RÉPUBLIQUE ALGÉRIENNE DÉMOKRATIQUE ET POPULAIRE
MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR ET DE LA RECHERCHE SCIENTIFIQUE
UNIVERSITE HADJ LAKHDAR - BATNA -

T H E S E

pour obtenir le diplôme de

DOCTORAT EN SCIENCES

Spécialité : Informatique

présentée et soutenue publiquement

par

Riadh HOCINE

le :14/02/2013

Titre :

Une méthodologie de conception automatique de modèles formels pour des descriptions SystemC

Directeur de thèse : Hamoudi KALLA

JURY

M. Noureddine Bouguechal	Professeur	U. Hadj Lakhdar, Batna	Président
M. Ammar Lahlouhi	MCA	U. Hadj Lakhdar, Batna	Examinateur
M. Abdelkrim Amirat	MCA	U. Mohamed-Cherif Messaadia, Souk-Ahras	Examinateur
M. Kamal Eddine Melkemi	MCA	U. Mohamed Khidhar, Biskra	Examinateur
M. Mohamed Chaouki Babahnini	MCA	U. Mohamed Khidhar, Biskra	Examinateur
M. Hamoudi Kalla	MCA	U . Hadj Lakhdar, Batna	Rapporteur

Table des matières

Table des figures **6**

1 Introduction **9**

I Conception et vérification des systèmes embarqués **15**

2 Flot de conception des systèmes embarqués **17**
2.1 Introduction . . . 17
2.2 Systèmes embarqués . . . 18
2.2.1 Caractéristiques d'un système embarqué . . . 19
2.2.2 Contraintes temps réel . . . 20
2.2.3 Architecture des systèmes embarqués . . . 20
2.3 Flot de conception d'un système embarqué . . . 21
2.3.1 Flot de conception de base . . . 21
2.3.2 Flot de conception modulaire . . . 22
2.3.3 Flot de conception à base d'IPs . . . 23
2.4 Flot de conception des systèmes sur puces . . . 25
2.4.1 Flot de conception logiciel pour les systèmes sur puces . . . 25

3 SystemC, un langage de conception au niveau système **27**
3.1 Introduction . . . 27
3.2 Présentation du langage SystemC . . . 28
3.2.1 Historique . . . 28
3.2.2 Objectifs . . . 29
3.3 Architecture de SystemC . . . 30
3.4 Les particularités de SystemC . . . 30
3.5 Principaux composants SystemC . . . 31
3.5.1 Les modules . . . 31
3.5.2 Les processus . . . 32
3.5.3 Les Ports . . . 34
3.5.4 Les Interfaces . . . 34
3.5.5 Les Canaux . . . 35

3.6 Simulation par SYSTEMC . . . 37
3.6.1 Le modèle du temps . . . 37
3.6.2 Le Noyau SYSTEMC . . . 38
3.7 Exemple de description en SYSTEMC . . . 38
3.8 conclusion . . . 41

4 Vérification des systèmes embarqués 43
4.1 Introduction . . . 43
4.2 Vérification par simulation . . . 44
4.2.1 Étapes d'exécution d'un simulateur . . . 45
4.2.2 Le compilateur . . . 46
4.2.3 Génération du programme simulable . . . 46
4.2.4 Exécution de la simulation . . . 47
4.3 Vérification formelle . . . 48
4.3.1 Approches de vérification formelle . . . 49
4.3.2 Apport des méthodes formelles à la conception conjointe . . . 52
4.4 Vérification des systèmes décrits en SYSTEMC . . . 55
4.4.1 Techniques de validation de modèles SYSTEMC . . . 55
4.4.2 Intégration des méthodes formelles dans les flots de conception . . . 57
4.5 Conclusion . . . 59

II Transformation de descriptions SYSTEMC en modèles formels 61

5 Etat de l'art sur les langages de spécifications formelles 63
5.1 Introduction . . . 63
5.2 Les trois familles des langages de spécifications formelles . . . 64
5.2.1 Langages de spécifications exécutables mais pas prouvables . . . 64
5.2.2 Langages de spécifications prouvables mais pas exécutables . . . 65
5.2.3 Langages de spécifications exécutables et prouvables . . . 65
5.3 Outils pour la vérification formelle . . . 68
5.3.1 Coq . . . 68
5.3.2 SIGALI . . . 70
5.4 Conclusion . . . 71

6 SIGNAL, un langage formel synchrone 73
6.1 Introduction . . . 73
6.2 Conception synchrone des systèmes réactifs . . . 74
6.2.1 Les systèmes réactifs . . . 74
6.2.2 Caractéristiques des systèmes réactifs . . . 75
6.2.3 Modèles synchrones pour systèmes réactifs . . . 76
6.3 Le formalisme synchrone . . . 77
6.4 Hypothèse synchrone . . . 78

6.4.1 Concepts de base . . . 79
6.5 Polychrony, un environnement synchrone pour la conception des systèmes . . . 80
6.6 Le paradigme synchrone . . . 81
6.6.1 Le style impératif : ESTEREL et SYNCCHARTS . . . 82
6.7 Langage SIGNAL . . . 83
6.7.1 Exemple de modélisation en SIGNAL . . . 85
6.7.2 Les éléments de base du langage SIGNAL : les signaux . . . 85
6.7.3 Les opérateurs de signaux . . . 88
6.7.4 Compilation de SIGNAL . . . 90
6.8 Conclusion . . . 90

7 Transformation de descriptions SYSTEMC en modèles formels de SIGNAL 93
7.1 Introduction . . . 93
7.1.1 Présentation générale de l'approche proposée . . . 94
7.1.2 Restriction syntaxique et hypothèses . . . 95
7.2 Préliminaires . . . 96
7.2.1 Graphe de flot de données . . . 96
7.2.2 Graphe de flot de contrôle . . . 97
7.2.3 Le formalisme Static Single Assignment . . . 98
7.3 Phase d'extraction structurelle . . . 99
7.4 Phase d'extraction comportementale . . . 101
7.4.1 Transformation des types de données . . . 101
7.4.2 Transformation des opérateurs et des fonctions standards . . . 101
7.4.3 Transformation des blocs SSA . . . 102
7.4.4 Transformation de la fonction PHI . . . 102
7.4.5 Transformation des expressions conditionnelles . . . 103
7.4.6 Transformation des expressions d'affectation . . . 103
7.4.7 Transformation des boucles . . . 104
7.4.8 Transformation des modules . . . 105
7.5 Codage des pointeurs SYSTEMC en SIGNAL . . . 106
7.5.1 Présentation du problème . . . 106
7.5.2 Lecture des pointeurs (load) . . . 106
7.5.3 Écriture des pointeurs (store) . . . 110
7.6 Conclusion . . . 110

8 Implémentation 111
8.1 introduction . . . 111
8.2 Format intermédiaire SSA . . . 111
8.3 Le compilateur GCC-4.1.0 . . . 112
8.4 Adaptation du Compilateur GCC-4.1.0 . . . 113
8.5 L'outil de transformation SC2SIG . . . 114
8.6 Exemples de traduction . . . 115

8.6.1 Exemple 1 . . . 115
8.6.2 Exemple 2 : Finite Impulse Response . . . 116
8.7 Compilation des modèles SIGNAL . . . 121
8.7.1 Compilation en code C exécutable . . . 121
8.7.2 Compilation en format Z3Z . . . 121
8.8 Conclusion . . . 121

9 Conclusion et perspectives **123**

Bibliographie **125**

A Publications **131**

Table des figures

1.1 Méthodologie proposée 11

2.1 Structure typique d'un système embarqué 19
2.2 Flot général de conception conjointe 21
2.3 Flot conception modulaire 22
2.4 Bibliothèques et niveaux d'abstraction 23
2.5 Technique de réutlisabilté 24

3.1 Organisation de SYSTEMC [**?**] 30
3.2 Organisation structurelle d'un système en SYSTEMC 32
3.3 Diagramme d'état d'un processus SYSTEMC 34
3.4 Diagramme UML des cannaux SYSTEMC 35
3.5 Diagramme des échelles de temps (simulation SYSTEMC) 37
3.6 Système mémoire - processeur 39

4.1 Environnement de simulation 45

5.1 Représentation d'un arbre de preuve en KIV 67
5.2 l'environnement Pcoq 69
5.3 Vérification formelle par SIGALI 71

6.1 Modèle d'un système réactif 74
6.2 Exemple d'un système réactif 75
6.3 Architecture de l'environnement Polychrony [**?**] 81

7.1 Transformation automatique en deux phases 94
7.2 Graphe de flot de données 96
7.3 Graphe de flot de contrôle 97
7.4 Static Single Assignment 99
7.5 Phase d'extraction structurelle 100
7.6 Règles de transformation des types standards 101
7.7 Encodage des blocs de base SSA en SIGNAL 102
7.8 Encodage de la fonction ϕ en SIGNAL 103
7.9 Encodage d'expressions conditionnelles en SIGNAL 103

7.10 Encodage d'expressions d'affectation en SIGNAL 104
7.11 Exemple de boucle *for* 104
7.12 Encodage des boucles en SIGNAL 104
7.13 Exemple d'un module SYSTEMC 105

8.1 Architecture du compilateur GCC-4.1.0 112
8.2 GCC-4.1.0 avant (a) et après (b) l'adaptation 113
8.3 Architecture de SC2SIG (GCC adapté) 114
8.4 IPs de FIR 116
8.5 Compilation de modèle SIGNAL en code C 121
8.6 Compilation de modèles SIGNAL en code Z3Z 121

Remerciements

Je tien à exprimer ma reconnaissance à mon directeur de thèse et mon ami Monsieur Hamoudi Kalla, maître de conférence au département d'informatique de l'université de Batna et chef de l'équipe REDS, pour m'avoir proposer ce sujet intéréssant et m'avoir donné l'occasion de poursuivre ses travaux de recherche post-doctoraux, pour tous les précieux conseils qu'il m'a donnés et pour le temps qu'il m'a consacré lors la rédaction de cette thèse.

Je remercie sincèrement Monsieur Noureddine Bouguechal professeur au département d'électronique de l'université de Batna, d'avoir accepté de présider mon jury. Je le remercier encore d'avoir accepté un jour d'être mon encadreur dans les moments les plus difficiles de mon parcours scientifique.

Je tien à exprimer ma profonde gratitude à Monsieur Ammar Lahlouhi maître de conférence au département d'informatique de l'université de Batna, d'avoir accepté d'être membre de mon jury, et pour m'avoir orienté et soutenir durant ma dernière année de thèse.

Je remercie Monsieur Abdelkrim Amirat maître de conférence au département d'informatique de l'université de Souk-Ahras, qui m'a particulièrement honoré par sa présence en qualité de membre de jury.

Je remercie également Monsieur Kamal Eddine Melkemi maître de conférence au département d'informatique de l'université de Biskra, d'avoir accepté d'être membre de mon jury, pour m'avoir fait part de ses précieux conseilles et critiques pertinentes, et surtout pour sa gentillesse.

J'associe à ces remerciements Monsieurs Mohamed Chaouki Babahnini maître de conférence au département d'informatique de l'université de Biskra, d'avoir accepté sans hésitation de faire partie de mon jury.

Je n'oublie pas l'ensemble des membres de notre équipe REDS du laboratoire LaSTIC, et particulièrement Chafik Arar, Salim Kalla, Mejda Abdessemed, Sonia Sabrina Bendib, Noura Alloui et Hassina Fedala, pour leurs encouragements.

Mes derniers remerciements s'adressent aux membres de ma famille : mes parents, ma femme, mes soeurs et mes frères, qui m'ont encouragé au cours de ces longues années de préparation de cette thèse.

Chapitre 1

Introduction

Contexte

Les systèmes embarqués sont de plus en plus utilisés dans plusieurs domaines applicatifs importants tels que le transport (automobiles, robots, avions, ...) et les systèmes de contrôle de processus industriels. Aujourd'hui, ils sont omniprésents dans notre entourage. On en trouve dans les téléphones mobiles, les appareils électroménagers et aussi dans les consoles de jeux vidéo. Certains de ces systèmes sont souvent critiques, comme les avions ou les automobiles, d'autres moins, comme les montres ou les machines à laver. Ces systèmes réalisent des tâches complexes critiques et sont soumis à des fortes contraintes en termes de temps, de distribution, de consommation d'énergie et de sûreté. En effet, au vu des conséquences catastrophiques (perte d'argent, de temps, ou pire de vies humaines) que pourrait entraîner une panne, ces systèmes doivent être extrêmement fiables. Par ailleurs, les progrès accomplis en électronique et en informatique ont apporté beaucoup à l'augmentation de la puissance de calcul des machines et à l'amélioration des performances des systèmes embarqués. Vu l'accroissement de la complexité des systèmes embarqués, la problématique du développement de nouvelles méthodes, outils de conception et langages de description de matériel/logiciels pour ces systèmes fait actuellement l'objet de nombreuses recherches.

Ces dernières années ont vu une importante utilisation dans l'industrie des langages de programmation pour la modélisation des systèmes embarqués, tels que C/C++ et Java. Il existe donc aujourd'hui des évolutions importantes dans le domaine des langages de description de matériel (Hardware Description Languages - HDL) : souplesse de spécification, rapidité de simulation, va-

lidation formelle, etc. SYSTEMC fait partie de ces évolutions, il est comme Verilog et VHDL, un langage de description de matériel, permettant de représenter ou modéliser des systèmes sous forme comportementale ou structurelle, dérivé du C++ mais incluant certains concepts matériels (les modules, les ports, les FIFO, les signaux,...), idéal pour décrire les systèmes matériels et logiciels.

Les principaux avantages de SYSTEMC est qu'il permet de modéliser les systèmes à haut niveau et qu'ils offrent de bonnes performances en vitesse de simulation. Il offre aussi plusieurs niveaux d'abstraction pour spécifier un système matériel/logiciel. En plus, afin de répondre aux fortes contraintes de performance, de coût de production et de temps de mise sur le marché, les industriels d'aujourd'hui s'intéressent à la réutilisation des composants existants, et donc aux composants IP (Intellectual Property). Ces composants permettent de maîtriser la complexité croissante des systèmes. Etant donné que ces composants sont hétérogènes et peuvent être de plusieurs sources, leur conception et leur intégration doivent être vérifiées. Le passage par un processus de vérification est indispensable afin d'éviter les anomalies de fonctionnement causées par des erreurs de spécification.

Motivations et contributions

La vérification basée sur la simulation est le seul moyen qui permet de vérifier une telle modélisation en langages de haut niveau. Cependant, la simulation ne permet généralement pas d'obtenir une couverture exhaustive des erreurs. Il est donc important de faire recours à d'autres techniques de vérification complémentaires, telles que les techniques de la vérification formelle. Vue que SYSTEMC est une bibliothèque C++ et ne présente pas de propriétés formelles, il est donc impossible d'utiliser les techniques de la vérification formelle pour vérifier formellement des descriptions SYSTEMC. Le seul moyen possible est de décrire les descriptions formelles à la main, ce qui implique un surcoût en temps, en argent et en moyens humains plus importants.

L'objectif de ce travail de recherche est donc d'étudier la possibilité de traduction automatique des descriptions SYSTEMC vers des descriptions formelles. Dans ce but, nous avons choisi comme cible le langage formel SIGNAL, destiné aux systèmes embarqués. SIGNAL, est un langage synchrone flot de données de type équationnel à sémantique formelle orientée preuve. Ce choix de SIGNAL est motivé par le fait qu'aujourd'hui ce langage est parmi les langages de haut niveau les plus populaires pour la modélisation et la simulation des systèmes embarqués, et aujourd'hui SIGNAL est connecté à plusieurs outils de vérification formelle. De plus, plusieurs descriptions de SIGNAL sont similaires à ceux de SYSTEMC.

La traduction de modèles SYSTEMC en modèles formels pose plusieurs problèmes aux chercheurs, tels que la représentation formelle des pointeurs. Pour résoudre ces problèmes, plusieurs techniques ont été proposées dans la littérature. Nous proposons, dans cette thèse, une méthodologie de traduction automatique à plusieurs étapes comme décrit dans la Figure **??**. La vérification des modèles SYSTEMC peut être réalisée sur les modèles formels SIGNAL générés en utilisant les mécanismes de compilation et de vérification de cohérence du compilateur SIGNAL, et aussi

des outils de vérification formelle, tels que le model-checking SIGALI [**?**] et l'assistant de preuve Coq [**?**].

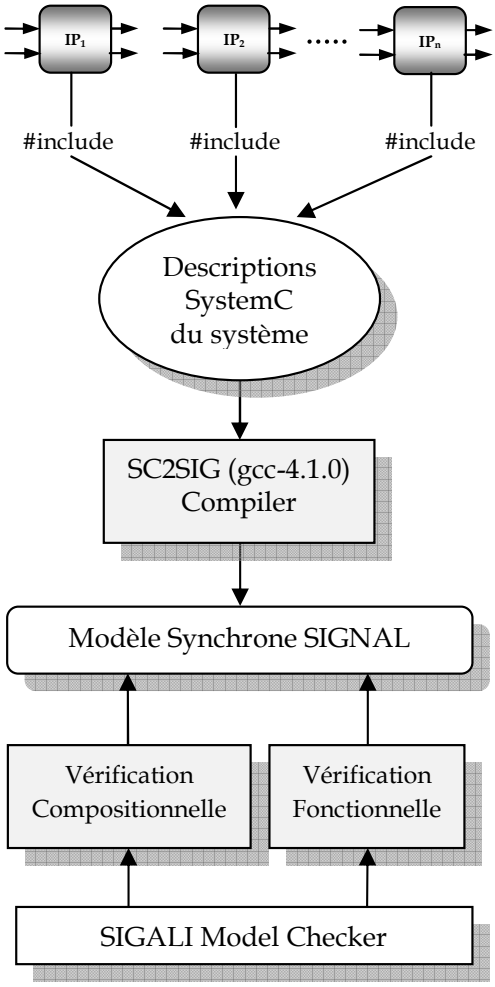

FIGURE 1.1 – Méthodologie proposée

Notre contribution s'est située en différents points :

- La contribution principale de cette thèse est, contrairement aux solutions existantes aujourd'hui, la proposition et la mise en œuvre d'un générateur original pour traduire à la fois la structure et le comportement de modules SYSTEMC en SIGNAL. Certaines solutions s'intéressent uniquement à l'extraction comportementale comme les solutions proposées par Kalla et Talpin dans [**?**], Grobe et Drechsler dans [**?**], Karlsson et Eles dans [**?**] et Habibi et Tahar dans [**?**]. De plus la solution proposée dans [**?**] concerne uniquement les modèles C/C++. D'autres solutions s'intéressent uniquement à l'extraction structurelle comme les solutions proposées par Berner et Hiren dans [**?**] et Moy et Maraninchi dans [**?**].
- La conception par composition : afin d'intégrer de tels composants dans la conception d'un

système, il est impératif de pouvoir s'assurer que ces composants sont conformes à leur spécification d'usage. Un tel assemblage de composants peut générer plusieurs erreurs, telles que les erreurs d'interopérabilité et d'interaction entre composants. Pour détecter facilement les sources de ces erreurs d'assemblage et de les corriger rapidement, les informations structurelles des modèles SYSTEMC doivent êtres préservés en SIGNAL. Donc, le premier travail réalisé est la génération de la structure (squelette) décrivant le modèle d'assemblage d'un tel système, ce qui implique la génération d'une interface formelle pour chaque composant SYSTEMC. L'interface d'un composant SYSTEMC ne contient traditionnellement pas beaucoup d'informations. Souvent, seuls les signaux d'entrée et de sortie sont spécifiés ainsi que leurs types respectifs. Après avoir généré ces interfaces formelles, le compilateur de SIGNAL permet de vérifier les propriétés d'interaction entre composants, par exemple vérifier si les types de données de deux signaux connectés sont compatibles.

- L'extraction du comportement : après l'extraction des interfaces des composants, nous avons travaillé sur l'extraction du comportement de chaque composant en comportement formelle en SIGNAL. Pour cela, nous avons utilisé le compilateur GCC de SYSTEMC et son format intermédiaire SSA (Static Single Assignement). L'idée est de transformer les constructions SYSTEMC en code SSA, et ensuite de transformer ce code SSA en code SIGNAL. Le choix est motivé par le fait que la structure de contrôle d'un code SSA est beaucoup plus simple que celle d'un code SYSTEMC.
- la synchronisation des horloges : la plupart des travaux existants, comme [**?**], sur la traduction de constructions C/C++ en SIGNAL s'appuient sur une hypothèse de synchronisation globale des horloges. Cette hypothèse impose comme conséquence une exécution séquentielle des programme SIGNAL. Afin de résoudre ce problème, nous avons proposé une technique basée sur les processus et les modules SYSTEMC pour le calcul d'horloges. Elle permet d'identifier les relations de synchronisation entre les évènements des modules SYSTEMC et donc de synchroniser uniquement les évènements dépendants de chaque module. Elle détermine un ordre partiel entre les Blocks SSA d'un module sous la forme d'un arbre.
- Extension du langage SIGNAL : afin de traduire toutes les constructions SYSTEMC en SIGNAL, nous avons proposé une extension du langage SIGNAL.

Donc, la contribution de cette thèse consiste d'une part à étendre la solution proposée dans [**?**] aux descriptions SYSTEMC afin de prendre en considération les constructions matérielles et les primitives caractérisant le langage, en formalisme synchrone du langage SIGNAL, et d'autre part de combiner l'extraction comportementale et structurelle pour générer des modèles synchrones formellement prouvables avec prise en considération de la multitude d'horloge de signaux.

Plan de la thèse

Cette thèse est structurée en deux grandes parties en plus de la présente introduction et une conclusion générale.

La première partie regroupe les chapitres deux, trois et quatre. Elle présente d'une manière générale la conception et la vérification des systèmes embarqués ainsi que les approches suivies pour la mise en place de ces systèmes.

Dans le deuxième chapitre, nous mettrons tout d'abord l'accent sur les systèmes embarqués et les terminologies liées à ce concept. Ensuite, nous évoquerons l'ensemble des trois approches adoptées dans la littérature pour concevoir des systèmes embarqués, à savoir l'approche classique, l'approche modulaire et l'approche à base de composants virtuels. Cette dernière représente une nouvelle méthodologie de conception des systèmes embarqués complexes en se basant sur le principe de réutilisation des composants d'une bibliothèque, et ceci afin de réduire le temps de conception ainsi que l'effort et le coût de développement de nouveaux composants. Nous aborderons à la fin de ce chapitre les systèmes embraqués sur puces (System On Chip) et les méthodologies de conception associées à cette catégorie de systèmes.

Dans le troisième chapitre, nous présenterons SYSTEMC qui est un langage de description de matériel. Nous décriverons en détail le langage SYSTEMC sur les plans : architectural, syntaxique, sémantique ainsi que son noyau de simulation. Ce langage constitue le cadre de nos travaux.

Le quatrième chapitre expose deux méthodes de vérification de spécifications des systèmes embarqués : la simulation avec ses inconvénients et la vérification formelle avec ses complexités et difficultés d'application. Ce chapitre présente ensuite un état de l'art sur les approches utilisées dans la littérature pour la vérification formelle des systèmes embarqués, et particulièrement ceux décrits en SYSTEMC, qui fait l'objet de ce travail.

La deuxième partie de cette thèse, constituée des chapitres cinq, six, sept, huit et neuf, est consacrée à la description de notre méthodologie pour intégrer la vérification formelle dans les flots de conception des systèmes embarqués à base d'IPs décrits en SYSTEMC.

Dans le cinquième chapitre, nous donnerons un état de l'art sur les langages de spécification formelle. Nous présenterons aussi deux outils (SIGALI et Coq) de vérification formelle qui nous intéresse dans ce travail pour vérifier les spécifications SIGNAL générées.

Dans le sixième chapitre, nous présenterons en détail le langage synchrone SIGNAL.

Dans le septième chapitre et en se basant sur les concepts décrits dans le chapitre précédent, nous présenterons notre propre vision pour permettre l'intégration de la preuve formelle dans un flot de conception à base de SYSTEMC. Nous montrerons la possibilité d'utiliser le formalisme synchrone par une passe de génération automatique du code SIGNAL à partir de descriptions SYSTEMC.

Dans le huitième chapitre et dans l'optique du chapitre précédent, nous proposerons un ensemble de règles sémantiques nécessaires à la traduction des constructions syntaxiques principales de SYSTEMC (sous-ensemble) en langage SIGNAL, pour permettre une génération de modèles formels en formalisme synchrone du langage SIGNAL à partir de composants non-formels de SYSTEMC.

Le neuvième chapitre présente le développement logiciel effectué pendant cette thèse. Il s'agit d'une implémentation de notre méthodologie qui consiste à adapter la version open source du

compilateur GCC-4.1.0 de GNU pour générer automatiquement des modèles synchrones SIGNAL à partir d'IPs SYSTEMC. Cette adaptation du compilateur consiste à détourner sa phase de génération de code objet pour la génération du code SIGNAL.

Enfin, nous terminerons par une conclusion générale qui résume nos contributions et dresse le bilan des travaux effectués dans le cadre de cette thèse ainsi qu'un certain nombre de perspectives.

Première partie

Conception et vérification des systèmes embarqués

Chapitre 2

Flot de conception des systèmes embarqués

2.1 Introduction

Beaucoup d'applications reposent aujourd'hui sur l'utilisation de circuits intégrés complexes et particulièrement les microprocesseurs, et plus largement, sur l'utilisation de systèmes embarqués incluant des parties matérielles et logicielles (système d'exploitation et/ou logiciel d'application) intégrés dans un même circuit. La confiance pouvant être accordée à ces systèmes complexes dépend beaucoup de leur capacité à réagir de façon sûre face aux différents événements générés par l'environnement extérieur pendant l'exécution.

Les ordinateurs et, plus généralement, les systèmes électroniques et informatiques sont devenus omniprésents dans notre vie quotidienne. Téléphones portables, baladeurs MP3, PDAs, cartes de paiement. Ceci est sans citer d'autres applications moins répandues comme : la conduite assistée pour voitures, monitorage du trafic, contrôles d'accès, systèmes de sécurité, etc. Pour atteindre un tel résultat, l'électronique a dû progressivement s'adapter pour devenir portable, puis ultra-miniaturisée : c'est la naissance des systèmes embarqués [**?**].

Au début, les applications des systèmes embarqués étaient surtout dans le domaine militaire à cause d'un certain nombre de facteurs, notamment leur prix et l'avantage stratégique d'une telle technologie. Cependant, leur grande potentialité a rapidement ouvert les portes vers des applications civiles (pilotes automatiques pour avions de ligne, etc). En plus, le progrès technologique a considérablement baissé leur prix et par conséquent augmenté leurs possibilités d'utilisation.

Actuellement, ces systèmes trouvent des applications partout dont la taille, le poid et même la consommation sont devenus des facteurs clés. Un exemple est l'évolution des téléphones portables, depuis les valises des années 80 jusqu'aux minuscules modèles d'aujourd'hui.

Une des caractéristiques importantes de ces systèmes est d'être réactif. Un système réactif est un système qui réagit continûment avec son environnement à un rythme imposé par cet environnement. Il reçoit, par l'intermédiaire de capteurs, des entrées provenant de l'environnement, appelés stimuli, réagit à tous ces stimuli en effectuant un certain nombre d'opérations et produits, grâce à des actionneurs, des sorties utilisables par l'environnement, appelés réactions.

2.2 Systèmes embarqués

Qu'est-ce un système informatique embarqué ? La définition qui semble la plus communément admise stipule que ce sont des systèmes informatiques dont les ressources sont limitées. Par ressource, on entend ressources de taille, de poids, de consommation électrique et de puissance de calcul. Certes, tout ordinateur de bureau a une consommation électrique limitée par la puissance de l'installation électrique de la prise sur laquelle il est branché. Il en va de même pour toutes les limites mentionnées ci-dessus puisque le monde est fini. Mais, dans le cas des ordinateurs de bureau, cette limite n'est en pratique pas prise en compte par les fabricants. En revanche, pour un PDA, un téléphone portable ou un satellite, elle est critique.

Ces systèmes réalisent des tâches complexes critiques et sont soumis à de fortes contraintes en termes de temps, de distribution, de consommation d'énergie et de sûreté. En effet, au vu des conséquences catastrophiques (perte d'argent, de temps, ou pire de vies humaines) que pourrait entraîner une panne, ces systèmes doivent être extrêmement fiables. Par ailleurs, les progrès accomplis en électronique et en informatique ont apporté beaucoup à l'augmentation de la puissance de calcul des machines et à l'amélioration des performances des systèmes embarqués.

Définition 1 (Système embarqué) *Un système embarqué est un système électronique et informatique autonome qui ne possède pas d'entrées/sorties standards.*

Définition 2 (Système embarqué) *Un système embarqué peut être défini comme un système électronique et informatique autonome, qui est dédié à une tâche bien précise. Ses ressources disponibles sont généralement limitées. Cette limitation est généralement d'ordre spatial (taille limitée) et énergétique (consommation restreinte) [Wikipédia].*

Définition 3 (Système embarqué) *Un système embarqué est un système commandé et contrôlé par un calculateur combinant diverses technologies (mécanique, hydraulique ou électrique). Il doit répondre à des impératifs de criticité, de réactivité, d'autonomie, de robustesse et de fiabilité. L'interaction se fait par l'intermédiaire de capteurs qui fournissent des informations sur l'état du système. Ces informations sont utilisées par le calculateur pour générer la commande à appliquer au système au moyen d'actionneurs. [**?**]*

Un système embarqué est généralement composé de matériel exécutant du logiciel (Figure **??**), ce qui implique un ou plusieurs processeurs, des mémoires et éventuellement des circuits dédiés à des applications spécifiques de type ASIC (Application Specific Integrated Circuits). Ces dernières années, la capacité des circuits intégrés a augmenté considérablement : les systèmes qui autrefois étaient composés de plusieurs puces reliées entre elles sur un circuit imprimé peuvent maintenant être directement implantés dans une seule puce.

Remarque 1 *Les systèmes embarqués font très souvent appel à l'informatique, et notamment aux systèmes temps réel.*

Remarque 2 *Le terme de système embarqué désigne aussi bien le matériel que le logiciel utilisé.*

Les systèmes embarqués deviennent donc souvent des systèmes sur puce (SoC : System on Chip). Tous les avantages des systèmes embarqués (vitesse, taille, consommation, ...) sont augmentés, mais aussi la complexité du développement. En pratique, même si un certain nombre de leurs caractéristiques peuvent être différentes, surtout du point de vue technologique, les systèmes intégrés et les systèmes embarqués tendent souvent vers des architectures semblables basées sur les mêmes éléments (processeurs, mémoires, logique dédiée, ...).

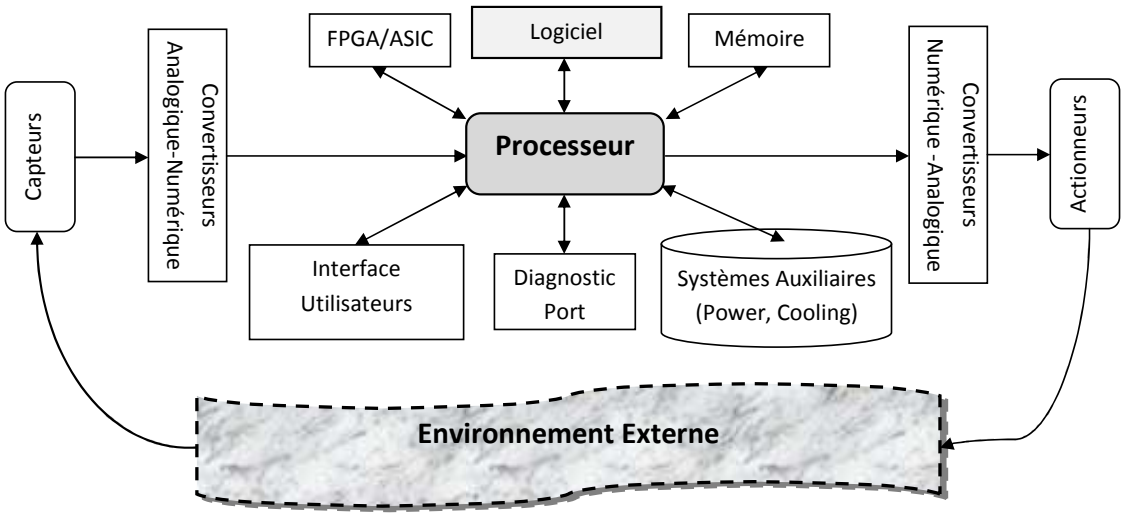

FIGURE 2.1 – Structure typique d'un système embarqué

2.2.1 Caractéristiques d'un système embarqué

Les principales caractéristiques d'un système embarqué sont les suivantes :

- C'est un système principalement numérique.
- Il met en oeuvre généralement un processeur.

- Il exécute une application logicielle dédiée pour réaliser une fonctionnalité précise et n'exécute donc pas une application scientifique ou grand public traditionnelle.
- Il n'a pas réellement de clavier standard (Bouton Poussoir, clavier matriciel...). L'affichage est limité (écran LCD...) ou n'existe pas du tout.
- Ce n'est pas un ordinateur en général mais des architectures similaires basse consommation sont de plus en plus utilisées pour certaines applications embarquées.

De point de vue architecture, on peut dire que :

- Un ordinateur standard (PC par exemple) peut exécuter tout type d'applications car il est généraliste alors qu'un système embarqué n'exécute qu'une seule application dédiée.
- Une interface IHM d'un système embarqué, peut être aussi simple qu'une diode led qui clignote ou aussi complexe qu'un tableau de bord d'avion de ligne.
- L'ensemble des circuits numériques ou des circuits analogiques sont ajoutés à l'architecture de base des systèmes embarqués pour augmenter les performances et la fiabilité

2.2.2 Contraintes temps réel

Certains systèmes embarqués sont très liés au temps, par exemple : un distributeur de billets, un radar ou un système de freinage ABS. Les systèmes embarqués doivent donc généralement respecter des contraintes temporelles strictes/souples (Soft/Hard Real Time).

Définition 4 (Système temps réel) *Un système est qualifié temps réel lorsque l'information après acquisition et traitement reste encore pertinente.*

Suivant les exigences temporelles d'un système embarqué, on peut distinguer deux classes de ces systèmes :

- **Système embarqués à contraintes temps réel strictes** : Ces systèmes doivent impérativement respecter les contraintes de temps imposées par l'environnement, et le non respect d'une contrainte temporelle peut avoir des conséquences catastrophiques. Le système de contrôle de trafic aérien et de conduite de missile sont deux exemples de ces systèmes.
- **Systèmes embarqués à contraintes temps réel souples** : Ces systèmes se caractérisent par leurs souplesses envers les contraintes temps réel imposées par l'environnement, il s'agit d'exécuter dans les meilleurs délais les fonctions. Le non respect d'une contrainte temporelle peut être acceptable par le système

2.2.3 Architecture des systèmes embarqués

Les systèmes embarqués utilisent généralement des microprocesseurs à basse consommation d'énergie ou des micro-contrôleurs, dont la partie logicielle est en partie ou entièrement programmée dans le matériel, généralement en mémoire, dans une mémoire morte (ROM), EPROM, EEPROM, FLASH, ... (Donc firmware).

2.3 Flot de conception d'un système embarqué

2.3.1 Flot de conception de base

Généralement, la conception d'un système embarqué se fait grâce au co-développement ou conception conjointe. La conception conjointe permet de développer conjointement les diverses parties d'un système hétérogène (logiciel, matériel, . . .). Il n'existe pas un unique flot de conception de systèmes embarqués mais plusieurs, basés sur des outils différents avec leurs avantages et leurs inconvénients, qui peuvent être la spécification de départ et la manière dont les raffinements sont effectués. Mais on peut extraire un flot de conception théorique qui est commun aux différentes méthodes, en faisant abstraction de tout flot spécifique (Figure **??**). Tout d'abord, on part d'une spécification fonctionnelle du système que l'on veut obtenir. C'est le cahier des charges du système.

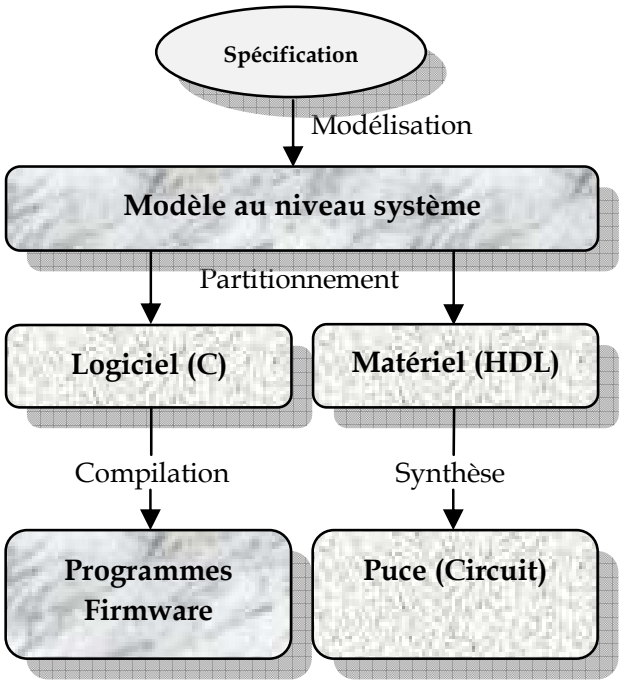

FIGURE 2.2 – Flot général de conception conjointe

Ensuite, il faut partitionner les tâches entre tâches matérielles et tâches logicielles. Il faut aussi choisir le matériel que l'on va utiliser : choisir le(s) processeur(s), le type de mémoire, la taille de la mémoire, . . . Puis, on produit un modèle de l'architecture ainsi que l'application (logiciel) que l'on va raffiner et valider par différentes simulations jusqu'à obtenir ce que l'on veut, en terme de temps d'exécution du système, de surface de la puce et de consommation d'énergie. C'est l'exploration de l'espace de conception. Une fois le système est validé par la simulation, on produit un prototype pour vérifier qu'il n'y a pas d'erreurs, puis le système est fabriqué.

