Méthode de validation globale pour les systèmes monopuces

F. Husinger

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Nº attribué par la bibliothèque

THESE

Pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité : « Micro et Nano Electronique »

Préparée au **laboratoire TIMA** dans le cadre de **l'Ecole Doctorale d' : « Electronique, Electrotechnique, Automatique, Télécommunication, Signal »**

Présentée et soutenue publiquement

Par

Frédéric HUNSINGER

Le 6 mars 2006

Titre :

Méthode de validation globale pour les systèmes monopuces

Directeur de thèse : Ahmed Amine JERRAYA

Co-encadrant : Sébastien FRANCOIS

JURY

M. Dominique BORRIONE	, Présidente
M. Guy BOIS	, Rapporteur
M. Gérard BERRY	, Rapporteur
M. Ahmed Amine JERRAYA	, Directeur de thèse
M. Sébastien FRANCOIS	, Co-encadrant
M. Régis LEVEUGLE	, Examinateur
M. Christian BERTHET	, Invité

Remerciements

Durant ces trois années de thèse, j'ai eu l'honneur de travailler d'une part au laboratoire TIMA et d'autre part dans la société STMicroelectronics. Je tiens à exprimer mes remerciements à toutes les personnes qui ont contribué à l'accomplissement de cette thèse. Je remercie :

M. Ahmed Amine Jerraya directeur du groupe SLS du laboratoire TIMA, mon directeur de thèse qui m'a accueilli dans son équipe et qui m'a offert la possibilité de réaliser une thèse industrielle. Je le remercie également de m'avoir soutenu et d'avoir partagé ses connaissances.

M. Christian Berthet directeur R&D de la division TV de STMicroelectronics qui a rendu cette collaboration possible et au contact de qui j'ai énormément appris au travers de ses conseils et ses critiques.

M. Sébastien François manager de l'équipe de vérification de la division TV de STMicroelectronics dans laquelle j'ai effectué ma thèse et qui m'a aidé tout au long de celle-ci.

M. Gérard Berry directeur scientifique de Esterel Technologies et M. Guy Bois professeur à l'école Polytechnique de Montréal d'avoir accepté d'être les rapporteurs de mes travaux de thèse.

M. Régis Leveugle professeur à l'INPG d'avoir accepté de participer à mon jury de thèse.

Mme Dominique Borrione professeure à l'UJF d'avoir présidée mon jury de thèse et que je remercie aussi pour l'enseignement qu'elle m'a dispensé à Polytech'Grenoble.

J'aimerai également remercier toutes les personnes avec qui j'ai eu l'opportunité de collaborer ou que j'ai eu le bonheur de rencontrer pendant cette thèse :

Aux membres du groupe SLS du laboratoire TIMA : Nacer, Frédéric R., Frédéric P., Paul, Wander.

A tous les thésards du groupe SLS (anciens et actuels) : Damien, Lovic, Amer, Gabriela, Ferid, Sami, Ludovic, Arif, Arnaud, Adriano, Ivan, Lobna, Wassim, Aimen, Iuliana, Marius, Patrice, Alexandre B., Alexandre C., Lorenzo, Maxime.

A mes collègues de l'équipe de vérification : Thierry V., Michael, Denis, un merci tout spécial à Laurence pour la correction de ce mémoire et sa gentillesse.

Aux collègues de la division TV de STMicroelectronics : Thierry B., Frédéric S-P., Luc, Pierre-Hugues, Florence, Hung, Thierry G., Philippe, Cécile, Christophe M., Alexis, Bruno, Nicolas, Jean-Louis P, Jean-Louis D., Laurent, Cyril pour leur accueil.

Aux personnes de STMicorelectronics que j'ai eu la chance de rencontrer : Christophe C., Jérôme, Richard, Jean-José, Gilles, Jean-Philippe, Bruno, Thierry S., Olivier pour les repas distrayants que nous avons partagé.

Un grand merci à Lyes pour avoir partagé avec moi ses connaissances dans le domaine de la vérification formelle.

Un très grand merci à Sonja sans qui tout aurait été beaucoup plus triste et difficile.

Et toutes les personnes de TIMA et de STMircroelectronics que j'ai rencontrées.

