux esclaves, au moyen d'un canal hiérarchique avec deux FIFOs. Chaque esclave est chargé de récupérer les données dans la FIFO dédiée. Les modules mandataires P1, P2 et P3, introduits dans le design, s'occupent de faire suivre la communication et de générer les événements associés à la réalisation des communications.

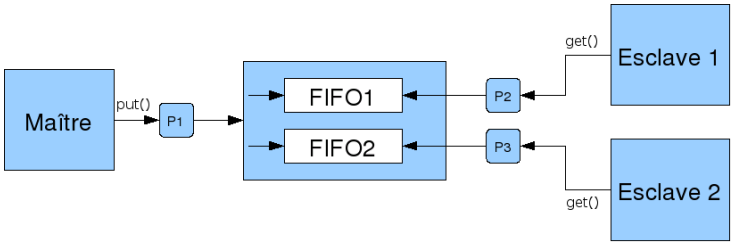

FIGURE 3.6 – Exemple de systéme avec *proxies*

L'utilisation de modules mandataires est une alternative à notre modèle. Toutefois, les auteurs ne précisent pas si ce mécanisme est facilement adaptable aux designs hybrides, ni sa complexité et l'impact de l'instrumentation. L'infrastructure est très complète mais également complexe. L'évaluation d'une séquence est modélisée par un jeton qui se propage à travers tous les éléments de la séquence, comme indiqué par les flèches continues de la figure 3.7. Chaque fois que le jeton atteint une expression de la couche booléenne (indiquée par toute boîte BL dans la figure), l'expression est alors évaluée. Les jetons sont introduits dans la structure par les modules mandataires. Une caractéristique avantageuse de cette solution est la gestion des variables locales de SVA, propagées à l'aide des jetons.

Dans [EEH+07] sont reportés les temps de simulation pour un cas d'étude où différents

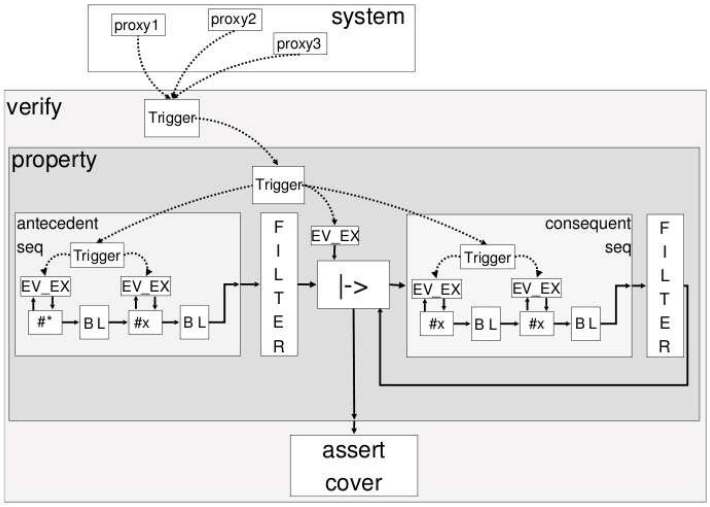

FIGURE 3.7 – Structure interne d'un moniteur (source : [EEH+07])

modules (modélisant potentiellement des microprocesseurs) sont organisés en pipeline. Ces temps CPU ne peuvent pas être comparés avec précision à ceux que nous donnerons dans le chapitre 7, car nous n'utilisons pas la même étude de cas. Néanmoins, nos résultats expérimentaux paraissent bien meilleurs. Parmi les temps annoncés dans [EEH+07], le meilleur résultat comporte un temps de simulation égal à 250% celui observé sans instrumentation du design, cela pour environ 15000 évaluations de la propriété. L'instrumentation la plus coûteuse entraîne un facteur 10 sur le temps de simulation pour seulement 381000 évaluations.

Tabakov et Vardi Les travaux de Tabakov et al. [TVKS08, TV10a, TV10b] discutent la définition de nouveaux langages temporels, ou l'adaptation des langages de spécification existants (PSL, SVA...) dans un contexte de vérification par assertions pour SystemC. Leur point de vue insiste sur la prise en compte des caractéristiques du noyau de simulation. On retrouve dans ces travaux tous les concepts des solutions présentées précédemment [Lah06, LGH+06, EEH+06a, EEH+06b, MMMC06]. Dans [TV10a] les auteurs suggèrent une instrumentation du design, et du noyau, basée sur un mécanisme d'observation similaire au nôtre [PF08], et dans [TV10b] ils décrivent leur méthode de construction des moniteurs qui reprend les principes classiques de la transformation d'une formule temporelle en automate reconnaisseur de traces finies violant la formule. Les exemples utilisés pour justifier l'instrumentation du noyau [TV10a], notamment un processus réalisant des additions, ne font pas intervenir d'aspects transactionnels et ne sont donc pas représentatifs de plates-formes SystemC TLM.

3.2.3 Outils commerciaux

3.2.3.1 Vers la validation par ABV

Si le niveau transactionnel de SystemC, relativement inexploré au cours des années passées, commence à connaître un certain essor dans le milieu industriel, les outils commerciaux qui offrent des solution de vérification adaptées à ce niveau sont encore très peu nombreux.

Synopsys, Jasper DA et SpringSoft EDA proposent de riches solutions d'ABV dynamique, cependant encore très orientées RTL :

- VCS [Syn] de Synopsys est un environnement de simulation élaboré, avec un grand nombre de fonctionnalités liées à la vérification (surveillance d'assertions, analyse de couverture de code, etc.). La VCS *Verification Library* offre une panoplie d'outils d'aide au *debug* et à la vérification, parmi lesquels des générateurs de trafic et des IPs pour vérifier le comportement d'autres IPs. L'usage de cette bibliothèque est limité au niveau RTL.
- JasperCore et JasperGold de Jasper DA annoncent des mécanismes de preuve et de *debug* sophistiqués mais dédiés à la vérification formelle de designs RTL.
- L'outil Verdi de SpringSoft supporte l'ABV dynamique et inclut un grand nombre de fonctionnalités de trace et d'exploration de code, en particulier en cas de violation d'une assertion. À l'issue d'une première simulation, une caractéristique notable est la possibilité d'ajouter et observer le comportement d'autres assertions sans besoin de simuler de nouveau. Il supporte VHDL, Verilog et SystemVerilog, mais pas SystemC. D'après la documentation de l'outil, il intègrerait SVA mais pas PSL.

JEDA Technologies offre une suite de validation pour modèles SystemC décrits à différents niveaux d'abstraction. Parmi les caractéristiques supportées, figurent la génération de trafic, plusieurs métriques d'analyse de couverture et la vérification dynamique de propriétés temporelles.

Dans [KT07], Kasuya et Tesfaye décrivent un langages d'assertions inspiré de SVA et adapté à SystemC, nommé NSCa (pour *Native SystemC assertion mechanism*). Il s'agit d'une bibliothèque d'opérateurs et d'assertions qui peut être liée au code C++/SystemC lors de la compilation : des statistiques relatives aux assertions (violations, taux d'activation des sous-parties d'une assertion, etc.) sont alors affichées lors de la simulation du design instrumenté [KTZ06]. Au niveau CA de SystemC, la syntaxe et le fonctionnement de NSCa sont très similaires à ceux de SVA. Les assertions sont évaluées par rapport aux cycles d'horloge. Parmi les opérateurs et les macros disponibles figurent notamment la séquence (`nsc_sequence`), la répétition et le retard en nombre de cycles (`[*m:n]` et `@[m:n]`), l'intersection et la disjonction de séquences (`nsc_intersect` et `nsc_or`). Aux niveaux les plus abstraits (PV ou PVT), NSCa introduit un ensemble de primitives spécifiques dénommé TLA. Dans ce contexte, le langage décrit le déclement de l'évaluation d'assertions par rapport à des événements. Par exemple, l'assertion élémentaire `nsc_always( @@addr_phase_done) |-> check_data_phase() )` effectue une vérification de la phase d'échange de données (`check_data_phase()`) chaque fois que l'événement `addr_phase_done` est rencontré. Le moyen d'implémenter ces événements, ainsi que la sémantique précise des opérateurs, ne sont toutefois pas détaillés. Les auteurs mentionnent l'extension de l'*Analysis Port* de la TLM 2.0 comme technique de notification des événements : un mécanisme de *callback* serait à l'origine des événements relatifs aux transactions. Il n'apparaît pas clairement que les outils actuels de Jeda supportent cette solution.

3.2.3.2 Retour d'expériences

Nous avons eu à notre disposition et avons donc pu expérimenter trois outils industriels : FoCs v2.04 de IBM[7], Questa v6.3a de Mentor Graphics[8] et IUS (*Incisive Unified Simulator*) v6.11 de Cadence[9].

Les principes de FoCs ont été présentés dans la section 3.2.2. Nous avons mené des comparaisons en insérant les moniteurs générés par l'outil d'IBM dans notre mécanisme de surveillance, à la place de nos moniteurs. Les détails relatifs à cette étude seront présentés dans le chapitre 7.

Questa offre un support pour l'ABV dynamique de design VHDL ou Verilog avec des propriétés écrites en PSL ou SVA. Le simulateur montre les activations, les violations et d'autres statistiques relatives aux assertions. Nous n'avons pu identifier aucun support pour PSL en SystemC dans la version 6.3a de l'outil.

IUS offre une collection d'outils pour compiler, simuler, visualiser et vérifier des designs décrits en VHDL, Verilog et SystemC, ou mélanges des trois. Durant nos expérimentations nous nous sommes restreint à la simulation textuelle avec NC-SC, qui reste indépendant de l'environnement graphique SimVision. NC-SC supporte les assertions PSL pour la vérification de designs SystemC. Une grande partie de la syntaxe PSL est acceptée par la version 6.11, avec toutefois quelques imprécisions : l'opérateur **`next_event`** et les variantes fortes d'autres opérateurs, comme **`until`**`!` et **`before`**`!`, ne sont pas disponibles ; de plus, l'opérateur **`eventually`**`!` n'avertit pas si son opérande ne se produit pas avant la fin de la trace. NC-SC supporte à la fois des assertions synchronisées sur une horloge et celles sans horloge. Dans le deuxième cas, l'ensemble des événements par défaut de tous les signaux présents dans l'assertion est utilisé comme "liste de sensibilité" de la propriété. Cela signifie que la propriété est évaluée chaque fois que l'un de ces événements est notifié. Il s'agit d'événements SystemC (`sc_event`) : pour un signal, c'est l'événement indiquant qu'il a changé de valeur (`value_changed_event`). Les assertions sans horloge acceptent uniquement les (événements des) objets de type `sc_signal`, `sc_in`, `sc_out` et `sc_inout` (des signaux d'entrée/sortie d'un module) dans leur liste de sensibilité. Elles ne réagissent pas en présence de types C++, de modules SystemC et de classes définies par l'utilisateur.

3.3 Bilan

Depuis la sortie de SystemC, plusieurs efforts d'ABV statique et dynamique ont été proposés. La solution présentée par Große et Drechsler [GD03] constitue l'une des premières avec une démarche mathématique appliquée à la vérification de designs SystemC ; la solution est toutefois restreinte au niveau RT. Karlsson *et al.* [KEP06] offrent une formalisation intéressante mais de très grand taille. Habibi et Tahar [HT05] introduisent un flot unifié, basé sur des diagrammes UML, mais les exemples évoqués raisonnent en termes de cycles d'horloge. Moy *et al.* [MMMC06] montrent un flot élaboré s'appuyant sur un modèle intermédiaire avec une sémantique formelle bien définie ; ils ne considèrent toutefois pas les langages d'assertions tels que PSL. Kroening *et al.* [KS05] proposent une solution pour partitionner automatiquement le système entre logiciel et matériel, afin d'en optimiser la vérification, mais ils se limitent encore une fois aux niveaux les moins

[7] `https://www.research.ibm.com/haifa/projects/verification/focs`
[8] `http://www.mentor.com/questa`
[9] `http://www.cadence.com/products/sd/enterprise_simulator`

abstraits de SystemC.

Toutes les méthodes de vérification par *model checking* souffrent encore des limitations dues à l'explosion en nombre d'états du système [Var07].

Parmi les efforts d'ABV dynamique, ceux proposés par Lahbib [Lah06] et par Ecker *et al.* [EEH+06b, EEH+07] revèlent des solutions tangibles, dont certains aspects offrent des points de réflexion importants. En particulier, nous partageons le besoin de caractériser, dans TLM, à la fois les propriétés et les traces d'exécution qui permettent de les évaluer. Avec notre contribution, détaillée dans les chapitres 4, 5 et 6, nous nous efforçons de résoudre un certain nombre de limitations, en visant ainsi une solution plus mature.

Le chapitre 4 se concentre sur la technologie de base de notre solution d'ABV : d'une part une technique efficace de construction de moniteurs TLM à partir de propriétés PSL, inspirée de l'approche originale introduite dans [BLMA+05, BLOF06], et d'autre part une méthode spécifique d'observation des actions de communication nécessaire au "déclenchement" des vérifications par les moniteurs. Le chapitre 5 étend significativement les possibilités offertes par notre approche, en proposant pour les assertions PSL à la fois un support formel et une mise en œuvre pratique pour les variables globales (couche "*modeling*") et pour les variables locales nécessaires à la réentrance des propriétés. Le chapitre 6 détaille la mise en œuvre de tous ces concepts dans l'outil prototype ISIS, ainsi que quelques études annexes à cette mise en œuvre. Enfin le chapitre 7 décrit une variété d'expérimentations avec ISIS.

Chapitre 4

Surveillance de propriétés au niveau transactionnel

Introduction

Comme expliqué dans les chapitres précédents, les assertions offrent un moyen concis et non ambigu pour exprimer les comportements que le design doit ou ne doit pas exhiber. En vérification dynamique, il existe essentiellement deux solutions à base d'assertions : la première consiste à utiliser un outil de simulation qui intègre la possibilité d'exprimer et de surveiller les assertions durant la simulation, la deuxième repose sur l'instrumentation du DUV par des composants moniteurs de surveillance générés à partir des assertions (cf. section 1.3.2.1).

Généralement, l'emploi d'un environnement de simulation dédié permet d'avoir un affichage des informations relatives aux assertions agréablement intégré dans les résultats de la simulation. L'instrumentation à l'aide de moniteurs présente un avantage dans le sens où le résultat est complètement indépendant de tout simulateur, mais elle doit être conçue de façon à fournir des résultats tout aussi lisibles.

Nous rappelons qu'un moniteur de surveillance est un composant matériel synthétisé à partir d'une assertion, destiné à être connecté au système à vérifier et donc décrit dans le même langage que celui-ci. L'intérêt d'introduire des moniteurs dans l'architecture ne s'arrête donc pas à la vérification par simulation. Comme proposé dans [GD04, PLBN05, BLOF06], les descriptions RTL d'un circuit et de ses moniteurs peuvent être synthétisées pour émulation sur FPGA, processus plus efficace que la simulation. Les moniteurs synthétisés peuvent également être laissés dans le dispositif final, pour des raisons d'auto-surveillance et pour l'observation de propriétés critiques après conception. Dans cette approche orientée moniteurs, les auteurs de [ABG+00] proposent l'outil FoCs, capable de générer automatiquement ces modules à partir d'une spécification formelle. L'approche adoptée dans FoCs est une méthode qui emploie des automates pour traduire les propriétés de la logique temporelle (cf. section 3.2.1). La traduction de l'assertion en machine à états est généralement exponentielle par rapport au nombre d'opérateurs [MAB06b].

L'outil MBAC, présenté dans [BZ05], vise les mêmes objectifs que FoCs. La génération des moniteurs RTL à partir des propriétés suit toujours l'approche par automate. Après génération, MBAC effectue des optimisation sur les machines à états : dans la plupart

des cas, les résultats obtenus après synthèse sur FPGA sont meilleurs que ceux de FoCs, en nombre de flip-flops, de LUTs (*Look-Up Tables*) et en fréquence de fonctionnement [BZ06, BZ07].

D'autres techniques ont été développées sur des principes approchants [OH02, GD04, BZ05, PLBN05]. Toutes ces solutions se situent au niveau RTL. Comme nous l'avons présenté dans le chapitre précédent, assez peu de résultats utilisables en pratique ont été proposés pour la vérification de propriétés au niveau TLM.

L'objectif de cette thèse est de fournir une solution concrète, et utilisable pour la validation de plates-formes présentant toutes les facettes de la modélisation TLM (présence de temps ou pas, communications par divers types de canaux, interfaces bloquantes et non bloquantes...). Cela nous a avant tout conduits à résoudre deux problèmes :

- la construction de moniteurs de surveillance SystemC à partir d'assertions temporelles, et surtout
- l'instrumentation du DUV par ces moniteurs de façon à garantir leur activation aux instants appropriés de la simulation.

Nous développons ces deux aspects dans ce chapitre. Pour la construction des moniteurs, nous n'utilisons pas de techniques à base d'automates, mais nous adoptons une solution inspirée de l'approche modulaire présentée dans [BLOF06, MAB06b]. L'instrumentation du DUV s'appuiera sur un mécanisme d'observation permettant d'échantillonner la simulation sur les actions pertinentes vis à vis de l'assertion à vérifier.

4.1 Construction de moniteurs de surveillance

Nous rappelons tout d'abord la méthode développée dans [BLOF06, MAB06b] pour le niveau RTL, qui a inspiré notre solution[1].

4.1.1 Une approche modulaire au niveau RTL prouvée correcte

Cette méthode de synthèse de moniteurs au niveau RTL, à partir d'assertions PSL, ne suit pas l'approche classique, qui consiste à obtenir une machine à états finis, plus ou moins optimisée, à partir d'une formule de logique temporelle. Il s'agit d'une méthode modulaire, qui réalise une construction linéaire des moniteurs, qui fournit des bons résultats à la synthèse et, surtout, qui est prouvée correcte.

La technique est basée sur une construction compositionnelle à partir d'une bibliothèque VHDL de moniteurs élémentaires. Ces moniteurs, chacun correspondant à un opérateur FL de PSL, sont les composants de base. Une méthode d'interconnexion systématique permet de construire un moniteur pour une propriété complexe à partir d'instances de ces moniteurs primitifs. L'interconnexion de ces modules primitifs n'entraîne pas d'explosion dans la taille du moniteur final. La structure du moniteur est linéaire par rapport au nombre d'opérateurs dans la formule. En outre, le fonctionnement de tous les moniteurs de la bibliothèque, ainsi que leur méthode d'interconnexion, ont été prouvés corrects [MAB06b], par vérification formelle au moyen de l'assistant de preuve PVS[2] [OSR92].

[1] La méthode pour les moniteurs VHDL a connu des mises à jour depuis sa première version. Nos travaux sont basés sur la méthode initiale : cette solution convient parfaitement à nos besoins.

[2] *Prototype Verification System*, `http://pvs.csl.sri.com`.

Tous les moniteurs élémentaires se conforment à la même interface, comme montré par la figure 4.1. Dans chaque moniteur élémentaire deux parties peuvent être identifiées. La première est dédiée au calcul de la fenêtre d'observation et détermine quand l'évaluation de l'opérateur correspondant est effectuée. La deuxième accomplit l'évaluation de l'opérateur en utilisant les valeurs des opérandes.

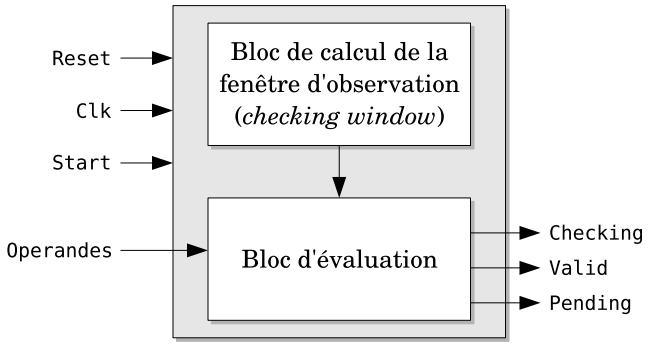

FIGURE 4.1 – Interface et structure d'un moniteur élémentaire

En suivant cette interface commune, tout moniteur VHDL de base possède les entrées suivantes :

- un signal *Reset* et un signal d'horloge,
- un signal *Start* pour son activation,
- l'ensemble des signaux du DUV correspondants aux opérandes de l'opérateur considéré.

Les trois sorties produites ont la signification suivante :

- *Checking* : sa valeur passe à `'1'` pour indiquer que la sortie *Valid* sera effective au prochain instant de synchronisation. Cette sortie est indispensable lors de la composition du moniteur complet par interconnexion des moniteurs élémentaires, car elle active le prochain moniteur élémentaire.
- *Valid* : sa valeur indique le respect de l'opérateur ; `'0'` signifie erreur, `'1'` signifie absence d'erreur.
- *Pending* : une valeur égale à `'1'` indique que le moniteur a été démarré mais que le niveau de satisfaction (cf. section 2.2.3.4) est encore *en attente*. Cette sortie est utilisée pour la version forte des opérateurs PSL (cf. sections 2.2.3.3 et 2.2.3.4).

Le moniteur global, correspondant à une formule PSL P, est obtenu en composant les moniteurs de base grâce à la technique d'interconnexion spécialisée. Cette interconnexion s'appuie sur l'arbre syntaxique de la propriété P. Chaque nœud dans l'arbre représente un opérateur FL, associé à un moniteur de la bibliothèque de base. Les feuilles dans l'arbre sont les opérandes dans P. Par exemple, l'arbre syntaxique associé à la formule $P_1 =$ `always (a -> next! (b before! c))` est montré sur la figure 4.2.

L'arbre est parcouru depuis sa racine vers ses feuilles. Pour tout opérande rencontré, si cet opérande représente une simple expression booléenne, alors il est connecté à l'entrée correspondante du moniteur associé à l'opérateur. Si l'opérande est une formule temporelle, l'entrée correspondante dans l'opérateur est alors affectée à `'1'` : l'évaluation de l'opérande dépend des autres moniteurs qui vont suivre dans l'arborescence.

Les signaux *Checking* et *Valid* primaires sont ceux du dernier moniteur de base, i.e. l'opérateur FL le plus à droite dans la formule. Cela signifie que les sorties du moniteur complet sont celles du dernier moniteur élémentaire connecté. Le signal *Pending* primaire

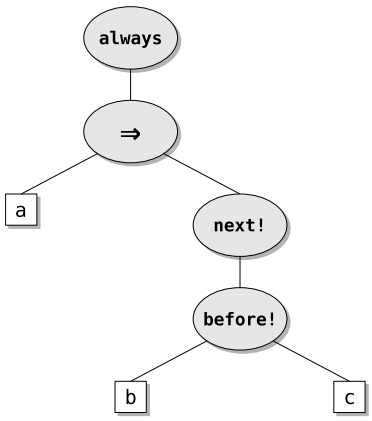

FIGURE 4.2 – Arbre syntaxique pour la formule PSL P_1

est obtenu par disjonction des sorties *Pending* de tous les opérateurs forts dans la propriété. Un module `Gen_init` fournit le *Start* initial au moniteur proprement dit. Ce module est utilisé pour déclencher l'état initial d'observation, juste après le dernier *Reset*.

Le résultat de cette méthode, appliquée à l'exemple de la formule P_1, est montré sur la figure 4.3. Ici les entrées *expr* des trois moniteurs **`always`**, `->` et **`next`**`!` sont affectées à `'1'` et les trois sorties *Valid* sont laissées déconnectées : le dernier module seul fournit les résultats.

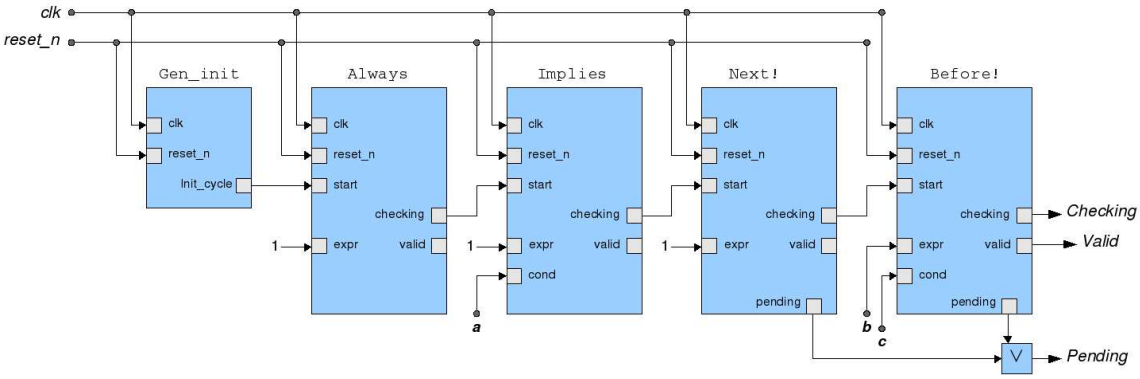

FIGURE 4.3 – Moniteur RTL composite pour la formule P_1

Instrumentation du DUV L'équipe VDS de TIMA a développé l'environnement Horus [OMAB08], outil prototype qui met en oeuvre les principes que nous venons de décrire. Il s'agit d'un logiciel capable de produire automatiquement la description matérielle VHDL de moniteurs de surveillance à partir de propriétés PSL.

L'approche choisie vise la généricité et la possibilité de réutiliser les modules produits : à sa création, un moniteur ne possède aucun lien explicite avec le DUV. L'étape d'instrumentation du modèle pour l'observation de propriétés PSL se fait en dernier lieu : les signaux de chaque moniteur (i.e. les opérandes de la propriété PSL demandée) sont connectés par l'outil aux signaux à observer dans le design, cela au moyen de l'instruction de création/connexion d'instances de VHDL (*port map*). L'horloge et le *reset* du moniteur sont connectés à l'horloge et au *reset* du design. Le résultat obtenu est le code contenant à la fois le DUV et tous les moniteurs connectés. Ce code peut être simulé ou synthétisé.

4.1.2 Construction des moniteurs orientés TLM

Pour construire des moniteurs SystemC dédiés à la surveillance de designs TLM, une variante de cette méthode d'interconnexion originale (cf. section 4.1.1) a été conçue. En premier lieu, nous avons transformé tous les moniteurs de base de la bibliothèque VHDL en une version SystemC sans horloge.

4.1.2.1 Implémentation de la bibliothèque de moniteurs C++

Afin d'éviter une trop forte dépendance vis-à-vis du noyau de simulation SystemC, nous avons choisi de modéliser ces composants avec aussi peu de constructions SystemC que possible. Pour tous les composants de la bibliothèque, uniquement des variables C++ sont utilisées, et aucun signal SystemC n'est employé. Chaque moniteur est une classe C++ munie d'une méthode `update`, utilisée pour la mise à jour de l'état du moniteur à chaque activation. Dans un contexte C++, le code doit être rendu séquentiel. L'ordre dans lequel les instructions sont exécutées doit préserver l'ordre d'évaluation qui correspond à la sémantique VHDL. Ceci peut s'expliquer à l'aide d'un exemple, comme celui de l'opérateur `always` qui suit. La colonne de gauche montre la version VHDL, celle de droite la version C++ :

```vhdl
-- Moniteur elementaire VHDL
entity mnt_always is
  port (clk,reset_n,start,expr : IN BIT;
        checking, valid : OUT BIT);
end mnt_always;

architecture monitor of mnt_always is
  signal start_always, start_t1 : BIT;
  signal valid_t : BIT := '1';
begin
-- processus synchrones
evaluate_start: process (clk)
begin
  if clk'event and clk='1' then
    if reset_n='0' then
      start_t1 <= '0';
    elsif start = '1' then
      start_t1 <= '1';
    end if;
  end if;
end process;
evalulate_expr: process (clk)
begin
  if clk'event and clk='1' then
    if reset_n='0' or start_always='0'
      then valid_t <= '1';
    elsif expr = '1' then
      valid_t <= '1';
    else
      valid_t <= '0';
    end if;
  end if;
end process;
-- partie combinatoire
start_always <= start or start_t1;
valid        <= valid_t;
checking     <= start_always;
end monitor;
```

```cpp
// Moniteur elementaire C++
class mnt_always : public Monitor {
  protected:
    bool reset_n, start, expr;
    bool start_always, start_t1,
         valid_t, valid, checking;
  public:
    mnt_always(const char *n)
     : Monitor(n) {
      valid_t = true;
    }
    void update() {
      // partie 1
      start_always = start || start_t1;
      // partie 2
      if ((reset_n == false) ||
          (start_always == false))
        valid_t = true;
      else
        if (expr == true)
          valid_t = true;
        else
          valid_t = false;
      // partie 3
      if (reset_n == false)
        start_t1 = false;
      else
        if (start == true)
          start_t1 = true;
      // partie 4
      valid = valid_t;
      // partie 5
      checking = start_always;
    }
};
```

Dans la description VHDL, `start_t1` et `valid_t` sont des flip-flops synchronisés sur front montant d'horloge : leur valeurs sont modifiées à chaque front montant mais elles sont disponibles uniquement lors du prochain cycle de simulation (cycle delta). Chaque mise-à-

jour asynchrone de l'entrée `start` provoque une réévaluation de la valeur de `start_always`. Cette valeur dépend de celle chargée dans le registre `start_t1` au front d'horloge précédent. Donc, l'affectation de la variable `start_always` du code C++ (partie 1) doit être faite avant l'affectation de `start_t1` (partie 3), pour pouvoir utiliser la valeur précédente. Puisque `checking` et `valid_t` dépendent de `start_always`, leurs affectations (respectivement parties 5 et 2) doivent être effectuées après celle de `start_always`. Enfin, de la même façon que ce qui a été expliqué pour la mise-à-jour de `start_always`, l'affectation de la variable `valid` (partie 4) devrait se faire avant celle de `valid_t`. Toutefois, si dans le code VHDL la disponibilité de la nouvelle valeur pour la sortie `valid` est volontairement retardée d'un cycle d'horloge par rapport aux autres sorties, cela pour des raisons de synchronisation, dans le code SystemC il est préférable de l'obtenir à l'instant d'évaluation courant. Sa mise-à-jour est donc faite après celle de `valid_t`.

Nous avons conçu la totalité de la bibliothèque SystemC suivant ce principe, qui peut être généralisé comme suit : on procède d'abord avec les affectations des signaux combinatoires depuis les entrées vers les sorties du moniteur, puis on calcule les valeurs des registres dans l'ordre inverse. Les sorties peuvent être affectées en dernier lieu.

Le code de chaque opérateur prévoit un mode "déboguage", avec un affichage détaillé relatif à l'état interne de chaque moniteur élémentaire. La bibliothèque inclut tous les opérateurs indiqués dans la règle syntaxique `FL_Property` de l'annexe A de [PSL05].

4.1.2.2 Interconnexion des composants C++ élémentaires

La méthode d'interconnexion des moniteurs de base pour former le moniteur composite associé à une propriété est différente de la solution VHDL car il ne s'agit pas ici de connecter des composants matériels, mais d'enchaîner des appels aux fonctions `update` des moniteurs de base. Dans le moniteur global, le module correspondant à l'opérateur le plus "à droite" dans la formule PSL est celui qui recevra la notification d'activation en premier. Ce module appelle alors la méthode `update` du module associé à l'opérateur "à sa gauche", et l'enchaînement d'appels remonte jusqu'à l'opérateur le plus à gauche dans la formule. En réalité, dans les spécifications pour la vérification dynamique (sous-ensemble simple de PSL, cf. section 2.2.3.1), et sauf pour l'opérateur *et*, tous les opérateurs PSL ont *au plus* un seul opérande non booléen (donc une sous-formule temporelle). Rappelons maintenant la formule PSL P_1 = **`always`** `(a ->` **`next`**`! (b` **`before`**`! c))`, dont l'arbre syntaxique est montré par la figure 4.2 à la page 64. Le moniteur SystemC correspondant à P_1 comporte quatre instances de moniteurs de base, un pour chaque opérateur, tout comme sa contrepartie VHDL. Le premier bloc notifié est donc celui de l'opérateur **`before`**. Ce bloc appelle la méthode `update` du bloc pour l'opérateur parent, dans ce cas **`next`**. Cet enchaînement d'appels continue de nœud fils à nœud parent, jusqu'à l'opérateur à la racine de l'arbre syntaxique (ici **`always`**). L'évaluation finale se réalise en sens inverse (de parent en fils). Ce mécanisme est représenté sur la figure 4.4.

Chaque bloc élémentaire dans la propriété est une instance d'une sous-classe de l'opérateur correspondant dans la bibliothèque de base. Dans chaque sous-classe, la méthode `update` est surchargée pour rendre possible le mécanisme décrit ci-dessus. Les *évaluations* indiquées sur la figure 4.4 correspondent à des appels à la méthode `update` de la classe de base. Le dernier bloc élémentaire (le **`before`** dans l'exemple de la propriété P_1) fournit les sorties *Valid*, *Checking* et *Pending* finales. Au choix, ces sorties peuvent être affichées lors de chaque activation du moniteur global ou uniquement quand la propriété est violée. Dans ce cas, le moniteur affichera également l'état de toutes ses "entrées", c'est-à-dire des

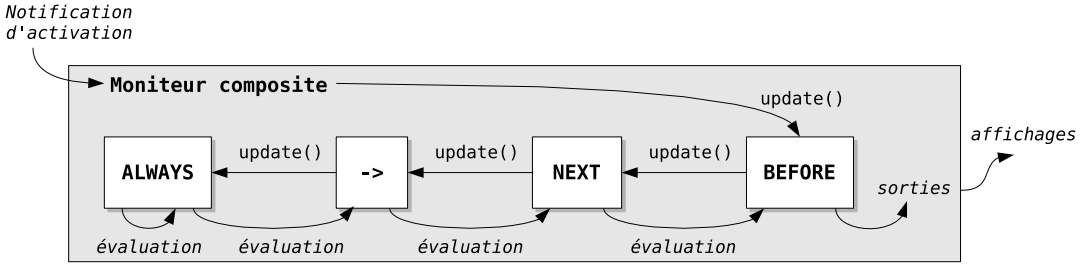

FIGURE 4.4 – Mécanisme d'évaluation des moniteurs élémentaires pour P_1

paramètres `a`, `b` et `c` dans l'exemple de la propriété P_1.

Opérateur binaire *et* Comme mentionné ci-dessus, presque tous les opérateurs ont au plus un opérande de la classe FL (cf. section 2.2.3.1) de PSL. Par conséquent, l'interconnexion des moniteurs élémentaires s'effectue toujours selon le même principe. Seul l'opérateur *et* représente une exception puisque ses deux opérandes peuvent être de nature FL.

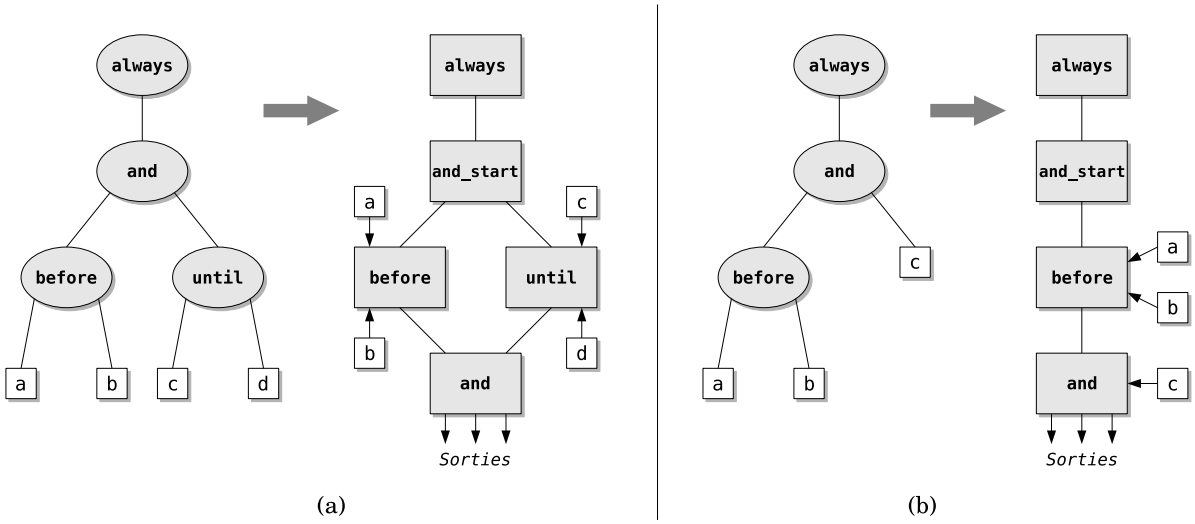

FIGURE 4.5 – Cas particulier de l'opérateur *et*

Il faut alors distinguer trois cas de figure, chacun comportant une construction différente :

1. conjonction entre deux expressions booléennes : aucun bloc relatif à un moniteur élémentaire n'est construit, car il s'agit d'une simple expression booléenne ;
2. conjonction entre deux sous-formules temporelles : un bloc relatif au moniteur élémentaire *et* est construit ; ce bloc déclenche l'évaluation des deux sous-formules temporelles (figure 4.5 a) ;
3. conjonction entre une expression booléenne et une sous-formule temporelle : une seconde version du bloc relatif au moniteur élémentaire *et* est construit ; ce bloc évalue immédiatement l'expression booléenne et déclenche l'évaluation de la sous-formule temporelle (figure 4.5 b) ;

4.2 ABV au niveau TLM

Les modèles transactionnels (cf. section 2.1.3) sont nettement plus abstraits que leur contrepartie RTL. Du point de vue *temporel*, nous nous trouvons dans un contexte où aucune horloge pour la synchronisation n'est (potentiellement) présente. Comme introduit au début du chapitre 3, il faut parvenir à des traces de simulation pertinentes pour la vérification dynamique des assertions. Il est alors indispensable d'identifier les "points d'échantillonnage" de la trace de simulation, mais aussi de trouver un modèle permettant d'observer ces instants dans le système et de déclencher les moniteurs. Pour cela, il est évident que nous ne pouvons plus nous référer à l'horloge du design, comme c'était le cas pour les moniteurs RTL.

4.2.1 Définition de trace

4.2.1.1 Le problème de l'échantillonnage

La sémantique de PSL se définit sur des traces (cf. section 2.2.3.3), mais aucune contrainte n'est imposée quant à la manière d'obtenir les instants qui composent ces traces (chaque instant est une lettre du mot, i.e. la trace de simulation). À titre d'exemple, considérons un système décrit au niveau RTL, synchronisé sur front montant d'une horloge `clk` et comportant trois signaux `a`, `b` et `c`. Un exemple de comportement de ce système est montré par la figure 4.6.

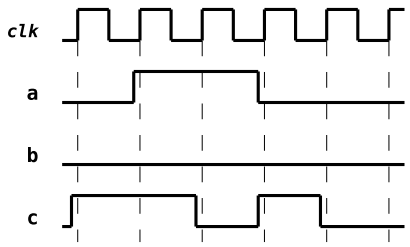

FIGURE 4.6 – Exemple de comportement d'un système séquentiel synchrone

Comme expliqué dans les sections 2.2.2.2 et 2.2.3.3, les instants d'évaluation d'une propriété PSL pour un tel système synchrone peuvent être considérés simplement comme étant les fronts montants de l'horloge. Dans ce cas, cette trace finie comporte 6 lettres.

Dans un système décrit en SystemC aux niveaux transactionnels les plus abstraits (algorithmique, ou les équivalents de PV, cf. section 2.1.3.1, et LT, cf. section 2.1.3.3), le signal d'horloge est en principe absent. Il peut même n'y avoir aucun temps physique de simulation, comme c'est le cas pour la plate-forme avec contrôleur DMA de la section 2.1.3.2 (cf. figure 2.8 à la page 19) qui est décrite au niveau TLM PV, c'est-à-dire sans temps. Reprenons cette plate-forme à titre d'exemple pour illustrer notre propos quant à l'échantillonnage. Un diagramme de séquences simplifié[3] traduisant un extrait de simulation, durant lequel le DMA effectue une copie entre les deux mémoires, est donné sur la figure 4.7. Il reflète la séquence suivante d'appels de fonctions :

- le contrôleur DMA effectue une première requête de lecture à l'adresse `src` via le Router (nous rappelons que dans un canal *memory-mapped* tel que le Router, l'adresse permet d'identifier l'esclave suite à une opération de décodage) ;

[3] Pour des raisons de lisibilité, le diagramme est volontairement simplifié, seul les principaux appels de fonctions sont représentés.

- la première mémoire dessert la requête (opération `read`) ;
- le DMA effectue une requête d'écriture à l'adresse `dst` ;
- la deuxième mémoire dessert cette requête (opération `write`) ;
- le DMA effectue une deuxième requête de lecture à l'adresse `src+1` ;
- la première mémoire dessert cette requête ;
- le DMA effectue une deuxième requête d'écriture à l'adresse `dst+1` ;
- la deuxième mémoire dessert cette requête.

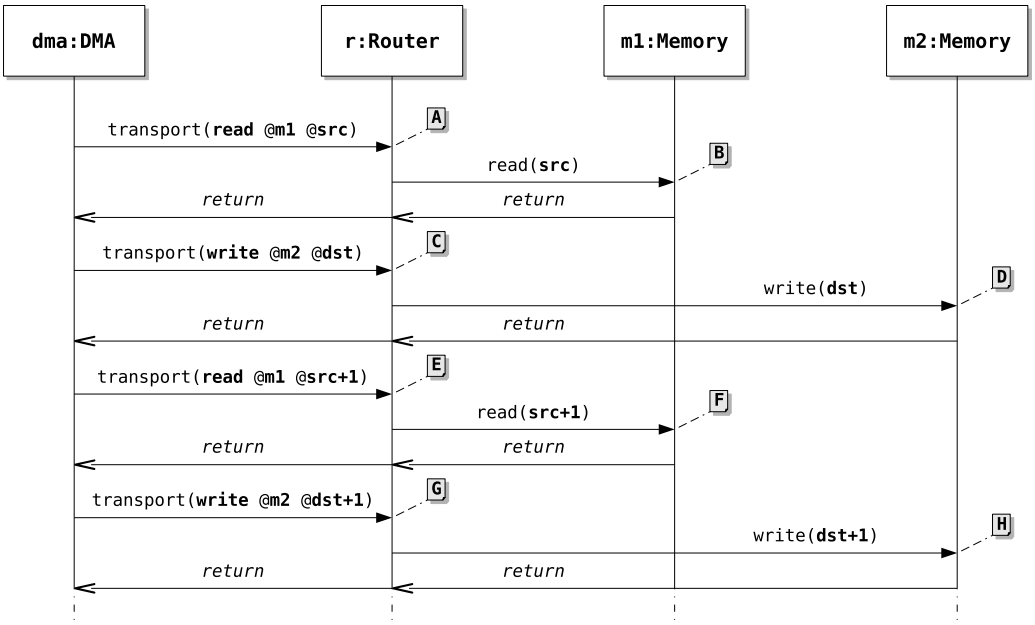

FIGURE 4.7 – Extrait de simulation du DMA sous la forme de diagramme de séquence UML simplifié

L'ordre dans lequel se déroulent ces appels de fonctions est donné par l'exécution du programme C++ avec son noyau de simulation SystemC. Dans le cas où la description comporte la notion de temps de simulation, et pour deux actions a_1 et a_2 se déroulant à des temps de simulation t_1 et t_2 tels que $t_1 < t_2$, il est clair que a_1 se trouve nécessairement dans la trace de simulation avant a_2. C'est le cas également, que la description soit avec ou sans temps, si a_1 et a_2 découlent de l'activation de deux processus éligibles "successivement" (du point de vue du noyau de simulation), ou de l'exécution séquentielle d'un unique processus. Mais dans le cas où deux processus SystemC sont éligibles "simultanément" (du point de vue du noyau de simulation), l'ordre de réactivation de ces processus n'est pas garanti et, en conséquence, l'ordre des actions découlant de leur réactivation non plus. Ces facteurs doivent être pris en compte pour une définition judicieuse de l'échantillonnage de la simulation (cette remarque a également été soulevée par [EEH+06a]). Par ailleurs, il faut éviter autant que possible la prise en compte d'informations parasites. Par exemple, sur la simulation illustrée par la figure 4.7, si la propriété considérée porte sur les comportements du Router et des deux mémoires, il sera indispensable de prendre en considération dans la trace les points A, B, C, D, E, F, G et H qui correspondent à toutes les actions de ces trois composants. Par contre, si la propriété ne concerne par exemple que les écritures dans la mémoire 2 (imaginons une propriété comme "*toutes les valeurs écrites dans la mémoire 2 sont positives*"), il sera préférable de n'extraire de la trace de simulation que les points D et H (ou C et G selon le point de vue où l'on souhaite se placer).

Non seulement une telle sélection des points d'échantillonnage permettra d'optimiser les vérifications, mais elle permettra aussi de supprimer toute information non pertinente qui pourrait fausser l'évaluation de la propriété (nous reviendrons sur ce point dans notre discussion sur l'opérateur `next`). Si une propriété P porte uniquement sur des actions de communications $c_1, c_2, ..., c_n$, alors l'ensemble $\{c_1, c_2, ..., c_n\}$ peut être vu comme une sorte de "liste de sensibilité" de P, suivant une notion similaire à celles de VHDL ou SystemC par exemple.

Il est alors primordial de se doter d'une solution pour réaliser un tel échantillonnage de la trace. Comment isoler ces points d'observation ? Il n'est pas souhaitable d'utiliser une solution liée au simulateur SystemC. Par exemple, s'appuyer sur la présence de `sc_event` que le programmeur aurait pu utiliser pour synchroniser ses communications ne serait pas raisonnable (pas plus d'ailleurs que le choix du programmeur de se fier exclusivement à de telles constructions pour assurer ses synchronisations). Nous avons choisi de mettre en œuvre notre propre mécanisme d'observation, indépendant des opérations du simulateur. Il sera décrit plus précisément dans la section 4.2.2. Il s'appuie sur l'occurrence des appels de fonctions (généralement les actions de communication) et sur les mises à jour des objets qui leur sont associés. Les écritures via des canaux dans des objets "esclaves" (mémoires, co-processeurs, DMA...) provoquent généralement des réactions instantanées (parmi ces "réactions" nous comptons également les attentes temporelles lors des communications bloquantes) ; leur observation nous conduit donc à prendre en compte immédiatement les actions et les nouvelles valeurs. Les signaux représentent toutefois un cas particulier : leur valeur n'est mise à jour qu'une fois par cycle delta (dans la phase de mise à jour) ; nous faisons donc en sorte d'autoriser potentiellement plusieurs écritures dans un même cycle delta, mais de ne prendre en compte qu'un seule valeur de mise à jour, à la fin du cycle.

4.2.1.2 À propos des SEREs et de l'opérateur *next*

Au niveau RTL, les propriétés PSL emploient très couramment l'opérateur **`next`**, ainsi que les formules à base d'expressions rationnelles (SEREs). Cela est à la fois indispensable et approprié, par exemple pour pouvoir exprimer le fait que la valeur d'un signal passera à 1 *au prochain* front d'horloge (**`next`**) ou *exactement dans 5* fronts (**`next`**`[5]`). De même, des SEREs telles que `a; b[*3]; c` permettent d'exprimer une succession précise de signaux sur une succession de cycles d'horloge.

Cependant, en TLM, puisqu'il n'existe plus d'échantillonnage naturel induit par l'horloge, l'emploi d'opérateurs qui indiquent des décalages exacts dans la trace (comme le **`next`** et le "`;`" des SEREs) n'est pas pertinent. En effet, reprenons l'exemple de la section précédente et une propriété ne portant que sur les écritures dans la mémoire 2 : si nous choisissons, comme nous le préconisons, de considérer une trace minimale constituée des points D et H, l'action "suivant" D est H. Mais, pour tout contexte d'observation qui prendrait en compte tous les points de la séquence (de A à H), l'action "suivant" D serait E. L'utilisation de l'opérateur **`next`** dans la propriété devrait de préférence être évitée (ou réalisée avec prudence, en toute connaissance de l'échantillonnage choisi). L'utilisation de **`next`** dans n'importe quel contexte est bien sûr soumis aux mêmes contraintes, mais l'échantillonnage sur les fronts d'horloge est tellement naturel dans un système RTL synchrone (mono-horloge) que l'utilisation de cet opérateur se fait de façon aisée. La discussion est similaire pour l'opérateur de séquentialité sur les SEREs (le "`;`"). Un autre aspect qui rend ces opérateurs peu adaptés dans le contexte TLM est le non-déterminisme lié au noyau de simulation SystemC : l'ordre d'exécution des processus éligibles n'étant pas

spécifié, il n'est pas toujours possible de prévoir l'ordre précis entre certaines transactions du système.

Pour ces raisons, au cours de nos travaux, nous avons considéré plus intéressant et opportun d'utiliser l'opérateur **next_event**, qui permet de vérifier une sous-formule PSL *la prochaine fois qu*'une condition est satisfaite, sans toutefois contraindre le nombre d'instants avant que cette condition se produise.

En conclusion, et plus généralement, il apparaît préférable d'exprimer de spécifications temporelles qui font référence à des relations d'occurrences et d'ordre sans imposer des conditions précises quant au nombre d'instants de décalage entre deux ou plusieurs actions.

Il faut toutefois souligner que l'opérateur **next** peut être utile pour exclure l'instant d'évaluation courant dans une sous-formule PSL. Pour expliquer cela, il suffit de considérer l'assertion exprimant que si `cond` se produit, alors `expr` doit être satisfaite avant la prochaine occurrence de `cond`. La sémantique de l'opérateur d'implication impose la vérification de l'opérande droit dès satisfaction de l'opérande gauche, au même instant. Par conséquent, l'assertion ne peut pas être traduite par la formule `cond -> (expr before cond)`. En effet, dans ce cas, si `cond` est vraie au premier instant de la trace, alors la formule est violée : `cond` est déjà vrai, et cela sans avoir observé `expr` avant. L'opérateur **next** permet alors de démarrer l'évaluation d'une sous-formule à l'instant suivant : `cond -> next (expr before cond)`.