Les différents flots de conception existants sont basés sur des outils, mais on peut les évaluer selon plusieurs critères :

- temps de développement et difficulté,
- coût de production final,
- réutilisabilité,
- qualité du contrôle pendant la phase de conception,
- pas d'échec de réalisation finale.

2.3.2 Flot de conception modulaire

Vu la complexité de la plupart des projets, l'écriture du modèle au niveau système en un seul bloc est devenu une tâche difficile. La solution consiste à découper la spécification en morceaux (Figure **??**) qui forment des unités fonctionnelles et de définir des interfaces de communication entre ces entités pour pouvoir ensuite les traiter séparément. C'est de cette manière que plusieurs concepteurs peuvent travailler relativement indépendamment sur des modules distincts d'un même projet. En formant les modules, on obtient le système final, comme il est décrit dans la spécification initiale. Un modèle modulaire a aussi l'avantage de permettre la simulation et la vérification séparée des modules, donc un débogage plus localisé.

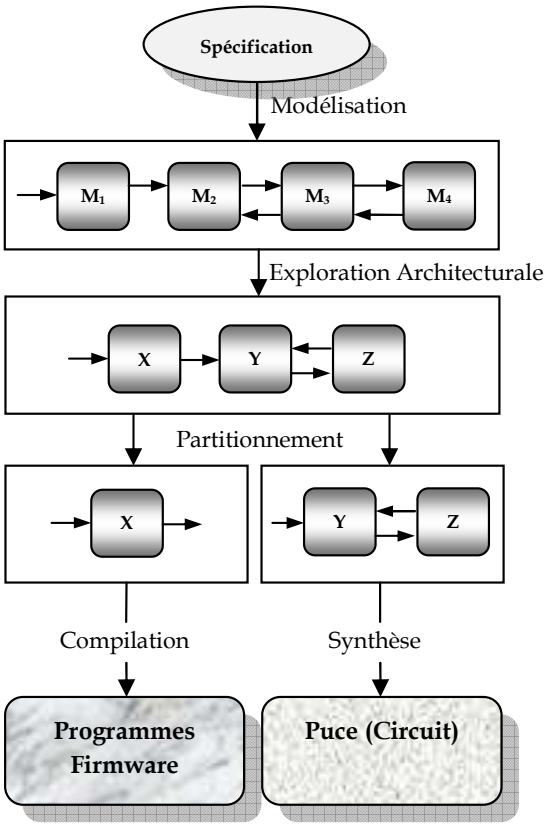

FIGURE 2.3 – Flot conception modulaire

2.3.3 Flot de conception à base d'IPs

2.3.3.1 La réutilisation

Les concepteurs de systèmes ont dû mal à adapter les méthodes de conception traditionnelles aux systèmes de grandes complexités. Ils sont obligés de gérer, non seulement un nombre énorme de transistors, mais aussi une grande complexité de conception. Les possibilités architecturales pour concevoir de tels systèmes sont extrêmement nombreuses [**?**]. Donc l'évaluation des différentes solutions architecturales devient un passage obligé de la conception moderne des systèmes.

A ces facteurs s'ajoute la forte demande de réduction du temps de mise sur le marché avec des performances de plus en plus exigeantes, font qu'il n'est plus possible de développer un système en partant de zéro. La solution à tous ces problèmes est la réutilisation de blocs déjà conçus et validés. La notion de composants génériques réutilisables s'impose comme une nouvelle alternative de conception.

2.3.3.2 Les composants réutilisables

Un composant (bloc) réutilisable est un élément d'une bibliothèque dont le concepteur dispose et qu'il peut directement l'utiliser ou l'instancier sans avoir à le concevoir [**?**]. L'un des problèmes majeurs concernant la réduction du temps de conception réside dans la bonne ou la mauvaise réutilisation (l'adéquation) des blocs existants. Les composants se répartissent entre les différents niveaux d'abstraction présentés dans la méthodologie de conception. Ainsi, tout au long du processus on utilise la bibliothèque qui correspond au niveau d'abstraction dans lequel on se trouve.

La Figure **??** présente les bibliothèques que nous pouvons constituer et utiliser aux différents niveaux d'abstraction.

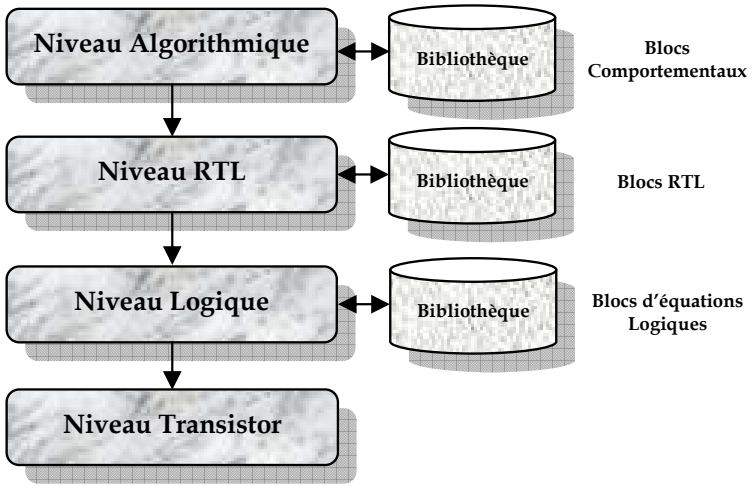

FIGURE 2.4 – Bibliothèques et niveaux d'abstraction

2.3.3.3 Méthodologie de réutilisabité

Le concepteur d'une nouvelle architecture doit, tout d'abord et durant le processus de conception, inspecter la bibliothèque de composants réutilisables pour déterminer avec précision l'ensemble des blocs qui peuvent être intégrés à son système (Figure **??**). Il doit être en mesure de comprendre les caractéristiques de chaque composant, de comparer ses performances avant de choisir les plus appropriés. Généralement, le composant choisi est sous une forme générique, ce qui nécessite une étape d'adaptation pour en faire un composant spécifique à l'application visée [**?**]. La dernière étape consiste à intégrer le composant dans l'architecture globale.

L'un des défis de l'intégration de composants d'une bibliothèque dans un système est le problème de la communication entre les deux ensembles. Pour vaincre ce problème, une description commune synthétisable peut permettre au concepteur d'instancier chaque composant en utilisant les outils de synthèse classique. Ensuite, pour intégrer le composant, le concepteur doit comprendre son interface de communication (architecture externe) avec l'extérieur, et l'insérer dans le système.

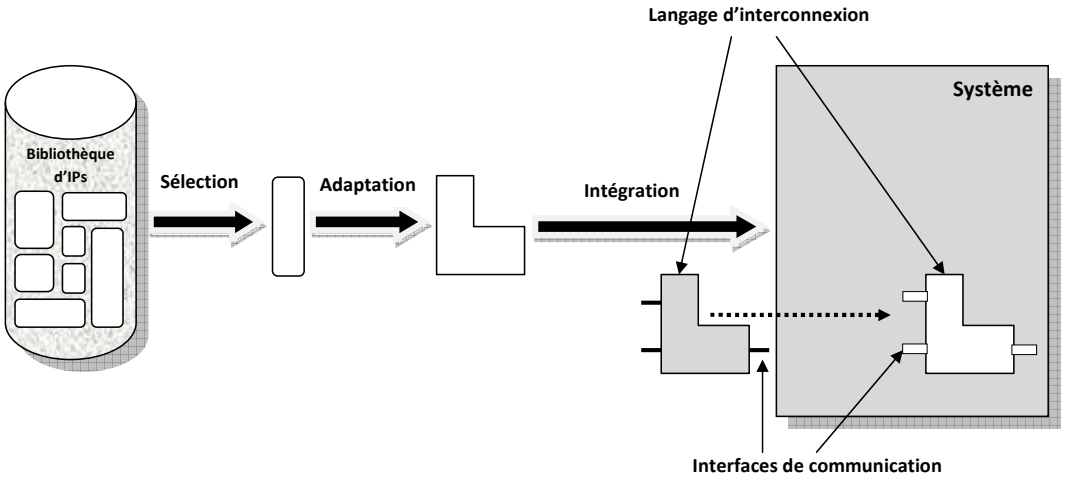

FIGURE 2.5 – Technique de réutlisabilté

2.3.3.4 Les composants virtuels

Un composant virtuel (IP : Intellectual Property) est la spécification d'un composant matériel ou logiciel réalisant une fonction bien déterminée pouvant être synthétisé, donc réutilisée, par un concepteur qui n'a pas participé à la spécification de ce composant. Cette méthodologie consiste à connaître uniquement la fonction réalisée par le composant sans savoir la manière de spécification. Cependant, l'absence de ces connaissances détaillées n'empêche pas le concepteur d'IPs à aboutir à des performances identiques à celles obtenues pas un processus de synthèse classique (flot de conception de base).

Types de composants virtuels En termes de type et selon [**?**], il existe trois catégories de composants virtuels en fonction du niveau d'abstraction de leur description :

1. *IP Hardware (durs)* : Ces composants sont décrits au niveau porte, optimisés pour une cible technologique donnée. Ils sont caractérisés par leurs performances mais aussi par l'absence de souplesse (modification). Cette classe permet d'optimiser la puissance, la taille ou les performances et le plan pour une technologie spécifique.
2. *IP Firmware (semi-durs)* : Un composant de cette classe, est une combinaison de Hardware et de Software. Ils sont décrits dans un langage de description de matériel (VHDL, Verilog ou SYSTEMC), conçus par une synthèse physique au niveau des plans de masse ; ils reposent sur l'emploi de technologies génériques et sont donc indépendants de la taille des transistors CMOS, chez un même fondeur.
3. *IP Software (souples)* : Ces composants sont décrits le plus souvent dans un langage de description de matériel au niveau RTL de la synthèse. La source de ces composants est disponible, ce qui permet ainsi de les modifier à volonté (mais avec prise en compte de la propriété intellectuelle). On devra optimiser le composant lors de son intégration dans le système. Ils sont intégrés par une étape de synthèse logique, ce qui assure à ces composants une indépendance totale vis à vis de la technologie (ASIC, FPGA).

2.4 Flot de conception des systèmes sur puces

L'évolution de la technologie d'intégration sur puce a permis d'augmenter considérablement le nombre de transistors qui peuvent être implantés sur une puce. Ce qui a permis à la naissance des System on Chip (SoC), qui consiste à intégrer plusieurs composants et unités (de traitement et stockage) sur la même puce. Cependant, les outils permettant la conception de ces systèmes ont dû mal à suivre le progré technologique rapide. Les technologies de mise en puce plus de transistors, sont plus avancées que les concepteurs sont capables de gérer dans un temps raisonnable, ce qui engendre des pertes de productivité. Afin de résoudre ce problème, il faut développer de nouveaux outils et méthodes de conception des systèmes embarqués. Cela est possible peut être grâce à des modifications dans le flot de conception des systèmes, et notamment sur le plan vérification et validation. C'est pourquoi une validation effectuée tôt semble judicieuse.

2.4.1 Flot de conception logiciel pour les systèmes sur puces

La conception des systèmes sur puce ou *SoC* tend vers une complexité hyper-croissante, depuis les débuts de l'industrie du semi-conducteur. Ces systèmes sont généralement conçus sur mesure en intégrant de nombreux blocs de processeurs et d'éléments matériels aux architectures dédiées, ces infrastructures ont satisfait depuis une dizaine d'années des besoins applicatifs très spécifiques manifestés par l'apparition des produits de types téléphones portables ou décodeurs numériques. Cependant, Cette croissance de ces besoins d'application, a engendré une compétitivité sans précédente au niveau des fenêtres temporelles d'accès au marché et les coûts élevés de fabrication dûs à la complexité de conception. Cette situation oblige certainement les concepteurs à s'orienter vers des circuits plus souples et adaptables afin de produire des familles de produits qui peuvent êtres personnalisés vers plusieurs types d'applications et plusieurs types de marchés.

2.4.1.1 Caractéristiques des systèmes SoC

Les systèmes sur puce ont été largement utilisés dans une variété d'applications tel que les téléphones portables, les GPS et les dispositifs multimédia à grand usage publique (récepteurs satellite, décodeurs des chaînes de télévision, ...). Ces systèmes présentent plus au moins des caractéristiques auxquels sont confrontés leurs concepteurs. La suite de cette section est consacrée à la présentation de ces caractéristiques.

Les systèmes sur puce deviennent de plus en plus complexes ; système programmable, blocs hardware dédiés, mémoire, inter-connections entre les différents composants, IPs intégrées dans un autre système plus complexe. Les besoins de conception évoluent sans cesse, les systèmes doivent être conçus de plus en plus vite, de plus en plus sûrement, et ils sont de plus en plus complexes.

Pour satisfaire au plus tôt le besoin de développement des logiciels embarqués et d'analyse de l'architecture des system-on-chip, les ingénieurs énumèrent les conditions suivantes comme nécessaires pour leur activité :

1. *vitesse* : les modèles utilisés pour les activités ci-dessus, exigent des millions de cycles simulés, dans certains cas. Il n'est pas acceptable d'attendre même un jour pour accomplir une simulation, particulièrement en cas de sessions interactives.
2. *exactitude* : une certaine analyse exige une exactitude cycle à cycle des modèles, afin d'obtenir des résultats fiables. On exige comme condition minimum que le modèle doit être assez détaillé pour exécuter le logiciel embarqué.
3. *modélisation légère* : n'importe quelle activité de modélisation, sans compter le codage RTL (qui est obligatoire de toute façon pour le silicium), doit être assez légère de sorte que son coût ne soit pas perçu comme plus élevé que les avantages résultants par la gestion de projet de SoC.

2.4.1.2 Flot de conception des systèmes sur puce

La conception des systèmes sur puce et d'une manière générale les circuits intégrés passe le plus souvent par plusieurs étapes et niveaux d'abstraction. Ce flot dépend de la complexité du système à concevoir et du nombre de tâches qu'il doit accomplir. Cette complexité ne cesse d'augmenter pour les raisons suivantes :

- la taille et l'espace de la puce,
- la puissance et la vitesse de calcul croissante,
- la consommation d'énergie et l'autonomie optimale du système,
- le temps de commercialisation de produits dans le marché,
- la rentabilité rapide de la production de ces systèmes.

Chapitre 3

SYSTEMC, un langage de conception au niveau système

3.1 Introduction

Les systèmes embarqués sont utilisés un peu partout et deviennent plus complexes, il devient donc de plus en plus important de vérifier la conception de ces systèmes avant leur mise en oeuvre. Un système embarqué est construit sur des composants électriquement configurables (FPGA = Field Programmable Logic Array) ou des circuits intégrés spécifiques (ASIC = Application Specific Integrated Circuit). Ces composants/circuits sont écrits soit dans un langage de description du matériel HDL (Harware Descritpion Language) ou dans un langage de programmation PL (Programming language).

La différence entre un HDL et un PL est que le compilateur, d'un code écrit en PL, traduit ce code source en code objet prêt à être exécuté sur un processeur tandis que le compilateur d'un HDL (habituellement appelé un synthétiseur) traduit ce code source en netlist équivalent des primitifs génériques de matériel pour mettre en application le comportement indiqué.

Aujourd'hui, les langages de description du matériel sont devenus extrêmement populaires dans la conception des systèmes embarqués (FPGA ou ASIC), puisque le même langage peut être utilisé par les ingénieurs à la fois pour la conception et aussi pour la vérification. Les langages les plus populaires sont Verilog et VHDL. Un langage HDL permet d'écrire la description comportementale ou fonctionnelle d'un système et de spécifier cette description par interconnexion de plusieurs composants IPs élémentaires (description structurelle ou physique). Il permet aussi de gérer des descriptions hiérarchisées et de valider par simulation les différents types de description.

L'inconvénient majeur des flots de conception classiques par les HDLs est la multitude des langages utilisés pour la modélisation des IPs. Pour l'intégration de ces IPs des transcriptions manuelles sont nécessaires ce qui augmente considérablement le risque des erreurs. De nombreux travaux de recherche ont été menés depuis plusieurs années pour faire face à la demande en langages d'abstraction plus élevés que VHDL et Verilog. En plus, les concepteurs des systèmes embarqués cherchent souvent à créer des spécifications exécutables, et pour cela ils utilisent des langages de programmation.

Récemment, C et C++ ont été proposés pour créer des spécifications exécutables. Cependant, ces langages sont conçus pour écrire des programmes informatiques, et pas pour spécifier du matériel. En plus, ils n'offre pas les fonctionnalités nécessaires à la spécification des systèmes embarqués comme les horloges et les signaux.

Pour résoudre ces problèmes trois voies sont proposées. D'une part, l'adaptation d'un langage de programmation comme C/C++ à la description matérielle et à la vérification, et d'autre part, l'extension par augmentation des langages HDL vers la programmation au niveau système. La troisième possibilité est la définition d'un nouveau langage adapté aux nouvelles exigences : HDL, vérification et spécification exécutable. SYSTEMC est l'une des solutions proposées pour résoudre ces problèmes. Le langage SYSTEMC, qui est une évolution du C/C++, a pris une certaine avance dans ce domaine. L'objectif de SYSTEMC est de retenir un seul langage dans un flot de conception.

Nous nous intéressons dans ce travail à la vérification des systèmes écrits en SYSTEMC. Nous présentons dans ce chapitre, via un exemple, le langage SYSTEMC, ses particularités et aussi ces principaux composants.

3.2 Présentation du langage SYSTEMC

3.2.1 Historique

SYSTEMC est un langage de spécification et de vérification, de la même famille que VHDL et Verilog, qui s'étend de la conception à la mise en oeuvre du matériel et du logiciel [**?**]. L'Open SYSTEMC Initiative (OSCI) [**?**] est le consortium majeur des EDA [1] et des sociétés IP majeur qui contribue au développement et à la distribution de SYSTEMC. Il a été proposé en septembre 1999. Depuis, SYSTEMC est devenu le standard de la conception au niveau système. En conséquence, plusieurs fournisseurs d'IPs commencent à fournir des modèles compatibles SYSTEMC de leur IPs.

Comme VHDL et Verilog, SYSTEMC est le fruit de la contribution de plusieurs sociétés. Synopsys met en 1989 son outil commercial Scenic dans le domaine libre, et crée la version 0.9 de SYSTEMC. La contribution de Frontier Design donne lieu à la version 1.0 et à la version 1.1 par CoWare en 2001 qui est la première version officielle de SYSTEMC [**?**]. L'OSCI, la société en charge de diffuser SYSTEMC, propose la version 2.0. Sa mission principale est de promouvoir

1. Electronic Design Automation : est l'association internationale des sociétés de développement d'outils de CAO et des services qui permettent aux ingénieurs de créer des produits électroniques.

SYSTEMC comme un standard pour la conception des systèmes sur puces (SoC). Depuis décembre 2005, SYSTEMC est standardisé auprès de l'IEEE sous le nom de IEEE 1666TM-2005 [**?**]. IEEE a approuvé ainsi une version de référence de SYSTEMC pour le standard IEEE1666-2005 qui correspond à la version 2.2.0 de SYSTEMC, qui est l'actuelle version stable.

La prochaine version 3.0 de SYSTEMC permettra la conception haut niveau des circuits intégrés complexes et systèmes sur puces. Sa bibliothèque de classes sera étendu pour permettre la modélisation des systèmes d'exploitation.

3.2.2 Objectifs

SYSTEMC est une extension du langage C++, sous la forme de classes représentant des éléments matériels d'un système appelés modules. Les utilisateurs de SYSTEMC peuvent développer des modèles SYSTEMC en utilisant le compilateur de C++. Une description SYSTEMC peut être structurelle, comportementale, de type flot de données ou un mélange de ces trois types de base.

Les descriptions comportementales en SYSTEMC doivent se faire en deux modes : mode concurrent et mode procédural. Dans le mode concurrent, les données se propagent dans des éléments dont le comportement est décrit par des déclarations qui ont un effet permanent. Dans le mode procédural, les données sont manipulées par des séquences d'instructions.

SYSTEMC offre un lien entre la conception du matériel et du logiciel ainsi. Il est constitué d'une librairie de classes C++ et d'un noyau de simulation pour des conceptions au niveau comportemental et RTL (Register Transfer Level). Il fourni ainsi un environnement de développement commun pour assister les concepteurs de parties logicielles qui utilisent les langages C/C++ et les concepteurs de parties matérielles qui utilisent les langages de description de matériel (HDL) comme VHDL et Verilog.

Les avantages de la description en SYSTEMC sont :

- *adoption d'un langage commun* : cela permet de garder un même langage d'un bout à l'autre du flot de conception tout au long du développement du système.
- *simulation du système* : SYSTEMC permet d'exécuter la description du système en lui imposant des séquences de stimuli. Les résultats issus de cette simulation sont comparés aux résultats prévus, permettant de détecter les éventuelles erreurs de conception. La co-simulation matériel/logiciel permet aux concepteurs d'évaluer et de vérifier les systèmes avant leur mise en oeuvre.
- *synthétiser le système* : la synthèse est le passage d'un domaine comportemental à un domaine structurel [**?**]. Elle consiste en une suite de raffinements successifs qui transforment une spécification comportementale algorithmique de haut niveau, en une liste de portes ou d'opérateurs numériques interconnectés. Plusieurs travaux ont été proposées dans la littérature pour synthétiser des architectures matérielles et logicelles à partir de descriptions SYSTEMC comme dans les cas de [**?**, **?**].

– *vérification formelle* : en plus de la vérification par simulation, différents modèles de vérification formels de descriptions SYSTEMC ont été proposés [**?**, **?**, **?**, **?**, **?**, **?**, **?**].

3.3 Architecture de SYSTEMC

L'architecture de l'environnement de conception de SYSTEMC est organisée en couche comme représentée sur la figure **??**. Elles sont structurées successivement du niveau le plus haut vers le niveau le plus bas.

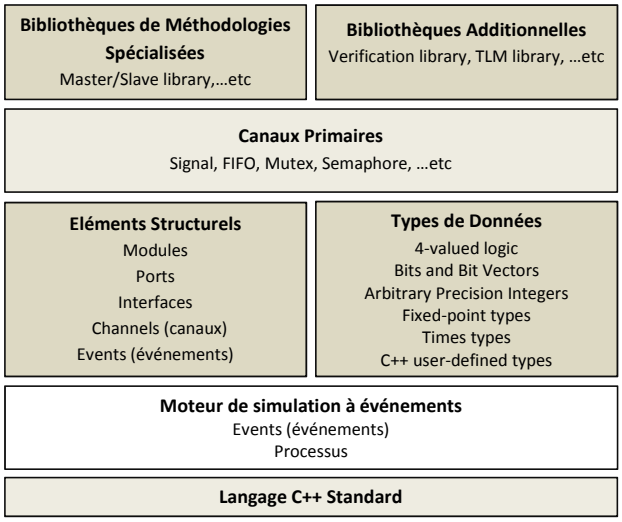

FIGURE 3.1 – Organisation de SYSTEMC [**?**]

Le paradigme orienté objet en SYSTEMC n'est pas un accessoire, mais central. SYSTEMC utilise l'approche orientée objet pour atteindre l'abstraction, la modularité, la compositionnalité, et la réutilisabilité. Il implémente toutes les fonctionnalités orientées objet : les classes (y compris les classes abstraites et templates), les objets, les méthodes (y compris les méthodes virtuelles), et l'héritage (y compris l'héritage multiples).

En SYSTEMC, les types de données de C++ ont été enrichis par des types de données adaptées à la modélisation matériel. Certains de ces types de données sont utilisés pour représenter du matériel (modules, ports, ...) et du logiciel (sémaphores, mutex), et d'autres pour représenter en même temps du matériel ou du logiciel (canaux, événements, ...).

3.4 Les particularités de SYSTEMC

Les particularités de SYSTEMC sont :

– *langage fortement typé* : Afin de modéliser du matériel, SYSTEMC implémente ses propres types de données basés sur C++. Ceci permet aux concepteurs d'écrire des spécifications

beaucoup plus lisibles et souvent de lever les ambiguïtés. Par exemple, pour définir les différents états d'un automate, il est plus agréable d'utiliser un type énuméré plutôt qu'un type entier. Chaque nouveau type de données en SYSTEMC possède ses fonctions de manipulation et de conversion. Il est aussi possible de surcharger des fonctions existantes d'un type de données pour les adapter à d'autres types. Un des avantages de SYSTEMC est que son compilateur permet de vérifier la bonne utilisation des types et de leurs fonctions de manipulation. La définition d'une multitude de fonctions de conversion et de manipulation pour chaque types de données est un inconvénient pour SYSTEMC.

– *langage de description mixte* : SYSTEMC permet de modéliser des systèmes matériels, logiciels, ou mixtes. Il est souvent utilisé dans la conception des systèmes embarqués et particulièrement les systèmes sur puces.
– *langage de simulation* : la simulation est l'interprétation des déclarations des HDL dans le but de produire des sorties lisibles, tel qu'un chronogramme qui décrit le comportement d'un matériel avant sa fabrication. La simulation est similaire à l'exécution d'un programme dans un langage classique de haut niveau, tels que Java Script ou BASIC. Le standard de SYSTEMC [**?**, **?**] spécifie, en plus de la grammaire et de la syntaxe du langage, la sémantique de simulation associée à l'utilisation de chacune des instructions. La couche de base de SYSTEMC est basée sur un noyau de simulation à événements.
– *langage orienté objet* : SYSTEMC est un langage de description de matériel tout comme VHDL et Verilog. C'est une extension du C++ permettant une modélisation orientée objet de systèmes embarqués sous forme de blocs.

3.5 Principaux composants SYSTEMC

Les éléments de base de SYSTEMC sont les modules et les canaux de communications. Les modules comportent des processus (les fonctionnalités), des ports de communications, des variables partagées entre les processus, mais aussi d'autres modules pour avoir une représentation hiérarchique de l'architecture. Les différents modules sont connectés entre eux par des canaux de communications liés au port. Ces canaux de communications peuvent véhiculer des types primitifs (bit par exemple) mais aussi des structures définies par l'utilisateur pour s'abstraire des protocoles de communications.

3.5.1 Les modules

L'entité de base pour décrire un système en SYSTEMC est appelé *module*. Il est composé d'un nombre quelconque d'entrées, de sorties, d'entrées/sorties, ou de processus. Le module est l'élément de base pour décrire un système en SYSTEMC et s'obtient par dérivation de la classe SC_MODULE. Le code suivant représente un exemple d'une déclaration d'un module en SYSTEMC :

```
1 SC_MODULE (my_mod)
  {
3    // ... déclaration des ports, données, signaux internes
```

```
    // ... déclaration des process et fonctions membres
5   // ... définition du constructeur du module
}
```

Un module peut contenir des structures de données C++ aussi bien que d'autres modules SYSTEMC, et de canaux de communication entre ces modules (signaux, ...). La figure **??** représente comment les modules se communiquent en SYSTEMC.

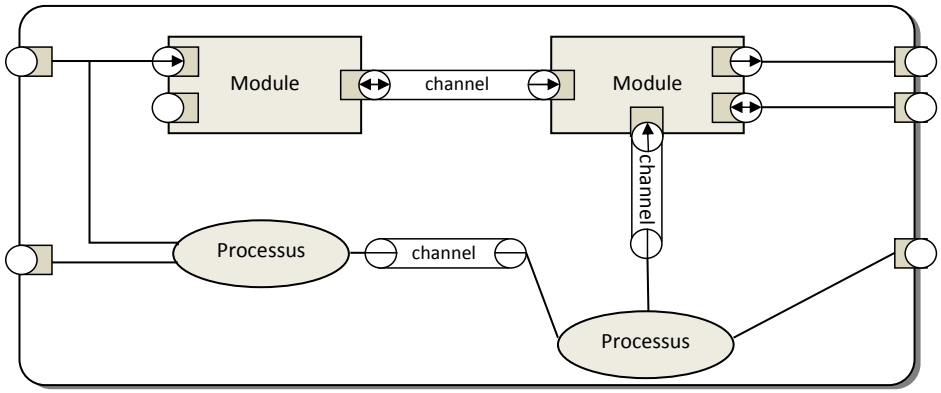

FIGURE 3.2 – Organisation structurelle d'un système en SYSTEMC

Les canaux et les ports sont utilisés comme moyens de communication entre les modules. Chaque classe implémentant un module doit avoir au moins un constructeur.

3.5.2 Les processus

Les processus en SYSTEMC sont similaires à ceux de Verilog et VHDL. Ils s'exécutent de façon concurrente et ils définissent le comportement des modules. Les processus ne sont pas hiérarchiques, c'est-à-dire un processus ne peut pas appeler un autre processus directement. Chaque processus SYSTEMC dispose de listes de sensibilité (signaux) qui causent le processus qui puisse être invoqué, dès que la valeur d'un signal dans cette liste est modifiée. Les signaux sont utilisés donc pour les communications inter-processus.

Il existe trois types de processus en SYSTEMC : les méthodes (SC_METHOD), les threads (SC_THREAD) et les threads horloges (SC_CTHREAD).

- SC_METHOD : un processus de type SC_METHOD s'exécute et rend la main au noyau de simulation. Il ne peut pas être suspendu et réactivé comme les autres processus.

- SC_CTHREAD : un processus de type SC_THREAD peut être suspendu et réactivé, en utilisant la méthode wait(). Il s'exécute en boucle infinie et il pourrait être suspendu grâce à la méthode wait(x). Ces processus sont plus lents en simulation que les SC_METHOD à cause de la sauvegarde du contexte.

– SC_THREAD : La liste de sensibilité des processus de type SC_CTHREAD se réduit à un front d'une horloge, et il dispose de deux constructions supplémentaires : watching(x) [2] et wait_until(x.delayed()==true) [3].

Le comportement d'un processus SYSTEMC est défini par une procédure C++ qui lui est associée. La simulation d'un modèle SYSTEMC consiste donc à exécuter de façon pseudo concurrente le code C++ de chacun de ses processus. La version 2.1.0 de SYSTEMC distingue deux catégories de processus :

– Les processus sans contexte qui s'exécutent dans le contexte de l'ordonnanceur du simulateur.

– Les processus avec contexte qui ont leurs propres contextes d'exécution (pile, tas,...).

Les processus se distinguent aussi en fonction de la manière dont ils sont créés. Un processus statique est créé par l'ordonnanceur du simulateur, durant la phase d'élaboration, en appelant le constructeur du module qui contient la méthode C++ associée à ce processus.

Le code suivant représente un additionneur décrit en SYSTEMC :

```
SC_MODULE (S)
{
  sc_in<T> in1, in2 ;
  sc_out<T> out ;

  SC_CTOR ( S )
   {
    SC_METHOD ( behavior ) ;
    sensitive << in1 << in2 ;
   }

  private :
    void behavior ( )
    {
      out.write ( in1.read() + in2.read() ) ;
    }
};
```

Un processus dynamique est créé en appelant la fonction SC_SPAWN soit par l'ordonnanceur pendant la phase d'élaboration, soit par un autre processus au cours de la phase de simulation. Cette méthode de création de processus a été utilisée pour autoriser un processus SYSTEMC à s'exécuter avec prise en compte de plusieurs communications parallèlement. L'ordonnancement de l'exécution des processus est assuré par le noyau de simulation en fonction de l'état courant de chaque processus.

Comme illustré sur la figure **??**, un processus peut être dans l'un des trois états suivants : élu, prêt ou bloqué. Lorsqu'un processus est dans l'état élu, il est exécuté jusqu'à ce qu'il demande explicitement à passer dans l'état bloqué. Lorsqu'un processus passe dans l'état bloqué sa liste de sensibilité est mise à jour. Une liste de sensibilité définit les événements qui doivent survenir pour

2. réinitialisation quand x est vérifié
3. process suspendu jusqu'à ce que la condition soit vérifiée

réveiller un processus bloqué pour qu'il puisse devenir prêt. Lorsqu'un processus devient prêt, donc il est prêt à être exécuté. Cependant, il n'est exécuté que lorsqu'il passe à l'état élu. La politique de sélection d'un processus n'est pas définie par le standard et dépend donc de l'implémentation de la bibliothèque SYSTEMC utilisée.

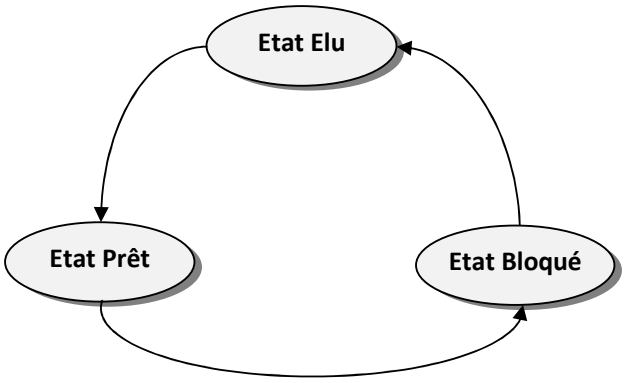

FIGURE 3.3 – Diagramme d'état d'un processus SYSTEMC

3.5.3 Les Ports

Un port, classe SC_PORT, est un élément central dans l'interconnexion, c'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 manière indirecte, à partir d'un port, en exploitant la notion d'interface. Les ports fournissent un moyen de description d'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 l'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.

3.5.4 Les Interfaces

Une interface est un ensemble de primitives de communication, sans définir leurs implémentations. Son rôle est la définition des primitives de communication d'un canal qui sont accessibles par un port. Ce dernier permet de transmettre les appels de ses méthodes vers le canal auquel il est connecté. Cette technique de transmission permet directement le remplacement d'un canal par un autre canal implémentant les mêmes interfaces et de permettre par conséquent une indépendance totale entre les parties fonctionnelles et de communication d'un modèle SYSTEMC.

3.5.5 Les Canaux

Les canaux de communication sont des éléments permettant la modélisation de la communication au sein d'un système. SYSTEMC définit plusieurs classes de canaux de communication de base tels que les SC_SIGNAL ou les SC_FIFO. SC_SIGNAL est une classe de canaux, dont chacun modélise le comportement d'un fil selon l'hypothèse de discrétisation des valeurs et du temps des systèmes synchrones. La classe SC_FIFO modélise les fils d'attente FIFO, elle peut être instanciée avec une taille supérieure ou égale à un élément.

Pour la synchronisation d'exécution des processus qui appel à ses primitives de communication, un canal nécessite des événements ainsi que la technique *request-update*. Cette technique de synchronisation consiste à découper l'accès au canal en deux étapes successives pour permettre de résoudre équitablement les multitudes d'accès concurrents à ce canal. Cependant, elle n'est pas toujours adaptée pour modéliser les systèmes asynchrones.

Un avantage majeur de SYSTEMC est sa possibilité d'extension à travers l'ajout de nouveaux canaux de communication à l'aide de la classe de base SC_PRIM_CHANNEL qui définit l'ensemble des caractéristiques de base des canaux de communication de SYSTEMC (mécanisme de connexion à un port, méthode *request-update*,...etc). La figure **??** montre à travers un diagramme UML, la manière de définition d'un nouveau canal de communication (*new_channel*) à partir de la classe SC_PRIM_CHANNEL. La nouvelle classe *new_channel* hérite des deux interfaces INTERFACE_IN et INTERFACE_OUT en permettant de se connecter à travers n'importe quel port de ces deux interfaces.

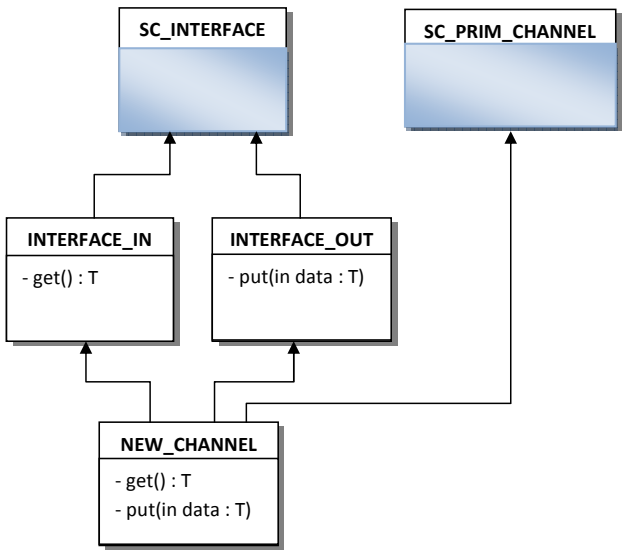

FIGURE 3.4 – Diagramme UML des cannaux SYSTEMC

3.5.5.1 Evènements, Sensibilité et Notification

Les événements sont les éléments de base utilisés pour synchroniser l'exécution des processus. Pour qu'un processus en attente d'un événement soit réveillé, il faut que cet événement soit notifié. La notification d'un événement s'effectue en deux étapes : la demande de notification et le déclenchement de la notification.
La demande de notification d'un événement consiste à définir à quelle date le déclenchement de la notification aura lieu. Une demande de notification s'effectue avec la méthode *notify* d'un événement qui peut être appelée des trois façons suivantes :

- *e.notify()* est une demande de notification immédiate de l'événement *e*. Le déclenchement de la notification s'effectue au même instant de simulation que la demande de notification.