A mes parents

&

To my wonderful girlfriend Sarah

Table des matières

INTRODUCTION **11**

CHAPITRE 1 : DE LA DIVERSITE DES CIRCUITS NUMERIQUES A LA CONCEPTION DES SYSTEMES MONOPUCES **15**

I. INTRODUCTION **16**

II. LES CIRCUITS NUMERIQUES **16**

II.A. LES COMPOSANTS MATERIELS 17
II.B. LES PROCESSEURS 17
II.C. LES SYSTEMES MONOPUCES 18
II.C.1. Présentation des systèmes monopuces *19*
II.C.2. La conception des systèmes monopuces et les interfaces logicielles / matérielles *20*

III. PROBLEMATIQUES DE LA VERIFICATION DES SYSTEMES MONOPUCES **23**

IV. OBJECTIFS ET CONTRIBUTIONS **24**

IV.A. OBJECTIFS 24
IV.B. CONTRIBUTIONS 25

V. CONCLUSION **25**

CHAPITRE 2 : LA DIVERSITE DES OUTILS ET METHODES DE VERIFICATION **27**

I. INTRODUCTION **28**

II. GENERALITES SUR LA VERIFICATION DES SYSTEMES NUMERIQUES **29**

II.A. DEFINITIONS 29
II.B. LES DEUX TECHNIQUES DE VERIFICATION 31

III. LES CRITERES D'ANALYSE ET D'EVALUATION DES METHODES DE VERIFICATION **31**

III.A. LES CRITERES D'ANALYSE 32
III.A.1. Le type du système sous test *33*
III.A.2. La modélisation du système *33*
III.A.3. Le type de vecteurs de test *33*
III.A.4. Le modèle d'exécution *34*
III.B. LES CRITERES D'EVALUATION 34
III.B.1. Taux de couverture *35*
III.B.2. Coût d'utilisation et de mise en œuvre *35*
III.B.3. Limitations *36*

IV. LA VERIFICATION DES BLOCS MATERIELS **36**

IV.A. LA VERIFICATION FORMELLE 36
IV.B. LA VERIFICATION DYNAMIQUE 38
I.A.1. L'environnement de test *38*
I.A.2. Les langages spécifiques pour la vérification du matériel (HVL : Hardware Verification Language) *41*

V. LA VERIFICATION DES PROCESSEURS **42**

V.A. LA VERIFICATION DYNAMIQUE BASEE SUR L'UTILISATION D'UN ENVIRONNEMENT DE TEST 42
V.B. LA VERIFICATION DYNAMIQUE BASEE SUR LES PROGRAMMES DE TEST 43

VI. LA VERIFICATION DES SYSTEMES MONOPUCES **46**

VI.A. LA VERIFICATION DYNAMIQUE BASEE SUR L'UTILISATION D'UN ENVIRONNEMENT DE TEST 46
VI.B. LA VERIFICATION DYNAMIQUE BASEE SUR LES PROGRAMMES DE TEST 47
VI.B.1. X-Gen *47*
VI.B.2. La TLV (Top Level Validation) *50*
VI.B.3. La méthode de vérification du PIAGET *53*

VII. CONCLUSION **56**

CHAPITRE 3 : PROPOSITION D'UNE METHODOLOGIE DE VALIDATION GLOBALE POUR LES SYSTEMES MONOPUCES 61

I. INTRODUCTION 62

II. LES INTERFACES LOGICIELLES / MATERIELLES DANS LE CADRE DE LA CONCEPTION ET DE LA VERIFICATION DES SYSTEMES MONOPUCES 62

III. DESCRIPTION DU FLOT DE VERIFICATION GLOBAL ET L'IMPORTANCE DES INTERFACES LOGICIELLES / MATERIELLES 66

III.A. LA DECOMPOSITION DU FLOT DE VALIDATION GLOBAL 66
III.B. LA VERIFICATION DES INTERFACES HW : METHODE FORMELLE 67
III.C. LA VERIFICATION DU RESEAU DE COMMUNICATION : METHODE DYNAMIQUE UTILISANT UN ENVIRONNEMENT DE TEST SPECIFIQUE 69
III.D. LA VERIFICATION DE L'INTEGRATION : METHODE UTILISANT DES PROGRAMMES DE TEST DE HAUT NIVEAU 72

IV. TECHNIQUES DE BASE POUR LA VERIFICATION DE L'INTEGRATION PAR DES PROGRAMMES DE TEST LOGICIEL DE HAUT NIVEAU 73

IV.A. DESCRIPTION DE L'ENVIRONNEMENT ROSES 73
IV.B. LE MODELE DE REPRESENTATION DES SYSTEMES : COLIF 75
IV.B.1. Une description structurelle modulaire 75
IV.B.2. Les différents niveaux d'abstraction 77
IV.B.3. Composants virtuels et ports hiérarchiques 77
IV.C. PRESENTATION DU GENERATEUR DE SYSTEME D'EXPLOITATION : ASOG 78
IV.C.1. Concept : le ciblage logiciel 78
IV.C.2. Les mécanismes d'assemblage 78
IV.C.3. Structure de la bibliothèque 78
IV.C.4. Le flot de génération automatique de systèmes d'exploitation 80