4.2.2 Modèle de surveillance

La surveillance des propriétés est rendue possible via un modèle ([Pie07, PF08]) qui tire avantage du paradigme orienté objet et qui s'inspire d'un patron de conception affirmé et répandu, le motif observateur-observable [GHJV95]. Dans cette approche, l'objet défini comme *sujet* (l'élément observé) maintient une liste d'*observateurs* et il les notifie de façon automatisée lorsqu'il change d'état. Cette notification s'effectue généralement à l'aide de l'une des fonctions de l'observateur. De façon générale, dans notre contexte, chaque *moniteur* issu de la synthèse d'une propriété PSL est placé dans un *wrapper*. Ce dernier s'occupe d'observer tous les composants du design (canaux, signaux, ports...) qui font transiter une communication impliquée dans la propriété. Le diagramme de classes UML de la figure 4.8 illustre les éléments impliqués dans ce principe.

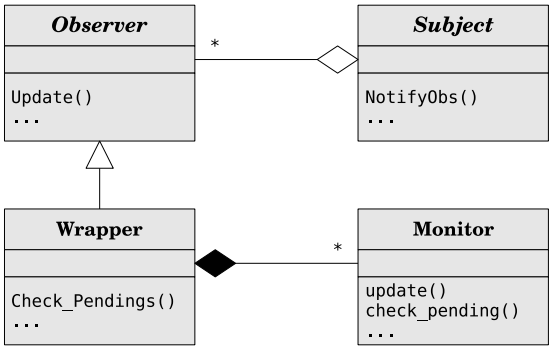

FIGURE 4.8 – Modèle d'observation - Diagramme de classes UML

Dans ce diagramme on identifie les classes suivantes :

- *Observer*, la classe abstraite qui définit l'interface de tout observateur. Elle déclare la méthode appelée et exécutée lors de chaque notification.

- *Wrapper*, la classe qui hérite de la précédente (tout *Wrapper* est un observateur) et qui englobe un ou plusieurs moniteurs.
- *Monitor*, la classe parente de tout moniteur qui traduit une propriété PSL. Elle déclare la méthode d'évaluation de la propriété.
- *Subject*, la classe parente de chaque composant observé dans le DUV. Elle déclare la méthode de notification de tous ses observateurs.

Chaque *wrapper* peut comporter une collection de moniteurs, tout comme chaque sujet observé peut avoir une collection d'observateurs. Quand un événement pertinent pour la propriété se produit, le sujet concerné notifie cet événement à l'ensemble de ses observateurs par l'intermédiaire de la méthode `NotifyObs`. Il est important de préciser que cette action se fait par rappel (*callback*), et qu'aucun élément de type `sc_event` n'est employé. La méthode `Update` de chaque observateur attaché est alors exécutée.

La classe *Wrapper* hérite de la classe *Observer*, et les *wrappers* effectivement présents dans la description SystemC du design instrumenté sont en réalité des sous-classes de *Wrapper*. Suite à une notification, chaque instance de *Wrapper* provoque l'évaluation des moniteurs qu'elle contient par appel à leur méthode `update`. Tous les moniteurs implémentent cette méthode, qui effectue un calcul des sorties *Valid*, *Checking* et *Pending* (cf. section 4.1) en fonction des entrées du moniteur. Tout moniteur associé à une propriété PSL hérite de la classe *Monitor*.

La définition d'un composant observé se fait simplement de la façon suivante : la méthode de communication qui correspond à l'événement observé est surchargée pour effectuer un appel à la méthode originale, suivi (ou précédé) par un appel à `NotifyObs`. Par exemple, dans une architecture où un maître et un esclave communiquent via une FIFO, l'observation des écritures des messages dans la FIFO (méthode `write`) peut s'effectuer par l'intermédiaire de la classe `observed_fifo`, dont le principe est schématisé comme suit :

```
// Double heritage, a la fois de la classe originale du design et
//  de la classe "Subject" du modele d'observation :
class observed_fifo : public Fifo, public Subject {
  public:
    observed_fifo(sc_module_name name) : Fifo(name) { }

    // La methode surchargee comporte un appel a la methode originale
    //  de la classe mere, plus une notification avec l'ensemble des
    //  donnees de la communication (parametres et valeur de retour de
    //  la fonction) :
    void write(char c) {
      fifo::write(c);

      // La notification peut s'effectuer juste avant, juste apres,
      //  ou avant et apres appel a la methode originale :
      NotifyObs(...);
    }
};
```

La notification des instants d'observation se fait donc suivant un mécanisme de rappel (*callback*). Ceci est important pour préserver une certaine indépendance des moniteurs par rapport au simulateur. En effet, il faut identifier immédiatement toute condition pertinente à l'évaluation de la propriété, et il doit être possible d'activer immédiatement le moniteur correspondant. Par conséquent, les composants SystemC qui implémentent les moniteurs ne sont pas des processus SystemC avec liste de sensibilité (contrairement aux modules VHDL décrit dans la section 4.1.1) : leur méthode `update` est appelé au moment

opportun. Le mécanisme de notification peut être schématisé comme montré sur la figure 4.9.

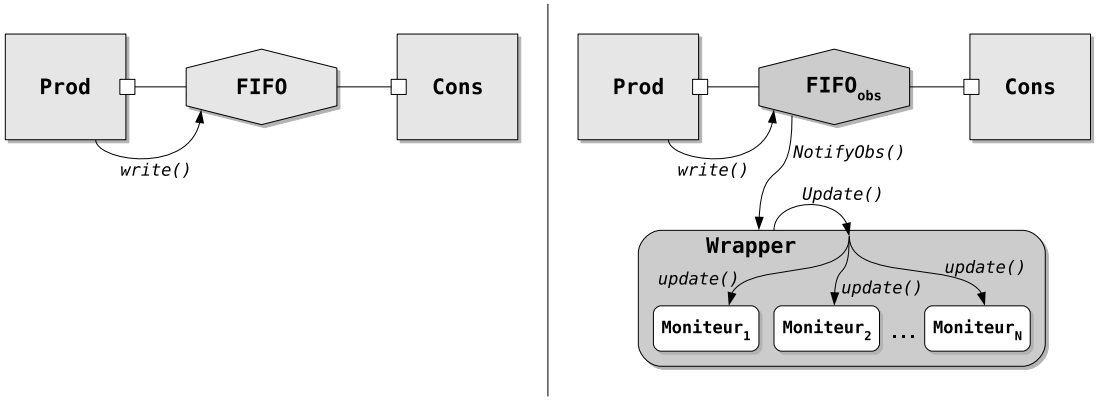

FIGURE 4.9 – Mécanisme d'observation

Quand une propriété porte sur les opérations dans la FIFO, par exemple sur ses écritures, une sous-classe de la FIFO est automatiquement générée comme expliqué ci-dessus. L'instance de la FIFO dans le design est remplacée par une instance FIFO$_{obs}$ de la sous-classe observée. Grâce à l'héritage multiple, FIFO$_{obs}$ comporte les mêmes fonctionnalités que la FIFO, plus les aspects liés à sa nature de sujet observé (classe `Subject`). Chaque écriture provoque la notification du *wrapper* qui englobe le moniteur de surveillance. À son tour, le *wrapper* actionne l'évaluation du moniteur en appelant sa fonction `update`, ceci après lui avoir transmis les valeurs des paramètres observés (par exemple, la donnée écrite dans la FIFO). Cette transmission s'effectue

- par passage de paramètre quand FIFO$_{obs}$ notifie le *wrapper* : la notification comporte aussi une structure regroupant les valeurs de toutes les données observées (paramètres et valeurs de retour de la méthode observée, comme expliqué dans la section suivante) ;
- par affectation de variables, et donc "effet de bord", quand le *wrapper* actionne l'évaluation du moniteur.

Souvent les méthodes de communication, comme la méthode `write` ci-dessus, sont déclarées comme *virtuelles*[4]. En cas de surveillance d'une méthode non virtuelle, il est nécessaire de changer la spécification de cette méthode pour qu'elle soit déclarée virtuelle. Ceci est permis dans notre solution, contrairement à des solutions qui ne prévoient pas l'instrumentation du code source, comme celle proposée dans [dSS09] et étudiée dans la section 4.2.5.

4.2.3 Couche booléenne et choix syntaxiques

Lors de l'écriture des propriétés, en tirant avantage du fait que la couche booléenne de PSL s'appuie sur le langage de modélisation de matériel sous-jacent, il est possible d'inclure des appels aux fonctions dans le texte. Ceci s'avère très utile pour exprimer aisément des conditions complexes et, en particulier, pour désigner la présence d'une opération de

[4] Une méthode virtuelle permet d'effectuer le *late binding*, mécanisme selon lequel, après identification de la classe exacte d'un objet en cours d'exécution du programme, la méthode membre appropriée de cette classe sera appelée (et non pas la méthode de sa classe parente, par exemple) [Eck03]

communication. A partir de l'ensemble d'opérations de communication choisies par l'utilisateur pour un composant donné, notre outil ISIS (décrit plus en détail dans le chapitre 6) génère automatiquement des méthodes booléennes *ad-hoc*, qui permettent d'indiquer l'*appel* ou le *retour* d'une fonction de communication, c'est-à-dire, le *début* ou la *fin* d'une transaction. Puisqu'une transaction peut se décomposer à son tour en plusieurs "sous-transactions", l'observation de la fin d'une transaction peut conduire à une notification de cet événement bien après le début de la transaction (et plusieurs autres sous-transactions peuvent avoir été effectuées entre-temps). L'utilisation d'interfaces de communication bloquantes ou non-bloquantes n'a pas d'impact négatif sur les possibilités offertes par le mécanisme d'observation : l'observation du début d'une transaction est identique dans les deux cas ; l'observation de la fin d'une transaction, en étant basée sur le retour de la fonction de communication associée, permet d'avoir une notification précise dans les deux cas.

Pour chaque méthode de communication choisie comme observée par l'utilisateur, ISIS génère trois méthodes booléennes en ajoutant les suffixes `_CALL`, `_START` et `_END` au nom de la fonction originale. Les deux premiers suffixes sont synonymes. Ainsi, par exemple, si l'utilisateur choisit d'observer les écritures dans une FIFO (méthode `write`), il pourra alors utiliser les expressions booléennes `fifo.write_CALL()`, `fifo.write_START()` et `fifo.write_END()` dans le texte de la propriété. Par ailleurs, il est possible de prendre en compte dans les assertions tous les paramètres et la valeur de retour d'une fonction, i.e. les données d'une communication. Syntaxiquement, cela s'obtient par la forme `composant.methode.p#`, où `#` dénote la position du paramètre désiré de la méthode `methode` (`p0` est la valeur de retour)[5]. Une structure qui réunit tous les éléments de la communication est automatiquement définie par ISIS. Lors de la notification à ses observateurs, le composant observé transmet un pointeur vers l'instance de cette structure qui contient toutes les valeurs courantes des paramètres. Il est évident qu'une valeur de retour ne peut être observée qu'en fin de transaction (suffixe `_END`).

Par défaut, l'ensemble de toutes les méthodes utilisées dans une propriété constitue implicitement la "liste de sensibilité" de cette propriété. Comme rappelé aussi dans [Lah06] (cf. section 3.2.2), PSL prévoit l'opérateur `@` pour définir des expressions booléennes d'horloge(s) qui vont échantillonner la trace sur laquelle la propriété considérée sera évaluée. Dans notre contexte, l'utilisateur peut utiliser ce même opérateur pour indiquer explicitement la liste des méthodes booléennes de type `CALL`, `START` et `END` qui vont déterminer l'échantillonnage. Par exemple, la trace considérée pour la propriété suivante

```
always ( (x.write_START() && x.write.p1 > 0) ->
         next_event(y.read_END())(y.read.p0 > 0) )
```

est la suite ordonnée de tous les débuts d'écriture dans `x` et les fins de lecture dans `y`. Il en serait de même si on avait précisé :

```
(always ( (x.write_START() && x.write.p1 > 0) ->
          next_event(y.read_END())(y.read.p0 > 0) ))
@ (x.write_START() || y.read_END())
```

Une caractéristique cruciale du mécanisme d'observation décrit auparavant est le fait que, en termes de fonctionnalité, il n'est pas intrusif :

– toutes les versions observées des composants se limitent à notifier leurs observateurs par simple appel de fonction et à exécuter la méthode de communication originale,

[5] Il est également possible d'utiliser le nom du paramètre formel, si présent dans la déclaration de la méthode, par exemple : `bus.write.adress`.

– il n'y a pas de "retour" depuis un observateur ou un moniteur vers un composant du design,
– aucun mécanisme lié au simulateur (comme des notifications de `sc_event`) n'est utilisé pour l'observation, par conséquent la simulation n'est pas altérée.

4.2.4 Exemple : FIFO avec arbitre

La solution présentée dans ce chapitre peut-être résumée à l'aide de l'exemple décrit ci-dessous. D'autres études plus évoluées sont présentées dans le chapitre 7.

Dans ce cas d'étude, deux producteurs envoient des messages à un consommateur, chaque message étant une simple suite de caractères. La communication, contrôlée par un arbitre, se fait à l'aide d'une FIFO. Les deux producteurs utilisent respectivement les caractères `'&'` et `'!'` pour marquer le début de chaque message ; le caractère `'@'` dénote la fin d'un message. L'architecture se présente comme montré par la figure 4.10.

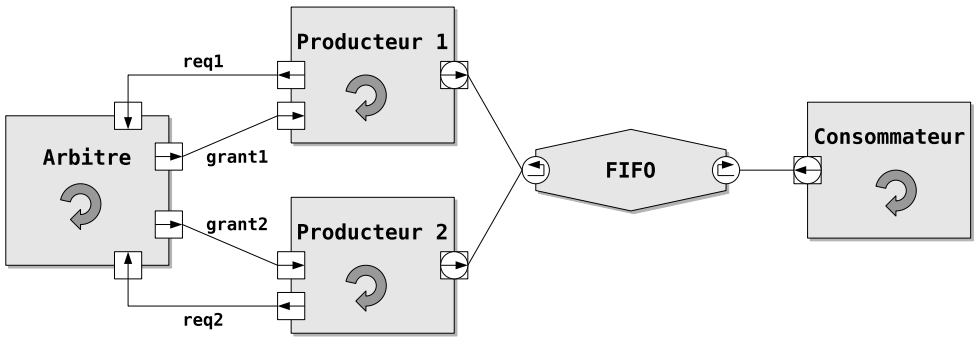

FIGURE 4.10 – Modèle de communication par FIFO avec arbitre

Cinq canaux de communication sont utilisés : les quatre signaux (canaux primaires) *req1*, *req2*, *grant1*, *grant2* permettent les échanges entre l'arbitre et les deux producteurs ; la FIFO permet la transmission des messages à destination du consommateur. L'arbitre utilise les signaux *grant1* et *grant2* pour accorder les droits de transmission aux producteurs.

Nous ne considérons ici qu'une seule propriété : *chaque fois qu'une permission d'écriture est accordée au premier producteur, c'est bien le premier producteur qui commencera son envoi*, autrement dit, lors de la prochaine écriture dans la FIFO après avoir accordé la permission au premier producteur, c'est bien le caractère de début de message du producteur 1 qui sera écrit. En PSL, et avec les conventions d'écriture de notre outil, cela peut se traduire par la formule 4.1 :

```
always ( grant1 -> next_event(fifo.write_CALL())(fifo.write.p1 == '&') )
```
(4.1)

Cette propriété fait intervenir simultanément un canal primaire (le signal *grant1*) et un canal hiérarchique simple (la FIFO). Pour les signaux booléens, nous admettons une simplification syntaxique qui rend les propriétés plus concises : la simple utilisation du nom d'un signal dans la propriété est équivalente à l'observation d'un front montant sur ce signal. Quand l'utilisateur énonce la propriété 4.1, il choisit d'observer les changements de valeur dans le signal et les écritures dans la FIFO. Il faut remarquer que les propriétés énoncées de cette manière sont génériques, dans la mesure où les noms des variables (ici

`grant1` et `fifo`) ne sont pas significatifs. Chaque identifiant est ensuite associé par l'utilisateur à un composant du design et ISIS se charge de générer l'infrastructure nécessaire, c'est-à-dire l'ensemble de classes C++/SystemC utilisées par le mécanisme d'observation.

En premier lieu, à partir de l'énoncé de la formule PSL 4.1, un moniteur complet est généré par "interconnexion" des sous-classes des moniteurs élémentaires de la bibliothèque. La classe associée à ce moniteur complet crée l'instance du moniteur élémentaire "le plus à droite" dans la formule. Dans sa fonction d'évaluation, il comporte un appel à la fonction d'évaluation à ce moniteur élémentaire :

```
// Contructeur du moniteur :
prop_fifo_mnt::prop_fifo_mnt(const char * name) : Monitor(name) {
  // Instances d'une classes qui reunit des donnees associees a
  //  chaque moniteur :
  data = new prop_fifo_data();
  ...
  // Creation de l'instance du moniteur elementaire "next_event" :
  data->next_event_1_inst = new next_event_1(..., data);
  data->init_cycle = true;
  // Compteur du nombre de violations :
  data->nb_fails = 0;
  ...
}

// Fonction d'evaluation du moniteur :
void main_mnt::update() {
  ...
  // Evaluation du moniteur elementaire "next_event" :
  data->next_event_1_inst->update();
  // Obtention des sorties finales :
  data->Valid = data->valid_next_event_1;
  data->Checking = data->checking_next_event_1;
  // Affichages...
  ...
  if( ! data->Valid ) {
    data->nb_fails++;
    // Affichages...
    ...
  }
}
```

Ensuite, une sous-classe de `sc_signal` et une de `Fifo` sont générées afin de pouvoir observer les communications appropriées. Ces classes suivent le même principe présenté avec l'exemple de FIFO observée de la section 4.2.2. Pour le signal, la différence est que la notification à l'observateur s'effectue lors de sa phase de mise à jour.

Une sous-classe de *Wrapper* est aussi générée : il s'agit de l'observateur attaché aux deux instances des deux composants observés. Cet observateur englobe le moniteur décrit ci-dessus et contrôle son évaluation. Il contient donc un pointeur vers chacune des instances des sous-classes observées :

```
prop_fifo_wrap::prop_fifo_wrap(
  ..., observed_sc_signal * s1, observed_Fifo * s2) : Wrapper(n)
{
  ...
  // Liaison du premier sujet observe, le signal :
  signal_subject = s1;
  signal_subject->Attach(this);
```

```
  // Liaison du deuxieme sujet observe, la fifo :
  fifo_subject = s2;
  fifo_subject->Attach(this);
  // Creation et liaison du moniteur pour la propriete :
  prop1 = new prop_fifo::prop_fifo_mnt("prop1");
  Attach_monitor(prop1);
}
```

La méthode `Update` héritée de la classe *Observer* est ici surchargée pour contrôler l'évaluation du moniteur pour la propriété 4.1. Cette méthode est appelée par l'intermédiaire de la fonction de notification `NotifyObs()` des sujets observés.

Enfin, les déclarations des instances des composants observés remplacent les déclarations originales. Ainsi, dans cet exemple, le type du signal *grant1* et le type de la FIFO sont remplacés par ceux des sous-classes observés dans le module *Top* :

```
class Top : public sc_module {
  public:
    Arbiter arbiter_inst;
    Producer producer1_inst, producer2_inst;
    // Le type declare de la fifo est remplace par celui de la
    //  sous-classe observee :
    observed_Fifo fifo_inst;
    Consumer consumer_inst;
    // La declaration du signal grant1 est isolee des autres et son
    //  type est remplace par celui de la sous-classe observee :
    sc_signal<bool> req1, req2, grant2;
    observed_sc_signal grant1;
  ...
}
```

Si certains composants "à surveiller" sont déclarés privés, alors soit une déclaration d'amitié soit des méthodes d'accès sont ajoutées dans la classe qui les contient.

4.2.5 Approches alternatives

Le mécanisme d'observation présenté dans ce chapitre offre un certain nombre d'avantages qui le rendent applicable à une panoplie de situations différentes. En premier lieu, une caractéristique évidemment indispensable est le fait qu'il n'altère pas la fonctionnalité du design (cf. section 4.2.3). En second lieu, bien qu'appliquée au contexte SystemC TLM, la méthode reste très générique. En C++ nous avons exploité le potentiel du langage et nous nous sommes appuyés sur l'héritage multiple [Str89] pour offrir une solution élégante lors de la création des sous-classes pour les composants observés (cf. section 4.2.2). La solution n'est pas limité à ce langage, au contraire elle convient dans toute modélisation reposant sur le paradigme orienté objet. Dans un langage qui n'offrirait pas l'héritage multiple, il suffirait, à la génération automatique d'une classe observée, d'y inclure le corps des méthodes d'un *Subject*. Par ailleurs, le fait d'observer les méthodes d'une classe quelconque permet d'étendre la portée de la solution à d'autres cas de figure, non nécessairement liés aux communications entre blocs matériels. Enfin, il s'agit d'une solution automatisée, indépendante du simulateur et qui offre des surcoûts modérés en termes de temps de simulation (voir chapitre 7).

Cependant, l'infrastructure d'observation nécessite la génération de plusieurs classes et requiert des efforts conséquents pour l'analyse syntaxique du code source du design. La mise en œuvre d'une automatisation complète, capable de couvrir un très grand nombre

de cas, a réclamé un travail non négligeable. Par ailleurs, la solution que nous avons proposée comporte des changements dans la partie déclarative du design, par exemple pour pouvoir remplacer les types des instances des composants observés.

La solution de Ecker *et al.* (cf. section 3.2.2) est une alternative à notre mécanisme d'observation par insertion de modules mandataires entre les composants. L'avantage est qu'elle ne change pas le types des composants quand des communications sont observées. Par contre, cette solution entraîne l'insertion de nouveaux modules dans le design et elle ne permet pas d'observer *toutes* les méthodes d'un composant, mais se limite à celles qui transitent sur les ports.

La surveillance et l'enregistrement des transaction fait aussi partie des aspects liés à la SCV (*SystemC Verification Library*) [IS03, Gro03]. Il s'agit d'une bibliothèque conçue par l'OSCI comme une extension de la bibliothèque SystemC, pour assister l'utilisateur durant les étapes de génération de jeux de tests et de vérification dynamique. Les fonctionnalités apportées par la SCV incluent [IS03]

- l'introspection des données, permettant d'extraire des informations (par exemple le nombre de champs d'une structure) à partir d'objets de types quelconques, après définition d'extensions ;
- la génération aléatoire contrainte et pondérée,
- l'enregistrement des transactions,
- des aides à la gestion des exceptions.

L'enregistrement des transactions s'appuie sur une interface de programmation (API, pour *Application Programming Interface*) qui offre des fonctions pour créer et manipuler des bases de données et des flux de transactions (classes `scv_tr_db` et `scv_tr_stream`), ainsi que pour créer et enregistrer des transactions (classes `scv_tr_generator` et `scv_tr_handle`). Il est important de remarquer que, même dans ce cadre, la connotation de la transaction doit être définie manuellement par l'utilisateur : via les méthodes de l'API, il est possible de signaler qu'une transaction est en train de démarrer ou de se conclure. Par exemple, des transactions simples peuvent être manipulées à l'aide des méthodes `begin_transaction()` et `end_transaction()` de la classe `scv_tr_generator` [Gro03] :

```
class MyComponent : ... {
  // Un flux de transactions permet d'enregistrer un ensemble de
  // transactions et de les grouper au sein d'une bases de donnees :
  scv_tr_stream r_stream;
  // Un generateur de transactions permet de creer des transactions
  // qui peuvent etre ajoutees a une bases de donnees :
  scv_tr_generator<addr_t, data_t> read_gen;
  ...
  MyComponent(...) : ... ,  r_stream("ReadOperations"),
                            read_gen("Read", r_stream, ...) { ... }
  ...
  // Dans l'une des fonctions du composant, en plus des instructions
  // liees a l'operation en question, on peut utiliser l'API de la SCV
  // pour creer et enregistrer chaque nouvelle transaction :
  data_t read(const addr_t * addr) {

    // Creation d'une nouvelle transaction a l'instant courant
    // avec l'attribut initial "*addr" :
    scv_tr_handle h = read_gen.begin_transaction(*addr);
    ...
    // Ajout eventuel d'un autre attribut a la transaction :
    if( ... )
```

```
    h.record_attribute("other_attribute", other_attr);
  ...
  // Fin de la nouvelle transaction a l'instant courant
  // et avec l'attribut final "data" :
  read_gen.end_transaction(h, data);
  return data;
}
...
```

Comme la solution de Ecker *et al.* ou la nôtre, cette riche API permet la mise en œuvre d'un mécanisme de notification, ou d'enregistrement pour l'analyse des traces de simulation *à posteriori*. Pour cela, la définition de classes utilisateur spécifiques avec les méthodes de cette API (ou l'instrumentation de certaines classes du design par ces mêmes méthodes) reste inévitable. La version courante de la bibliothèque[6] ne fournit pas de solution pour la vérification de formules temporelles.

Dans [dSS09], da Silva et Sanchez proposent une technique inspirée de l'instrumentation dynamique du fichier binaire de l'application (DBI, *Dynamic Binary Intrumentation*) dans le but d'intercepter les appels de certaines fonctions sans modification ou instrumentation du code source du design. Chaque fois qu'une classe C++ possède des méthodes virtuelles, soit par déclaration directe soit par héritage, le compilateur crée une VTABLE, une table unique associée à la classe et comportant les adresses des fonctions virtuelles. Les auteurs de [dSS09] préconisent alors la modification des VTABLEs pour certaines instances en remplaçant l'adresse d'une fonction f par celle d'une fonction g qui réalise des instructions additionnelles avant d'appeler f. Cette solution permettrait de surveiller l'exécution de certaines méthodes, en offrant une alternative à toutes les techniques de surveillance décrites auparavant et basées sur la modification du code source : son avantage principal est précisément celui de ne pas intervenir dans le code source. Cependant, cette solution est limitée aux méthodes virtuelles. Par ailleurs, les auteurs ne précisent pas s'il est possible de choisir finement les actions à exécuter autour de la fonction originelle, par exemple en effectuant des tâches avant et après cette fonction. Enfin, quelle que soit l'approche choisie, le comportement global de l'application doit être altéré afin de pouvoir insérer des nouveaux traitements. Puisque cette modification est nécessaire, nous estimons qu'elle est préférable au niveau du code source plutôt qu'au niveau de l'exécutable. En effet, en générant un nouveau code source instrumenté, l'utilisateur en aura une meilleure visibilité, et il pourra encore le modifier ou l'instrumenter ultérieurement pour d'autres besoins.

[6] *SystemC Verification Library release 1.0p2* (`http://www.systemc.org/downloads/standards`).

Chapitre 5

Propriétés avec variables auxiliaires

Introduction

La surveillance des données d'une transaction est vitale pour les propriétés les moins élémentaires. Se limiter uniquement à l'observation les occurrences des communications restreindrait énormément la portée de la vérification, puisque souvent la valeur d'un ou plusieurs paramètres d'une transaction permet d'identifier et d'analyser l'action en cours. Par exemple, l'inspection de l'adresse destination d'une requête d'écriture lors de la programmation du contrôleur DMA présentée dans la section 2.1.3.2 (cf. figure 2.8, page 19) permet d'identifier l'étape de la programmation (transmission de la longueur du transfert, transmission de l'adresse source pour la copie mémoire, etc.).

Toutefois, se contenter d'*observer* les données n'est pas non plus suffisant. Parfois il est nécessaire de *mémoriser* ces données pour les utiliser plus tard, par exemple lors d'une comparaison dans une expression booléenne. Considérons encore une fois la plate-forme avec contrôleur DMA : pour s'assurer que l'adresse de début de lecture `src_addr`, écrite dans le registre approprié du DMA, sera utilisée pour la première lecture en mémoire lors du début du transfert, il faut pouvoir mémoriser la valeur de `src_addr` quand le DMA est programmé. Ensuite, il faut comparer la valeur mémorisée avec celle réellement utilisée quand la première lecture en mémoire a lieu.

PSL comporte une *couche modélisation* (cf. section 2.2.3.1) qui permet d'introduire des variables auxiliaires. Ces variables peuvent être utilisées pour stocker certaines valeurs à des moments précis, mais elles peuvent surtout être *modifiées* plusieurs fois durant la vérification de la propriété. Un exemple est celui d'un compteur pour le nombre d'opérations d'écriture dans une plage spécifique en mémoire : la valeur du compteur est incrémentée chaque fois qu'une écriture à l'adresse opportune est effectuée. La couche modélisation ne permet pas uniquement de déclarer des variables auxiliaires, mais aussi de caractériser leur évolution par le moyen de blocs d'instructions.

Nous ne connaissons aucune sémantique qui mette en relation les instructions de la couche modélisation avec les propriétés dans une *vunit*, le manuel de référence de PSL se limitant uniquement à une caractérisation syntaxique. Plus exactement, la version en syntaxe SystemC de la couche modélisation peut comporter toutes les *déclarations* légales dans le contexte d'un `sc_module`, ainsi que toutes les *instructions* autorisées dans le corps du constructeur du module [PSL05, PSL10]. Le code relatif à la couche modélisation peut être inséré au même niveau que les propriétés dans la *vunit*. Cependant, l'absence

d'une sémantique ne permet pas de décrire précisément le fonctionnement des propriétés avec variables auxiliaires. Ces variables ne font pas partie du design : bien que liées à la syntaxe du langage de description de matériel sous-jacent, il est essentiel de spécifier *quand et comment* elles évoluent, cela par rapport à l'évaluation de la propriété. Les moments où les instructions de la couche modélisation sont exécutées, et leur impact sur les variables et sur les propriétés, doivent être définis.

Dans la section 5.1 nous proposons une sémantique qui comprend les variables globales introduites via la couche modélisation, et nous présentons les principes de sa mise en œuvre dans notre outil.

La caractérisation sémantique des variables de la couche modélisation est un point de départ naturel, mais les variables introduites de cette manière ne permettent pas de couvrir toutes les relations entre les transactions. Comme identifié dans la figure 3.5 à la page 54, les transactions peuvent se dérouler en pipeline dans un canal de communication. Quand une variable globale est utilisée pour mémoriser la donnée d'une communication, uniquement le cas où les transactions ne se superposent pas peut être considéré. Comme mentionné dans la section dédiée aux variables locales de SVA, à la page 31, dans certaines situations, les valeurs des expressions dans une assertion doivent être mémorisées de façon indépendante alors que plusieurs tentatives d'évaluation de l'assertion coexistent. Par exemple, lors de la vérification de designs en pipeline, chaque nouvelle donnée qui entre dans le flot nécessite d'être mémorisée dans une variable indépendante afin de pouvoir être utilisée plus tard.

Nous avons donc étendu la sémantique proposée dans la section 5.1 pour considérer un phénomène très proche de celui de *fonction réentrante*, afin de modéliser plusieurs évaluations distinctes et concurrentes d'une même propriété. Ces concepts, ainsi qu'une étude des travaux connexes existants, sont présentés dans la section 5.2.

Le travail présenté dans ce chapitre a fait l'objet des publications [FP10a] et [PF10].

5.1 Couche modélisation et variables globales

La *syntaxe* d'un langage définit sa structure grammaticale, c'est-à-dire la façon dont les lexèmes peuvent se combiner pour former des instructions ou des formules. Ainsi, la syntaxe de PSL nous indique par exemple que l'opérateur **always** doit toujours être suivi par une sous formule φ, car seul il ne peut jamais constituer une propriété. La *sémantique* du langage définit alors la signification des instructions ou des formules grammaticalement correctes. Dans la section 2.2.3.3 du chapitre 2 nous avons rappelé la sémantique de certains opérateurs de PSL, dont **always** : **always** φ signifie que φ doit être satisfaite à tout instant de la trace.

La façon de formaliser les notions relatives à la sémantique d'un langage peut suivre des démarches différentes, qui sont souvent regroupées selon trois familles [NN99] :

- sémantique *opérationnelle*, où la signification d'une instruction ou formule f est spécifiée par les calculs que f comporte quand elle est évaluée ;
- sémantique *dénotationnelle*, où la signification de f est spécifiée par des objets mathématiques qui représentent l'effet de l'exécution de f ;
- sémantique *axiomatique*, où les propriétés dérivées de l'effet de l'exécution de f sont exprimées par des équations.

En particulier, la sémantique opérationnelle s'intéresse principalement à *comment* l'effet du calcul se produit, alors que la sémantique dénotationnelle s'intéresse à l'*effet* lui-même.

5.1.1 Une sémantique opérationnelle pour PSL

La sémantique de PSL décrite dans l'annexe B de [PSL05] est une sémantique définie sur des traces d'exécution. Elle utilise une notion inductive de satisfaction, notée $\models_F$, qui calcule la satisfaction d'une propriété sur une trace donnée (cf. section 2.2.3.3). Dans un premier temps, il était naturel d'envisager une extension de cette sémantique afin d'introduire le support pour les variables globales de la couche modélisation.

Nous rappelons qu'une vue simplifiée d'une *vunit* munie d'une partie "couche modélisation" se présente comme suit :

```
vunit v {
  d   // Declarations (partie declarative de la couche modelisation)
  m   // Modelisation (bloc d'instructions de la couche modelisation)
  Phi // Assertion
}
```

Pour pouvoir attribuer une sémantique formelle à une assertion φ enrichie par un bloc m d'instructions de la couche modélisation, il faut avant tout formaliser le fait que les affectations de m provoquent des "effets de bord"[1] sur les variables qui peuvent être utilisées dans φ. Il est important de pouvoir caractériser, à chaque étape de l'évaluation de l'assertion, le fait que, même si son niveau de satisfaction n'a pas changé, les variables auxiliaires peuvent avoir été mises à jour.

La sémantique de PSL ne comporte pas cet aspect, puisqu'elle se concentre sur l'effet d'une formule, ou, plus exactement, sur sa satisfaction. Par conséquent, cette sémantique ne fournit pas indication quant à la manière de vérifier dynamiquement une formule [CM04].

5.1.1.1 *Weak* PSL

Dans [CM04], les auteurs définissent une sémantique opérationnelle pour *Weak* PSL, un sous-ensemble du langage apte à spécifier des propriétés de sûreté[2]. Pour cela, ils adoptent un style nommé *structural operational semantics* en anglais, selon lequel les règles énoncées sont dirigées par la syntaxe [Win93]. Il s'agit d'une sémantique "à petits pas", lettre par lettre. Le principe général[3], ici présenté de façon intuitive, est le suivant : pour vérifier si un mot $v = (\ell_0\ell_1\ell_2...\ell_n)$ satisfait la formule PSL φ, il est possible de vérifier si le mot $v' = (\ell_1\ell_2...\ell_n)$, sans la lettre initiale ℓ_0, satisfait une nouvelle formule φ'. Ce principe est itéré jusqu'à la fin du mot/trace.

Le langage WPL (*Weak Property Language*) défini dans [CM04] comporte les booléens, les expressions rationnelles, la conjonction et la disjonction de formules WPL, l'opérateur *next* faible X, l'opérateur *until* faible W, l'implication suffixe et l'opérateur *abort*. Pour les raisons exposées dans les chapitres précédents, nous ne nous focalisons pas sur les expressions rationnelles. Par ailleurs, nous considérons l'opérateur de négation uniquement

[1] De l'expression anglaise *side effect*, qui dénote un effet secondaire autre que celui directement visible.

[2] Il s'agit de propriétés basées essentiellement sur la version faible des opérateurs temporels, et qui expriment le fait qu'une erreur ne se produira jamais. Nous verrons comment enrichir la sémantique pour pallier en partie cette limitation.

[3] Le principe présenté par Claessen et Mårtensson est inspiré des travaux de Brzozowski [Brz64].

au niveau booléen, ce qui est cohérent avec la notion de sous-ensemble simple de PSL introduite dans la section 2.2.3. Enfin, sauf précision, les symboles utilisés par la suite sont issus du manuel de référence de PSL et ils ont été présentés dans la section d'introduction à ce langage.

La sémantique opérationnelle de WPL se base essentiellement sur quatre concepts :

1. les règles opérationnelles, qui montrent les dérivations des formules, i.e. le passage de φ à φ', pour tous les opérateurs de base ;
2. l'application itérative, lettre par lettre, des règles précédentes sur une formule et un mot donnés, notée $\varphi\langle v\rangle$;
3. une fonction, nommée ok, qui définit si une formule donnée n'a pas encore été contredite, compte tenu de la séquence de lettres déjà visitées ;
4. la notion de satisfaction d'une formule φ par un mot v, qui utilise la fonction ok pour déterminer le résultat de l'application itérative $\varphi\langle v\rangle$ des règles opérationnelles sur la formule pour la totalité du mot : $v \models \varphi \Leftrightarrow \mathsf{ok}(\varphi\langle v\rangle)$.

Par exemple, parmi les règles opérationnelles de WPL (point 1 ci-dessus), celle pour l'opérateur *next*, sans prémisse, entraîne l'abandon de l'opérateur entre deux lettres :

$$X\,\varphi \xrightarrow{\ell} \varphi \tag{5.1}$$

L'explication intuitive est très simple : vérifier $X\,\varphi$ sur une trace $v = (\ell\ell_1...\ell_n)$ équivaut à vérifier φ sur la trace $v^{1..} = (\ell_1...\ell_n)$.

La règle 5.2 relative à l'opérateur *until* se comprend de la façon suivante : si φ et ψ évoluent respectivement en φ' et ψ' par la lettre ℓ, alors $\varphi\,W\,\psi$ évolue en $\psi' \vee (\varphi' \wedge (\varphi\,W\,\psi))$, c'est-à-dire soit ψ' est satisfaite, soit φ' l'est et l'évaluation du *until* continue.

$$\frac{\varphi \xrightarrow{\ell} \varphi' \qquad \psi \xrightarrow{\ell} \psi'}{\varphi\,W\,\psi \xrightarrow{\ell} \psi' \vee (\varphi' \wedge (\varphi\,W\,\psi))} \tag{5.2}$$

Les règles pour la conjonction et la disjonction sont identiques et nécessitent simplement l'application de la dérivation aux deux opérandes. Ici nous rappelons uniquement la première :

$$\frac{\varphi \xrightarrow{\ell} \varphi' \qquad \psi \xrightarrow{\ell} \psi'}{\varphi \wedge \psi \xrightarrow{\ell} \varphi' \wedge \psi'} \tag{5.3}$$

Enfin, pour tout booléen b, les auteurs définissent la règle suivante :

$$b \xrightarrow{\ell} \begin{cases} T & \text{si } \ell \Vdash b \ \text{ (ce qui signifie que } b \text{ est satisfait par } \ell) \\ F & \text{sinon} \end{cases} \tag{5.4}$$

L'application itérative (point 2 ci-dessus) des règles opérationnelles est définie de façon inductive comme suit :

$$\begin{aligned} \varphi\langle\epsilon\rangle &= \varphi \\ \varphi\langle\ell v\rangle &= \varphi'\langle v\rangle, \text{ où } \varphi \xrightarrow{\ell} \varphi' \end{aligned}$$

et la fonction ok est définie de la façon suivante :

$$
\begin{array}{rcl}
\mathsf{ok}(F) & = & \textbf{false} \\
\mathsf{ok}(T) & = & \textbf{true} \\
\mathsf{ok}(b) & = & \textbf{true} \\
\mathsf{ok}(\varphi \wedge \psi) & = & \mathsf{ok}(\varphi) \text{ et } \mathsf{ok}(\psi) \\
\mathsf{ok}(\varphi \vee \psi) & = & \mathsf{ok}(\varphi) \text{ ou } \mathsf{ok}(\psi) \\
\mathsf{ok}(X\,\varphi) & = & \textbf{true} \\
\mathsf{ok}(\varphi\, W\, \psi) & = & \mathsf{ok}(\varphi) \text{ ou } \mathsf{ok}(\psi)
\end{array}
$$

Par exemple, considérons la propriété $P_1 = X\,a$ et le mot $v_1 = (\ell_0\ell_1)$, avec $a \notin \ell_0$ et $a \in \ell_1$. On souhaite savoir si $v_1 \vDash P_1$, ce qui revient à calculer $\mathsf{ok}(X\,a\langle\ell_0\ell_1\rangle)$. Puisque $X\,a \xrightarrow{\ell_0} a$, on a $X\,a\langle\ell_0\ell_1\rangle = a\langle\ell_1\rangle$. Ensuite, $a\langle\ell_1\rangle = T\langle\epsilon\rangle$, parce que $a \xrightarrow{\ell_1} T$. Enfin, $\mathsf{ok}(T) = \textbf{true}$. En conclusion, P_1 est donc satisfaite sur v_1.

5.1.2 Proposition d'une extension pour la sémantique opérationnelle

Nous avons étendu la sémantique WPL de [CM04] pour intégrer les variables de la couche modélisation. Pour cela, nous avons ajouté la notion d'environnement (associations de variables et valeurs) et celle de "bloc d'instructions" de la couche modélisation. Toutes les règles de la sémantique opérationnelle, sauf celles pour les SEREs, ont été reprises pour inclure ces éléments. Ici, la différence majeure est que, en plus de l'évolution de la formule PSL le long de la trace, nous considérons aussi celle de l'environnement. Ensuite, nous avons caractérisé trois des quatre niveaux de satisfaction d'une formule (cf. section 2.2.3.4), en combinant la fonction ok de [CM04], ici complétée pour les opérateurs forts, avec une nouvelle fonction met. Cela nous permet de considérer également les opérateurs forts dans les nouvelles règles de la sémantique opérationnelle. Enfin, nous avons implémenté ces principes pour nos moniteurs orientés TLM dans l'outil ISIS.

5.1.2.1 Notations

Quelques notions utilisées par notre sémantique étendue ont été ajoutées à celles d'alphabet Σ, de mot sur cet alphabet et d'ensemble $\mathcal{FL}$ des expressions FL de PSL. L'ensemble $\mathcal{DV}$ est l'ensemble de toutes les variables déclarées dans la couche modélisation, le domaine de ces variables est noté $\mathcal{D}$. Un bloc d'instructions de la couche modélisation, indiqué par m, appartient à l'ensemble $\mathcal{M}$ des instructions C++, dont une définition syntaxique peut être trouvée par exemple dans [CPP03]. L'environnement courant ρ est la liste d'associations de chaque variable globale avec sa valeur, c'est-à-dire un élément de l'ensemble $\mathcal{P} = \mathcal{DV} \to \mathcal{D}$. Par exemple, l'environnement $\rho_1 = [x \mapsto 3, y \mapsto 1]$ comporte les deux variables x et y dont les valeurs respectives sont 3 et 1. On écrit aussi $\rho_1(x) = 3$ et $\rho_1(y) = 1$.

Pour indiquer "l'interprétation" d'une formule PSL ou d'un bloc d'instructions de la couche modélisation, nous adoptons une notation similaire à celle de [NN99] lors de l'application des *fonctions sémantiques*. Ainsi, $[\![\varphi]\!]^m_\rho$ dénote l'interprétation de φ, où les variables de $\mathcal{DV}$ ont été remplacées par les valeurs qui leur sont associées dans ρ, compte tenu du bloc d'instructions m. De même, $[\![m]\!]^\ell_\rho$ est l'interprétation de m pour la lettre ℓ et pour l'environnement ρ ; cette interprétation rend un nouvel environnement. Formellement

- $[\![\varphi]\!]^m_\rho : \mathcal{FL} \times \mathcal{M} \times \mathcal{P} \to \mathcal{FL}$
- $[\![m]\!]^\ell_\rho : \mathcal{M} \times \Sigma \times \mathcal{P} \to \mathcal{P}$

La dérivation $\xrightarrow{\ell}$ de [CM04] est ici appliquée aux interprétations : $[\![\varphi]\!]^m_\rho \xrightarrow{\ell} [\![\varphi']\!]^m_{\rho'}$ signifie donc que, afin de vérifier si un mot qui commence par la lettre ℓ satisfait $[\![\varphi]\!]^m_\rho$, il est possible de vérifier si $[\![\varphi']\!]^m_{\rho'}$ est satisfait par le suffixe du mot sans ℓ. Outre le changement de φ en φ', il est important de remarquer l'évolution de l'environnement ρ dans la dérivation.

5.1.2.2 Règles sémantiques et satisfaction d'une formule

Définition 1 Notre extension [FP10a] de la sémantique opérationnelle est définie par les règles 5.5 à 5.14[4]. Comme dans [CM04], la conjonction et la disjonction (règles 5.5 et 5.6) nécessitent simplement l'application des règles aux deux opérandes.

$$\frac{[\![\varphi]\!]^m_\rho \xrightarrow{\ell} [\![\varphi']\!]^m_{\rho'} \qquad [\![\psi]\!]^m_\rho \xrightarrow{\ell} [\![\psi']\!]^m_{\rho'}}{[\![\varphi \wedge \psi]\!]^m_\rho \xrightarrow{\ell} [\![\varphi' \wedge \psi']\!]^m_{\rho'}} \tag{5.5}$$

$$\frac{[\![\varphi]\!]^m_\rho \xrightarrow{\ell} [\![\varphi']\!]^m_{\rho'} \qquad [\![\psi]\!]^m_\rho \xrightarrow{\ell} [\![\psi']\!]^m_{\rho'}}{[\![\varphi \vee \psi]\!]^m_\rho \xrightarrow{\ell} [\![\varphi' \vee \psi']\!]^m_{\rho'}} \tag{5.6}$$

Dans ces deux règles, et dans toutes celles qui suivent, $\rho' = [\![m]\!]^\ell_\rho$ est le nouvel environnement obtenu après exécution de m dans l'ancien environnement et dans le contexte des propositions atomiques de ℓ, comme expliqué précédemment. Par conséquent, si les deux formules φ et ψ deviennent φ' et ψ' dans le nouvel environnement ρ', alors leur conjonction (resp. disjonction) devient la conjonction (resp. disjonction) de φ' et ψ' dans ce même nouvel environnement ρ'.

L'opérateur *next* (règles 5.7 et 5.8) est simplement abandonné quand la lettre est consommée, et la nouvelle interprétation de la formule se fait dans le nouvel environnement.

$$[\![X!\varphi]\!]^m_\rho \xrightarrow{\ell} [\![\varphi]\!]^m_{\rho'} \tag{5.7}$$

$$[\![X\varphi]\!]^m_\rho \xrightarrow{\ell} [\![\varphi]\!]^m_{\rho'} \tag{5.8}$$

Comme pour l'opérateur *next*, nous donnons la même définition aux versions forte et faible de l'opérateur *until* (règle 5.9) : leur distinction est opérée plus loin, lors de la définition de la fonction met et donc de celle des niveaux de satisfaction *Holds* et *Pending*.

$$\frac{[\![\varphi]\!]^m_\rho \xrightarrow{\ell} [\![\varphi']\!]^m_{\rho'} \qquad [\![\psi]\!]^m_\rho \xrightarrow{\ell} [\![\psi']\!]^m_{\rho'}}{[\![\varphi\, U\, \psi]\!]^m_\rho \xrightarrow{\ell} [\![\psi' \vee (\varphi' \wedge (\varphi\, U\, \psi))]\!]^m_{\rho'}} \tag{5.9}$$

Comme pour tous les autres opérateurs, la nouvelle interprétation, à droite de la relation $\xrightarrow{\ell}$ dans la conclusion de la règle, s'effectue dans le nouvel environnement ρ'. Dans le cas spécifique du *until*, nous pouvons remarquer que ρ' est commun aux nouvelles sous-formules φ' et ψ', mais surtout qu'il est utilisé lors de la réévaluation de $\varphi\, U\, \psi$ dans le nouveau mot sans ℓ. Ceci souligne l'évolution systématique de l'environnement, et donc des variables globales, avec la progression de la trace de simulation.

[4] Nous rappelons que les règles pour les opérateurs $\wedge$, $\vee$, X, W et *abort* étaient définies dans [CM04] pour les formules WPL simples, sans les notions liées à l'interprétation $[\![\,]\!]$ et à la couche modélisation.

Les règles pour les opérateurs *always* (règle 5.10) et *eventually* (règle 5.11) sont caractérisées comme suit :

$$\frac{[\![\varphi]\!]^m_\rho \xrightarrow{\ell} [\![\varphi']\!]^m_{\rho'}}{[\![G\,\varphi]\!]^m_\rho \xrightarrow{\ell} [\![\varphi' \wedge G\,\varphi]\!]^m_{\rho'}} \tag{5.10}$$

$$\frac{[\![\varphi]\!]^m_\rho \xrightarrow{\ell} [\![\varphi']\!]^m_{\rho'}}{[\![F\,\varphi]\!]^m_\rho \xrightarrow{\ell} [\![\varphi' \vee F\,\varphi]\!]^m_{\rho'}} \tag{5.11}$$

La règle pour les expressions booléennes (règle 5.12) utilise les interprétations T ("définitivement vrai") et F ("définitivement faux"). Par conséquent, $\forall \ell \in \Sigma$, $T \xrightarrow{\ell} T$ et $F \xrightarrow{\ell} F$, quels que soient l'environnement ou le bloc d'instructions de la couche modélisation considérés.

$$[\![b]\!]^m_\rho \xrightarrow{\ell} \begin{cases} T & \text{si } \ell \Vdash_{\rho'} b \\ F & \text{sinon} \end{cases} \tag{5.12}$$

La règle 5.12 utilise la relation de satisfaction $\Vdash_\rho$ définie par $\ell \Vdash_\rho b \iff \ell \Vdash b_{[\mathcal{DV} \leftarrow \rho(\mathcal{DV})]}$, où $b_{[\mathcal{DV} \leftarrow \rho(\mathcal{DV})]}$ est l'expression booléenne b dans laquelle chaque identifiant de $\mathcal{DV}$ est remplacé par sa valeur dans ρ.