- *e.notify(*SC_ZERO_TIME*)* est une demande de notification retardée de l'événement *e*. Le déclenchement de la notification s'effectuera au prochain delta-cycle.

- *e.notify(200,* SC_NS*)* est une demande de notification temporisée de l'événement *e*. Le déclenchement de la notification s'effectuera dans 200 nanosecondes.

Un événement ne peut avoir, à tout instant, qu'une seule demande de notification en attente. Les règles suivantes permettent de résoudre les notifications multiples à un événement qui n'ont pas encore été déclenchées : Une demande de notification plus récente va toujours écraser une demande plus tardive, sachant que :

- Une notification immédiate est plus récente qu'une notification retardée,

- Une notification retardée est plus récente qu'une notification temporisée.

La méthode CANCEL d'un événement permet d'annuler une demande de notification temporisée ou retardée. Par contre il est impossible d'annuler une demande de notification immédiate. Lorsque la date de déclenchement de notification d'un événement est atteinte, toutes les listes de sensibilité des processus en attente de cet événement sont mises à jour. En outre, les processus endormis, dont la liste de sensibilité ne contient plus d'événement en attente, sont réveillés.

Une différence fondamentale entre SYSTEMC et les autres langages de description de matériel (VHDL et Verilog) est de ne pas avoir un comportement complètement déterministe : malgré les règles de résolution des notifications multiples, le comportement d'un modèle SYSTEMC, utilisant des notifications immédiates, peut dépendre de l'ordonnancement de ses processus [**?**] Dans le cadre de la modélisation de circuits asynchrones, cette contrainte est un avantage car elle permet de modéliser fidèlement leurs comportements.

SYSTEMC utilise la classe SC_EVENT pour les modèles d'évènements. Les évènements représentent les éléments de base pour une simulation pilotée par évènement. Un évènement n'a pas de temps ou durée fixe. Avec ces évènements, on peut effectuer deux actions : attente *wait* sur eux ou *fire*. Donc un évènement est l'outil principal de synchronisation entre les différents processus pendant une simulation. Un processus peut être déclaré sensible à un ou plusieurs évènements. A

l'arrivée d'une occurrence d'évènement qui figure dans la liste de sensibilité d'un processus, le noyau de simulation lance l'exécution de ce processus.

3.6 Simulation par SYSTEMC

Le simulation en SYSTEMC permet d'exécuter de manière concurrente l'ensemble des processus d'un modèle SYSTEMC dans le but de valider le comportement du système modélisé. Cette simulation se base sur un modèle de temps discret plus adapté à la validation de modèles à base de composants matériels synchrones. La simulation se déroule en plusieurs étapes durant lesquelles des possibilité de garder trace, peuvent être explorées afin d'enregistrer l'activité de ces canaux.

La simulation en SYSTEMC s'appuie sur un modèle de temps absolu. La bibliothèque utilise un nombre entier non signé de 64 bits pour représenter le temps.

3.6.1 Le modèle du temps

Le modèle de temps discret d'une simulation SYSTEMC, utilise une échelle de temps réel (seconde, microseconde, nanoseconde,...etc), mais chaque instant de cette simulation peut être divisé en un nombre quelconque de « *delta-cycles*» (échelle de temps virtuel pour modéliser l'activité pseudo-parallèle des processus). Un *delta-cycle* ressemble à un intervalle de temps infiniment petit qui ne fait pas progresser le temps physique du simulateur (voir figure **??**), il est constitué de trois phases principales qui sont :

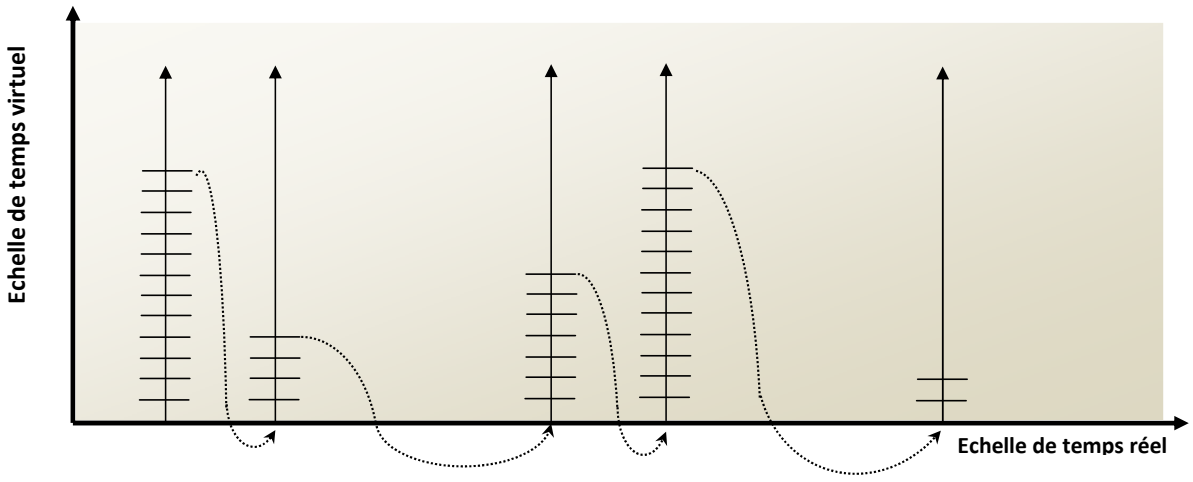

FIGURE 3.5 – Diagramme des échelles de temps (simulation SYSTEMC)

3.6.1.1 La phase d'évaluation

Durant cette phase tous les processus qui sont dans l'état prêt sont exécutés d'une manière séquentielle. Cette phase se termine lorsque'il n'y a plus de processus prêt.

3.6.1.2 La phase de mise à jour

Pendant cette phase, l'ordonnanceur du simulateur lance l'exécution des méthodes *update()* des canaux primaires dont les méthodes *request_update()* ont été appelées durant la phase d'évaluation.

3.6.1.3 La phase de notification retardée

Au cours de cette phase, l'ordonnanceur déclenche les demandes de notification retardées qui ont été effectuées durant les phases d'évaluation et de mise à jour. Le concept du delta-cycle est souvent employé afin de déterminer à tout instant, le comportement d'un modèle décrit dans un langage de description de matériel. En SYSTEMC, les phases d'évaluation et de mise à jour sont utilisées en se basant sur la technique *request-update* pour déterminer avec précision les accès concurrents à un canal SC_SIGNAL_RESOLVE.

Généralement, les *delta-cycles* sont utilisés par les canaux SYSTEMC pour que leurs valeurs de sortie restent constantes au cours d'une phase d'évaluation. Ce qui permet ainsi de respecter les hypothèses de discrétisation du temps des systèmes synchrones. Cependant, l'utilisation des ces *delta-cycles* ne permet pas de modéliser correctement le comportement des systèmes asynchrones.

3.6.2 Le Noyau SYSTEMC

L'exécution de la simulation en SYSTEMC est gérée par le noyau SYSTEMC. L'ordonnancement du noyau ne gère ni la préemption, ni la notion de priorité entre les processus. Il lance en exécution les processus prêts à être exécutées car ils ont été réveillés au temps précédent par un évènement (changement de valeur d'un signal qui figure dans la liste de sensibilité du processus, fin d'une attente temporelle,...etc). Ce noyau permet de :

- modéliser l'ordonnancement des processus,
- gérer la préemption des processus,
- introduire un système d'exploitation temps réel générique,
- annoter en délai des processus.

3.7 Exemple de description en SYSTEMC

L'exemple choisi consiste à simuler un système de mémoire en utilisant un processeur comme stimulateur (générateur de requètes de lecture/écritre, adresses et données mémoire). Le système global est composé de deux module : une mémoire de type RAM et un processeur CPU. La figure **??** illustre ce système.

Le code suivant représente les descriptions SYSTEMC du module RAM :

```
1 // Le fichier entête du module mémoire « ram.h »
```

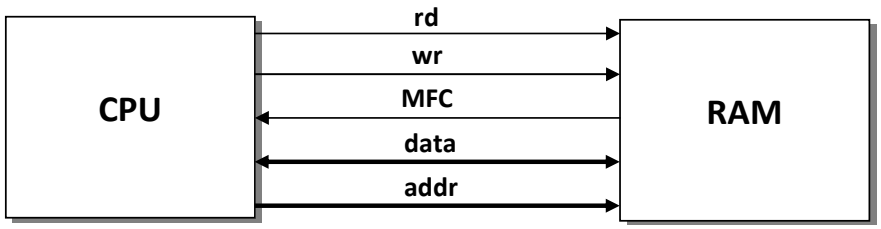

FIGURE 3.6 – Système mémoire - processeur

```
#include"systemc.h"
SC_MODULE(memoire)
{
 //Déclaratin des ports
    sc_in <bool> clck;
    sc_in <bool> rd;
    sc_in <bool> wr;
    sc_in <int> addr;
    sc_inout_rv <16> data;
    sc_out <bool> MFC;
    int i;
 //Déclaration des méthodes
    void read_write();
    sc_lv <16> ram_data[256];
 //Déclaration des constructeurs
    SC_CTOR(memoire)
      {
        SC_THREAD(read_write);
        sensitive<<wr<<rd<<addr<<data<<clck;
        //Initialisation à 0 des mots mémoire
        for (i=0;i<256;i++)
        {
          ram_data[i]=0;
        }
      }
};
```

```
// Le fichier implémentation du module mémoire « ram.cpp »
#include "systemc.h"
#include "ram.h"
   sc_lv <16> ram_data[256];
   void memoire::read_write()
   {
     data.write (ram_data[addr.read]);
     ram_data [addr.read()] = data.read();
   }
```

Le code suivant représente les descriptions SYSTEMC du module processeur CPU :

```
// Le fichier entête du module processeur « cpu.h »
#include"systemc.h"
 SC_MODULE(cpu)
 {
  //Déclaration des ports
    sc_in <bool> clck;
    sc_inout_rv <16> data;
    sc_out <int> addr;
```

```
    sc_in <bool> MFC;
  //Déclaration des méthodes
    void MFC ();
  //Déclaration des constructeurs
    SC_CTOR (cpu)
    {
      SC_THREAD (MFC);
      sensitive <<MFC<<clck;
    }
};
```

```
// Le fichier implémentation du module processeur « cpu.cpp »
#include "systemc.h"
#include "cpu.h"
#include <stdlib.h>

  sc_lv <16> ram_data[256];

  void cpu::MFC()
  {
    while (true)
  {
  srand(rand());
  wr=rand()%2;
  rd=rand()%2;
  addr=rand()%256;
  cout << " wr= " << wr << " rd= " << rd << endl;
  {
    if (!(rd xor wr))
  {
  cout << "erreur? lecture écriture au même temps " << endl;
  }
    if ((rd==1)&&(wr==!rd))
  {
    cout << " lecture " << endl;
    data = ram_data [addr.read()];
    cout << " addr= "" << addr.read() << " data= " << data.read()
    << endl;
  }
  else
  {
    if ((wr==1)&&(rd==!wr))
  {
    data = rand();
    cout << " Ecriture " << endl;
    ram_data[addr.read()] = data;
    cout << " ram_data[ " << addr.read() << "] = " <<
    ram_data[addr.read()] << endl;
  }}
  wait();
  }
  }
```

Le code suivant représente la description SYSTEMC du module programme principal de simulation (le simulateur) :

```
// Le fichier du programme principal de simulation
#include"systemc.h"
#include"ram.h"
#include"cpu.h"
#include<stdlib.h>
```

```
7  int sc_main (int argc, char *argv[])
     {
9      memoire RAM ("memoire");
       cpu CPU ("processeur");
11     //cpu avec ram
       sc_signal <bool> cpuram_sigrd;
13     sc_signal <bool> cpuram sig_wr;
       sc_signal <bool> cpuram sigout;
15     sc_signal <short int> cpuram_sigin;
       sc_signal <short int> cpuram_sig_in;
17     sc_signal <short int> cpuram_sigout;
       //Connexion des ports du module ram
19     RAM.rd (cpuram_sigrd);
       RAM.wr (cpuram_sigwr);
21     RAM.addr (cpuram_sig);
       RAM.data (cpuram_sig);
23     RAM.MFC (cpuram_sigout);
       cout<<"démarrage de la simulation........\n ";
25     sc_start(-1);
       return 0;
27 }
```

3.8 conclusion

Nous avons présenté dans ce chapitre SYSTEMC qui est un langage de conception au niveau système. Ce langage, incluant un noyau de simulation, permet la description d'un système à plusieurs niveaux d'abstraction et aussi le passage d'un niveau à un autre. La vérification des descriptions SYSTEMC est réalisée grâce à son compilateur (simulateur). Etant donnée que la simulation ne permet pas de découvrir toutes les erreurs de conception, nous proposerons dans le chapitre **??** une approche formelle pour vérifier des descriptions SYSTEMC.

Chapitre 4

Vérification des systèmes embarqués

4.1 Introduction

La conception des systèmes de plus en plus complexes se heurte à des problèmes de vérification. La vérification cherche à assurer que le système construit effectue correctement les fonctions spécifiées. Cette tâche doit être intégrée dans tout le processus de conception du système, afin de minimiser le coût de développement. Le développement d'un produit sur un marché à forte concurrence impose des contraintes de résultats fiables et rapides. Le retard de commercialisation d'un produit, ou la détection d'une erreur lorsque le système est commercialisé, ternit l'image de la société développant le produit. La vérification de systèmes est donc une nécessité absolue pour les concepteurs de systèmes informatiques. Elle doit s'intégrer dès les premières phases de conception du système, afin de détecter les erreurs au plus tôt. Elle doit intervenir dans toutes les phases de développement du produit afin de s'assurer que le système construit correspond bien à sa spécification.

Deux approches de base ont été alors utilisées pour la vérification des systèmes. Une première technique de vérification consiste en l'élaboration d'un prototype du système, qui est testé dans son environnement. Il s'agit là de la simulation du comportement du système : le comportement simulé est alors comparé au comportement attendu. S'il y a adéquation, le système est vérifié. Malheureusement, la simulation ne peut évaluer tous les comportements possibles d'un système complexe, du fait de l'explosion combinatoire du nombre de comportements possibles de ces systèmes. De ce fait, la simulation ne peut vérifier que le comportement du système sur un jeu de tests extrêmement réduit au regard de tous les comportements possibles du système.

D'autres voies de vérification ont été étudiées : elles consistent à remplacer la vérification expérimentale du système par une preuve de sa correction. On cherche à montrer formellement que le système est en adéquation avec sa spécification, ou qu'il vérifie un ensemble de propriétés décrivant partiellement la spécification. Au cours de ce chapitre nous présentons d'abord les différentes approches utilisées pour la vérification des systèmes et nous nous concentrons beaucoup plus sur la vérification formelle, puis nous présentons quelques méthodes pour vérifier et valider des spécifications décrites dans le langage SYSTEMC.

4.2 Vérification par simulation

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

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

La simulation est un type de modélisation d'un système qui peut être mise en oeuvre sur ordinateur. Elle permet de définir la manière dont ce système évolue en fonction du temps, c'est-à-dire le comportement du système, et donc elle peut être vue comme un processus de modélisation dans lequel une réalité dynamique est imitée grâce à des actions sur ordinateur. Elle sert comme un moyen direct d'observation du comportement hypothétique du système dans des conditions différentes à partir d'un état initial choisi [**?**].

La simulation en terme général ne permet pas d'établir une preuve de correction du comportement au sens mathématique ou formel, mais elle facilite la compréhension des aspects sélectionnés du comportement du modèle, et aussi des relations entre les composants du systèmes modélisé. L'exécution d'une simulation peut être réitérée plusieurs fois pour gagner en confiance au comportement correct du modèle, en examinant différents modes de travail, différentes valeurs des données d'entrée, ou bien différents états initiaux. Par conséquent, la simulation est une technique expérimentale basée sur essais et analyse d'erreurs, qui facilite l'évaluation et la validation des stratégies et des décisions concernant l'architecture ainsi que la fonctionnalité du système en cours de développement.

Le comportement réel ou proposé du système est modélisé par le biais d'une description (un modèle) qui peut être exprimée sous la forme de description structurelle ou comportementale, en utilisant des langages de description de matériel (HDL).

La figure **??** présente une vue globale de l'environnement de la simulation dans lequel la description d'un système, avec les vecteurs de stimuli (test), est traité par le simulateur qui offre un

moyen de déboguer le code de la description ainsi que de visualiser et de stocker les résultats de la simulation de ces vecteurs de stimuli.

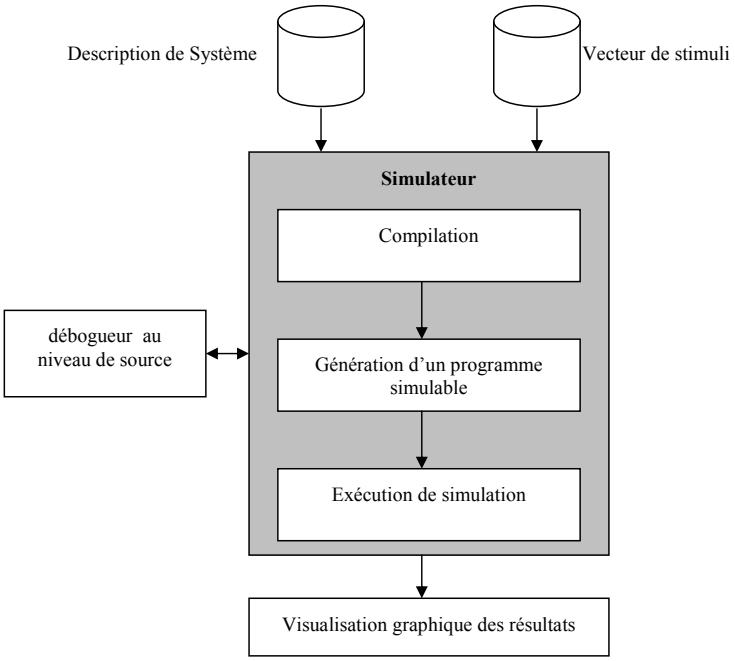

FIGURE 4.1 – Environnement de simulation

4.2.1 Étapes d'exécution d'un simulateur

Les étapes logiques à suivre dans une simulation par un simulateur sont :

- Analyse du code source au niveau syntaxique et sémantique.
- Génération du code associé à une description donnée en langage de description de matériel.
- Assemblage du code spécifique à une description donnée et du code de simulateur (noyau du simulateur) qui est une partie commune à toutes les descriptions de système dans une technologie particulière de simulation. Cet étape réalise l'association des liens et la lecture des données spécifiques au circuit simulé.
- Élaboration de la hiérarchie du système pendant laquelle est construite une structure de données propre à la description du circuit. Cette structure est utilisée dans l'exécution de la simulation.
- Initialisation de la structure de données.
- Exécution de chaque processus de la structure de données jusqu'à son interruption.
- Exécution des cycles de simulation qui produisent les résultats de simulation.

Les étapes cotées ainsi sont censées couvrir différents types de simulateurs acceptant d'utiliser des langages de description de matériel : simulateurs compilés, interprétés ou dirigés par les événements ou le temps.

4.2.2 Le compilateur

Le premier composant, le compilateur, sert à l'analyse de la syntaxe et de la sémantique d'une description exprimée en langage de description de matériel (ici langage de haut niveau) en une représentation d'un niveau plus bas, nommée représentation cible. Cette dernière peut être formulée en utilisant les formats ou les représentations suivantes : du code C, du code assembleur, un code exécutable sur la machine spécifique sur laquelle le code sera directement exécuté, et un format intermédiaire. Le choix du format de la représentation cible dépend de l'ensemble des outils employés dans le simulateur, notamment si le compilateur utilise un outil d'édition de liens ou un assembleur ou s'il génère directement du code exécutable.

4.2.3 Génération du programme simulable

4.2.3.1 Processus noyau du simulateur

Le processus noyau (*simulator kernel process*) est une partie de simulateur qui est commune à tous les éléments simulés. Le noyau est associé à la partie appropriée du système simulé pour construire un programme simulable en utilisant trois méthodes différentes : la première méthode consiste en l'association des liens avec le code objet spécifique au système, dans la deuxième méthode le noyau lit progressivement le fichier contenant l'information sur le circuit, tandis que la troisième méthode est une combinaison des deux précédentes. Cette distinction donne les types compilés ou interprétés de simulateurs.

4.2.3.2 Simulateur de type interprété

Le simulateur interprété contient tout le code exécutable dans le noyau de simulateur. La partie spécifique au système n'est pas intégrée au noyau, mais reste sous la forme de données qui sont interprétées dans le processus de simulation.

4.2.3.3 Simulateur de type compilé

Plus le code exécutable est généré uniquement pour le circuit à simuler, plus le simulateur devient un simulateur compilé. Le simulateur entièrement compilé est celui pour lequel on ne peut pas fournir de vecteurs de stimuli après l'étape de sa génération. La construction d'un simulateur de type compilé génère un code exécutable qui reflète le comportement complet du système pour le jeu de vecteur de stimuli fourni.

Les simulateurs de type compilé sont considérés comme plus rapides que les simulateurs de type interprété pour les longues exécutions de simulation car toute l'information sur l'exécution de la simulation est déjà connue au moment de la construction du simulateur. Cependant, chaque

changement de vecteurs de stimuli appliqués au système simulé déclenche une nouvelle compilation et une reconstruction du simulateur et cela présente un coût supplémentaire considérable, spécialement quand il s'agit de circuit d'une taille importante et d'exécutions courtes de la simulation. Dans ces cas particuliers les simulateurs interprétés offrent le meilleur rendement. Une solution qui peut être un remède à ce problème consiste à appliquer des compilateurs incrémentiels [**?**] permettant une compilation partielle du code après les modifications mineurs, en mettant les vecteurs de test dans une unité *test bench* interfacée avec le module du circuit.

La génération du programme simulable comporte trois phases qui préparent l'exécution de cycles de simulation : l'élaboration, l'initialisation et l'exécution.

Phase 1 - Elaboration : l'élaboration est un processus de création, à partir de la description du circuit, de la structure de données exécutée pendant la simulation. Cette structure de données sous la forme de code exécutable est l'ensemble des processus, créés pour chaque objet de la description initiale, interconnectés en réseaux.
L'élaboration d'une hiérarchie d'entités de conception commence par l'élaboration de l'instruction de bloc externe par l'entité et est suivie par l'élaboration de chaque instance de ces composants internes.

Phase 2 - Initialisation de la structure de données : au début de l'initialisation, le temps courant est mis à 0. Les vecteurs initiaux sont affectés, à partir des valeurs effectives, à tous les signaux.

Phase 3 - Exécution initiale des processus : la dernière phase de la construction d'un simulateur est la phase d'exécution de chaque processus de la structure de données, une seule fois jusqu'à son interruption. Le temps du premier cycle de simulation est calculé selon les règles des étapes du cycle de simulation.

4.2.4 Exécution de la simulation

L'objectif de la simulation est la modélisation de la dynamique des systèmes, c'est-à-dire l'évolution des systèmes en fonction du temps. La façon dont l'avancement du temps est modélisée, permet de distinguer les principales méthodes d'exécution de la simulation. Nous en distinguons trois : la simulation dirigée par les événements (*event-drive simulation*), la simulation dirigée par le temps (*time-drive simulation*) et la simulation dirigée par l'horloge (*cycle-based simulation*). La première méthode est utilisée dans les simulateurs basés sur les langages des description de matériel VHDL, Verilog et récemment SYSTEMC, comme définis par les standards [**?**], [**?**] et [**?**]. Certains types de modèles, notamment les modèles des systèmes synchrones peuvent être simulés en utilisant la simulation dirigée par l'horloge où l'avance de la simulation est associée au progrès de l'horloge du système (exemple : les simulateurs intégrés dans les environnements d'ESTEREL). La simulation dirigée par le temps est une simulation dans laquelle le temps est incrémenté d'une valeur fixe.

4.3 Vérification formelle

Les approches de vérification non-formelle (simulation et test), ne sont pas très efficaces pour les systèmes complexes. Elles sont très coûteuses, trop lentes et non exhaustives, voir même impossible de tester ou simuler toutes les entrées possibles d'un système dont le nombre d'entrées est excessivement grand. Les besoins de produire de plus en plus vite et à moindre coût sont incompatibles avec des systèmes partiellement vérifiés. La simulation et le test ne garantissent pas une conception sans erreurs, c'est pourquoi les méthodes formelles sont intégrés au flot de conception.

Définition 5 (vérification formelle) *La vérification formelle consiste à démontrer qu'un système va se comporter comme attendu. Cette démonstration est exhaustive par nature et détecte les problèmes même pour des configurations auxquelles le concepteur n'aurait pas pensé [**?**].*

La vérification formelle peut, soit comparer les données informatiques qui vont permettre la réalisation d'un futur système avec sa spécification, soit vérifier que ce futur système répond à certaines contraintes caractérisant son bon fonctionnement. Un objectif souvent formulé à l'encontre de la vérification formelle, consiste à suggérer l'utilisation des outils de synthèse dans lesquels un futur système est automatiquement construit à partir de sa spécification. Cette approche alléchante n'est pas autant à l'abri des erreurs qu'il n'y paraît à première vue. En effet, ces outils doivent être adéquats pour prendre en compte les nouvelles possibilités technologiques et les nouvelles techniques de conception. Il en résulte qu'ils n'ont pas toujours le temps d'être bien adaptés. De plus, l'optimalité des systèmes qu'ils produisent n'est pas aussi bonne que celle obtenue par une conception manuelle. L'existence d'outils de vérification indépendants de la chaîne de conception présente donc de nombreux avantages.

La vérification formelle d'un système cherche à caractériser le comportement de ce dernier par une analyse mathématique, contrairement aux techniques de simulation ou simulation symbolique qui représentent une analyse expérimentale du système. La vérification formelle cherche à démontrer l'équivalence ou l'inclusion de deux modèles formels, l'un représentant une spécification et l'autre une implémentation du système, ou bien les deux représentant deux implémentations différentes d'un même système.

Une implémentation représente le modèle qui devrait être vérifié afin de savoir s'il correspond ou pas à sa description initiale (spécification). La spécification représente les propriétés auxquelles se réfère le processus de vérification pour conclure de la conformité d'un système. Elle peut être exprimée de différentes manières telles qu'une description comportementale, une description structurelle abstraite ou un ensemble de propriétés temporelles. Du fait que le raisonnement est formel, un formalisme est indispensable pour exprimer les trois entités : l'implémentation, la spécification et la relation entre elles.

Un certain nombre de techniques de descriptions formelles sont apparues. Elles sont basées sur des langages de description formelle et disposent d'outils de vérifications automatiques de ces descriptions. Un langage est formel si sa sémantique est précise et non ambiguë. Cette base formelle de langage permet la vérification et la correction de la spécification. Un état de l'art sur les langages formels sera présenté dans le chapitre **??**.

Une fois vérifiées, ces descriptions formelles sont implantées en matériel ou en logiciel. L'ingénierie des systèmes informatiques matériels s'est également préoccupée d'intégrer des techniques de vérification dans son processus de synthèse. Si les niveaux de descriptions proches du matériel disposent d'outils de preuve formelle, les descriptions d'un plus haut niveau sont encore trop souvent vérifiées uniquement par la simulation. Des techniques complémentaires de vérifications ont été proposées et sont utilisées pour les descriptions plus abstraites du matériel. Elles sont présentées dans la section suivante.

L'utilisation d'un unique langage de description de matériel tout au long du cycle de conception est une données supplémentaire à prendre en compte dans l'établissement de méthodes de vérification : un langage unique, permettant de représenter le système dans les différentes phases de conception, simplifie le processus de développement. Intégrer les méthodes de vérification aux types de descriptions utilisées lors de la conception permet de se placer au niveau des besoins des concepteurs.

Les méthodes de preuve formelle nécessitent l'élaboration d'un modèle formel sur lequel le raisonnement mathématique de construction de la preuve peut s'appliquer. Au modèle formel est associé une sémantique formelle, qui peut être opérationnelle, dénotationnelle ou axiomatique. Le modèle formel le plus couramment employé est la logique. Ce modèle évalue la structure de propositions et déductions représentées par la sémantique formelle. Différentes logiques ont été proposées. Citons les logiques du premier ordre, d'ordre supérieur, les logiques temporelles, ...etc.

4.3.1 Approches de vérification formelle

Trois approches de vérification formelle ont été proposées :

- Approche basée sur des méthodes déductives (*Theorem-Proving*) : elle est basée sur la construction de théorèmes à partir de faits et de règles d'inférence, avec l'aide d'outils de preuve automatique de théorèmes. Ces méthodes consistent à démontrer, généralement dans un formalisme d'ordre supérieur, qu'une description détaillée a la même fonctionnalité qu'une spécification de haut niveau.
- Approche basée sur des méthodes booléennes (*Model-Checking*) : elle repose sur une exploration symbôlique de l'espace des états atteignables par le système en utilisant les machines à états finis (FMS). Ces méthodes consistent à montrer l'équivalence de deux descriptions de niveaux comparables, d'un même système, en terme de l'équivalence de toutes leurs fonctions booléennes (preuve de tautologie).
- Approche basée sur l'abstraction (*Abstract Construction*) : elle repose sur l'idée d'approximation qui consiste à remplacer le raisonnement sur une sémantique concrète exacte par le calcul sur une sémantique abstraite. Elle permet de formaliser l'idée qu'une sémantique est plus ou moins précise selon le niveau d'observation auquel on se place.

Pour mettre en place ces approches, on utilise souvent des langages de spécifications formelles tel que :

1. *Les langages fonctionnels* : sont basés sur l'utilisation des principes de programmation fonctionnelle pour décrire d'une manière formelle les systèmes hardware. Ces langages ont l'avantage sur d'autres de permette des descriptions à différents niveaux d'abstraction.

2. *Le langage OBJ3* : est un langage de spécification algébrique. L'utilisation de ce langage pour la description du hardware a l'avantage de permettre des descriptions hiérarchiques et abstraites.

3. *Le langage LOTOS* : est basé sur l'utilisation des types abstraits de données et les processus pour décrire un comportement d'un contrôleur. Les types abstraits de données sont les plus adaptés pour la modélisation de simples circuits. Pour décrire des systèmes plus complexes, il est préférable d'utiliser des processus. Dans ce cas, la connexion entre composants matériels peut être décrite par l'utilisation des différents opérateurs de synchronisation de ce langage.

4. *Le langage PROMELA* : est un langage de spécification de systèmes asynchrones. Ce qui veut dire que ce langage permet la description de systèmes concurrents, comme les protocoles de communication. Il autorise la création dynamique de processus. La communication entre ces différents processus peut se faire en partageant les variables globales ou alors en utilisant des canaux de communication. On peut ainsi simuler des communications synchrones ou asynchrones.

5. *Les langages de programmation des systèmes réactifs* : cette classe de langages est une nouvelle famille de langages synchrones pour la spécification et la vérification des systèmes réactifs, tel que ESTEREL, LUSTRE, SIGNAL, SML, STATCHARTS...

6. *Le langage VHDL* : est un langage de description du matériel basé sur une sémantique formelle [**?**]. Des programmes écrits en VHDL ont été traduits (compilés) en un modèle formel en se basant sur sa sémantique de simulation pour pouvoir appliquer des méthodes de vérification formelles.

4.3.1.1 Preuves automatiques de théorèmes (Theorem-Proving)

Ces approches permettent de comparer les théories engendrées par deux ensembles de prédicats. Le fonctionnement d'un système séquentiel est décrit par un ensemble de prédicats récursifs, qui est comparé à l'ensemble des prédicats décrivant sa spécification, ou son implémentation.

Cette approche est basée sur les systèmes logiques du premier ordre ou d'ordre supérieur. Elle consiste à la vérification de systèmes présentant une forte régularité, ou pour laquelle on peut décrire le comportement sous forme de prédicats récursifs. De ce fait, elle est adaptée à la description de systèmes à des niveaux très abstraits, ou de systèmes synchrones.

Citons les démonstrateurs de théorème de BoyerMoore [**?**], basés sur la logique du premier ordre sans quantificateur, et qui construit des preuves par induction, et HOL [**?**], basé sur la logique d'ordre supérieur. Les démonstrateurs de théorèmes ont été utilisés pour aider la preuve de

l'équivalence de deux théories ou de l'implication d'une théorie par l'autre dans différents domaines : démonstration de théorèmes mathématiques, validation d'algorithmes, et vérification de systèmes matériels.

Dans le domaine de la vérification des systèmes embarqués, citons les travaux de [**?**], qui a réalisé une des premières preuves significatives en vérifiant le microprocesseur FM8501. Il permet de postuler le problème de vérification à différents niveaux d'abstraction. Cette approche a connu des succès significatifs dans la vérification de conception des processeurs [**?**] et [**?**]. Ces démonstrateurs de théorèmes sont utilisés aussi pour la vérification des systèmes décrits en VHDL.

Cependant, l'inconvénient majeur de cette approche vient du fait que le pouvoir d'expression de la logique du premier ordre et surtout d'ordre supérieur, ne permet pas d'automatiser complètement les preuves [**?**]. De nos jours, la majorité des systèmes de *theorem-proving* sont semi-automatiques et exigent plus d'efforts du côté de l'utilisateur afin de développer des spécifications pour chaque composant et à superviser le processus d'inférence. D'où seuls les experts et les avertis sont capables d'utiliser cette technique sans trop de peine.

Il est impensable, à court terme, de demander aux concepteurs de circuits de guider ces systèmes de preuves. D'autres voies, plus automatisables, ont été développées comme dans le cas de [**?**] et [**?**].

4.3.1.2 Le Model Checking

Les méthodes *model-checking* et *equivanlente checking* font partie de la catégorie de vérification sur des machines à états finis (FMS). Elles requièrent une exploration exhaustive de l'espace des états du modèle. L'exploration de l'ensemble des états du système est une technique simple et automatisable de vérification de propriétés logiques ou de logiques temporelles. Ces techniques, couramment appelées *Model Checking*, ou vérification de modèle, reposent sur :

- la construction d'un graphe d'états représentant les comportements du système.
- l'expression de la propriété à vérifier dans une logique temporelle.

Des algorithmes de parcours et étiquetage du graphe d'états ont été proposés en fonction de la formule à vérifier. Malheureusement, cette exploration provoque, dans le cas des systèmes de grande taille, un problème d'explosion combinatoire d'état. Ceci est donc une limitation de l'usage de ces méthodes. Pour remédier, des tentatives d'amélioration ont été proposées. McMillan a proposé de ne pas représenter explicitement tous les états du modèle [**?**], mais uniquement les états stables.

Les problèmes d'explosion combinatoire de la taille des graphes pour des systèmes même simples limiteraient l'intérêt de ces techniques si elles n'avaient pas été étendues à des représentations symbôliques du graphe des états [**?**] [**?**]. On parle alors de *vérification de modèle symbôlique* ou *Symbolic Model Checking*. Cette technique est basée sur une représentation symbôlique de la relation entre les états du système. L'état d'un système est composé d'un certain nombre de variables d'états binaires, et la relation symbôlique permet de déterminer un ensemble d'états successeurs d'un ensemble d'états donnés.

La manipulation d'ensembles d'états plutôt que d'un état à la fois a été rendue possible par l'introduction, en 1986, des diagrammes de décision binaire [**?**] (BDD : *Binary Decision Diagrams*), qui sont une représentation souvent compacte d'une fonction booléenne. Les ensembles d'états, manipulés par la relation de succession, sont décrits par une fonction caractéristique qui est une fonction booléenne représentée par un BDD.

Les BDD ont permis de repousser les limites des techniques de validation basée sur l'explosion combinatoire des états d'un système. La représentation de fonctions booléennes sous forme de BDD est généralement plus compacte que les représentations sous formes normales conjonctives ou formes normales disjonctives.

Les BDD ont été appliqués à la vérification symbôlique des systèmes, aussi bien pour tester l'équivalence de deux automates que pour valider des propriétés sur ces systèmes.

Le principe de fonctionnement est simple : Le système à vérifier est décrit dans un langage de spécification formelle (ESTEREL, LOTOS,...), ou un langage propriétaire (MURPHY, SMV). A partir de cette description, un système de transitions symbôlique est extrait, sur lequel la vérification symbôlique de modèle est appliquée.