V. CONCLUSION 81

CHAPITRE 4 : PROPOSITION D'UNE METHODE DE VERIFICATION DE L'INTEGRATION POUR LES SYSTEMES MONOPUCES 83

I. INTRODUCTION 84

II. PRESENTATION DE LA METHODE DE VERIFICATION DE L'INTEGRATION 84

III. PROPOSITION D'UNE METHODE DE GENERATION DES PROGRAMMES DE TEST LOGICIEL DE HAUT NIVEAU 86

III.A. LA MODELISATION DES COMPOSANTS ET LE CONCEPT DE SERVICES OFFERTS ET SERVICES REQUIS 88
III.B. LA MODELISATION DU SYSTEME 90
III.B.1. Présentation de la modélisation des composants 91
III.B.2. Présentation des éléments de description d'un composant 92
III.B.3. Présentation de la structure du modèle d'un système 95
III.B.4. Résumé des points clé de la modélisation 97
III.C. LES PROGRAMMES DE TEST LOGICIEL DE HAUT NIVEAU 98
III.D. LA GENERATION DES PROGRAMMES DE TEST LOGICIEL 102
III.D.1. Les programmes de test de bas niveau pour le test des registres 104
III.D.2. Les programmes de test de haut niveau 104

IV. LE RAFFINEMENT DES PROGRAMMES DE TEST LOGICIEL DE HAUT NIVEAU : LE GENERATEUR DE SYSTEME D'EXPLOITATION 107

IV.A. LES PROGRAMMES DE TEST DE HAUT NIVEAU ET LA SPECIFICATION DES PARAMETRES DE MODELISATION DE LA PARTIE LOGICIELLE ET DE L'API 109
IV.B. LA BIBLIOTHEQUE DES ELEMENTS DU SYSTEME D'EXPLOITATION 111
IV.C. LA GENERATION AUTOMATIQUE DU SYSTEME D'EXPLOITATION ET LES PROGRAMMES DE TEST DE BAS NIVEAU 115

V. CONCLUSION 116

CHAPITRE 5 : ETUDE DE CAS : APPLICATION DE LA METHODE SUR UN SYSTEME INDUSTRIEL 119

I. INTRODUCTION 120

II. LE SYSTEME PIAGET 120

II.A. GENERALITES 120
II.B. LE SOUS SYSTEME SELECTIONNE 122

III. L'APPLICATION DE LA METHODE 123

III.A. LA MODELISATION DU SOUS-SYSTEME 124
III.B. LES PROGRAMMES DE TEST DE HAUT NIVEAU 128
III.C. LA BIBLIOTHEQUE DU SYSTEME D'EXPLOITATION ET L'API DE VERIFICATION 131
III.D. LE RAFFINEMENT DES PROGRAMMES DE TEST LOGICIEL DE HAUT NIVEAU 133

IV. RESULTATS ET ANALYSES 135

IV.A. RESULTATS 136
IV.B. ANALYSES 137

V. CONCLUSION 138

CONCLUSION ET PERSPECTIVES 141

Liste des figures

Figure 1 : Représentation d'un bloc matériel et de son équivalent en terme de logiciel 17
Figure 2 : Représentation simplifiée de la composition d'un processeur 18
Figure 3 : Représentation de la composition d'un système monopuce 19
Figure 4 : Représentation en couche de la structure d'un système monopuce 21
Figure 5 : Représentation de l'architecture d'un système monopuce dans le flot de conception et de vérification idéal 22
Figure 6 : Présentation générale du flot de conception et de vérification des systèmes électroniques 30
Figure 7 : Les critères d'analyse des méthodes de vérification 32
Figure 8 : Représentation pour la vérification dynamique, des deux types d'élément générant les stimuli : (a) Environnement de test ; (b) Programmes de test 34
Figure 9 : Représentation du flot de vérification basée sur l'utilisation d'un environnement de test dans le flot de vérification et conception des systèmes électroniques 39
Figure 10 : Représentation du flot Genesys-Pro dans le flot de vérification et de conception des systèmes électroniques 44
Figure 11 : Flot de vérification des systèmes monopuces : X-Gen (IBM) dans le flot de vérification et conception idéal 48
Figure 12 : Flot de vérification des systèmes monopuces : TLV (Top Level Validation : Esterel Techonologies) dans le flot de conception et de vérification idéal 51
Figure 13 : Représentation des parties vérifiées dans un système monopuce par les méthodes (a) X-Gen, (b) TLV et (c) écriture manuelle de programmes de test 58
Figure 14 : Représentation de l'architecture d'un système monopuce dans le flot de conception et de vérification idéal 63
Figure 15 : Représentation de la composition de l'interface logicielle / matérielle 64
Figure 16 : Représentation de la vérification sur le modèle en couche d'un système monopuces : (a) vérification de l'interface matérielle, (b) vérification du réseau de communication, (c) vérification de l'intégration et la validation de l'interface logicielle 66
Figure 17 : Représentation du flot de génération de propriétés pour la vérification des interfaces matérielles basé sur le protocole STBus (COPS) 69
Figure 18 : Représentation schématique de la génération d'un environnement de test spécifique pour l'intégration d'un réseau de communication (STBus) 71
Figure 19 : Représentation du flot de conception des systèmes monopuces : ROSES du groupe SLS du laboratoire TIMA 75
Figure 20 : Illustration de la représentation Colif d'un système 76
Figure 21 : Représentation du flot du générateur de systèmes d'exploitation : ASOG 80
Figure 22: Présentation du flot de la méthode de vérification proposée et illustrée sur le flot de conception et de vérification idéal 85
Figure 23 : Présentation du flot de génération des programmes de test logiciel de haut niveau basés sur une API. 86
Figure 24 : Représentation de la modélisation des services (a) et illustration par un exemple d'un composant issue du système PIAGET (b) 89
Figure 25 : Présentation de la modélisation d'un composant matériel (a) et d'un composant logiciel (b) 91
Figure 26 : Présentation de la définition des registres sous la forme d'une grammaire et exemple de description 93
Figure 27 : Exemple de modélisation pour le composant HDPP des paramètres d'abstraction des entrées/sorties et de la configuration 95