Enfin, l'opérateur *abort* a deux dérivations possibles (règles 5.13 et 5.14), en fonction de l'état de la formule et de la condition booléenne. Si l'opérande gauche n'est pas contredit dans son état actuel (fonction ok définie plus loin) et si la condition booléenne est satisfaite, alors la formule dérive en T : elle est valide et son évolution s'arrête (règle 5.13). Si la condition booléenne n'est pas satisfaite, alors l'opérateur *abort* est gardé, mais l'opérande gauche est dérivé par ℓ (règle 5.14).

$$\frac{[\![b]\!]^m_\rho \xrightarrow{\ell} T \qquad \mathsf{ok}(\varphi) = \mathbf{true}}{[\![\varphi \; abort \; b]\!]^m_\rho \xrightarrow{\ell} T} \tag{5.13}$$

$$\frac{[\![b]\!]^m_\rho \xrightarrow{\ell} F \qquad [\![\varphi]\!]^m_\rho \xrightarrow{\ell} [\![\varphi']\!]^m_{\rho'}}{[\![\varphi \; abort \; b]\!]^m_\rho \xrightarrow{\ell} [\![\varphi' \; abort \; b]\!]^m_{\rho'}} \tag{5.14}$$

Comme dans [PSL05], tous les autres opérateurs FL sont exprimés en fonction du *next* et du *until*. À ce propos, nous rappelons que la réécriture de certains opérateurs en fonction du *until*, bien qu'impliquant parfois la négation d'un opérande, est toujours possible dans le contexte du sous-ensemble simple de PSL.

Définition 2 En accord avec [CM04], l'application itérative de toutes ces règles à une formule φ sur les lettres d'un mot v, dans un environnement ρ, est notée $\varphi\langle v\rangle_\rho$ et définie récursivement :

$$\begin{aligned} \varphi\langle \epsilon\rangle_\rho &= \varphi \\ \varphi\langle \ell v\rangle_\rho &= \varphi'\langle v\rangle_{\rho'}, \text{ où } [\![\varphi]\!]^m_\rho \xrightarrow{\ell} [\![\varphi']\!]^m_{\rho'} \end{aligned}$$

Tant que le mot v n'est pas vide, les règles de la définition 1 sont utilisées pour calculer "l'effet de la transition" de v vers son suffixe privé de la première lettre ℓ. Autrement, la fonction φ cesse de changer.

Définition 3 La relation de satisfaction, avec l'environnement initial ρ_0, est définie par

$$v \vDash \varphi \iff \mathsf{ok}(\varphi\langle v\rangle_{\rho_0})$$

où la fonction $\mathsf{ok} : \mathcal{FL} \to \{\mathbf{true}, \mathbf{false}\}$ calcule si une formule donnée n'a pas encore été contredite, compte tenu de la séquence de lettres déjà visitées.

Définition 4 Dans la définition de la fonction ok de [CM04] nous avons ajouté uniquement les cas des opérateurs forts, pour lesquels le résultat est identique à leur contrepartie faible, ainsi que les opérateurs *always* (G) et *eventually* (F) :

$$\begin{array}{rcl}
\mathsf{ok}(F) & = & \mathbf{false} \\
\mathsf{ok}(T) & = & \mathbf{true} \\
\mathsf{ok}(b) & = & \mathbf{true} \\
\mathsf{ok}(\varphi \wedge \psi) & = & \mathsf{ok}(\varphi) \text{ et } \mathsf{ok}(\psi) \\
\mathsf{ok}(\varphi \vee \psi) & = & \mathsf{ok}(\varphi) \text{ ou } \mathsf{ok}(\psi) \\
\mathsf{ok}(X\,\varphi) & = & \mathbf{true} \\
\mathsf{ok}(X!\,\varphi) & = & \mathbf{true} \\
\mathsf{ok}(\varphi\,W\,\psi) & = & \mathsf{ok}(\varphi) \text{ ou } \mathsf{ok}(\psi) \\
\mathsf{ok}(\varphi\,U\,\psi) & = & \mathsf{ok}(\varphi) \text{ ou } \mathsf{ok}(\psi) \\
\mathsf{ok}(G\,\varphi) & = & \mathsf{ok}(\varphi) \\
\mathsf{ok}(F\,\varphi) & = & \mathbf{true} \\
\mathsf{ok}(\varphi\ \mathit{abort}\ b) & = & \mathsf{ok}(\varphi)
\end{array}$$

5.1.2.3 Sémantique pour les états d'une formule

Puisque le travail de Claessen et Mårtensson considère uniquement les opérateurs faibles, la notion de satisfaction exprimée dans la définition 3 est suffisante dans leur contexte.

Cependant, PSL définit quatre niveaux de satisfaction pour caractériser les quatre états possibles de toute propriété durant la simulation. Les deux niveaux de satisfaction *Holds* et *Holds strongly* requièrent "l'absence de tout état d'erreur" ainsi que "la rencontre de toutes les obligations futures". La différence entre les deux est que, pour la satisfaction simple *Holds*, la propriété pourrait ne pas être satisfaite pour toute extension possible de la trace. La fonction ok correspond uniquement au premier besoin, c'est-à-dire à l'absence d'erreurs.

Définition 5 Nous avons défini une nouvelle fonction, nommée met. $\mathsf{met}(\varphi)$ exprime le fait que toutes les obligations futures de φ ont été rencontrées :

$$
\begin{array}{rcl}
\mathsf{met}(F) & = & \textbf{true} \\
\mathsf{met}(T) & = & \textbf{true} \\
\mathsf{met}(b) & = & \textbf{false} \\
\mathsf{met}(\varphi \wedge \psi) & = & \mathsf{met}(\varphi) \text{ et } \mathsf{met}(\psi) \\
\mathsf{met}(\varphi \vee \psi) & = & (\mathsf{ok}(\varphi) \text{ et } \mathsf{met}(\varphi)) \text{ ou} \\
 & & (\mathsf{ok}(\psi) \text{ et } \mathsf{met}(\psi)) \text{ ou} \\
 & & (\mathsf{met}(\varphi) \text{ et } \mathsf{met}(\psi)) \\
\mathsf{met}(X\,\varphi) & = & \textbf{true} \\
\mathsf{met}(X!\,\varphi) & = & \textbf{false} \\
\mathsf{met}(\varphi\, W\, \psi) & = & \textbf{true} \\
\mathsf{met}(\varphi\, U\, \psi) & = & \mathsf{met}(\varphi) \text{ et } \mathsf{ok}(\psi) \text{ et } \mathsf{met}(\psi) \\
\mathsf{met}(G\,\varphi) & = & \textbf{false} \\
\mathsf{met}(F\,\varphi) & = & \mathsf{ok}(\varphi) \text{ et } \mathsf{met}(\varphi) \\
\mathsf{met}(\varphi\ \mathit{abort}\ b) & = & \mathsf{met}(\varphi)
\end{array}
$$

Si le résultat de la fonction ok est toujours identique pour les deux versions forte et faible d'un opérateur, au contraire la fonction met souligne la réelle différence. Par exemple, bien que $\mathsf{ok}(X\,\varphi) = \mathsf{ok}(X!\,\varphi) = \textbf{true}$, la fonction met appliquée aux deux versions fournit deux résultats opposés. L'explication intuitive est très simple et elle est liée à ce qui a été mentionné pour cet opérateur dans la section 2.2.3.3 (formule 2.4) : la version forte requiert que son opérande soit satisfait au prochain instant d'évaluation de la trace d'exécution, instant qui *doit exister* dans la trace ; la présence d'un prochain instant d'évaluation n'est *pas obligatoire* dans la version faible.

La combinaison des fonctions ok et met nous permet de caractériser entièrement le niveau de satisfaction *Holds*. Ces deux fonctions nous permettent également de définir les niveaux *Pending* et *Fails*. Le premier requiert toujours l'absence d'états d'erreur, sans avoir atteint toutes les obligations futures. Le deuxième indique la présence d'erreurs.

Définition 6 Les trois niveaux de satisfaction d'une formule φ sur un mot v, pour l'environnement initial ρ_0, sont définis comme suit :

$$
\begin{array}{rcl}
Holds_{v,\rho_0}(\varphi) & = & \mathsf{ok}(\varphi\langle v\rangle_{\rho_0}) \text{ et } \mathsf{met}(\varphi\langle v\rangle_{\rho_0}) \\
Pending_{v,\rho_0}(\varphi) & = & \mathsf{ok}(\varphi\langle v\rangle_{\rho_0}) \text{ et } \neg\mathsf{met}(\varphi\langle v\rangle_{\rho_0}) \\
Fails_{v,\rho_0}(\varphi) & = & \neg\mathsf{ok}(\varphi\langle v\rangle_{\rho_0})
\end{array}
$$

Avec les trois niveaux ci-dessus, nos moniteurs peuvent apporter plus de détail durant la simulation. De plus, bien que le niveau *Holds strongly* ne fasse pas partie de la définition précédente, il est souvent possible de le calculer pour une formule ou sous-formule. En effet, si l'on exclut l'opérateur *always*, qui ne peut jamais être satisfait fortement, la satisfaction forte est caractérisée comme expliqué dans la section 2.2.3.4 : il suffit de remplacer tout opérateur par sa version forte dans une formule et vérifier si la nouvelle formule obtenue est satisfaite simplement (fonction *Holds*). Nous verrons dans la section 5.2.3 que cette notion peut s'avérer très utile quand les propriétés comportent aussi des variables locales.

5.1.2.4 Adaptation au niveau transactionnel

Dans notre modèle de surveillance au niveau transactionnel, les instants d'observation sont liés aux actions de communication, et les lettres de la trace d'exécution peuvent

comporter aussi les paramètres et les valeurs de retour de ces communications. Pour prendre en compte cet aspect, nous avons étendu la notion de lettre pour qu'elle contienne également les valeurs de ces paramètres. Dans ce contexte, $\mathcal{ID}$ dénote l'ensemble de tous les identifiants des variables du design (il ne s'agit donc pas de variables auxiliaires) ainsi que ceux des paramètres et des valeurs de retour des fonctions (l'identifiant d'un paramètre a la syntaxe `composant.fonction.p#` introduite dans la section 4.2.3). Une lettre ℓ comporte alors une liste d'associations de ces identifiants avec leur valeurs, c'est-à-dire un élément de type $\mathcal{ID} \rightarrow (\mathcal{D} \cup \{\vartheta\})$. La valeur d'un identifiant peut être indéfinie dans une lettre donnée, si la communication correspondante ne se produit pas à l'instant considéré. Cette valeur indéfinie est notée ϑ. Par exemple, à l'instant où `fifo.write_CALL()` = **true**, le paramètre `fifo.write.p1` aura sans doute une valeur, mais pas `fifo.read.p0`. Les expressions telles que `fifo.write.p1` > 0 sont donc des expressions booléennes dont l'identifiant `fifo.write.p1` possède une valeur dans la lettre courante. La liste des associations des identifiants avec leurs valeurs dans la lettre ℓ est notée $\ell|_{\mathcal{ID}}$. Comme pour l'environnement, $\ell|_{\mathcal{ID}}(x)$ retourne la valeur de x dans ℓ, pour $x \in \mathcal{ID}$. Par la suite, afin de simplifier la notation, nous écrirons simplement $\ell(x)$. Enfin, il faut remarquer que, dans ce contexte, ρ devient également un élément de $\mathcal{DV} \rightarrow (\mathcal{D} \cup \{\vartheta\})$.

Définition 7 La règle 5.12 doit être modifiée pour considérer les lettres étendues et la notion de valeur indéfinie :

$$[\![b]\!]^m_\rho \xrightarrow{\ell} \begin{cases} T & \text{si } \ell \Vdash_{\rho'} b \\ F & \text{sinon} \end{cases} \tag{5.15}$$

où $\rho' = [\![m]\!]^\ell_\rho$ et

$$\ell \Vdash_\rho b \Leftrightarrow \begin{cases} \textbf{false} & \text{si } \exists x \in \mathcal{ID}_b \text{ tel que } \ell(x) = \vartheta \\ & \text{ou si } \exists y \in \mathcal{DV}_b \text{ tel que } \rho(y) = \vartheta \\ \ell \Vdash b' & \text{sinon} \end{cases} \tag{5.16}$$

Dans l'equation 5.16

- $\mathcal{ID}_b \subseteq \mathcal{ID}$ est l'ensemble des identifiants présents dans l'expression booléenne b,
- $\mathcal{DV}_b \subseteq \mathcal{DV}$ est l'ensemble des variables globales dans l'expression booléenne b,
- $b' = b_{[\mathcal{ID} \leftarrow \ell(\mathcal{ID})][\mathcal{DV} \leftarrow \rho(\mathcal{DV})]}$ est l'expression booléenne b dans laquelle chaque variable de $\mathcal{DV}$ est remplacée par sa valeur dans ρ, après avoir remplacé chaque identifiant de $\mathcal{ID}$ par sa valeur dans ℓ.

5.1.3 Mise en œuvre pour les variables globales

Le contenu d'une unité de vérification PSL peut comporter plusieurs éléments, nommés `Vunit_Item`, dans un ordre quelconque. La syntaxe définie dans [PSL05] se limite à indiquer la nature de ces éléments :

```
Vunit_Item ::=
    HDL_DECL  // Declaration de la couche modelisation
  | HDL_STMT  // Intruction de la couche modelisation
  | PSL_Declaration  // Declaration d'une priopriete ou sequence
  | PSL_Directive    // Directive de verification
                     // (par exemple, assertion d'une propriete)
```

Afin de permettre à notre outil de reconnaître aisément ces différentes parties, nous imposons la présentation de chaque *vunit* en trois sections à l'aide de commentaires contenant les trois mots-clés `HDL_DECLs`, `HDL_STMTs` et `PROPERTY`. Typiquement, une *vunit* sera donc organisée comme suit :

```
vunit maVunit {
  //  HDL_DECLs :
  ...declarations des variables globales...
  //  HDL_STMTs :
  ...instructions pour la modification des variables...
  // PROPERTY :
  ...assertions...
}
```

Partage de variables globales Les notions présentées jusqu'à présent considèrent toujours l'évolution de l'environnement et d'une seule formule PSL. Toutefois, l'environnement contient des variables globales qui pourraient être partagées par plusieurs propriétés à l'intérieur d'une *vunit*. Cela peut être expliqué à l'aide de l'exemple qui suit, volontairement caricatural. Considérons la *vunit* `v` qui déclare deux compteurs globaux `read_n` et `write_n` et qui comporte les deux assertions `A1` et `A2`. Dans cette *vunit*, les deux compteurs sont communs aux deux assertions.

```
vunit v {
  // HDL_DECLs :
  int read_n = 0;
  int write_n = 0;
  // HDL_STMTs :
  if( channel.read_CALL() ) { read_n++; }
  if( channel.write_CALL() ) { write_n++; }
  // PROPERTY :
  assert always (channel.read_CALL() -> (read_n+write_n > ...));  // A1
  assert always (channel.write_CALL() -> (write_n-read_n < ...)); // A2
```

Considérons maintenant la trace de simulation de la figure 5.1 : en suivant le modèle d'échantillonnage que nous avons défini dans le chapitre précédent, les ronds correspondent aux instants d'évaluation de `A1` et les triangles correspondent aux instants d'évaluation de `A2`.

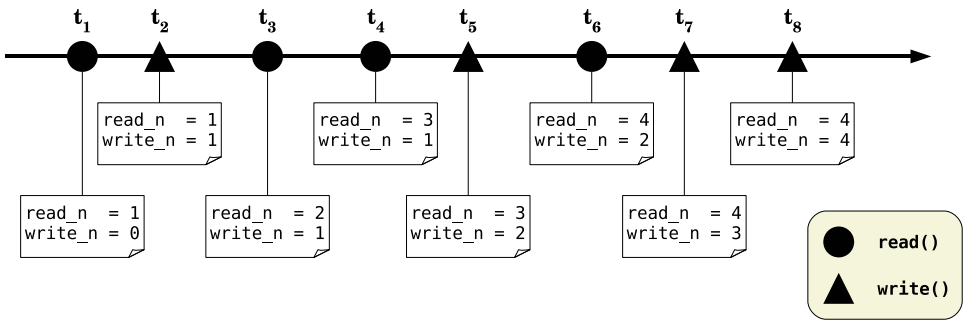

FIGURE 5.1 – Exemple de trace de simulation pour $A1$ et $A2$

Selon la sémantique que nous avons proposée dans la section précédente, le bloc d'instructions de la couche modélisation relative une propriété P est évalué pour chaque lettre de la trace associée à P. Dans le cas de la *vunit* `v` ci-dessus, la couche modélisation est

commune à deux propriétés. Par conséquent, les instructions qui suivent `HDL_STMTs` seront évaluées une seule fois pour chaque instant de la trace qui comporte toutes les occurrences de `channel.read()` et `channel.write()` (i.e. tous les instants de t_1 à t_8 sur la trace de la figure 5.1). Plus généralement, dans le cas de plusieurs propriétés $\varphi_1, \varphi_2, ..., \varphi_n$ avec "listes de sensibilité" (i.e. communications observées) respectives $S_1, S_2, ..., S_n$ dans une seule *vunit* V, les instructions de couche modélisation de V seront exécutées sur la trace obtenue en considérant une seule "liste de sensibilité" $S = S_1 \cup S_2 \cup ... \cup S_n$. Par contre, chaque propriété φ_k sera toujours évaluée en suivant sa propre liste de sensibilité S_k. Ce contexte où une variable globale est partagée entre plusieurs assertions est similaire à celui de l'exécution, pour un langage de programmation, de plusieurs fonctions distinctes utilisant une même variable globale.

Mécanisme d'évaluation La totalité des variables globales déclarées dans une *vunit* `v` est regroupée dans une classe C++ nommée `v_data`. Cette classe est générée automatiquement et elle comporte un constructeur qui assure leurs initialisations, indiquées dans la section `HDL_DECLs`. Un *wrapper*, module englobant les moniteurs, est généré et associé à `v`. Lors de la création des instances de tous les observateurs, l'instance de la classe `v_data` est créée par le *wrapper*. Cette instance est associée à tous les moniteurs inclus et partagée, comme montré dans la partie gauche de la figure 5.2.

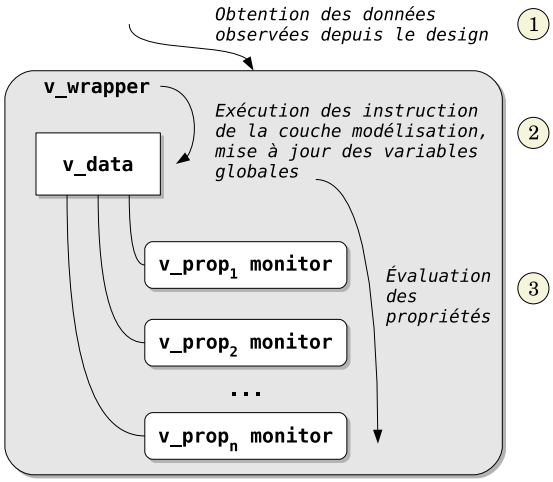

FIGURE 5.2 – Structure du *wrapper* et évaluation des moniteurs

Quand la méthode `Update` du *wrapper* est appelée (cf. section 4.2.2), après identification du composant du design à l'origine de la notification, et après obtention des données relatives à la communication observée (par exemple, l'adresse d'une écriture ou la donnée écrite), le bloc d'instructions de la couche modélisation est exécuté et les variables globales sont mises à jour. Ensuite les moniteurs sont évalués.

À l'heure actuelle, afin de simplifier l'analyse syntaxique et l'étape de génération des moniteurs, l'outil ISIS n'autorise qu'une seule propriété dans chaque *vunit*. Par conséquent, le modèle implémenté est en partie simplifié : l'étape de mise-à-jour des variables est effectuée directement au début de la fonction d'évaluation du moniteur (le seul de la *vunit*) et non pas dans celle du *wrapper*. Cependant, le principe décrit ci-dessus reste identique. Le code de la section `HDL_STMTs` est donc retranscrit au début de la fonction du moniteur plutôt que dans celle du *wrapper*.

Exemple Pour illustrer brièvement les principes décrits dans cette section, nous allons considérer une implémentation SystemC du protocole de communication par canal défectueux décrit dans [CM88]. Dans ce modèle, un module émetteur utilise un canal potentiellement défectueux pour envoyer des messages à un module récepteur. Le canal est passible de perdre ou de dupliquer les messages. Le récepteur acquitte les messages reçus via un signal `ack`, chaque acquittement étant le numéro du dernier message reçu. En fonction de la valeur transmise par `ack`, un même message peut être ré-émis ou non par l'émetteur.

Nous avons utilisé la couche modélisation de PSL pour exprimer la propriété suivante, mentionnée dans [CM88] : il est toujours vrai que $ack \leq nb_sent \leq ack + 1$, où nb_sent est le numéro du dernier message envoyé par l'émetteur. Outre ce qui a été expliqué pour la couche modélisation, la syntaxe de la propriété suit les principes présentés dans le chapitre précédent :

```
vunit protocol_prop1 {
    // HDL_DECLs :
    unsigned int nb_sent = 0;
    // HDL_STMTs :
    if( channel.write_CALL() )
      nb_sent = (channel.write.p1).number;
    // PROPERTY :
    assert
    (always ( ack.read_CALL() ->
              (ack.read.p0 <= nb_sent && nb_sent <= (ack.read.p0+1)) ))
    @ ( channel.write_CALL() || ack.read_CALL() );
}
```

Notons que le moniteur doit être activé sur les écritures dans le canal (pour les mises à jour de `nb_sent`) et sur les lectures dans `ack` (pour le test effectif de l'assertion).

La classe `protocol_prop1_data` générée par ISIS inclut la variable auxiliaire `nb_sent`, initialisée avec la valeur 0 :

```
class  protocol_prop1_data {
  public:
    ...
    // Donnees observees (valeurs assignees par le wrapper) :
    bool channel__write_CALL;
    message channel__write__p1;
    bool ack__read_CALL;
    int ack__read__p0;
    // Declarations auxiliaires de la couche modelisation :
    unsigned int nb_sent;
    ...
    // Constructeur avec initialisations de la partie declarative :
    protocol_prop1_data() : nb_sent(0) { }
};
```

Chaque fois qu'il y a une écriture dans le canal ou une lecture dans le signal d'acquittement, les instructions qui suivent `HDL_STMTs` sont exécutées. La fonction d'évaluation du moniteur principal exécute ces instructions avant d'évaluer la propriété. Si une écriture dans le canal a lieu, le numéro du message écrit est alors mémorisé dans `nb_sent` :

```
void main_mnt::update() {
  // Execution de la couche modelisation :
```

```
  if( channel__write_CALL )
    nb_sent = (channel__write__p1).number;
  // Evaluation du moniteur elementaire "always" :
  data->always_1_inst->update();
  // Obtention des sorties finales :
  data->Valid = data->checking_always_1;
  data->Checking = data->checking_always_1;
  // Affichages...
  ...
  if( ! data->Valid ) {
    data->nb_fails++;
    // Affichages...
  }
}
```

5.2 Propriétés réentrantes et variables locales

Comme il a été souligné au début du chapitre, la présence de variables globales ne permet pas de couvrir toutes les relations entre les transactions. Quand les communications surveillées par une assertion s'effectuent en pipeline, ou dans le cas plus général où plusieurs transactions se superposent, une seule variable globale n'est pas suffisante pour mémoriser les données de toutes les communications : chaque nouvelle donnée nécessite d'être mémorisée dans une variable indépendante.

Considérons l'exemple du *Packet Switch* fourni avec la bibliothèque SystemC. Dans ce design, montré sur la figure 5.3, quatre émetteurs et quatre récepteurs sont connectés au *switch*. Ce dernier utilise un anneau de registres à décalage pour acheminer les paquets vers les ports de sortie appropriés. Les rotations entre ces registres s'effectuent sur les fronts montants d'une horloge `switch_ctrl`. Une FIFO de taille limitée, associée à chaque port d'entrée et de sortie, permet de stocker les paquets en attente. Tout paquet comporte un identifiant *id*, une donnée *data* et une liste de drapeaux booléens pour marquer ses destinataires.

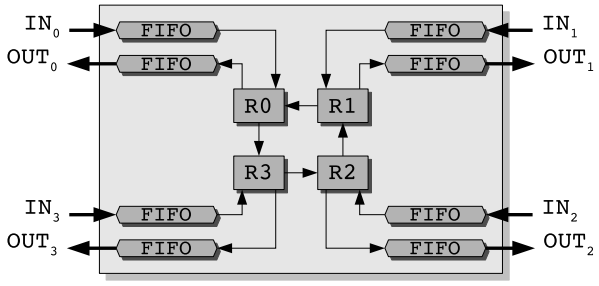

FIGURE 5.3 – *Packet Switch* de la bibliothèque SystemC

Une propriété fonctionnelle cruciale est la suivante : "tout paquet en entrée du *switch* par un port X et à destination d'un port Y sera inévitablement délivré sur le port Y". Pour pouvoir exprimer cette propriété, il faut pouvoir mémoriser chaque paquet en entrée du *switch* par $\texttt{IN}_X$ et s'assurer qu'il sortira par $\texttt{OUT}_Y$. L'emploi d'un variable globale n'est clairement pas suffisant, parce que plusieurs paquets distincts, émis par X et à destination de Y, peuvent être présents dans le *switch* en même temps. Par exemple, la propriété `switch_bad_prop` suivante utilise une seule variable globale et ne convient certainement

pas : chaque nouveau paquet en entrée par le port 3 et à destination de 0 est mémorisé dans la même variable `packet`, en écrasant la valeur précédente.

```
vunit switch_bad_prop {
// HDL_DECLs :
pkt packet;
// HDL_STMTs :
// "dest0" est le drapeau booleen pour le premier destinataire
if( in3.write_CALL() && (in3.write.p1).dest0 )
  packet = in3.write.p1;
// PROPERTY :
assert always(in3.write_CALL() && (in3.write.p1).dest0
              -> (eventually! (out0.write_CALL() &&
                               (out0.write.p1).id == packet.id &&
                               (out0.write.p1).data == packet.data)));
}
```

Nous avons donc étendu la sémantique proposée dans la section précédente pour pouvoir traiter des propriétés *réentrantes*[5], avec variables locales. Chaque fois qu'une variable locale est introduite dans une formule, cette formule est évaluée indépendamment, avec la nouvelle variable et sa nouvelle valeur.

5.2.1 Réentrance avec simple mémorisation

Dans [PF10], nous avons proposé une extension de la sémantique de la section 5.1. Cette extension comporte un nouvel opérateur qui permet l'introduction de nouvelles variables dans toute formule φ. Lorsque la nouvelle variable est introduite dans φ, une évaluation indépendante de φ est démarrée. Dans un premier temps, comme nous le verrons dans la sémantique qui suit, les variables ainsi introduites ne sont pas appelées à être modifiés et sont assimilés à des paramètres effectifs de fonctions.

L'introduction d'une nouvelle construction syntaxique s'avère nécessaire, la notion de variable locale étant complètement absente de [PSL05][6]. Nous avons ajouté l'opérateur new, avec la structure new($garde$? $type\ variable = expression$)($\varphi$), où *garde* est une condition booléenne de garde pour l'affectation, *type* est le nom du type de la variable dans la syntaxe du langage de modélisation sous-jacent (ici SystemC/C++), *variable* est la variable nouvellement introduite dans la (sous)formule φ et *expression* est l'expression en langage de modélisation sous-jacent dont la valeur sera affectée à *variable*. La présence d'une condition de garde est justifiée par le fait que l'introduction d'une variable locale pourrait être considérée comme non pertinente à certains instants de la trace. Par exemple, l'affectation d'une variable avec la valeur du paramètre d'une opération d'écriture n'a pas de sens aux instants où l'écriture n'a pas lieu. Il est alors possible d'utiliser le prédicat associé à l'occurrence de l'opération d'écriture (`write_CALL()`) comme condition de garde. Le *type* de la variable est volontairement omis dans les règles sémantiques. Il n'est pas nécessaire, puisque le domaine $\mathcal{D}$ des variables regroupe toutes les valeurs possibles, sans distinction de type. L'ensemble $\mathcal{FL}$ des formules FL inclut maintenant le nouvel opérateur new.

[5] Comme indiqué précédemment, cette notion est proche du concept de fonction réentrante en informatique, selon lequel la fonction est utilisable simultanément par plusieurs tâches, sans risque de corruption des données [Jha05].

[6] Les travaux de cette thèse sont entièrement basés sur le standard PSL de 2005. Une nouvelle révision de la norme est parue au courant de l'année 2010 et sera commentée par la suite.

Définition 8 Soient b une expression booléenne de garde, x l'identifiant de la nouvelle variable locale à φ et e l'expression à affecter à la variable, la sémantique de l'opérateur new est définie par la règle de dérivation 5.17 et par les deux règles de réduction 5.18 et 5.19 :

$$\frac{[\![\mathsf{new}(b\ ?\ x = e)(\varphi)]\!]^m_\rho \rightarrow [\![(\lambda x.\varphi)(a)]\!]^m_\rho \qquad [\![(\lambda x.\varphi)(a)]\!]^m_\rho \xrightarrow{\ell} [\![\varphi']\!]^m_{\rho'}}{[\![\mathsf{new}(b\ ?\ x = e)(\varphi)]\!]^m_\rho \xrightarrow{\ell} [\![\varphi']\!]^m_{\rho'}} \tag{5.17}$$

$$\frac{[\![b]\!]^m_\rho \xrightarrow{\ell} T \qquad [\![e]\!]^\ell_\rho = e'}{[\![\mathsf{new}(b\ ?\ x = e)(\varphi)]\!]^m_\rho \rightarrow [\![(\lambda x.\varphi)(e')]\!]^m_\rho} \tag{5.18}$$

$$\frac{[\![b]\!]^m_\rho \xrightarrow{\ell} F}{[\![\mathsf{new}(b\ ?\ x = e)(\varphi)]\!]^m_\rho \rightarrow [\![(\lambda x.\varphi)(\vartheta)]\!]^m_\rho} \tag{5.19}$$

où ϑ est la valeur indéfinie et $[\![e]\!]^\ell_\rho$ est l'interprétation de l'expression e dans l'environnement courant ρ et dans la lettre courante ℓ. Soit $\mathcal{E} \subset \mathcal{M}$ l'ensemble des expressions C++, $[\![e]\!]^\ell_\rho : \mathcal{E} \times \Sigma \times \mathcal{P} \rightarrow \mathcal{D} \cup \{\vartheta\}$. Une expression peut avoir la valeur indéfinie ϑ si elle comporte la référence à un paramètre d'une communication qui n'a pas lieu dans ℓ, comme expliqué dans la section 5.1.2.4. Les expressions booléennes, en particulier celles dans $[\![(\lambda x.\varphi)(\vartheta)]\!]^m_\rho$, peuvent être évaluées en suivant une interprétation *stricte* ou *paresseuse* par la dérivation $\xrightarrow{\ell}$. Dans notre sémantique, nous considérons que toute expression booléenne qui comporte la valeur indéfinie ϑ évalue directement à **false** (interprétation stricte).

La règle sémantique 5.17 utilise une λ-abstraction avec β-réduction [Mit96] pour représenter le fait que φ possède maintenant une variable x dont la valeur est celle de e'. Ceci peut être vu comme un passage de paramètre et constitue l'aspect central de la réentrance. Techniquement, une nouvelle instance d'un moniteurs qui surveille φ est créée lorsque $\mathsf{new}(b\ ?\ x = e)(\varphi)$ est évaluée. Ce concept est expliqué plus en détail dans la section 5.2.3. La règle 5.17 se lit donc comme suit : si $\mathsf{new}(b\ ?\ x = e)(\varphi)$ se réduit en φ, où x reçoit la valeur a, et si φ avec la valeur a pour x dérive en φ' par ℓ, alors $\mathsf{new}(b\ ?\ x = e)(\varphi)$ dérive en φ' par ℓ. L'environnement $\rho' = [\![m]\!]^\ell_\rho$ évolue toujours comme décrit dans la section 5.1.

Il est également possible d'introduire plusieurs variables $x_1, x_2, ..., x_n$, ayant les types $t_1, t_2, ..., t_n$, avec un seul new. La syntaxe est alors généralisée par

$$\mathsf{new}(b\ ?\ t_1\ x_1 = e_1, t_2\ x_2 = e_2, ..., t_n\ x_n = e_n)(\varphi)$$

La seule différence dans la règle sémantique 5.17 est l'exercice de plusieurs λ-abstractions et β-réductions : $((((\lambda x_1.\lambda x_2. \ldots .\lambda x_n.\varphi)(a_1))(a_2))...)(a_n)$.

Grâce à l'extension décrite par la définition 8, il est maintenant possible d'exprimer la propriété sur *Packet Switch* évoquée auparavant : "tout paquet en entrée par le port 3 et à destination du port 0 sera inévitablement délivré à la bonne destination" :

```
vunit switch_route_prop1 {
// HDL_DECLs :
// HDL_STMTs :
// PROPERTY :
assert always(in3.write_CALL() && (in3.write.p1).dest0 ->
              new(true ? pkt packet = in3.write.p1)
                 (eventually! (out0.write_CALL() &&
```

```
                                    (out0.write.p1).id == packet.id &&
                                    (out0.write.p1).data == packet.data)));
}
```

5.2.2 Réentrance avec mise à jour des variables locales

La sémantique proposée dans [PF10] décrit l'introduction d'une variable locale qui reçoit et conserve une valeur dans l'assertion. Cet aspect permet l'évaluation réentrante de la propriété mais ne prévoit pas de modification ultérieure de la variable locale.

Considérons une fois de plus le *Packet Switch* de la bibliothèque SystemC, montré sur la figure 5.3. Une variante intéressante de la propriété `switch_route_prop1` de la section précédente est la bonne sortie du paquet *avant un temps limite*, par exemple : "tout paquet en entrée par le port 3 et à destination du port 0 sera délivré sur la sortie 0 avant 20 fronts montants de l'horloge de contrôle du *switch*"[7]. Puisque la propriété surveille à la fois les écritures dans les ports 3 et 0 et les changements de valeur de l'horloge du *switch*, elle ne peut toujours pas employer l'opérateur **`next`**`[20]` : les envois de messages sur le port 3 ne sont pas synchronisés sur cette horloge. Dans ce cas, il peut être utile d'ajouter une deuxième variable locale dans `switch_route_prop1`, plus précisément un compteur du nombre de fronts montants de l'horloge du *switch*. Ensuite, au lieu d'exprimer le fait que le message sera inévitablement délivré, c'est-à-dire **`eventually`**`! deliver`, il faudrait exprimer sa sortie avant écoulement du temps limite, c'est-à-dire `deliver` **`before`**`! count==20`. Dans la sémantique de la section précédente, il n'est pas possible d'utiliser un compteur local à une sous-formule.

Nous avons donc étendu ultérieurement la sémantique afin de permettre l'expression de la modification des variables locales, suivant un principe proche de celui adopté pour les variables globales. Dans la nouvelle extension, les "effets de bord" sur les variables locales à une formule φ sont, eux aussi, locaux à φ et ils sont calculés par un nouveau bloc d'instructions. Il n'est plus nécessaire de "garder" les initialisations des variables par une condition booléenne, la présence des instructions autorise l'utilisation de toute affectation conditionnelle si nécessaire.

La syntaxe de l'opérateur new est étendue par l'ajout d'un bloc d'instructions local à la sous-formule φ :

$$\mathsf{new}(t_1\ x_1 = e_1, t_2\ x_2 = e_2, ..., t_n\ x_n = e_n;\ instructions)(\varphi)$$

Comme pour les variables globales, la modification des variables locales s'appuie sur une notion d'environnement, ici local à une sous-formule, noté σ. En réalité, comme nous le verrons grâce à la sémantique qui suit, il n'existe plus de vraie différence entre l'environnement global et les environnements locaux. Le premier représente simplement l'environnement le plus externe : nous choisissons de lui conserver l'identificateur ρ. Les environnements locaux, qui peuvent "s'empiler", seront notés $\sigma_1, \sigma_2, ..., \sigma_n$. Nous désignons par $i \in \mathcal{M}$ tout bloc d'instructions local à une sous-formule. L'ensemble des variables déclarées locales à une sous-formule est noté $\mathcal{LV}$ et défini comme suit :

$$\mathcal{LV}[\mathsf{new}(t_1\ x_1 = e_1, ..., t_n\ x_n = e_n;\ i)(\varphi)] = \{x_1, ..., x_n\} \tag{5.20}$$

[7] Nous rappelons que, bien que notre solution s'applique principalement au niveau transactionnel sans horloge, elle nous permet également de surveiller des designs SystemC synchronisés sur une ou plusieurs horloges.

Plusieurs sous-formules $\varphi_1, \varphi_2, ..., \varphi_n$ peuvent donc avoir des environnements locaux $\sigma_1, \sigma_2, ..., \sigma_n$ distincts. Chaque environnement σ_k est défini comme étant un élément de $\mathcal{LV}_k \to \mathcal{D} \cup \{\vartheta\}$, pour $1 \leq k \leq n$. Chaque formule φ_k peut avoir un bloc d'instruction local i_k. On note alors par $[\![\varphi_k]\!]^{i_k}_{\sigma_k}$ l'interprétation de φ_k où les variables de $\mathcal{LV}_k$ sont remplacées par les valeurs qui leur sont associées par l'environnement local σ_k, compte tenu du bloc d'instructions local i_k. Enfin, plusieurs environnement locaux peuvent "s'empiler" les uns sur les autres, en donnant lieu à une série d'interprétations imbriquées. Nous noterons alors :

$$[\![\, [\![\dots [\![\varphi]\!]^{i_n}_{\sigma_n} \dots]\!]^{i_2}_{\sigma_2} \,]\!]^{i_1}_{\sigma_1} = [\![\varphi]\!]_{\sum_1^n \sigma}$$

Définition 9 La nouvelle extension de la sémantique opérationnelle n'est donc plus définie pour une seule interprétation dans l'environnement global, mais pour une série d'interprétations locales $[\![\,]\!]_{\sum_1^n \sigma}$ dans l'environnement global. Les règles 5.21 à 5.24 remplacent les règles données au début de la section 5.1.2.2.

$$\frac{[\![\, [\![\varphi]\!]_{\sum_1^n \sigma} \,]\!]^m_\rho \xrightarrow{\ell} [\![\, [\![\varphi']\!]_{\sum_1^n \sigma'} \,]\!]^m_{\rho'} \qquad [\![\, [\![\psi]\!]_{\sum_1^n \sigma} \,]\!]^m_\rho \xrightarrow{\ell} [\![\, [\![\psi']\!]_{\sum_1^n \sigma'} \,]\!]^m_{\rho'}}{[\![\, [\![\varphi \wedge \psi]\!]_{\sum_1^n \sigma} \,]\!]^m_\rho \xrightarrow{\ell} [\![\, [\![\varphi' \wedge \psi']\!]_{\sum_1^n \sigma'} \,]\!]^m_{\rho'}} \tag{5.21}$$

$$\frac{[\![\, [\![\varphi]\!]_{\sum_1^n \sigma} \,]\!]^m_\rho \xrightarrow{\ell} [\![\, [\![\varphi']\!]_{\sum_1^n \sigma'} \,]\!]^m_{\rho'} \qquad [\![\, [\![\psi]\!]_{\sum_1^n \sigma} \,]\!]^m_\rho \xrightarrow{\ell} [\![\, [\![\psi']\!]_{\sum_1^n \sigma'} \,]\!]^m_{\rho'}}{[\![\, [\![\varphi \vee \psi]\!]_{\sum_1^n \sigma} \,]\!]^m_\rho \xrightarrow{\ell} [\![\, [\![\varphi' \vee \psi']\!]_{\sum_1^n \sigma'} \,]\!]^m_{\rho'}} \tag{5.22}$$

$$[\![\, [\![X!\varphi]\!]_{\sum_1^n \sigma} \,]\!]^m_\rho \xrightarrow{\ell} [\![\, [\![\varphi]\!]_{\sum_1^n \sigma'} \,]\!]^m_{\rho'} \tag{5.23}$$

$$\frac{[\![\, [\![\varphi]\!]_{\sum_1^n \sigma} \,]\!]^m_\rho \xrightarrow{\ell} [\![\, [\![\varphi']\!]_{\sum_1^n \sigma'} \,]\!]^m_{\rho'} \qquad [\![\, [\![\psi]\!]_{\sum_1^n \sigma} \,]\!]^m_\rho \xrightarrow{\ell} [\![\, [\![\psi']\!]_{\sum_1^n \sigma'} \,]\!]^m_{\rho'}}{[\![\, [\![\varphi\, U\, \psi]\!]_{\sum_1^n \sigma} \,]\!]^m_\rho \xrightarrow{\ell} [\![\, [\![\psi' \vee (\varphi' \wedge (\varphi\, U\, \psi))]\!]_{\sum_1^n \sigma'} \,]\!]^m_{\rho'}} \tag{5.24}$$

Dans toutes ces règles
- $\rho' = [\![m]\!]^\ell_\rho$
- $\sigma'_1 = [\![i_1]\!]^\ell_{\rho' \oslash \sigma_1}$
- $\forall k, 1 < k \leq n,\ \ \sigma'_k = [\![i_k]\!]^\ell_{\sigma'_{k-1} \oslash \sigma_k}$

Nous utilisons un opérateur noté $\oslash$ pour traduire la portée et l'éventuel masquage des variables locales : soient deux environnements en_1 et en_2,

$$(en_1 \oslash en_2)(x) = \begin{cases} en_2(x) & \text{si } x \text{ est liée dans (la sous-formule associé à) } en_2 \\ en_1(x) & \text{sinon} \end{cases} \tag{5.25}$$

Définition 10 La règle 5.15 doit aussi être reprise dans ce cas, afin qu'elle considère une série d'interprétations imbriquées $[\![\,]\!]_{\sum_1^n \sigma}$:

$$[\![\, [\![b]\!]_{\sum_1^n \sigma} \,]\!]^m_\rho \xrightarrow{\ell} \begin{cases} T & \text{si } \ell \Vdash_{\rho', \sum_1^n \sigma'} b \\ F & \text{sinon} \end{cases} \tag{5.26}$$

où les environnements ρ' et $\sigma'_{1 \leq k \leq n}$ sont calculés comme indiqué après la règle 5.24 de la définition 9, et où

$$\ell \Vdash_{\rho, \sum_1^n \sigma} b \Leftrightarrow \begin{cases} \textbf{false} & \text{si } \exists x \in \mathcal{ID}_b \text{ tel que } \ell(x) = \vartheta \\ & \text{ou si } ND(\rho, \sum_1^n \sigma) \\ \ell \Vdash b' & \text{sinon} \end{cases} \tag{5.27}$$

Dans l'equation 5.27

- $\mathcal{ID}_b \subseteq \mathcal{ID}$ est l'ensemble des identifiants présents dans l'expression booléenne b,
- $\mathcal{DV}_b \subseteq \mathcal{DV}$ est l'ensemble des variables globales dans l'expression booléenne b,
- $(\mathcal{LV}_b)_k \subseteq \mathcal{LV}_k$ est l'ensemble des variables locales au niveau k, $1 \leq k \leq n$, présentes dans l'expression booléenne b,
- $ND(\rho, \sum_1^n \sigma)$ dénote la condition suivante :
  $\exists$ un identifiant pour $y \in \mathcal{DV}_b$ tel que
  $\rho(y) = \vartheta$ et $\forall j, 1 \leq j \leq n,\ (y \notin (\mathcal{LV}_b)_j$ ou $\sigma_j(y) = \vartheta)$ ou
  $\exists$ un indentifiant pour $z \in (\mathcal{LV}_b)_k$ tel que
  $\sigma_k(z) = \vartheta$ et $\forall j, k < j \leq n,\ (z \notin (\mathcal{LV}_b)_j$ ou $\sigma_j(z) = \vartheta)$
  ce qui signifie que, pour une variable globale ayant la valeur indéfinie dans ρ, ou pour une variable locale ayant la valeur indéfinie dans σ_k, aucune variable avec le même nom et une valeur définie n'a été trouvée dans les environnements plus internes.
- $b' = b_{[\mathcal{ID} \leftarrow \ell(\mathcal{ID})][\mathcal{LV}_n \leftarrow \sigma_n(\mathcal{LV}_n)] \ldots [\mathcal{LV}_1 \leftarrow \sigma_1(\mathcal{LV}_1)][\mathcal{DV} \leftarrow \rho(\mathcal{DV})]}$ est l'expression booléenne b dans laquelle
  1. chaque identifiant de $\mathcal{ID}$ est remplacé par sa valeur dans ℓ
  2. ensuite, chaque variable locale de $\mathcal{LV}_k$ est remplacée par sa valeur dans σ_k (avec ordre décroissant sur k, c'est-à-dire du plus interne vers le plus externe)
  3. ensuite, chaque variable de $\mathcal{DV}$ est remplacée par sa valeur dans ρ

Définition 11 Enfin, la sémantique de l'opérateur `new` pour l'introduction d'une nouvelle variable x locale à φ dans une série d'interprétations imbriquées $[\![\,]\!]_{\sum_1^n \sigma}$ est définie par la règle 5.28 suivante :

$$\frac{[\![\,[\![\,[\![\varphi]\!]^{i_{n+1}}_{\sigma_{n+1}}\,]\!]_{\sum_1^n \sigma}\,]\!]^m_\rho \xrightarrow{\ell} [\![\,[\![\,[\![\varphi']\!]^{i_{n+1}}_{\sigma'_{n+1}}\,]\!]_{\sum_1^n \sigma'}\,]\!]^m_{\rho'}}{[\![\,[\![\mathsf{new}(x = e;\ i_{n+1})(\varphi)]\!]_{\sum_1^n \sigma}\,]\!]^m_\rho \xrightarrow{\ell} [\![\,[\![\,[\![\varphi']\!]^{i_{n+1}}_{\sigma'_{n+1}}\,]\!]_{\sum_1^n \sigma'}\,]\!]^m_{\rho'}} \tag{5.28}$$

Les environnements ρ' et $\sigma'_{1 \leq k \leq n}$ sont toujours calculés comme indiqué après la règle 5.24 de la définition 9 et

$$\sigma'_{n+1} = [\![i_{n+1}]\!]^\ell_{\sigma'_n \oslash \sigma_{n+1}} \quad \text{où } \sigma_{n+1} = [x \mapsto e'] \quad \text{avec } e' = [\![\,[\![\,e\,]\!]_{\sum_1^n \sigma}\,]\!]^\ell_\rho$$

Grâce à cette dernière extension de la sémantique, la variante avec compteur local de la propriété `switch_route_prop1` peut être exprimée comme l'indique la propriété `switch_route_prop2` ci-dessous[8]. Nous rappelons son énoncé : "tout paquet en entrée par le port 3 et à destination du port 0 sera délivré sur la bonne sortie avant 20 fronts montants de l'horloge de contrôle du *switch*".

```
vunit switch_route_prop2 {
// HDL_DECLs :
// HDL_STMTs :
// PROPERTY :
assert (always(in3.write_CALL() && (in3.write.p1).dest0 ->
               new(pkt packet = in3.write.p1,
                   int count = 0;
                   if(switch_ctrl){ count++ })
```

[8] Dans [PF10] nous avons pu exprimer une propriété comparable à `switch_route_prop2`, en utilisant un compteur global et un paramètre local `max` ayant la valeur du compteur+20.

```
                    ( (out0.write_CALL() &&
                       (out0.write.p1).id == packet.id &&
                       (out0.write.p1).data == packet.data)
                      before!
                      (count == 20) )))
        @ (in3.write_CALL() || out0.write_CALL() || switch_ctrl);
}
```

5.2.3 Mise en œuvre pour les variables locales

5.2.3.1 Principes d'implémentation

Création d'instances Durant la vérification dynamique d'une propriété avec variables locales, chaque évaluation de l'opérateur **new** démarre une nouvelle évaluation indépendante de la formule sous la portée de cet opérateur. La formule comporte une instance fraîche des variables locales concernées. La nature modulaire de nos moniteurs offre un avantage non négligeable pour accomplir cette tâche. Comme expliqué dans la section 4.1 (cf. figure 4.2 à la page 64 par exemple), la structure des moniteurs suit celle de l'arbre syntaxique de la propriété. L'opérateur **new** est considéré comme un opérateur $\mathcal{FL}$, il peut donc être utilisé suivant les mêmes règles syntaxiques que les autres opérateurs, ainsi que les même restrictions liées au sous-ensemble simple de PSL. Par exemple, il pourrait apparaître directement sous la portée d'un **always**, comme montré sur la partie gauche de la figure 5.4. Dans ce cas, une nouvelle instance de la sous-formule φ du **new** sera créée et amorcée à chaque instant de la trace. Si la trace est de longueur n, il y aura n instances potentiellement concurrentes de φ (partie droite de la figure 5.4).

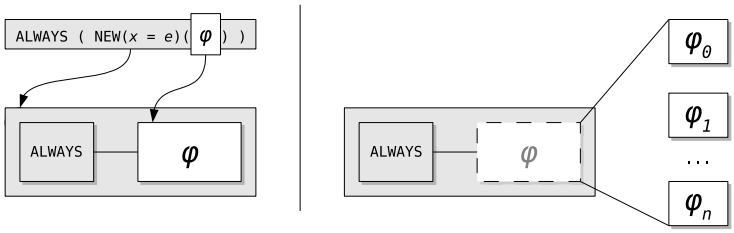

FIGURE 5.4 – Opérateur *new* et instances d'une sous-propriété

Techniquement, une classe C++ relative au moniteur pour φ est générée, comme s'il s'agissait d'une propriété à vérifier seule. Comme expliqué dans la section 4.1.2.2, chaque moniteur élémentaire est responsable de l'activation de l'évaluation de la partie qui le suit ; cette activation se fait en utilisant la sortie *Checking* comme entrée *Start* du moniteur suivant. Dans ce cas, chaque fois que le bloc relatif à φ doit être activé, c'est-à-dire chaque fois qu'il reçoit un *Start*, une nouvelle instance de la classe qui le représente est alors créée, et son évaluation démarrée immédiatement après.