Cependant, et selon [**?**] cette méthode présente toutefois des limitations :

- La relation symbolique entre états n'est pas toujours facile à extraire. C'est notamment le cas des systèmes décrits en VHDL.
- Cette méthode ne peut s'appliquer qu'à des systèmes à nombre d'états fini. De plus, si elle est bien adaptée pour vérifier le contrôle d'applications, elle ne permet pas de traiter l'interaction du contrôle avec de larges chemins de données [**?**]. La prise en compte des chemins de données induirait un nombre d'états trop important pour être manipulé avec des BDD.
- La manipulation des BDD est une tâche délicate, car l'évolution de la taille des BDD ne dépend pas du nombre d'états qu'ils représentent mais plutôt de la régularité de ces états. De ce fait, des systèmes présentant peu d'états peuvent créer de gros BDD difficilement réductibles, alors que d'autres systèmes à grand nombre d'états peuvent être représentés par de petits BDD.
- La fourniture d'une trace d'exécution en cas d'erreur est un problème difficile. Cette trace est pourtant nécessaire pour aider le concepteur à suivre pas à pas le fonctionnement du système lorsqu'une erreur a été détectée.

4.3.2 Apport des méthodes formelles à la conception conjointe

Les méthodes formelles sont basées sur des techniques mathématiques pour la description de propriétés dans un processus de conception conjointe de systèmes (matériels/logiciels). Les méthodes formelles servent pour la spécification, le développement et la vérification des systèmes d'une façon systématique. Il y a plusieurs raisons pour justifier l'adoption des méthodes formelles dans la conception conjointe, en particulier pour la prévention et la détection d'erreurs.

4.3.2.1 Complétude de la spécification

Dans un processus de conception conjointe, l'utilisation des méthodes formelles a pour but d'assurer que la spécification est conforme à tout ce qui est décrit dans le cahier des charges du système à concevoir. Il est pratiquement délicat d'assurer qu'une spécification non-formelle contienne tout ce qui a été prévu au départ.

En plus, il est difficile de veiller à ce que la spécification ne contienne pas de détails d'implémentation. Si elle contient des détails superflous, elle va certainement poser de problèmes d'implémentation pour respecter ces spécificités, ce qui peut entraîner des inconvénients sur le plan performance et structure. Une spécification formelle n'est pas totalement à l'abri face à ces problèmes, mais elle en contient conceptuellement peu.

4.3.2.2 Réduction des taux d'erreurs au niveau de la spécification

Dans ce contexte, l'utilisation des méthodes formelles a pour but d'obtenir une spécification qui contient moins d'erreurs. Contrairement à la raison précédente, il ne s'agit pas ici d'inclure ou d'exclure des comportements dans la spécification initiale mais plutôt d'être sûr que ce qui est spécifié est correct. Car tout type d'erreur dans la spécification est susceptible d'aparaître dans la phase d'implémentation et que les conséquences impliquées en matière de coût sont considérables. Ces conséquences se manifestent sous forme de successions de corrections tardives des erreurs dans le processus de développement, ce qui augmente considérablement les coûts.

Sur le plan commercial, il est aussi important d'investir plus d'efforts dans la prévention d'erreurs afin d'éviter des coûts plus élevés après. Il est surtout nécessaire d'éviter les problèmes de fonctionnalité sur ce premier niveau, puisqu'il est difficile de vérifier des comportements globaux sur un modèle plus détaillé et plus complexe. Des simulations avec des vecteurs de stimuli limités d'une spécification non-formelle ne peuvent pas détecter tous les problèmes. Cependant, avec des méthodes formelles on peut facilement les découvrir.

4.3.2.3 Réduction des taux d'erreurs au niveau de l'implémentation

L'implémentation peut contenir des problèmes, et une bonne partie du développement est consacrée au test et au débogage de cette implémentation. Donc, beaucoup de tests et de simulations sont indispensables, les vecteurs de stimuli choisis ne peuvent couvrir l'ensemble des états possibles atteignables du système. Les méthodes formelles représente un complément important pour éliminer plus de problèmes fonctionnels et logiques dans une implémentation.

4.3.2.4 Réduction du coût de développement

L'un des sept mythes évoqués dans [**?**] dit que les méthodes formelles sont coûteuses à utiliser et que leur utilisation demande beaucoup de temps et d'efforts. Bien que l'utilisation de méthodes formelles prenne beaucoup de temps et coûte de l'argent, son rendement peut être énorme s'il aide à éviter seulement quelques erreurs critiques. En utilisant des méthodes formelles, la phase de spécification du système prend typiquement plus de temps, par contre, avec une spécification plus

claire et plus correcte, l'implémentation, l'intégration et la phase de tests peuvent être effectuées beaucoup plus rapidement.

Il est difficile de mesurer objectivement la productivité de développement et la qualité d'un système pour différentes méthodologies de conception. Cependant, beaucoup d'entreprises utilisant des méthodes formelles parlent de temps de développements et de coûts réduits par rapport à des méthodologies bien spécifiques.

4.3.2.5 Augmentation de la fiabilité du Système

Le concept de fiabilité n'est pas un sujet primordial pour certains produit comme les lecteurs portable multimédia et les système HI-FI de véhicule. L'utilisateur accepte plus au mois quelques inconvénients, mais trop de fautes et des blocages fréquents vont être une expérience risquée pour le client. Pour des téléphones portables, la fiabilité devient déjâ un sujet crucial et il est considéré comme normal qu'un nouveau téléphone nécessite des mises à jour du fabricant dans les premiers mois pour rectifier des erreurs importantes. Les délais de développement très limités par le marché vont généraliser cette tendance dans les années à venir, on va connaître des mises à jour pour le téléviseur, le récepteurs satéllitaire, les appareils photo, le climatiseur et le chauffage domestique jusqu'au four et la machine à laver.

Même si une multitude d'erreurs peuvent être corrigées avec des modifications de firmware et que les mises à jour vont devenir de plus en plus courantes pour les utilisateurs, chacune cause des coûts au fabricant et diminue la confiance du consommateur dans le produit et dans la marque. La vérification formelle pour les systèmes embarqués de base pourrait contrer cette tendance et permettre au fabricants de livrer des produits d'une qualité supérieure.

Il existe une classe de systèmes complètement différente pour laquelle les méthodes formelles se sont déjà intégrées dans les flots de conception standard, ce sont les systèmes critiques. Des systèmes comme le contrôle des réacteurs nucléaires, des parties critiques d'un avion ou la commande de missiles stratégiques nécessitent une fiabilité maximale qui ne peut être atteinte qu'avec l'utilisation de méthodes formelles tout au long du processus de développement. Le succès que les méthodes formelles ont connu dans ces domaines et le manque croissant de fiabilité dans les produits quotidiens mettent en évidence l'inévitable universalisation des méthodes formelles.

4.3.2.6 Respecter les standards

Durant le développement de systèmes et particulièrement à grande échelle, le respect des standards joue un rôle très important. Être en mesure de prouver le respect d'un standard pour des protocoles de bus ou des API standardisés, facilite considérablement l'intégration à grande échelle. Si un standard est bien formulé et si on peut prouver que deux composants respectent celui-ci, on peut supposer que les deux composants vont bien pouvoir communiquer ensemble.

L'utilisation de méthodes formelles peut assurer qu'un composant suit un certain standard d'une façon beaucoup plus fiable que de simples tests ou simulations. Des propriétés formelles

peuvent vérifier les fonctionnalités importantes des interfaces et ainsi valider le respect du standard.

4.4 Vérification des systèmes décrits en SYSTEMC

L'objectif principal de SYSTEMC est de permettre la modélisation et la vérification de très haut niveau d'abstraction, en permettant une exploration rapide de l'espace de conception au niveau système. Actuellement, les travaux de la littérature sont concentrés principalement sur les techniques traditionnelles de validation dynamiques [**?**]. Il est vrai que le développement des techniques de vérification formelle des modèles SYSTEMC est à ses débuts. En raison d'une activité récente intensive dans le développement de techniques de vérification formelle destinées aux logiciels, l'extension et l'adaptation de ces techniques pour SYSTEMC représente un défi promotteur. La difficulté de cette adaptation provient à la fois de la nature orientée objet de SYSTEMC , qui est fondamentale pour sa philosophie de modélisation, et de sa complexité due à sa sémantique strictement orientée simulation dirigée par évènement (*event-driven simulation*).

La solution immédiate pour une validation formelle de descriptions SYSTEMC est la validation dynamique [**?**]. Les modèles SYSTEMC sont par nature destinés à être simulés. Le standard de vérification SYSTEMC fournit des API pour une vérification au niveau transaction, des contraintes et des pondérations aléatoires, une gestion des exceptions, et d'autres tâches de vérification [**?**]. Le standard inclus également les API de connections des éléments HDL, pour permettre le développent des séries de tests (*testbenches*) pour des modèles de haut niveau qui peuvent être réutilisés après raffinement de ces modèles au niveau RTL.

Les techniques de vérification de systèmes sont appliquées aux systèmes décrits en SYSTEMC . De nombreuses sémantiques formelles de SYSTEMC ont été proposées. On retrouve les trois types de vérification : démonstrateurs de théorèmes, construction de l'ensemble des états permettant de vérifier l'équivalence de deux descriptions et l'analyse de propriétés de logique temporelle. Dans tous les cas, un modèle formel de la sémantique de SYSTEMC , ou d'un sous ensemble de cette sémantique, est construit, puis des outils de preuve automatique sont appliqués pour permettre de vérifier des propriétés sur le modèle formel.

4.4.1 Techniques de validation de modèles SYSTEMC

4.4.1.1 Validation à base de Model Checking

Il n'existe pratiquement pas de grande différence entre cette technique et celle de la validation à base d'assertions. Conceptuellement elles sont très proches. Dans une validation dynamique on génère un ensemble de tests d'exécution et on gère également toutes les assertions. Dans le cas de model checking d'état explicite, on applique une sorte de conception exhaustive de modèles formels en gardant trace de tous les choix non déterministes (les valeurs d'entrée). *Java Pathfinder* [**?**] est un exemple de ce type de model checker, qui a été élaboré par l'adaptation de la machine virtuelle Java (*JVM*). Théoriquement, on peut adapter cette méthode pour des descrip-

tions SYSTEMC. Cependant d'implémentation sera sans doute très coûteuse car il faut modifier le noyau de simulation de SYSTEMC afin de permettre l'application de la conception exhaustive.

L'inconvénient majeur de cette approche est l'explosion combinatoire des états [**?**]. Pour faire face au grand espace des états, il est inévitable de passer par la technique d'interprétation abstraite, par contre l'automatisation de telles techniques nécessite généralement l'utilisation de *symbolic model checking* (BDD, ...).

4.4.1.2 Validation à base d'assertions

Dans ce type de validation [**?**], les propriétés doivent être spécifiées dans un langage formel (comme PSL et SVA). Par la suite, le moteur de simulation gère ces propriétés pendant la simulation. Par exemple. INTEL a utilisé son outil *Fedex* [**?**], qui permet la gestion des assertions écrites en langage formel *ForSpec*. Il n'est pas du tout difficile d'incorporer cette technique de vérification à base d'assertions dans le processus de validation dynamique des descriptions SYSTEMC [**?**]. Simplement, il faut intégrer un package BDD à la bibliothèque de SYSTEMC comme dans le cas de [**?**].

Enrichir SYSTEMC par la vérification à base d'assertions veut dire que les mêmes assertions peuvent être utilisées dans un environnement associé à SYSTEMC et également dans un environnement RTL après raffinement dans le cadre d'un processus de synthèse. Une telle réutilisation devrait être un élément important pour l'ensemble de flots de conception et de vérification.

4.4.1.3 Validation par simulation symbolique

Dans cette technique on exécute une description SYSTEMC dans un niveau abstrait, en manipulant des symbôles, plutôt que des valeurs concrètes de variables. Chaque trace de simulation symbôlique représente une entière classe d'exécutions possibles de la description. Ceci est possible par une représentation symbôlique de cette classe d'exécutions, on peut ainsi raisonner sur cette formalisation symbôlique et générer des cas de test [**?**]. Des travaux en simulation symbôlique pour Java [**?**], disent que cette technique peut être appliquée sur des descriptions SYSTEMC. Cependant SYSTEMC est destiné à décrire des systèmes concurrents alors que Java est destiné plus aux systèmes séquentiels. Il s'agit d'une direction intéressante de recherche, un peu moins ambitieuse que les techniques d'analyse statique des chemins d'exécution dynamiques dans le programme, bien que ces techniques aient fait un succès remarquable dans le développement des logiciels et peuvent ainsi être adaptées aux descriptions SYSTEMC [**?**].

4.4.1.4 Validation par symbolic model checking

Cette technique vise la vérification des propriétés temporelles dans un flot de conception, au lieu de chercher l'espace des états explicites du système, ces états seront représentés au moyen du raisonnement symbôlique [**?**]. La difficulté d'étendre les bibliothèques de SYSTEMC par ce type de raisonnement, est que cette technique exige une sémantique formelle pour décrire les relations entre transitions dans le modèle. Ce qui n'est pas du tout facile en SYSTEMC , vue qu'il repose

sur le paradigme orienté objet et sa sémantique est strictement orientée simulation. Cependant des travaux comme [**?**], ont tenté d'étendre des langages orientés objets comme Java par ce type de raisonnement.

4.4.1.5 Validation par preuve d'équivalence

Cette classe de techniques repose sur le développement de méthodes formelles pour prouver l'équivalence de deux modèles : un modèle SYSTEMC et un modèle RTL, une méthode similaire à celle proposée dans [**?**] oú une preuve d'équivalence est établie entre des modèles RTL avec des modèles *Netlist*. L'objectif principal consisterait à établir des équivalences de type : compatibilité et conformité entre des modèles SYSTEMC et des modèles RTL. Par exemple, au niveau des descriptions SYSTEMC il est tout à fait normal de spécifier directement les événements liés à la microarchitecture [**?**]. La compatibilité entre SYSTEMC et les modèles RTL exige que les événements de la microarchitecture doivent être gérés au niveau RTL.

4.4.2 Intégration des méthodes formelles dans les flots de conception

Plusieurs travaux ont été développés pour permettre l'intégration des méthodes formelles dans le flot de conception à base de SYSTEMC [**?**, **?**, **?**]. D'autres tentent de lier les méthodes formelles ou semi-formelles aux flots de conception SYSTEMC. Certains d'entre eux sont SYSTEMPERL [**?**], EDG [**?**], ou C++ comme dans le cas de l'outil BALBOA [**?**].

Cependant, chacune de ces approches a ses propres inconvénients. SYSTEMPERL et GSYSC [**?**] par exemple, exigent que l'utilisateur ajoute certaines indications dans le fichier source, en plus il ne reconnaît pas toutes les constructions de C++. GDE est un analyseur (parser) frontal pour C/C++ qui transforme le code C/C++ en structures de données, qui peuvent ensuite être utilisées pour interpréter les constructions de SYSTEMC. Cependant, l'interprétation des constructions SYSTEMC est une tâche complexe et longue, en plus GDE est un outil commercialisé qu'on ne peut pas l'utiliser librement. L'implémentation de BALBOA a son propre mécanisme de réflexion en C++ qui ne gère qu'un sous-ensemble du langage SYSTEMC. D'autres approches telles que [**?**] nécessitent certaines modifications de la bibliothèques SYSTEMC.

4.4.2.1 Intégration par partitionnement automatique Hardware/Software

Cette approche vise une gestion séparée de parties matérielle et logicielle des descriptions SYSTEMC. Elle consiste à établir un partitionnement automatique et uniforme de descriptions SYSTEMC en parties synchrones (matérielles) et parties asynchrones (logicielles) [**?**]. Ceci est possible grâce à une formalisation de la sémantique de SYSTEMC, en utilisant des structures appelées KRIPKE (LKS : Labled Kripke Structure) [**?**]. Cette notation de LKS est basée sur l'étiquetage des états et des transitions du modèle. Les étiquettes des états modélisent les données du programme, et celles des transitions sont utilisées comme modèle de synchronisation entre les processus (threads).

Les descriptions SYSTEMC sont modélisées par un ensemble de LKSs, elles sont automatiquement partitionnées en parties matérielles et parties logicielles par identification syntaxique

combinatoire et par des processus synchronisés (clocked threads). Les processus combinatoires seront par la suite éliminés, et les transitions des processus synchronisés seront compressés, ce qui va simplifier considérablement le processus de vérification. Cette technique a été implémentée et appliquée au niveau de description système de plusieurs exemples industriels [**?**].

4.4.2.2 Intégration par transformation automatique en langage formel

- *Transformation des descriptions SystemC en langage* PROMELA : Cette approche est basée sur la transformation automatique des descriptions SYSTEMC en langage PROMELA, qui est un langage de spécification formelle de systèmes parallèles asynchrone, il permet la description des systèmes concurrents, en particulier les protocoles de communication des réseaux. Son avantage est qu'il dispose d'un outil de vérification formelle appelé SPIN (ou XSPIN) basé sur l'approche model checker.

  [**?**] propose une méthode de génération automatiques de modèles formels PROMELA à partir de descriptions SYSTEMC asynchrones. Des règles sémantiques de traduction sont définies pour un sous-ensemble de SYSTEMC dédié à la concurrence des processus et les constructions de communication entre modules, tels que : les processus, les primitives de synchronisation, et les variables partageables. Dans ce même travail, une expérimentation a été menée à travers le développement d'un outil de traduction de ce sous-ensemble de SYSTEMC vers le langage PROMELA et des tests ont été réalisés sur des exemples de modèles de communication entre une mémoire, un CPU et un circuit DMA, qui sont tous décrits en SYSTEMC. Après traduction de ces modèles en langage PROMELA, ils sont soumis à l'outil XSPIN pour une vérification formelle d'un certain nombre de propriétés.

  Bien que cette technique est très prometteuse en matière d'application de méthodes formelles sur des modèles SYSTEMC, elle ne couvre pas tous les cas des descriptions synchrones et elle est limitée uniquement à un sous ensemble des constructions de SYSTEMC

- *Transformation des descriptions* SYSTEMC *en langage AsmL :* Asml [**?**] est une variante de langages développés pour la description en formalisme ASM (Abstract State Machines). Il est écrit en C++, il s'agit d'un des langages de la technologie .NET de Microsoft. Il supporte la modélisation orientée objet en haut niveau d'abstraction par rapport à C++ et Java.

  [**?**] propose une formalisation de la sémantique d'un sous ensemble de SYSTEMC en sémantique du langage AsmL, basée sur la sémantique des points fixes de SYSTEMC dans le cas de la modélisation orientée objet, selon une fonction d'observation α, qui constitue l'état (la trace) du système modélisé juste après la phase *Update* du cycle de simulation SYSTEMC. Ensuite, Il définit pour chaque description SYSTEMC, sa projection en langage AsmL en préservant toutes les propriétés sémantiques.

  L'objectif de cette approche est la définition d'une relation entre les processus SYSTEMC actifs durant un *delta-cycle* et l'ensemble des méthodes permettant son exécution en modèle AsmL. Les processus représentés en SYSTEMC par les couples (méthode, liste de sensibilité), sont représentés par les couples (méthode core, pré-condition) en modèles AsmL. L'avantage de

cette technique est surtout la réduction considérable de la complexité des modèles SYSTEMC après la transformation, ceci est dû à l'utilisation de l'interprétation abstraite. Ce qui permet par la suite d'appliquer la preuve formelle, soit par la technique de model checking ou par les approches de preuve de théorèmes sur les modèles AsmL et ASM dans le cas général.

4.5 Conclusion

Au cours de ce chapitre, nous avons distingué deux grandes approches pour la validation des systèmes embarqués décrits en SYSTEMC : la vérification par simulation et la vérification formelle. Nous avons tout d'abord montré comment la simulation est incapable de vérifier les systèmes complexes, vu son incapacité de couverture exhaustive de tous les cas possibles.

Nous avons ensuite introduit les approches de vérification formelle. Nous avons présenté l'ensemble des méthodes formelles proposées dans la littérature, à savoir : les méthodes booléennes, les méthodes déductives et les méthodes basées sur l'interprétation abstraite. Ces méthodes s'appuient sur le raisonnement mathématique pour prouver des propriétés de systèmes.

L'approche de vérification formelle que nous proposons dans ce travail est comparable à celles proposées dans [**?**, **?**]. La particularité de notre contribution est la prise en compte de l'aspect synchrone des descriptions SYSTEMC, et aussi l'utilisation de deux approches de vérification formelles qui sont le model checking via SIGALI et la preuve de théorèmes via Coq.

Deuxième partie

Transformation de descriptions SystemC en modèles formels

Chapitre 5

Etat de l'art sur les langages de spécifications formelles

5.1 Introduction

Dans ce chapitre nous présentons quelques travaux dans le domaine de développement des langages de spécifications formelles. Nous essayons de faire d'abord une classification de ces langages du point de vue de leurs capacités de développer des preuves des propriétés de ces spécifications et d'interpréter ces spécifications. Nous classifions ces langages de spécifications en trois familles :

- La famille des langages de spécification exécutables qui n'ont pas des connexions à des outils de preuves, afin de produire des interprètes tels que ASM [**?**, **?**, **?**], AsmGofer [**?**], XASM [**?**], Asml [**?**] et Typol [**?**].
- La famille des langages de spécification implémentés dans des outils de développement de preuve afin de développer des preuves des propriétés de ces spécifications tels que PVS [**?**] et Isabelle/HOL [**?**]. Ces langages ne sont pas exécutables.
- La famille des langages de spécification exécutables, implémentés ou connectés à des outils de preuves tels que ELAN [**?**], Coq, CASL [**?**] et KIV [**?**].

5.2 Les trois familles des langages de spécifications formelles

5.2.1 Langages de spécifications exécutables mais pas prouvables

De nombreux travaux ont porté sur la conception et l'implantation des langages de spécifications exécutables. Ces efforts ont donné naissance d'une part à des langages de spécifications tel que ASM (Abstract State Machine), et d'autre part à des outils tels que AsmGofer, Asml et XASM, qui implémentent ces langages et offrent des environnements graphiques pour exécuter ces spécifications. Cependant, il leur manque une connexion à des outils de preuves afin de vérifier des propriétés de ces spécifications, nous donnerons dans ce qui suit une brève présentation du langage ASM et les outils XASM, Asml et AsmGofer.

5.2.1.1 Abstract State Machine

ASM est une méthodologie pour la spécification de la sémantique opérationnelle dynamique des langages de programmation, développée par Yuri Gurevich à l'Université de Michigan. La spécification en ASM est donnée par : des types de données, des fonctions (statiques et dynamiques) et un ensemble d'états, la sémantique des types de données et des fonctions statiques est spécifiée par des axiomes et la sémantique des fonctions dynamiques est spécifiée par des règles de transition. Une spécification, vue comme un ensemble de règles de transition, combine les concepts déclaratifs de la logique du premier ordre avec les concepts de la programmation impérative. Ces spécifications sont exécutables à l'aide de plusieurs outils qui implémentent le langage ASM tels que XASM, AsmGofer et Asml.

5.2.1.2 XASM

XASM (Extensible ASM) est un outil qui implémente le langage ASM, et à partir d'une spécification formelle XASM génère un programme écrit dans le langage C. L'outil XASM fournit une animation graphique en cours de l'exécution de ces spécifications, mais il n'a pas de connexion à des outils de preuves.

5.2.1.3 AsmGofer

AsmGofer est une extension du système TkGofer, qui utilise Gofer comme un langage de programmation fonctionnelle et les concepts d'ASM pour la spécification formelle de la sémantique opérationnelle dynamique. Une telle spécification (appelée script en AsmGofer) est exécutable sur plusieurs plates-formes. AsmGofer peut générer des vues graphiques pour des spécifications formelles en cours de l'exécution. Comme XASM, AsmGofer n'est pas relié à des outils de preuves.

5.2.1.4 AsmL

Asml (Abstract State Machine Language) est un langage de spécifications exécutables, intégré dans Microsoft Visual Studio 6.0, ce langage est basé sur la méthodologie ASM et sur l'approche orientée objets. Comme XASM et AsmGofer, Asml n'a pas de connexion à des outils de preuves.

5.2.1.5 Typol

Typol est un langage de spécifications sémantiques implémenté dans le système Centaur. La spécification d'un langage de programmation en Typol est donnée par un ensemble de jugements. A chaque jugement est associé un ensemble de règles regroupées dans un set. Une règle Typol est constituée d'un ensemble de séquents et d'un ensemble de prédicats Prolog importés. Les spécifications sémantiques écrites en TYPOL peuvent être traduites en PROLOG pour produire des vérificateurs de type et des évaluateurs. Delphine Terrasse [**?**, **?**] a défini une traduction de Typol dans le système de preuves interactif Coq pour une classe de langages, afin de développer des preuves des propriétés des spécifications Typol.

5.2.2 Langages de spécifications prouvables mais pas exécutables

L'intérêt porté au développement des preuves des propriétés des langages a conduit à l'apparition de plusieurs outils de preuves. La plupart de ces outils fournit des spécifications non exécutables. Nous donnons dans ce qui suit une brève présentation de deux outils de preuves PVS et Isabelle/HOL.

5.2.2.1 PVS

PVS est un système de vérification, développé au sein de l'équipe des méthodes formelles du SRI International, écrit en Common LISP. PVS se compose d'un langage de spécification et d'un outil interactif d'aide à la preuve. Le langage de spécification est basé sur la logique d'ordre supérieur, enrichie par une notion de sous-typage. PVS introduit de multiples extensions permettant les types dépendants, la paramétrisation, la possibilité de construire des types de données abstraits et la modélisation des fonctions partielles. La notion de définition inductive n'est pas disponible dans ce système. Parce que le langage de spécification de PVS est conçu pour servir comme un langage de spécification logique et non un langage de programmation, la spécification en PVS n'est pas toujours exécutable.

5.2.2.2 Isabelle/HOL

Isabelle/HOL est un assistant de preuve, développé à Cambridge et Munich, permet d'écrire des spécifications des systèmes et de construire des preuves sur ces spécifications. Le système Isabelle/HOL est constitué de deux assistants de preuves : Isabelle et HOL. Parce que les fonctions dans Isabelle/HOL sont vues comme des relations, comme PVS la spécification en Isabelle/HOL n'est pas toujours exécutable.

5.2.3 Langages de spécifications exécutables et prouvables

Afin d'unifier ces deux approches complémentaires, plusieurs chercheurs se sont penchés avec beaucoup d'intérêt sur l'étude et la conception des nouveaux langages de spécification exécutable et d'implémenter ou de connecter ces langages à des systèmes de preuves. Ceci a donné naissance

à plusieurs langages et outils tels que KIV et CASL. Nous donnerons dans ce qui suit une brève présentation de ELAN, COQ, CASL et KIV.

5.2.3.1 ELAN

L'INRIA a développé un environnement, nommé ELAN, pour conduire des preuves en logique classique, de spécifier et d'exécuter en logique de réécriture. ELAN, écrit en Java, est modulaire et chaque module définit un ensemble de sortes, une liste d'opérateurs avec leurs types et un ensemble de règles, les règles dans ELAN peuvent être conditionnelles et sont enrichies par une construction d'affectations locales. ELAN contient un interpréteur et un compilateur qui génère un programme C à partir d'un programme ELAN. L'exécution des spécifications (règles) ELAN est non déterministe, ELAN est le premier langage de programmation basé sur la logique de réécriture qui utilise la notion de stratégies : décrivant des calculs et contrôlant l'application des règles de réécriture. ELAN permet d'effectuer des preuves par réécriture. Une interface entre ELAN et le système de preuves Coq est en cours de réalisation.

5.2.3.2 Coq

Coq est un système de vérification à usage général qui intègre un langage de spécification basé sur la logique d'ordre supérieur typée et un assistant de preuve interactif. Coq développé à l'INRIA fournit une interface interactive dans laquelle l'utilisateur développe une théorie et énonce des propriétés exprimées dans le langage de spécification. Le langage de spécification de Coq comporte les opérations standard des langages de programmation fonctionnelle tel que les fonctions récursives.

Le système Coq est basé sur le calcul des constructions inductives. Les types inductifs de Coq sont pour nous essentiels car les termes et les relations sémantiques d'un langage de programmation sont des structures récursives. Coq repose sur une théorie des types constructifs qui permet de modéliser des fonctions complexes sous la forme d'algorithmes exécutables. Le système Coq permet la production de programmes (extraction) dans plusieurs dialectes de ML tels que Caml Light et Gofer.

5.2.3.3 CASL

Le projet CoFI (Common Framework Initiative for Algebraic Specifications) rassemble une grande partie de la communauté européenne travaillant sur les spécifications algébriques. Il en émane la proposition du langage CASL (Common Algebraic Spécification Language) synthétisant les principales caractéristiques des langages algébriques. Un des objectifs du projet CoFI est de réutiliser les outils existants, tel que ELAN, dans la communauté pour exécuter les programmes CASL et vérifier leurs propriétés. L'outil de preuve HOL-CASL est développé spécialement pour CASL afin de prouver des propriétés des spécifications CASL. CASL permet la spécification de sous-typage, des fonctions partielles, des prédicats et des axiomes de la logique de premier ordre. CASL remplace l'utilisation des règles d'inférences dans la spécification de SOS (sémantique opérationnelle structurelle) par l'utilisation de l'implication.

HasCASL (Haskell and Common Algebraic Spécification Language) est un langage de spécifications sémantiques développé par le projet DFG en Allemagne. Le but du projet est la conception et la mise en place d'une extension évoluée de CASL qui établit une connexion avec le langage de programmation fonctionnelle Haskell.

5.2.3.4 KIV

KIV est un outil pour le développement des systèmes formels, écrit en Java et développé à l'Université de Kalsruhe. L'outil KIV utilise une interface graphique pour le développement des preuves, ces preuves sont représentées en KIV par des arbres (Figure **??**) : la racine de l'arbre représente le théorème et les feuilles de l'arbre représentent les axiomes. La spécification peut se faire en Abstract State Machine (ASM) ou en logique d'ordre supérieur.

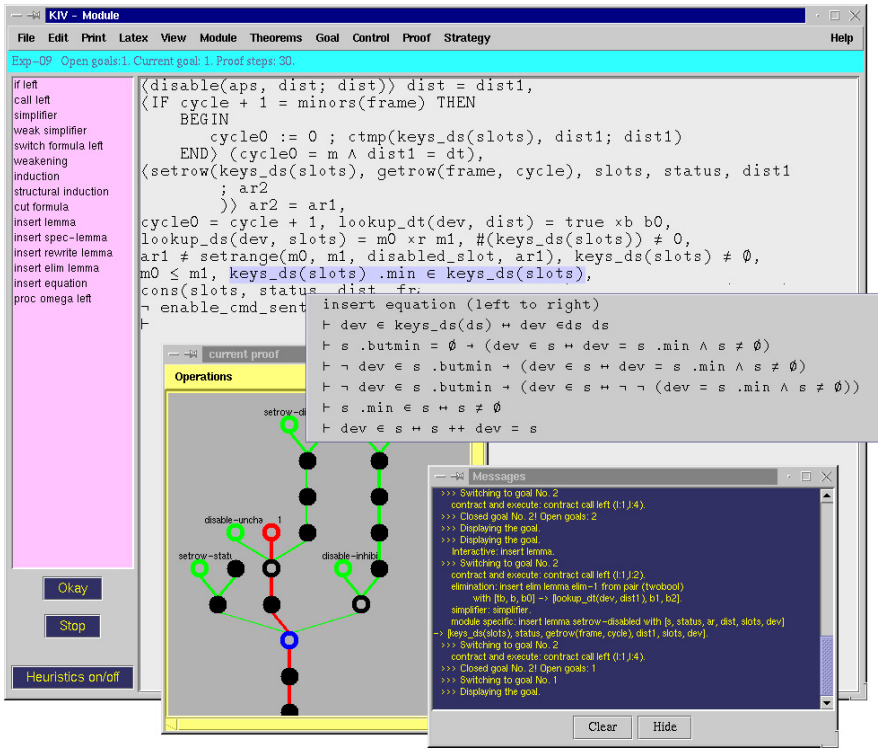

FIGURE 5.1 – Représentation d'un arbre de preuve en KIV

L'outil KIV implémente un langage de spécification algébrique exécutable similaire à CASL. Cet outil fournit une vue graphique (arbre, Figure **??**) des preuves pour aider l'utilisateur à construire ces preuves, mais pas une vue graphique pour la visualisation de l'exécution de spécifications KIV. L'outil KIV implémente aussi un langage de programmation impérative similaire à Pascal qui est exécutable.

5.3 Outils pour la vérification formelle

Nous présentons dans cette sections les deux outils de vérification formelle qui peuvent être utilisés pour la vérification formelle des spécifications SYSTEMC traduites en SIGNAL.

5.3.1 Coq

Coq est un outil de preuve développé par le Projet LogiCal à l'INRIA. La spécification dans le langage Coq est basée sur une logique d'ordre supérieur appelé Calcul des Constructions et les principales fonctionnalités de Coq sont au niveau du formalisme :

- Une structure primitive de définitions mutuellement inductives permettant des spécifications de haut niveau, fonctionnelle et relationnelle.
- Une structure primitive de définitions co-inductives permettant de représenter directement des structures infinies.
- Une interprétation des preuves comme des programmes certifiés mise en oeuvre dans une compilation des preuves sous forme de programmes ML.

5.3.1.1 Spécification syntaxique dans Coq

Le langage de spécification de l'assistant Coq permet de définir des types de données structurés en utilisant les définitions inductives. Les arbres de syntaxe abstraite se codent naturellement dans Coq comme des types de données structurés et les concepts importants de la définition d'une syntaxe abstraite sont les opérateurs et les types.

5.3.1.2 Spécification sémantique dans Coq

La spécification de la sémantique opérationnelle d'un langage de programmation en Coq, peut se faire de deux manières différentes, soit en utilisant l'approche fonctionnelle, soit en utilisant l'approche relationnelle. Dans les deux cas, on peut développer des preuves sur ces spécifications (nécessite une très bonne connaissances de Coq) en utilisant par exemple l'environnement Pcoq.

Approche relationnelle Dans cette approche, la spécification de la sémantique opérationnelle est donnée par des relations (ou des prédicats) inductives de type Prop (type des propositions en Coq).

Approche fonctionnelle Dans cette approche, la spécification de la sémantique opérationnelle est donnée par des fonctions récursives et non récursives.

5.3.1.3 Présentation de l'environnement Pcoq

Pcoq [**?**] fournit un environnement de travail pour le système de preuve Coq, il est construit et maintenu par l'équipe Lemme à l'INRIA. Il réunit les caractéristiques suivantes :

- Interface graphique.
- Séparation entre l'interface et le système de preuves : l'interface graphique et Coq sont deux processus indépendants.
- Des mécanismes d'édition et de présentation structurées.
- Proof by pointing : L'environnement utilise la structure des formules logiques pour aider l'utilisateur à effectuer les étapes du raisonnement en les désignant à la souris.
- écrit en Java et en Ocaml, Pcoq bénéficie de leurs portabilités combinées.
- Langue naturelle : les preuves en cours de développement peuvent être visualisées sous forme d'un texte en français ou en anglais, sur lequel les mécanismes de proof-by-pointing fonctionnent.

La figure **??** représente une vue graphique du développement de la preuve du théorème SRT (Subject Reduction theorem) du langage Exp sous l'environnement Pcoq en utilisant l'approche relationnelle.

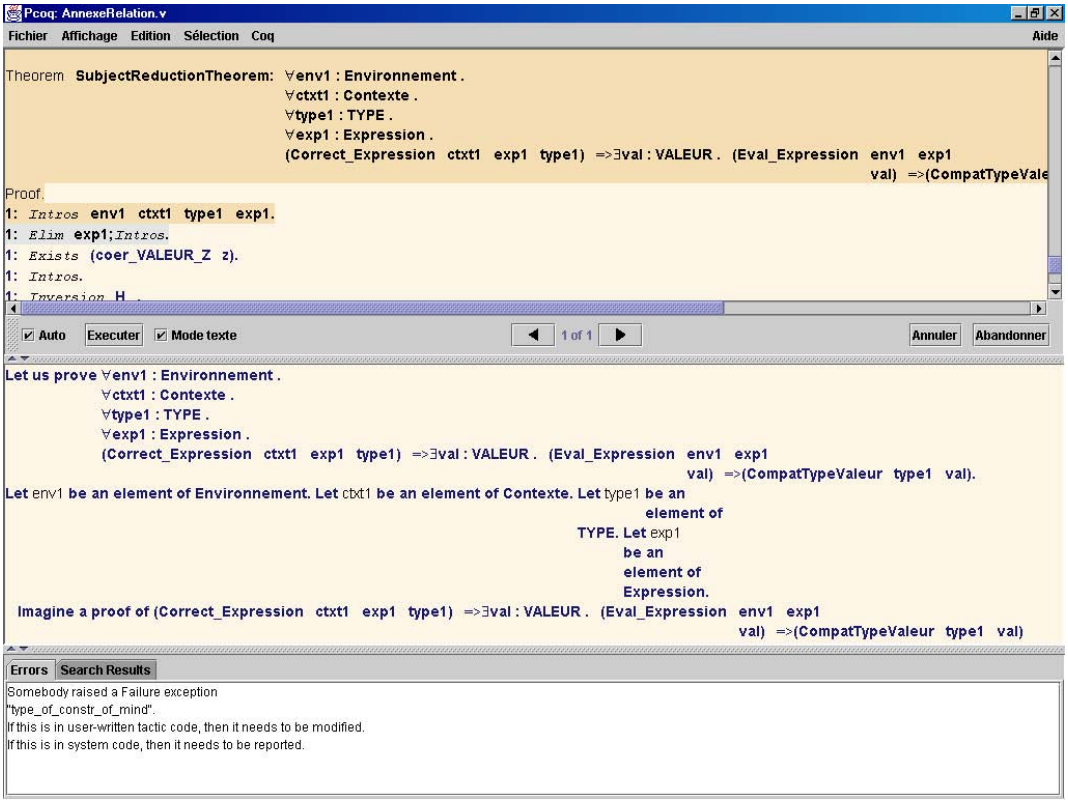

FIGURE 5.2 – l'environnement Pcoq

5.3.1.4 Coq : exécutions et preuves ?

Le système Coq offre un environnement interactif pour le développement de preuves, il permet la production de programmes (extraction) dans plusieurs dialectes de ML tels que Caml Light et Gofer qui est une implémentation de ML. L'extraction [Pot00] dans le système Coq est un procédé automatique qui permet de transformer certains termes du calcul des constructions inductives (CCI) en un programme ML, ce qui rend la spécification en Coq exécutable. Cependant, on ne peut pas visualiser et contrôler l'exécution des spécifications écrites en Coq.