Figure 28 : Illustration de la modélisation d'un système 96
Figure 29 : Illustration de la représentation informatique de la structure du système de la Figure 28 97
Figure 30 : Illustration de la différence entre un programme de test de haut niveau et un programme de test de bas niveau 99
Figure 31 : Exemple de la structure d'un programme de test de haut niveau (a) et de la mise en œuvre la synchronisation par un exemple d'exécution du logiciel de test (b) 100
Figure 32 : Exemple de profil des programmes de test de haut niveau et de la structure d'une tâche 101
Figure 33 : Présentation des règles de test 103
Figure 34 : Le flot de génération détaillé des programmes de test de haut niveau 107
Figure 35 : Présentation du flot de raffinement des programmes de test logiciel de haut niveau 108
Figure 36 : Les programmes de test de haut niveau et la spécification de l'API et des paramètres de modélisation de la partie logicielle pour la génération du système d'exploitation. 110
Figure 37 : La bibliothèque des éléments du système d'exploitation initiale 112
Figure 38 : La bibliothèque des éléments du système d'exploitation et l'ajout de l'API 114
Figure 39 : Présentation du raffinement des programmes de test de haut niveau par la génération d'un système d'exploitation spécifique 116
Figure 40 : Synoptique de la structure du système PIAGET 121
Figure 41 : Schéma du sous-ensemble sélectionné pour l'étude 122
Figure 42 : Modélisation de la structure du sous-ensemble du PIAGET et des services pour les composants non adressés par la vérification 125
Figure 43 : Modélisation des services pour les composants : HDPP (a) et HVD (b) 126
Figure 44 : Modélisation des services pour les composants : NR (a) et T2LUMA (b) 127
Figure 45 : La modélisation des services du composant Sync & Conv 127
Figure 46 : Représentation Colif de la structure logicielle pour les tests adressant l'intégration du composant HDPP 129
Figure 47 : Représentation Colif de la structure logicielle du test d'intégration pour le sous-système Uncompressed 129
Figure 48 : Illustration du code de la tâche : task_aInit 130
Figure 49 : Illustration du code de la tâche : task_b_uds_T2main 130
Figure 50 : Illustration du code de la tâche : task_b_uds_main 131
Figure 51 : Représentation de la bibliothèque du système d'exploitation étendue 132
Figure 52 : Illustration des paramètres spécifiant les services pour la tâche task_b_uds_T2main 133
Figure 53 : Illustration des paramètres spécifiant les interruptions 134
Figure 54 : Illustration du code généré pour l'appel système permettant la configuration du composant HDPP 135
Figure 55 : Représentation des parties vérifiées dans un système monopuce par la méthodologie de validation globale proposée 143

Liste des tableaux

Tableau 1 : Tableau récapitulatif sur les outils et méthodes de vérification dynamique 57
Tableau 2 : Comparaison entre les programmes de test de haut niveau et de bas niveau 137

INTRODUCTION

L'avancée technologique, dans le domaine des circuits intégrés, permet maintenant de mettre sur une seule puce ce qui était auparavant réalisé sur une ou plusieurs cartes électroniques. La miniaturisation et le fonctionnement plus rapide apportés par ces systèmes monopuces les placent au cœur de tous les produits de nouvelles technologies (vidéo, téléphonie, automobile...). Ces systèmes sont constitués de dizaines de millions de transistors et se composent de plusieurs processeurs avec leurs périphériques. Le nombre de composants que l'on trouve maintenant dans ces systèmes rend la validation de plus en plus complexe et longue.