En réalité, pour des raisons d'optimisation et de choix de conception[9], les instances de φ ne sont pas créées dynamiquement, en cours de simulation. Une collection de ces instances est créée avant de démarrer la simulation. Toutes les instances de φ dans la collection sont à l'arrêt, elle ne vérifient pas la formule. Chaque fois que le moniteur

[9] Nous avons conçu deux versions de la bibliothèque C++ pour le modèle de surveillance de la section 4.2.2 : dans l'une des deux versions, la classe *Moniteur* est un composant SystemC, par conséquent dans cette version un moniteur ne peut être créée qu'avant le début de la simulation (cf. section 2.1.2.3).

relatif à l'opérateur **new** reçoit un *Start*, une de ces instances est sortie de la collection et initialisée, puis son évaluation est démarrée.

Un moniteur élémentaire `mnt_new_phi`, ajouté à la bibliothèque SystemC des moniteurs primitifs, est associé au nouvel opérateur **new**. Outre la fonction `update`, commune à tous les moniteurs élémentaires, il comporte trois nouvelles méthodes :

- `instantiate`, appelée lorsqu'une nouvelle instance de φ doit être démarrée (i.e. réception d'un *Start*), utilisée pour l'initialisation des données locales et pour celle de l'état du moniteur ;
- `sync`, appelée au début de la fonction `update` du **new**, utilisée pour mettre à jour les données locales à φ à partir des données transmises par le *wrapper* et par exécution du bloc d'instructions local (si présent) ;
- `destroy`, appelée lorsque l'instance de moniteur a terminé la vérification de φ, utilisée pour repositionner le moniteur dans la collection de moniteurs à l'arrêt.

Destruction d'instances L'identification du moment où un moniteur a terminé la vérification d'une formule est très important. Cela permet d'arrêter son fonctionnement (fonction `destroy`), en économisant ainsi de la mémoire et du temps CPU précieux. La satisfaction forte d'une formule, i.e. son niveau *Holds strongly*, est définitive, puisque même en prolongeant indéfiniment la trace de simulation le verdict resterait immuable. La satisfaction d'un opérateur fort, i.e. le résultat de la fonction *Holds* de la définition 6, équivaut à sa satisfaction forte *Holds strongly*. Comme rappelé dans la section 5.1.2.3, même pour une formule avec opérateurs faibles, le niveau *Holds strongly* peut être facilement identifié, car il équivaut au niveau *Holds* quand tous les opérateurs sont considérés dans leur version forte. Par conséquent, nous considérons toujours le résultat de la fonction met (ce qui équivaut à la sortie *Pending* de chaque opérateur) même pour les opérateurs faibles, et nous sommes toujours en mesure de savoir si une formule est satisfaite fortement ou pas. Quand l'une des instances de φ actives rentre dans l'état *Holds strongly*, elle est immédiatement arrêtée (fonction `destroy`). Le niveau *Fails* est aussi définitif et il entraîne la destruction de l'instance, toutefois il est possible de ne pas provoquer l'arrêt du moniteur dans ce cas, afin de permettre l'identification d'autres violations futures.

Évaluation Dans les moniteurs pour les propriétés sans opérateur **new**, les expressions booléennes sont évaluées en utilisant directement les données transmises par le *wrapper* du moniteur. Par exemple, pour évaluer l'expression booléenne `c.write.p1 > 0`, la donnée `c.write.p1` est transmise par le composant observé au *wrapper*, ensuite par ce dernier au moniteur via la structure *maVunit*`_data` décrite dans la section 5.1.3. Puisque l'opérateur **new** entraîne la création de plusieurs instances s'évaluant de façon concurrente (en termes de sémantique de simulation), chaque instance doit obtenir les valeurs des données transmises par le *wrapper* au moniteur global, comme indiqué sur la figure 5.5. La fonction `sync` est alors utilisée non seulement pour récupérer les valeurs de ces données, mais aussi pour effectuer la mise-à-jour des variables locales de chaque instance en exécutant le bloc d'instructions local à φ.

Nous n'avons pas utilisé des *threads* Unix pour exécuter de façon indépendante les différentes instances φ. En effet, dans cette solution liée au système d'exploitation, le nombre maximal de *threads* et de processus est limité. Par ailleurs, de nombreux changements de contexte peuvent être coûteux. Par conséquent, toutes les instances sont exécutées dans un même contexte, mais chacune utilise une instance indépendante de la classe `data` qui

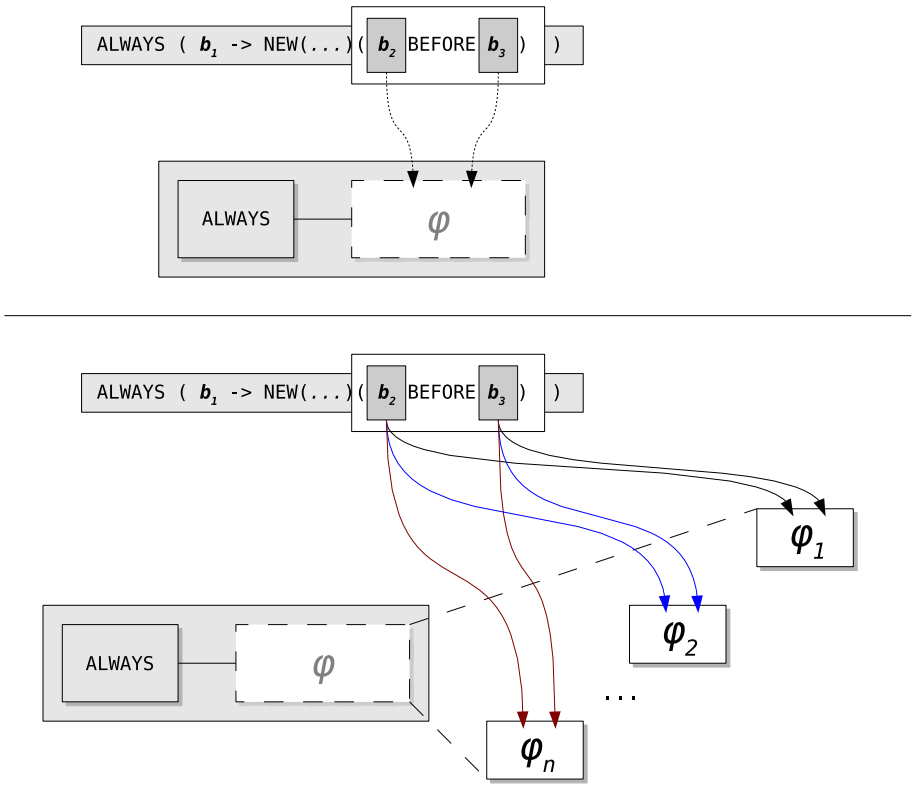

FIGURE 5.5 – Transmission des données aux instances d'une même sous-formule

contient les données. Toutes ces instances partagent aussi une même instance de `data` pour les variables globales.

Concernant le principe d'évaluation présenté dans la section 4.1.2.2, le moniteur global n'appelle plus la fonction d'évaluation du dernier opérateur dans la formule (le **before** dans la formule de la figure 5.5), mais celle du moniteur responsable de démarrer le **new** (moniteur pour l'implication logique dans la formule de la figure 5.5). Bien évidemment, une propriété peut comporter plusieurs opérateurs **new**, imbriqués les uns dans les autres. Dans tous les cas, le moniteur principal démarre le dernier moniteur avant celui du premier **new**, c'est-à-dire le **new** le plus en haut dans l'arbre syntaxique de la formule. Chaque instance reporte l'état de sa propre évaluation, en affichant ses sorties *Valid*, *Checking* et *Pending*, ainsi que les informations relatives aux données locales, de la même façon que ce qui est indiqué dans la section 4.1.2.2 pour les moniteurs simples.

Le mécanisme global de surveillance et d'évaluation des propriétés avec variables locales peut être schématisé comme montré sur la figure 5.6.

Quand le *wrapper* démarre l'évaluation du moniteur global, suite à une notification de la part d'un composant observé (points 2 et 3 sur la figure 5.6), le moniteur amorce le chaîne d'évaluations comme expliqué ci-dessus. Cette évaluation peut donner lieu à la "création" d'une nouvelle instance (point 5), qui est aussitôt attachée au *wrapper* (cf. exemple de la section 4.2.4) comme s'il s'agissait d'un moniteur indépendant. La nouvelle instance comporte une structure contenant ses données locales. Enfin, chaque nouvelle instance est immédiatement évaluée par le *wrapper*, cela avant la prochaine notification de la part de tout composant observé.

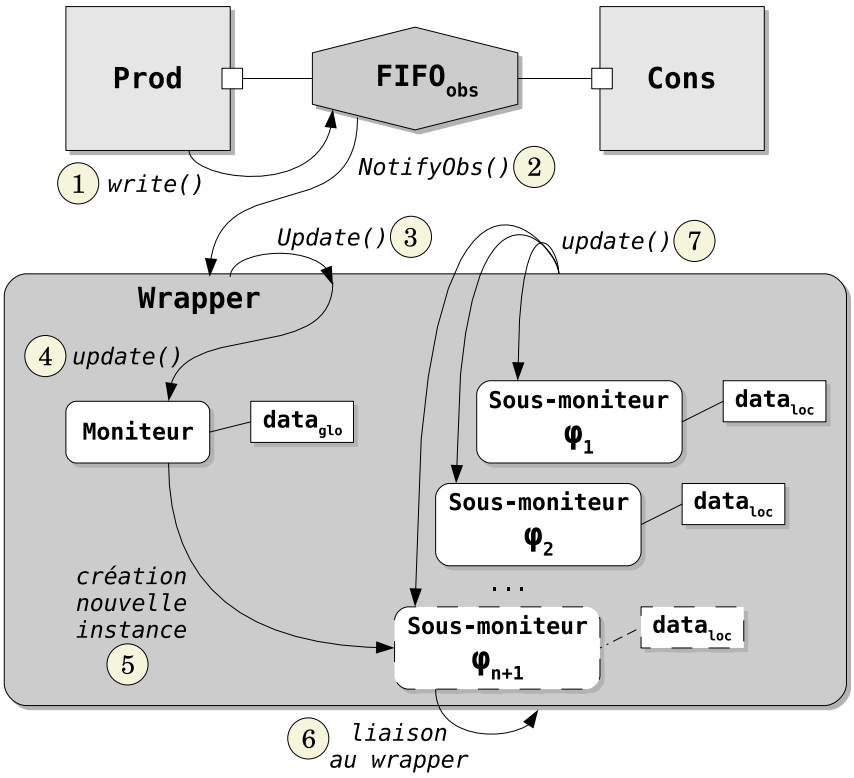

FIGURE 5.6 – Mécanisme global de surveillance

5.2.3.2 Application à l'exemple du *Packet Switch*

Les concepts présentés dans la section 5.2.3.1 peuvent être précisés sur l'exemple du *Packet Switch*. Pour des raisons de clarté, nous montrons le fonctionnement de la solution uniquement pour la propriété `switch_route_prop1`, énoncée dans la section 5.2.1 : "tout paquet en entrée par le port 3 et à destination du port 0 sera inévitablement délivré à la bonne destination". Nous rappelons que la propriété est exprimée en PSL comme suit :

```
vunit switch_route_prop1 {
// HDL_DECLs :
// HDL_STMTs :
// PROPERTY :
assert always(in3.write_CALL() && (in3.write.p1).dest0 ->
              new(true ? pkt packet = in3.write.p1)
                 (eventually! (out0.write_CALL() &&
                               (out0.write.p1).id == packet.id &&
                               (out0.write.p1).data == packet.data)));
}
```

Considérons la trace d'exécution montrée sur la figure 5.7, et observée par la propriété `switch_route_prop1`.

À l'instant t_1 une instance indépendante de la sous-propriété suivante est créée et activée :

```
eventually! ( out0.write_CALL() &&
              (out0.write.p1).id == packet.id &&
              (out0.write.p1).data == packet.data )
```

Cette instance, nommée φ_1, est relative à la transaction 1 de la figure 5.7. Elle comporte la variable locale `packet` dont la valeur est $pkt1$, et son état est *Pending* à t_1. Une nouvelle

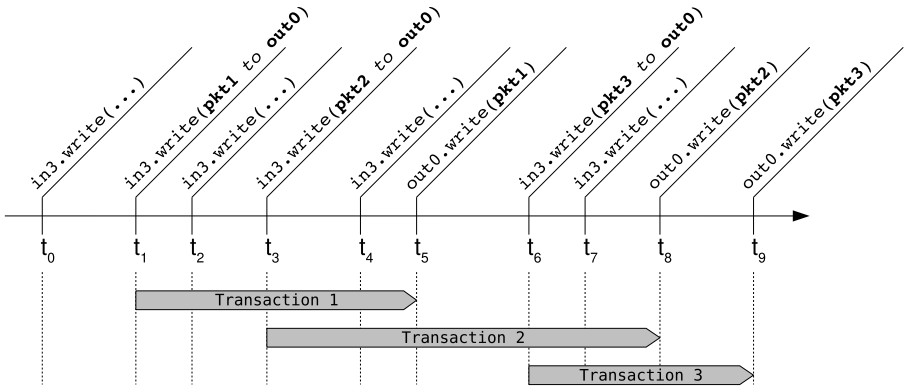

FIGURE 5.7 – Exemple de trace de simulation pour le *Packet Switch*

écriture sur le port `in3` à l'instant t_2 déclenche l'évaluation du moniteur principal pour `switch_route_prop1`, sans toutefois engendrer une nouvelle instance (le paquet émis n'est pas à destination du port 0). À l'instant t_3, une nouveau paquet à destination du port 0 est émis par `in3` : une nouvelle instance φ_2 est activée. À cet instant, les deux instances φ_1 et φ_2 s'évaluent donc de façon concurrente et indépendante. Les deux instances sont dans l'état *Pending*. À t_5 le paquet *pkt*1 est écrit sur le port `out0` : l'instance φ_1 passe de l'état *Pending* à l'état *Holds strongly*, ce qui entraîne sa destruction par arrêt du moniteur correspondant. Par contre, le moniteur relatif à φ_2 est encore actif et continue sa vérification jusqu'à l'instant t_8, instant où une nouvelle instance φ_3 existe pour la transaction 3. À l'instant t_9 la dernière instance créée sera aussi arrêtée après avoir identifié la sortie du paquet *pkt*3. Si la simulation devait s'arrêter à l'instant t_8, le moniteur relatif à φ_3 aurait alors affiché un avertissement en indiquant son état *Pending* et en affichant le paquet et le destinataire concernés.

5.2.4 Spécifications temporelles avec variables

La sémantique proposée dans ce chapitre introduit de façon concise le support pour les variables dans les formules de la logique temporelle de PSL. En particulier, grâce aux règles opérationnelles et à la notion d'interprétation d'une formule dans un environnement, il est possible de caractériser

- l'évolution "pas à pas" des variables lors de la progression temporelle dans la trace de simulation,
- l'influence que ces variables opèrent dans l'évaluation de la propriété,
- la notion de portée des variables.

Les deux premiers points ont rendu possible une implémentation directe et aisée des aspects liés à cette sémantique dans la génération des moniteurs orientés TLM. Le troisième point nous a permis de caractériser de façon cohérente à la fois la présence de variables auxiliaires globales, déclarées à l'aide de la couche modélisation de PSL, et la présence de variables locales, absentes dans [PSL05].

Un certain nombre de travaux liés à l'emploi de variables locales dans l'ABV existent. Les quelques travaux antérieurs à notre contribution ne sont pas adaptés au contexte visé dans cette thèse et, bien qu'ils avancent d'importants éléments de réflexion, ils ne constituent pas des solutions convenables. D'autres travaux concurrents aux nôtres présentent des aspects plus intéressants, mais ils offrent généralement des solutions plus complexes.

Aucun de ces travaux ne considère à la fois les variables de la couche modélisation et celles introduites localement dans une propriété.

5.2.4.1 Vérification avec instances multiples

Dans [OH02], les auteurs présentent une technique de génération de moniteurs d'interface à partir d'expressions rationnelles ; ces moniteurs peuvent surveiller les signaux échangés entre les IPs. La solution, bien que liée au contexte RTL, présente deux concepts intéressants : l'utilisation de variables pour la mémorisation des données et l'introduction d'un opérateur de pipeline. Le langage de [OH02] est inspiré à la fois de ForSpec de Intel [AFF+02] et de Sugar de IBM [BBDE+01], précurseur de PSL, mais les auteurs de [OH02] ne fournissent pas de sémantique formelle. La technique de construction des moniteurs est similaire à celle adoptée dans [BLOF06], dans la mesure où les moniteurs complexes sont assemblés par interconnexion de blocs de base. Par exemple, le bloc élémentaire `Y` est connecté à la suite du bloc `X` dans la séquence `X, Y`. L'utilisation de variables, combinée avec l'introduction de l'opérateur de pipeline `@`, permet de traiter certaines situations où les transactions (il s'agit ici de suites d'échanges de signaux) se superposent. La construction des moniteurs est linéaire en taille et en temps de génération par rapport à la formule, cependant elle comporte plusieurs restrictions. Parmi ces restrictions, la plus importante est la limite de réactivations et d'exécutions concurrentes des moniteurs en pipeline : au plus une activation par étage du pipeline est admise. L'opérateur `@` indique le point où la suite du calcul peut se superposer avec la partie qui le précède. Par exemple, dans la formule `(req @ resp)`, chaque fois que l'opérateur de pipeline est rencontré, deux évaluations concurrentes sont démarrées : la première continue avec l'évaluation de la suite de l'expression (`resp`), la deuxième démarre une nouvelle évaluation. La limitation imposée sur le fonctionnement des moniteurs en pipeline ne permet qu'une seule activation de `resp` à la fois. Si une nouvelle évaluation de `resp` est démarrée pendant qu'une vérification de cet opérande est déjà en cours, une violation de l'opération de pipeline est alors produite (il ne s'agit pas d'une violation de la propriété). Ceci ne permet pas de modéliser une réelle réentrance, telle que présentée tout au long de la section 5.2.

Les auteurs de [LS07] décrivent une technique pour vérifier formellement un sous-ensemble des propriétés de sûreté exprimées en SVA. Chaque propriété est traduite en un circuit synchrone de surveillance, comme le font les auteurs de [MAB06a, BZ07] par exemple. Les moniteurs ainsi obtenus sont connectés au DUV et le nouveau design instrumenté est vérifié par *model checking*. Bien que le contexte soit différent du nôtre, la solution présentée est intéressante car elle permet de vérifier statiquement des propriétés qui comportent des variables locales. Les valeurs de ces variables sont propagées à travers les blocs qui composent les moniteurs. Les auteurs soulignent le fait que l'emploi de variables locales introduit de la concurrence additionnelle dans l'évaluation de la propriété : ceci rejoint le concept d'évaluation réentrante d'une propriété. En vérification formelle, la taille de l'espace de stockage pour les valeurs des différentes copies des variables doit être définie statiquement à l'avance [LS07]. Pour cela, les auteurs identifient plusieurs cas de figure.

1. Quand l'intervalle entre l'affectation et l'utilisation d'une variable est connu, il suffit d'utiliser des registres en pipeline pour garder et propager la valeur de chaque variable. Le nombre de registres est proportionnel à la taille de l'intervalle : cela permet de traiter aussi le cas où plusieurs évaluations de la propriétés se superposent.

2. Quand l'intervalle entre l'affectation et l'utilisation d'une variable est inconnu, il est parfois possible de déterminer le nombre maximal de registres nécessaires. Par exemple, pour l'assertion

   ```
   (req, x = d_in) |=> (!req [*0: ] ##1 (!req && ack && x==d_out))
   ```

   un seul registre suffit car *req* ne peut pas être vrai tout au long de la sous-séquence à droite de l'implication.
3. Dans les autres cas où l'intervalle entre l'affectation et l'utilisation n'est pas connu, des entrées non déterministes pour les valeurs des variables sont utilisées dans les blocs qui composent les moniteurs.

Comme souligné par les auteurs, la troisième solution n'est malheureusement pas suffisante si les variables sont affectées à droite d'une implication et si l'intervalle est de longueur inconnue : l'emploi d'entrées non déterministes provoque la recherche d'une seule valeur possible (i.e. quantificateur existentiel $\exists$) lors de l'étape de *model checking*, mais la vérification devrait être effectuée pour toutes les valeurs des variables (i.e. quantificateur universel $\forall$). Dans ce cas, si la solution 2 n'est pas applicable, la génération des moniteurs de surveillance est alors avortée. En vérification dynamique, la taille de l'espace de stockage pour les différentes copies des variables locales n'a pas besoin d'être connue précisément à l'avance. La seule vraie limitation réside dans la quantité de ressources disponibles lors de la simulation, c'est-à-dire dans la mémoire disponible sur la machine. Par conséquent, nous ne sommes pas confrontés à cette limitation.

Dans la proposition de brevet [PR09], les auteurs décrivent un principe de vérification d'assertions avec variables locales. Ce principe se base sur l'exécution concurrente de plusieurs instances d'un automate modélisant une assertion. Le flot d'exécution envisage l'évaluation d'instances qui peuvent

- être détruites après avoir atteint un état puits (ceci est comparable à notre destruction en cas de rencontre de l'état *Holds strongly*),
- être fusionnées en cas de synchronisation et d'observation de conditions identiques.

Bien que le principe introduit mette en avant des idées intéressantes, comme l'optimisation par destruction d'instances que nous avons définie et implémentée dans nos travaux, il ne fournit pas d'explication claire relative à la gestion des variables, à la sémantique de leur évolution ou encore au fonctionnement des automates.

5.2.4.2 Sémantiques avec variables locales

La notion de variable locale dans la spécification de propriétés, introduite par CBV de Motorola [HLMS02], est adoptée dans SVA depuis la version de 2005 du standard IEEE [SV005], où l'impact des variables sur la satisfaction d'une formule est caractérisé formellement. Cependant, la sémantique des variables locales dans [SV005] est dépourvue d'une définition précise de portée. En effet, la déclaration elle-même des variables n'est pas prise en compte dans les règles sémantiques, et une variable utilisée dans une séquence peut en réalité en sortir. Les fonctions *sample*, *block* et *flow* sont utilisées pour calculer automatiquement la visibilité des variables le long d'une formule. La fonction *sample* retourne les variables qui ont été affectées dans une séquence. La fonction *block* calcule les variables qui restent "bloquées" dans la séquence et qui ne peuvent pas être utilisées à l'extérieur. La fonction *flow* identifie les variables qui peuvent "sortir" d'une séquence en fonction de l'ensemble des variables qui "rentrent" dans la séquence. Ces trois fonctions

sont utilisées dans la définition de la relation de satisfaction, par conséquent, au lieu d'imposer des restrictions syntaxiques, elles influencent la valeur de vérité retournée par la relation de satisfaction [EF08]. Pour cette raison, les règles de la sémantique de [SV005] sont peu claires et elles fournissent parfois des résultats non intuitifs. Par exemple, dans la règle sémantique de l'intersection entre deux séquences, l'affectation d'une valeur à une variable peut avoir deux effets distincts :

- si la variable est modifiée dans une seule des deux séquences, alors elle possédera la nouvelle valeur à la "sortie" de l'intersection ;
- si la variable est modifiée dans les deux séquences, alors elle ne pourra plus être utilisée à l'extérieur de l'intersection avant d'être affectée une nouvelle fois.

Considérons l'environnement initial dans lequel x possède la valeur 8 et la séquence

$S_1 =$ `((true, x=4) ##1 x<5)` **`intersect`** `(true ##1 x>5)`.

Cette séquence est satisfaite sur une trace de longueur 2 : la variable x vaut 4 dans la première sous-séquence, elle vaut 8 dans la deuxième sous-séquence et elle vaut 4 à la sortie de l'intersection. Bien que x soit une variable locale à S_1, elle possède en même temps deux valeurs distinctes. Maintenant considérons le même environnement de départ et la séquence

$S_2 =$ `((true, x=4) ##1 x<5)` **`intersect`** `((true, x=8) ##1 x>5)`.

Cette séquence est également satisfaite sur une trace de longueur 2, mais la variable x ne sera plus visible à la sortie de l'intersection tant qu'elle n'aura pas été modifiée de nouveau. Nous considérons que ce comportement ne traduit pas calirement la notion de variable locale : dans les deux exemples, x devrait être une variable locale à toute la séquence (S_1 ou S_2), mais elle se comporte en partie comme si elle était locale à chacun des opérandes de l'intersection.

Dans [BH06], les auteurs analysent la complexité de la satisfaction des propriétés SVA, en particulier en présence de variables locales. Comme expliqué ci-dessus, ils soulignent la nécessité de s'assurer que les variables dans la propriété possèdent toujours une valeur bien définie, principalement en cas de modifications divergentes dans les deux opérandes d'un opérateur binaire. Dans le manuel SVA, ceci est effectué à l'aide des trois fonctions *sample*, *block* et *flow*. Bien que ces fonctions déterminent la visibilité de chaque variable après l'opérateur binaire, la solution n'empêche pas une évaluation de la propriété avec deux valeurs différentes pour une même variable. La sémantique de [BH06] utilise les trois fonctions *sample*, *block* et *flow* de [SV005], mais présente des définitions plus simples pour la relation de satisfaction des formules. Cependant, ces définitions ne résolvent pas le problème des valeurs divergentes dans les deux opérandes d'un opérateur binaire. Par ailleurs, dans le cas de l'opérateur **`intersect`** mentionné ci-dessus, la sémantique proposée induit une asymétrie dans l'affectation des valeurs aux variables, comme remarqué dans [EF08] : si une variable x est modifiée à gauche de l'intersection, alors elle gardera cette nouvelle valeur, sinon elle aura la valeur possédée dans l'opérande droit. Alors que les trois fonction *sample*, *block* et *flow* devraient empêcher d'utiliser x par la suite, elles ne sont plus utilisées explicitement dans les règles sémantiques de la relation de satisfaction. Cette sémantique ne permet donc pas de savoir si x sera visible ou pas par la suite.

Dans la révision de 2009 du standard IEEE pour SVA [SV009], on trouve la caractérisation de la règle pour la *déclaration* des variables. Cette règle permet d'introduire une nouvelle variable qui ne sera pas utilisable tant qu'elle n'aura pas été affectée au moins

une fois. Cependant, même dans la nouvelle sémantique, le comportement des variables modifiées dans les deux opérandes d'un opérateur binaire présente les problèmes décrits ci-dessus pour les sémantiques de [SV005] et de [BH06].

Les auteurs de [EF08] proposent une sémantique pour une logique temporelle basée sur les expressions rationnelles, fortement connexe à PSL, enrichie par des variables locales. Une solution alternative à la sémantique de [SV005, BH06] est ici définie. L'élément principal de cette nouvelle solution est la gestion explicite de la portée des variables, à l'aide des deux nouveaux opérateurs `new` et `free`. Le premier permet de déclarer, et donc d'introduire, une nouvelle variable locale. Le deuxième permet de "libérer" une variable, en la plaçant hors de portée. Selon les auteurs, la portée des variables ne peut pas être déterminée automatiquement sans briser certaines propriétés algébriques, comme la distributivité de l'union de séquences par rapport à l'intersection.

Cette nouvelle sémantique comporte la notion de "accord" entre deux valuations par rapport à un ensemble $\mathcal{V}$ de variables, noté $\gamma_1 \overset{\mathcal{V}}{\sim} \gamma_2$: une valuation γ_1, c'est-à-dire une liste d'associations de valeurs aux variables, est en accord avec une valuation γ_2 si et seulement si toutes les variables dans γ_1 possèdent la même valeur dans γ_2. Cette notion est utilisée dans la définition de la relation de satisfaction du langage de [EF08]. Lors de l'évaluation d'expressions booléennes, la valuation γ_2 obtenue après l'évaluation de l'expression doit être en accord avec la valuation de départ γ_1. Dans le cas des opérateurs binaires, si une variable locale à toute la formule est modifiée de façon discordante dans les deux opérandes, alors il y aura forcément un moment où la valuation γ_1 dans un opérande ne sera pas en accord avec la prochaine valuation γ_2. Par conséquent, une formule avec modifications discordantes dans les deux opérandes d'un opérateur binaire ne pourra jamais être satisfaite. Dans cette sémantique, les deux séquences S_1 et S_2 définies précédemment ne sont pas satisfaites.

La révision de 2010 du standard IEEE pour PSL [PSL10] adopte cette même approche pour caractériser les variables locales. À l'aide de l'exemple de la propriété P_2 qui suit, mentionnée dans le manuel de référence, il est possible d'analyser la différence principale entre la sémantique de [EF08, PSL10] (PSL 2010), celle de [SV005, SV009] (SVA) et celle que nous proposons. Dans la syntaxe de [PSL10], la structure `[[ stmt ]]` permet de déclarer des variables locales ou de les modifier. La propriété P_2 utilise les deux compteurs `count_r` et `count_w` pour vérifier si le nombre de requêtes de lecture dans une FIFO est égal au nombre de requêtes d'écriture, cela entre deux moments où la FIFO est vide. Nous considérons d'abord la sémantique de [EF08, PSL10]. Une première version de P_2 est exprimée comme suit (nous adoptons ici une syntaxe en style C++) :

```
always ( [[ int count_r = 0; int count_w = 0; ]]
         { fifo_empty;
            { read_req[->][[count_r++;]][*]; !read_req[*] }
            &&
            { write_req[->][[count_w++;]][*]; !write_req[*] };
           fifo_empty } |-> {count_r == count_w} )
```

Dans cette version l'opérateur `free` n'est pas utilisé. Puisque l'opérande gauche de la conjonction entre SEREs incrémente `count_r`, mais l'opérande droit ne le fait pas, la conjonction n'est jamais satisfaite (de façon similaire, `count_w` est incrémenté dans l'opérande droit uniquement). Par conséquent, la SERE à gauche de l'implication n'est jamais reconnue et l'implication suffixe sera toujours vraie par vacuité (**faux** $\Rightarrow$ *condition*). Pour exprimer cette propriété avec la sémantique de [EF08, PSL10], il faut ajouter `free(count_w)`

au début de l'opérande gauche du `&&`, et `free(count_r)` au début de l'opérande droit, afin de placer les deux variables "hors de portée". Dans ce cas, il faut remarquer que `count_w` par exemple ne pourra être utilisée nulle part dans l'opérande gauche de la conjonction, bien que déclarée locale à toute la propriété sous la portée du **`always`**.

Avec la sémantique de [SV005, SV009] (SVA), la propriété P_2 aurait pu être satisfaite, puisque chaque variable était modifiée uniquement d'un côté de la conjonction. Cependant, l'utilisation d'un même compteur dans les deux opérandes aurait pu produire des effets tels que ceux décrits précédemment pour les séquences S_1 et S_2.

Nous considérons que dans les deux sémantiques ci-dessus (PSL 2010 et SVA), l'interprétation de *variable locale*, bien que formellement définie dans les deux cas, ne correspond pas exactement à celle usuellement attendue : dans ces sémantiques, une variable locale à une propriété peut ne pas toujours être vue comme telle dans deux sous-formules de la propriété.

Dans notre approche, si une variable est déclarée locale à une formule φ, elle pourra être modifiée uniquement au niveau "le plus externe" de φ. Donc, bien qu'elle soit visible partout dans φ, elle ne sera pas modifiée dans ses sous-formules. Ceci nous garantit une lecture homogène de la valeur de la variable, partout dans la formule. Dans la propriété P_2, on écrirait alors[10] :

```
always ( new(int count_r = 0, int count_w = 0;
             if(read_req){count_r++;}
             if(write_req){count_w++;})
            ({ fifo_empty;
                { read_req[->][*]; !read_req[*] }
                &&
                { write_req[->][*]; !write_req[*] };
               fifo_empty } |-> {count_r == count_w}) )
```

Les variables `count_r` et `count_w` sont alors réellement locales à toute la formule sous l'opérateur **`always`**, et elles conservent toujours une valeur cohérente.

Bilan

Nous avons décrit, dans les chapitres 4 et 5, les concepts à la base de notre solution d'ABV dynamique pour SystemC TLM. Le chapitre 4 a présenté les principes de construction des moniteurs de surveillance orientés TLM, ainsi que la méthode d'instrumentation d'un design. Le chapitre 5 complète cette présentation par notre caractérisation formelle des variables globales et locales dans PSL, ainsi que les principes de leur mise en oeuvre.

Dans les deux chapitres qui suivent, nous allons présenter la mise en oeuvre de ces travaux dans notre outil prototype ISIS, et les résultats de l'utilisation de ce prototype avec un ensemble varié de cas d'étude.

[10] Nous rappelons que nous ne considérons pas les SEREs. Cette propriété, visiblement conçue pour le niveau RTL, n'est considérée ici que pour ses aspects liés aux variables.

Chapitre 6

Mise en oeuvre

Introduction

Les principes présentés dans les chapitres 4 et 5 ont été entièrement implémentés dans l'outil ISIS[1], un prototype pour la vérification semi-formelle de systèmes matériels et logiciels en cours de conception. ISIS permet la surveillance, par instrumentation automatisée du DUV, de propriétés fonctionnelles et temporelles exprimées en PSL. Ces propriétés sont automatiquement transformées en moniteurs de surveillance SystemC (cf. section 4.1.2), liés au design à vérifier par un mécanisme d'observation original (cf. section 4.2.2). L'objectif de la section 6.1 est de présenter le flot complet d'ISIS, en se focalisant sur les étapes clés. En particulier, nous examinerons les choix auxquels l'utilisateur est confronté dans le flot, qui peuvent être issus de choix relatifs à la spécification et à la vérification.

Le concept central de l'outil est le fait qu'il génère une version instrumentée du design à vérifier. Le code ainsi obtenu peut être simulé par n'importe quel moyen existant, par exemple à l'aide d'un simulateur commercial avec support pour SystemC TLM, ou par simple compilation via un compilateur C++ et exécution de l'application obtenue. Au cours de nos travaux, nous avons toujours utilisé cette dernière solution. La technique d'instrumentation mise en œuvre à l'heure actuelle dans ISIS produit un modèle où

- la surveillance des propriétés se fait toujours "en direct", c'est-à-dire durant la simulation même du design ;
- la vérification est effectuée dans le même contexte d'exécution que celui de la simulation, par le même processus qui fait appel à la fonction `sc_start` (cf. section 2.1.2.3).

Dans le but d'améliorer cette approche, nous avons conduit des expérimentations visant à une parallélisation des opérations de surveillance des propriétés, cela afin d'étudier dans quelle mesure il pouvait être intéressant de dissocier la simulation de la fonction de *monitoring*. Ces études sont relatées dans les sections 6.2.1 et 6.2.3.

Comme nous l'avons souligné dans la section 2.2.1.2, l'efficacité de la vérification durant la simulation est étroitement liée à l'ensemble de jeux de test utilisé. Cet ensemble doit être suffisamment riche et pertinent. ISIS n'a pas vocation à couvrir la génération de *stimuli* pour le DUV, c'est pourquoi, afin de rendre notre solution plus complète, nous avons effectué des études où le prototype a été couplé avec TOBIAS[2] [LDdB+07], un

[1] ISIS : *ABV of SystemC TLM Designs* (`http://tima.imag.fr/vds/Isis/index.html`).
[2] TOBIAS : *A tool for combinatorial testing* (`http://vasco.imag.fr/Tobias`).

outil d'aide à la génération massive de séquences de tests développé au LIG (Laboratoire d'Informatique de Grenoble). Ces travaux sont présentés dans les sections 6.2.2 et 6.2.3.

Le travail relaté dans ce chapitre a fait l'objet des publications [FP10b] et [FPLdB08], ISIS a fait l'objet d'un dépôt APP[3] en décembre 2010.

6.1 ISIS

ISIS [FP10b] est un prototype entièrement réalisé en Java, avec une interface utilisateur graphique et avec la possibilité de fonctionnement en ligne de commande, cela par exécution d'un fichier de directives. Ce fichier utilise le format XML (*eXtensible Markup Language*) pour organiser les directives et les options de fonctionnement. Le prototype permet

- la génération automatique de moniteurs SystemC de surveillance a partir de spécifications en PSL,
- la génération automatisée de classes C++ et de composants SystemC pour la liaison des moniteurs de surveillance au DUV.

6.1.1 Flot d'instrumentation de designs SystemC pour l'ABV dynamique

La figure 6.1 montre le flot d'ABV proposé. Dans ce flot, les deux ellipses du haut indiquent l'intervention d'ISIS. L'outil lit un ensemble d'assertions écrites en PSL, selon la syntaxe et les extensions détaillées dans la section 4.2.3 et dans le chapitre 5, et il construit les classes C++ pour les moniteurs orientés SystemC TLM.

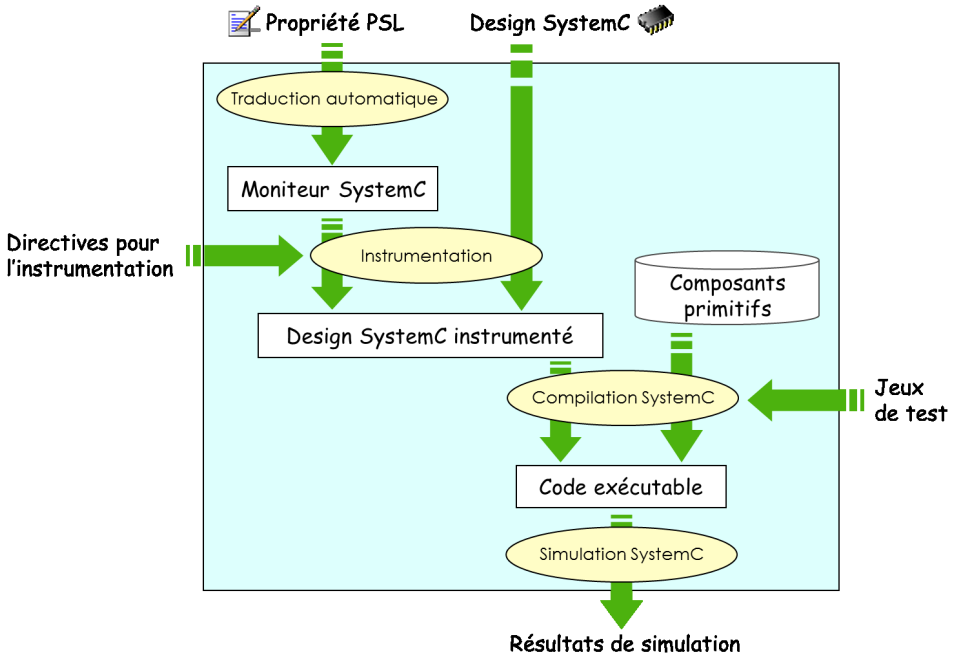

FIGURE 6.1 – Flot d'ABV avec instrumentation du DUV à l'aide d'ISIS

[3] Agence pour la Protection des Programmes (`http://app.legalis.net`).

Suivant les choix des composants que l'utilisateur souhaite surveiller dans le design, ISIS établit des correspondances entre les identifiants utilisés dans les propriétés et les composants réellement présents dans le design. Ces "correspondances" sont traduites par la génération d'un ensemble de classes qui permet la surveillance des propriétés, c'est-à-dire l'évaluation des moniteurs durant la simulation du design. Des instances de ces classes sont automatiquement ajoutées au code source du design.

Le code instrumenté du design est compilé en utilisant notre bibliothèque de moniteurs élémentaires (cf. section 4.1.2.1), et la simulation SystemC du DUV peut être effectuée normalement. Durant la simulation, les moniteurs signalent toute violation des assertions, en fournissant des informations utiles relatives à l'état des formules PSL (sorties *Valid*, *Checking* et *Pending*) et aux données observées (occurrences des communications, paramètres, valeurs de retour et variables auxiliaires).

ISIS requiert donc différentes entrées utilisateur. Les énoncés des propriétés PSL peuvent être organisés en plusieurs fichiers texte, ou saisies directement à l'aide de l'interface graphique. Les en-têtes des classes du code source du design doivent aussi être analysées, puisque l'outil doit être capable d'identifier tous les composants et leurs méthodes. Il s'agit d'un aspect crucial pour l'instrumentation. Enfin, l'utilisateur doit identifier les composants qu'il souhaite surveiller, c'est-à-dire les éléments du design visés par ses assertions. Ce dernier choix fait aussi partie des entrées indispensables à l'outil.

La sortie finale produite par ISIS est le code du design instrumenté. Comme expliqué dans la section 4.2.2 relative au modèle de surveillance, outre les descriptions des moniteurs, l'instrumentation comporte des classes spéciales pour les observateurs (*wrappers*) et pour les composants observés. Les fichiers générés sont classifiés suivant trois catégories :

- les moniteurs et les observateurs (*wrappers*),
- les nouvelles classes pour les composants observés,
- la version modifiée de tous les éléments du design qui contiennent au moins un composant observé (le type d'un ou plusieurs sous-composants aura été remplacé par le sous-type observé dans le fichier original).

Une nouvelle version du fichier principal, i.e. le fichier qui contient le point d'entrée du programme, est aussi générée : les instances de tous les *wrappers* avec les moniteurs de surveillance sont ajoutées avant l'instruction de début de simulation. L'ensemble des fichiers générés peut être facilement intégré dans les sources originales du design. Dans les cas les plus simples, l'utilisateur peut compiler directement la nouvelle version du fichiers principal, en indiquant le chemin d'accès vers le répertoire qui réunit les fichiers nouvellement générés. Par exemple, dans une approche à base de *Makefile*, une démarche systématique est possible. Un exemple commenté de *Makefile* type est disponible dans l'annexe A. Dans les cas où les environnements de compilation plus complexes sont utilisés, les fichiers générés doivent parfois être déplacés afin d'être intégrés proprement dans les sources originales. Il est important de remarquer que le fonctionnement des moniteurs ne nécessite aucune entrée supplémentaire. La simulation originale peut être menée en utilisant les jeux de test initiaux.

Enfin, quel que soit le mode de fonctionnement choisi pour ISIS (mode graphique ou *script* en ligne de commande), les mêmes interventions de la part de l'utilisateur restent nécessaires. Dans la section suivante, nous allons nous concentrer sur les étapes qui demandent ces interventions, et nous allons analyser leurs motivations ainsi que leurs impact.

6.1.2 Étapes fondamentales de l'instrumentation

Analyse de l'architecture du DUV Comme nous l'avons souligné auparavant, les propriétés exprimées au niveau transactionnel portent essentiellement sur les communications entre les blocs du système. Pour permettre à l'outil d'instrumenter le code du DUV en suivant le principe décrit dans le chapitre 4, le design doit être analysé afin d'en extraire la liste des composants et la liste des méthodes de chaque composant. Tous ces composants et ces méthodes pourront alors être impliqués dans les assertions. La section de l'outil qui s'occupe de cette analyse syntaxique produit une représentation hiérarchique des composants et de leurs méthodes de communication. Cette représentation hiérarchique utilise encore XML comme format de stockage, car ce standard[4] s'avère particulièrement adapté à la représentation d'informations textuelles structurées en champs arborescents. Nous avons utilisé la sortie produite par l'outil gccxml[5] lors de l'analyse du code C++ du design. Cette sortie, sous la forme d'un fichier XML non hiérarchisé de taille considérable, est utilisée par ISIS pour produire une deuxième représentation XML hiérarchisée et optimisée de l'architecture du design.

Choix des composants observés et écriture des assertions Quand un développeur traduit une spécification en une implémentation, il est contraint de choisir des noms et des structures concrets. De la même façon, quand il souhaite énoncer une assertion qui représente une fonctionnalité précise de son implémentation, il est contraint de choisir les communications et les éléments concernés par l'assertion. Cela peut s'expliquer à l'aide d'un exemple. Considérons la plate-forme avec contrôleur DMA de la section 2.1.3.2 (cf. figure 2.8 à la page 19). Dans ce modèle, une propriété à respecter est la suivante : "chaque fois que le processeur commande un transfert au DMA, il attendra une notification de fin de transfert par le DMA avant de commander un nouveau transfert". Cette propriété peut être traduite en PSL selon le schéma suivant :

```
always ( cpu_starts_dma_transfer ->
         next (end_transfer_notification
               before
               cpu_starts_dma_transfer) )
```

Il reste à clarifier le sens de `cpu_starts_dma_transfer` et `end_transfer_notification` sur cette plate-forme. C'est à l'utilisateur d'apporter cette signification. Par exemple, la définition de "commander un transfert" (i.e. `cpu_starts_dma_transfer`) dans cette plate-forme est "écrire la valeur *start* dans le registre de contrôle du DMA". De même, la notification de la fin d'un transfert se traduit par une interruption via le signal `dma_irq` du design. Par ailleurs, puisqu'il s'agit de vérifier si ce protocole est respecté par l'application, les programmations du contrôleur DMA observées seront celles effectuées via le port initiateur du CPU. L'énoncé PSL *abstrait* et générique ci-dessus est alors décliné *concrètement* comme suit :

```
vunit dma_protocol_v1 {
  // HDL_DECLs :
  int CONTROL = 0x4000 + pv_dma::CONTROL;
  int START = pv_dma::START;
  // HDL_STMTs :
```

[4] XML est un standard du consortium *World Wide Web* (`http://www.w3.org/standards/xml`).

[5] Une extension de gcc/g++ pour la génération, à partir de la représentation intermédiaire de gcc, d'une description XML des classes et des structures d'un programme C++ (`http://www.gccxml.org`).

```
  // PROPERTY :
  assert
  always ( (cpu_init_port.write_CALL() &&
            cpu_init_port.write.address == CONTROL &&
            cpu_init_port.write.data == START) ->
           next (dma_irq
                 before
                 (cpu_init_port.write_CALL() &&
                  cpu_init_port.write.address == CONTROL &&
                  cpu_init_port.write.data == START)) );
}
```

Bien évidemment, plusieurs solutions techniques sont possibles. Par exemple, un fichier de correspondances pourrait traduire, en une seule fois, toutes les conditions abstraites par leur expression concrète. La couche modélisation de PSL peut aussi permettre d'exprimer chaque condition de l'assertion, en rendant cette dernière plus lisible :

```
vunit dma_protocol_v2 {
  // HDL_DECLs :
  int CONTROL = 0x4000 + pv_dma::CONTROL;
  int START = pv_dma::START;
  bool cpu_starts_dma_transfer = false;
  bool end_transfer_notification = false;
  // HDL_STMTs :
  cpu_starts_dma_transfer = false;
  end_transfer_notification = false;
  if( cpu_init_port.write_CALL() &&
      cpu_init_port.write.address == CONTROL &&
      cpu_init_port.write.data == START )
  { cpu_starts_dma_transfer = true; }
  if( dma_irq )
  { end_transfer_notification = true; }
  // PROPERTY :
  assert
  always ( cpu_starts_dma_transfer ->
           next (end_transfer_notification
                 before
                 cpu_starts_dma_transfer) );
}
```

Il est important de remarquer que cette nécessité de clarification de propriétés relativement aux identificateurs du design n'est pas spécifique à TLM. Au niveau RTL aussi, il est nécessaire de préciser les assertions en termes d'événements sur les signaux du design.

En mode graphique, le choix des méthodes observées (dans l'exemple, les écritures dans le port initiateur du CPU et les changements de valeur du signal d'interruption) se fait explicitement. Juste avant l'écriture de chaque assertion, l'utilisateur est amené à choisir les méthodes concernées. La figure 6.2 montre la sélection de la méthode `write` du port initiateur du CPU (composant `testbench` dans le design). Après cette étape, un résumé textuel de toutes les méthodes choisies est proposé, en indiquant ainsi à l'utilisateur quels sont les noms des méthodes booléennes générées (suffixes `CALL`, `START` et `END`, cf. section 4.2.3) et quels sont les paramètres et les valeurs de retour de chaque méthode à sa disposition lors de l'écriture de l'assertion.