Le langage de spécification de Coq est classé parmi les langages de spécifications exécutables et prouvables :

- Les types inductifs de Coq sont essentiels car les termes et les relations sémantiques d'un langage de programmation sont des structures récursives.
- Coq fournit un environnement confortable pour raisonner sur des objets définis par induction.
- Coq permet l'extraction des programmes exécutables à partir des preuves.

Coq fournit deux approches pour la spécification sémantique, relationnelle et fonctionnelle. L'utilisation de l'approche fonctionnelle pour la spécification des langages de programmations dans Coq offre à l'utilisateur une possibilité d'exécuter et de vérifier des propriétés de ces spécifications. La spécification dans Coq en utilisant l'approche relationnelle n'est pas exécutable. Cette spécification, vue comme un ensemble de prédicats inductifs, facilite le développement des preuves des propriétés de ces spécifications dans Coq grâce à des tactiques de preuves par induction, tel que la tactique Inversion.

5.3.2 SIGALI

SIGALI [**?**] est un système de calcul formel interactif spécialisé dans les calculs algébriques sur l'anneau $F_3[X]$ X^3 - X. Il permet la vérification des propriétés statiques et dynamiques de programmes SIGNALet plus généralement de tout système dynamique polynomial dans F_3. Il permet également la synthèse de contrôleurs de systèmes à événements discrets [**?**] [**?**].

L'outil SIGALI a été developpé par l'équipe *ESPRESSO* de l'IRISA de Rennes [**?**], basé sur le prinicpe de *model-checking* qui se base sur les systèmes polynomials dynamiques (*PDS : Polynomial Dynamic System*), c'est une simple rerpésentation des automates, des modèle intermdiaires pour la reprsentation des systèmes à événement discret. Il offre des fonctionnalités pour la vérification des systèmes réactifs et la synchtèse des contrôleurs discrets.

La technique utilisée dans SIGALI consiste à manipuler un système d'équations modélisant l'ensemble des solutions pour éviter l'énumération exhaustive de tous les états du système. Chaque ensemble d'état est caractérisés par un polynôme unique et les opérations sur les ensmbles peuvent être efectuées sur les polynômes associés. Donc plusieurs propriétés telles que la vivacité, l'invariance, l'accessibilité et l'attractivité, peuvent être vérifiées par SIGALI.

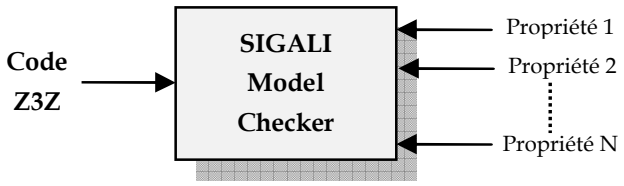

FIGURE 5.3 – Vérification formelle par SIGALI

SIGALI accepte en entrée un fichier z3z, issu de la compilation d'un modèle SIGNAL, afin de vérifier formellement certaines propriétés comme : la vivacité, l'invariance, l'atteignabilité,...

5.4 Conclusion

Les langages de spécifications de la première famille ont des spécifications exécutables mais n'offrent pas une connexion à des outils de preuves afin de développer des preuves des propriétés de ces spécifications. Pour le développement des preuves des propriétés de ces spécifications, il faut concevoir un nouveau système de preuve pour chaque langage de spécifications.

Les langages de spécifications de la deuxième famille sont implémentés dans des systèmes de preuves mais ne sont pas exécutables. Ces langages de spécifications disposent d'un pouvoir d'expression pour aider l'utilisateur à spécifier et à développer ces preuves, mais pas à exécuter ces spécifications. Il existe des outils tels que PVS et Isabelle/HOL qui permettent d'écrire des spécifications exécutables mais pas pour toutes les spécifications.

Les langages de spécifications de la troisième famille sont exécutables et sont implémentés dans des outils de preuves. Cependant, ils leur manque un environnement puissant pour l'exécution de ces spécifications. A l'exception de KIV qui offre un environnement graphique pour le développement des preuves, mais pas pour l'exécution.

Chapitre 6

SIGNAL, un langage formel synchrone

6.1 Introduction

Les applications temps réel se distinguent des autres applications par la présence d'interactions entre le programme et son environnement, qui impose sa propre échelle de temps. Un système temps réel reçoit ses informations à certains instants, et agit lui même sur cet environnement à des instants précis. L'ordre d'apparition des données, leur entrelacement, comptent autant que les valeurs de ces données. N'importe quelle machine avec plusieurs capteurs ou boutons de commande est un système temps réel. Elle se comporte comme un automate, qui réagit, ou change d'état, en fonction des différentes actions qu'elle reçoit. Certaines applications, dites également temps réel, prennent en compte le temps d'exécution de tâches, en imposant par exemple des contraintes temporelles sur leur terminaison. C'est pourquoi ces systèmes sont aussi appelés systèmes réactifs, ou enfouis, ou embarqués. Ces applications se trouvent dans des contextes de plus en plus critiques (avions, centrales nucléaires, armes, ...), nécessitant un maximum degré de fiabilité. Cependant, la vérification des programmes temps réel est une tâche difficile, ainsi que leur débogage (reconstitution de la séquence d'évènements ayant conduit à un crash).

Les systèmes temps réel ont longtemps été programmés soit au moyen de primitives de bas niveau, soit avec des langages de programmation du parallélisme comme CSP et Ada. Cette approche, dite asynchrone, est non-déterministe, et se prête mal au développement de preuves de propriétés. L'approche synchrone vise à écrire des programmes déterministes. Les spécifications et les programmes devront faciliter la vérification automatique de certaines propriétés du système, telle que l'absence de blocage. SIGNAL est un des langages synchrones disponibles, en même temps que LUSTRE et ESTEREL.

6.2 Conception synchrone des systèmes réactifs

6.2.1 Les systèmes réactifs

Le concept de système réactif a été défini, dans la littérature par plusieurs travaux. Selon [**?**], un système réactif est défini de la manière suivante :

Définition 6 (système réactif) *Un système réactif est un système qui réagit continuement avec son environnement à un rythme imposé par cet environnement. Il reçoit, par l'intermédiaire de capteurs des entrées provenant de l'environnement, appelées stimuli, réagit à tout ces stimuli en effectuant un certain nombre d'opérations et produit, grâce à des actionneurs, des sorties utilisées par l'environnement, appelées réactions ou commandes (figure* **??***).*

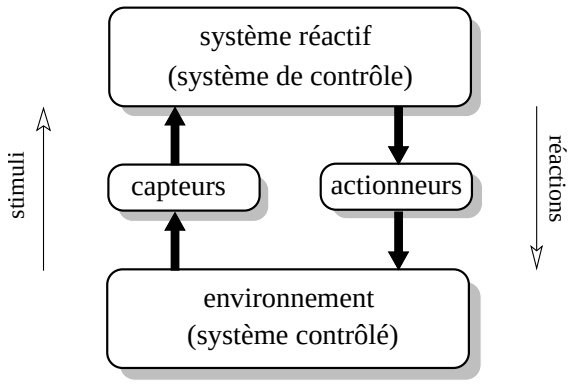

FIGURE 6.1 – Modèle d'un système réactif

Nous appelons systèmes réactifs, les systèmes logiciels qui réagissent de manière continue à leur environnement, et à une vitesse déterminée par cet environnement. Cette classification a été introduite par [**?**] afin de différencier ces systèmes des systèmes transformationnels d'une part (les systèmes classiques dont les entrées sont disponibles en début d'exécution et qui fournissent des sorties quand ils se terminent) et des systèmes interactifs d'autres part (systèmes qui interagissent de manière continue avec leur environnement mais à leur vitesse propre, comme dans le cas des systèmes d'exploitation). La plupart des systèmes industriels *temps-réel* sont des systèmes réactifs. Nous pouvons également citer comme autres exemples les protocoles de communication dans les réseaux ainsi que les interfaces homme-machine.

Les systèmes réactifs sont des systèmes à événements discrets. Leur comportement peut être représenté par une séquence de réactions à des stimuli. Classiquement, les évolutions séquentielles sont décrites en termes d'états et de transitions. Les modèles simples comme les machines séquentielles sont inadaptés lorsqu'il s'agit de représenter des applications modernes. Ces applications sont généralement complexes et conçues comme un ensemble de sous-systèmes communicants et évoluant d'une manière concurrente avec de nombreuses interactions (synchronisations, préemptions). Un sous-système peut lui-même être décomposé. En conséquence, les modèles pour

systèmes réactifs devraient être hiérarchiques, supporter les évolutions parallèles, des formes de synchronisation et de communication diverses.

Dans un système réactif, la validité d'une commande ne dépend pas uniquement de la validité de la valeur de son résultat, mais aussi de son instant de délivrance. Parfois dans la littérature, le système est appelé *système de contôle*, et l'environnement est appelé *système contrôlé*.

La figure **??** est un exemple très simple d'un système réactif de la régulation de niveau d'eau dans un réservoir. Dans cet exemple, l'environnement est constitué d'un réservoir d'eau, d'une vanne et deux capteurs sensibles à la présence d'eaux. Supposons qu'à l'instant t=0 le niveau d'eau dans le réservoir soit au niveau du capteur 1 et que la vanne soit ouverte. Le rôle de ce système est de maintenir le niveau d'eau entre les deux capteurs 1 et 2 : si le capteur 2 est mouillé, le système doit envoyer une commande de fermeture de la vanne avant que le réservoir déborde, et si le capteur 1 est sec le système doit envoyer une commande d'ouverture de la vanne.

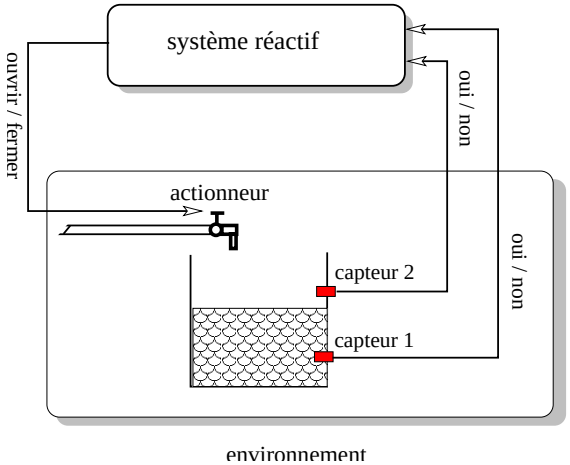

FIGURE 6.2 – Exemple d'un système réactif

Les exigences fonctionnelles et temporelles sont donc deux caractéristiques essentielles des systèmes réactifs qu'ils doivent respecter. Les exigences fonctionnelles imposent au système de produire des résultats corrects du point de vue leurs valeurs. Les exigences temporelles imposent au système de produire ces résultats à temps, c'est-à-dire que le système doit réagir à chaque événement de l'environnement externe avant l'échéance imposée par l'environnement pour le prochain événement.

6.2.2 Caractéristiques des systèmes réactifs

Les systèmes réactifs sont caractérisés par :

- La Concurrence : la concurrence entre le système et son environnement doit être prise en compte. Le plus souvent, il est important de considérer un tel système comme un ensemble de composants (processus ou tâches) parallèles qui coopèrent pour atteindre un comportement

désiré. Ces systèmes sont parfois implémentés sur des architectures distribuées afin d'accroître leurs performances et leurs rentabilité.

- Ils sont soumis à des contraintes de temps strictes. Ces contraintes concernent le temps de réponse du système à ces entrées. Ces contraintes doivent être exprimées dans les spécifications du système (cahier des charges), doivent être prises en compte à la conception et doivent être satisfaites sur l'implémentation. Le respect de ces contraintes de temps nécessitent une implémentation efficace, et plus spécialement une évaluation précise des temps d'exécution.
- Ils sont déterministes : Les sorties des systèmes réactifs sont entièrement déterminées par les valeurs et les occurrences dans le temps des entrées. Cet aspect déterministe distingue les systèmes réactifs des systèmes interactifs : la majorité des systèmes interactifs sont intrinsèquement non-déterministes. Un système d'exploitation, par exemple, dispose d'un ordonnanceur qui active et désactive dynamiquement les processus suivant certains critères (charge du CPU, disponibilité des ressources, priorité des tâches, ...). Le résultat d'un appel système dépend généralement de ces paramètres. La conception, l'analyse et le débogage d'un système déterministe sont bien plus aisés. Par conséquent, le déterminisme des spécifications de systèmes réactifs se doit d'être préservé lors de leur implémentation.
- Leur fiabilité est un point crucial : Il est important que la présence d'erreurs dans un système réactif peut avoir des conséquences graves : ils impliquent des vies humaines et un coût financier très important. Les conséquences économiques et humaine liées à une erreur dans un logiciel de commande d'un satellite ou d'une centrale nucléaire est bien évidemment inestimable. Par conséquent, ces systèmes nécessitent des méthodes de conception particulièrement rigoureuses et constituent un champ ou les techniques formelles de vérification doivent être considérées.

6.2.3 Modèles synchrones pour systèmes réactifs

Un modèle est une formalisation mathématique d'un système réactif (à concevoir ou à étudier), qui peut être structuré selon la complexité de ce système, ou selon le niveau de détail désiré dans la description. Ce modèle peut avoir plusieurs représentations différentes selon les outils de synthèse ou de vérification qu'on veut lui faire. La définition d'un modèle approprié à la modélisation des systèmes réactifs qu'on veut étudier permet d'appliquer le raisonnement formel sur ces systèmes, particulièrement pour faire de la vérification.

Les modèles ne sont toujours pas utilisables pour une description directe des systèmes. Ils ne peuvent que constituer une représentation intermédiaire entre un langage, et les outils de synthèse ou de vérification associés avec ce langage. Un langage de description des systèmes doit disposer d'une sémantique formelle qui décrit la manière dont les programmes peuvent être compilés en modèles plus concrets. Il est important que cette sémantique présente des propriétés à base desquelles le raisonnement formel est possible.

Divers formalismes et langages sont proposés pour décrire les systèmes réactifs, cependant l'approche synchrone s'est présentée comme une alternative incontournable pour la modélisation de cette catégorie de systèmes, car elle est fondée sur les hypothèses les plus simples et repose

sur des langages dédiés au formalisme synchrone. Les langages synchrones les plus connus sont ESTEREL, LUSTER et SIGNAL. Les STATECHARTS sont parfois classés parmi les langages synchrones, bien que leurs sémantiques n'exigent pas forcément une loyauté totale à l'hypothèse de synchronisme. On trouve également de nombreux formalismes et langages qui reposent sur des hypothèses synchrones, et pourrait donc être classés avec les langages synchrones. C'est le cas par exemple du GRAPHSET dont des travaux ont tenté de traduire ce formalisme en langage ESTEREL pour construire des modèles à sémantique formelle et de permettre par conséquent un raisonnement (vérification) formel.

Les langages synchrones proposent des constructions qui permettent de décrire des systèmes réactifs, telles que la composition (comme le parallélisme) de plusieurs sous-systèmes communiquant par diffusion synchrone. L'hypothèse de synchronisme suppose que le temps de réaction d'un système est nul, c'est à dire les sorties sont considérées comme simultanées aux entrées qui les provoquent. Cette hypothèse synchrone sera décrite en détails dans la suite de ce chapitre, qui est entièrement consacré à l'approche synchrone ainsi qu'au langage SIGNAL.

6.3 Le formalisme synchrone

Le principe de base du formalisme synchrone est fondé sur l'abstraction du temps : il s'agit de considérer qu'un automatisme réactif, embarqué dans un environnement donné, interagit d'une façon significative avec cet environnement (au moyen d'un ensemble de médias de communication) et à des instants bien précis (qui forment un ensemble dénombrable et ordonné). A chaque instant, plusieurs actions (messages reçus, calculés et émis) peuvent être effectuées. Ces événements sont par hypothèse simultanés. Ils possèdent donc un même indice temporel.

La perception synchrone de l'écoulement du temps résulte des successions de ces communications, mais aussi de changements explicites décrits dans l'algorithme que le programme met en oeuvre. Selon les formalismes, les sorties calculées sont (sauf changement explicite dans le programme) doivent être simultanées aux entrées qui ont provoqué leur calcul.

Les modèles synchrones adoptent une hypothèse simplificatrice : le système n'évolue que par phases temporellement disjointes appelées instants, dont la durée est nulle. De plus, les signaux sont instantanément diffusés (*instantaneous broadcast*) à tout le programme. Ainsi, lors d'une réaction (instantanée), tous les signaux résultant d'interactions complexes entre sous-systèmes, sont simultanés. Donc ils sont synchrones. Ces fortes hypothèses permettent de garantir des comportements déterministes tout en ayant du parallélisme et des préemptions.

Selon le formalisme synchrone [**?**], le temps est logique et discret, la relation entre le temps logique et le temps réel n'est pas définie par le langage synchrone. A chaque instant, le système réagit aux entrées et envoie les sorties ; c'est le principe des systèmes réactifs. En général, les sorties dépendent de l'histoire des entrées (les signaux) et peuvent dépendre également d'entrées en même temps : c'est le principe de l'hypothèse synchrone.

Dans l'approche synchrone, le temps est manipulé en accord avec les aspects suivants : ordre partiel des événements, la simultanéité des événements, et les délais entre les événements. Le temps

est modélisé chronologiquement et les durées sont des contraintes à vérifier lors de l'exécution. Il est possible de considérer que les calculs ont des durées négligeables en regard des durées de réaction de l'environnement dans lequel est plongé le système. Cela implique que le temps pris par les calculs soit borné.

En formalisme synchrone, la notion d'instant doit être vue comme *instant logique* : l'histoire d'un système est une séquence d'instants logiques ; à chacun de ces instants, zéro, un ou plusieurs évènements interviennent. Les évènements intervenant au même instant logique sont considérés comme simultanés ; ceux qui interviennent à des instants différents sont ordonnés suivant les instances de leur occurrence. Mise à part ces instants logiques, aucun évènement et aucune réaction du système n'a lieu. Par conséquent, tous les processus du système ont la même connaissance des évènements ayant lieu à un même instant.

En pratique, l'approche synchrone est une hypothèse selon laquelle un programme s'exécute assez rapidement pour percevoir tous les évènements externes. Autrement dit, l'exécution du programme est assez rapide pour se terminer avant le prochain instant logique. Si cette hypothèse est satisfaite et plus précisément si cette hypothèse peut être vérifiée, l'hypothèse synchrone est alors aussi réaliste que celle considérant qu'une machine travaille avec des entiers ou des nombres réels.

6.4 Hypothèse synchrone

En formalisme synchrone, on ne prend pas en compte l'intervalle du temps absolu entre deux valeurs d'un signal, mais seulement l'entrelacement - ou l'ordre relatif - ou la simultanéité des signaux. Une hypothèse importante est que l'environnement doit être capable de préciser si deux valeurs arrivent au même instant, ou à des instants différents. A un instant donné, on peut avoir un signal présent, ou plusieurs, ou aucun. On ne s'intéressera évidemment qu'aux instants oú au moins un signal est présent. Une autre hypothèse fondamentale est que les calculs se passent en un temps nul.

L'exactitude d'un programme (fonctionne comme prévu) et son efficacité (l'exécution aussi rapide que possible) ont une importance majeure en informatique, mais ils sont toujours les plus strictes dans le domaine embarqué, car aucune ligne de débogage est réalisable, et des budgets-temps sont souvent impératifs (exemple : les applications multimédia).

- L'exactitude d'un programme est visée par l'introduction de constructions syntaxiques appropriées et des langages dédiés, ce qui rend les programmes de plus en plus compréhensibles par les humains, ainsi que de permettre une modélisation de haut niveau sans oublier les techniques de vérification associées.
- L'efficacité d'un programme est traditionnellement gérée par logiciel à travers les techniques d'analyse de complexité des algorithmes en haut niveau d'abstraction. Cependant dans les systèmes modernes, ce haut niveau de la complexité reflète plutôt mal le bas niveau de la complexité en nombre de cycles d'horloge. Dans le domaine matériel, les concepteurs considèrent plusieurs niveaux de modélisation, ce qui correspond à plus d'abstraction donc plus de précision.

- Une manière possible de mettre en place des langages de programmation synchrones, est de prendre en considération par analogie la programmation en *cycle-accurate* de simulation dans le cadre général, incluant du logiciel. Cette analogie est supportée pat le fait que les environnements de simulation de plusieurs domaines (les simulations HDL par exemple) utilisent le plus souvent des paradigmes de calcul discret, très proche du calcul synchrone à base de cycle. Dans ce contexte, les cycles représentent des instants logiques et non pas le temps physique. Bien sûr, l'analyse temporelle est toujours possible par la suite, après simplification de ce temps par division en cycles.
- Le but principal des langages synchrones est de permettre une modélisation ainsi qu'une programmation des systèmes où la précision au niveau cycle est nécessaire. L'objectif est de fournir des langages structurés sur un domaine spécifique pour leurs descriptions, et d'étudier les techniques de correspondance pour une conception plus efficace. Ceci est dans le cadre de la compilation, la synthèse, l'optimisation, l'analyse et la vérification. La seule condition assurant la faisabilité de cette conception efficace est l'hypothèse synchrone.

6.4.1 Concepts de base

6.4.1.1 Instants et réactions

Les activités comportementales sont divisées en fonction du temps discret (logiquement ou abstraitement), autrement dit, les calculs sont divisés en fonction d'une succession d'instants d'exécution. A chaque instant, les signaux d'entrée éventuellement produits (étant échantillonnés), les calculs internes, le contrôle et la donnée sont propagés jusqu'à ce que les valeurs de sortie soient calculés et le nouvel état du système soit atteint. Ce cycle d'exécution représente la réaction du système pour les signaux d'entrée.

Dans la suite de ce chapitre, l'utilisation du mot *temps* ne concerne pas l'implication du temps réel physique, et les durées instantanées ne doivent pas être uniformes. Le plus important est que les réactions convergent et les calculs soient entièrement réalisés avant la fin de l'instant d'exécution courante et avant qu'un autre commence. Cela renforce l'abstraction conceptuelle évidente que les calculs sont infiniment rapides (instantanés ou temps-zéro) et ont lieu seulement à des points distincts dans un temps, sans durée.

6.4.1.2 Les signaux

La diffusion des signaux est adopté pour faire propager l'information. A chaque instant d'exécution, un signal peut être *présent* ou *absent*. En cas de présence d'un signal, il comporte aussi une certaine valeur d'un type prescrit (les signaux de type *event* portent uniquement leurs états de présence). Le principe fondamental est qu'un signal doit être conforme (le même statut de présence et les mêmes données) pour toutes les opérations de lecture à un instant donné, particulièrement la cohérence des lectures parallèles, ce qui signifie que les signaux agissent comme des variables partagées contrôlées.

6.4.1.3 La causalité

La tâche cruciale qui décide à chaque fois qu'un signal peut être déclaré *absent* a une grande nécessité dans le principe des systèmes réactifs synchrones, et constitue une partie importante de la théorie de l'hypothèse synchrone. Ceci est particulièrement vrai pour les signaux locaux, qui sont à la fois générés et testés à l'intérieur du système. La règle fondamentale est que le statut de présence et la valeur d'un signal doivent être définis avant qu'ils ne soient lus et testés. Cette contrainte prend plusieurs formes en fonction du langage ou du formalisme considéré. L'hypothèse synchrone veille à ce que tous les ordonnancements possibles des éventuelles opérations, mènent au même résultat, ce qui conduit à la bonne définition des programmes, par opposition à des mauvais comportements qui ne conduisent à aucun ordonnancement causal trouvé.

6.4.1.4 Les Conditions d'activations et les horloges

Chaque signal peut être considéré comme définissant (ou générateur) d'une nouvelle horloge, avec l'instant où elle va se produire, c'est les portes des horloges (*clocks gated*) (domaine de la conception du matériel). Les horloges ainsi que les sous-horloges, soient les conséquences d'une génération internes ou externe, peuvent être utilisées comme des entités de contrôle pour activer (ou non) les blocs (les unités) de composants le système. Ce sont les conditions d'activation.

6.5 Polychrony, un environnement synchrone pour la conception des systèmes

Polychrony est un environnement de développement, composé d'un ensemble d'outils logiciel open source dédiés à la conception des systèmes critiques, à base de SIGNAL, un langage temps réel polychrone de type flot de données. Il fournit un environnement d'échantillonnage de modèles unifiés pour une exploration de conception en utilisant des méthodologies de conception ascendantes formelles, basées sur les transformations de modèles à partir de la spécification jusqu'à l'implémentation et du synchronisme vers l'asynchronisme. Il peut être intégré dans les processus de conceptions des systèmes hétérogènes (synchrones/asynchrone) à travers divers formalisme d'entrée et des langages de sortie, selon son architecture présentée dans la figure **??**.

Polychrony est un environnement de développement intégré comprenant un ensemble d'outils de transformation de programmes parmi lesquels un compilateur, des utilitaires de vérification et des éditeurs graphiques. La principale caractéristique de Polychrony est qu'il peut être utilisé pour décrire les systèmes contenant des composants de différentes ou de même horloges indépendantes. Par conséquent, la description d'un système ne nécessite pas une définition a priori d'une horloge globale en tant que descriptions endochrones (dépend d'une seule horloge) comme dans le cas de LUSTRE. Au lieu de définir une horloge globale, le calcul d'horloge de Polychrony se fait à base d'arbres d'horloges, qui résultent les dépendances de différentes horloges exprimées dans la description du système. Pour générer un code exécutable, cette description doit être simplifiée à un état où toutes les horloges peuvent être intégrées dans un arbre d'horloge unique. Ce qui signifie

également que le système est *endochrone*.

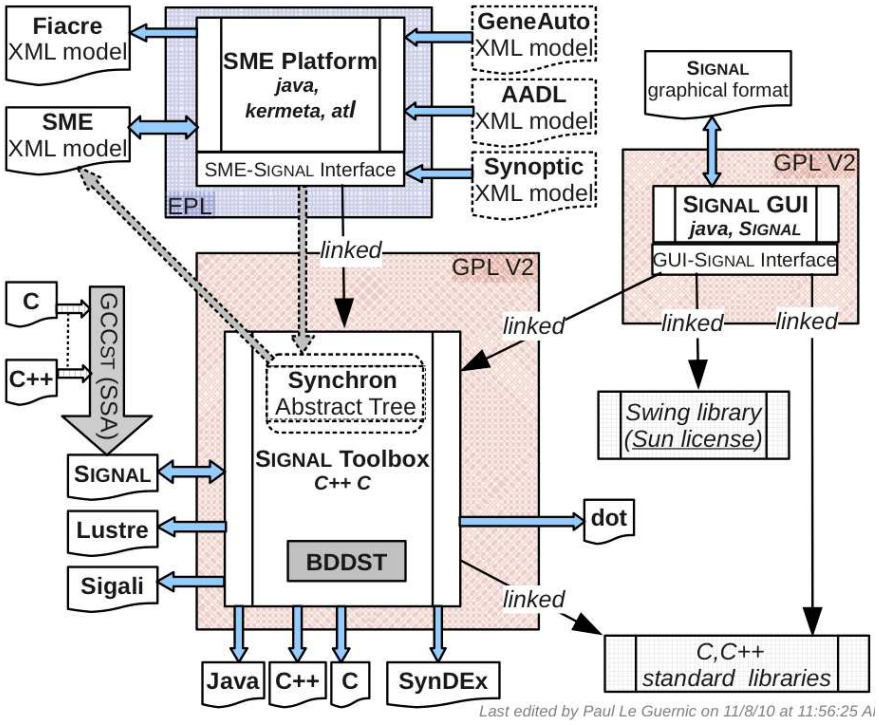

FIGURE 6.3 – Architecture de l'environnement Polychrony [**?**]

Le modèle de calcul polychrone (multi-horloges) offre un haut niveau d'expression. Il supporte la description comportementale non déterministe, qu'on peut trouver dans l'interaction des systèmes embarqués temps réel avec leurs environnements. C'est une propriété importante pour la spécification de haut niveau de plusieurs types de systèmes, car il permet la description de niveaux très abstraits, contenant plusieurs composants fonctionnellement indépendants qui peuvent par la suite être transformés en une description plus intégrée où toutes les communications et les informations de synchronisation sont présentes.

Si un industriel veut générer du code pour une architecture particulière, il va, par exemple, utiliser certains des outils de cette boîte qu'il pourra intégrer dans son propre environnement de travail.

6.6 Le paradigme synchrone

Les langages synchrones ont été conçus pour rendre la tâche du programmeur plus facile, en lui fournissant les primitives idéales qui permettront de considérer le programme comme réagissant instantanément aux évènements extérieurs. Les évènements internes et les évènements de sortie sont datés précisément en respect avec les évènements d'entrée. Le comportement d'un programme est entièrement déterministe, aussi bien du point de vue fonctionnel que du point de vue gestion du

temps. En réalité, la notion de temps physique (chronométrique) est remplacé par la notion d'ordre sur les évènements : les seules notions intéressantes sont la simultanéité et la précédence entre les évènements. Le temps physique ne joue aucun rôle (comme dans le cas pour Ada) ; le temps sera pris en compte comme un évènement extérieur, exactement comme les évènements en provenance de l'environnement externe du programme. il s'agit de *temps multi-forme*.

Les langages synchrones ont été introduits spécialement pour la programmation des systèmes réactifs. Plusieurs langages structurés ont été introduits pour la modélisation et la programmation synchrone réactive des applications. Ils sont grossièrement classés en deux familles : impératif et déclaratif.

6.6.1 Le style impératif : ESTEREL et SYNCCHARTS

Les langages impératifs, comme ESTEREL [**?**] et SYNCCHARTS, fournissent des constructions nécessaire, pour formaliser les systèmes dominés par le contrôle.

Dans ce paradigme, un programme est un automate synchrone hiérarchique, dans le sillage du formalisme de STATECHARTS, avec une grande partie de traitement de simulatanéité, priorité et notification d'absence de signal dans une réaction donnée. Dans la suite, nous décrivons ce style impératif à travers le langage ESTEREL.

Historiquement, le premier langage Synchrone est ESTEREL et a été développé au Centre de Mathématiques Appliquées (CMA) de l'école des Mines de Paris à Sophia-Antipolis. juste après que l'INRIA commence aussi les recherches et le développement pour le compte de ce langage. ESTEREL est un langage impératif qui fut initialement inspiré de CCS et SCCS. ESTEREL introduit des constructions telles que la préemption et la communication par diffusion synchrone de messages. ESTEREL est un langage dédié à la programmation de systèmes à évènements discrets. L'entreprise ESTEREL Technologies propose une version industrielle du compilateur ESTEREL.

En ESTEREL [**?**], un langage synchrone impératif, le concept de signal apporte une représentation abstraite à la fois de la communication et de la synchronisation. Les réactions du système réactif, en réponse aux stimuli, sont caractérisées par des émissions et réceptions de signaux. Les signaux sont également les causes de préemptions. Les préemptions se manifestent sous deux formes principales : la suspension ou l'abandon de l'activité de sous-systèmes. Cette deuxième forme est dénommée *abortion* en anglais. Les modèles synchrones adoptent une hypothèse simplificatrice : le système n'évolue que par phases temporellement disjointes appelées instants, dont la durée est nulle. De plus, les signaux sont instantanément diffusés (instantaneous broadcast) à tout le programme. Ainsi, lors d'une réaction (instantanée), tous les signaux résultant d'interactions complexes entre sous-systèmes, sont simultanés. On dit aussi qu'ils sont synchrones. Ces fortes hypothèses permettent de garantir des comportements déterministes tout en ayant du parallélisme et des préemptions.

La version complète du langage ESTEREL contient un grand nombre de constructions qui facilitent la modélisation, mais il existe un noyau réduit contenant des primitives de déclaration à base desquelles toutes les autres constructions peuvent être dérivées. Ce qui est très important pour

les approches fondées sur des modèles, car seules les primitives nécessaires qui doivent affecter la sémantique comme les transformations dans l'espace modèle. Les sémantiques des primitives sont ensuite combinées pour obtenir la sémantique des constructions composées.

ESTEREL introduit la construction *pause*, utilisée pour diviser les comportements en instants successifs (réactions). Le contrôle est géré à travers des instructions séquentielles, parallèles et de type IF-THEN-ELSE, effectuant des opérations sur des données ainsi que les opérations de synchronisation interprocessus. Mais ces dernières opérations s'arrêtent une fois l'instruction *pause* rencontrée, en mémorisant le contexte dans ce point de suspension pour permettre une reprise. Ce qui fournit un mécanisme d'atomicité, à partir de l'instant où toutes les activités parallèles rencontrent l'instruction *pause*.

6.7 Langage SIGNAL

SIGNAL [**?**] est un langage de programmation et de spécification des systèmes réactifs et temps réel, faisant partie de la famille des langages synchrones, on fait ainsi l'hypothèse que toutes les actions d'un programme sont instantanées [**?**]. Ce langage fait partie de l'environnement Polychrony [**?**] qui dispose d'outils comme le vérificateur de modèle, SIGALI [**?**, **?**]. SIGNAL est un langage synchrone orienté flot de données de style déclaratif qui permet de représenter des modèles polychrones, c'est-à-dire des modèles pouvant contenir plusieurs horloges. Il est construit autour d'un noyau d'opérateurs de base. Ces opérateurs manipulent des signaux, qui sont des suites non bornées de valeurs typées, dont une horloge associée détermine les instants auxquels la valeur est présente.

Par exemple un signal *X* dénote la séquence (x_t) où $t \in T$ de données indexées par le temps *t* dans un domaine *T*. Des signaux d'un type particulier appelés *event* sont caractérisés seulement par leur horloge, c'est-à-dire leur présence (ils ont la valeur booléenne *true* à chaque occurrence). L'horloge du signal *X* est donnée par l'expression *event X*, qui donne l'évènement présent simultanément à *X*.

Les constructeurs du langage permettent de spécifier dans un style équationnel des relations entre les signaux, c'est-à-dire entre leurs valeurs et entre leurs horloges. Des systèmes d'équations sont construits en utilisant la composition. Le compilateur procède à une analyse de la connaissance du système d'équations, et détermine si les contraintes de synchronisation sont bien vérifiées. Si c'est le cas, et si le programme est contraint de façon à calculer une solution unique, alors un code exécutable est produit (en C ou en FORTRAN).

Le langage est construit autour d'un noyau et comporte des opérateurs dérivés pour les tableaux ou des variables par exemple. On peut définir des sortes de macro-instructions : des schémas de processus, qui ont un nom, des paramètres et des signaux d'entrées et de sortie typés, un corps et des déclarations locales. Les instances de schémas de processus rencontrées dans un programme sont expansées par un préprocesseur du compilateur. Le code suivant représente la structure d'un programme SIGNAL :

```
process NAME =
  { [parameters] }
  ( ? [input signals];
    ! [output signals] )
  (|
     [equations]
   |)
    where
      [local declarations];
    end;
```

6.7.0.1 Noyau de SIGNAL

En SIGNAL, les opérateurs de base définissent des processus élémentaires, chacun correspondant à une équation :

- **Les opérateurs fonctionnels** : Ils sont définis sur les types du langage (par exemple la négation booléenne du SIGNAL *E : not E*). Le signal Y_i, défini par la fonction f dans : $Y_i = f(X1_t, X2_t, ..., Xn_t)$ est écrit de la façon : $Y := f(X_1, X_2, ..., X_n)$.
  Les expressions fonctionnelles sont monochromes, ce qui signifie que les signaux $Y, X_1, X_2, ..., X_n$ sont dits synchrones : ils partagent la même horloge. En d'autre terme, pour calculer la valeur de Y_t, tous les X_i doivent être disponibles à l'instant t ; pour cette raison ils sont contraints à avoir la même horloge, celle de Y.
- **Le retard (délai)** : Il donne la valeur passée d'un signal, ce qui est généralement noté $ZX_t = X_{t-d}$ avec la valeur initiale $ZX_t = V_i$, avec $0 \leq i \leq d$; en SIGNAL, pour le cas simple $d=1$, on écrit : $ZX_t = X\$1$ avec l'initialisation *ZX init V0*. Le délai est monochrome lui aussi, c'est-à-dire que X et ZX ont la même horloge.