Cette augmentation croissante de la complexité des systèmes monopuces (SoC : pour System on Chip) pousse à la recherche de nouveaux outils et méthodes de mise au point. La validation ou le test fonctionnel consiste en la vérification du bon fonctionnement d'un système par rapport à la spécification. Elle est très importante lors de la conception d'un SoC, et correspond à l'étape la plus coûteuse en temps. Actuellement, la validation d'un système correspond à 70% du temps passé pour la conception. Il est donc nécessaire, pour satisfaire les délais de mise sur le marché d'un produit, d'apporter de nouvelles solutions afin de réduire le temps nécessaire à la phase de validation.

L'étude des pratiques industrielles montre que les outils et méthodes de vérification actuels, bien qu'efficaces, sont généralement spécialisés pour des parties du circuit ou des étapes particulière du cycle de conception.

Le but de cette thèse est de définir une méthode de validation globale pour les systèmes monopuces et de l'appliquer à un système industriel conçu par l'équipe TV de STMicroelectronics. Le système utilisé dans ce contexte s'appelle le PIAGET, il est destiné aux applications pour la télévision numérique haute définition. Ce système permet tout au long de ce mémoire d'illustrer les différents travaux réalisés.

Ce mémoire décrit une proposition de méthodologie pour la vérification des systèmes sur puce. Cette méthodologie exploite différentes méthodes existantes pour adresser les spécificités de la vérification des systèmes sur puces et ainsi bénéficier des avantages de chacune. Cette méthodologie consiste pour l'essentiel à réaliser la vérification par simulation. Cette proposition de méthodologie intègre une nouvelle méthode de vérification qui cible la vérification de l'intégration de composants autour des processeurs. Cette méthode se base sur l'utilisation de programmes de test logiciel de haut niveau qui sont exécutés par les processeurs contenus dans le système.

Ce mémoire se décompose en cinq chapitres. Le premier chapitre décrit le contexte de la conception et de la vérification des systèmes sur puce. Cette description montre l'évolution de la conception des circuits numériques. Cette étude permet d'exhiber la complexité des systèmes monopuces et de leur vérification.

Le deuxième chapitre présente un état de l'art sur la vérification des systèmes numériques. Il décrit, dans un premier temps, les méthodes et outils de vérification généraux. Ces méthodes et outils sont applicables à tous les types de circuits numériques. Dans un deuxième temps, ce chapitre présente les outils et méthodes de vérification employés spécifiquement pour la vérification des processeurs. Il s'achève en présentant les outils et méthodes spécifiques à la vérification des systèmes monopuces. Cette section sur la vérification des systèmes monopuces contient également une description de la méthode de vérification utilisée pour le système PIAGET.

Le troisième chapitre présente la méthodologie de validation globale pour les systèmes monopuces qui est proposée. Cette méthodologie se décompose en trois parties spécifiques pour réaliser la vérification efficace d'un système sur puce. Ces trois parties ciblent les composants, l'interconnexion et l'intégration.

Le quatrième chapitre constitue la description de la principale contribution de ces travaux de thèse. Il s'agit de la présentation d'une méthode de vérification de l'intégration pour les systèmes sur puce. Cette méthode utilise les processeurs que contient le système pour le stimuler. Elle consiste en la génération de programmes de test logiciel de haut niveau. Ces programmes de test logiciel de haut niveau sont construits sur une API (Application Programming Interface). Le raffinement de ces tests est ensuite réalisé par un générateur de système d'exploitation spécifique. Cette méthode autorise l'emploi de mécanisme de synchronisation qu'offre un système d'exploitation.

Le cinquième chapitre de ce mémoire réalise la description d'une étude de cas. Cette étude de cas se rapporte à l'application de la méthode de vérification de l'intégration présentée au chapitre quatre. Elle est réalisée sur un sous-ensemble du système PIAGET, décrit brièvement dans ce chapitre. Il comprend également la présentation de l'application de la méthode de vérification ainsi qu'une analyse des résultats obtenus.

Le mémoire s'achève par la conclusion et par la description des perspectives qu'offrent ces travaux de thèse.

Chapitre 1 : DE LA DIVERSITE DES CIRCUITS NUMERIQUES A LA CONCEPTION DES SYSTEMES MONOPUCES

I. INTRODUCTION 16

II. LES CIRCUITS NUMERIQUES 16

II.A. LES COMPOSANTS MATERIELS 17
II.B. LES PROCESSEURS 17
II.C. LES SYSTEMES MONOPUCES 18
II.C.1. Présentation des systèmes monopuces 19
II.C.2. La conception des systèmes monopuces et les interfaces logicielles / matérielles 20

III. PROBLEMATIQUES DE LA VERIFICATION DES SYSTEMES MONOPUCES 23

IV. OBJECTIFS ET CONTRIBUTIONS 24

IV.A. OBJECTIFS 24
IV.B. CONTRIBUTIONS 25

V. CONCLUSION 25

I. Introduction

Ce premier chapitre réalise la présentation de la diversité des circuits numériques et montre l'évolution réalisée dans la conception des systèmes électroniques qui permet l'assemblage sur une puce de circuits qui étaient auparavant assemblés sur une carte. Cette évolution permet de mettre en évidence la complexité des systèmes monopuces.