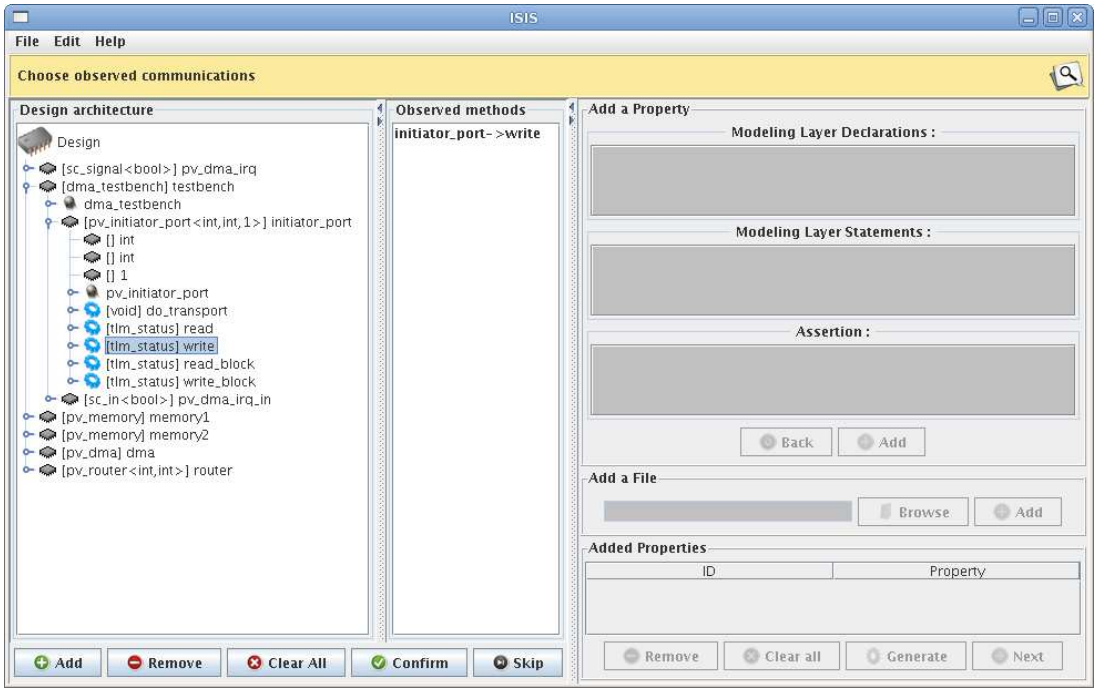

FIGURE 6.2 – Choix de la méthode observée `write` du port initiateur via l'interface graphique d'ISIS

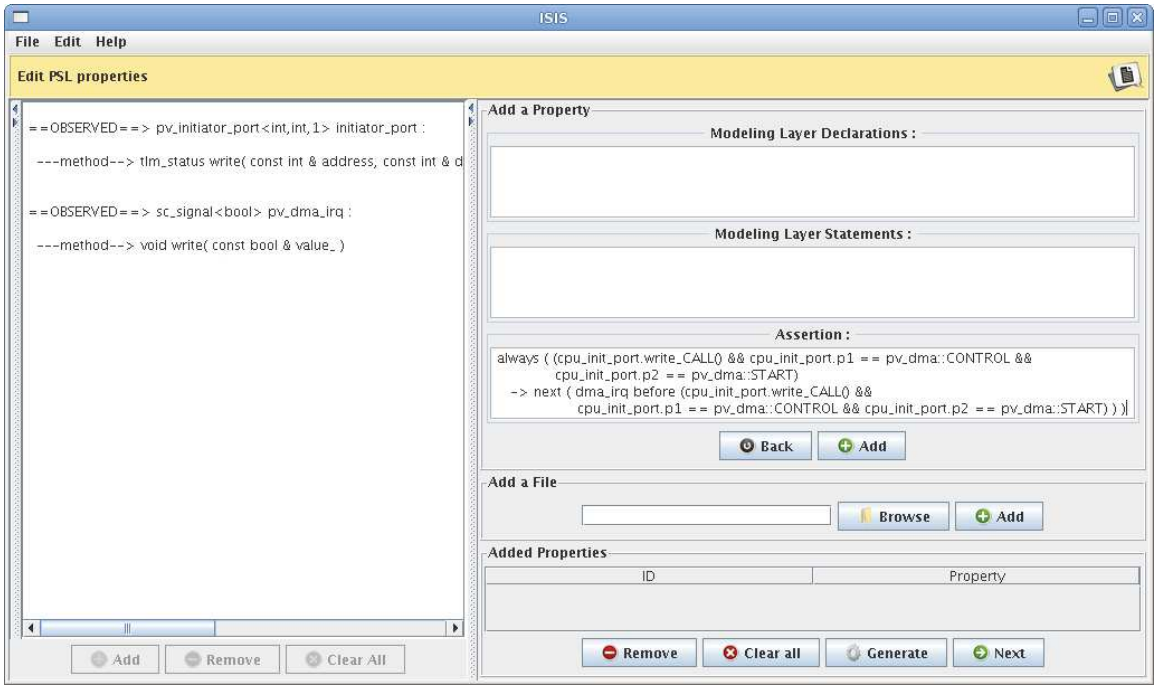

FIGURE 6.3 – Saisie de la propriété à l'aide de l'interface graphique d'ISIS

En mode *script*, le choix des méthodes observées se fait implicitement, par analyse syntaxique du texte de la *vunit* qui contient l'assertion : en lisant le texte `cpu_init_port.write_CALL()`, par exemple, ISIS déduit automatiquement qu'il s'agit de l'observation de la méthode `write` du `cpu_init_port`[6]. Le fichier de *script* se limite alors à indiquer le chemin d'accès vers tous

[6] Dans le *script*, il est possible de renommer une méthode suivant sa signature, de façon à permettre

les fichiers PSL avec les propriétés. En mode graphique la saisie de chaque propriété se fait à l'aide de champs dédiés, comme montré sur la figure 6.3.

Instrumentation du DUV Comme pour l'expression précise des conditions sur les communications et sur leurs paramètres, les identifiants utilisés dans une propriété doivent être liés aux composants du DUV par l'utilisateur. Par exemple, les noms des identifiants `cpu_init_port` et `dma_irq` dans la propriété `dma_protocol_v1` doivent être liés respectivement au port initiateur du CPU et au signal d'interruption utilisé par le DMA. Tous les identifiants dans le texte d'une propriété sont considérés comme les paramètres formels d'une fonction. Cela permet de réutiliser la propriété pour d'autres parties du design, ou pour d'autres designs, de la même façon qu'une fonction peut être appelée plusieurs fois avec des paramètres effectifs différents.

En mode graphique, la mise en relation des identifiants avec les composants se fait par exploration interactive de l'architecture du design, de façon similaire au choix des méthodes observées. Chaque identifiant est sélectionné et mis en relation avec un composant de l'architecture, comme montré sur la figure 6.4. Tous les appels de méthodes et toutes les références aux paramètres et aux valeurs de retour sont automatiquement liés (cf. tableau en bas à gauche dans la figure 6.4).

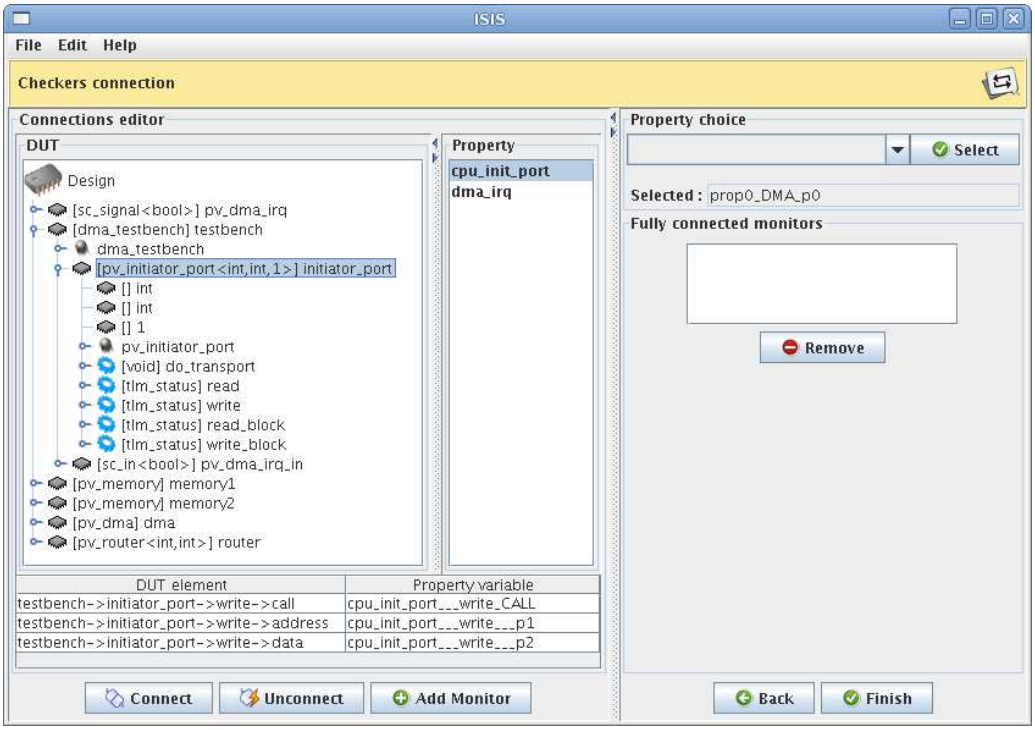

FIGURE 6.4 – Liaison du moniteur au DUV via l'interface graphique d'ISIS

En mode *script*, un fichier XML associé à chaque propriété contient la liaison des identifiants de la propriété aux composants du DUV. Une mise en correspondance dans ce fichier de liaison s'écrit simplement comme suit (nous rappelons que le composant `testbench` représente le CPU dans ce design) :

```
<link>
```

l'observation de plusieurs méthodes avec le même nom mais avec des paramètres différents.

```
    <component>top/testbench/initiator_port</component>
    <variable>cpu_init_port</variable>
  </link>
```

Suite à cette dernière étape, ISIS dispose de toutes les informations nécessaires pour instrumenter automatiquement le design :

- l'ensemble des propriétés pour la génération des moniteurs et des *wrappers* ;
- les classes des composants observés, et les méthodes observées, pour la génération des sous-classes ;
- la liaison entre les identifiants de la propriété et les composants du design pour la substitution de chaque composant par sa version observée.

Aucune autre intervention de la part de l'utilisateur n'est nécessaire. Par exemple, l'outil s'occupe de générer une implémentation des méthodes événementielles `CALL`, `START` et `END` dans la sous-classe de tout composant observé.

6.2 Travaux connexes

ISIS produit un code SystemC qui est prêt à être simulé. Toutefois, il s'agit d'une instrumentation du code source où la vérification est effectuée dans le même contexte d'exécution que celui du processus responsable de la simulation SystemC. L'un des intérêts principaux des modèles transactionnels étant leur rapidité de simulation, toute technique de vérification dynamique se doit de minimiser son impact afin de pénaliser le moins possible le temps d'exécution. Les résultats expérimentaux du chapitre 7 montrent globalement une bonne performance pour nos moniteurs de surveillance, dans la mesure où la simulation du design instrumenté n'est que modérément ralentie par rapport à celle du design original. Nous nous sommes toutefois interrogés sur l'opportunité de dissocier la simulation et le *monitoring* en "parallélisant" ces tâches dans des processus disjoints. Le flot de données étant unidirectionnel (du design simulé vers les moniteurs), nous espérions ainsi pénaliser très modérément le processus de simulation, seulement impacté par des écritures de données "bufferisées" à destination des moniteurs.

6.2.1 Parallélisation

L'idée initiale visait l'emploi de MPICH/MPICH2 [MPI], l'une des implémentations de MPI [GLS99] existantes, pour effectuer la simulation et le *monitoring* en parallèle. Les moniteurs ne doivent en aucun cas perturber le comportement du design, ils se limitent à recevoir et analyser les données du DUV, sans lui transmettre aucune information. Par conséquent, même dans le cas d'un design instrumenté, la simulation pourrait se dérouler normalement, avec comme seul complément la transmission des données à observer à un processus indépendant, destiné à appeler les moniteurs. Malheureusement, l'implémentation d'un premier prototype, qui utilisait des communications avec tampon sur réseau *ethernet*, n'a pas apporté les résultats attendus. Le nombre de notifications de la part des composants observés, et donc le nombre de communications depuis le processus dédié à la simulation vers celui dédié au *monitoring*, était très élevé. Cette tâche de communication s'est avérée particulièrement pénalisante car d'un coût élevé, du fait de la limitation en taille du tampon de transmission. Nous nous sommes alors intéressés à d'autres moyens pour dissocier la simulation de la vérification, cela en utilisant des mécanismes de communication inter-processus (IPC, *interprocess communication*) de plus bas niveau sous

Unix/Linux. Nous avons effectué des expérimentations avec deux ou plusieurs processus communiquant par FIFO ou au moyen de segments de mémoire partagée et se synchronisant à l'aide de sémaphores. Cette solution, bien que toujours trop désavantageuse en termes de temps de simulation, nous a conduits à d'autres observations quant à l'utilité de la nouvelle infrastructure.

En effet, trois situations différentes peuvent se présenter en cas d'erreur durant une simulation :

1. Une erreur dans le modèle apparaît explicitement sur la trace de simulation. Il s'agit du cas où les moniteurs de surveillance sont moins utiles, puisque leur contribution se limite à la confirmation de l'erreur ou à l'amélioration de sa visibilité et de sa compréhension.
2. Une erreur n'apparaît pas sur la trace de simulation. Dans ce cas, le moniteur s'avère indispensable car il signale et informe sur le comportement inattendu.
3. Une erreur n'apparaît pas explicitement sur la trace de simulation, mais la simulation ne se déroule plus normalement. Par exemple, elle pourrait être avortée brusquement.

Dans le cas d'un abandon soudain de la simulation, par exemple durant une opération de communication, le processus système peut être arrêté avant que le moniteur n'ait eu le temps de signaler une violation, ni de notifier l'état de la formule à l'utilisateur. Nous avons alors développé un prototype d'infrastructure de simulation où le processus initial se divise en deux ou plus processus fils : l'un des processus fils est dédié à la simulation et les autres s'occupent de la surveillance des propriétés. Une exemple d'application de cette solution est présenté dans la section 6.2.3. Le principe général est schématisé sur la figure 6.5.

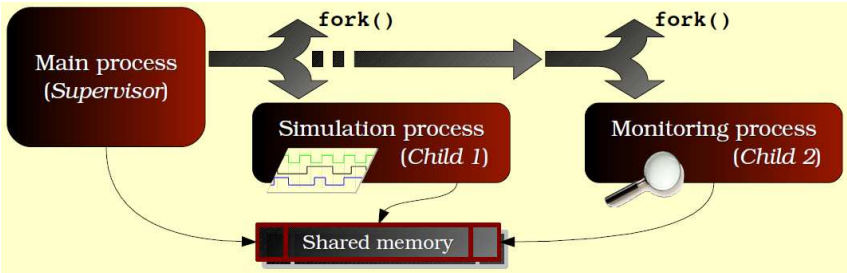

FIGURE 6.5 – Infrastructure avec plusieurs processus

Des segments de mémoire partagée sont utilisés pour la transmission des données entre les processus fils (la simulation écrit ses données dans la mémoire, les processus de surveillance consomment ces données). Le processus parent supervise l'exécution de tous les autres : si la simulation est arrêtée malencontreusement (mort du processus de simulation), il en informe les processus de surveillance, qui lisent les données restantes dans la mémoire partagée, les analysent et reportent l'état de chaque propriété avant de s'arrêter convenablement. De façon générale, chaque composant observé reçoit un identifiant unique. L'un des segments de mémoire partagée contient une FIFO avec la liste ordonnée des sujets qui effectuent des notifications. Les autres segments sont des tampons pour la transmission des données relatives aux communications. L'accès à ces tampons est gouverné par des sémaphores.

Il convient de rappeler que la SCV [Gro03] offre une API permettant l'enregistrement des transactions dans une base de données (cf. section 4.2.5). L'utilisation d'une base de données commune, couplée avec l'infrastructure à plusieurs processus décrite ci-dessus, pourrait constituer une alternative à cette solution.

6.2.2 Campagnes de test

Toutes les techniques et les solutions de vérification dynamique étudiées jusqu'à ce point portent sur la spécification du comportement attendu et sur l'analyse du résultat de simulation. L'infrastructure introduite dans la section précédente n'est qu'une amélioration possible de la démarche d'analyse. Comme expliqué dans la section 2.2.1.2, le principe de vérification dynamique implique également la présence d'un ensemble de jeux de tests suffisamment riche et pertinent. Il s'agit d'un aspect particulièrement délicat, puisque la qualité des tests influence inévitablement celle de la vérification en cours de simulation. L'étape de test, aussi bien pour les systèmes logiciels que matériels[7], comporte typiquement trois éléments :

1. la sélection ou la génération des ensembles et des séquences de données en entrée du système,
2. l'exécution du système avec ces entrées,
3. l'analyse du résultat, qui produit un verdict pour le test ("oracle").

Dans le contexte de nos travaux, le troisième point s'appuie essentiellement sur les moniteurs de surveillance pour des spécifications PSL. Le deuxième point correspond à la simulation SystemC avec les entrées définies dans le premier point. Celui-ci utilise le plus souvent une approche qui ne considère que les interfaces des modules, le principal avantage étant son indépendance de toute évolution future de l'implémentation interne. Puisque les systèmes décrits à haut niveau d'abstraction permettent l'exécuter de millions de jeux de test, l'emploi d'un outil d'aide à la génération de ces jeux de test est primordial.

6.2.2.1 Génération de tests pour des spécifications de haut niveau

Il existe une panoplie d'outils permettant la génération automatique de données pour le test à haut niveau d'abstraction. Les approches de *Model-Based Testing* (MBT) [UL07], génèrent les données ainsi que "l'oracle" pour le résultat à partir d'une analyse de la spécification formelle du système. Par exemple, TGV [JJ05] permet la synthèse automatique de tests à partir de la spécification d'un système réactif. Dans ces approches, l'automatisation permet l'obtention de tests bien conçus avec un coût relativement faible (il faut néanmoins considérer l'investissement pour l'automatisation), mais une erreur dans le modèle entraîne toute une collection de tests défectueux. Par ailleurs, ce type d'approche peut être limité par le pouvoir d'expression du langage de spécification. Une alternative est alors la séparation entre la génération des données du test et la spécification du DUV. Cette spécification est utilisée uniquement pour la construction de l'oracle (les moniteurs de surveillance dans notre cas), et les entrées sont obtenues par d'autres moyens. Les techniques de génération aléatoire des données offrent un moyen simple et rapide pour générer un très grand nombre de tests. Le problème principal est la redondance des tests

[7] Il s'agit néanmoins du test d'un *modèle logiciel* de circuit. Les techniques de test de prototypage et de fabrication ne font pas l'objet d'une étude dans cette thèse.

produits : la génération aléatoire révèle moins facilement les erreurs qui dépendent d'une combinaison spécifique de paramètres. Certains compléments permettent de pallier, au moins partiellement, cette limitation. La génération aléatoire contrainte de la SCV en est un exemple. Dans [OMAB06, Odd09], les auteurs présentent une technique de construction de modules matériels capables de générer aléatoirement des vecteurs de tests conformes à des spécifications formelles en PSL. Ces modules s'inspirent du même principe de construction modulaire introduit dans [BLOF06]. Avec la directive *assume* de PSL, la propriété est alors interprétée comme un comportement de l'environnement du système, et la description du générateur de vecteurs de tests est produite automatiquement. La solution proposée se restreint actuellement au niveau RTL.

Le test combinatoire fournit une autre alternative pour la génération de grandes quantités de données. En effectuant des combinaisons de valeurs définies par l'utilisateur, il permet d'obtenir directement des collections de tests qui, à la différence de celles générées aléatoirement, incluent inévitablement les combinaisons de valeurs spécifiquement identifiées. Les deux points critiques dans ce cas sont l'explosion combinatoire et la nécessité de concevoir des schémas de test (*patterns*) qui produiront un nombre suffisant de tests pertinents.

Comme pour le MBT, il existe plusieurs outils pour le test combinatoire. Par exemple, JMLUnit [CL02] est un outil qui combine un appel au constructeur d'une classe Java avec un appel à chaque méthode de la classe. Un ensemble de valeurs pertinentes doit être fourni en entrée de JMLUnit, cela pour chaque type utilisé dans la signature du constructeur ou dans celle des méthodes de la classe. Une combinaison des appels aux méthodes est calculée pour les valeurs ainsi définies.

6.2.2.2 Test combinatoire et ABV

L'outil TOBIAS [LDdB+07], développé au laboratoire LIG, permet de générer des séquences d'appels de méthodes en utilisant une approche combinatoire, potentiellement enrichie par un mécanisme de filtrage afin de contenir l'explosion combinatoire. Une grande quantité de jeux de test peut être obtenue automatiquement à partir de quelques lignes de descriptions combinatoires [dBLM+04, DCdBBL05]. Au travers d'une collaboration avec ses concepteurs, nous avons couplé cette solution d'aide à la génération de jeux de test avec notre solution de surveillance de propriétés temporelles [FPLdB08].

L'entrée attendue par TOBIAS est la description des motifs à déplier, aussi nommées groupes de test, ainsi que la description optionnelle des interfaces des classes à tester. L'outil génère un ensemble de jeux de tests, exprimés dans un format intermédiaire, à partir des motifs indiqués par l'utilisateur. Ces jeux de test *abstraits* peuvent ensuite être traduits vers différents langages cibles, comme Java. La figure 6.6 présente le flot de TOBIAS adapté à la génération de tests en C++/SystemC.

La saisie des motifs à déplier se fait à l'aide d'une interface graphique. Le résultat est un fichier source en format `.itb` contenant la représentation des groupes de test. Le fichier de sortie en format `.otb`, basé sur le standard XML et indépendant de toute technologie visée, est traduit par un composant dédié du flot en une série d'instructions SystemC exécutables, représentant les différents jeux de test. Cet ensemble d'instructions est prêt à être intégré dans le *testbench*. Par exemple, dans le cadre de l'application à la plate-forme DMA (voir section suivante), ces instructions sont insérées directement dans le corps du *thread* exécuté par le `dma_testbench` (le composant qui représente le CPU dans la plate-forme).

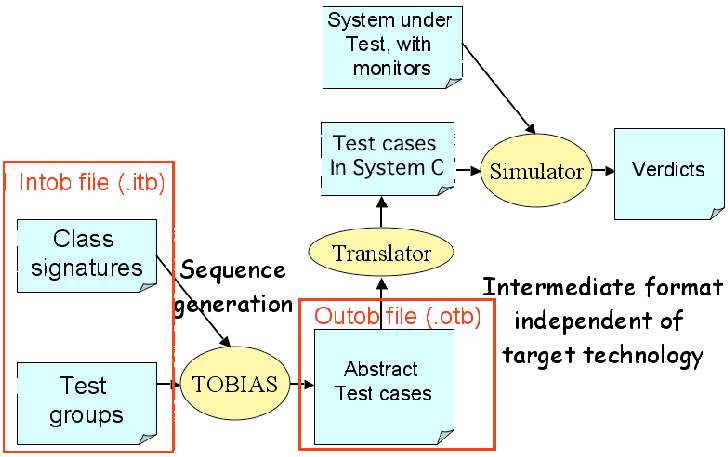

FIGURE 6.6 – Test combinatoire avec TOBIAS

Deux mécanismes ont été prévus pour pouvoir permettre à TOBIAS de maîtriser l'explosion combinatoire. Le premier est le mécanisme de filtrage. Il s'agit de l'expression de propriétés qui doivent être satisfaites par les jeux de test générés. Par exemple, il serait possible de contraindre la différence minimale entre deux paramètres. Ainsi, la sélection d'une valeur parmi celles indiquées par l'utilisateur pour le premier paramètre en exclura automatiquement certaines autres pour le deuxième. Le deuxième mécanisme est la sélection de sous-ensembles de jeux de test, suivant soit des critères spécifiques définis par l'utilisateur, soit des techniques de sélection aléatoire.

6.2.3 Application à l'exemple du DMA

L'infrastructure qui permet de dissocier la simulation de la surveillance des propriétés, présentée dans la section 6.2.1, et la méthode de test combinatoire à l'aide de TOBIAS ont été couplées profitablement dans certaines expérimentations menées avec la plate-forme DMA de la section 2.1.3.2. Son architecture est décrite par la figure 2.8 à la page 19. Ici nous rappelons brièvement son fonctionnement :

- le CPU, représenté par le composant `dma_testbench`, programme le DMA pour qu'il effectue des transferts entre les deux mémoires ; pour cela
  - il écrit la valeur de l'adresse source pour la copie dans le registre approprié du DMA : l'opération d'écriture, appelée sur le port initiateur du CPU, est acheminée par le router et atteint le DMA ;
  - de même, il écrit l'adresse destination et la longueur du transfert ;
  - enfin, il commande le début du transfert par écriture du *start* dans la registre de contrôle du DMA ;
- le DMA effectue la copie entre les mémoires en alternant lectures et écritures ;
- enfin, le DMA notifie la fin du transfert au CPU à l'aide d'une interruption (signal `dma_irq`).

Dans un premier temps, nous avons généré aléatoirement plusieurs valeurs pour les adresses et pour la longueur du transfert. Tous ces valeurs, bien qu'en grande quantité, étaient obtenues depuis des intervalles ne comportant que des cas nominaux, c'est-à-dire des cas avec des données usuelles et valides. Ceci a donné lieu à des tests n'impliquant que des comportements valides. Ensuite, nous avons considéré des cas limites où les valeurs pouvaient dépasser les paramètres attendus. À l'aide de TOBIAS, il a été possible d'obtenir

plusieurs combinaisons comportant ces cas limites, cela en spécifiant des ensembles succincts de valeurs.

Plusieurs groupes de test sont définis soit avec des opérations (ou des séquences d'opérations), soit avec des énumérations de valeurs possibles pour les paramètres. Par exemple, les deux groupes suivants comportent les valeurs choisies pour les adresses et pour la longueur du transfert :

```
AddrGr ::= {-10, 0, 0x20, 0x40, 0xFF, 0x100, 0x4000}
LenGr ::= {-64, 0, 1, 2, 64, 128}
```

Un troisième groupe utilise les deux premiers groupes de données pour décrire une séquence d'appels de méthodes :

```
WriteGr ::= {begin seq writeSrc(AddrGr);
                       writeDest(AddrGr);
                       writeLen(LenGr); end seq}
```

Cette séquence initialise l'adresse source, l'adresse destination et la longueur du transfert. TOBIAS utilise ces groupes de valeurs et d'opérations pour générer automatiquement tous les jeux de test qui correspondent à toutes les combinaisons possibles. Dans ce cas on obtient 294 possibilités pour le groupe `WriteGr` (7 adresses source $\times$ 7 adresses destination $\times$ 6 longueurs). Chaque opération du groupe est définie simplement pour qu'elle appelle l'une des méthodes du port initiateur du `dma_testbench`.

Nous traitons ici deux propriétés qui révèlent les deux dernières situations possibles durant une simulation, comme expliqué dans la section 6.2.1 : une erreur n'apparaissant pas sur la trace et une simulation corrompue.

Cas 1 La première propriété est proche de celle évoquée dans la section 6.1.2, néanmoins on ne se place plus du point de vue de l'application exécutée par le CPU, mais du point de vue du DMA lui-même. Nous considérons donc l'énoncé "chaque fois que le DMA voit son registre de contrôle programmé avec un *start*, il notifiera la fin de transfert avant de recevoir toute autre tentative de programmation", ce qui s'exprime comme suit :

```
vunit dma_prop1 {
  assert
  always ( (dma.write_CALL() &&
            dma.write.address == pv_dma::CONTROL &&
            dma.write.data == pv_dma::START)
           -> next (dma_irq before dma.write_CALL()) );
}
```

La surveillance de la propriété pour des cas nominaux montre que le DMA se comporte comme attendu, c'est-à-dire que l'interruption est toujours levée après complétion d'un transfert. Cependant, une combinaison particulière de données a décelé un comportement inattendu. Considérons le cas de figure suivant : l'application doit copier la même donnée deux fois, la première fois à l'adresse destination 0x40, qui se trouve dans la première mémoire, ensuite à l'adresse destination 0x100 qui se trouve dans la deuxième mémoire, mais, lors de la première programmation, la première adresse destination est initialisée par erreur avec la valeur 0x4000. Cette valeur s'avère être l'adresse du registre pour l'adresse source dans le DMA. Une séquence de test comportant le cas de figure décrit ci-dessus est la suivante :

1. écriture de l'adresse source 0x20 pour la copie,

2. écriture de l'adresse destination (erronée) 0x4000,
3. écriture de la longueur 1 pour le transfert (une seule donnée à copier),
4. écriture du bit de *start* 0x1 dans le registre de contrôle pour lancer le premier transfert, et attente de la notification de fin de transfert,
5. écriture de l'adresse destination 0x100 pour la deuxième copie (la même adresse source est gardée),
6. écriture de la longueur et démarrage du deuxième transfert.

Cette séquence donne lieu à la trace de simulation (sans moniteurs) suivante, en mode *debug*. En mode d'exécution normal, la trace comporterait beaucoup moins d'informations.

```
 1. DEBUG dma: write 0x20 in pv_dma source address register
 2. DEBUG dma: write 0x4000 in pv_dma dest address register
 3. DEBUG dma: write 0x1 in pv_dma length address register
 4. DEBUG dma: write 0x1 in pv_dma control register
 5. DEBUG dma: pv_dma started
 6. DEBUG dma: pv_dma transfer started. Source address: 0x20 -
               destination address: 0x4000 - length: 1
 7. DEBUG memory1: Read 0x8 at 0x20
 8. DEBUG dma: write 0x8 in pv_dma source address register
 9. DEBUG dma: rise transfer end IRQ
10. DEBUG dma: read pv_dma control register, returns 0x10
11. DEBUG dma: read pv_dma control register, returns 0x10
12. DEBUG dma: write 0 in pv_dma control register
13. DEBUG dma: clear transfer end IRQ
14. DEBUG dma: write 0x100 in pv_dma dest address register
15. DEBUG dma: write 0x1 in pv_dma length address register
16. DEBUG dma: write 0x1 in pv_dma control register
17. DEBUG dma: pv_dma started
18. DEBUG dma: pv_dma transfer started. Source address: 0x8 -
               destination address: 0x100 - length: 1
19. DEBUG memory1: Read 0x70 at 0x8
20. DEBUG memory2: Write 0x70 at 0
21. DEBUG dma: rise transfer end IRQ
22. DEBUG dma: read pv_dma control register, returns 0x10
23. ...
```

Les lignes 1 à 13 montrent les informations relatives au premier transfert. L'opération de lecture est effectuée proprement (ligne 7), mais l'opération d'écriture n'a pas l'effet désiré (ligne 8) : en employant le router comme s'il effectuait la copie, le DMA utilise l'adresse destination 0x4000 et il écrase la valeur contenue dans son propre registre pour l'adresse source. Il écrit dans ce registre la donnée 0x8 lue dans la première mémoire. Par conséquent, lors du deuxième transfert (après la ligne 14), l'adresse source 0x8 est utilisée au lieu de 0x20 (ligne 19), et la mauvaise donnée est copiée dans la deuxième mémoire (ligne 20). Ni le DMA ni le router n'ont été conçus pour éviter ce type de comportement. La simulation ne signale aucun avertissement ou erreur.

Durant la simulation instrumentée, le moniteur pour la propriété `dma_prop1` signale une violation dès que le registre pour l'adresse source est malencontreusement programmé, parce qu'aucune interruption (signal `dma_irq`) n'a été levée depuis la programmation du registre de contrôle :

```
 1. DEBUG dma: write 0x20 in pv_dma source address register
```

```
 2. DEBUG dma: write 0x4000 in pv_dma dest address register
 3. DEBUG dma: write 0x1 in pv_dma length address register
 4. DEBUG dma: write 0x1 in pv_dma control register
 5. DEBUG dma: pv_dma started
 6. DEBUG dma: pv_dma transfer started. Source address: 0x20 -
               destination address: 0x4000 - length: 1
 7. DEBUG memory1: Read 0x8 at 0x20
 8. DEBUG dma: write 0x8 in pv_dma source address register
    ===>  0 s: dma_prop1  Valid    = FALSE  <===
    ===>  0 s: dma_prop1  Checking = TRUE   <===
               dma___write_CALL = 1
               dma___write___address = 0
               dma___write___data = 0x8
               CTRL_REG = 0xc
               START = 0x1
               dma_irq = 0

    /!\  Property dma_prop1 VIOLATION  /!\  at 0 s

 9. DEBUG dma: rise transfer end IRQ
10. DEBUG dma: read pv_dma control register, returns 0x10
11. DEBUG dma: read pv_dma control register, returns 0x10
12. ...
```

Cas 2 La deuxième propriété permet de vérifier qu'il y a au moins une copie après chaque programmation du DMA. Plus précisément : "chaque fois qu'un nouveau transfert est démarré, *au moins* (opérateur fort) une opération de lecture est accomplie avant une écriture, et cette dernière se produit *au moins* une fois avant la notification de fin de transfert."

```
vunit dma_prop2 {
  assert
  always ( (dma.write_CALL() &&
            dma.write.p1 == pv_dma::CONTROL &&
            dma.write.p2 == pv_dma::START) ->
           (next! ( (m1.read_END() || m2.read_END())
                    before!
                    (m1.write_START() || m2.write_START()) ) &&
                  ( (m1.write_END() || m2.write_END())
                    before!
                    dma_irq ) ) );
}
```

Quand une valeur négative est utilisée pour la longueur du transfert, comme la valeur -64 du groupe `LenGr`, deux situations peuvent se produire. En cas de transfert mot-à-mot, où plusieurs lectures et écritures alternent, aucun transfert n'est effectivement accompli, mais la trace de simulation ne montre aucun avertissement. La copie semble commencer, mais la fin de transfert est notifiée juste après. Ce comportement, si incorporé dans une longue trace de simulation, sera difficilement identifiable. Dans ce cas, le moniteur pour la propriété `dma_prop2` signale une violation dès que l'interruption est levée, puisqu'aucune écriture n'a pas été observée auparavant.

En cas de transfert en mode *par bloc*, c'est-à-dire en copiant directement toute une portion de la mémoire, le processus de simulation est avorté avec une erreur de segmentation. L'opération de lecture n'est pas finie. Grâce à l'infrastructure proposée dans la section 6.2.1, le processus dédié à la surveillance de la propriété continue à s'exécuter et

le moniteur signale un avertissement à la “fin” de la simulation, en indiquant que la propriété est encore en attente (*Pending*). Un mode d’affichage verbeux du moniteur indique précisément les opérateur qui sont encore en attente, en montrant que le **next** et le premier **before** sont encore *Pending*, ce qui signifie que même l’opération de lecture en mémoire n’a pas été finie :

```
1. DEBUG dma: write 0x20 in pv_dma source address register
2. DEBUG dma: write 0x4000 in pv_dma dest address register
3. DEBUG dma: write 0x1 in pv_dma length address register
4. DEBUG dma: write 0x1 in pv_dma control register
5. DEBUG dma: pv_dma started
6. DEBUG dma: pv_dma transfer started. Source address: 0x20 -
              destination address: 0x4000 - length: -64
7. DEBUG memory1: Read block (0x3ffffff0) at 0x20
8. Simulation process: segmentation fault
   /!\  Property dma_prop2 still PENDING  /!\
        pending_next_0 = 1
        pending_before_0 = 1
        pending_before_1 = 0
```

Chapitre 7

Expérimentations

Introduction

Dans ce chapitre nous allons étudier les résultats de l'application de notre solution pour l'ABV dynamique. Outre la démonstration de la viabilité de l'approche dans des situations différentes, nous effectuerons une analyse de ses possibilités et une synthèse des résultats obtenus.

Les concepts développés tout au long des trois chapitres précédents ont été appliqués à un certain nombre de cas d'étude. Cet ensemble inclut des designs de tailles et de caractéristiques variées : designs qui mélangent canaux primitifs et hiérarchiques, sans ou avec notion de temps de simulation, sans ou avec horloge. Certains exemples n'utilisent pas de bibliothèque TLM pour SystemC.

7.1 Études de cas

Des expérimentations sur sept designs sont relatées par la suite. Pour chaque cas d'étude, quelques propriétés caractéristiques sont présentées et commentées. Les temps CPU pour les différentes simulations, sans et avec surveillance des propriétés, sont étudiés dans la section 7.2.

7.1.1 Communication par FIFO

7.1.1.1 Échanges simples

Le premier cas d'étude se base sur l'exemple présenté dans la section 2.1.1.5 (cf. figure 2.3 à la page 12). Il s'agit du design le plus élémentaire : un module producteur, équipé d'un `sc_thread`, génère des messages de caractères et il les écrit dans une FIFO de taille limitée (il s'agit du modèle de FIFO de la distribution SystemC) ; un module consommateur, lui aussi muni d'un `sc_thread`, lit les caractères depuis la FIFO. Le protocole d'échange prévoit les deux caractères spécifiques `'&'` et `'@'` pour marquer le début et la fin de chaque message. Le design n'utilise aucune horloge de synchronisation, mais il comporte des attentes temporelles, par exemple pour représenter le temps d'accès à la FIFO en lecture et en écriture. Ces deux opérations sont bloquantes. La bibliothèque TLM n'est pas utilisée dans cet exemple.

Nous considérons ici deux propriétés représentatives du type d'assertions pertinentes dans ce genre de modèles. La première n'utilise pas la couche modélisation (sauf pour la déclaration de deux constantes, ce qui améliore uniquement la lisibilité), elle emploie plusieurs opérateurs temporels et considère le paramètre de la communication. La deuxième propriété utilise un compteur dans la couche modélisation et elle surveille à la fois les lectures et les écritures.

$\boldsymbol{P_1}$: "L'envoi d'un nouveau message ne commence pas tant que l'envoi précédent n'est pas terminé". Nous décidons ici d'observer les actions du côté du producteur, afin d'observer si le protocole est respecté par celui-ci. Il est également possible d'observer les lectures, côté consommateur, mais une telle spécification mettrait aussi en jeu le fonctionnement de la FIFO.

```
vunit simple_fifo_protocol {
  // HDL_DECLs :
  char START_C = '&';
  char END_C = '@';
  // HDL_STMTs :
  // PROPERTY :
  assert
  always ( (fifo.write_CALL() && fifo.write.p1 == START_C) ->
           next((! (fifo.write_CALL() && fifo.write.p1 == START_C))
                until
                (fifo.write_CALL() && fifo.write.p1 == END_C)) );
}
```

$\boldsymbol{P_2}$: "Les opérations dans la FIFO respectent toujours sa taille", c'est-à-dire que la différence entre le nombre de lectures et le nombre d'écritures réussies (il s'agit ici de communications bloquantes) est toujours comprise entre 0 et `FIFO_SIZE`.

```
vunit simple_fifo_count {
  // HDL_DECLs :
  int FIFO_SIZE = 10;
  int char_count = 0;
  // HDL_STMTs :
  if( fifo.write_END() ) { char_count++; }
  if( fifo.read_END() ) { char_count--; }
  // PROPERTY :
  assert
  ( always ( 0 <= char_count && char_count <= FIFO_SIZE ) )
  @ ( fifo.write_END() || fifo.read_END() );
}
```

Dans les deux cas, pour les deux propriétés, le seul composant observé est la FIFO. Si le design est instrumenté uniquement avec la propriété P_1, alors la sous-classe observée de la FIFO surchargera uniquement la méthode `write`, comme décrit dans la section 4.2.2. Si le design est instrumenté avec les deux propriétés, les deux méthodes de communication seront observées dans la sous-classe finale. Dans ce cas, deux *wrappers*, chacun dédié à une propriété, surveillent le canal de communication. Lors d'une écriture, les deux *wrappers* sont notifiés et les deux propriétés sont évaluées. Lors d'une lecture, uniquement la deuxième propriété est évaluée.

7.1.1.2 Échanges avec arbitre

Une version légèrement plus élaborée que premier design est l'exemple présenté dans la section 4.2.4 (cf. figure 4.10 à la page 75). Le même modèle de FIFO est utilisé pour

les échanges, mais ce deuxième design comporte deux producteurs dont l'accès à la FIFO est accordé par un arbitre. Dans cet exemple, les quatre signaux $req1$, $req2$, $grant1$ et $grant2$ permettent les échanges entre l'arbitre et les deux producteurs : les deux premiers sont utilisés pour les requêtes d'envoi, les deux derniers pour accorder l'accès à la FIFO. Le protocole d'échange prévoit deux caractères spécifiques `'&'` et `'!'`, chacun associé à un producteur, pour marquer les débuts respectifs des messages, plus un caractère de fin de message `'@'` commun aux deux producteurs.

Deux propriétés sont considérées :

$\boldsymbol{P_3}$: "Les deux permissions d'écriture données par l'arbitre sont en exclusion mutuelle".

```
vunit arbitered_fifo_mutex {
  assert never ( grant1 && grant2 );
}
```

$\boldsymbol{P_4}$: "Lors de la prochaine écriture dans la FIFO, après permission de transmission accordée au premier producteur, c'est bien le caractère de début de message du producteur 1 qui sera écrit".

```
vunit arbitered_fifo_protocol {
  // HDL_DECLs :
  char P1_START_C = '&';
  // HDL_STMTs :
  // PROPERTY :
  assert
  always ( grant1 ->
           next_event(fifo.write_CALL())
                     (fifo.write.p1 == P1_START_C) );
}
```

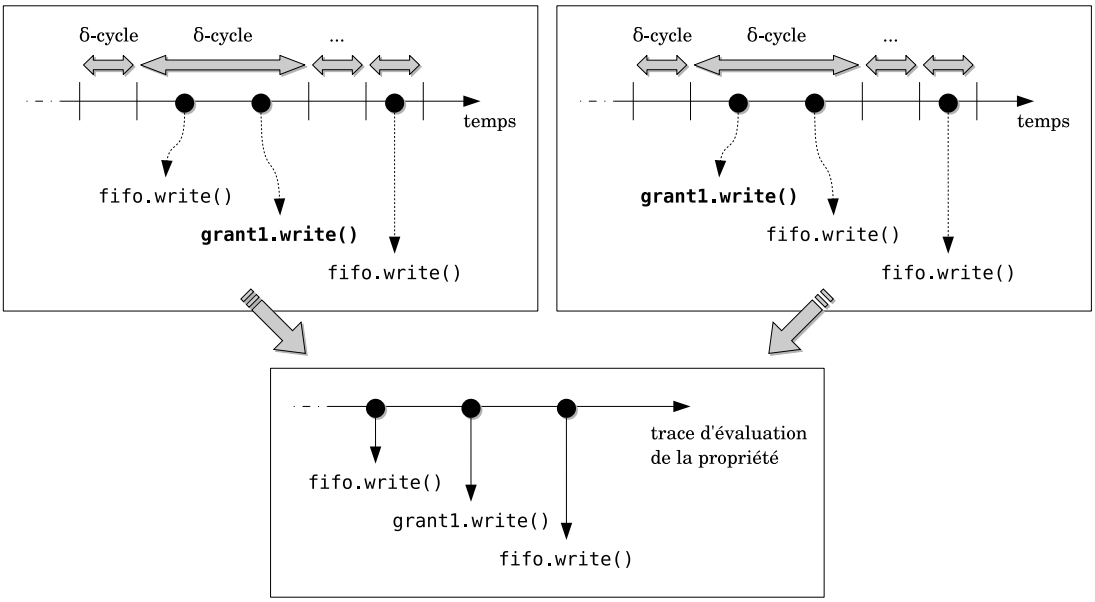

FIGURE 7.1 – Obtention de la trace d'évaluation de la propriété P_4

La propriété P_3 fait intervenir deux canaux primaires, dont la valeur est comparée au même instant. Comme expliqué dans la section 4.2.1, les canaux primaires sont une exception à l'ordonnancement purement séquentiel des actions de communication : deux

signaux écrits dans un même cycle delta changeront de valeur “en même temps” du point de vue du moniteur. Nous rappelons que l’emploi du nom d’un signal booléen dans le texte de l’assertion est une façon d’alléger l’expression de la propriété : cela est interprété par ISIS comme l’observation du passage à **true** de la valeur du signal.

La propriété P_4 implique un canal primaire ainsi que la FIFO. Dans ce cas, si le signal $grant1$ et la FIFO sont écrits dans le même cycle delta, dans un ordre quelconque, alors la trace d’évaluation de la propriété fera apparaître la communication par la FIFO avant celle via le signal, comme montré sur la figure 7.1.

7.1.2 Communication par canal défectueux

Le protocole de communication par canal défectueux [CM88], présenté dans la section 5.1.3, est similaire au premier design avec échanges entre un producteur et un consommateur via une FIFO. Toutefois, dans ce nouveau modèle, dont l’architecture de notre implémentation SystemC est montrée sur la figure 7.2, le canal utilisé peut perdre ou dupliquer les messages.

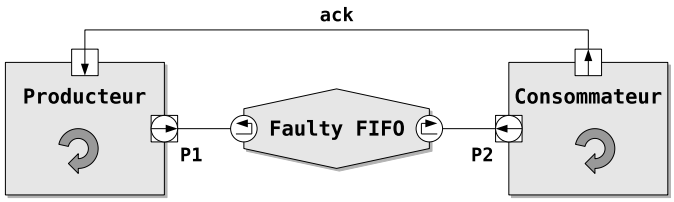

FIGURE 7.2 – Modèle de communication par canal défectueux

L’émetteur envoie une séquence de messages. Chaque message comporte un identifiant numérique et un contenu (ici de type caractère, pour simplifier). Le récepteur acquitte les messages reçus via le signal `ack`, chaque acquittement étant le numéro du dernier message reçu. Le récepteur affiche (fonction `print`) tous les messages qu’il considère comme devant être acceptés (messages “frais”). Comme dans [CM88], le canal, bien que défectueux, présente les caractéristiques suivantes :

- Tout message envoyé par ce canal peut être perdu, toutefois un même message n’est pas perdu indéfiniment ;
- Tout message envoyé par ce canal peut être dupliqué, toutefois un même message n’est pas dupliqué indéfiniment ;
- Les messages ne sont pas permutés, l’ordre est respecté ;
- Les messages ne sont pas altérés/corrompus.

Nous considérons deux assertions qui représentent les propriétés caractérisant la correction du protocole [CM88]. Dans les deux cas, l’énoncé PSL emploie une variable dans la couche modélisation.

$\boldsymbol{P_5}$: “Il est toujours vrai que $ack \leq nb_sent \leq ack + 1$, où nb_sent est le numéro du dernier message envoyé par l’émetteur”.

```
vunit fc_invariant_1 {
  // HDL_DECLs :
  int nb_sent = 0;
  // HDL_STMTs :
  if( fifo.write_CALL() )
```

```
    nb_sent = (fifo.write.p1).number;
  // PROPERTY :
  assert
  ( always (ack.read_END() ->
            (ack.read.p0 <= nb_sent && nb_sent <= (ack.read.p0+1))) )
  @ (fifo.write_CALL() || ack.read_END());
}
```

$\boldsymbol{P_6}$: “L’acquittement du récepteur est toujours égal à la longueur de la portion de texte original effectivement acceptée”. Cela signifie que, quand l’émetteur lit le numéro du dernier message m acquitté par le récepteur, ce dernier aura reçu tous les messages jusqu’à m.

```
vunit fc_invariant_2 {
  // HDL_DECLs :
  int length = 0;
  // HDL_STMTs :
  if( receiver.print_CALL() ) { length++; }
  // PROPERTY :
  assert
  ( always (ack.read_END() -> (ack.read.p0 == length)) )
  @ (ack.read_END() || receiver.print_CALL());
}
```

Par ailleurs, comme expliqué précédemment, bien que le canal soit défectueux, il garde ses caractéristiques de FIFO. Il est donc intéressant d’exprimer une contrainte sur l’ordre de réception des messages, traduite par la propriété P_f suivante. Dans P_f, une variable locale permet de mémoriser chaque nouveau message écrit dans le canal. Ce message doit être lu par le module récepteur avant le prochain.

$\boldsymbol{P_f}$: “Tout message émis par le producteur est reçu par le consommateur avant le prochain”.

```
vunit fc_is_fifo {
  assert
  always ( fifo.write_CALL() ->
           new(message x = fifo.write.p1)
              ( (fifo.read_END() &&
                 (fifo.read.p1).number == x.number)
                before
                (fifo.read_END() &&
                 (fifo.read.p1).number == x.number + 1) ) );
}
```

7.1.3 Contrôleur DMA

La plate-forme avec contrôleur DMA, distribuée avec la première version d’évaluation de la bibliothèque TLM 2.0, constitue un exemple représentatif des designs transactionnels fonctionnels sans notion de temps. Cette plate-forme a déjà été présentée dans la section 2.1.3.2 du chapitre 2 et reprise dans le chapitre précédent : nous nous limitons ici à rappeler son architecture à l’aide de la figure 7.3. Le CPU programme le DMA via le Router. Le DMA utilise ce même Router pour effectuer les transferts entre les mémoires. Le signal `dma_irq` permet au DMA de notifier la fin d’un transfert.

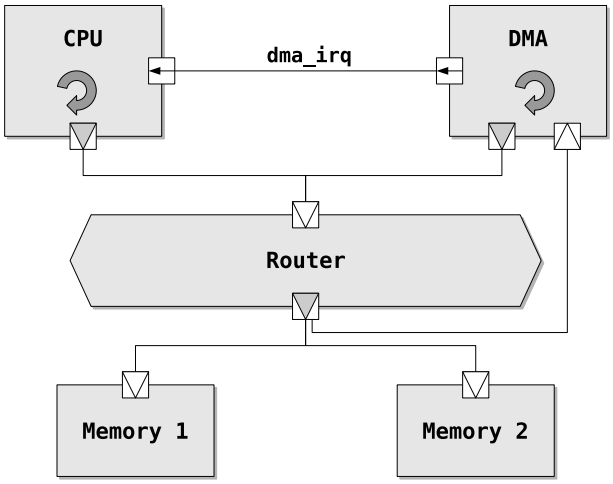

FIGURE 7.3 – Plate-forme avec contrôleur DMA

Les deux premières propriétés considérées portent sur le protocole de programmation du DMA et sur celui pour la copie. Elles expriment des relations d'ordre entre les communications du système. La troisième propriété porte également sur l'exactitude des données.