X:	0	5	⊥	0	4	1	⊥	3	2	...
ZX:	1	0	⊥	5	0	4	⊥	1	3	...

- **Le filtre** : le sous-échantillonnage d'un signal X selon une condition C, est écrit : $Y := X$ *when* C. Cet opérateur est polychrone : les opérateurs et le résultat n'ont pas la même horloge. Le signal Y est présent si et seulement si X et C sont présents au même instant que X et que Y à la fois : l'intersection des horloges de X et Y (c'est-à-dire les instants ou l'expression peut être évaluée) inclut l'horloge de Y (qui ne comporte que les instants ou C s'évalue à true). Quand Y est présent, sa valeur est celle de X.

X:	0	5	⊥	3	4	1	⊥	3	2	…
C:	1	0	⊥	1	0	1	⊥	0	0	…
Y:	0	⊥	⊥	3	⊥	1	⊥	⊥	⊥	…

- **La fusion** : On définit la fusion de deux signaux *X* et *Y* par *Z := X default Y*. Cet opérateur aussi est polychrone : l'horloge de *Z* est l'union de celles de *X* et *Y*, elle est donc plus fréquente que chacune d'elles. La valeur de *Z* est celle de *X* quand il est présent, sinon celle de *Y* quand il est présent.

X:	3	⊥	⊥	9	⊥	2	⊥	⊥	⊥	…
Y:	0	5	⊥	0	4	1	⊥	3	2	…
Z:	3	5	⊥	9	4	2	⊥	3	2	…

- **La composition** : Les processus élémentaires peuvent être composés par l'opérateur commutatif et associatif | qui dénote l'union de systèmes d'équations. En SIGNAL, pour des processus P_1 et P_2 on écrit : (| P_1 | P_2)

6.7.1 Exemple de modélisation en SIGNAL

L'équation x_t = *f*(x_{t-1})+*1*, transformée à : x_t = *f*(zx_{t-1}) +*1* et $zx_{t-1} = x_{t-1}$ est codée en SIGNAL par le programme suivant :

```
process f =
  ( ? ZX integer;
    ! X integer )
(|
   X := f(ZX)+1
 | ZX := X$1
|)
```

6.7.2 Les éléments de base du langage SIGNAL : les signaux

6.7.2.1 Notion de signal

Un signal échantillonné, faisant référence à un temps discret, il provient souvent de l'échantillonnage des signaux analogiques (réels). Il ne s'agit pas seulement d'une suite de valeurs appartenant à un certain domaine, il contient également la suite des instants oú ces valeurs apparaissent. Cette suite est l'horloge du signal.

La suite des instants d'occurrence des valeurs d'un signal fait référence implicitement à une certaine mesure du temps, plus ou moins universelle. Mais toute mesure du temps ne peut se faire qu'en mettant en relation la suite d'événements avec une autre suite choisie comme référence : les vibrations d'un quartz, le passage du soleil au point le plus haut de sa course, etc. Pour comparer les instants d'occurrence de deux signaux, il faut pouvoir décider si deux occurrences sont simultanées, ou si elles ont lieu à des instant différents.

La mise en oeuvre d'un tel mécanisme de décision peut être difficile. Une hypothèse fondamentale pour la sémantique du langage SIGNAL est que, pour tous les signaux considérés, on peut toujours décider si deux valeurs de signaux sont présentes simultanément ou nom [**?**]. D'une façon plus simple, on peut dire qu'un SIGNAL est absent ne présente pas de valeur alors qu'un autre est présent. Par contre, on ne suppose pas l'existence d'un SIGNAL prédéfini ayant une valeur d'horloge universelle.

6.7.2.2 Notions de : ports, traces et flots

Comme dans tout langage de programmation, un signal particulier est désigné par un nom, auquel est associé le type des valeurs que peut prendre ce signal. Ces noms se distinguent cependant des variables classiques par plusieurs points :

- ces variables n'ont de valeurs qu'à des instants bien précis, et ne sont pas accessibles en dehors de ces instants (notion de *valeur absente*).
- Si une variable classique peut changer plusieurs fois de valeur au cours d'un programme (par exemple dans une itération), on ne s'intéresse jamais aux valeurs passées de cette variable. En SIGNAL, par contre, on fera souvent référence à la valeur précédente du SIGNAL S, ou à la N$^{\text{ème}}$ occurrence passée de S, ou même à la suite des M dernières valeurs du signal (fenêtre glissante).
- Un signal évoque davantage une suite de valeurs plutôt qu'une valeur unique, même si à un instant donné, une variable n'a qu'une seule valeur, ou est absente. C'est pourquoi on préfère souvent le terme de port à celui de variable, en parlant de valeur présente portée par un port.

Pour dénoter l'absence de valeur sur un port à un instant donné, on utilise le symbôle *bouttom* $\perp$. Il est systématiquement ajouté au domaine des valeurs des signaux. Soit A = { a_1, a_2, ..., a_n } un ensemble de signaux utilisés dans une application. A un instant donné, chacun de ces signaux porte une valeur, ou il est absent. Un *évènement* dénote l'ensemble de ces valeurs de signaux à cet instant, la valeur $\perp$ est affectée aux signaux absents.

Définition 7 (Une trace) *Une trace sur A est une suite d'évènements portant les valeurs des signaux de A en des instants successifs. Une trace peut être visualisée en représentant sur une verticale les valeurs des signaux à un même instant.*

Soit les quatre signaux présentés dans le tableau **??**, on constate qu'il ne s'est rien passé aux instants 3, 4 et 6 : ce sont des évènements vides. De telles traces n'auraient d'intérêt que s'il existait un autre signal permettant de marquer ces instants oú il ne se passe rien. Sinon, ils peuvent

a_1 :	1	2	$\perp$	$\perp$	7	$\perp$	5	$\perp$
a_2 :	$\perp$	10	$\perp$	$\perp$	123	$\perp$	5	-1
a_3 :	0	6	$\perp$	$\perp$	13	$\perp$	2	$\perp$
a_4 :	$\perp$	11	$\perp$	$\perp$	$\perp$	$\perp$	1	$\perp$

TABLE 6.1 – Exemple de trace

être supprimés sans inconvénient. Le dernier évènement non vide peut cependant être suivi d'un nombre quelconque d'évènements vides.

Définition 8 (un flot) *Un flot est une trace ne contenant aucun évènement vide, ou dont tous les évènements sont vides à partir du premier évènement vide qu'elle contient.*

Le Tableau **??** représente le flot correspondant à la trace du tableau **??**.

a_1 :	1	2	7	5	$\perp$
a_2 :	$\perp$	10	123	5	-1
a_3 :	0	6	13	2	$\perp$
a_4 :	$\perp$	11	$\perp$	1	$\perp$

TABLE 6.2 – Flot spécifique à la trace de l'exemple précédent

Définition 9 (synchronisation) *Deux signaux sont dits synchrones s'ils sont présents (ou absents) exactement aux mêmes instants (a_1 et a_3 de l'exemple précédent).*

Définition 10 (synchronisme) *Le synchronisme est une relation d'équivalence entre signaux, et chaque classe d'équivalence définit une horloge : tous les signaux appartenant à la même classe ont la même horloge.*

Il existe aussi une relation d'ordre sur les horloges : l'horloge de a_4 est moins fréquente (ou inférieure) à celle de a_3, car tous les instants de a_4 sont des instants de a_3, alors que a_3 peut être présent sans que a_4 soit là. Cette relation d'ordre n'est que partielle : ainsi les horloges de a_1 et a_2 ne sont pas comparables.

6.7.2.3 différents types de signaux

1. **Type *event* :** Il ne faut pas le confondre avec un évènement défini comme l'ensemble des valeurs de divers signaux à un instant donné. Il s'agit ici de top, ou d'impulsion. Un objet de ce type n'a qu'une valeur possible, le vrai des booléens. Donc on n'a pas besoin de représenter cette constante, du moins comme valeur de type *event*. Un signal de type event peut être produit par les opérateurs unaires horloge() et *when*.

2. **Type *boolean* :** Une valeur de ce type est obtenue par les notations de constantes true et false, ou par un signal de type event, ou par l'un des six opérateurs de comparaison : = «= »=, ou par les opérateurs NOT, AND, OR. Les priorités sont celles que l'on rencontre dans la plupart des langages de programmation.

3. **Type *integer* :** Opérateurs habituels : + et - unaires, **, *, + et - binaires.
   – utilisation de fonctions : F(paramètre ;...) ; ces fonctions doivent être définies dans la section des déclarations locales.
   – les fonctions prédéfinies (telles que la racine carrée *sqrt*, ou la valeur absolue *abs*) sont celles du langage objet intermédiaire (C) ; seul leur en-tête doit être déclarée localement.
   – le seul opérateur de division est le /. Si I et J sont entiers, I / J donne le quotient entier ; sinon, le résultat est réel.
   – les règles de conversion de type sont celles du langage intermédiaire.

4. **Types *real, dreal,...* :** Notations habituelles. Toute expression à résultat entier est acceptable en opérande lorsque l'autre opérande est réel. Les fonctions prédéfinies suivent les mêmes règles que pour les entiers.

6.7.2.4 Déclarations de signaux

Tous les signaux intervenant dans un processus doivent être déclarés, précédés de leur type. Il peut s'agir de signaux d'entrée, définis dans l'interface derrière le symbôle ?, ou de signaux de sortie, derrière le symbôle !, ou encore de signaux locaux, déclarés en fin de processus après le symbôle where. Les signaux de sortie et les signaux locaux peuvent recevoir une valeur initiale constante lors de leur déclaration. L'exemple suivant illustre divers points de syntaxe :

```
process DECLARATIONS =
  ( ? real a;
      event UN, HH2; integer B
    ! boolean ok init false, TROUVE )
(| XX := sqrt(fabs(-A))
 | ...
 | OK := inter <= 1 when TROUVE
|)
where
  real XX, YY init -1.5; % seul YY est initialise %
  integer inter;
 % en-t^etes de fonctions %
 function sqrt = (? dreal A !dreal B); % en C, paramètre "double" %
 function fabs = (? dreal A !dreal B)
end
```

6.7.3 Les opérateurs de signaux

Dans un processus, des signaux d'entrée sont transformés, au moyen d'un certain nombre d'opérations, pour obtenir des signaux de sortie. Ces transformations nécessitent le plus souvent l'élaboration de signaux intermédiaires, définis localement. La suite de cette section est consacrée

à l'étude des opérations de base permettant de définir un signal. Chacun des opérateurs devra fixer à la fois l'horloge et la valeur du signal produit.

6.7.3.1 Syntaxe relative aux signaux

Un signal est défini par une équation de la forme : <*signal_name*> := <*definition_expression*>
Exemple : *X := Y <= 1 when EXISTE*

L'expression de définition combine divers signaux et constantes au moyen d'opérateurs. Ces opérateurs sont :

- soit des opérateurs liés directement aux types des signaux, et agissant sur leurs valeurs. Tous leurs opérandes doivent être présents aux mêmes instants (opérateurs monochrones). Dans l'exemple, <= compare deux valeurs entières pour donner une valeur booléenne. L'accès à une valeur passée d'un signal (retard) est également un opérateur monochrone.
- soit des opérateurs plus généraux, dits polychrones, faisant intervenir l'horloge des signaux ; le *when* de l'exemple indique que *X* ne prendra de valeur que lorsque le signal *EXISTE* sera présent en même temps que *Y*, et que ce signal *EXISTE* aura la valeur vrai.

Le type de la valeur rendue par l'expression doit être le même que celui du signal défini. La seule exception est que si l'expression est un *event*, le signal défini peut être un booléen (boolean). Pour les autres conversions de type, des fonctions prédéfinies du langage intermédiaire sont utilisées.

6.7.3.2 Horloge d'un signal

L'horloge du signal défini doit être déterminée avant sa valeur, car un signal n'a aucune importance sans information sur sa présence.

6.7.3.3 Structuration de programme SIGNAL

Dans un cadre général, le corps d'un processus SIGNAL est composé d'un ensemble de définitions (équations) de signaux et de contraintes sur l'horloge de chaque signal. Ces définitions sont écrites dans un ordre quelconque, et séparées par des barres verticales.

Le calcul d'horloges est effectué par le compilateur, ce dernier détermine la suite des instants dont au moins un signal peut posséder ou prendre une valeur.

Le calcul de la valeur d'un signal *S* à un instant donné peut faire intervenir :

- des constantes, toujours définies
- des valeurs de signaux d'entrées présents à cet instant
- des valeurs de d'autres signaux, présents à cet instant déja calculés
- des valeurs de *S* à des instants précédents (l'opérateur *retard*)

- des valeurs retardées relatives à d'autres signaux présents à cet instant (mais pas forcément déja calculés)

Il existe des relation de dépendance entre signaux, qui doivent être calculées dans un ordre bien déterminé. Ces relations de dépendance sont évaluées à la compilation. Elle ne doivent pas comporter de cycle, c'est-à-dire qu'un signal ne peut dépendre directement ou non de sa propore valeur à l'instant présent. C'est ce qui empêche d'écrire des définitions de type *S := S + 1*. Par contre, les valeurs d'entrée et les valeurs retardées (de *S* ou d'un autre signal) sont disponibles dès le début du calcul.

6.7.3.4 Les opérateurs de type monochrone

Les signaux intervenant comme opérandes de ces opérateurs doivent être tous *synchrones* (présents ou absents aux même instants). Si leurs horloges sont déjà fixées avant, l'égalité des horloges est vérifiée. Si certaines horloges étaient libres, elles deviennent contraintes à être égales entre elles. Concernant les opérateurs propres aux types des signaux, des opérateurs arithmétiques de comparaison et des fonctions disponibles, on ne peut additionner ou comparer deux valeurs que si elles sont présentes au même instant. Le résultat apparaît également à ce même instant, car on suppose le temps de calcul négligeable.

6.7.3.5 Les opérateurs de type polychrone

Ces opérations combinent des expressions de définition de signaux pouvant avoir des horloges différentes, le résultat ayant également sa propre horloge.

6.7.4 Compilation de SIGNAL

L'intérêt d'un langage évolué pour la programmation des systèmes embarqués temps réel est de permettre la vérification dès la compilation de la complétude et de la cohérence des définitions d'horloges spécifiées dans le programme. En plus des problèmes classiques d'analyse syntaxique, compilation et génération de code, la mise en oeuvre de SIGNAL extrait les horloges des signaux, vérifie de leur cohérence, établit leurs relations (inclusions en particulier) et détermine l'ordre des calculs à partir des relations de dépendance.

La génération du code séquentiel à partir d'un programme SIGNAL, commence par l'analyse de la synchronisation explicite et les relations d'ordonnancement. Cette analyse produit un graphe de flot de contrôle et de données qui définit la classe des spécifications séquentiellement exécutables et permet ainsi de générer le code en sortie.

6.8 Conclusion

SIGNAL est un langage conçu pour la programmation synchrone des systèmes réactifs. Il s'appuie sur des modèles mathématiques solides et dispose d'outils de compilation et de vérification

automatique de code. Les éléments de base de ce langage sont les signaux et les processus, qui définissent une relation entre des signaux d'entrée et de sortie soit fonctionnelles ou temporelles. Il dispose d'un petit nombre d'opérateurs de base (noyau), qui permettent de spécifier dans un style équationnel les relations entre les signaux.

Chapitre 7

Transformation de descriptions SystemC en modèles formels de Signal

7.1 Introduction

Plusieurs solutions sont proposées dans la littérature pour intégrer les méthodes formelles dans un flot de conception conjointe traditionnel. L'utilisation de méthodes formelles a deux principaux objectifs majeurs qui sont la prévention et la détection d'erreurs. La définition d'un modèle capable de représenter de manière précise la sémantique d'expression d'une description SystemC pour être utilisé par la vérification formelle est la tâche principale de ce travail. Ce modèle doit conserver toutes les informations pertinentes et doit aussi garder l'équivalence sémantique avec SystemC. L'idée consiste à définir un ensemble de règles sémantiques avec lesquelles des descriptions SystemC peuvent être traduites automatiquement en descriptions formelles. La solution que nous proposons consiste donc à transformer la spécification initiale SystemC dans un formalisme synchrone qui est le langage Signal. Ce dernier est généralement utilisé pour la spécification et la conception des systèmes embarqués. Il permet ainsi l'application des techniques de vérification formelle sur des descriptions SystemC transformées grâce aux Model Checker de Sigali et l'assistant de preuve de Coq associés à Signal.

Dans [**?**] et [**?**], nous avons présenté une théorie qui repose sur le principe de la construction automatique de modèles formels à partir de modèles non-formels. Le choix de Signal comme destination formelle pour SystemC réside dans le fait que les modélisations (sémantiques) des systèmes embarqués dans les deux langages se ressemblent beaucoup. Le formalisme synchrone est bien adapté à cette tâche entre autres parce qu'il permet d'exprimer d'une façon naturelle les notions de dépendance et de synchronisation entre signaux.

7.1.1 Présentation générale de l'approche proposée

La génération de modèles formels en formalisme synchrone du langage SIGNAL à partir de composants non-formels SYSTEMC est une tâche très difficile, qui nécessite la construction d'un modèle de passage pour vaincre la richesse syntaxique du langage C++ ainsi que le gap sémantique constaté entre les deux langages : sémantique informelle orientée objet et sémantique formelle synchrone.

La transformation automatique de descriptions SYSTEMC en modèles SIGNAL que nous proposons est réalisée en deux phases successives et complémentaires [**?**], comme cela le montre la figure **??**. La première phase, que nous appelons phase d'extraction structurelle, consiste à extraire la structure (squelette) de la description SYSTEMC et ceci dans le but de préserver la forme de la description initial en SIGNAL. La deuxième phase, que nous appelons phase d'extraction comportementale, consiste à extraire le comportement de chaque objet SYSTEMC de la structure.

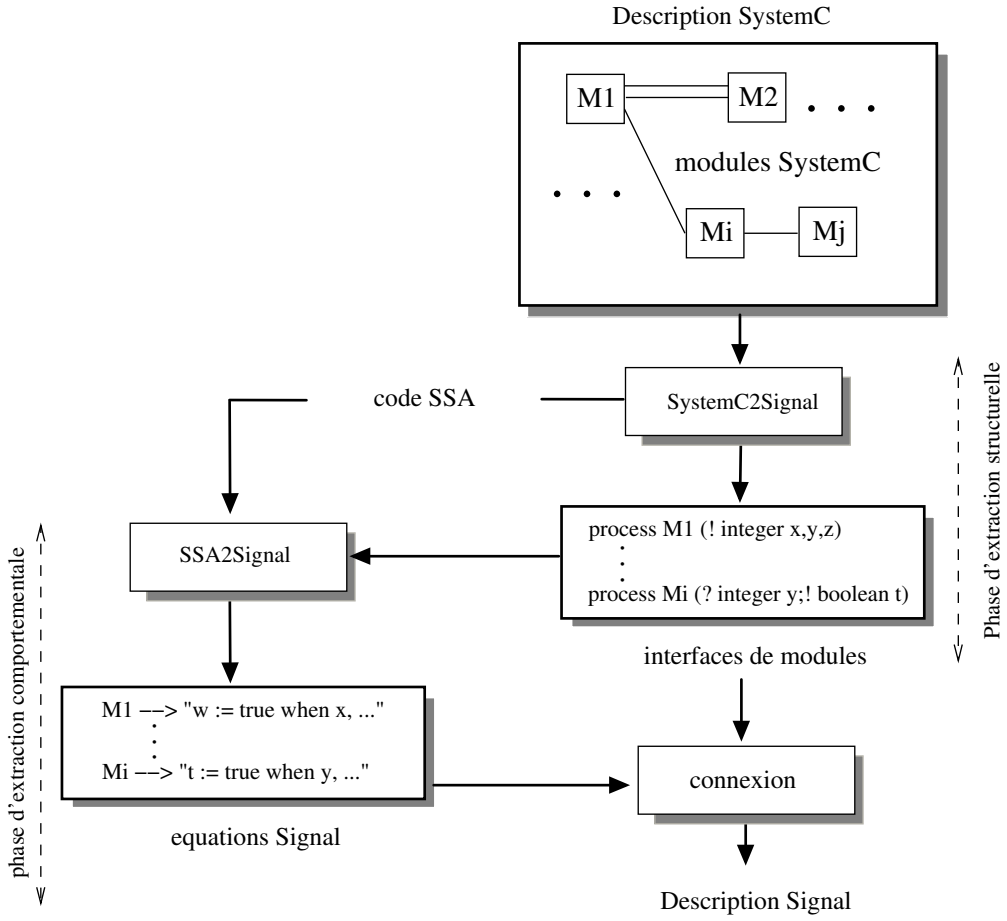

FIGURE 7.1 – Transformation automatique en deux phases

7.1.2 Restriction syntaxique et hypothèses

SYSTEMC est un langage très vaste qui a été initialement prévu pour réaliser des composants synchrones. La version actuelle permet de décrire des composants asychrones, et elle dispose d'une bibliothèque de modélisation ASC (Asynchronous SystemC). SYSTEMC offre une grande variété de styles de description d'un système :

- Une description flot de données : généralement composée d'un ensemble d'instructions concurrentes sur signaux qui décrivent les connexions entre portes logiques et les chargements de registres. Dans ce cas, la notion de processus SYSTEMC composé d'instructions séquentielles n'apparaît pas explicitement dans la description, par contre le regroupement d'instructions concurrentes en modules SYSTEMC est fréquemment utilisé.

- Une description comportementale : généralement structurée sous forme de processus composés d'instructions séquentielles faisant intervenir des variables et des objets de types complexes (tableaux, pointeurs, enregistrements, appels de fonctions, ...).

- Une description structurelle : exprime de la connexion de l'ensemble des IPs SYSTEMC dont le fonctionnement est décrit dans le module associé lors de la spécification pour former le système global. Le style de description du comportement de deux IPs connectés peut être complètement différent.

Si les styles de description peuvent être très différents, ils soumissent forcément aux même règles sémantiques (le même niveau sémantique) qui s'appliquent sur le modèle issu de la phase d'élaboration en vue d'une simulation. Dans ce modèle, toutes les instructions concurrentes sont remplacées par leurs processus équivalents, et toutes les informations structurelles du modèle doivent disparaître. La sémantique de SYSTEMC est définie à partir du modèle élaboré pour la simulation. A ce niveau, il est possible de modéliser un système décrit en SYSTEMC. Ce choix nous permet de nous concentrer uniquement sur quelques éléments significatifs du langage SYSTEMC qui sont les processus et les signaux, sachant que d'autres éléments non considérés peuvent être éliminés par une restriction syntaxique.

Tous les éléments d'un modèle SYSTEMC ne sont pas facilement représentables en formalisme synchrone du langage SIGNAL. C'est notamment le cas des objets de type : boucles, structures conditionnelles, constructeurs de modules, pointeurs et aussi les éléments de l'allocation dynamique reconnus par SYSTEMC. Cependant d'autres trouvent leurs équivalences dans le formalisme synchrone comme : les signaux d'entrée/sortie et beaucoup moins les processus. Nous ne considérons dans ce travail que les descriptions SYSTEMC synchrones.

Etant donnée que la bibliothèque de modélisation SYSTEMC est assez riche et qu'elle ne cesse pas d'évoluer, nous présentons dans ce travail uniquement les règles sémantiques de transformation de base. Ces règles sont facilement extensibles, et nous avons transformé la plupart des classes de la bibliothèque SYSTEMC de la version 2.0 en langage SIGNAL.

7.2 Préliminaires

7.2.1 Graphe de flot de données

Les graphes de flot de données sont l'une des représentations les plus utilisées dans le domaine de la conception et la synthèse des systèmes embarqués. La plupart des algorithmes, des langages et des outils reposent sur cette représentation. En effet, l'évolution des calculateurs de l'architecture conventionnelle de Von Neumann vers les architectures massivement parallèles (architectures multiprocesseurs) a inspiré de nouveaux modèles de calcul conformes à la nature parallèle du matériel (parallélisme de données).

Les sommets du graphe de flot de données DFG (Data Flow Graph) représentent des opérations telles que les opérations arithmétiques et logiques, alors que les arcs représentes les opérandes (valeurs ou données) des ces opérations. Les arcs sont généralement associés aux variables mémorisant les valeurs intermédiaires dans un calcul complexe. Ces arcs représentent les dépendances de données entre les diverses opérations qui déterminent le parallélisme et les opérations.

Formellement, un graphe de flot de données est un graphe orienté G =(V, E), où :

- V = {v_1, v_2,..., v_n} est un ensemble fini de sommets,
- E $\subset$ V $\times$ V est une relation asymétrique de dépendance de données dont les éléments sont des arcs orientés.

Les sommets correspondent aux opérations et les arcs aux dépendances de données. Un arc orienté e_{ij} de $v_i \in$ V à $v_j \in$ V signifie que la donnée produite par l'opération v_i est à consommer par l'opération v_i.

La figure **??** représente un exemple d'un graphe de flot de données. Il représente le calcul de l'expression *R = (A + B) * (C + D)*. Ce graphe est composé de trois sommets et sept arcs nommés selon les noms des variables intermédiaires.

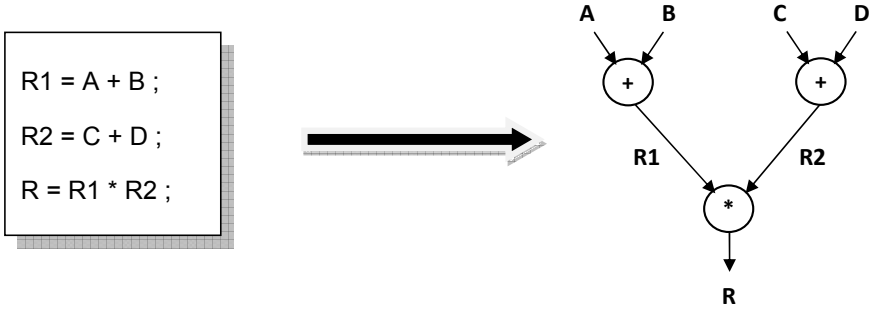

FIGURE 7.2 – Graphe de flot de données

7.2.2 Graphe de flot de contrôle

Le modèle de flot de contrôle tient ses origines de la machine conventionnelle de Von Neumann dans laquelle, les instructions machines sont localisées et sélectionnées par un compteur ordinal, pour être exécutées séquentiellement. Un graphe de flot de contrôle CFG (Control Flow Graph) est une représentation sous forme de graphe de tous les chemins qui peuvent être suivis par un programme durant son exécution.

Un graphe de flot de contrôle est un graphe G = (V, E) orienté, où les sommets correspondant à ce que l'on appelle des blocs de base et les arcs décrivent le flot de contrôle. Un bloc de base est une séquence d'instructions dans lequel il n'existe aucune instruction de contrôle. Dans la suite de cette section, les blocs de base ne sont pas considérés comme des sommet, mais chaque instruction représente un sommet.

Formellement un graphe de flot de contrôle est un graphe orienté G = (V, E), où :

- V = {v_1, v_2,..., v_n} est l'ensemble de sommets.
- E $\subset$ V $\times$ V est une relation de flot de contrôle dont les éléments sont des arcs orientés de séquencement.

La figure **??** représente un exemple de programme sous forme d'un graphe de flot de contrôle. Il est composé de 5 sommets V ={ B1, T1, B2, B3, B4} représentant les instructions du programme et de 5 arcs E = {e_0, e_1, e_2, e_3, e_4}, représentant les relations de précédence entre instructions.

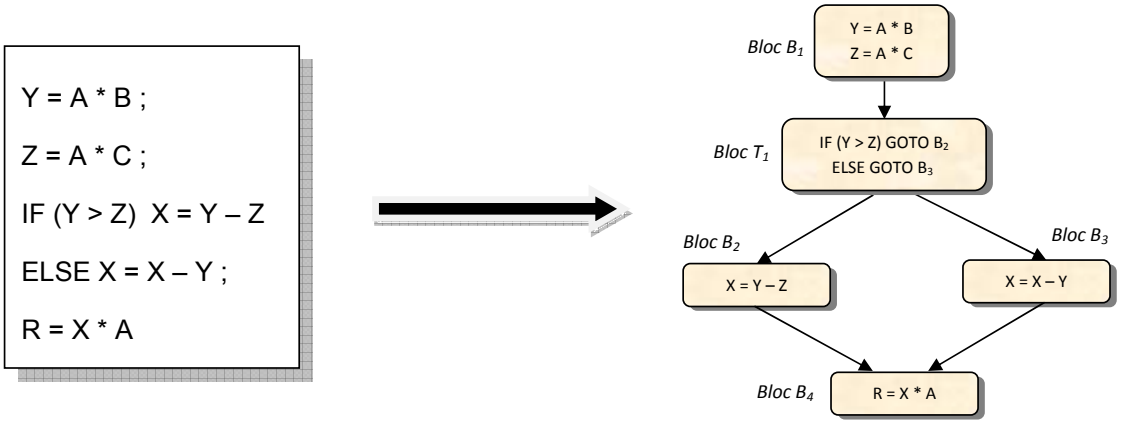

FIGURE 7.3 – Graphe de flot de contrôle

Les sommet d'un graphe de flot de contrôle sont répartis en trois classes (blocs) :

- BASIC BLOCKS : ces sommets d'instruction représentent les instructions simples telles que les affectations des variables et des signaux (cas des langages HDL), les opérateurs arithmétiques et logiques et les appels de fonctions et de procédures. Cette classe de sommets est représentée par le sous-ensemble de B_i de V.

- TEST BLOCKS : ces sommets de branchement modélisent les instructions de conditionnement telles que if, switch, while, ... Ils sont représentés par le sous-ensemble T_i de V.

- JOINT BLOCKS : ces sommets de synchronisation correspondent à une attente d'événements extérieurs tels qu'un changement de valeurs d'un signal. Cette classe de sommets est représentée par le sous-ensemble J_i de V.

Les sommets d'opérations possèdent un seul successeur immédiat alors que les sommets de branchement peuvent avoir plusieurs successeurs. Les sommets de synchronisation ont deux successeurs dont le premier est le même sommet modélisant l'attente active par un cycle (rebouclage).

Un arc d'un graphe de flot contrôle est un arc (v_i, v_j) représentant une relation de séquencement entre deux instructions. Un arc (v_i, v_j) est étiqueté par une expression booléenne $cond_{ij}$ qui signifie que l'instruction représentée par v_j sera exécutée si l'instruction représentée par v_i est exécutée et que l'expression $cond_{ij}$ est vérifiée (vrai).

Si un sommet v_i représente une instruction simple, autrement dit $v_i \in V_0$ et v_j est le successeur de v_i alors $cond_{ij}$ = vrai.

Dans ce modèle de graphe, il est supposé que les expressions des arcs issus d'un sommet représentant une instruction de branchement sont exclusives deux à deux et que pour un exemple quelconque de données, une seule parmi l'ensemble des expressions est *vrai*. Formellement, si v_1, v_2, ..., v_n sont les *n* successeurs immédiats de v_i, alors :

- $cond_{ii_m} \wedge cond_{ii_h}$ = faux, pour tout *m, h* $\in$ { 1,...,k} avec m $\neq$ h.
- $\sum_{n=1}^{k} cond_{ii_h}$ = vrai.

7.2.3 Le formalisme Static Single Assignment

Les compilateurs transforment un programme source en une représentation intermédiaire sur laquelle des transformations de programme seront appliquées. Cette représentation peut être basée sur un arbre syntaxique ou un graphe de flot de contrôle du programme. GCC utilise trois principales représentations intermédiaires pour représenter le programme lors de la compilation : GENERIC, GIMPLE et RTL. la forme GENERIC est utilisée pour servir d'interface entre l'analyseur et l'optimisateur. les représentations GIMPLE et RTL sont utilisées pour optimiser des programmes. La grammaire de GIMPLE est plus restrictive que GENERIC : les expressions ne contiennent pas plus de 3 opérandes. La gimplification (la mise en forme GIMPLE) est la transformation de l'arbre GENERIC en une représentation GIMPLE.

La forme SSA (Static Single Assignement) est un outil d'optimisation, elle est obtenu directement de GIMPLE. Les compilateurs récents réalisent la forme SSA sur les représentations intermédiaires qu'ils utilisent.

Définition 11 (Static Single Assignment) *Un programme est décrit en forme SSA si chacune de ces variables, est l'objet d'une seule assignation (affectation) dans le corps de ce programme [**?**]*

Dans le domaine de la conception des compilateurs, la représentation SSA est une représentation intermédiaire (RI) du code source d'un programme dont la particularité est de ne permettre à une variable d'être affectée qu'une et une seule fois. Les variables existantes dans la première représentation sont divisées en plusieurs versions, les nouvelles variables reprenant le nom original avec des extensions différentes.

La représentation SSA a été développée par Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, et F. Kenneth Zadeck, chercheurs de IBM dans les années 1980.

L'intérêt principal de la forme SSA est la simplification et l'amélioration des résultats de nombreuses optimisations, en simplifiant les propriétés des variables. La figure **??** montre un exemple de passage vers la forme SSA.

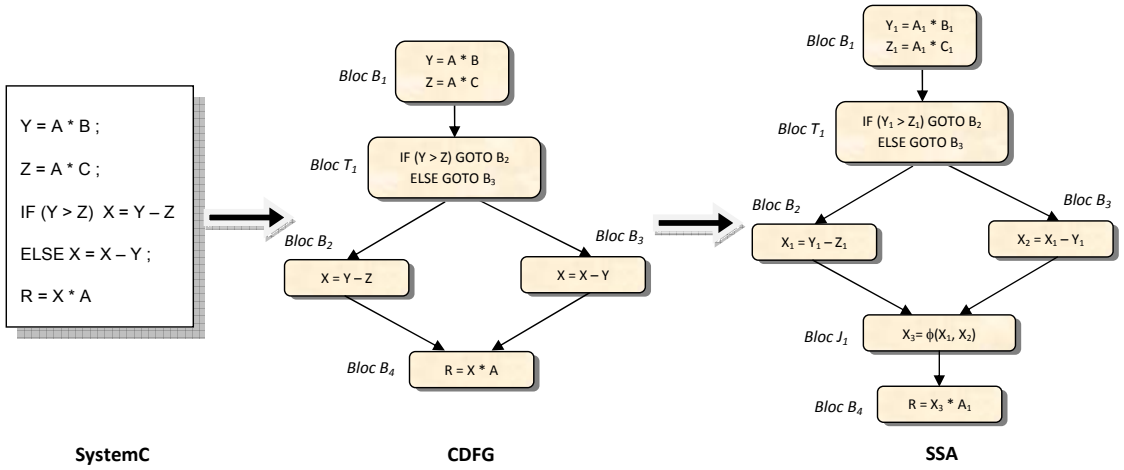

FIGURE 7.4 – Static Single Assignment

7.3 Phase d'extraction structurelle

Afin d'intégrer de tels composants IPs dans la conception d'un système, il est impératif de pouvoir s'assurer que ces composants sont conformes à leur spécification d'usage. Un tel assemblage de composants peut générer plusieurs erreurs, telles que les erreurs d'interopérabilité et d'interaction entre composants. Pour détecter facilement les sources de ces erreurs d'assemblage et de les corriger rapidement, les informations structurelles des modèles SYSTEMC doivent êtres préservées en SIGNAL.

Dans ce but, nous avons généré des interfaces comportementales formelles à partir de blocs IPs non-formelles. L'interface d'un composant IPs ne contient traditionnellement pas beaucoup d'informations. Souvent, seuls les signaux d'entrées et de sorties sont spécifiés ainsi que leurs types respectifs. En plus de cela, la documentation associée aux composants peut contenir des informations sur la communication avec le composant pour en assurer le bon fonctionnement. Pour plus de détails on peut, d'habitude, se renseigner dans le code source des composants. Par contre

pour des IPs achetés, cette option n'est souvent pas disponible puisque les vendeurs préfèrent fournir des boites noires pour ne permettre ni l'accès au code ni son contrôle et pour empêcher leurs clients d'effectuer eux-mêmes des adaptations.

Les interfaces comportementales peuvent contenir bien plus d'informations sur un composant que les noms et les types des entrées/sorties. Elles peuvent contenir des informations comportementales du composant comme la synchronisation de signaux, des relations temporelles entre signaux et tout autre détail de comportement jusqu'à sa représentation complète. De la même façon qu'un compilateur vérifie si les types de données de deux signaux connectés sont compatibles, un outil formel peut signaler des erreurs dans l'interaction de deux composants quand les interfaces formelles sont composées.