Dans un deuxième temps, ce chapitre présente les problèmes qui sont rencontrés dans le cadre de la vérification fonctionnelle des systèmes monopuces. La description de ces problèmes est suivie par la définition des objectifs fixés pour cette thèse. Enfin, ce premier chapitre se termine par la présentation des contributions apportées au cours de ces travaux de recherche.

II. Les circuits numériques

Cette section présente une classification des systèmes électroniques, on distingue trois catégories de circuits numériques :

- La catégorie des **composants matériels** ou encore IP matérielle (Intellectual Property) ;
- La catégorie des **processeurs** qui comprend l'ensemble des composants exécutant du logiciel comme les microcontrôleurs, les processeurs ou les DSP (Digital Signal Processor) ;
- La catégorie des **systèmes monopuces** ou système sur puce définie comme étant des systèmes intégrant sur une même puce un ou plusieurs processeurs et des composants matériels.

Cette classification est également le fruit de l'évolution technologique. En effet, la miniaturisation permet de nos jours de concevoir des systèmes comprenant de plus en plus de fonctionnalités (blocs matériels) qui étaient dans le passé des circuits séparés. Les systèmes monopuces résultent en partie de cette capacité d'intégration.

II.A. Les composants matériels

Cette section décrit les blocs matériels et présente leurs principales caractéristiques. La Figure 1 donne une représentation d'un bloc matériel et son équivalent dans le domaine du logiciel sous la forme d'une fonction. Cette équivalence a un intérêt tout particulier dans le cadre de la vérification. Elle permet d'introduire un formalisme mathématique qui permet de mettre en œuvre des méthodes pour prouver la validité du bloc matériel comme cela est le cas pour une fonction logicielle.

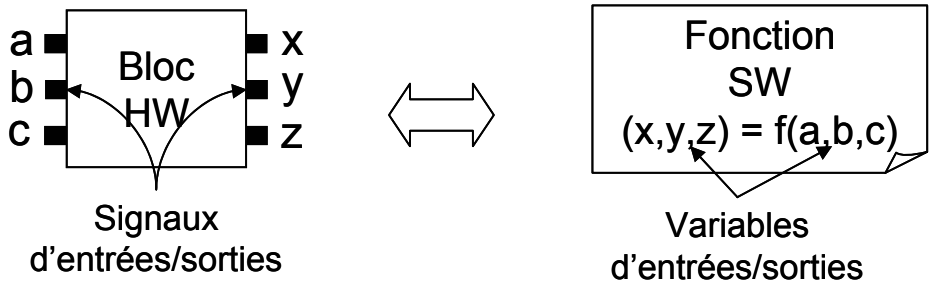


Figure 1 : Représentation d'un bloc matériel et de son équivalent en terme de logiciel

Un bloc matériel est un composant réalisant une fonction donnée, tout comme une fonction logicielle. Il est spécifié par son comportement externe. D'un point de vue du matériel cela correspond aux signaux d'entrées/sorties et du point de vue logiciel aux variables d'entrées et de sorties de la fonction.

II.B. Les processeurs

Cette section décrit dans les principales caractéristiques des composants tels que les microprocesseurs, les processeurs de traitement du signal (DSP : Digital Signal Processor), ainsi que tous les différents types d'architecture des processeurs (VLIW : Very Large Instruction Word, RISC : Reduced Instruction Set Chip, CISC : Complex Instruction Set Chip …). La Figure 2 donne une représentation simplifiée d'un processeur et de sa composition.