$\boldsymbol{P_7}$: "Chaque fois que le CPU demande un nouveau transfert au DMA (i.e. programmation du registre de contrôle du DMA avec la valeur *start*), une notification de fin de transfert se produit avant la prochaine modification du registre de contrôle du DMA". Nous rappelons que, selon le type de vérification visé, il est par exemple possible de choisir d'observer les opérations de programmation du DMA depuis le port initiateur du CPU ou directement depuis le DMA. Dans le premier cas, on surveillera le respect du protocole d'utilisation par le CPU. Dans le deuxième cas, on vérifiera toute programmation du DMA, pour n'importe quel composant à l'origine de cette programmation. Nous nous intéressons ici au premier cas.

```
vunit dma_prog_protocol {
  // HDL_DECLs :
  int CONTROL = 0x4000 + pv_dma::CONTROL;
  int START = pv_dma::START;
  // HDL_STMTs :
  // PROPERTY :
  assert
  always ( (cpu_i_port.write_CALL() &&
            cpu_i_port.write.p1 == CONTROL &&
            cpu_i_port.write.p2 == START) ->
            next ( dma_irq
                   before
                   (cpu_i_port.write_CALL() &&
                    cpu_i_port.write.p1 == CONTROL) ) );
}
```

$\boldsymbol{P_8}$: "Alternance entre lectures et écritures, après une lecture". Cette propriété exprime le fait qu'après chaque lecture dans la première mémoire, une écriture dans la deuxième mémoire est effectuée avant de lire de nouveau dans la première mémoire. Il s'agit d'une propriété vérifiée uniquement quand le CPU commande des transferts depuis la première vers la deuxième mémoire.

```
vunit dma_copy_protocol {
  assert
```

```
  always ( mem1.read_CALL() ->
           next ( mem2.write_CALL() before mem1.read_CALL() ) );
}
```

$\boldsymbol{P_9}$: "Chaque fois que le CPU programme le DMA pour qu'il effectue un transfert depuis une adresse source donnée dans la première mémoire, la prochaine lecture dans cette mémoire sera effectuée à l'adresse choisie". L'utilisation de la version forte de l'opérateur **next_event** permet également de s'assurer qu'une lecture dans la première mémoire aura inévitablement lieu.
Le bloc d'instructions de la couche modélisation est exécuté juste avant chaque évaluation de l'assertion. Si l'opération à l'origine de la notification du moniteur est une *écriture à destination du registre pour l'adresse source dans le DMA*, alors la valeur écrite (i.e. le deuxième paramètre `p2`) est mémorisée dans la variable auxiliaire `req_src_addr`. La prochaine fois qu'une lecture aura lieu en mémoire (opérateur **next_event**), la valeur de `req_src_addr` sera comparée avec celle effectivement utilisée pour la lecture.

```
vunit read_addr_ok {
  // HDL_DECLs :
  int SRC_REG = 0x4000 + pv_dma::SRC_ADDR;
  int M1_BOUND = 0x100;
  // Declaration de la variable auxiliaire :
  int req_src_addr;
  // HDL_STMTs :
  // Memorisation de la donnee ecrite dans le registre du DMA :
  if(cpu_i_port.write_CALL() && cpu_i_port.write.p1 == SRC_REG)
    req_src_addr = cpu_i_port.write.p2;
  // PROPERTY :
  assert
  always ( (cpu_i_port.write_CALL() &&
            cpu_i_port.write.p1 == SRC_REG &&
            cpu_i_port.write.p2 < M1_BOUND) ->
            next_event!(memory1.read_CALL())
                       (memory1.read.p1 == req_src_addr) );
}
```

La propriété P_7 implique à la fois un canal primaire (le signal d'interruption) et les opérations de communication via un port initiateur. La propriété P_9 utilise la couche modélisation pour mémoriser l'adresse source demandée par le CPU : une seule variable globale est suffisante dans cet exemple, puisque le DMA n'est pas multi-canal et il gère un seul transfert à la fois.

7.1.4 Packet Switch

Le *Packet Switch* de la distribution SystemC offre un exemple intéressant pour souligner l'utilité et la nécessité des variables locales. L'architecture, schématisée sur la figure 5.3 à la page 94, comporte quatre émetteurs de paquets et quatre récepteurs connectés au *switch*. Le design utilise deux horloges avec deux fréquences différentes, l'une pour le fonctionnement des émetteurs et l'autre pour le fonctionnement du *switch*. Ce dernier effectue une rotation de l'anneau de registres pour l'acheminement des paquets lors de chaque front montant de son horloge `switch_cntrl`. Cet aspect ne présente aucune impasse pour notre mécanisme de surveillance s'appuyant sur les opérations de communications. Au contraire, il est aussi possible d'utiliser l'horloge du *switch* pour mesurer le délai de sortie

d'un paquet, sans restreindre pour autant la surveillance de l'assertion uniquements aux fronts de cette horloge.

Quatre propriétés sont considérées. Bien que présentées ici pour un port de sortie donné, les trois dernières propriétés peuvent être réutilisées pour tout émetteur et pour tout récepteur connectés au *switch*.

$\boldsymbol{P_{10}}$: "Chaque paquet sortant du *switch* par la sortie out_i devait bien prendre cette destination".

```
vunit right_receiver {
  assert
  always ( (out0.write_CALL() -> (out0.write.p1).dest0) &&
           (out1.write_CALL() -> (out1.write.p1).dest1) &&
           (out2.write_CALL() -> (out2.write.p1).dest2) &&
           (out3.write_CALL() -> (out3.write.p1).dest3) );
}
```

$\boldsymbol{P_{11}}$: "Tout paquet en entrée du *switch* par un port X et à destination du port 0 sortira inévitablement par le port 0".

```
vunit pkt_delivery {
  assert
  always ( inX.write_CALL() && (inX.write.p1).dest0 ->
           new(pkt x = inX.write.p1)
              (eventually! (out0.write_CALL() &&
                            (out0.write.p1).id == x.id &&
                            (out0.write.p1).data == x.data)) );
}
```

$\boldsymbol{P_{12}}$: "Tout paquet en entrée par un port X et à destination du port 0 sortira par le port 0 avant N fronts montants de l'horloge de contrôle du *switch*". Afin de compter le nombre de fronts montants de l'horloge du *switch*, cette propriété peut utiliser soit un compteur local, comme expliqué dans la section 5.2.2, soit un seul compteur global, comme nous l'avons fait dans [PF10]. Ici nous considérons la deuxième version.
Si un paquet en entrée du *switch* est à destination du port 0, il est alors mémorisé dans la variable locale x, et une nouvelle instance de la sous-formule sous la portée du **new** est activée. Cette instance comporte aussi la constante `MAX`, égale au nombre courant de fronts montants de l'horloge plus `N` (le retard maximal choisi). Le message x devra alors sortir par 0 avant que le nombre de fronts montants de l'horloge n'ait atteint `MAX`.

```
vunit pkt_delivery_time_limit {
  // HDL_DECLs :
  unsigned int N = 20;
  unsigned int step = 0;
  // HDL_STMTs :
  if( switch_cntrl ) { step++; }
  // PROPERTY :
  assert
  always ( inX.write_CALL() && (inX.write.p1).dest0 ->
           new(pkt x = inX.write.p1 , unsigned int MAX = step+N)
              ( (out0.write_CALL() &&
                 (out0.write.p1).id == x.id &&
                 (out0.write.p1).data == x.data)
                before!
                (switch_cntrl && step == MAX) ) );
```

```
}
```

P_{13} : “Deux paquets écrits successivement sur le même port d’entrée X et à destination du port 3 seront reçus dans le même ordre”. Chaque fois qu’un paquet en entrée du *switch* est à destination du port 3, il est alors mémorisé dans une variable locale x. Le paquet suivant à destination de 3 est mémorisé dans une deuxième variable locale y. Le premier paquet x devra alors sortir par le port 3 avant le deuxième paquet y (opérateur **before** dans l’assertion). Toutefois, si le paquet x sort par le port 3 avant qu’un deuxième paquet à destination de 3 n’accède au *switch*, la vérification de la sous-formule sous la portée du premier **new** doit être arrêtée : cette condition est exprimée à l’aide de l’opérateur **abort** dans la propriété.

```
vunit order_preservation {
  assert
  always (
    next_event(inX.write_CALL() && (inX.write.p1).dest3)
      // Memorisation du premier paquet dans "x" :
      (new(pkt x = inX.write.p1)
          ( (next (next_event(inX.write_CALL() && (inX.write.p1).dest3)
              // Memorisation du paquet suivant dans "y" :
              (new(pkt y = inX.write.p1)
                  ( out3.write_CALL() &&
                    (out3.write.p1).data == x.data &&
                    (out3.write.p1).id == x.id
                    before  // le paquet "x" doit sortir avant "y"
                    out3.write_CALL() &&
                    (out3.write.p1).data == y.data &&
                    (out3.write.p1).id == y.id ))))
            abort (out3.write_CALL() &&
                   (out3.write.p1).data == x.data &&
                   (out3.write.p1).id == x.id) ) ) );
}
```

La propriété P_{10} est connectée une seule fois à tous les signaux qui transportent les paquets en sortie du *switch*. Les trois autres propriétés peuvent être connectées plusieurs fois. Par exemple, P_{11} peut être vérifiée pour les paquets en entrée par $\texttt{IN}_1$ et à destination de $\texttt{OUT}_0$, ainsi que pour les paquets en entrée par $\texttt{IN}_3$ et à destination de $\texttt{OUT}_0$.

La propriété P_{13} ne s’intéresse pas à la vérification de la bonne réception des paquets, mais uniquement à la préservation de l’ordre. Par conséquent, l’emploi d’opérateurs forts n’est pas pertinent. La propriété utilise deux opérateurs **new** imbriqués : chaque instance de la sous-formule associée au **new** le plus interne est créée depuis une instance existante de la formule relative au **new** externe, comme montré sur la figure 7.4. Encore une fois, la nature modulaire de nos moniteurs permet d’implémenter relativement aisément ce comportement, dont la sémantique est caractérisée dans la section 5.2.

La surveillance des propriétés P_{11} et P_{12} nous a conduits à détecter une erreur de conception : même en cas de faible affluence de paquets en entrée du *switch*, ceux à destination de la sortie 0 sont parfois perdus. Dans ce cas, les instances du moniteur pour la formule sous la portée du **new** dans P_{11} signalent un avertissement à la fin de la simulation, en indiquant que l’état de la formule est encore *Pending*. Le problème d’une assertion avec l’opérateur **eventually!** est que sa sémantique induit qu’il ne peut jamais être violé durant la simulation. Si la condition du **eventually!** ne se produit jamais, la seule information intéressante affichée par le moniteur sera son état *en attente* à la fin de la simulation. C’est pourquoi nous nous sommes orientés vers une deuxième version de

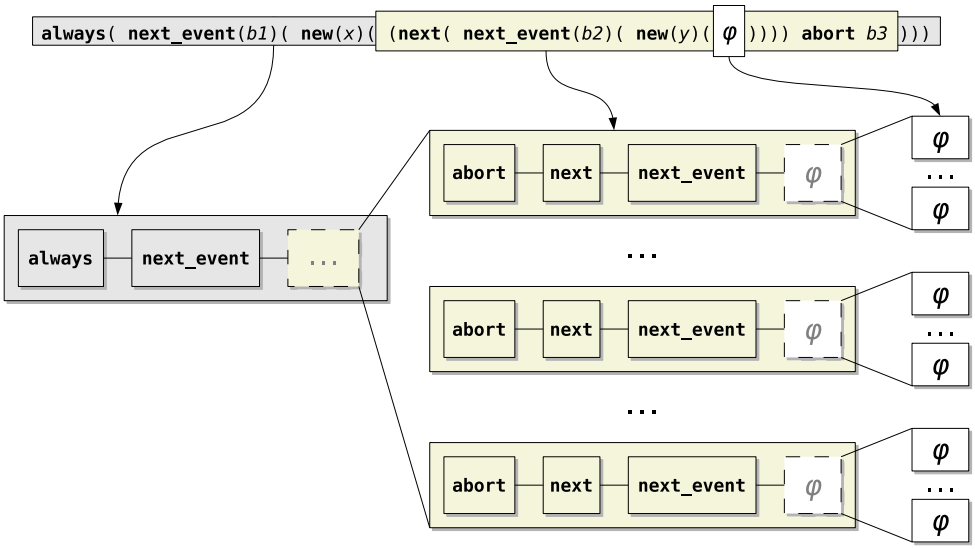

FIGURE 7.4 – Utilisations imbriquées du *new* et sous-instances multiples

la propriété P_{11} qui utilise un **before!** fort : dans cette version, l'écriture du paquet sur le port 0 doit se produire inévitablement avant écoulement du temps limite. Ainsi, il est possible d'observer des violations effectives : les instances dans P_{12} signalent une violation juste après écoulement du temps limite, si le paquet n'est pas sorti. Un extrait d'une trace de simulation commentée prsentant cette caractéristique est disponible dans l'annexe B.

Les auteurs de [KEP06] avaient également mentionné la violation d'une propriété relative à la bonne réception de tous les messages. Cependant, ils avaient justifié la violation comme une conséquence de la sémantique des signaux : si plusieurs paquets identiques sont envoyés consécutivement par un même émetteur, seulement le premier paquet sera effectivement transmis, car l'écriture des autres paquets n'entraînera pas la phase de mise-à-jour du signal qui lie le port de l'émetteur avec le port d'entrée du *switch*. Bien que correcte, cette explication avait masqué la découverte d'une vraie erreur dans l'implémentation du *switch* : tout paquet qui accède à l'anneau de registres et qui est à destination de plusieurs sorties, dont 0, est perdu avant d'être livré à la sortie 0. Le paquet est effacé par erreur lorsqu'il est délivré à l'avant-dernier destinataire.

7.1.5 Plate-forme de décodage Motion-JPEG

Certaines de nos expérimentations relatives aux variables globales de la couche modélisation ont été appliquées à une plate-forme de décodage Motion-JPEG décrite au niveau TA (*Transaction Accurate*) [GGP08], schématisée sur la figure 7.5. Parmi ses composants, la plate-forme comporte un nombre variable de processeurs, représentés par les unités d'exécution (EU, *Execution Unit*) et une mémoire globale. Nous n'avons ici utilisé qu'un seul processeur. L'application de décodage MJPEG lit les données à partir du générateur de trafic TG, effectue le décodage et remet en forme les données décodées avant leur envoi au composant matériel RAMDAC qui permet l'affichage [GGP08, Ger09].

Les deux propriétés suivantes surveillent le bon acheminement des données à destination du RAMDAC. La première vérifie l'absence de perte de données transmises via le Crossbar. La deuxième vérifie l'intégrité de ces données. Nous rappelons que ces deux propriétés

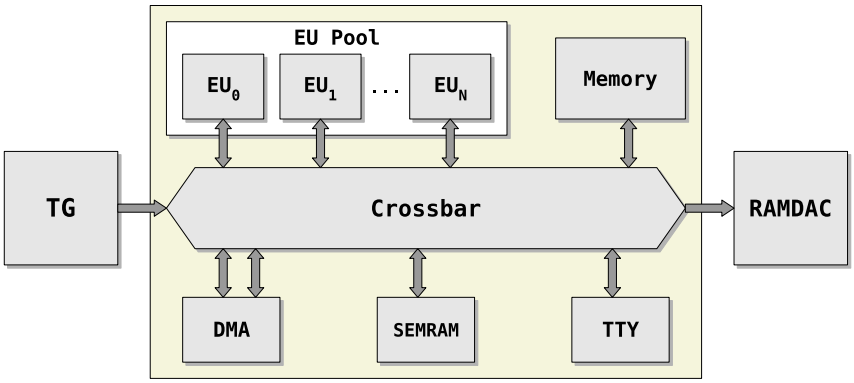

FIGURE 7.5 – Plate-forme de décodage Motion-JPEG

considèrent un seul processeur qui effectue les requêtes d'écriture dans le Crossbar.

$\boldsymbol{P_{14}}$: "Le nombre d'écritures reçues par le RAMDAC est égal au nombre de requêtes d'écriture du processeur". Dans cette assertion, deux compteurs globaux permettent de mémoriser le nombre d'écritures demandées par le processeur (`req_count`) et le nombre d'écritures effectivement reçues par le RAMDAC (`write_count`). Chaque fois que le RAMDAC est écrit, ces deux compteurs doivent être égaux. Puisque la fonction booléenne indiquant l'occurrence de `eu.write()` n'apparaît que dans la couche modélisation, nous utilisons l'opérateur `@` pour indiquer que l'évaluation de la propriété se fait aussi en présence de cette opération de communication.

```
vunit lossless_transmission {
  // HDL_DECLs :
  unsigned int req_count = 0;
  unsigned int write_count = 0;
  // HDL_STMTs :
  if( eu.write_CALL() ) { req_count++; }
  if( rdac.write_CALL() ) { write_count++; }
  // PROPERTY :
  assert
  ( always (rdac.write_CALL() -> (req_count == write_count))
  @ (eu.write_CALL() || rdac.write_CALL());
}
```

$\boldsymbol{P_{15}}$: "Les données écrites dans le RAMDAC sont exactement celles émises par le processeur". Chaque fois que le processeur effectue une requête d'écriture, la donnée écrite (paramètre `p2` de la méthode `write`) est mémorisée dans la variable auxiliaire `req_data`. La prochaine fois que le RAMDAC sera écrit, la donnée écrite sera comparée avec celle mémorisée dans `req_data`. Une seule variable globale suffit dans ce cas pour mémoriser les données transmises par le processeur, puisqu'il n'y a qu'un seul processeur dans la plate-forme et puisque les communications via le Crossbar ne sont pas en pipeline.

```
vunit noise_absence {
  // HDL_DECLs :
  unsigned int req_data;
  // HDL_STMTs :
  // Memorisation de la donnee ecrite par le processeur :
  if( eu.write_CALL() )
    req_data = eu.write.p2;
  // PROPERTY :
```

```
    assert
    always ( eu.write_CALL() ->
             next_event(rdac.write_CALL())(req_data == rdac.write.p2) );
}
```

La plate-forme utilise une bibliothèque de composants développée dans le cadre de [Ger09] et qui inclut le Crossbar. Pour nos expérimentations, nous avons modifié l'implémentation de ce dernier afin qu'il altère aléatoirement certaines des données transmises au RAMDAC[1]. Les images affichées sont alors corrompues et le moniteur reporte chaque violation de P_{15} lorsqu'une donnée écrite dans le RAMDAC ne correspond pas à celle attendue, en indiquant le temps de simulation et les valeurs observées (paramètres et variables), comme montré sur la figure 7.6.

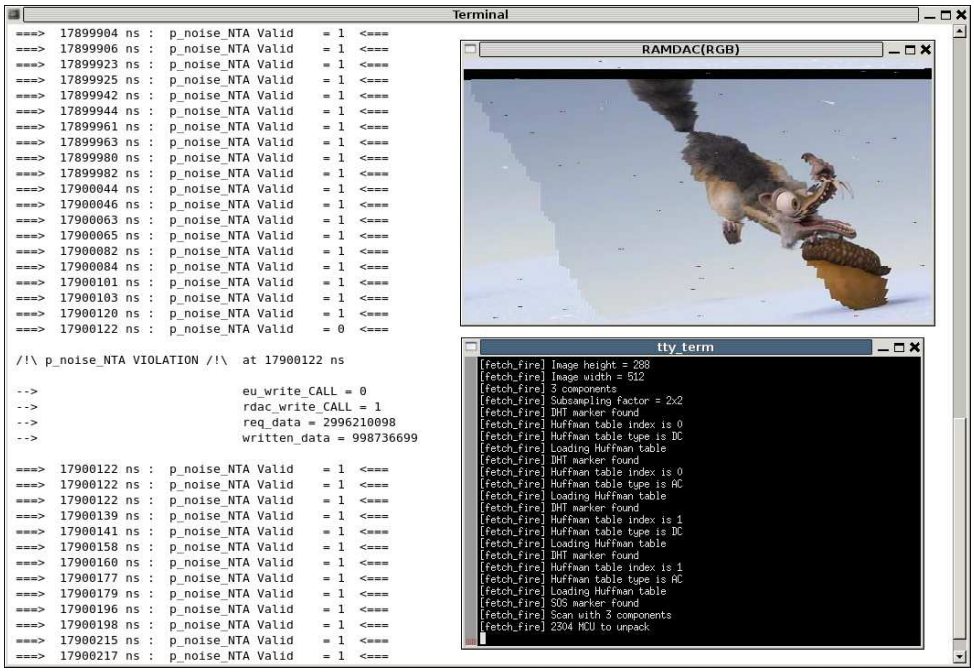

FIGURE 7.6 – Simulation instrumentée de la plate-forme MJPEG comportant une version défectueuse du Crossbar

7.1.6 Modèle transactionnel d'un SoC de traitement de données bord pour charge utile spatiale

Ce dernier exemple est un design développé par Astrium comme cas d'étude dans le domaine Guidage/Navigation/Contrôle. Il s'agit d'un algorithme de traitement d'image dans le cadre de l'identification et du suivi d'objets avec comme objectif principal la compression des données instrument (charge utile satellite) afin d'être compatible avec des contraintes de débit sur le lien de télémesure bord-sol. L'architecture montrée sur la figure 7.7 est l'une des architectures possibles étudiées par Astrium afin d'identifier les solutions optimales par rapport à différents ensembles de paramètres dans l'algorithme.

[1] La version originale des composants de la bibliothèque ne comporte aucune des erreurs que nous avons volontairement introduites.

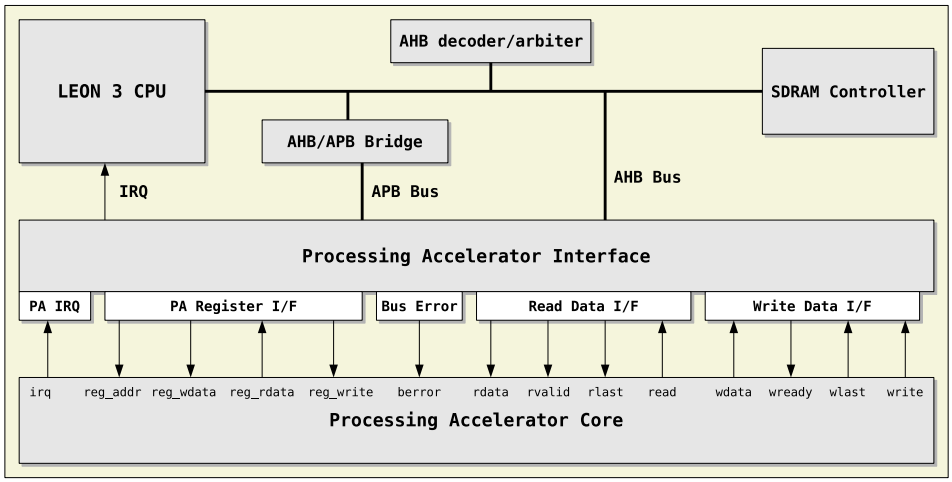

FIGURE 7.7 – Diagramme de l'architecture du SoC

Nos expérimentations avec ISIS ont été effectuées sur une implémentation SystemC simplifiée, cependant représentative, de cette architecture[2]. Ces résultats seront publiés dans [FPA+11]. La plate-forme, montrée sur la figure 7.8, modélise le comportement fonctionnel ainsi que les contraintes temporelles, grâce à la mise en œuvre d'une technologie permettant de séparer les aspects temporels des des aspects fonctionnels en TLM [Cor08]. Dans cette modélisation, où TTP est acronyme de *Timed TLM Protocol*, les composants de type "PV" représentent la fonctionnalité et les composants "T" ajoutent la couche temporelle dans les communications.

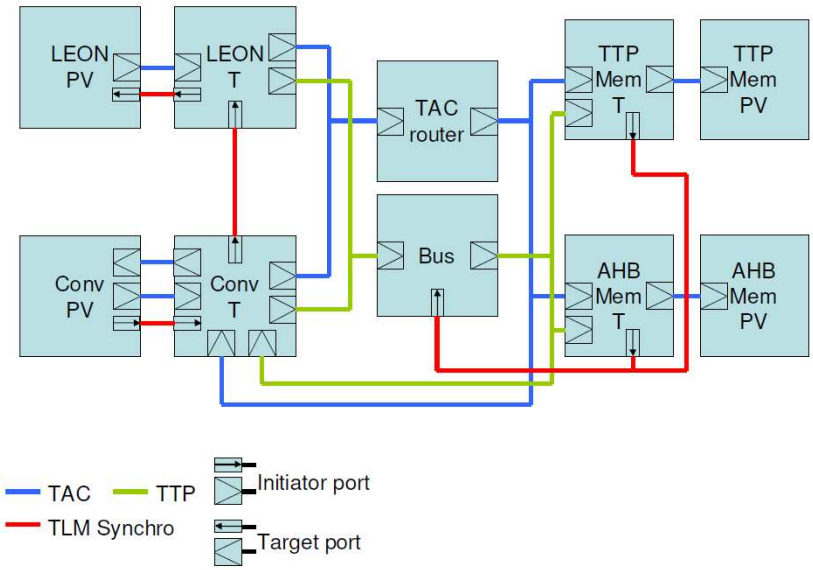

FIGURE 7.8 – Plate-forme SystemC TLM fonctionnelle avec modélisation des contraintes temporelles

Le "Leon PV" est un pseudo-processeur qui émule le comportement du logiciel. Deux canaux de communication coexistent : le Router achemine chaque transaction vers l'un des composants esclaves ; en parallèle, une transaction est aussi envoyée à travers le Bus,

[2] L'implémentation SystemC de la plate-forme a également été effectuée par Astrium.

qui présente des caractéristiques de construction telles que la latence ou l'arbitrage. Le composant "Conv" est également équipé d'une couche temporelle. Il s'agit d'un bloc de convolution capable de récupérer et enregistrer des images en mémoire, et d'effectuer des calculs de convolution sur ces images. Il peut être programmé par le Leon lorsque des calculs de convolution sont requis.

Plusieurs catégories de propriétés peuvent être considérées pour ce type de plate-forme. Les trois propriétés qui suivent, identifiées par les auteurs de la plate-forme, représentent chacune une catégorie différente. La première est relative au fonctionnement correct du logiciel vis-à-vis du matériel, par conséquent elle exprime un besoin similaire à celui spécifié par la propriété P_7 du DMA. La deuxième peut être vue comme une contrainte de codage liée à la qualité du design ou aux besoins du matériel. La troisième porte sur les contraintes liées à l'architecture et sur leur respect par le logiciel.

$\boldsymbol{P_{16}}$: "Le processeur ne commence pas un nouveau calcul de convolution avant la terminaison du précédent". Comme pour les autres propriétés, il est impératif d'identifier la signification des termes "début" et "fin" d'un calcul de convolution dans la plate-forme. Le *début* d'un calcul est indiqué par l'*écriture dans le registre pour l'adresse de lecture* de l'unité de convolution (`A_READ_ADDR`). La *fin* d'un calcul correspond à la *lecture d'une valeur égale à la taille de l'image* (`IMAGE_SIZE`) dans le registre pour la longueur (`A_WRITE_LENGTH`). L'assertion PSL est donc une traduction du texte suivant : "chaque fois que le processeur démarre un nouveau calcul de convolution, il attendra la fin du calcul avant d'en demander un autre". Cette assertion est valide uniquement quand le drapeau `PVT_NO_CHECK`, qui représente un mode de debug particulier, est désactivé. La couche modélisation est ici utilisée uniquement pour déclarer des constantes. La structure de la propriété PSL est assez simple : suite à chaque "début" de calcul, l'opérateur **before** permet d'exprimer le fait que la "fin" du calcul doit se produire avant le prochain "début".

```
vunit wait_for_completion {
  // HDL_DECLs :
  unsigned int IMAGE_SIZE = 8*8;
  unsigned int HW_CONV_ADDRESS = 0x20000000;
  unsigned int A_READ_ADDR =
      HW_CONV_ADDRESS + ttp_validation::conv_reg::a_read_addr;
  unsigned int A_WRITE_LENGTH =
      HW_CONV_ADDRESS + ttp_validation::conv_reg::a_write_length;
  unsigned int NO_CHECK = prt_tlm_tac::PVT_NO_CHECK;
  // HDL_STMTs :
  // PROPERTY :
  assert
  always ( (leonPV_i_port.write_CALL() &&
            leonPV_i_port.write.p1 == A_READ_ADDR &&
            leonPV_i_port.write.p5 != NO_CHECK)
          -> next ( (leonPV_i_port.read_END() &&
                     leonPV_i_port.read.p1 == A_WRITE_LENGTH &&
                     leonPV_i_port.read.p2 == IMAGE_SIZE &&
                     leonPV_i_port.read.p5 != NO_CHECK)
                    before
                    (leonPV_i_port.write_CALL() &&
                     leonPV_i_port.write.p1 == A_READ_ADDR &&
                     leonPV_i_port.write.p5 != NO_CHECK) ) );
}
```

$\boldsymbol{P_{17}}$: “L’adresse destination et l’adresse source en mémoire pour le calcul de convolution doivent être transmises à l’unité de convolution avant de démarrer un calcul”. Dans la plate-forme, la transmission de l’adresse source (`A_READ_ADDR`) à l’unité de convolution indique le début du calcul, comme mentionné pour P_{16}. Par conséquent, elle doit être transmise après celle de destination (`A_WRITE_ADDR`). L’assertion est en réalité une conjonction de deux sous-propriétés : la première couvre le tout premier calcul, la deuxième caractérise tous les calculs suivants puisque chaque nouveau calcul est identifié par la “fin” du précédent.

```
vunit good_conv_programming {
  // HDL_DECLs :
  unsigned int IMAGE_SIZE = 8*8;
  unsigned int HW_CONV_ADDRESS = 0x20000000;
  unsigned int A_READ_ADDR =
      HW_CONV_ADDRESS + ttp_validation::conv_reg::a_read_addr;
  unsigned int A_WRITE_ADDR =
      HW_CONV_ADDRESS + ttp_validation::conv_reg::a_write_addr;
  unsigned int A_WRITE_LENGTH =
      HW_CONV_ADDRESS + ttp_validation::conv_reg::a_write_length;
  unsigned int NO_CHECK = prt_tlm_tac::PVT_NO_CHECK;
  // HDL_STMTs :
  // PROPERTY :
  assert
  // Premier calcul :
  ( (leonPV_i_port.write.p1 == A_WRITE_ADDR &&
     leonPV_i_port.write.p5 != NO_CHECK)
    before
    (leonPV_i_port.write_CALL() &&
     leonPV_i_port.write.p1 == A_READ_ADDR &&
     leonPV_i_port.write.p5 != NO_CHECK) )
  &&
  // Deuxieme calcul et suivants :
  ( always ( (leonPV_i_port.read_END() &&
              leonPV_i_port.read.p1 == A_WRITE_LENGTH &&
              leonPV_i_port.read.p2 == IMAGE_SIZE &&
              leonPV_i_port.read.p5 != NO_CHECK)
             -> next ( (leonPV_i_port.write_CALL() &&
                        leonPV_i_port.write.p1 == A_WRITE_ADDR &&
                        leonPV_i_port.write.p5 != NO_CHECK)
                       before
                       (leonPV_i_port.write_CALL() &&
                        leonPV_i_port.write.p1 == A_READ_ADDR &&
                        leonPV_i_port.write.p5 != NO_CHECK) ) ) );
}
```

$\boldsymbol{P_{18}}$: “La mémoire AHB n’émet pas deux *splits*[3] consécutifs pour le même maître”, ce qui signifie que si un maître fait un accès *split*, alors son prochain accès sera forcément sans *split*. Ici, la couche modélisation est nécessaire pour mémoriser l’identifiant du dernier maître ayant accédé à la mémoire. Dans le chapitre précédent nous avons souligné le rôle crucial de l’utilisateur dans le choix des communications observées. Cette propriétés en est un exemple typique, car il est possible de choisir de surveiller les opérations soit par le port initiateur du bus AHB, soit directement dans la mémoire AHB. La version qui suit couvre le premier cas (P_{18} v1).

[3] Découpage d’une opération de copie pour qu’elle soit reprise plus tard.

```
vunit no_successive_splits_v1 {
  // HDL_DECLs :
  // ID du master qui a fait l'acces split precedent :
  unsigned int prev_master;
  // ID du master qui fait l'acces split :
  unsigned int master = 999;
  // Pour savoir si la cible de la communication est la
  //  memoire AHB :
  bool to_ahb = false;
  // Pour savoir si le mode est split :
  bool is_split = false;
  prt_tlm_ttp::ttp_response<ttp_ahb::ahb_status> resp;
  prt_tlm_ttp::ttp_status<ttp_ahb::ahb_status> status;
  // HDL_STMTs :
  // Par defaut, on suppose que le target n'est pas la memoire
  //  AHB et que l'acces n'est pas split :
  to_ahb = false;
  is_split = false;
  if( bus_i_port.do_transport_END() ) {
    // la memoire AHB est le target 0 :
    to_ahb = (bus_i_port.do_transport.p3 == 0);
    if( to_ahb ) {
      // Si l'operation de communication est adressee a la
      //  memoire AHB, alors il faut sauvegarder l'ID du
      //  maitre precedent et memoriser l'ID du maitre qui
      //  demande l'acces actuel :
      prev_master = master;
      master = (bus_i_port.do_transport.p1).get_master_id();
      resp = bus_i_port.do_transport.p2;
      status = resp.get_ttp_status();
      // Le mode "split" est connu grace a la methode "is_split"
      //  de l'etat relatif a la reponse de la communication
      //  (il s'agit de la reponse donnee par la memoire,
      //  passible d'accorder un acces split) :
      is_split = (status.access_extension())->is_split();
    }
  }
  // PROPERTY :
  assert
  // Chaque fois que nous avons un acces split, le prochain acces
  //  par le meme master (condition "master == prev_master" dans
  //  le "next_event") ne sera pas split :
  always ( (bus_i_port.do_transport_END() && to_ahb && is_split)
           -> next ( next_event( bus_i_port.do_transport_END() &&
                                 to_ahb && (master == prev_master))
                               (! is_split) ) );
}
```

La deuxième version (P_{18} v2) surveille les communications directement dans la mémoire AHB, par conséquent il n'est plus indispensable d'identifier les opérations qui lui sont adressées. Cette deuxième version est donc presque identique à la précédente : la seule différence dans son énoncé est que la couche modélisation ne comporte plus la variable booléenne `to_ahb` et elle ne contient plus le test relatif au *target* de la communication par le bus.

```
vunit no_successive_splits_v2 {
  // HDL_DECLs :
```

```
    unsigned int prev_master;
    unsigned int master = 999;
    prt_tlm_ttp::ttp_response<ttp_ahb::ahb_status> resp;
    prt_tlm_ttp::ttp_status<ttp_ahb::ahb_status> status;
    bool is_split = false; // vrai en cas d'acces split
    // HDL_STMTs :
    is_split = false;
    if( ahb_mem.do_transport_END() ) {
      prev_master = master;
      master = (ahb_mem.do_transport.p1).get_master_id();
      resp = ahb_mem.do_transport.p0;
      status = resp.get_ttp_status();
      is_split = (status.access_extension())->is_split();
    }
    // PROPERTY :
    assert
    always ( (ahb_mem.do_transport_END() && is_split)
             -> next ( next_event( ahb_mem.do_transport_END() &&
                                   (master == prev_master) )
                                 (! is_split) ) );
}
```

Il est à noter que, les moniteurs étant appelés à être combinés au design, nous pouvons tout simplement, dans la couche modélisation notamment, utiliser des fonctions d'accès (*getters*) et d'autres fonctions des classes de la plate-forme pour calculer les valeurs des variables auxiliaires. C'est le cas, par exemple, pour `get_ttp_status()` qui est une méthode de la classe `ttp_response`.

Quand il s'agit de surveiller des communications dans une plate-forme TLM plus évoluée qu'un simple modèle de communication par FIFO, il faut apporter un soin tout particulier à la formalisation de la spécification, c'est-à-dire, entre autres, à la traduction adéquate des conditions énoncées dans le texte en langage courant. Dans l'exemple de la section 7.1.1.1, l'envoi d'un message par le producteur correspond à une simple action d'écriture dans la FIFO. De même, l'identification du début d'une transmission équivaut à observer l'écriture d'un caractère spécial. Le contexte des trois derniers cas d'étude est bien plus évolué. Ce dernier cas d'étude, en particulier, montre la nécessité "d'explorer" le contenu de chaque communication pour pouvoir exprimer convenablement chacune des conditions. Cela s'obtient essentiellement par l'accès aux paramètres et aux valeurs de retour des méthodes. Il s'agit d'un aspect lié au code de l'implémentation de la plate-forme et indépendant de l'outil de vérification utilisé. Il s'agit aussi d'un aspect inévitable, mais qui pourrait être automatisé, au moins en partie. L'un des intérêts de la bibliothèque transactionnelle, outre celui lié à l'abstraction, est l'adoption d'un standard de communication qui rende les modèles le plus possibles réutilisables et *inter-changeables*. Dans ce contexte, l'identification d'un ensemble de conditions récurrentes sur ces communications, comme l'accès en mode *split*, pourrait être regroupée dans une bibliothèque d'aide à la spécification. Ces conditions pourraient être utilisées directement dans les assertions. Par exemple, il serait possible d'écrire directement `always (x.split_access -> ...)` : la condition `split_access` serait alors interprétée automatiquement, à l'aide de la bibliothèque, comme

- l'observation d'une méthode de communication standard dans `x`,
- l'obtention de la valeur de retour de la méthode, et la vérification du mode *split*

(méthode `is_split()` de l'extension de `ttp_status` dans l'exemple).

Bien évidemment, selon le type de propriété, d'autres éléments devraient être pris en compte, comme la mémorisation de l'identifiant du maître ayant demandé un accès. Dans notre exemple, cela se réalise grâce à la couche modélisation.

Le choix des méthodes observées est une étape autant essentielle que délicate. Les deux versions exprimées pour la propriété P_{18} diffèrent uniquement dans ce choix, mais les deux sont également pertinentes. L'utilisateur peut choisir l'une ou l'autre en fonction de sa confiance dans le fonctionnement des composants, et il peut également garder les deux. Bien que notre technique de surveillance permette d'observer toutes les méthodes d'une classe, les développeurs de cette plate-forme pour la télémesure ont souhaité que les assertions soient formalisées en ne se référant qu'aux méthodes de communication. Si une méthode correspond à une action interne à un composant, comme l'opération de décodage d'un bus ou encore l'exécution du calcul de convolution, il est discutable de l'utiliser dans l'assertion. Son existence est éphémère et liée uniquement aux choix de codage du programmeur. Au contraire, une opération de communication entre composants est un élément stable, qui s'appuie sur un standard (ou sur une convention imposée par la technologie adoptée). Par conséquent, une méthode d'une interface de communication offre un point d'observation idéal dans une plate-forme transactionnelle.

Pour des raisons proches de celles expliquées ci-dessus, même dans la couche modélisation, l'emploi de méthodes autres que celles de communication doit être réfléchi. Bien qu'utiles à rendre l'écriture de l'assertion plus concise, ces méthodes peuvent introduire dans l'assertion les mêmes erreurs de codage que celles qui peuvent être commises par le programmeur. Par exemple, si la fonction de décodage du bus est erronée, son utilisation directe dans l'assertion fournirait de même un résultat erroné et elle rendrait l'assertion inutile. Au contraire, l'intérêt d'un moniteur de surveillance est celui d'avoir une *deuxième* implémentation du comportement attendu du modèle. Dans notre flot d'instrumentation, cette deuxième implémentation est synthétisée de façon automatisée à partir de PSL. Par conséquent, les étapes non automatiques, comme la traduction de la spécification informelle vers une propriété PSL, doivent être aussi indépendantes que possible des choix de programmation.

7.2 Analyse des performances

Toutes les propriétés présentées dans la section précédente ont été vérifiées dynamiquement au cours de plusieurs types de simulations. Nous avons mené des simulations ayant des durées différentes, parfois avec génération aléatoire des entrées, parfois avec une macro-génération combinatoire selon la technique introduite dans la section 6.2.2.2. Dans la plupart des cas, nous avons aussi introduit des erreurs volontaires dans les designs, de façon à contredire les comportements décrits par les assertions : tous les comportements erronés ont été signalés par les moniteurs de surveillance.

Dans la section 7.2.1 nous réunissons les temps CPU pour certaines des simulations effectuées. Dans les résultats relatés, chaque design a été exécuté plusieurs fois, suivant 8 simulations de durée croissante. Chaque simulation a été effectuée d'abord sans aucune instrumentation par ISIS, ensuite avec la surveillance d'une propriété à la fois, enfin, dans certains cas, en surveillant toutes les propriétés simultanément. Toutes ces simulations ont été conduites pour les versions des designs sans erreurs volontaires, et sans utiliser la technique de parallélisation proposée dans la section 6.2.1. Le mode verbeux des moniteurs

et des composants du design a été désactivé si possible. Les temps CPU sont organisés en graphes où

- l'axe des abscisses indique le paramètre qui influence la durée de la simulation (par exemple, le nombre et la taille des messages envoyés par un producteur),
- l'axe des ordonnées indique le temps CPU.

Pour chaque graphe, un tableau résume

- le nombre d'évaluations de chaque moniteur,
- le pourcentage de surcharge induite par la vérification sur le temps CPU de la simulation originale.

La machine utilisée pour les mesures possède un Intel Core2 Duo à 3GHz, 2GB de RAM et Linux Debian. Afin d'éviter des aléa et des fausses mesures, chaque simulation a été répétée plusieurs fois.

7.2.1 Temps CPU et synthèse

Communication par FIFO simple Dans les résultats du graphique de la figure 7.9 sont montrés les temps CPU pour les 8 simulations du premier cas d'étude (cf. section 7.1.1.1).

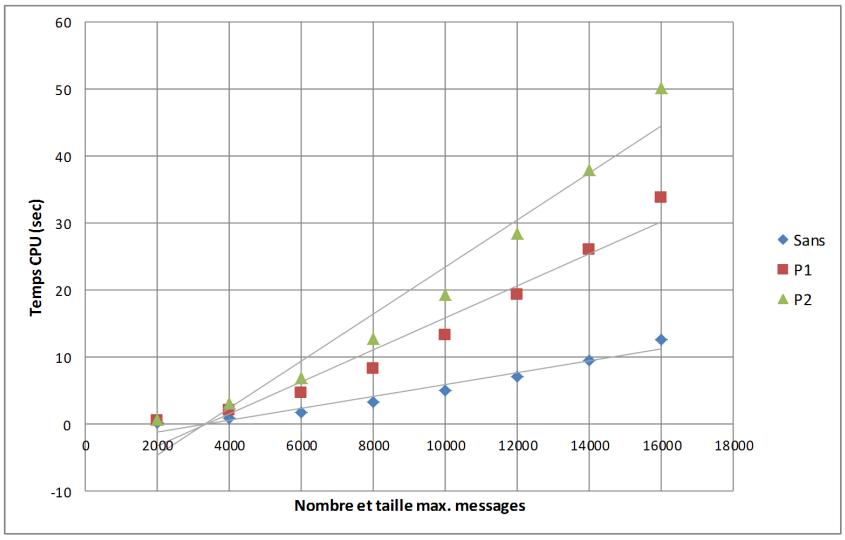

FIGURE 7.9 – FIFO, échanges simples : temps CPU pour les propriétés P_1 et P_2

Nombre et taille max. messages	Évaluations de la propriété		% Excédent temps CPU	
	P1	P2	P1	P2
2 000	~2 millions	~4 millions	173,7	289,5
4 000	~8 millions	~16 millions	173,1	297,4
6 000	~18 millions	~36 millions	172,4	295,4
8 000	~32 millions	~64 millions	159,7	300,3
10 000	~50 millions	~100 millions	167,7	290,7
12 000	~72 millions	~144 millions	174,4	305,0
14 000	~98 millions	~198 millions	173,4	299,7
16 000	~128 millions	~257 millions	168,9	299,4

FIGURE 7.10 – FIFO, échanges simples : nombre d'activations des moniteurs pour les propriétés P_1 et P_2

Chaque entrée sur l'axe des abscisse indique le nombre de messages envoyés par le producteur, ainsi que la taille maximale de chaque message. Par exemple, la première valeur dénote une simulation avec transmission de 2000 messages de longueur aléatoire comprise entre 1 et 2000 caractères. Chaque nuage de 8 points avec la même forme et la même couleur est associé à un mode de simulation. Les diamants représentent toujours la simulation sans instrumentation, identifiée par l'étiquette "Sans" dans la légende du graphique. Le nombre d'évaluations de chaque propriété est montré sur le tableau de la figure 7.10 : pour P_1, toutes les écritures dans la FIFO sont observées ; pour P_2 le nombre d'évaluation est double, puisque la propriété porte aussi sur les lectures dans le canal. Ce nombre d'évaluations est supérieur à 2 millions pour la simulation la plus courte avec P_1, et il dépasse les 250 millions pour la simulation la plus longue avec P_2. Dans cet exemple et dans les deux suivants, la moyenne des évaluations des propriétés est donc très élevée, ce qui implique naturellement un plus grand impact négatif sur le temps de simulation, comme indiquée par la colonne "% Excédent temps CPU" du tableau de la figure 7.10.

Communication par FIFO avec arbitre Le graphique de la figure 7.11 et le tableau de la figure 7.12 montrent les résultats relatifs au cas d'étude de la section 7.1.1.2. En ce qui concerne P_4, les résultats sont comparables à ceux du premier cas d'étude. Par contre, la propriété P_3 a un impact très limité puisqu'elle représente uniquement un invariant booléen et les changements des deux signaux booléens impliqués dans P_3 sont beaucoup moins nombreux que les échanges par la FIFO. Les valeurs sur l'axe des abscisses indiquent le nombre de messages envoyés par chaque producteur ; la taille maximale de chaque message est le double de ce nombre (par exemple, 1000 signifie que chaque producteur envoie 1000 messages de taille comprise entre 1 et 2000).

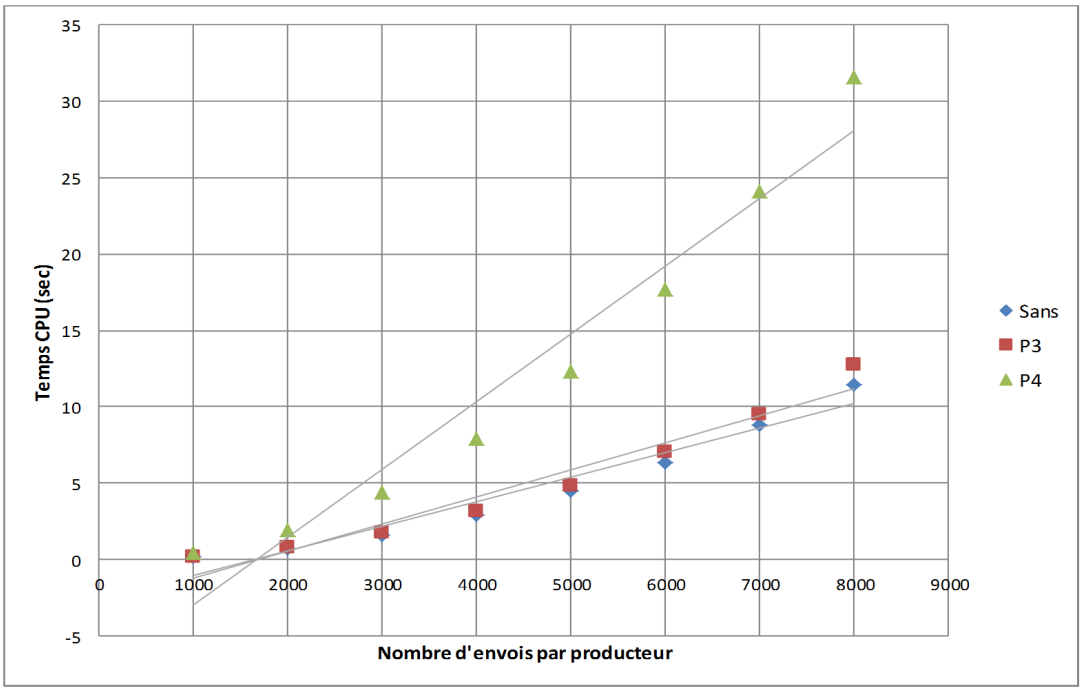

FIGURE 7.11 – FIFO, échanges avec arbitre : temps CPU pour les propriétés P_3 et P_4

Nombre d'envois par producteur	Évaluations de la propriété		% Excédent temps CPU	
	P3	P4	P3	P4
1 000	4 000	~2 millions	5,6	166,7
2 000	8 000	~7 millions	12,5	172,2
3 000	12 000	~18 millions	11,3	176,3
4 000	16 000	~32 millions	10,7	173,4
5 000	20 000	~50 millions	8,9	174,3
6 000	24 000	~71 millions	10,5	177,2
7 000	28 000	~98 millions	7,8	172,9
8 000	32 000	~128 millions	11,1	175,8

FIGURE 7.12 – FIFO, échanges avec arbitre : nombre d'activations des moniteurs pour les propriétés P_3 et P_4

Communication par canal défectueux Les deux premières propriétés vérifiées sur le design avec canal défectueux de la section 7.1.2 ont un impact visiblement plus limité que celles des deux exemples précédents, comme montré sur la figure 7.13. En effet, bien que le nombre d'activations des moniteurs soit toujours très élevé (cf. figure 7.14), les propriétés P_5 et P_6 n'utilisent aucun opérateur temporel complexe autre que `always`. Ainsi, nous pouvons remarquer que, à nombre équivalent d'évaluations, la performance de notre solution de vérification dynamique est influencée par la nature de la propriété plutôt que par celle du design. Cet aspect est repris et approfondi plus loin, après les cas d'études plus complexes.

En ce qui concerne la propriété P_f, le nombre d'évaluations est encore supérieur à celui de P_5 ou P_6, puisqu'aux activations du moniteur global s'ajoutent toutes les activations des instances des moniteurs pour la sous-propriété du `new` (ce qui est indiqué part l'addition dans la colonne pour P_f de la figure 7.14). Par ailleurs, cette propriété comporte aussi un deuxième opérateur temporel. Comme pour le premier design, chaque valeur sur l'axe des abscisse indique le nombre de messages envoyés par le producteur, ainsi que la taille maximale de chaque message.