La structure d'une description est principalement le partitionnement modulaire, la hiérarchie à l'intérieur du modèle et aussi la liste d'interconnexions entre les modules. Il est important de conserver la structure d'un modèle, car cela permet de localiser facilement les erreurs. La figure **??** présente comment extraire et transformer la structure (squelette) d'une description SYSTEMC en SIGNAL. La transformation est réalisée en trois étapes principales : (i) générer l'entête de chaque processus SYSTEMC, (ii) l'analyse de ces entêtes et identifier l'information structurelle nécessaire, et (iii) générer la structure (squelette) en SIGNAL.

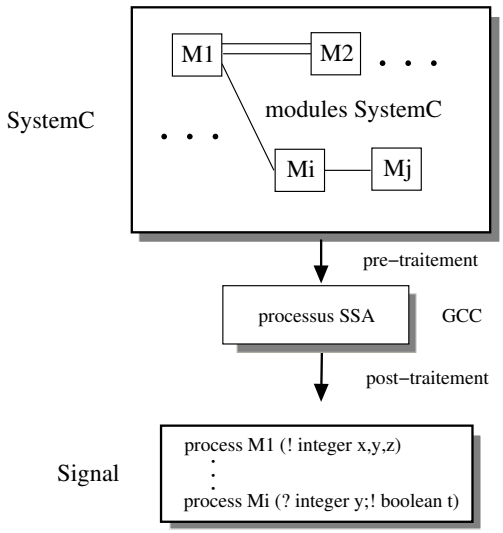

FIGURE 7.5 – Phase d'extraction structurelle

L'extraction de la structure d'une description SYSTEMC est réalisée automatiquement par un outil que nous avons appelé SC2Sig. Après avoir générer les interfaces formelles, le compilateur de SIGNAL permet de vérifier les propriétés d'interaction entre composants, par exemple vérifier si les types de données de deux signaux connectés sont compatibles.

7.4 Phase d'extraction comportementale

Après l'extraction des interfaces des composants, nous présenterons dans cette section comment transformer le comportement de chaque composant en comportement formelle SIGNAL. Pour cela, nous avons utilisé le compilateur GCC de SYSTEMC et son format intermédiaire SSA. L'idée est de transformer les constructions SYSTEMC en code SSA, et ensuite de transformer ce code SSA en code SIGNAL. Le choix est motivé par le fait que la structure de contrôle d'un code SSA est beaucoup plus simple que celle d'un code SYSTEMC.

7.4.1 Transformation des types de données

Le modèle résultant de la compilation ne doit faire référence qu'à des objets proche du formalisme synchrone de SIGNAL, de taille finie et ayant une sémantique précise. SYSTEMC est un langage fortement typé, impératif et orienté objet, donc il est important de conserver toutes les informations concernant les types. Néanmoins seuls les types utilisables en vérification et notamment ceux définis par la norme IEEE 1666TM-2005 [**?**] doivent être considérés.

Les types spécifiés par la norme ont aussi été définis en vue de la simulation et certains d'entre eux comportent des valeurs qui n'ont aucune signification, ni en terme de formalisme synchrone, ni en terme de vérification formelle [**?**]. La figure **??** donne un exemple de transformation de type.

	SystemC	**SIGNAL**
Les structures modulaires	SC_MODULE (<module_name>)	Processsus <module_name >
Les ports de communication	sc_in <type> sc_out <type>	? <type> ! <type>
Les types de données	sc_bit sc_int sc_float sc_bv <size>	boolean integer real [size] boolean

FIGURE 7.6 – Règles de transformation des types standards

7.4.2 Transformation des opérateurs et des fonctions standards

Les opérateurs et les fonctions standards, comme ceux définis par la norme IEEE 1666TM-2005 [**?**], sont bien sûr connus au niveau du langage SIGNAL, et il existe donc une correspondance entre leur utilisation et une famille d'opérateurs synchrones du noyau de base de SIGNAL.

La transformation d'un type complexe comme le type entier en un type élémentaire impose une transposition de toutes les fonctions et de tous les opérateurs associés à ce type. Les opérateurs

complexes tels que les opérateurs arithmétiques peuvent être représentés différemment suivant les propriétés à prouver au niveau formel. Cependant, la tâche de la compilation n'est pas de choisir une représentation précise pour ces opérateurs et ces fonctions, mais beaucoup plus de préserver l'équivalence sémantiques de ces opérateurs.

La compilation doit simplement préserver toutes les informations qui sont susceptibles d'être utilisées par la phase de génération et donc en particulier, elle ne doit pas chercher à expanser les opérateurs et les fonctions standards.

7.4.3 Transformation des blocs SSA

Il existe trois types de blocs en SSA : les blocs de base (BASIC BLOCKS), les blocs de test (TEST BLOCKS) et les blocs de branchement (JOINT BLOCKS). Chacun de ces blocs contient une instruction élémentaire, chaque variable contenue dans les blocs de base et de branchement contient exactement une seule expression d'affectation. Il est donc possible d'exécuter toutes les instructions élémentaires ainsi que les expressions conditionnelles par un signal de condition booléenne (figure **??**) et l'instruction de ce bloc est par la suite sélectionnée pour être exécutée uniquement dans le cas où ce signal est présent et sa valeur est vraie.

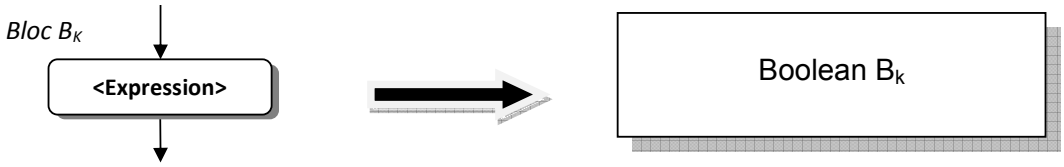

FIGURE 7.7 – Encodage des blocs de base SSA en SIGNAL

7.4.4 Transformation de la fonction PHI

Convertir du code ordinaire dans une représentation SSA est essentiellement un problème simple. Il suffit de remplacer chaque assignation d'une variable par une nouvelle variable "versionnée" (la variable x se décompose en x1, x2, x3 ... lors des assignations successives). Une fonction PHI du bloc de branchement (jointure) J_k dans SSA fonctionne comme un multiplexeur, elle sélectionne la bonne valeur (versions des variables) parmi ses entrées et la retourne sans la modifier (voir figure **??**). La sélection dépend du chemin suivi dans le graphe du programme au moment du calcul de la fonction.

En langage SIGNAL, cette fonction est représentée par une équation d'échantillonnage. La figure **??** montre comment coder l'expression du bloc j_k en SIGNAL.

7.4.5 Transformation des expressions conditionnelles

Pour chaque bloc de test SSA, une instruction de branchement conditionnelle est associée. Par exemple, l'exécution des deux successeurs (les blocs B_j et B_k) du bloc de test T_m dépend de la

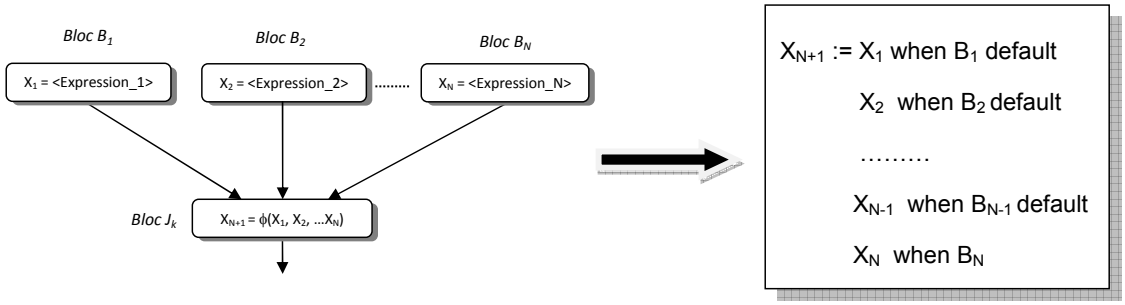

FIGURE 7.8 – Encodage de la fonction ϕ en SIGNAL

valeur de sa condition (voir figure **??**).

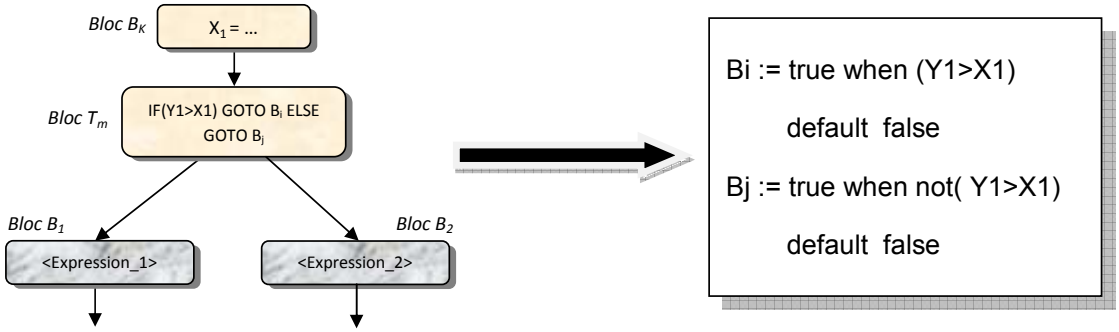

FIGURE 7.9 – Encodage d'expressions conditionnelles en SIGNAL

7.4.6 Transformation des expressions d'affectation

Une expression d'affectation en forme SSA ne contient jamais plus de trois opérandes (à l'exception des appels de fonctions) et elle a des effets secondaires implicites sur le déroulement d'exécution [**?**]. Les équations d'affectation des signaux en langages SIGNAL, ont la même forme que celle des expressions d'affectation de la forme SSA, ce qui rend facile le processus de passage SSA vers SIGNAL (voir figure **??**) sur le plan sémantique [**?**].

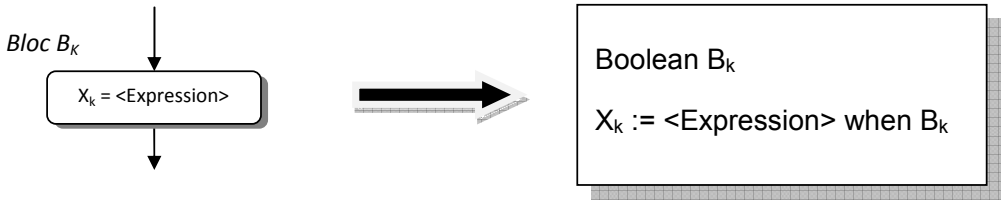

FIGURE 7.10 – Encodage d'expressions d'affectation en SIGNAL

L'instruction est exécutée quand le bloc correspondant est activé, cette condition est réalisée

en SIGNAL par un signal booléen qui correspond au bloc de la forme SSA après traduction, à l'exception des blocs de jointure.

7.4.7 Transformation des boucles

La figure **??** montre un exemple de boucle *for* en SYSTEMC :

```
{...
 for(i=1 ; i<10 ; i++)
 {
   ...
 }
 ...}
```

FIGURE 7.11 – Exemple de boucle *for*

Dans le graphe (voire figure **??**) représentant la forme SSA de la boucle de la figure **??**, le bloc B_j est défini avant l'expression conditionelle *(i_2 < 10)*, et son exécution dépend de la valeur de l'expression conditionnelle.

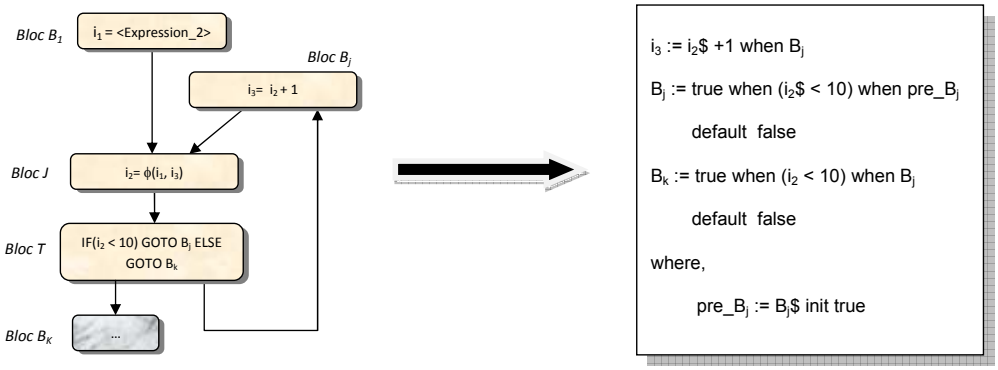

FIGURE 7.12 – Encodage des boucles en SIGNAL

7.4.8 Transformation des modules

Un module est l'élément de base de la conception SYSTEMC, qui contient des ports, des constructeurs, des données membres, des fonctions membres ainsi que des variables et des fonctions locales. Un module SYSTEMC modélise un processus logiciel ou un composant matériel. La figure **??** montre un exemple d'un module SYSTEMC.

La description SYSTEMC du module de la figure **??** est la suivante :

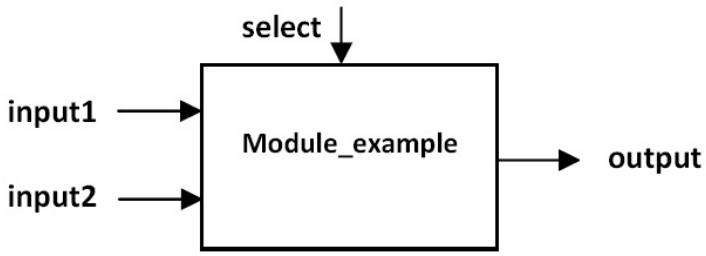

FIGURE 7.13 – Exemple d'un module SYSTEMC

```
#include "systemc.h"

SC_MODULE(module_example)
 {
    sc_in<int> input1;
    sc_in<int> input2;
    sc_in<bool> select;
    sc_out<int> output;

    void module_process()
    {
      output = input1 + input2;
    }

    SC_CTOR(module_example) //constructor
    {
      SC_METHOD(module_process);
      Sensitive << select ;
    };
};
```

La transformation du code SYSTEMC de la figure **??** donne le code SIGNAL suivant :

```
process module_example =
  ( ? integer input1, input2 ;
    ? Boolean select ;
    ! integer output ;
   )
 (
  | output := input1 + input2 when B1
  | B1     := true when select default false
 )
 where Boolean B1;
end
```

L'idée consiste à remplacer le processus constructeur de module, dont la liste de sensibilité contient dans cet exemple le signal *select*, par un nouveau processus SIGNAL qui s'active sur la présence du signal booléen associé au bloc B_k.

Afin de résoudre le problème posé par le compilateur SIGNAL pour le calcul d'horloges, Kalla et Talpin [**?**] ont proposé une solution basée sur la synchronisation globale de toutes les variables et les processus C++. Cependant, l'inconvénient majeur de cette solution est que le comporte-

ment SYSTEMC est totalement différent de celui des processus SIGNAL générés. Ce qui implique un passage d'une sémantique d'exécution multi-horloge (SYSTEMC) vers une sémantique d'exécution séquentielle mono-horloge (SIGNAL). Nous proposons une solution originale basée sur les processus et les modules SYSTEMC, ce qui permet l'identification des relations de synchronisation entre les évènements des modules SYSTEMC et donc de synchroniser uniquement les évènements dépendants de chaque module. Elle détermine un ordre partiel entre les Blocks SSA d'un module sous la forme d'un arbre.

7.5 Codage des pointeurs SYSTEMC en SIGNAL

7.5.1 Présentation du problème

Le problème majeur avec les pointeurs, est que SIGNAL ne supporte pas ces éléments ! En SYSTEMC (C/C++), la sémantique des pointeurs consiste à gérer des adresses d'éléments en mémoire. On utilise deux types d'instructions pour les manipuler : les instructions de lecture (*load*) à partir de pointeurs (... = *P ou ... = P) et les instruction d'écriture (*store*) sur les pointeurs (*P = ... ou P = ...). Il sont aussi utilisés pour : passer des paramètres par référence, accéder aux éléments d'un tableaux ainsi que la mise en place de l'allocation dynamique de l'espace mémoire.

Nous proposons dans cette section une solution originale pour représenter les pointeurs spécifiés dans des descriptions SYSTEMC (C/C++), en langage SIGNAL après une étape d'analyse d'alias (*alias analysis*) comme dans le cas de [**?**]. Notre solution est basée sur l'approche proposée dans [**?**] et [**?**] pour l'encodage des pointeurs en vue d'une synthèse matérielle à partir de descriptions en langage C.

7.5.2 Lecture des pointeurs (load)

Pour cette classe d'expressions, nous avons deux types :

- Les affectations de type : A_i = *f(*P_k*)* avec P_k est un pointeur sur un ensemble de données.
- Les conditions (blocs de branchement/jointure) de : *while, for, if-else et switch*, contenant au moins un pointeur P_k.

L'idée directrice de notre solution, consiste à attribuer pour chaque expression de type A_i = *f(...,*P_k,...)* (avec P_k est un pointeur sur un ensemble fini de variables ou des éléments de tableaux) un certains nombre de nouvelles variables. Le pointeur P_k pointe sur y_k avec $y_k \in \{y_0, y_1, ..., y_n\}$.

La première étape consiste à remplacer chaque pointeur P_k du programme SYSTEMC, par les nouvelles variables suivantes :

- ***Start***_P_k : qui contient la valeur de la variable y_k, pointée par P_k à ce moment.
- P_k***_tag*** : qui contient *k* avec $k \in \{0, 1, ..., n\}$ une valeur associée à chaque éléments de y_k pointé par P_k.

- P_k***_index*** : qui contient l'indice de l'élément du tableau pointé par P_k (uniquement dans le cas des tableaux pointés par P_k*)*.

La deuxième étape consiste à éliminer les symbôles des pointeurs, en se basant sur le principe de renommage des variables SSA, sur la variable **Start**_P_k uniquement pour les expressions conditionnelles.

- Eliminer les symbôles * et & associés aux pointeurs dans la forme SSA.
- Remplacer chaque expression de la forme a_j = *P_k par le code suivant :

```
IF(P_k_tag == 0) Start_P_k = y_0;
ELSE IF(P_k_tag == 1) Start_P_k = y_1;
ELSE …
…
ELSE IF(P_k_tag ==n) Start_P_k = y_n;
```

On propose ici de remplacer l'imbrication des structures *if-else* par des structures *if* indépendantes, et ceci afin de réduire la complexité de génération automatique.

```
IF(P_k_tag == 0) Start_P_k_0 = y_0;
IF(Pk_tag == 1) Start_P_k_1 = y1;
…
IF(Pk_tag == n) Start_P_k_n = y_n;
Start_P_k_m = ϕ( Start_P_k_0, Start_P_k_1,…, Start_P_k_n)
```

Finalement, à ces expressions conditionnelles s'ajoute l'instruction a_j = *start_*P_k*_m* pour remplacer complètement a_j = *P_k.

La dernière étape consiste à traduire le tout en code SIGNAL (voir la figure suivante).

```
| Start_P1 := y0 when Bk
| Start_P2 := y1 when Bk+1
    …
| Start_Pn+1 := yn when Bk+n
| Start_Pm := Start_P1   when Bk   default
               Start_P2   when Bk+1 default
               ...
               Start_Pn   when Bk+1 default
               Start_Pn+1 when Bk+n default   false
// associating code with the assignment of reading the value
pointed by Pk (Start_Pm)
| Bk   := true when(P_tag1 = 0)
| Bk+1 := true when(P_tag2 = 0)
…
| Bk+n := true when(P_tagn = 0)
```

Pour des raisons de simplification de la génération automatique à partir du code SSA, on propose le code SIGNAL suivant :

```
| Start_Pk := y0 when (P_tag = 0) when Bk
    default  y1 when (P_tag = 1) when Bk+1
    …
    default  yn when (P_tag = n) when Bk+n
```

7.5.2.1 Exemples de transformation des expressions de lecture des pointeurs

1. **Premier exemple** : la traduction de l'expression SYSTEMC : *X = *P + 1* et en supposant que *P* pointe sur l'espace des variables *{a, b, c, tab[]}*, et *P_tag = {0, 1, 2, 3 }*. Le code suivant est la transformation de l'instruction précédente en code SSA :

```
Bk:   IF(_tag1 == 0) Start_P1 = a1;
Bk+1: IF(_tag1 == 1) Start_P2 = b1;
Bk+2: IF(_tag1 == 2) Start_P3 = c1;
Bk+3: IF(_tag1 == 3) Start_P4 = tab[P_index];
J:    Start_P5 = ϕ( Start_P1, Start_P2, Start_P3, Start_P4)
Bj:   X1 = Start_P5 + 1;
```

La transformation en code SIGNAL est :

```
| Start_P1 := a when Bk
| Start_P2 := b when Bk+1
| Start_P3 := c when Bk+2
| Start_P4 := tab[P_index] when Bk+3
| Start_P5 := Start_P1 when Bk    default
               Start_P2 when Bk+1  default
               Start_P3 when Bk+2  default
               Start_P4 when Bk+3
| X1 := (Start_P5+1) when Bj
| Bj := true when Bk     default
        true when Bk+1   default
        true when Bk+2   default
        true when Bk+3   default  false
| Bk   := true when(P_tag1=0) default false
| Bk+1 := true when(P_tag1=1) default false
| Bk+2 := true when(P_tag1=2) default false
| Bk+3 := true when(P_tag1=3) default false
```

2. **Deuxième exemple** : la traduction de l'expression SYSTEMC : *IF(X>*P) X=X-1 ELSE X=0* et en supposant que *P* pointe sur l'espace des variables *{a, b, c, tab[]}*, et *P_tag = {0, 1, 2, 3 }*. Le code suivant est la transformation de l'instruction précédente en code SSA :

```
Bk:     IF(_tag1 == 0) Start_P1 = a1;
Bk+1:   IF(_tag1 == 1) Start_P2 = b1;
Bk+2:   IF(_tag1 == 2) Start_P3 = c1;
Bk+3:   IF(_tag1 == 3) Start_P4 = tab[P_index];
J1:     Start_P5 = ϕ( Start_P1, Start_P2, Start_P3, Start_P4)
Bj/ Bi: IF(X1 > Start_P5) X2= X1 – 1;  ELSE X3= 0;
J2:     X5 = ϕ(X1, X2, X3, X4)
```

La transformation en code SIGNAL est :

```
| Start_P1 := a when Bk
| Start_P2 := b when Bk+1
| Start_P3 := c when Bk+2
| Start_P4 := tab[P_index] when Bk+3
| Start_P5 := Start_P1 when Bk    default
               Start_P2 when Bk+1  default
               Start_P3 when Bk+2  default
               Start_P4 when Bk+3
| X2 := X1 -1 when Bi
| X3 := 0 when Bj
| X4 := X2 when Bi  default  X3  when  Bj
| Bi := true when (X1 > Start_P5) when B default false
| Bj := true when not(X1 > Start_P5) when B default false
| B :=true when Bk+1 default
       true when Bk+2 default
       true when Bk+3 default false
| Bk   := true when(P_tag1=0) default false
| Bk+1 := true when(P_tag1=1) default false
| Bk+2 := true when(P_tag1=2) default false
| Bk+3 := true when(P_tag1=3) default false
```

7.5.3 Écriture des pointeurs (store)

Il existe deux cas d'écriture des pointeurs à savoir :

- le cas de modification d'adresse (P_k = $\&x_i$) avec x_i est une variable quelconque.
- le cas de modification de la case mémoire pointée par P_k ($*P_k$ = *expression*).
- **Premier cas (P_k = $\&x_i$)** : Dans ce cas, il suffit de remplacer cette affectation au niveau SSA par :

B_j : $P_k\_tag_1$ = i ; // when i is the only value associated to x_i

B_i : $Start\_p_k\_1$ = x_i

Le code SIGNAL équivatent est :

| $P_k\_tag_1$:= i when B_j

| $Start\_p_k\_1$ = x_i when B_i

- **Deuxième cas ($*P_k$ = *expression*)** : Dans ce cas, il suffit de remplacer cette affectation au niveau SSA par :

B_j : $Start\_p_k\_1$ = <statement>

Le code SIGNAL équivatent est :

| $Start\_p_k\_1$: = <statement> when B_j

7.6 Conclusion

Nous avons présenté dans ce chapitre notre approche de transformation automatique de descriptions SYSTEMC vers SIGNAL. La transformation est réalisée en deux phase complémentaires. La première phases consiste en extraction structurelle, et la deuxième en extraction comportementale. Un ensemble de règles sémantiques de transformation est aussi décrit dans ce chapitre. Cette approche vise l'intégration des méthodes formelles synchrones dans le flot de conception des systèmes embarqués conçus à base de descrptions SYSTEMC.

Chapitre 8

Implémentation

8.1 introduction

L'objectif de ce chapitre est de présenter l'outil de transformation automatique de descriptions SYSTEMC vers modèles SIGNAL. Dans ce contexte, deux possibilités ont été envisagées :

1. La première consiste à développer un nouveau compilateur qui accepte en entrée des programmes SYSTEMC (C++). Cependant, vue la richesse et la complexité sémantique du langage C++, cette solution est très coûteuse en matière de temps et de moyens.

2. La deuxième solution consiste à modifier le code source d'un compilateur existant, en détournant sa phase de génération de code objet vers le langage SIGNAL, afin d'éviter l'écriture d'un nouveau compilateur.

Cette dernière possibilité a été adoptée, et pour cela on a choisi de modifier la version open source du compilateur GCC-4.1.0 de C++. L'idée consiste à mettre en place la génération du code SIGNAL en profitant du format intermédiaire SSA. On peut facilement forcer le compilateur par des options spéciales (comme fdump-tree-ssa) pour générer en sortie le code en formalisme SSA.

8.2 Format intermédiaire SSA

La raison principale pour laquelle on a choisi d'utiliser la forme SSA est que les équations d'affectation des signaux en langages SIGNAL ont la même forme que les expressions d'affectation de la forme SSA, ce qui rend facile le processus de passage SSA-SIGNAL sur le plan sémantique [**?**]. En plus, le format intermédiaire SSA a été adopté comme mécanisme fondamental pour l'optimisation par les compilateurs comme dans le cas de GCC et la machine virtuelle Java RVM [**?**]. Ce qui permet d'utiliser facilement notre approche par les concepteurs afin de vérifier formellement des spécifications des systèmes embarqués écrites en SYSTEMC, en utilisant des langages communs pour décrire ces systèmes.

Dans ce cas, des descriptions SYSTEMC font l'objet d'une traduction vers la forme intermédiaire SSA et par la suite vers le langage SIGNAL en adaptant la partie Front End du compilateur GCC-4.1.0.

8.3 Le compilateur GCC-4.1.0

GCC [**?**], abréviation de GNU Compiler Collection, est le compilateur créé par le projet GNU qui fonctionne sous LINUX. Il s'agit d'une collection de logiciels libres intégrés (code source en C) capables de compiler divers langages de programmation tels que C, C++, Objective-C, Java, Ada et Fortran. Ce compilateur introduit la forme GENERIC à partir des langages d'entrée, cette forme a le rôle d'éliminer le problème de manque d'une représentation unique [**?**], GCC utilise un *gimplifier* pour introduire la forme *Gimple* qui consiste à générer un code à trois adresses et a le rôle d'analyser et d'optimiser le code, ainsi que CFG qui nous donne un graphe de flot de contrôle complet sur notre code puis la forme SSA, et enfin la génération du code RTL (Register Transfer Language). Ces phases sont mentionnées dans le schéma de la figure **??**.

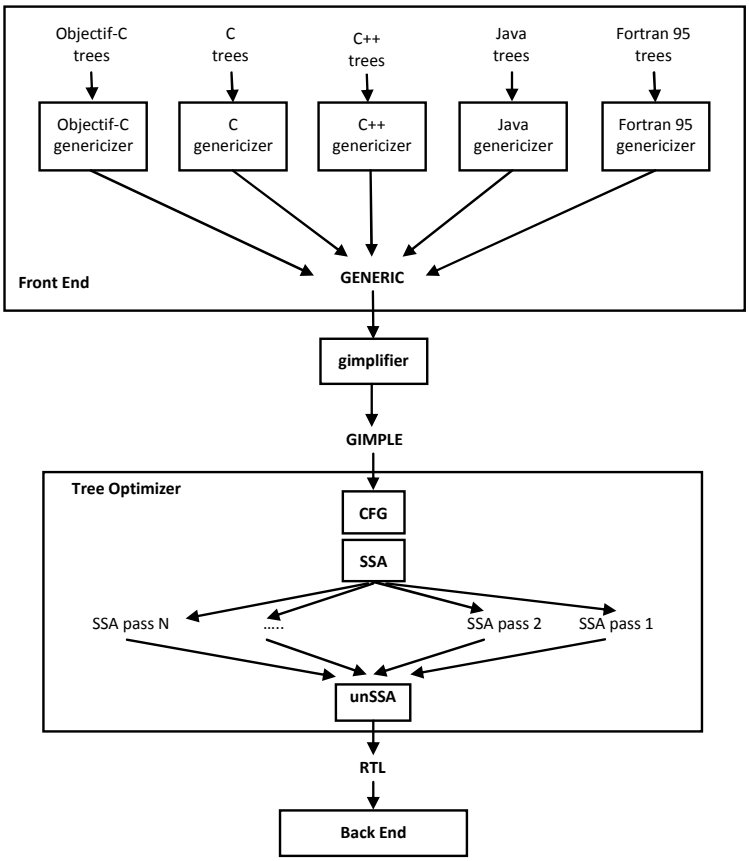

FIGURE 8.1 – Architecture du compilateur GCC-4.1.0

8.4 Adaptation du Compilateur GCC-4.1.0

On a utilisé le compilateur GCC et son format intermédiaire GIMPLE/SSA **??**. La structure de contrôle d'un code GIMPLE/SSA est beaucoup plus simple que celle d'un code C++ et en conséquence une représentation GIMPLE/SSA est plus facile à traduire. Par contre, on perd l'information sur la structure initiale du programme SYSTEMC. Une transformation purement comportementale, sans conservation de structure a cependant plusieurs inconvénients. Sans cette information structurelle, il est très difficile pour le compilateur SIGNAL de faire des optimisations de haut niveau comme par exemple pour l'ordonnancement. De plus, sans structuration, le code SIGNAL résultant devient illisible et complique donc toute modification manuelle du modèle SYSTEMC. Finalement, une fois une erreur se manifeste dans le modèle généré, il est difficile de la tracer dans le modèle d'origine sans informations structurées.

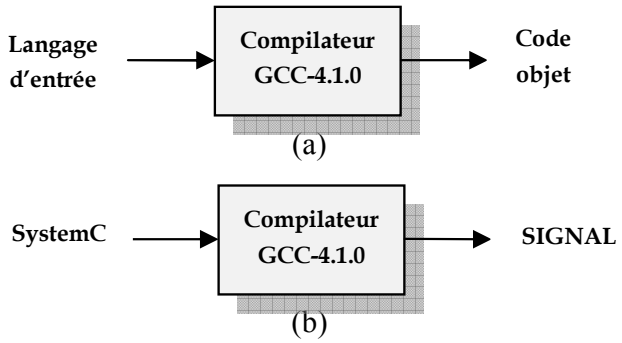

FIGURE 8.2 – GCC-4.1.0 avant (a) et après (b) l'adaptation

Pour permettre l'adaptation du compilateur GCC-4.1.0, il faut procéder à des modifications sur certains fichiers de la version source du compilateur (suppression de fonctions, insertion d'instructions, ...), puis le recompiler à nouveau (par un autre compilateur C sous Linux) et le réinstaller. L'ensemble des fichiers modifiés afin de permettre la génération automatique du code SIGNAL (voir figure **??**) sont :

- *my-pass.c* : ce fichier contient des fonctions qui implémentent les différentes phases d'optimisation.
- *my-signal-lib.h* : c'est un nouveau fichier entête qui contient toutes les nouvelles fonctions insérées pour permettre la génération du code SIGNAL.
- *my-tree-cfg.c* : ce fichier contient une seule fonction (*signal_dump_function_to_file()*), qui permet de générer le code SIGNAL équivalent à une fonction SYSTEMC.
- *my-tree-dump.c* : ce fichier permet la génération du nom de fichier SIGNAL de sortie (<*program_name.cpp.SIG*>).
- *my-tree-optimize.c* : parmi ces fonctions, on trouve signal_tree_rest_of_compilation(), dont la tâche est la génération du code SIGNAL.

- *my-tree-pretty-print.c* : ce fichier contient plusieurs fonctions telles que :
  - *signal-print-generic-expr()* : permettant de traduire les paramètres d'une fonction SYSTEMC en entrées et/ou sorties de processus SIGNAL (des fonctions SYSTEMC deviennent des processus SIGNAL).
  - *signal-print-generic-decl()* : traduire les variables déclarées dans une fonction SYSTEMC en variables locales de processus SIGNAL (décalées après le mot clé where).
  - *signal-dump-generic-bb()* : cette fonction est utilisée pour traduire le corps d'une fonction SYSTEMC en SIGNAL (code entre deux ¨ ¨).
  - *signal-dump-generic-node()* : cette fonction est l'implémentation des règles de traduction SYSTEMC-SIGNAL, proposées dans [**?**].

Les fichiers responsables de génération de format intermédiaire SSA, à partir de SYSTEMC sont :

- *gimplify.c* : ce fichier contient des fonctions qui nous permettent de générer le format GIMPLE (code à trois adresses).
- *tree-cfg.c* : ce fichier permet de générer le graphe de flot de contrôle (CFG).
- *tree-ssa.c* : ce fichier nous permet de générer du SSA.

8.5 L'outil de transformation SC2SIG

L'outil de transformation SC2SIG (SYSTEMC to SIGNAL) est la version adaptée du compilateur GCC-4.1.0. L'architecture de l'outil est schématisée dans la figure **??**.

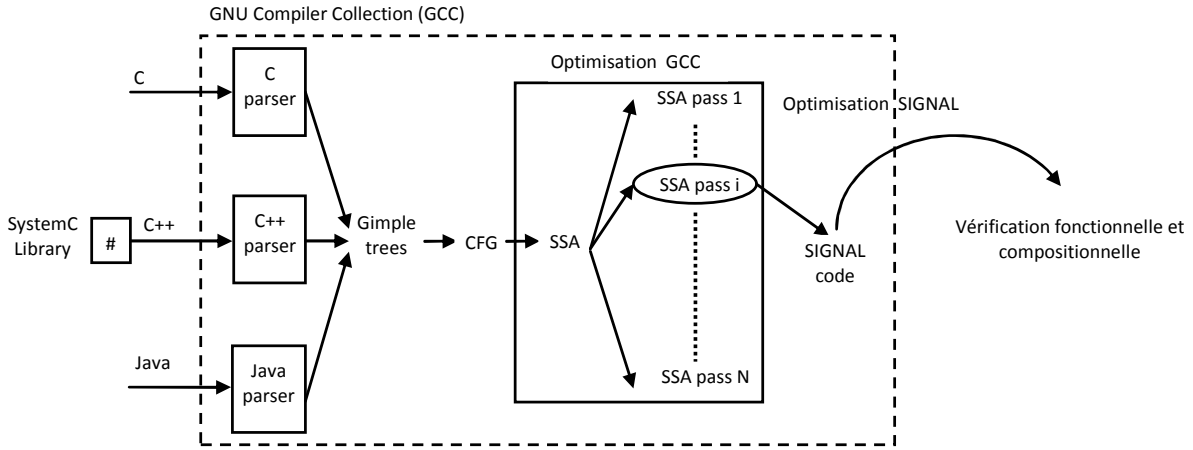

FIGURE 8.3 – Architecture de SC2SIG (GCC adapté)

8.6 Exemples de traduction

8.6.1 Exemple 1

Le code suivant est un exemple de programme SYSTEMC (*example.cpp*), c'est la description d'un module (module_example) équipé de deux signaux d'entrée (*input1* et *input2*), un signal de sélection (*select*) et un signal de sortie (*output*).

```
1  // example.cpp
   #include "systemc.h"
3
   SC_MODULE(module_example)
5   {
       sc_in<int> input1;
7      sc_in<int> input2;
       sc_in<bool> select;
9      sc_out<int> output;

11     void module_process()
       {
13       output = input1 + input2;
         module_function_example();
15     }

17     void module_function_example()
       {
19       int i,j;
         i=0;
21       i=i+1;
         j=j+i;
23     }

25     SC_CTOR(module_example) //constructor
       {
27       SC_METHOD(module_process);
         Sensitive << select ;
29     };
   };
```

Ce module permet de faire un simple traitement d'addition et englobant une petite fonction membre (*module_function_example()*) à titre d'exemple.

La traduction est réalisée suite au lancement du processus de compilation, en invoquant le compilateur adapté SC2SIG sous le système LINUX, par la ligne de commande suivante :

```
$ sc2sig -O fdump-tree-ssa -Wno-deprecated   -I/sc2sig_padh/ -c example.cpp
```

Avec :

- *-O -fdump-tree-ssa* : option pour forcer la génération du code SSA en sortie.
- *-Who-deprecated -I/sc2sig_path/ -c* : option pour inclure les fichiers de la nouvelle librairie SC2SIG.