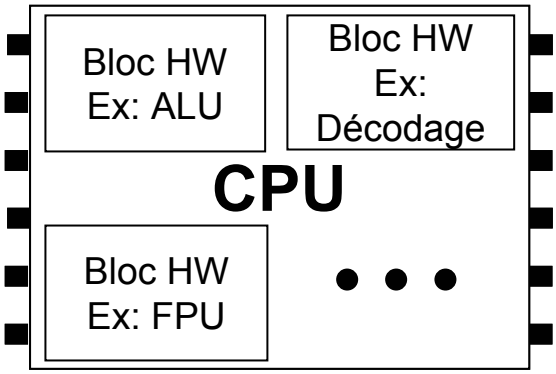


Figure 2 : Représentation simplifiée de la composition d'un processeur

Les processeurs actuels intègrent des fonctionnalités très complexes comme les mécanismes de cache, l'exécution déséquencée ou encore le branchement prédictif. Ces fonctionnalités sont réalisées par des blocs matériels. La complexité des processeurs est également liée aux domaines d'utilisation très divers dont ils font l'objet. A la différence des blocs matériels qui réalisent une fonction donnée, les processeurs ne sont pas uniquement définis à partir des entrées/sorties, mais ils le sont également par leur jeu d'instructions. Les processeurs exécutent en effet une partie logicielle correspondant à la (les) fonction(s) à réaliser. Ils sont capables de changer de comportement suivant le logiciel qu'ils exécutent. Enfin, comme le montre la Figure 2 l'architecture d'un processeur peut se décomposer en blocs matériels. Ceux-ci sont bien définis comme l'unité de chargement d'instructions, l'unité de décodage ou encore les unités d'exécution (chargement/enregistrement de données en mémoire, calcul flottant…). Les processeurs sont ainsi un assemblage de blocs matériels dont les fonctionnalités et les interfaces sont bien définies.

II.C. Les systèmes monopuces

Les systèmes monopuces sont conçus par assemblage de composants (bloc matériel, processeurs). On peut donner quelques exemples de systèmes sur puce : PIAGET et NOMADIK™ (STMicroelectronics), NEXPERIA™ (Philips) ou encore CELL™ (IBM, Sony, Toshiba). La combinaison de blocs matériels et de processeurs sur une même puce introduit le concept de l'interface logicielle/matérielle.

II.C.1. Présentation des systèmes monopuces

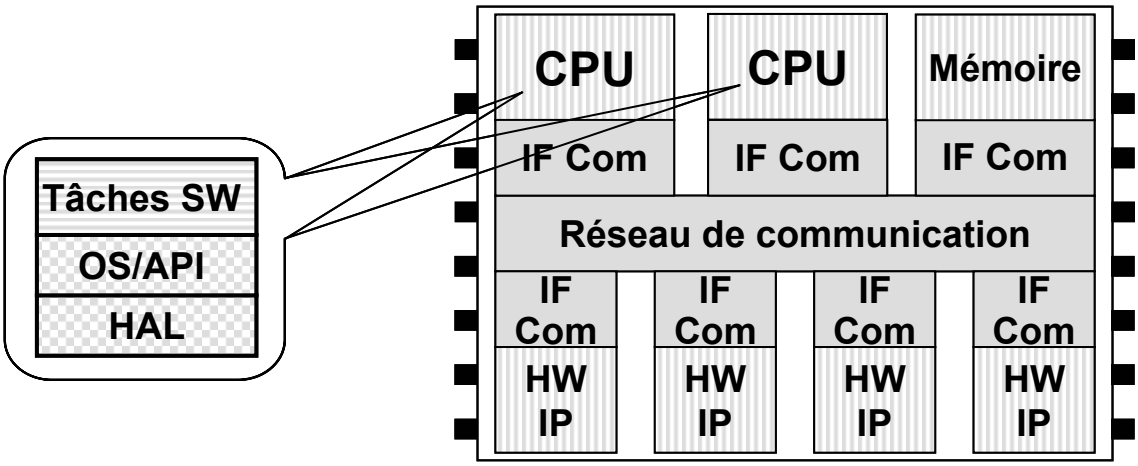


Figure 3 : Représentation de la composition d'un système monopuce

La troisième famille de circuit définie est donc celle des systèmes monopuces et multiprocesseurs monopuce (MPSoC). Ces systèmes sont un assemblage de composants des deux familles présentées auparavant (blocs matériels et processeurs, respectivement IP HW et CPU sur la Figure 3). Dans le cas de ces systèmes, on fait généralement l'hypothèse que les processeurs et les composants matériels ont déjà été vérifiés. Tout comme pour les processeurs, ces systèmes peuvent être très complexes. Ce sont des assemblages de composants hétérogènes exécutant un logiciel qui peut lui aussi être complexe.

Une autre différence importante des systèmes sur puce par rapport au processeur est la présence d'un réseau de communication. Il est en général conçu spécifiquement pour l'application afin de respecter les besoins par exemple en bande passante ou en latence. Ainsi pour certaines applications, ces réseaux peuvent devenir très complexes et comporter plusieurs bus hiérarchiques. L'intégration d'un réseau de communication nécessite en général la conception d'interface de communication (IF Com sur la Figure 3) entre les composants et le réseau. Enfin, les domaines d'application de tels systèmes sont plus spécifiques que ceux des processeurs. Une autre différence des systèmes monopuces est qu'ils s'appuient fortement sur la réutilisation de composants matériels c'est pourquoi sur les figures, les blocs matériels sont notées HW IP pour Hardware Intellectual Property. Il faut dans de tels systèmes

s'attacher à vérifier les interactions entre les différents composants : on parle de vérification de l'intégration.