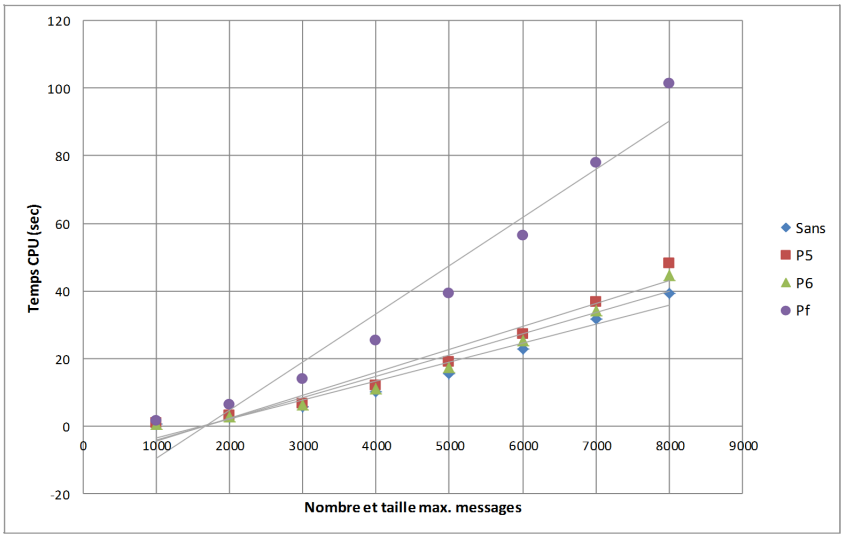

FIGURE 7.13 – Canal défectueux : temps CPU pour les propriétés P_5, P_6 et P_f

Nombre et taille max. messages	Évaluations de la propriété			% Excédent temps CPU		
	P5	P6	Pf	P5	P6	Pf
1 000	~1,5 millions	~1,2 millions	~1,5 + ~2 millions "*before*"	19,0	7,9	152,4
2 000	~6 millions	~5 millions	~6 + ~7 millions "*before*"	16,0	7,8	142,4
3 000	~13,5 millions	~11 millions	~13 + ~16,5 millions "*before*"	19,0	10,0	146,5
4 000	~24 millions	~20 millions	~24 + ~30 millions "*before*"	18,6	8,4	147,6
5 000	~37 millions	~31 millions	~37 + ~46,5 millions "*before*"	23,1	13,0	155,2
6 000	~53,5 millions	~45 millions	~53 + ~67 millions "*before*"	18,6	10,1	146,1
7 000	~73,5 million	~61 millions	~73,5 + ~91,5 millions "*before*"	16,2	8,0	146,0
8 000	~96 millions	~80 millions	~96,5 + ~120 millions "*before*"	22,8	13,6	158,7

FIGURE 7.14 – Canal défectueux : nombre d'activations des moniteurs pour les propriétés P_5, P_6 et P_f

Contrôleur DMA Les expérimentations effectuées avec le contrôleur DMA de la section 7.1.3 offrent des résultats bien plus représentatifs du type de vérification dynamique propre aux plate-formes transactionnelles. Pour cet exemple, nous avons conduit plusieurs types d'études. En premier lieu, nous avons mené des simulations sans instrumentation et avec une seule propriété à la fois. Ensuite, nous avons simulé le design instrumenté par les trois propriétés en même temps. Enfin, nous avons gardé le mécanisme de surveillance pour les trois propriétés, mais nous avons remplacé la fonction d'évaluation du moniteur (méthode `update` de la classe *Monitor* sur la figure 4.8 à la page 71) par une fonction vide, cela pour analyser l'impact du mécanisme de surveillance seul sur la simulation. Pour une meilleure lisibilité, la première étude est montrée sur la figure 7.15 et les deux dernières études sont présentées sur la figure 7.16. Dans les deux figures, l'axe des abscisses indique le nombre de transferts effectués. Le tableau de la figure 7.17 montre le nombre d'évaluations de chaque propriété.

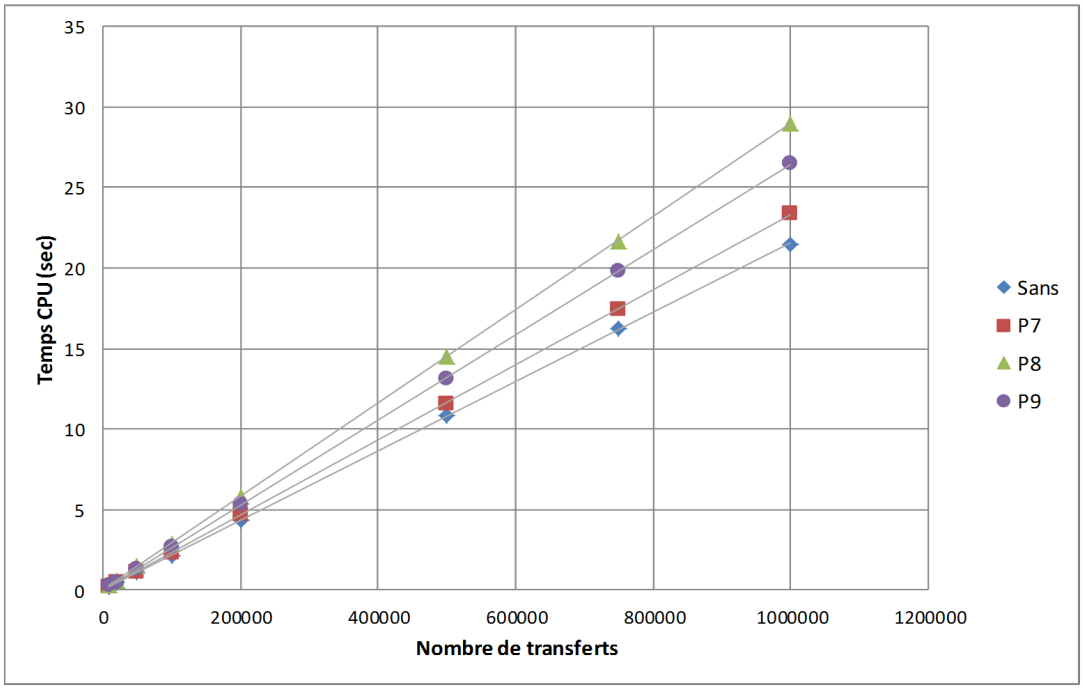

FIGURE 7.15 – DMA : temps CPU pour les propriétés P_7, P_8 et P_9

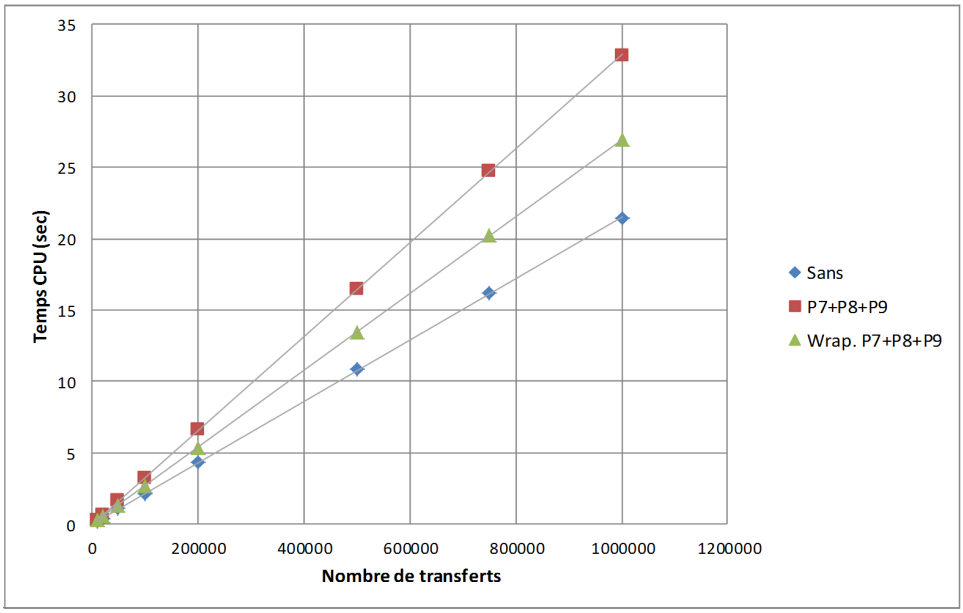

FIGURE 7.16 – DMA : temps CPU pour les trois propriétés P_7, P_8 et P_9 simultanément

Nombre de transferts	Évaluations de la propriété			% Excédent temps CPU				
	P7	P8	P9	P7	P8	P9	P7+P8+P9	Wrap. P7+P8+P9
10 000	70 000	320 000	210 000	4,8	33,3	19,0	52,4	23,8
20 000	140 000	640 000	420 000	7,0	34,9	20,9	51,2	23,3
50 000	350 000	1 600 000	1 050 000	7,4	33,3	22,2	51,9	24,1
100 000	700 000	3 200 000	2 100 000	7,4	33,3	22,2	51,4	23,1
200 000	1 400 000	6 400 000	4 200 000	7,6	34,0	22,5	53,0	24,3
500 000	3 500 000	16 000 000	10 500 000	7,1	33,9	21,3	52,1	24,3
750 000	5 250 000	24 000 000	15 750 000	7,8	33,9	22,2	52,6	24,6
1 000 000	7 000 000	32 000 000	21 000 000	8,9	35,1	23,6	53,2	25,4

FIGURE 7.17 – DMA : nombre d'activations des moniteurs pour les propriétés P_7, P_8 et P_9

La colonne "P7+P8+P9" indique le pourcentage d'excédent du temps CPU lors de la simulation avec les trois propriétés simultanément. La dernière colonne montre ce même pourcentage lors de la simulation avec le mécanisme de surveillance mais sans évaluation des moniteurs. Chaque transfert comporte la programmation du DMA, la copie entre les mémoires et la notification de fin de transfert. Tous les transferts sont programmés avec la même longueur, égale à 64 octets (une copie atomique exécutée par le DMA équivaut à 4 octets), et chaque transfert implique toujours le même nombre d'opérations de communication. Par conséquent, le nombre d'activations des moniteurs est exact et constant. Par exemple, le nombre d'évaluations de P_7 est toujours égal à 7 fois le nombre de transferts : pour un transfert, le CPU écrit 5 fois à destination du DMA (4 programmations avant la copie mémoire et un *reset* du registre de contrôle après la copie) et le signal d'interruption change deux fois de valeur (levé d'interruption et remise à 0).

Nous pouvons constater que l'augmentation du temps de simulation dans ce design est linéaire par rapport au nombre de transferts, et cela reste valable aussi bien pour la simulation simple que pour toutes celles instrumentées, même en présence de plusieurs propriétés. Par ailleurs, le "surcoût" provoqué par les moniteurs est plutôt faible, comme

indiqué sur la figure 7.17.

Les résultats affichés sur le graphique de la figure 7.16 montrent que le mécanisme de surveillance recouvre une bonne partie de la surcharge en termes de temps CPU. Les symboles carrés (étiquette “P7+P8+P9”) représentent les simulations avec les trois propriétés, tandis que les triangles (étiquette “Wrap. P7+P8+P9”) représentent les simulations avec le mécanisme de surveillance uniquement, sans calcul associé à l’évaluation des moniteurs. Les temps obtenus dans ce deuxième cas sont supérieurs à ceux pour P_7 ou P_9 dans la figure 7.15. Cet aspect est dû principalement aux notifications par les sujets observés, et il souligne clairement la bonne performance des moniteurs pour les propriétés. Enfin, il convient de remarquer que la surcharge induite par la vérification des trois propriétés simultanément est inférieure à la somme des surcharges comportées par chaque propriété séparément : puisqu’en surveillant plusieurs propriétés qui portent sur des communications communes le temps de notification est en partie factorisé, il est généralement préférable de mener une seule simulation avec n propriétés actives plutôt que n simulations avec une seule propriété à la fois.

Packet Switch Pour le design de la section 7.1.4 nous avons conduit des simulations représentant un nombre croissant d’itérations du fonctionnement du *switch*. Chaque itération comporte une lecture des entrées du *switch*, l’acheminement via une rotation de l’anneau de registres (sur les fronts montants de l’horloge du *switch*) et une écriture des sorties. La figure 7.18 résume les temps CPU de la simulation simple et instrumentée ; le pourcentage de surcharge et le nombre d’évaluations de chaque propriété sont indiqués sur le tableau de la figure 7.19.

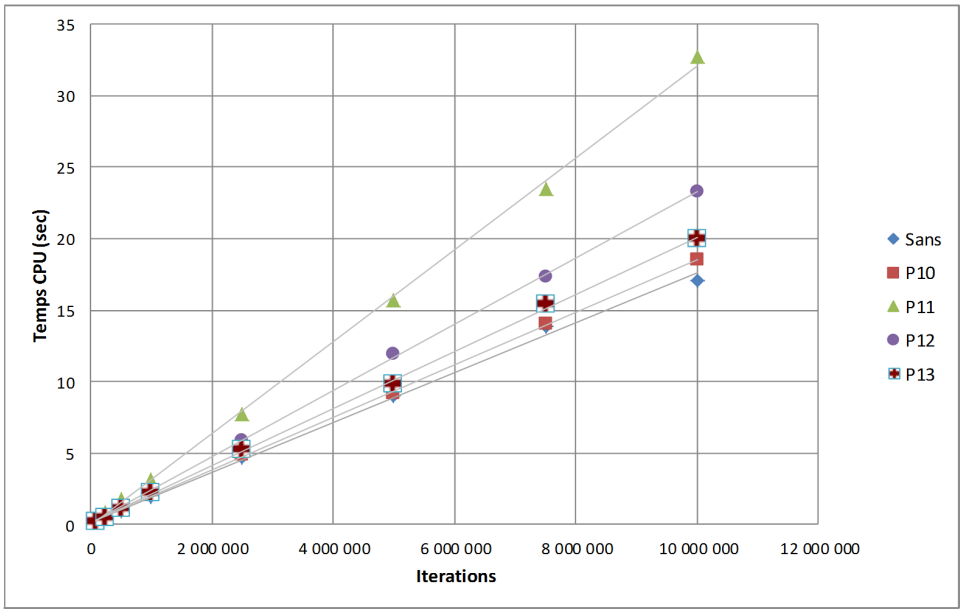

FIGURE 7.18 – Packet Switch : temps CPU pour les propriétés P_{10} à P_{13}

En cas de propriétés qui emploient l’opérateur **new**, l’identification du moment où une instance de moniteur peut être détruite est très important, comme expliqué dans la section 5.2.3.1. Dans notre solution, quand une propriété sous la portée du **new** est satisfaite fortement ou violée, elle est automatiquement désactivée. La différence de “surcoût” entre

Itérations	Évaluations de la propriété				% Excédent temps CPU			
	P10	P11	P12	P13	P10	P11	P12	P13
100 000	~100K	~33K + ~1,2 millions "*eventually*"	~150K + ~100K "*before*"	~35K + ~20K "*next_event*" + ~20K "*before*"	5,3	78,9	31,6	15,8
250 000	~240K	~85K + ~3 millions "*eventually*"	~390K + ~250K "*before*"	~85K + ~50K "*next_event*" + ~55K "*before*"	8,5	80,9	23,4	8,5
500 000	~480K	~170K + ~6 millions "*eventually*"	~780K + ~530K "*before*"	~200K + ~110K "*next_event*" + ~110K "*before*"	6,3	79,2	32,3	15,6
1 000 000	~950K	~320K + ~11,5 millions "*eventually*"	~1,5 millions + ~1 million "*before*"	~420K + ~220K "*next_event*" + ~220K "*before*"	7,8	63,2	22,8	17,1
2 500 000	~2 millions	~750K + ~28 millions "*eventually*"	~4 millions + ~2,5 millions "*before*"	~1 million + 500K "*next_event*" + ~500K "*before*"	3,4	63,4	26,0	12,1
5 000 000	~3,5 millions	~1,5 millions + ~60 millions "*eventually*"	~7,5 millions + ~5 millions "*before*"	~1,5 million + ~1 million "*next_event*" + ~1 million "*before*"	2,2	74,0	32,3	9,1
7 500 000	~5 millions	~2,5 millions + ~90 millions "*eventually*"	~11 millions + ~7,5 millions "*before*"	~2,5 millions + ~1,5 millions "*next_event*" + ~1,5 millions "*before*"	0,9	69,2	25,2	11,1
10 000 000	~7 millions	~3,5 millions + ~120 millions "*eventually*"	~15 millions + ~10 millions "*before*"	~3,5 millions + ~2 millions "*next_event*" + ~2 millions "*before*"	8,4	91,8	36,5	16,8

FIGURE 7.19 – Packet Switch : nombre d'activations des moniteurs pour les propriétés P_{10} à P_{13}

P_{11} et P_{12} est, en grande partie, due au fait que chaque sous-instance de P_{12} est systématiquement détruite, au plus tard, après écoulement du temps limite. Par contre, toutes les sous-instances de P_{11} pour lesquelles un paquet est perdu (on rappelle qu'il y a une erreur dans ce design, d'où les pertes) ne seront pas détruites jusqu'à la fin de la simulation : l'opérateur **`eventually`**! attend toujours son opérande.

Plate-forme Motion-JPEG Les figures 7.20 et 7.21 montrent les résultats de simulation pour le design de la section 7.1.5 pour le décodage d'une vidéo donnée.

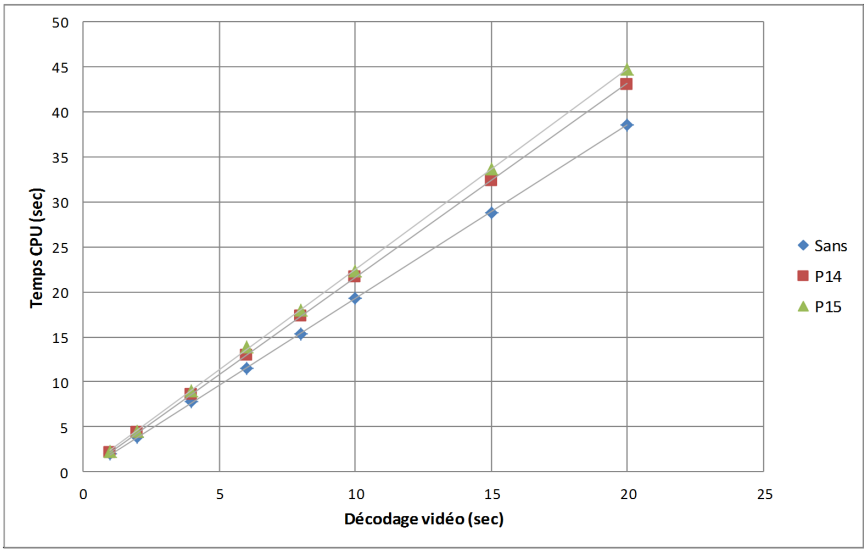

FIGURE 7.20 – Plate-forme MJPEG : temps CPU pour les propriétés P_{14} et P_{15}

Décodage vidéo (sec)	Évaluations de la propriété		% Excédent temps CPU	
	P14	P15	P14	P15
1	1 351 686	1 351 686	11,9	14,9
2	2 727 942	2 727 942	13,0	17,4
4	5 505 030	5 505 030	11,2	15,6
6	8 257 542	8 257 542	12,1	20,6
8	11 010 054	11 010 054	12,7	17,0
10	13 762 566	13 762 566	12,7	16,3
15	20 594 694	20 594 694	12,7	17,0
20	27 426 822	27 426 822	11,5	15,8

FIGURE 7.21 – Plate-forme MJPEG : nombre d'activations des moniteurs pour les propriétés P_{14} et P_{15}

Chaque valeur sur l'axe des abscisses correspond à un temps de décodage vidéo. Comme dans le cas du DMA, le nombre d'évaluations est constant. La surcharge provoquée par la vérification est plutôt limitée, essentiellement parce que, bien que le nombre d'évaluations des moniteurs soit de plusieurs millions, le calcul effectué par les moniteurs est moindre par rapport à celui nécessaire pour le décodage et surtout pour l'affichage de la vidéo. Comme pour le DMA, l'augmentation du temps de simulation est linéaire, sans ou avec surveillance des propriétés.

SoC de traitement de données bord pour charge utile spatiale Les résultats relatifs au dernier cas d'étude (section 7.1.6) sont montrés sur deux graphes distincts. Le graphe de la figure 7.22 comporte 8 simulations en mode *sans* temps pour le design sans instrumentation, ainsi qu'avec instrumentation par P_{16} et P_{17}. Dans ces mesures, le nombre d'images traitées par le bloc de convolution varie entre 5000 et 300000 sur l'axe des abscisses.

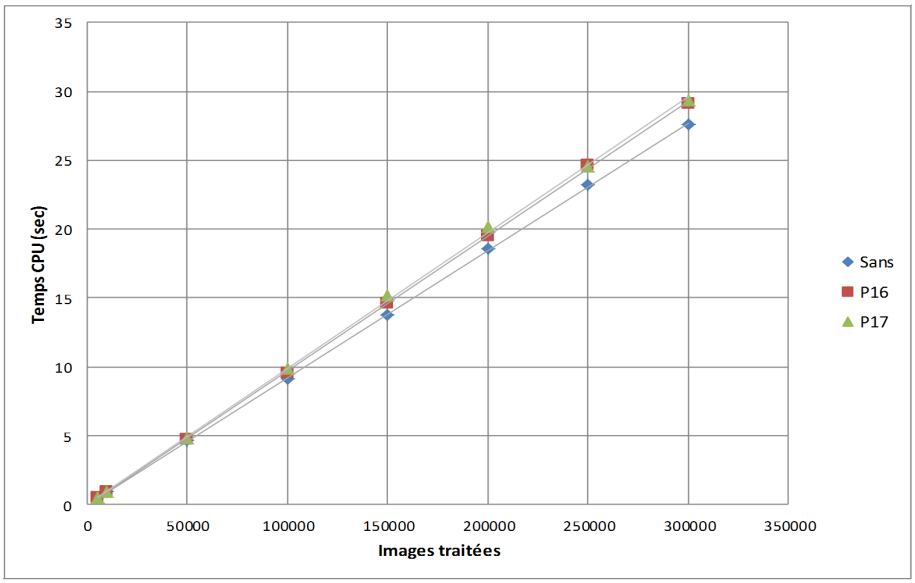

FIGURE 7.22 – Plate-forme de traitement vidéo pour la télémesure : temps CPU pour les propriétés P_{16} et P_{17}

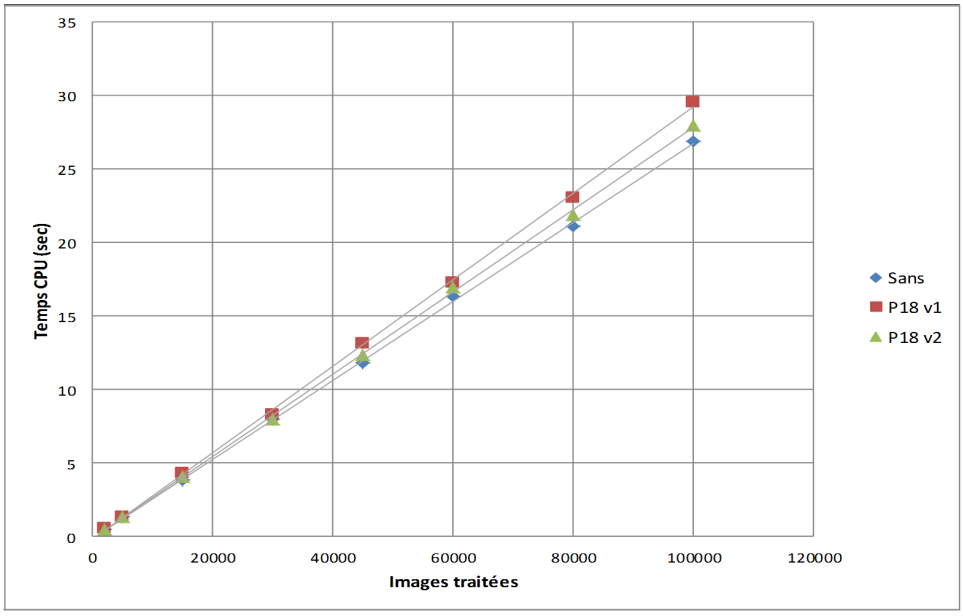

FIGURE 7.23 – Plate-forme de traitement vidéo pour la télémesure : temps CPU pour les propriétés P_{18} v1 et P_{18} v2

Images traitées	Évaluations de la propriété		% Excédent temps CPU	
	P16	P17	P16	P17
5 000	25 000	25 000	2,0	0,0
10 000	50 000	50 000	2,2	4,3
50 000	250 000	250 000	2,1	3,4
100 000	500 000	500 000	4,9	8,1
150 000	750 000	750 000	6,2	10,6
200 000	1 000 000	1 000 000	5,0	8,4
250 000	1 250 000	1 250 000	6,2	6,0
300 000	1 500 000	1 500 000	5,7	6,5

FIGURE 7.24 – Plate-forme de traitement vidéo pour la télémesure : nombre d'activations des moniteurs pour les propriétés P_{16} et P_{17}

Images traitées	Évaluations de la propriété		% Excédent temps CPU	
	P18 v1	P18 v2	P18 v1	P18 v2
2 000	40 000	24 000	14,6	6,3
5 000	100 000	60 000	4,6	1,5
15 000	300 000	180 000	12,9	6,8
30 000	600 000	360 000	3,6	0,1
45 000	900 000	540 000	11,3	5,2
60 000	1 200 000	720 000	6,0	4,0
80 000	1 600 000	960 000	9,4	4,0
100 000	2 000 000	1 200 000	9,8	4,1

FIGURE 7.25 – Plate-forme de traitement vidéo pour la télémesure : nombre d'activations des moniteurs pour les propriétés P_{18} v1 et P_{18} v2

Le graphe de la figure 7.23 affiche 8 simulations en mode *avec* notion de temps, durant lesquelles les images traitées sont mémorisées dans la mémoire AHB. Dans ce cas, nous avons fait varier entre 2000 et 100000 le nombre d'images traitées, et nous avons surveillé les propriétés P_{18} v1 et P_{18} v2. Le nombre d'activations des moniteurs est indiqué par les tableaux des figures 7.24 et 7.25. Une fois de plus, les résultats montrent une assez faible surcharge en présence des moniteurs de surveillance. Dans ce cas, bien que les propriétés soient plus complexes que celles énoncées pour la plate-forme Motion-JPEG, le nombre d'évaluations est bien plus limité (le million d'évaluations n'est dépassé ici que pour les trois dernières mesures). De plus, comme pour le cas d'étude précédent, le calcul effectué par les moniteurs est peu important par rapport à celui de la plate-forme, en particulier à cause des affichages de celle-ci. L'augmentation du temps de simulation est toujours linéaire par rapport au nombre d'images traitées.

Synthèse Le temps de simulation en présence des moniteurs de surveillance sont globalement intéressants. Même quand la sollicitation des moniteurs est très élevée, les résultats expérimentaux montrent des temps acceptables. Toute la pénalisation induite sur le temps de simulation orignal est due uniquement à la complexité de la propriété et au nombre d'occurrences des communications que l'utilisateur a choisi d'observer. En particulier, il faut souligner que, contrairement aux techniques de vérification statique, la taille du design à vérifier ne constitue pas un obstacle. Au contraire, tous les résultats expérimentaux de cette section montrent que plus un design est complexe, plus notre solution de vérification se montre efficace, puisque les moniteurs et leur surveillance constituent une partie très réduite dans la totalité du système instrumenté. Enfin, il convient de remarquer que le temps de construction des moniteurs et le temps d'instrumentation automatique par ISIS sont négligeables ; seulement l'étape d'analyse syntaxique du code du DUV peut nécessiter plusieurs secondes.

7.2.2 Flexibilité du modèle

L'un des avantages de la technique de vérification dynamique que nous avons proposée réside dans sa flexibilité d'utilisation. Notre solution s'appuie sur deux éléments essentiellement indépendants : le moniteur de surveillance et la méthode de surveillance. L'un ou l'autre pourraient être remplacés par d'autres solutions plus adaptées dans un contexte différent. Par exemple, si nécessaire l'API de la SCV introduite dans la section 4.2.5 pourrait être utilisée pour activer nos moniteurs, à la place de la méthode de surveillance étudiée dans cette thèse. De même, comme montré sur la figure 4.9 à la page 73, le moniteur est englobé dans un composant *wrapper* qui déclenche son évaluation, par conséquent il peut être remplacé par un autre composant aux fonctionnalités équivalentes, doté d'une fonction d'évaluation. Au cours de nos expérimentations, nous nous sommes appuyés sur cet aspect pour comparer les performances de nos moniteurs avec celles des moniteurs C++ générés par l'outil FoCs d'IBM[4] (cf. chapitre 3). Puisque FoCs ne prévoit pas l'expression de conditions booléennes complexes concernant les paramètres des communications et il ne supporte pas la couche modélisation de PSL, nous avons procédé comme suit :

- chaque expression booléenne b dans la propriété a été remplacée par un identifiant i, la valeur de b étant calculée dans le *wrapper* et affectée par celui-ci à i ;

[4] Tous les tests ont été effectués avec la version 2.04 de FoCs.

- les opérations de la couche modélisation, si présentes, ont été déplacées dans le *wrapper* ;
- l'instance du moniteur ISIS contenue dans le *wrapper* a été remplacée par celle du moniteur FoCs ;
- l'appel à la méthode `update` du moniteur ISIS a été remplacé par l'appel à la méthode `transition` d'évaluation du moniteur FoCs.

Les résultats de ces comparaisons, présentés dans [FP10b], ont montré une bien meilleure performance de nos moniteurs par rapport à ceux de FoCs. Dans la plupart des cas, les temps en présence des moniteurs FoCs étaient plus que doublés. Par exemple, dans le cas du contrôleur DMA de la section 7.1.3, les temps de simulation pour 200000 transferts avec les moniteurs FoCs pour les propriétés P_7, P_8 et P_9 étaient respectivement de 9, 27 et 38 secondes, alors qu'avec nos moniteurs, les temps n'atteignent pas les 7 secondes.

Chapitre 8

Conclusion et perspectives

L'adoption de méthodologies comme le "*platform-based design*" [SVM01] s'avère désormais inévitable pour lutter contre la formidable évolution en termes de complexité et de profusion des circuits électroniques modernes. SystemC semble faire ses preuves en ce qui concerne la conception et la modélisation de ces circuits, mais nous estimons qu'il reste encore à lui associer des solutions relatives à la *vérification* des systèmes modélisés [Var07, EEH+07]. La démarche de vérification dirigée par les assertions, ou ABV, possède une valeur désormais reconnue dans l'industrie depuis plusieurs années [Mal02, Das06] et qui a atteint une certaine maturité au niveau RTL. Avec les travaux présentés dans cette thèse, nous nous sommes attachés à fournir un support concret pour l'ABV à des niveaux plus abstraits. La solution adoptée, basée sur deux standards IEEE, SystemC et PSL, se veut réaliste et utilisable en pratique.

Notre contribution s'articule selon plusieurs axes, afin d'offrir un cadre aussi bien théorique que pratique.

8.1 Contributions

Moniteurs orientés TLM La première contribution est la construction automatique, à partir d'une spécification formelle en PSL, de composants de surveillance adaptés aux niveaux transactionnels les plus abstraits. Il s'agit d'une solution générique, qui favorise la réutilisation des composants, et qui est indépendante de tout simulateur existant. Le résultat obtenu est une bibliothèque C++ de composants élémentaires et une nouvelle méthode d'interconnexion de ces composants capable de produire des moniteurs orientés TLM très efficaces. Chaque moniteur est doté de trois sorties dont la combinaison des valeurs indique l'état de la propriété (valide, en attente, violée) à tout moment durant la simulation. Différents niveaux de verbosité peuvent être utilisés, par exemple pour signaler toutes les évaluations d'une formule, ou pour afficher les valeurs de tous les opérandes et variables auxiliaires ou encore l'état de chaque opérateur impliqué.

Modèle de surveillance Nous avons également fourni un modèle d'échantillonnage et un mécanisme de surveillance des propriétés logico-temporelles au niveau transactionnel. Ce mécanisme est à la fois indispensable au fonctionnement des moniteurs et indépendant de ces derniers. Un aspect essentiel est le fait qu'il n'altère pas la fonctionnalité du design : le modèle ne s'appuie pas sur des concepts liés au noyau de simulation SystemC, tels que

les événements ou les processus, et les observateurs se limitent à recevoir et analyser les données du DUV, sans lui transmettre aucune information. Le fait de pouvoir observer les méthodes d'une classe quelconque permet d'étendre la portée de la solution à d'autres cas de figure, non nécessairement liés aux communications entre blocs matériels. Par ailleurs, ce mécanisme permet de traiter aussi bien des designs sans notion de temps et d'horloge que des circuits avec une ou plusieurs horloges de synchronisation.

L'infrastructure d'observation nécessite la génération de plusieurs classes et une modification des parties déclaratives de certains modules, afin de remplacer les composants observés. Cette instrumentation est automatique, mais les fichiers sources des déclarations relatives au système modélisé doivent être à disponibles.

Sémantique et mise en œuvre pour les variables auxiliaires Si la surveillance des données des communications au niveau transactionnel est souvent indispensable, la mémorisation de ces données et la modélisation de valeurs auxiliaires constituent également deux aspects incontournables. Nous avons défini un support théorique et une mise en œuvre pour les variables auxiliaires. À notre connaissance, il s'agit de la première sémantique qui réunit à la fois les variables globales de la couche modélisation de PSL et le concept de variables locales à une sous-formule.

À la fois basée sur le standard IEEE de PSL [PSL05] et sur les travaux de [CM04], la sémantique proposée caractérise de façon cohérente et directe l'évolution des variables le long d'une trace de simulation, ainsi que leur incidence sur la vérification des propriétés.

Nous avons de plus réalisé la mise en œuvre du support pour ces variables auxiliaires. L'exemple du *Packet Switch*, entre autres, a permis de souligner le fait que l'impact des variables locales sur la surcharge en temps de simulation peut être différent selon la façon dont la propriété est écrite (cf. chapitre 7).

ISIS Tous ces concepts ont été entièrement implémentés dans ISIS, un prototype pour la vérification semi-formelle de systèmes matériels et logiciels en cours de conception. L'outil permet la génération automatique des moniteurs de surveillance et l'instrumentation automatisée du DUV selon le mécanisme d'observation proposé. Sa structure est présentée en annexe C.

Outre la présence d'une interface graphique simple, ISIS offre la possibilité de fonctionnement en ligne de commande, cela par exécution d'un fichier de directives. Ce mode *scripting* a été réalisé à la demande de partenaires industriels, afin de rendre l'outil manipulable en vraie grandeur.

Nous nous sommes aussi intéressés à des moyens de couvrir la génération de *stimuli* pour le DUV. Ainsi, nous avons étudié le couplage d'ISIS avec TOBIAS (LIG), nous permettant de parvenir à une génération massive de séquences de test de façon combinatoire, cela à partir d'ensembles succincts de valeurs et d'instructions choisies.

Bilan Les bonnes performances et la variété des expérimentations menées nous ont permis de prouver l'intérêt et l'applicabilité de la solution proposée. L'expressivité de PSL s'avère satisfaisante, toutefois les différents cas d'étude ont rendu clair le besoin d'employer la couche modélisation et les variables auxiliaires dans les spécifications les moins rudimentaires. C'est pourquoi nous avons porté un soin particulier à cet aspect dans notre contribution. Le résultat final est à la fois une méthodologie avec un flot de vérification dynamique basée sur les assertions, et un outil prototype mettant en œuvre

ces principes. À notre connaissance, il s'agit du premier outil utilisable en pratique quelle que soit la complexité du design SystemC TLM à vérifier.

Au delà de la vérification d'exigences fonctionnelles, la solution offerte peut être utilisable dans des contextes comme :

- L'analyse de couverture. En effet, l'utilisation conjointe des assertions et de la couche modélisation permet d'observer divers types de phénomènes dans la plate-forme et d'y associer des actions sur des variables auxiliaires. Il serait trivial d'utiliser cette capacité à but d'analyse de couverture, par exemple pour compter le nombre d'accès à certaines zones mémoire ou la proportion entre divers types de communications dans le système.
- Les tests de non régression. Une fois des propriétés système vérifiées avec certaines instances de composants, ces mêmes propriétés peuvent être "rejouées" après remplacement par d'autres composants. Par exemple, dans le cas de la plate-forme pour le traitement de données bord pour charge utile spatiale, vue dans le chapitre 7, des mises au point ont pu être faites avec un simple pseudo-processeur pour jouer le rôle du Leon3 ; elles permettront de s'assurer que le fonctionnement reste correct après remplacement par un véritable ISS pour ce processeur.

Ce travail fournit des bases solides pour une solution de vérification semi-formelle tout au long du flot de conception. Diverses améliorations à court terme peuvent déjà être envisagées. Les perspectives à plus long terme sont prometteuses.

8.2 Travaux futurs

Améliorations à court terme

Le code généré par ISIS lors de l'étape d'instrumentation du DUV n'est pas toujours optimisé pour réduire au minimum l'impact sur le temps de simulation. En particulier, il serait intéressant d'optimiser la façon dont les moniteurs sont déclenchés par les *wrappers* et, surtout, la façon dont ces derniers sont notifiés par les composants observés.

En ce qui concerne la technique de surveillance proposée, nous nous appuyons sur des mécanismes d'héritage pour repérer les occurrences des méthodes de communication, ainsi que les paramètres et les valeurs de retour de ces méthodes. Cette solution exploite les possibilités offertes par C++ mais présente un léger inconvénient qui est la nécessité de remplacer certains composants du design par leur variante observée (sous-classes de la classe d'origine). Une alternative pratique pouvant être intégrée dans notre méthode de surveillance serait l'emploi de la programmation orientée aspects [KLM+97, KLTB10] pour remplacer l'utilisation de sous-classes observées par des *points d'action* et des *greffons* : les premiers définissent les points de jonction, comme l'appel d'une méthode ou l'affectation d'une variable, les deuxièmes sont les programmes activés avant, autour ou après ces points de jonction (dans notre cas, les points d'action seraient associés aux instants d'observation et le *greffon* serait une simple notification des *wrappers* pour les moniteurs). Ces deux mécanismes propres à la programmation orientée aspects présentent l'avantage de ne pas nécessiter de modification du code source, mais ils requièrent un *tisseur d'aspects* : AspectC++[1] fournit une extension pour la programmation orientée aspects à C++ que

[1] `http://www.aspectc.org/`

nous pourrions utiliser ; il faudrait s'assurer de sa compatibilité intégrale avec SystemC et son noyau de simulation.

Si notre solution de vérification permet de considérer à la fois les interfaces bloquantes et non bloquantes, il reste à étudier de façon plus précise son applicabilité en cas de découplage temporel : une étude approfondie de tous les styles de la bibliothèque TLM 2.0 doit être menée afin de rendre nos travaux les plus génériques et universellement applicables.

Alors que le but principal des moniteurs est celui de signaler tout comportement erroné, il ne faut pas négliger leur apport possible durant la phase d'identification et de correction de l'erreur (*debug*). Pour cela, les sorties produites par les composants de surveillance peuvent être enrichies et améliorées. En particulier, nous nous sommes toujours restreints au mode de simulation textuel offert par le noyau OSCI de SystemC, mais il serait intéressant de doter les moniteurs d'une sortie graphique, par exemple en utilisant des diagrammes de séquence UML selon une approche similaire à celle de [dSS09].

La qualité de la sortie produite n'est pas le seul apport possible en matière de *debug*. Un voie intéressante à explorer est la déclinaison automatique d'une propriété en plusieurs variantes, afin qu'une seule relation entre transactions soit surveillée à des points différents et pour plusieurs composants le long d'un même chemin dans le système : en cas de violation d'un sous-ensemble des ces variantes, le composant potentiellement défaillant serait plus facilement identifié.

Perspectives

Afin de constituer la trace sur laquelle les propriétés sont évaluées, nous nous appuyons sur la séquentialité des actions durant l'exécution. Ainsi, nous obtenons un échantillonnage où deux opérations de communication ne peuvent pas être considérées comme "simultanées" : la conjonction de l'occurrence de deux communications c_1 et c_2, par exemple, est exclue dans ce contexte (à noter que l'opérateur *et* est toutefois significatif entre deux propriétés temporelles, ou entre l'occurrence d'une communication et une condition sur ses paramètres, par exemple). Aux niveaux de modélisation faisant intervenir la notion de temps, l'utilisateur pourrait vouloir raisonner à propos de communication "simultanées" du point de vue du temps de simulation, même s'il nous paraît pouvoir être dangereux de se baser sur de tels concepts pour la mise au point des plates-formes. Dans ce cas, la notion d'échantillonnage devra être revue, car il n'y aurait plus nécessairement de correspondance biunivoque entre les actions de communication et les instants d'évaluation. Par exemple, les cinq communications indiquées par des points noirs sur la trace de la figure 8.1 ne constitueraient pas cinq instants d'évaluation distincts mais trois seulement, car les trois communications c_1, c_2 et c_3 pourraient être considérées comme "simultanées".

Cette nouvelle interprétation, dont la mise en œuvre est tout à fait possible, impliquerait les extensions suivantes : d'une part, il faudrait étendre la sémantique proposée pour y inclure la notion de temps de simulation (un concept proche mais pas équivalent peut être celui de la sémantique avec horloge de [PSL05, PSL10]) et la notion de "simultanéité" de deux événements vis-à-vis de ce temps ; d'autre part, il faudrait étendre le fonctionnement des *wrappers* afin qu'ils n'évaluent chaque propriété qu'une seule fois avant chaque incrément du temps de simulation. Par exemple, pour la trace de simulation

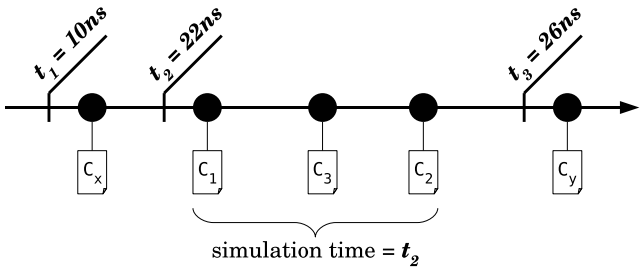

FIGURE 8.1 – Trace de simulation avec communications “simultanées”

avec temps de la figure 8.1, la propriété **always**(c_1 && c_2 -> c_3) ne devrait pas être évaluée trois fois à t_2, mais une seule fois après la communication c_2.

Les modèles TLM peuvent servir comme point de départ du flot de conception. Leur vérification ne permet pas, seule, de faire face à tous les besoins de validation du flot. Si ces modèles transactionnels sont raffinés puis synthétisés le long du flot, il est impératif de s’assurer que les fonctionnalités vérifiées pour ces modèles restent valables à de plus bas niveaux, typiquement RTL. Des études existent en matière d’adaptation des assertions entre TLM et RTL [BFP06, KT07, BFG10] ou en termes d’expression de ces propriétés [EESV07], mais le problème d’une définition claire de la contrepartie au niveau RTL des assertions énoncées en TLM reste irrésolu.

Enfin, une voie pour rendre notre solution de vérification dynamique plus complète serait de la coupler avec des techniques de vérification statique. Si notre solution est particulièrement adaptée aux systèmes les plus complexes, actuellement inaccessibles aux techniques formelles, elle n’apporte pas une réponse exhaustive quant à la satisfaction des propriétés. Des techniques de vérification statique peuvent permettre de s’assurer du bon fonctionnement des sous-blocs les plus critiques du système complet, dont le comportement global aurait préalablement été validé grâce à notre méthode.