La sortie est un programme équivalent en langage SIGNAL (example.cpp.SIG) :

```
% example.cpp.SIG
% SIGNAL implementation of Module process void module_example() %

process module_process =

 ( ? integer input1, input2 ;
   ? Boolean select ;
   ! integer output ;
 )

(| output := input1 + input2 when bb1
 | bb1 := true when select default false
 |)

where Boolean bb1;

% SIGNAL implementation of Function void module_function_example() (_Z1fv) %
process module_function_example =
 ( ? boolean clk_p;
   ! boolean R;
 )

(| i1 := (0) when bb0 default i1$
 | i2 := (i1 + 1) when bb0 default i2$
 | j4 := (j3$ + i2) when bb0 default j4$
 | R := true when bb0 default R$
 | bb0 := true when clk_p default false
 | zbb0 := bb0$ init true
 | clk_p ^= bb0 ^= j4 ^= i2 ^= i1 ^= R
 |)
where
   integer j4;
   integer i2, i1;
   boolean bb0, zbb0;
end; % void module_function_example()%
end; % void module_process()%
```

8.6.2 Exemple 2 : Finite Impulse Response

Considérons l'exemple de la figure **??** qui présente le filtre FIR (Finite Impulse Response) séparé dans trois composants IPs fonctionnels. Le bloc Stimulus génère des valeurs d'entrées (`sample`) pour FIR, FIR calcul, et le bloc Display affiche sur la sortie standard les les valeurs de sorties (`result`) reçues.

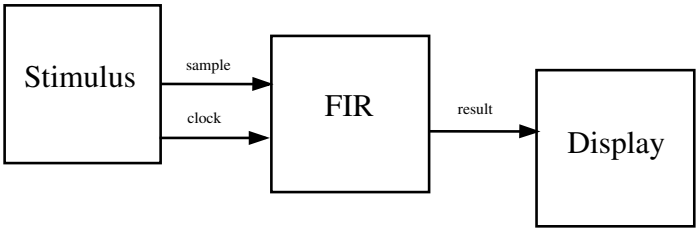

FIGURE 8.4 – IPs de FIR

Le code SYSTEMC du bloc FIR (les fichiers `fir.h` et `fir.cpp`) est :

```
// - fir.h - //

#include "systemc.h"
SC_MODULE (fir)
 {
   sc_in<bool> clock ;
   sc_in<float> sample ;
   sc_out<float> result ;
   float coeffs [16 ] ;
   float buff [16 ] ;

   void doit() ;
   SC_CTOR(fir) {
    SC_METHOD(doit) ;
    coeffs [0 ] = -6 ; coeffs [1 ]=-4 ; coeffs [2 ] = 13 ;coeffs [3 ] = 16 ;
    coeffs [4 ] = -18 ; coeffs [5 ] = -41 ;coeffs [6 ] = 23 ;
    coeffs [7 ] = 154 ; coeffs [8 ] = 222 ;coeffs [9 ] = 154 ;
    coeffs [10 ] = 23 ; coeffs [11 ] = -41 ;coeffs [12 ] = -18 ;
    coeffs [13 ] = 16 ; coeffs [14 ] = 13 ;coeffs [15 ] = -4 ;
    buff [0 ] = 0 ; buff [1 ] = 0 ; buff [2 ] = 0 ; buff [3 ] = 0 ;
    buff [4 ] = 0 ; buff [5 ] = 0 ; buff [6 ] = 0 ; buff [7 ] = 0 ;
    buff [8 ] = 0 ; buff [9 ] = 0 ; buff [10 ] = 0 ; buff [11 ] = 0 ;
    buff [12 ] = 0 ; buff [13 ] = 0 ; buff [14 ] = 0 ; buff [15 ] = 0 ;
    sensitive_pos<<clock ;}} ;

// - fir.cpp - //

#include "fir.h"
void fir :: doit()
  { // doit will calculate the output of FIR
    for(int i=16 ;i>0 ;i-)
      buff[i]=buff[i-1] ;
    buff[0]=sample ;
    float a=0 ;
    for(int i=0 ;i<16 ;i++)
    a+=coeffs[i]*buff[i] ;
    result=a ;
  } ;
```

Le code SYSTEMC du bloc Stimulus (les fichiers `stimulus.h` et `stimulus.cpp`) est :

```
// - stimulus.h - //

#include "systemc.h"
SC_MODULE(stimulus)
{
  sc_out<float> sample ; sc_out<bool> clock ;
  float send_value1 ;

  void do_generate() ;
  SC_CTOR(stimulus) {
    SC_METHOD(do_generate) ;
    dont_initialize() ;
    send_value1 = 0.0 ; }} ;

// - stimulus.cpp - //

#include "stimulus.h"
void stimulus : :do_generate()
  {
```

```
20      for(int i=0 ;i<=100 ;i++)
          { sample.write( send_value1 ) ;
22          clock.write( true ) ;
            send_value1 = send_value1 + 10.0 ;
24        }
    }
```

Le code SYSTEMC du bloc Display (les fichiers `display.h` et `display.cpp`) est

```
1 // - display.h - //
  #include "systemc.h"
3
  SC_MODULE (display)
5   {
      sc_in<float> result ;
7     void print_result() ;
      SC_CTOR( display )
9     {
        SC_METHOD( print_result ) ;
11      dont_initialize() ;}} ;

13 // - display.cpp - //

15 #include "display.h"
  void display : :print_result()
17 {
    cout << "Display : " << result << endl ;
19 } ;
```

Le code SIGNAL équivalent de l'exemple FIR est obtenu par notre transformation en deux étapes (extraction de la structure, extraction du comportement). Le code suivant est le résultat de l'extraction de la structure :

```
1 process stimulus = ( ! real sample ; boolean clock ;)
    (| |)
3
  process fir = ( ? real sample ; boolean clock ; ! real result)
5 (| |)

7 process display = ( ? real result)
  (| |)
9
  process main()
11 (
    |(real sample ; boolean clock) :=stimulus()
13  |(real result) :=fir(real sample ; boolean clock)
    |display(real result)
15 |)
   where
17   real sample, result ;
     boolean clock ;
19 end ;
```

Ensuite, nous générons le code complet pour chaque module. Le code SIGNAL de FIR est :

```
1 // -- fir.h -- //

3 (| doit() |)

5 // -- fir.cpp -- //
```

```
(| a5 := (0.0) when bb0  default a5\$
 | j8 := (16)  when bb0  default j8\$
 | j_023        := (j3$)                    when L0 default  j_023$
 | Tmp_D_174624 := (j3$ - 1)               when L0 default Tmp_D_174624$
 | Tmp_D_174725 := (buff[Tmp_D_174624]$) when L0 default  Tmp_D_174725$
 | array I to (0) of
     (| buff[j_023] := (Tmp_D_174725) when L0 default buff[j_023]$ |)
   end
 | j27 := (j3$ - 1) when L0 default j27$
 | j3  := j8 when bb0 default j27    when L0 default j3$
 | array I to (0) of
     (| buff[0] := (sample)  when L2 default buff[0]$  |)
   end
 | k12 := (0) when L2  default k12$
 | k_116 := (k4$)                      when L3  default k_116$
 | Tmp_D_175317 := (coeffs[k_116]$) when L3  default Tmp_D_175317$
 | k_118 := (k4$)                      when L3  default k_118$
 | Tmp_D_175419 := (buff[k_118]$)   when L3  default Tmp_D_175419$
 | Tmp_D_175520 := (Tmp_D_175317 * Tmp_D_175419) when L3  default  Tmp_D_175520$
 | a21 := (Tmp_D_175520 + a2$)     when L3  default a21$
 | k22 := (k4$ + 1)                when L3  default k22$
 | k4  := k12 when L2  default k22 when L3  default k4$
 | a2  := a5 when bb0  default a21 when L3  default a2$
 | result := (a2)  when L5  default result$
 | L5  := true when not ( k4  $<$= 15 ) when L4  default false
 | L3  := true when     ( k4$ $<$= 15 ) when L4$  default false
 | L4  := true when L2 default true   when L3       default false
 | L2  := true when not ( j3  $>$ 0 )   when L1  default false
 | L0  := true when     ( j3$ $>$ 0 )   when L1$ default false
 | L1  := true when bb0 default true  when L0  default false
 | bb0 := true when clock default false
 |)
```

Le code SIGNAL de Display est :

```
// -- display.h -- //

(| print_result() |)

// -- display.cpp -- //

(| write ("Display : ", result) |)
```

Le code SIGNAL de Stimulus est :

```
// -- stimulus.h -- //

(| do_generate()
 |)

// -- stimulus.cpp -- //

(| sample4      := (0.0) when bb0  default sample4$
 | send_value15 := (0.0) when bb0  default send_value15$
 | i6           := (0)   when bb0  default i6$
 | send_value19 := (send_value13$ + 1.0e+1) when L0 default send_value19$
 | sample10     := (send_value19)             when L0 default sample10$
 | i11          := (i1$ + 1)                  when L0 default i11$
 | send_value13 := send_value15 when bb0   default send_value19 when L0
                      %                    default send_value13$
```

```
 | sample2 := sample4 when bb0  default  sample10 when L0 default sample2$
 | i1          := i6  when bb0  default  i11      when L0 default i1$
 | Tmp_D_17417 := (sample2)     when L2   default Tmp_D_17417$
 | sample      :=  Tmp_D_17417  when L2   default sample$
 | clock       :=  true         when L2   default clock$
 | L2 :=  true when not ( i1  $<$= 100 )  when L1  default false
 | L0 :=  true when     ( i1$ $<$= 100 )  when L1$ default false
 | L1 :=  true when bb0 default true    when L0  default false
 | bb0 := true
 |)
```

Enfin, par exemple le code complet en SIGNAL du bloc FIR est :

```
// -- fir.h and fir.cpp -- //

process fir =
 (? real sample; boolean clock;
   ! real result
 )

(| doit()
 |)
 where
 [16]real coeffs init [ -6, -4, 13, 16, -18, -41, 23, 154, 222, 154, 23, -41,
                        -18, 16, 13, -4 ];
 [16]real buff   init [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0];
process doit =  ()
(| j8 := (16)   when bb0  default j8$
 | array I to (0) of
     (| buff[j3$] := (buff[(j3$ - 1),]$) when L0 default buff[j3$]$ |)
   end
 | j3 :=  j8 when bb0  default (j3$ - 1)   when L0 default j3$
 | array I to (0) of
      (| buff[0] := (sample)  when L2 default buff[0]$  |)
   end
 | k4  := (0)   when L2  default (k4$ + 1) when L3  default k4$
 | a2  := (0.0) when bb0 default ((coeffs[k4$]$ * buff[k4$]$)
         + a2$) when L3  default a2$
 | coeffs := coeffs$
 | result := (a2)  when L5  default result$
 | L5  :=  true when not ( k4  $<$= 15 )    when L4  default false
 | L3  :=  true when     ( k4$ $<$= 15 )    when L4$ default false
 | L4  :=  true when L2    default true   when L3  default false
 | L2  :=  true when not ( j3  $>$ 0 )      when L1  default false
 | L0  :=  true when     ( j3$ $>$ 0 )      when L1$ default false
 | L1  :=  true when bb0   default true   when L0  default false
 | bb0 :=  true when clock default false
 | L5 ^= L3 ^= L4 ^= L2 ^= L0 ^= L1 ^= bb0 ^= k4 ^= a2 ^= j3 ^=
         j8 ^= result ^= clock ^= sample
 |)
 where
  hspace{1ex}integer k4, j3, j8;
  hspace{1ex}real a2;
  hspace{1ex}boolean L5, zL5, L3, L4, L2, L0, L1, bb0;
 end;
end;
```

8.7 Compilation des modèles SIGNAL

8.7.1 Compilation en code C exécutable

Un programme SIGNAL peut être compilé par le compilateur SIG (voir figure **??**) pour générer du code C en vue d'une simulation du modèle.

```
> sig -c exemple.ccp.SIG
```

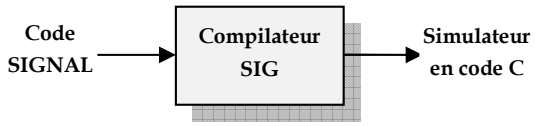

FIGURE 8.5 – Compilation de modèle SIGNAL en code C

8.7.2 Compilation en format Z3Z

Dans le but d'appliquer la vérification formelle en utilisant l'outil SIGALI, un programme SIGNAL peut être compilé par le compilateur SIG (voir figure **??**) en invoquant l'option -Z3Z pour permettre la génération d'un code polynômial, décrit dans une logique à trois états (-1, 0 et 1). Ce code est prouvable par l'outil SIGALI.

```
> sig -z3z exemple.ccp.SIG
```

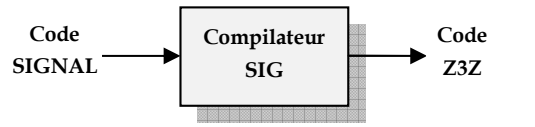

FIGURE 8.6 – Compilation de modèles SIGNAL en code Z3Z

8.8 Conclusion

Nous avons présenté dans ce chapitre l'implémentation SC2SIG de notre approche qui vise l'intégration des méthodes formelles synchrones dans le flot de conception des systèmes embarqués, dont la spécification initiale est décrite en SYSTEMC. Nous avons essayé de montrer dans quelle mesure la conception de systèmes embarqués peut profiter de l'utilisation de ces méthodes formelles. Notre approche consiste à obtenir la structure et l'implémentation du comportement de modèles SYSTEMC en utilisant les outils existants et donc se concentrer sur les problèmes réels.

Finalement, afin que notre outil (SC2SIG) supporte complètement SYSTEMC, une perspective consiste à l'étendre à tous les éléments syntaxiques et aussi de définir une bibliothèque équivalente dans SIGNAL pour permettre l'application de la vérification formelle sur n'importe quel modèle SYSTEMC.

Chapitre 9

Conclusion et perspectives

Les systèmes complexes à grande échelle et la réutilisation croissante de composants depuis des sources diverses rendent la validation de systèmes extrêmement difficile. Les tests et les simulations restent des méthodes de validation importantes, par contre ils ne suffisent plus pour un nombre d'applications de plus en plus importantes. Les modèles et les méthodes formelles peuvent être la solution possible à ces problèmes. Nous avons présenté dans ce document une approche qui vise l'intégration des méthodes formelles de catégorie synchrone dans le flot de conception des systèmes embarqués, il s'agit d'une méthodologie de conception de modèles formels SIGNAL à partir de descriptions SystemC non-formelles, et ceci afin de vérifier les spécifications système d'un haut niveau algorithmique avec prise en charge de l'aspect synchrone, dont la spécification initiale est décrite en SystemC. Nous avons montré à travers cette méthodologie, dans quelle mesure la conception de systèmes embarqués peut profiter de l'utilisation de ces méthodes formelles. Notre approche consiste à obtenir la structure et l'implémentation du comportement de descriptions SYSTEMC en utilisant des outils existants.

Le travail présenté dans ce document décrit une méthodologie avec laquelle il est possible d'extraire des modèles synchrones comportementaux formels à partir de descriptions SYSTEMC. Les avantages possibles d'un tel sujet sont évidents. Par contre, la puissance réelle d'une telle approche peut uniquement être démontrée et validée par un outil qui peut effectuer ces tâches automatiquement sur un sous-ensemble plus au moins complet du langage d'entrée. A l'etat actuel, il manque quelques étapes avant d'atteindre complètement ce but. Pendant que l'extraction automatique de la structure et la génération des structures des processus SIGNAL correspondants sont implémentées, il faut encore automatiser la transformation du code SSA vers SIGNAL au niveau des structures. Deux travaux ont été déja menés sur ce sujet et les premiers résultats sont présents dans [**?**] et [**?**]. Dès que le code SSA pourra être traduit automatiquement, une librairie SIGNAL correspondant à la librairie SYSTEMC en C++ devra être mise en place. Beaucoup de ces fonctions sont déja disponibles grâce aux travaux présentés dans [**?**], mais il faut encore les compléter et les tester. Un élément important de cette librairie sera la définition de l'équivalent de l'ordonnanceur SYSTEMC en SIGNAL, qui est indispensable pour assurer un comportement temps réel équivalent. Finalement,

la transformation SSA-SIGNAL devra être intégrée comme plugin dans l'outil Polychrony.

Une fois que l'extraction de types fonctionne de façon automatique, il serait possible de voir comment elle se comporte en ce qui concerne le passage à l'echelle des projets, et avec des différents types de systèmes. La transformation SSA-SIGNAL, une fois mise en place, devrait normalement fonctionner avec les autres langages supportés par GCC comme Fortran, Java et ADA, et ceci grâce à la caractéristique multilinguistique de la partie Front End de GCC. Donc il serait également intéressant de voir comment l'extraction comportementale marcherait pour ces langages. Une autre perspective serait d'etudier comment les performances de la simulation et de la vérification sont affectées à différents niveaux d'abstraction.

Finalement, sur le plan implémentation et afin que notre outil (SC2SIG) supporte complètement SYSTEMC et particulièrement les descriptions asynchrones, une dernière perspective consiste à l'étendre aux reste des éléments syntaxiques et de définir une bibliothèque équivalente dans SIGNAL pour permettre l'application de vérification formelle sur n'importe quel modèle SYSTEMC.

Bibliographie

[ACG92] Isabelle Attali, Jacques Chazarain, and Serge Gilette. Incremental Evaluation of Natural Semantic Specifications. Technical report, Research Report I3S 92-20, Programming Language Implementation and Logic Programming, LNCS 631, Leuven Belgique, 1992.

[All97] VSI Alliance. VSI, Alliance, Architecture Document, 1997.

[Anc97] François Anceau. *La Vérification Formelle de Circuits VLSI en Milieu Industriel*. Arago 20 - OFTA, 1997.

[Anl00] Matthias Anlauff. XASM - an Extensible, Component-based Abstract State Machines Language. In *International Workshop on Abstract State Machines, Lecture Notes on Computer Science (LNCS)*, 2000.

[Azi00] Mostafa Azizi. *Covérification des Systèmes Intégrés*. PhD thesis, Département d'Informatique et de Recherche Opérationnelle, Université de Montréal, Canada, December 2000.

[BCC+05] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An Overview of JML Tools and Applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7 :75–91, 2005.

[BCD+00] Peter Borovansky, Horatiu Cirstea, Hubert Dubois, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. ELAN : User Manual. Loria, Nancy, France, january 2000.

[Ber89] Gerar Berry. Real Time Programming : Special Purpose or General Purpose Languages. In *IFIP World Computer Congress*, 1989.

[BHD+05] David Berner, Patel Hiren, Mathaikutty Deepak, Jean-Pierre Taplin, and Shukla Sandeep. SystemCXML : An Extensible SystemC Front End Using XML. In *Proceedings of the Forum on specification and design languages (FDL)*, Lausanne, Switzerland, September 2005.

[BM88] Robert S. Boyer and J. Strother Moore. *A Computational Logic Handbook*. Academie Press Inc, 1988.

[Bry86] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *Computers, IEEE Transactions on*, C-35 :677 – 691, August 1986.

[BS99] Egon Börger and Wolfram Schulte. A Programmer Friendly Modular Definition of the Semantics of Java. J. Alves-Foss, editor, Formal Syntax and Semantics of Java, volume 1523 of Lect. Notes in Comp. Sci., pages 353–404. Springer-Verlag., 1999.

[CCO+04] Sagar Chaki, Edmund Clarke, Joël Ouaknine, Natasha Sharygina, and Nishant Sinha. State/event-based Software Model Checking. In *the International Conference On Integrated Formal Methods, LNCS 2999*, 2004.

[CGP00] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. Model Checking, 2000.

[CMNR10] Alessandro Cimatti, Andrea Micheli, Iman Narasamdya, and Marco Roveri. Verifying SystemC : a Software Model Checking Approach. In *Formal Methods in Computer-Aided Design (FMCAD)*, 2010.

[Col] GNU Compiler Collection. Free Software Foundation. The GNU Compiler Collection.

[Cor04] Microsoft Core. AsmL for Microsoft .NET Framework. Microsoft Core, 2004.

[Cou03] Philippe Coussy. *Synthèse d'Interface de Communication Pour les Composants Virtuels*. PhD thesis, Université de Bretagne Sud, France, 2003.

[CW04] Pasareanu Corina and Visser Willem. Verification of JAVA Programs Using Symbolic Execution and Invariant Generation. In *11th International SPIN SWorkshop on Model Checking Software*, 2004.

[DB97] Bruno Dutertre and Michel Le Borgne. SIGALI : un Système de Calcul Formel pour la Vérification de Programmes SIGNAL - Manuel d'utilisation. Technical report, IRISA/INRIA-Rennes, France, 1997.

[DG02] Rolf Drechsler and Daniel Große. Reachability Analysis for Formal Verification of SystemC. In *Proceedings of Digital System Design, 2002. Euromicro Symposium on*, 2002.

[EAH05] Christian J. Eibl, Carsten Albrecht, and Rainer Hagenau. gSysC : A Graphical Front End for SystemC. In *Proceedings of the 19th European Conference on Modeling and Simulation (ECMS), Riga, Latvia*, June 2005.

[EDG06] EDGC. Front-End, Edison Design Group C++ front-end. Website : http ://edg.com/cpp.html, 2006.

[Enc95] Emmanuelle Encrenaz. *Une Méthode de Vérification de Propriétés de Programmes VHDL basée sur des Modèles Formels de Réseaux de Petri*. PhD thesis, Laboratoire MASI, Université Pierre et Marie Curie, Paris, France, December 1995.

[FKL03] Harry Foster, Adam Krolnik, and David Lacey. *Assertion-based Design*. Kluwer, 2003.

[GD03] Daniel Große and Rolf Drechsler. Formal Verification of LTL Formulas for SystemC Designs. In *Proceedings of the 2003 International Symposium of Circuits and Systems 2003 (ISCAS '03)*, 2003.

[GD04] Daniel Große and Rolf Drechsler. Checkers for SystemC Designs. In *2nd ACM/IEEE Int conference on Formal Methods and Models for Co-design*, pages 171–178, 2004.

[GDLA03] Daniel Große, Rolf Drechsler, Lothar Linhard, and Gerhard Angst. Efficient Automatic Visualization of SystemC Designs. In *Forum on specification and design languages (FDL), Frankfurt, Germany*, September 2003.

[GKS03] Alain Girault, Hamoudi Kalla, and Yves Sorel. Une Heuristique d'ordonnancement et de Distribution Tolérante aux Pannes pour Systèmes Temps-Réel Embarqués. In *Modélisation des Systèmes Réactifs (MSR'03), Metz, France*. Hermes, October 2003.

[GM93] Mike J. C. Gordon and Tom F. Melham. *Introduction to HOL : A Theorem-Proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.

[GSCG01] Yuri Gurevich, Wolfram Schulte, Christopher S. Campbell, and Wolfgang Grieskamp. *Asml : The Abstract State Machine Language. Documentation prepared for Microsoft Research by Modeled Computation LLC. WebPage http ://research.microsoft.com/fse/asml*, 2001.

[Gur97] Yuri Gurevich. Draft of the ASM Guide. Technical report, EECS Department, University of Michigan, May 1997.

[Hal90] Anthony Hall. Seven Myths of Formal Methods. *IEEE Software*, 7, September 1990.

[Hal93] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer, 1993.

[HK11] Riadh Hocine and Hamoudi Kalla. Encoding SystemC Models in Formal Synchronous Formalism. *International Journal of Computer Applications (IJCA)*, 34(3) :26–32, November 2011.

[HKK12] Riadh Hocine, Hamoudi Kalla, and Salim Kalla. Une Approche Synchrone pour la Vérification des Systèmes Embarqués à base d'IP. In *International Conference : Sciences of Electronics, Technologies of Information and Telecommunication (SETIT'12)*, March 2012.

[HKKA12] Riadh Hocine, Hamoudi Kalla, Salim Kalla, and Chafik Arar. A Methodology for Verification of Embedded Systems based on SystemC. In *International Conference on Complex Systems (ICCS'12)*, November 2012.

[HKPM11] Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. The Coq Proof Assistant : A Tutorial. Technical report, WebPage at : http ://coq.inria.fr, INRIA, 2011.

[Hoc02] Riadh Hocine. Validation par Réseaux de Petri de Spécifications de Contrôleurs Dédiés Décrits dans un Environnement de Synthèse de Haut Niveau. Master's thesis, Département d'informatique, Université de Batna, 2002.

[HP00] Klaus Havelund and Thomas Pressburger. Model checking JAVA Programs using JAVA Pathfinder. *International Journal on Software Tools fo Technology Transfer (STTT)*, 4 :366–381, 2000.

[HT05a] Ali Habibi and Sofiene Tahar. Design for verification of SystemC Transaction Level Models. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 560 – 565 Vol. 1, march 2005.

[HT05b] Ali Habibi and Sofiene Tahar. On the Transformation of SystemC to AsmL Using Abstract Interpretation. *Electronic Notes in Theoretical Computer Science*, 131 :39–49, 2005.

[Hun94] Warren A. Hunt. *FM8501 : A Verified Microprocessor*. Springer-Verlag, 1994.

[IRI12] project ESPRESSO. IRISA. The polychrony workbench. http ://www.irisa.fr/espresso/Polychrony., 2012.

[IS03] C. Norris Ip and Stuart Swan. A Tutorial Introduction on the New SystemC Verification Standard. Technical report, www.systemc.org, 2003.

[ISO03] ISO/IEC. ISO/IEC 14882 :2003 International Standard - Programming Languages C++, October 2003.

[Jac99] Ludovic Jacomme. *Analyse Sémantique d'une Description VHDL Synchrone en vue de la Synthèse*. PhD thesis, Université Paris 6, France, October 1999.

[Joy89] Jeffrey J. Joyce. *Multi-Level Verification of Microprocessor-based Systems*. PhD thesis, Computer Laboratory, University of Cambridge, England, December 1989.

[KEP06] Daniel Karlsson, Petru Eles, and Zebo Peng. Formal Verification of SystemC Designs Using a Petri-Net Based Representation. In *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, volume 1, pages 1 –6, march 2006.

[Kin76] James C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7) :385–394, 1976.

[KS05] Daniel Kroening and Natasha Sharygina. Formal Verification of SystemC by Automatic Hardware/Software Partitioning. In *Proceedings of the Third ACM and IEEE International Conference, Formal Methods and Models for Co-Design (MEMOCODE '05)*, 2005.

[KSKH04] Zurab Khasidashvili, Marcelo Skaba, Daher Kaiss, and Ziyad Hanna. Theoretical Framework for Compositional Sequential Hardware Equivalence Verification in Presence of Design Constraints. In *the 2004 IEEE/ACM International Conference on Computer-Aided Design*, 2004.

[KTBB06] Hamoudi Kalla, Jean-Pierre Talpin, David Berner, and Loic Besnard. Automated Translation of C/C++ Models into Synchronous Formalism. In *Proceedings of the 13th Annual IEEE International Symposium and Workshop on (ECBS), Germany*, 2006.

[Lau99] Bernard Laurent. *Conception des Blocs Réutilisables ; Réflexion sur la Méthodologie*. PhD thesis, Institut National Polytechnique de Grenoble (INPG), France, Juin 1999.

[MBBG00] Hervé Marchand, Patricia Bournai, Michel Le Borgne, and Paul Le Guernic. Synthesis of Discrete-Event Controllers based on the SIGNAL Environment. *Discrete Event Dynamic System : Theory and Applications*, 10 :325–346, October 2000.

[McM92] Kenneth L. McMillan. *Symbolic Model Checking, An Approach to the State Explosion Problem*. PhD thesis, School of Computer Science, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1992.

[MJM11] Kevin Marquet, Bertrand Jeannet, and Matthieu Moy. Efficient Encoding of SystemC/TML in Promela. Technical report, Université Joseph Fourrier, Grenoble, France, 2011.

[MMMC06] Matthieu Moy, Florence Maraninchi, and Laurent Maillet-Contoz. LusSy : an Open Tool for the Analysis of Systems-on-a-Chip at the Transaction Level. *Design Automation for Embedded Systems*, 2006. special issue on SystemC-based systems.

[Mor00] Adam Jan Morawiec. *Amélioration des Performances de la Simulation des Modèles décrits en Langages de Description de Matériel*. PhD thesis, Université Joseph Fourrier, Grenoble1, France, October 2000.

[Mos99] Peter D. Mosses. CASL : A Guided tour of its Design. In recent Trends in Algebraic Development Techniques : Selected Papers from WADT'98, volume 1589 of LNCS, lisbon. Springer, 1999.

[MR02] Hervé Marchand and Eric Rutten. SIGNAL and SIGALI User's Manual. Technical report, Research Report IRISA / INRIA-Rennes, 2002.

[Nip98] Tobias Nipkow. Isabelle/HOL - the Tutorial. Institut für Informatik, Technische Universität München, 1998.

[Nov03] Diego Novello. Tree SSA - A New High-level Optimization Framework for the GNU Compiler Collection. In *Proceedings of the Nord/USENIX Users Conference*, February 2003.

[OS97] Sam Owre and Natarajan Shankar. The Formel Semantics of PVS. SRI, August 1997.

[Pan01] Preeti Ranjan Panda. SystemC : a Modeling Platform Supporting Multiple Design Abstractions. In *Proceedings of the 14th International Symposium on Systems Synthesis (ISSS'01)*, pages 75 – 80, 2001.

[PBdST09] Dumitru Potop-Butucaru, Robert de Simone, and Jean-Pierre Talpin. The Synchronous Hypothesis and Polychronous Languages. *Embedded Systems Design and Verification*, pages 6–16–6–27, 2009.

[Por06] Michele Portolan. *Conception d'un Système Embarqué Sûr et Sécurisé*. PhD thesis, Labotatoire TIMA, Institut National Polytechnique de Grenoble, France, December 2006.

[RDA+06] Armoni Roy, Korchemny Dmitry, Tiemeyer Andreas, Vardi Moshe, and Zbar Yael. Determinitic Dynamic Monitors for Lineair-time Assertions. In Springer, editor, *Workshop on Formal Approaches to Testing and Runetime Verification*, volume 4262. Lecture Note in computer Science, 2006.

[Rei95] Wolfgang Reif. The KIV-approach to Software Verification. Technical report, KORSO : Methods, Languages, and Tools for the Construction of Correct Software. Springer LNCS 1009, 1995.

[Sad07] Nabil Sadou. *Aide à la Conception des Systèmes Embarqués Sûrs de Fonctionnement*. PhD thesis, Laboratoire d'Analyse et d'Architecture des Systèmes du CNRS, Université Toulouse III - Paul Sabatier, France, November 2007.

[Sch01] Joachim Schmid. Introduction to AsmGofer. In *http ://www.uni-ulm-.de/ s_jschmi/AsmGofer.*, 26 March 2001.

[SDGK07] Rudrapatna K. Shyamasundar, Frederic Doucet, Rajesh K. Gupta, and Ingolf H. Krüger. Compositional Reactive Semantics Of SystemC And Verification With RuleBase. *Springer Science+Business Media*, 2007.

[Sir02] Brijesh Sirpatil. Software Synthesis of SystemC Models. Master's thesis, Faculty of the Virginia Polytechnic Institute, USA, 2002.

[SK02] Hayet Souffi-Kebbati. *Conception d'Opérateurs Numériques Réutilisables : Application à une Méthodologie d'Implantation Rapide et Optimale d'Algorithmes de Commande de Systèmes Electriques*. PhD thesis, Université Louis Pasteur, Strasbourg I, France, 2002.

[SM98] Luc Séméria and Giovanni De Micheli. SpC : Synthesis of Pointers in C Application of Pointer Analysis to the Behavioral Synthesis from C. In *Proceedings of the IEEE/ACM international conference on Computer-aided design (ICCAD '98)*, 1998.

[Sém01a] Luc Séméria. *Applying Pointer Analysis to the Synthesis of Hardware from C*. PhD thesis, Department of Electrical Engineering of Stanford University, USA, 2001.

[SM01b] Luc Séméria and Giovanni De Micheli. Encoding of Pointers for Hardware Synthesis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems - System Level Design*, Volume 9 Issues 6, 2001.

[Sny06] Wilson Snyder. SystemPerl - A Perl Library for SystemC. http ://www.veripool.com/systemperl.html, 2006.

[Sta93] IEEE Standard. IEEE Standard 1076-1993. IEEE Standard VHDL Language Reference Manual, 1993.

[Sta95] IEEE Standard. IEEE Standard 1364-1995. IEEE Standard Hardware Description Language based on the Verilog Hardware Description Language, 1995.

[Sta05] IEEE Standard. IEEE Standard 1666-2005. IEEE Standard SystemC Language Reference Manual, March 2005.

[Sys99] SystemC. Open SystemC Initiative (OSCI), 1999.

[Ter93] Delphine Terrasse. Translation from Typol to Coq. Pr. of the Tech. Work. BRA on Proving Properties of Programming Languages. Ed. J.Desperyroux, 1993.

[Ter95] Delphine Terrasse. *Vers un Environnement d'Aide au Développement de Preuves en Sémantique Naturelle*. PhD thesis, école Nationale des Ponts et Chaussées (ENPC), 1995.

[uIgpC01] Pcoq une Interface graphique pour Coq. Webpage at : http ://www-sop.inria.fr/lemme/pcoq. INRIA, 2001.

[Var07] Moshe Y. Vardi. Formal Techniques for SystemC Verification. In *Proceedings of the 44th annual Design Automation Conference (DAC '07)*, 2007.

[Zad11] Zadeck. *Static Single Assignment Book*. Springer, 2011.

[Zam98] Alexandre Zamulin. Specification of Dynamic Systems by Typed Gurvich Machines. In *Proceedings of the 13th International Conference on System Science*, pages 160–167, 1998.

Annexe A

Publications

- R. Hocine, H. Kalla, S. Kalla and C. Arar «A Methodology for Verification of Embedded Systems based on SystemC ». International Conference on Compelex Systems ICCS'12 (IEEE conference), Agadir, Morocco, November 2012.

- R. Hocine, H. Kalla and S. Kalla «Une approche Synchrone pour la Conception des Systèmes Embarqués à base d'IP ». International Conference : Sciences of Electronics, Technologies of Information and Telecommunication SETIT'12 (IEEE conference), Sousse, Tunisia, Mars 2012.

- R. Hocine and H. Kalla «Encoding SystemC Models in Formal Synchronous Formalism ». International Journal of Computer Applications IJCA (Published by Foundation of Computer Sciences, USA), Volume 34 - Number 3, November 2011.

« Une méthodologie de conception automatique de modèles formels pour des descriptions SYSTEMC *»*

Résumé :

La conception des systèmes embarqués par des langages de description de haut niveau rend leur vérification de plus en plus nécessaire. Le processus de vérification devient de plus en plus difficile lorsque on utilise des IPs (intellectual property) existantes pour la conception de ces systèmes. Ces IPs sont souvent hétérogènes et peuvent être de plusieurs sources, leur conception et leur intégration doivent être vérifiées. La vérification par simulation est largement utilisée, mais elle ne permet généralement pas d'obtenir une couverture exhaustive des erreurs. Il est donc important d'utiliser les techniques de la vérification formelle. Étant donné que nous visons des systèmes décrits par SYSTEMC qui ne présente pas de propriétés formelles, il est donc nécessaire de traduire ces descriptions SYSTEMC en modèles formels. Dans cet objectif, nous présentons dans cette thèse une méthodologie qui permet de générer automatiquement des modèles formels SIGNAL pour des descriptions non formels SYSTEMC. Le choix de SIGNAL est motivé par le fait qu'aujourd'hui ce langage est connecté à plusieurs outils de vérification formelle et que ses descriptions sont similaires à ceux de SYSTEMC. Le processus de génération est réalisé en deux étapes complémentaires : l'extraction structurelle et l'extraction comportementale. Enfin, nous illustrons la méthodologie proposée via le filtre FIR (Finite Impulse Response).

Mots-clés : Systèmes embarqués, langages de description de matériel, IP (Intellectual Property), méthodes formelles, SIGNAL, SYSTEMC, vérification fonctionnelle et compositionnelle, SSA, GCC.

« A methodology for automatic design of formal models for SYSTEMC *descriptions »*

Abstract :

The design of embedded systems with high-level description languages makes verification increasingly necessary. The verification process becomes more difficult when we use existing IPs (intellectual property) in system design. These IPs are often heterogeneous and can be from several sources, their design and integration must be verified. Verification by simulation is widely used, but it does not usually get comprehensive errors coverage. It is therefore important to use formal verification techniques. Since we are targeting systems designed with SYSTEMC which does not have formal properties, it is necessary to translate these SYSTEMC descriptions on formal models. For this purpose, we present in this thesis a methodology to automatically generate SIGNAL formal models for SYSTEMC non-formal descriptions. The choice of SIGNAL is motivated by the fact that today this language is connected to multiple formal verification tools and their descriptions are similar to those of SYSTEMC. The generation process is performed in two complementary steps : structural extraction and behavioral extraction. Finally, we illustrate the proposed methodology through the FIR filter (Finite Impulse Response).

Keywords : Embedded systems, hardware description languages, IP (Intellectual Property), formal methods, SIGNAL, SYSTEMC, functional and compositional verification, SSA, GCC.