Annexe A

Makefile type pour les designs instrumentés

A.1 *Makefile* original

```
#######################################################################
##                            TIMA Laboratory - VDS Team
#######################################################################
## File : Makefile
## ...
#######################################################################

#################################### Paths ##############################
PWD := $(shell pwd)
CURRENT_DIR = $(PWD)

SYSC_HOME = /home/softs/systemc/systemc-2.2.0
SYSC_INC  = $(SYSC_HOME)/include
SYSC_LIB  = $(SYSC_HOME)/lib-linux

TLM_HOME = /home/softs/systemc/TLM-2.0.1
TLM_INC  = $(TLM_HOME)/tlm

################################### Settings ############################
CC      = g++
CFLAGS  = -Wall ...
DFLAGS  = ...
LDFLAGS = -lsystemc
INCDIR  = -I$(CURRENT_DIR) -I$(SYSC_INC) -I$(TLM_INC)
LIBDIR  = -L$(SYSC_LIB)

#################################### Files ##############################
SRCS   = ...
OBJS   = $(SRCS:.cpp=.o)
TARGET = run.exe

#################################### Tasks ##############################
.PHONY: clean veryclean

all: compile

.pre-compile:
```

```
compile: .pre-compile $(TARGET)

$(TARGET): $(OBJS)
    $(CC) $(CFLAGS) $(INCDIR) $(LIBDIR) -o $@ $(OBJS) $(LDFLAGS)

$(OBJS): %.o: %.cpp
    $(CC) $(CFLAGS) $(DFLAGS) $(INCDIR) -c $< -o $@

clean:
    rm -f $(OBJS)
    rm -f $(TARGET)
```

A.2 *Makefile* pour le DUV instrumenté

```
#######################################################################
##                            TIMA Laboratory - VDS Team
#######################################################################
## File          : Makefile
## ...
#######################################################################

################################### Paths ##############################
PWD := $(shell pwd)
CURRENT_DIR = $(PWD)

# Ajout des repertoires avec l'instrumentation :
INSTRUMENT_DIR = $(CURRENT_DIR)/instrument
# Repertoires avec les types des composants observes :
SUBJECTS_DIR   = $(INSTRUMENT_DIR)/subjects
# Repertoires avec les moniteurs et les wrappers :
CHECKERS_DIR   = $(INSTRUMENT_DIR)/checkers
# Repertoires avec les fichiers sources modifies :
CHANGES_DIR    = $(INSTRUMENT_DIR)/changes

SYSC_HOME = /home/softs/systemc/systemc-2.2.0
SYSC_INC  = $(SYSC_HOME)/include
SYSC_LIB  = $(SYSC_HOME)/lib-linux

TLM_HOME = /home/softs/systemc/TLM-2.0.1
TLM_INC  = $(TLM_HOME)/tlm

# Ajout du repertoire avec la bibliotheque des moniteurs elementaires :
PSL_HOME = /home/softs/vds/mnt_libs/psl_sc_mnt
PSL_INC  = $(PSL_HOME)/include
PSL_LIB  = $(PSL_HOME)/lib

################################## Settings ############################
CC      = g++
# Les options de compilation sont inchangees :
CFLAGS  = -Wall ...
# Les drapeaux de debug peuvent etre inchanges, mais on peut aussi
#  ajouter les drapeaux pour les differents niveaux de trace des
#  moniteurs :
DFLAGS  = ... \
```

```
          -D_TRACE_ -D_TRACE_2_ -D_TRACE_3_ -D_ISIS_STATS_ \
          -D_ISIS_INTERACTIVE_
# Ajout de la bibliotheque des moniteurs elementaires :
LDFLAGS = -lsystemc -lpslmon
# Ajout des inclusions des en-tetes de la bibliotheque des moniteurs
#  elementaires, ainsi que des nouvelles en-tetes generees :
INCDIR  = -I$(CURRENT_DIR) -I$(SYSC_INC) -I$(TLM_INC) \
          -I$(SUBJECTS_DIR) -I$(CHECKERS_DIR) -I$(CHANGES_DIR)
# Ajout du repertoire de la bibliotheque des moniteurs elementaires :
LIBDIR  = -L$(SYSC_LIB) -L$(PSL_LIB)

################################## Files ##############################
# Il faut ajouter les nouveaux sources generes et enlever des sources
#  initiaux les fichiers qui auraient ete recrees dans CHANGES_DIR
#  (il s'agit des fichiers originaux qui contiennent maintenant un
#  ou plusieurs composants observes) :
SRCS   = ... \
         $(wildcard $(SUBJECTS_DIR)/*.cpp) \
         $(wildcard $(CHECKERS_DIR)/*.cpp) \
         $(wildcard $(CHANGES_DIR)/*.cpp)
OBJS   = $(SRCS:.cpp=.o)
TARGET = run.exe

################################## Tasks ##############################
.PHONY: clean veryclean

all: compile

.pre-compile:

compile: .pre-compile $(TARGET)

$(TARGET): $(OBJS)
    $(CC) $(CFLAGS) $(INCDIR) $(LIBDIR) -o $@ $(OBJS) $(LDFLAGS)

$(OBJS): %.o: %.cpp
    $(CC) $(CFLAGS) $(DFLAGS) $(INCDIR) -c $< -o $@

clean:
    rm -f $(OBJS)
    rm -f $(TARGET)
```

Annexe B

Packet Switch : extrait d'une trace de simulation avec P_{12}

Nous rappelons que, grâce aux propriétés P_{11} et P_{12} (cf. chapitre 7, section 7.1.4), nous avons pu identifier une erreur dans la description SystemC du *switch* : les paquets à destination du port 0 peuvent être perdus car ils sont malencontreusement effacés de l'anneau de registres lors de leur écriture sur l'avant-dernier port de destination. Nous rappelons ici la propriété P_{12} :

```
vunit pkt_delivery_time_limit {
  // HDL_DECLs :
  unsigned int N = 20;
  unsigned int step = 0;
  // HDL_STMTs :
  if( switch_cntrl ) { step++; }
  // PROPERTY :
  assert
  always ( inX.write_CALL() && (inX.write.p1).dest0 ->
           new(pkt x = inX.write.p1 , unsigned int MAX = step+N)
              ( (out0.write_CALL() &&
                 (out0.write.p1).id == x.id &&
                 (out0.write.p1).data == x.data)
               before!
               (switch_cntrl && step == MAX) ) );
}
```

Dans l'extrait de trace de simulation qui suit (voir plus loin pour les explications), instrumentée avec P_{12}, les textes entre traits pointillés sont les affichages originaux de la simulation (pour aider à la compréhension, nous avons ajouté les affichages relatifs aux états des FIFOs et à l'état de l'anneau de registres du *switch*) ; les affichages commençant par "`===>`" sont provoqués par nos moniteurs de surveillance.

```
.........................
1800 ns
New Packet Sent
Destination Addresses: 0 1
Packet Value: -126
Sender ID: 2
.........................
 ===>  1800 ns: pkt_delivery_time_limit_before_8  Valid   = TRUE   <===
 ===>  1800 ns: pkt_delivery_time_limit_before_8  Pending = TRUE   <===
```

```
                switch_cntrl = 0
                inX__write_CALL = 1
                out0__write_CALL = 0
                inX__write__p1 = 2::-126
                x = 2::-126
                N = 20
                step = 30
                MAX = 50

                    ..........................
                    1800 ns
                    FIFO q2_in
                    Incoming Packet = 2::-126
                    ..........................
                    ..........................
                    1800 ns
                    FIFO q2_in
                    Outcoming Packet = 2::-126
                    ..........................

                    ..........................
                    1840 ns
                    Switch Ring Rotation
                    Content = [ ] [2::-126] [ ] [0::118]
                    ..........................
===>  1840 ns: pkt_delivery_time_limit_before_8  Valid   = TRUE   <===
===>  1840 ns: pkt_delivery_time_limit_before_8  Pending = TRUE   <===
                switch_cntrl = 1
                inX__write_CALL = 0
                out0__write_CALL = 0
                x = 2::-126
                N = 20
                step = 31
                MAX = 50

                    ..........................
                    1840 ns
                    FIFO q1_out
                    Incoming Packet = 2::-126
                    ..........................
                    ..........................
                    1840 ns
                    FIFO q1_out
                    Outcoming Packet = 2::-126
                    ..........................
                                            ..........................
                                            1840 ns
                                            New Packet Received
                                            Receiver ID: 1
                                            Packet Value: -126
                                            Sender ID: 2
                                            ..........................
                    ..........................
                    1900 ns
                    Switch Ring Rotation
                    Content = [ ] [ ] [0::118] [ ]
                    ..........................
```

```
===>  1900 ns: pkt_delivery_time_limit_before_8  Valid   = TRUE   <===
===>  1900 ns: pkt_delivery_time_limit_before_8  Pending = TRUE   <===
               switch_cntrl = 1
               inX__write_CALL = 0
               out0__write_CALL = 0
               x = 2::-126
               N = 20
               step = 32
               MAX = 50

                   ........................
                   1960 ns
                   Switch Ring Rotation
                   Content = [1::-68] [ ] [3::91] [0::-124]
                   ........................
===>  1960 ns: pkt_delivery_time_limit_before_8  Valid   = TRUE   <===
===>  1960 ns: pkt_delivery_time_limit_before_8  Pending = TRUE   <===
               switch_cntrl = 1
               inX__write_CALL = 0
               out0__write_CALL = 0
               x = 2::-126
               N = 20
               step = 33
               MAX = 50

* * *
                   ........................
                   2980 ns
                   Switch Ring Rotation
                   Content = [1::9] [] [2::-77] [0::61]
                   ........................
===>  2980 ns: pkt_delivery_time_limit_before_8  Valid   = FALSE  <===
===>  2980 ns: pkt_delivery_time_limit_before_8  Pending = TRUE   <===
               switch_cntrl = 1
               inX__write_CALL = 0
               out0__write_CALL = 0
               x = 2::-126
               N = 20
               step = 50
               MAX = 50

      /!\  Property pkt_delivery_time_limit_before_8 VIOLATION  /!\
           at 2980 ns
```

À 1800ns , deux paquets sont envoyés respectivement par les entrées 0 (avec la valeur 118) et 2 (avec la valeur -126) ; seulement le paquet envoyé par le port 2 est affiché sur la trace montrée ici. Ce paquet est à destination des ports 1 et 0. Au moment où ce paquet est écrit sur le port d'entrée 2, une nouvelle instance de l'opérateur **`before`**! (la sous-propriété sous la portée du **`new`**, nommée `pkt_delivery_time_limit_before_8`) est créée et démarrée. La variable locale `x` contient le nouveau paquet émis et la variable `MAX` possède la valeur de `step+N`, i.e. 50. Les deux paquets reçus par le *switch* à 1800ns sont directement copiés dans les FIFOs d'entrée, et depuis ces FIFOs ils sont immédiatement copiés dans les registres correspondants de l'anneau interne. Ceci est possible puisque les deux FIFOs et les registres sont vides. Au prochain front montant de l'horloge du *switch*, à 1840ns, une rotation de l'anneau est effectuée et le paquet `2::-126` est déplacé dans le registre 1,

associé au nœud correspondant. De la même façon, le paquet envoyé par 0 est déplacé dans le registre 3. Le moniteur est alors déclenché (la propriété est "sensible", entre autres, au signal `switch_cntrl` qui a un front montant) et le nombre de pas (compteur `step`), c'est-à-dire le nombre de rotations de l'anneau de registres, est incrémenté. Le paquet `2::-126` est ensuite copié dans la FIFO de sortie 1, parce que cette sortie fait partie de ses destinataires. La trace de simulation montre que ce paquet traverse la FIFO et atteint le port de sortie. Nous pouvons alors remarquer que le paquet est malencontreusement effacé du registre, bien qu'il possède encore le port 0 comme destinataire : lors du prochain front montant de l'horloge (1900ns), le paquet `0::118` est une fois de plus déplacé dans l'anneau de registres, mais le paquet `2::-126` a disparu, alors qu'il aurait dû se trouver maintenant dans le registre 0. Lors du front montrant suivant de l'horloge (1960ns), trois nouveaux paquets accèdent au *switch*. Dans la suite de la simulation, d'autres paquets seront envoyés à destination du port 0, mais le paquet `2::-126`, effacé par erreur, ne sera jamais délivré. À 2980ns, l'instance du moniteur relative à ce paquet est évaluée pour la dernière fois : la condition `switch_cntrl && step == MAX` est satisfaite et le moniteur signale une violation, juste avant d'être arrêté.

Annexe C

Modules d'ISIS

ISIS est un logiciel écrit en Java, il contient plus de 150 classes, pour environ 60000 lignes de code.

La figure C.1 de la page 172 illustre le fonctionnement de ses différentes parties logicielles. La partie A se charge de l'analyse syntaxique des propriétés PSL et de celle de la partie déclarative de la couche modélisation. Chaque *vunit* est ainsi traduite en des représentations internes arborescentes. Un moteur de génération produit la description C++ des moniteurs de surveillance à partir de cette représentation interne.

La partie B s'occupe de l'analyse syntaxique du code source du DUV. Comme expliqué dans le chapitre 6, l'outil gccxml[1] est utilisé pour générer une première représentation XML non hiérarchisée des composants et des méthodes de communication du design. L'analyseur XML d'ISIS utilise cette représentation pour produire un deuxième fichier contenant une version réduite et hiérarchisée de la structure du design.

Enfin, la partie C utilise les deux sorties produites par les modules A et B, ainsi que le choix d'instrumentation de l'utilisateur (i.e. mise en correspondance entre les identifiants dans les propriétés et les composants du design) pour générer la version instrumentée du DUV, c'est-à-dire l'ensemble des fichiers sources du design, éventuellement adaptés, et des fichiers contenant les moniteurs, leurs *wrappers* et les sous-classes observées. Cette nouvelle version peut être compilée en utilisant un compilateur C++ tel que g++. L'application ainsi obtenue est prête à être exécutée.

Il convient de remarquer que les modules A, B et C et les moteurs de génération qu'ils comportent sont communs aux deux modes de fonctionnement d'ISIS, avec interface graphique ou en *scripting*. Dans les deux cas, la différence essentielle réside dans la façon de fournir les entrées utilisateur à l'outil (propriétés, sources du design et choix d'instrumentation).

[1] `http://www.gccxml.org`

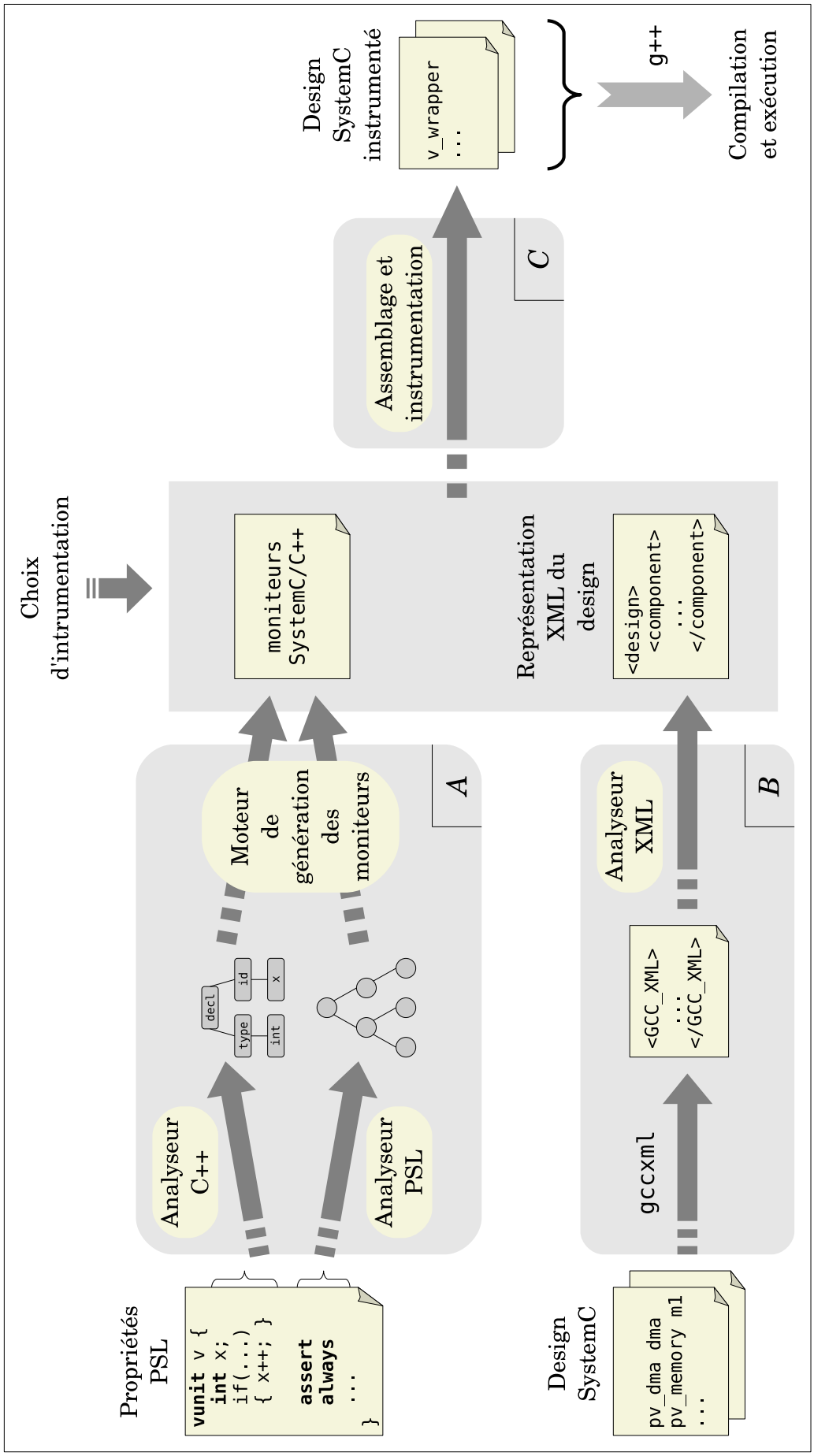

FIGURE C.1 – Modules du flot d'ISIS

Bibliographie

[ABF+06] Gadiel Auerbach, Lyes Benalycherif, Andrea Fedeli, Dana Fisman, Anthony McIsaac et Klaus Winkelmann: *Case Studies in Property-Based Requirements Specification.* PROSYD (Property-Based System Design) Project, 2006. http://www.prosyd.org/twiki/pub/Public/DeliverablePageWP1/prosyd1.4_1revision

[ABG+00] Yael Abarbanel, Ilan Beer, Leonid Glushovsky, Sharon Keidar et Yaron Wolfsthal: *FoCs: Automatic Generation of Simulation Checkers from Formal Specifications.* Dans *Computer Aided Verification (CAV)*, pages 538–542, 2000.

[Acc10] Accellera: *Universal Verification Methodology (UVM) 1.0 EA User's Guide.* Accellera, May 2010. http://www.accellera.org/activities/vip/uvm1.0ea.tar.gz.

[AFF+02] Roy Armoni, Limor Fix, Alon Flaisher, Rob Gerth, Boris Ginsburg, Tomer Kanza, Avner Landver, Sela Mador-Haim, Eli Singerman, Andreas Tiemeyer, Moshe Vardi et Yael Zbar: *The ForSpec Temporal Logic: A New Temporal Property-Specification Language.* Dans Joost Pieter Katoen et Perdita Stevens (rédacteurs): *Tools and Algorithms for the Construction and Analysis of Systems*, tome 2280 de *Lecture Notes in Computer Science*, pages 1–30. Springer Berlin / Heidelberg, 2002.

[Aho90] Alfred V. Aho: *Algorithms for finding patterns in strings.* Handbook of theoretical computer science (vol. A): algorithms and complexity, pages 255–300, 1990. MIT Press (Cambridge, MA, USA).

[Avi10] Ran Avinun: *Validate hardware/software for nextgen mobile/consumer apps using software-on-chip system development tools.* EETimes, December 2010. http://www.eetimes.com/design/embedded/4211507/Validate-hardware-software-for-nextgen-mobile-consumer-apps-using-software-on-chip-system-development-tools-.

[BAGK07] Mark Burton, James Aldis, Robert Günzel et Wolfgang Klingauf: *Transaction Level Modelling: A reflection on what TLM is and how TLMs may be classified.* Dans *FDL '07: Proceedings of the 2007 Forum on Specification & Design Languages*, September 2007.

[Bal06] Aniruddha Baljekar: *Dynamic Assertion Based Verification using PSL and OVL.* Design&Reuse Industry Articles, 2006. http://www.design-

reuse.com/articles/12947/dynamic-assertion-based-verification-using-psl-and-ovl.html.

[BAPM83] Mordechai Ben-Ari, Amir Pnueli et Zohar Manna: *The temporal logic of branching time*. Acta Informatica, 20(3):207–226, 1983.

[BBDE+01] Ilan Beer, Shoham Ben-David, Cindy Eisner, Dana Fisman, Anna Gringauze et Yoav Rodeh: *The Temporal Logic Sugar*. Dans Gerard Berry, Hubert Comon et Alain Finkel (rédacteurs): *CAV '01: Proceedings of the 13th international conference on Computer Aided Verification*, tome 2102 de *Lecture Notes in Computer Science*, pages 363–367. Springer Berlin / Heidelberg, 2001.

[BD04] David C. Black et Jack Donovan: *SystemC: From the Ground Up*. Kluwer Academic Publishers, 2004.

[BFG10] Nicola Bombieri, Franco Fummi et Valerio Guarnieri: *Automatic synthesis of OSCI TLM-2.0 models into RTL bus-based IPs*. Dans *HLDVT '10: Proceedings of the 2010 IEEE International High Level Design Validation and Test Workshop*, pages 105–112, June 2010.

[BFM+06] Lyes Benalycherif, Andrea Fedeli, Anthony McIsaac, Mark Moulin, Ohad Shacham, Klaus Winkelmann et Emmanuel Zarpas: *Case Studies in Property-Based Verification*. PROSYD (Property-Based System Design) Project, 2006. http://www.prosyd.org/twiki/pub/Public/DeliverablePageWP3/prosyd3.4_1rev1.1.pdf.

[BFP06] Nicola Bombieri, Franco Fummi et Graziano Pravadelli: *On the evaluation of transactor-based verification for reusing TLM assertions and testbenches at RTL*. Dans *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 1007–1012, European Design and Automation Association, 3001 Leuven, Belgium, March 2006. European Design and Automation Association.

[BH06] Doron Bustan et John Havlicek: *Some Complexity Results for SystemVerilog Assertions*. Dans *CAV '06: Proceedings of the 2006 international conference on Computer Aided Verification*, tome 4144, pages 205–218. Springer, 2006.

[BHA10] Zeineb Bel-Hadj-Amor: *Vers un outil d'aide à la spécification formelle des propriétés transactionnelles*, Juin 2010. Mémoire de Master 2 - Université Joseph Fourier.

[BKS08] Nicolas Blanc, Daniel Kroening et Natasha Sharygina: *Scoot: A Tool for the Analysis of SystemC Models*. Dans *TACAS '08: Proceedings of the Fourteenth International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, tome 4963 de *Lecture Notes in Computer Science*, pages 467–470. Springer, 2008.

[BLMA+05] Dominique Borrione, Miao Liu, Katell Morin-Allory, Pierre Ostier et Laurent Fesquet: *On-line assertion-based verification with a proven correct library of monitors*. Dans *ICICT'05: Proceedings of the 3rd International Conference on Information and Communication Technology*, pages 125–144, December 2005.

[BLOF06] Dominique Borrione, Miao Liu, Pierre Ostier et Laurent Fesquet: *PSL-Based Online Monitoring of Digital Systems*. Dans *Applications of Specifi-*

cation and Design Languages for SoCs, pages 5–22. Springer Netherlands, 2006.

[BM10] Brian Bailey et Grant Martin: *ESL Models and their Application - Electronic System Level Design and Verification in Practice*. Springer, 2010.

[Brz64] Janusz A. Brzozowski: *Derivatives of Regular Expressions*. Journal of the ACM, 11(4):481–494, 1964.

[BZ05] Marc Boulé et Zeljko Zilic: *Incorporating Efficient Assertion Checkers into Hardware Emulation*. Dans *ICCD '05: Proceedings of the 23rd IEEE International Conference on Computer Design*, pages 221–228, Washington, DC, USA, 2005. IEEE Computer Society.

[BZ06] Marc Boulé et Zeljko Zilic: *Efficient Automata-Based Assertion-Checker Synthesis of PSL Properties*. Dans *HLDVT '06: Proceedings of the 2006 IEEE International High Level Design Validation and Test Workshop*, pages 69–76, 2006.

[BZ07] Marc Boulé et Zeljko Zilic: *Efficient Automata-Based Assertion-Checker Synthesis of SEREs for Hardware Emulation*. Dans *ASP-DAC '07: Proceedings of the 12th Asia and South Pacific Design Automation Conference*, pages 324–329, Washington, DC, USA, 2007. IEEE Computer Society.

[CCG+02] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani et Armando Tacchella: *NuSMV 2: An OpenSource Tool for Symbolic Model Checking*. Dans *CAV'02: In Proceeding of International Conference on Computer-Aided Verification*, Lecture Notes in Computer Science, pages 241–268, 2002.

[CDH+00] James Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach et Hongjun Zheng: *Bandera: Extracting Finite-state Models from Java Source Code*. Dans *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.

[CEP00] Luis Alejandro Cortés, Petru Eles et Zebo Peng: *Verification of embedded systems using a Petri net based representation*. Dans *ISSS '00: Proceedings of the 13th international symposium on System synthesis*, pages 149–155. IEEE Computer Society, 2000.

[CF06] Ping Hang Cheung et Alessandro Forin: *A C-language binding for PSL*. Rapport technique MSR-TR-2006-131, Microsoft Research, Redmond, WA 98052, September 2006. http://research.microsoft.com/research/EmbeddedSystems/Giano/TR-2006-131.pdf.

[CG03] Lukai Cai et Daniel Gajski: *Transaction Level Modeling: an Overview*. Dans *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 19–24. ACM, 2003.

[CGP00] Edmund M. Clarke, Orna Grumberg et Doron A. Peled: *Model Checking*. The MIT Press, 2000.

[CL02] Yoonsik Cheon et Gary T. Leavens: *A Simple and Practical Approach to Unit Testing: The JML and JUnit Way*. Dans *ECOOP '02: Proceedings of*

the 16th European Conference on Object-Oriented Programming, tome 2374 de *Lecture Notes in Computer Science*. Springer, 2002.

[CM88] Kanianthra Mani Chandy et Jayadev Misra: *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[CM04] Koen Claessen et Johan Mårtensson: *An Operational Semantics for Weak PSL*. Dans *FMCAD '04: Proceedings of the 2004 International Conference on Formal Methods in Computer-Aided Design*, tome 3312/2004, pages 337–351, November 2004.

[Cor08] Jérôme Cornet: *Separation of Functional and Non-Functional Aspects in Transactional Level Models of Systems-on-Chip*. Thèse de doctorat, Grenoble INP, April 25 2008.

[Cor11] Intel Corporation: *Intel repère une erreur de conception pour un de ses chipsets et met en place une solution*. Intel website, january 2011. http://www.intel.com/cd/corporate/pressroom/emea/fra/468493.htm.

[CPP03] *ISO/IEC 14882:2003 International Standard - Programming Languages - C++*, October 2003.

[CVK04] Ben Cohen, Srinivasan Venkataramanan et Ajeetha Kumari: *Using PSL/-Sugar for Formal and Dynamic Verification 2nd Edition: Guide to Property Specification Language for Assertion-Based Verification*. VhdlCohen Publishing, 2004.

[Das06] Pallab Dasgupta: *A Roadmap for Formal Property Verification*. Springer-Verlag New York, Inc., 2006.

[dBLM+04] Lydie du Bousquet, Yves Ledru, Olivier Maury, Catherine Oriat et Jean Louis Lanet: *A Case Study in JML-Based Software Validation*. Dans *ASE '04: Proceedings of the 19th IEEE international conference on Automated Software Engineering*, pages 294–297. IEEE Computer Society, 2004.

[DCdBBL05] Sophie Dupuy-Chessa, Lydie du Bousquet, Jullien Bouchet et Yves Ledru: *Test of the ICARE Platform Fusion Mechanism*. Dans *DSVIS '05: Proceedings of the 2005 International Workshop on Design Specification and Verification of Interactive Systems*, tome 3941 de *Lecture Notes in Computer Science*, pages 102–113, 2005.

[DG02] Rolf Drechsler et Daniel Große: *Reachability Analysis for Formal Verification of SystemC*. Dans *DSD '02: Proceedings of the Euromicro Symposium on Digital Systems Design*, pages 337–340, 2002.

[DG05] Rolf Drechsler et Daniel Große: *System level validation using formal techniques*. Dans *IEE Proceedings - Computers and Digital Techniques*, tome 152, pages 393–406, 2005.

[DGG+05] Anat Dahan, Daniel Geist, Leonid Gluhovsky, Dmitry Pidan, Gil Shapir, Yaron Wolfsthal, Lyes Benalycherif, Romain Kamdem et Younes Lahbib: *Combining System Level Modeling with Assertion Based Verification*. Dans *ISQED '05: Proceedings of the 6th International Symposium on Quality of Electronic Design*, pages 310–315. IEEE Computer Society, 2005.

[Dou] Doulos: *SVA Properties for pipelined protocols*. Doulos Website. http://www.doulos.com/knowhow/sysverilog/overlapping_sva.

[dSS09] Antonio da Silva et Sebastian Sanchez: *Transactions Sequence Tracking by means of Dynamic Binary Instrumentation of TLM Models*. Dans *DSD '09: Proceedings of the 12th Euromicro Conference on Digital System Design / Architectures, Methods and Tools*, pages 723–728, August 2009.

[Dub05] Rohit Dubey: *Elements of Verification*. SOC Central, March 2005. http://www.soccentral.com/results.asp?CatID=488&EntryID=12420.

[EC80] E. Allen Emerson et Edmund M. Clarke: *Characterizing Correctness Properties of Parallel Programs Using Fixpoints*. Dans *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 169–181. Springer-Verlag, 1980.

[Eck03] Bruce Eckel: *Thinking in C++: Introduction to Standard C++, Volume One*. 2003. http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html.

[EEH+06a] Wolfgang Ecker, Volkan Esen, Michael Hull, Thomas Steininger et Michael Velten: *Execution semantics and formalisms for multi-abstraction TLM assertions*. Dans *MEMOCODE '06: Proceedings of the 4th ACM/IEEE International Conference on Formal Methods and Models for Co-Design*, pages 93–102, July 2006.

[EEH+06b] Wolfgang Ecker, Volkan Esen, Michael Hull, Thomas Steininger et Michael Velten: *Specification Language for Transaction Level Assertions*. Dans *HLDVT '06: Proceedings of the 11thIEEE International High-Level Design Validation and Test Workshop*, pages 77–84, November 2006.

[EEH+07] Wolfgang Ecker, Volkan Esen, Michael Hull, Thomas Steininger et Michael Velten: *Implementation of a Transaction Level Assertion Framework in SystemC*. Dans *DATE '07: Proceedings of the 2007 conference on Design, Automation and Test in Europe*, pages 894–899, 2007.

[EESV07] Wolfgang Ecker, Volkan Esen, Thomas Steininger et Michael Velten: *Requirements and Concepts for Transaction Level Assertion Refinement*. Dans Springer Boston (éditeur): *Embedded System Design: Topics, Techniques and Trends*, tome 231/2007 de *IFIP International Federation for Information Processing*, pages 1–14, 2007.

[EF08] Cindy Eisner et Dana Fisman: *Augmenting a Regular Expression-Based Temporal Logic with Local Variables*. Dans *FMCAD '08: Proceedings of the 8th conference on Formal Methods in Computer-Aided Design*, pages 1–8, November 2008.

[EH86] E. Allen Emerson et Joseph Y. Halpern: *"Sometimes" and "not never" revisited: on branching versus linear time temporal logic*. Journal of the ACM (JACM), 33(1):151–178, 1986.

[Eis07] Cindy Eisner: *PSL for Runtime Verification: Theory and Practice*. Dans *RV '07: Proceedings of the 7th Workshop on Runtime Verification*, pages 1–8. Springer Berlin / Heidelberg, 2007.

[ES84] E. Allen Emerson et A. Prasad Sistla: *Deciding Full Branching Time Logic*. Information and Control, 61(3):175–201, 1984.

[FC01] Harry Foster et Claudionor Coelho: *Assertions Targeting A Diverse Set of Verification Tools*. Dans *HDLCon'01: Proceedings of the 10th annual international HDL Conference*, March 2001.

[FHLM02] Navit Fedida, John Havlicek, Nissan Levi et Hillel Miller: *CBV Semantics (Draft)*, January 2002. http://www.vhdl.org/vfv/hm/att-0672/01-cbv_semantics.pdf.

[FKL03] Harry Foster, Adam Krolnik et David Lacey: *Assertion-Based Design*. Kluwer Academic Publishers, 2003.

[FLT06] Harry Foster, Kenneth Larsen et Mike Turpin: *Introduction to the New Accellera Open Verification Library*. Dans *DVCon'06: Proceedings of the Design and Verification Conference and exhibition*, 2006. http://www.vhdl.org/ovl/pages/pdfs/dvcon06_foster.pdf.

[FMW05] Harry Foster, Erich Marschner et Yaron Wolfsthal: *IEEE 1850 PSL: The Next Generation*. Dans *DVCon '05: Design and Verification Conference and exhibition*, 2005.

[FNL06] Alessandro Forin, Behnam Neekzad et Nathaniel Lynch: *Giano: The Two-Headed System Simulator*. Rapport technique MSR-TR-2006-130, Microsoft Research, Redmond, WA 98052, September 2006. http://research.microsoft.com/research/EmbeddedSystems/Giano/TR-2006-130.pdf.

[Fos09] Harry Foster: *Applied Assertion-Based Verification: An Industry Perspective*. Foundations and Trends in Electronic Design Automation, 3(1):1–95, 2009.

[FP10a] Luca Ferro et Laurence Pierre: *Formal Semantics for PSL Modeling Layer and Application to the Verification of Transactional Models*. Dans *DATE '10: Proceedings of the 2010 Conference & Exhibition on Design, Automation & Test in Europe*, pages 1207–1212, March 2010.

[FP10b] Luca Ferro et Laurence Pierre: *ISIS: Runtime Verification of TLM Platforms*. Dans Dominique Borrione (éditeur): *Advances in Design Methods from Modeling Languages for Embedded Systems and SoC's*, tome 63 de *Lecture Notes in Electrical Engineering*, chapitre 13, pages 213–226. Springer, 2010.

[FPA+11] Luca Ferro, Laurence Pierre, Zeineb Bel Hadj Amor, Jérôme Lachaize et Vincent Lefftz: *Runtime Verification of Typical Requirements for a Space Critical SoC Platform*. Dans *FMICS '11: Proceedings of the 16th International Workshop on Formal Methods for Industrial Critical Systems*, August 2011.

[FPLdB08] Luca Ferro, Laurence Pierre, Yves Ledru et Lydie du Bousquet: *Generation of Test Programs for the Assertion-Based Verification of TLM Models*. Dans *IDT '08: Proceedings of the 3rd IEEE International Design and Test Workshop*, pages 237–242, December 2008.

[GBA+99] Daniel Geist, Giora Biran, Tamara Arons, Michael Slavkin, Yvgeny Nustov, Monica Farkas, Karen Holtz, Andy Long, Dave King et Steve Barret: *A methodology for the verification of a system on chip*. Dans *DAC '99: Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, pages 574–579. ACM, 1999.

[GCMC+05] Frank Ghenassia, Alain Clouard, Laurent Maillet-Contoz, Jean Philippe Strassen, Eric Paire, Thibaut Bultiaux, Stephane Guenot, Serge Hustin,

Alexandre Blampey, Joseph Bulone, Matthieu Moy, Antoine Perrin, Gregory Poivre, Christophe Amerijckx et Amine Kerkeni: *Transaction-Level Modeling with Systemc: TLM Concepts and Applications for Embedded Systems*. Springer, 2005.

[GD03] Daniel Große et Rolf Drechsler: *Formal Verification of LTL Formulas for SystemC Designs*. Dans *ISCAS '03: Proceedings of the 2003 International Symposium on Circuits and Systems*, tome 5, pages 245–248, May 2003.

[GD04] Daniel Große et Rolf Drechsler: *Checkers for SystemC designs*. Dans *MEMOCODE '04: Proceedings of the 2nd ACM and IEEE International Conference on Formal Methods and Models for Co-Design*, pages 171–178, June 2004.

[Ger09] Patrice Gerin: *Modèles de simulation pour la validation logicielle et l'exploration d'architectures des systèmes multiprocesseurs sur puce*. Thèse de doctorat, Grenoble INP, Novembre 2009.

[GGP08] Patrice Gerin, Xavier Guérin et Frédéric Pétrot: *Efficient implementation of native software simulation for MPSoC*. Dans *DATE '08: Proceedings of the conference on Design, Aautomation and Test in Europe*, pages 676–681. ACM, 2008.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson et John M. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GHMZ08] Maziar Goudarzi, Shaahin Hessabi, Naser MohammadZadeh et Nasim Zainolabedini: *The ODYSSEY approach to early simulation-based equivalence checking at ESL level using automatically generated executable transaction-level model*. Elsevier Journal of Microprocessors and Microsystems (now J. Embedded Hardware Design, JEHD), 32(7):364–374, 2008.

[GLMS02] Thorsten Grötker, Stan Liao, Grant Martin et Stuart Swan: *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[GLS99] William Gropp, Ewing Lusk et Anthony Skjellum: *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1999. ISBN 0-262-57133-1.

[Gro03] SystemC Verification Working Group: *SystemC Verification Standard Specification - Version 1.0e*. www.systemc.org, May 2003.

[GYHG07] Amir Masoud Gharehbaghi, Benyamin Hamdin Yaran, Shaahin Hessabi et Maziar Goudarzi: *An assertion-based verification methodology for system-level design*. Computers and Electrical Engineering, 33(4):269–284, 2007.

[Hel04] Graham Hellesbrand: *La conception système s'impose à l'électronique automobile*. Electronique, 153:4, Décembre 2004. http://www.01net.com/Pdf/ELM200412010153038.pdf.

[HLMS02] John Havlicek, Nissan Levi, Hillel Miller et Kurt Shultz: *Extended CBV Statement Semantics*. Partial proposal presented to the Accellera Formal Verification Technical Committee, April 2002. http://www.eda.org/vfv/hm/att-0772/01-ecbv_statement_semantics.ps.gz.

[HT05] Ali Habibi et Sofiene Tahar: *Design for Verification of SystemC Transaction Level Models*. Dans *DATE '05: Proceedings of the 2005 conference on Design, Automation and Test in Europe*, pages 560–565, 2005.

[HT06] Ali Habibi et Sofiene Tahar: *Design and Verification of SystemC Transaction-Level Models*. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 14(1):57–68, January 2006.

[HW05] John Havlicek et Yaron Wolfsthal: *PSL and SVA: Two Standard Assertion Languages Addressing Complimentary Engineering Needs*. Dans *DVCon '05: Proceedings of the 2005 Design and Verification Conference and exhibition*, February 2005.

[IS03] C. Norris Ip et Stuart Swan: *A Tutorial Introduction on the New SystemC Verification Standard*. www.systemc.org - white paper, 2003.

[Jha05] Dipak Jha: *Use reentrant functions for safer signal handling*. IBM webpages, January 2005. http://www.ibm.com/developerworks/linux/library/l-reent.html.

[JJ05] Claude Jard et Thierry Jéron: *TGV: theory, principles and algorithms. A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems*. International Journal on Software Tools for Technology Transfer (STTT) - Special section on high-level test of complex systems, 7(4):297–315, August 2005.

[KEP06] Daniel Karlsson, Petru Eles et Zebo Peng: *Formal Verification of SystemC Designs Using a Petri-Net Based Representation*. Dans *DATE '06: Proceedings of the 2006 conference on Design, Automation and Test in Europe*, pages 1228–1233, 2006.

[KH09] Cédric Koch-Hofer: *Modélisation, Validation et Présynthèse de Circuits Asynchrones en SystemC*. Thèse de doctorat, Institut Polytechnique de Grenoble, Mars 2009.

[Kir03] Zeev Kirshenbaum: *Understanding the "e" verification language*. EETimes, May 2003. http://www.us.design-reuse.com/articles/article5646.html.

[Kle56] Stephen Cole Kleene: *Representation of Events in Nerve Nets and Finite Automata*, pages 3–41. Princeton University Press, 1956.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean Marc Loingtier et John Irwin: *Aspect-Oriented Programming*. Dans Mehmet Aksit et Satoshi Matsuoka (rédacteurs): *ECOOP'97: Proceedings of the European Conference on Object-Oriented Programming*, tome 1241 de *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin/Heidelberg, 1997.

[KLTB10] Meriam Kallel, Younes Lahbib, Rached Tourki et Adel Baganne: *Verification of SystemC Transaction Level Models using an Aspect-Oriented and Generic Approach*. Dans *DTIS '10: Proceedings of the 5th International Conference on Design and Technology of Integrated Systems in Nanoscale Era*, pages 1–6, March 2010.

[KS05] Daniel Kroening et Natasha Sharygina: *Formal verification of SystemC by automatic hardware/software partitioning*. Dans *MEMOCODE '05: Proceedings of the Third ACM and IEEE International Conference on Formal*

Methods and Models for Co-Design, pages 101–110. IEEE, July 11-14 2005. 2007.03.22.

[KT07] Atsushi Kasuya et Tesh Tesfaye: *Verification Methodologies in a TLM-to-RTL Design Flow*. Dans *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 199–204. ACM, 2007.

[KTZ06] Atsushi Kasuya, Tesh Tesfaye et Eugene Zhang: *Native SystemC Assertion mechanism with transaction and temporal assertion support*. EDA Tech Forum Journal, September 2006. http://www.jedatechnologies.net/base/files/articles/ 0609_Native_SystemC_Assertion_mechanism_with_transaction_and_temporal_assertion_

[Lah06] Younes Lahbib: *Extension of Assertion-Based Verification Approaches for the Verification of SystemC SoC Models*. Thèse de doctorat, University of Monastir (Tunisia), December 2006.

[LDdB+07] Yves Ledru, Frédéric Dadeau, Lydie du Bousquet, Sébastien Ville et Elodie Rose: *Mastering combinatorial explosion with the tobias-2 test generator*. Dans *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated Software Engineering*, pages 535–536. ACM, 2007.

[LGH+06] Younes Lahbib, Mohamed Arafet Ghrab, Maher Hechkel, Frank. Ghenassia et Rached Tourki: *A new synchronization policy between PSL checkers and SystemC designs at transaction level*. Dans *DTIS '06: Proceedings of the 2006 International Conference on Design and Test of Integrated Systems in Nanoscale Technology*, pages 85–90, September 2006.

[LS07] Jiang Long et Andrew Seawright: *Synthesizing SVA Local Variables for Formal Verification*. Dans *DAC '07: Proceedings of the 44th ACM/IEEE Design Automation Conference*, pages 75–80, June 2007.

[MAB06a] Katell Morin-Allory et Dominique Borrione: *On-line monitoring of properties built on regular expressions*. Dans *FDL '06: In Proceeding of the 2006 Forum on specification and Design Languages*, 2006.

[MAB06b] Katell Morin-Allory et Dominique Borrione: *Proven correct monitors from PSL specifications*. Dans *DATE '06: Proceedings of the 2006 conference on Design, Automation and Test in Europe*, pages 1246–1251, 2006.

[Mal02] David Maliniak: *Assertion-Based Verification Smoothes The Road To IP Reuse*. Electronic Design, September 2002. http://www.elecdesign.com/Articles/ArticleID/2748/2748.html.

[Mar97] Grant Martin: *HW-SW Co-Design: A Perspective*. EDA Vision, Volume 1, Number 1, October 1997.

[Mar98] Grant Martin: *Design methodologies for system level IP*. Dans *DATE '98: Proceedings of the conference on Design, Automation and Test in Europe*, pages 286–289, 1998.

[McM93] Kenneth L. McMillan: *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[McM00] Kenneth L. McMillan: *A methodology for hardware verification using compositional model checking*. Science of Computer Programming, 37(1-3):279–309, May 2000.

[Mit96] John C. Mitchell: *Foundations for Programming Languages*. The MIT Press, September 1996.

[MMMC06] Matthieu Moy, Florence Maraninchi et Laurent Maillet-Contoz: *LusSy: an open tool for the analysis of systems-on-a-chip at the transaction level*. Design Automation for Embedded Systems, 10:73–104, 2006.

[Moo65] Gordon Earle Moore: *Cramming more components onto integrated circuits*. Electronics, 38(8), April 19 1965.

[MP92] Zohar Manna et Amir Pnueli: *The temporal logic of reactive and concurrent systems*. Springer-Verlag, 1992.

[MPI] *MPICH2 Website*. http://www.mcs.anl.gov/research/projects/mpich2.

[Mur89] Tadao Murata: *Petri nets: Properties, Analysis and Applications*. Proceedings of the IEEE, 77(4):541–580, April 1989.

[NH06] B. Niemann et C. Haubelt: *Assertion-Based Verification of Transaction Level Models*. Dans *Proceedings of the 2006 ITG/GI/GMM Workshop*, tome 9, pages 232–236, February 2006.

[NN99] Hanne Riis Nielson et Flemming Nielson: *Semantics with Application: A Formal Introduction*. Wiley Professional Computing, Jonh Wiley & Sons, 1999.

[Odd09] Yann Oddos: *Vérification semi-formelle et synthèse automatique de circuits à partir de spécifications temporelles décrites en PSL*. Thèse de doctorat, University of Grenoble, November 2009. http://tel.archives-ouvertes.fr/docs/00/48/19/23/PDF/sfv_0318.pdf.

[OH02] Marcio T. Oliveira et Alan J. Hu: *High-Level specification and automatic generation of IP interface monitors*. Dans *DAC '02: Proceedings of the 39th conference on Design Automation*, pages 129–134, New York, NY, USA, 2002. ACM.

[OMAB06] Yann Oddos, Katell Morin-Allory et Dominique Borrione: *On-Line Test Vector Generation from Temporal Constraints Written in PSL*. Dans *VLSI-SOC '06: Proceedings of the 2006 IFIP International Conference on Very Large Scale Integration*, pages 397–402, October 2006.

[OMAB08] Yann Oddos, Katell Morin-Allory et Dominique Borrione: *Assertion-Based Design with Horus*. Dans *MEMOCODE '08: Proceedings of the 6th ACM/IEEE International Conference on Formal Methods and Models for Co-Design*, pages 75–76, June 2008.

[OMG09] OMG: *OMG Unified Modeling Language (OMG UML), Superstructure*, February 2009. `http://www.omg.org/spec/UML/2.2`, Version 2.2.

[OSC09] OSCI: *OSCI TLM 2.0 Language Reference Manual*, July 2009. http://www.systemc.org/members/download_files/check_file?agreement=tlm_2-0_lrm, version 2.0.1.

[OSR92] Sam Owre, Natarajan Shankar et John Rushby: *PVS: A Prototype Verification System*. Dans Deepak Kapur (éditeur): *CADE '92: Proceedings of the 11th International Conference on Automated Deduction*, tome 607 de *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, june 1992.

[OVL10] *Accellera Standard OVL V2 - Library Reference Manual*, July 2010.

[OVM] *Open Verification Methodology website*. `http://www.ovmworld.org`.

[PF08] Laurence Pierre et Luca Ferro: *A Tractable and Fast Method for Monitoring SystemC TLM Specifications*. IEEE Transactions on Computers, 57(10):1346–1356, October 2008.

[PF10] Laurence Pierre et Luca Ferro: *Enhancing the Assertion-Based Verification of TLM Designs with Reentrancy*. Dans *MEMOCODE '10: Proceedings of the 8th ACM/IEEE International Conference on Formal Methods and Models for Codesign*, 2010.

[Pie07] Laurence Pierre: *A Model for Assertion-Based Verification of TLM Designs*. Research Report TIMA-RR–07/09-01–FR, TIMA Laboratory, 2007.

[PLBN05] Michael Pellauer, Mieszko Lis, Don Baltus et Rishiyur Nikhil: *Synthesis of synchronous assertions with guarded atomic actions*. Dans *MEMOCODE '05: Proceedings of the 2nd ACM/IEEE International Conference on Formal Methods and Models for Co-Design*, pages 15–24. IEEE Computer Society, 2005.

[Pnu77] Amir Pnueli: *The temporal logic of programs*. Dans *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society, 1977.

[PR09] Dmitry Pidan et Sitvanit Ruah: *Design verification using directives having local variables*. US Patent Application 2009/0216513 A1, August 2009.

[PSL05] *IEEE Std 1850-2005, IEEE Standard for Property Specification Language (PSL)*, October 2005.

[PSL10] *IEEE Std 1850-2010, IEEE Standard for Property Specification Language (PSL)*, April 2010.

[RHKR01] Jürgen Ruf, Dirk W. Hoffmann, Thomas Kropf et Wolfgang Rosenstiel: *Simulation-guided property checking based on a multi-valued AR-automata*. Dans *DATE '01: Proceedings of the conference on Design, Automation and Test in Europe*, pages 742–748. IEEE Press, 2001.

[SC005] *IEEE Std 1666-2005, IEEE Standard System C Language Reference Manual*, March 2005.

[SC07] Cadence Design Systems et Mentor Graphics Corporation: *Open Verification Methodology (OVM) white paper*, 2007. http://www.ovmworld.org/whitepapers/OVM_Whitepaper_12-21-07.pdf.

[Sis94] A. Prasad Sistla: *Safety, Liveness and Fairness in Temporal Logic*. Formal Aspects of Computing, 6(55):495–511, September 1994.

[Smi05] Sean Smith: *IP reuse requires a verification strategy*. EETimes, February 2005. http://www.eetimes.com/showArticle.jhtml?articleID=59302133.

[Str89] Bjarne Stroustrup: *Multiple Inheritance for C++*. Computing Systems, 2(4):367–395, 1989.

[SV005] *IEEE Std 1800-2005, IEEE Standard for System Verilog - Unified Hardware Design, Specification and Verification Language*, 2005.

[SV009] *IEEE Std 1800-2009, IEEE Standard for System Verilog - Unified Hardware Design, Specification, and Verification Language*, December 2009.

[SVM01] Alberto L. Sangiovanni-Vincentelli et Grant Martin: *Platform-based design and software design methodology for embedded systems.* IEEE Design & Test of Computers, 18(6):23–33, 2001.

[Syn] Synopsys: *VCS datasheet.* Synopsys Website. http://www.synopsys.com/Tools/Verification/FunctionalVerification/Documents/vcs-ds.pdf.

[TV10a] Deian Tabakov et Moshe Vardi: *Monitoring temporal SystemC properties.* Dans *MEMOCODE '10: Proceedings of the 8th IEEE/ACM International Conference on Formal Methods and Models for Co-Design*, pages 123–132, July 2010.

[TV10b] Deian Tabakov et Moshe Vardi: *Optimized Temporal Monitors for SystemC.* Dans *RV'10: Proceedings of the international conference on Runtime Verification*, tome 6418 de *Lecture Notes in Computer Science*, pages 436–451. Springer Berlin / Heidelberg, 2010.

[TVKS08] Deian Tabakov, Moshe Y. Vardi, Gila Kamhi et Eli Singerman: *A temporal language for SystemC.* Dans *FMCAD '08: Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*, pages 22:1–22:9. IEEE Press, 2008.

[UL07] Mark Utting et Bruno Legeard: *Practical Model-Based Testing: A Tools Approach.* Morgan-Kaufmann, Elsevier Science, 2007. ISBN 0-12-372501-1.

[Var07] Moshe Y. Vardi: *Formal techniques for SystemC verification.* Dans *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 188–192. ACM, 2007.

[VER01] *IEEE Std 1364-2001, IEEE Standard Verilog Hardware Description Language*, September 2001.

[VHD02] *IEEE Std 1076-2002, IEEE Standard VHDL Language Reference Manual*, 2002.

[VLE08] *IEEE Std 1647-2008, IEEE Standard for the Functional Verification Language e*, August 2008.

[VW86] Moshe Y. Vardi et Pierre Wolper: *An automata-theoretic approach to automatic program verification.* Dans *LICS'86: Proceedings of the Symposium on Logic in Computer Science*, pages 332–345. IEEE Computer Society, June 1986.

[Win93] Glynn Winskel: *The Formal Semantics of Programming Languages - An Introduction.* The MIT Press, 1993.

Résumé – Au-delà de la formidable évolution en termes de complexité du circuit électronique en soi, son adoption et sa diffusion ont connu, au fil des dernières années, une explosion dans un très grand nombre de domaines distincts. Un système sur puce peut incorporer une combinaison de composants aux fonctionnalités très différentes. S'assurer du bon fonctionnement de chaque composant, et du système complet, est une tâche primordiale et épineuse. Dans ce contexte, l'Assertion-Based Verification (ABV) a considérablement gagné en popularité ces dernières années : il s'agit d'une démarche de vérification où des propriétés logico-temporelles, exprimées dans des langages tels que PSL ou SVA, spécifient le comportement attendu du design. Alors que la plupart des solutions d'ABV existantes se limitent au niveau transfert de registres (RTL), la contribution décrite dans cette thèse s'efforce de résoudre un certain nombre de limitations et vise ainsi une solution mature pour le niveau transactionnel (TLM) de SystemC. Une technique efficace de construction de moniteurs de surveillance à partir de propriétés PSL est proposée : cette technique, inspirée d'une approche originale existante pour le niveau RTL, est ici adaptée à SystemC TLM. Une méthode spécifique de surveillance des actions de communication à haut niveau d'abstraction est également détaillée. Les possibilités offertes par la technique présentée sont significativement étendues en proposant, pour les propriétés écrites en langage PSL, à la fois un support formel et une mise en œuvre pratique pour des variables auxiliaires globales et locales, qui constituent un élément essentiel lors des spécifications à haut niveau d'abstraction. Tous ces concepts sont également implémentés dans un outil prototype. Afin d'illustrer l'intérêt de la solution proposée, diverses expérimentations sont effectuées avec des designs aux dimensions et complexités différentes. Les résultats obtenus permettent de souligner le fait que la méthode de vérification dynamique suggérée reste applicable pour des designs de taille réaliste.

Abstract – Over the last years, the growing of electronic circuit complexity has experienced a tremendous evolution. Moreover, electronic circuits have become widespread elements in many different areas. This development leads to Systems-on-Chip incorporating a combination of components with highly heterogeneous features. Ensuring the correct behavior of each component, as well as validating the behavior of the whole system, is both a compelling and painful task. In this context, Assertion-Based Verification (ABV) has widely gained acceptance over the recent years : following this approach, temporal properties expressed using languages such as PSL or SVA specify the expected behavior of the design. While most existing ABV solutions are restricted to the register transfer level (RTL), the work of this thesis attempts to overcome some limitations by developing an actual ABV solution for the transaction level modeling (TLM) in SystemC. An effective technique for the construction of checker modules from PSL properties is proposed : this technique for SystemC TLM is inspired from a pioneering approach for RTL. A specific method for monitoring communication activities at a high level of abstraction is also described. The scope of the proposed technique is significantly improved by adding to PSL both a formal and a practical support for auxiliary global and local variables, which are compelling in higher level specifications. All these concepts are implemented in a prototype tool. In order to present the applicability of the proposed solution, we performed various experiments using designs of different sizes and complexities. The experimental results show that this dynamic verification methodology is also suitable for real-world designs.