La spécification des systèmes monopuces est réalisée en partie par l'intermédiaire des signaux d'entrées/sorties. Mais, comme pour les processeurs, cela n'est pas suffisant. Ainsi, ces systèmes sont également définis par le logiciel embarqué nécessaire au fonctionnement des processeurs et du système. Cette partie est représentée sur la gauche de la Figure 3 et se compose de : tâches SW (logiciel applicatif), OS/API (Operating System), HAL (Hardware Abstraction Layer). En effet, le logiciel fait partie intégrante de ces systèmes et prend une part de plus en plus importante au fur et à mesure que le nombre de processeurs intégrés augmente. La section suivante décrit la conception de tels systèmes afin de présenter le besoin de vérification conjointe du logiciel et du matériel.

II.C.2. La conception des systèmes monopuces et les interfaces logicielles / matérielles

Les systèmes monopuces sont composés de parties logicielles et matérielles. Afin de décrire un tel système de manière plus explicite, la Figure 4 donne une représentation de la structure en couche d'un système monopuce. Dans cette structure on note qu'il y a quatre types d'éléments principaux :

- Le logiciel applicatif ;
- Les processeurs ;
- Les blocs matériels ;
- Le réseau de communication.

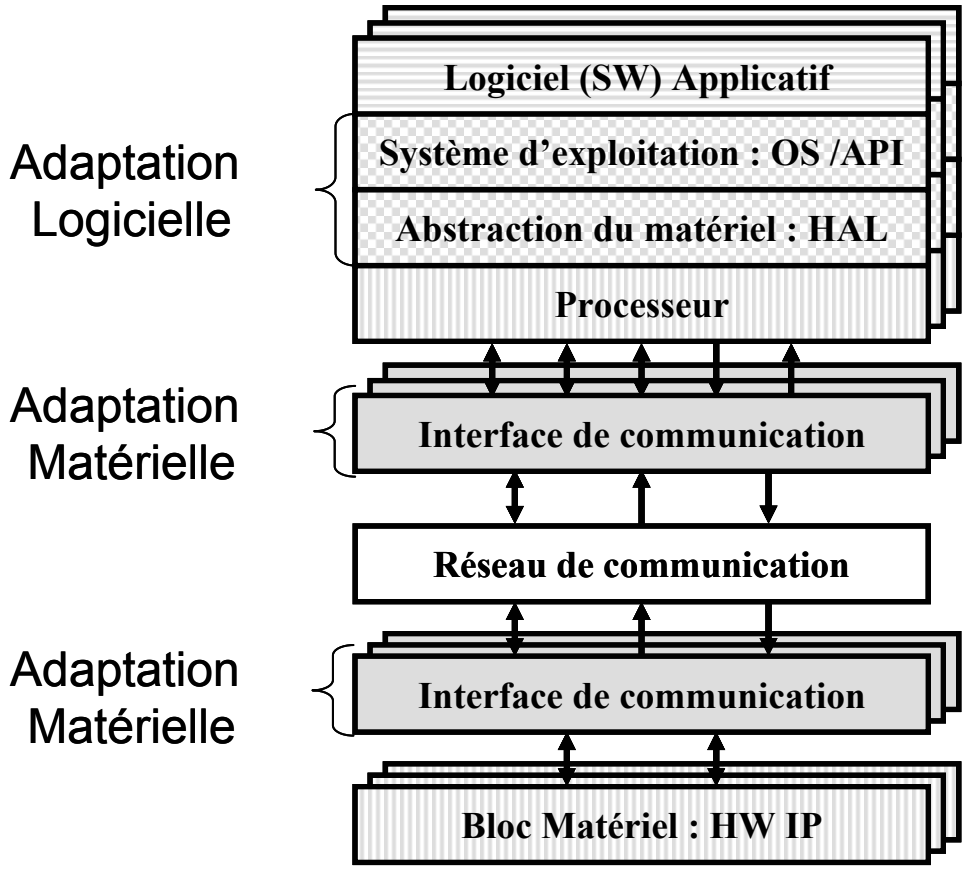


Figure 4 : Représentation en couche de la structure d'un système monopuce

D'autre part, lors de la conception d'un système monopuce, il est nécessaire pour effectuer l'intégration de tous ces éléments de réaliser à la fois des adaptations logicielles et des adaptations matérielles (Figure 4). L'adaptation logicielle réalise l'interface entre le logiciel applicatif et le processeur. L'adaptation matérielle réalise quant à elle l'interface pour la communication entre les différents composants matériels et le réseau de communication.

La Figure 5 présente l'évolution de l'architecture d'un système monopuce dans le flot de conception et de vérification idéal. Cette évolution est décrite de la phase de spécification du système à la réalisation du système au niveau RTL (Register Transfert Level). Elle montre les changements de l'architecture pendant le raffinement d'un niveau d'abstraction abstrait (par exemple TLM : Transaction Level Modelling) et le niveau RTL.

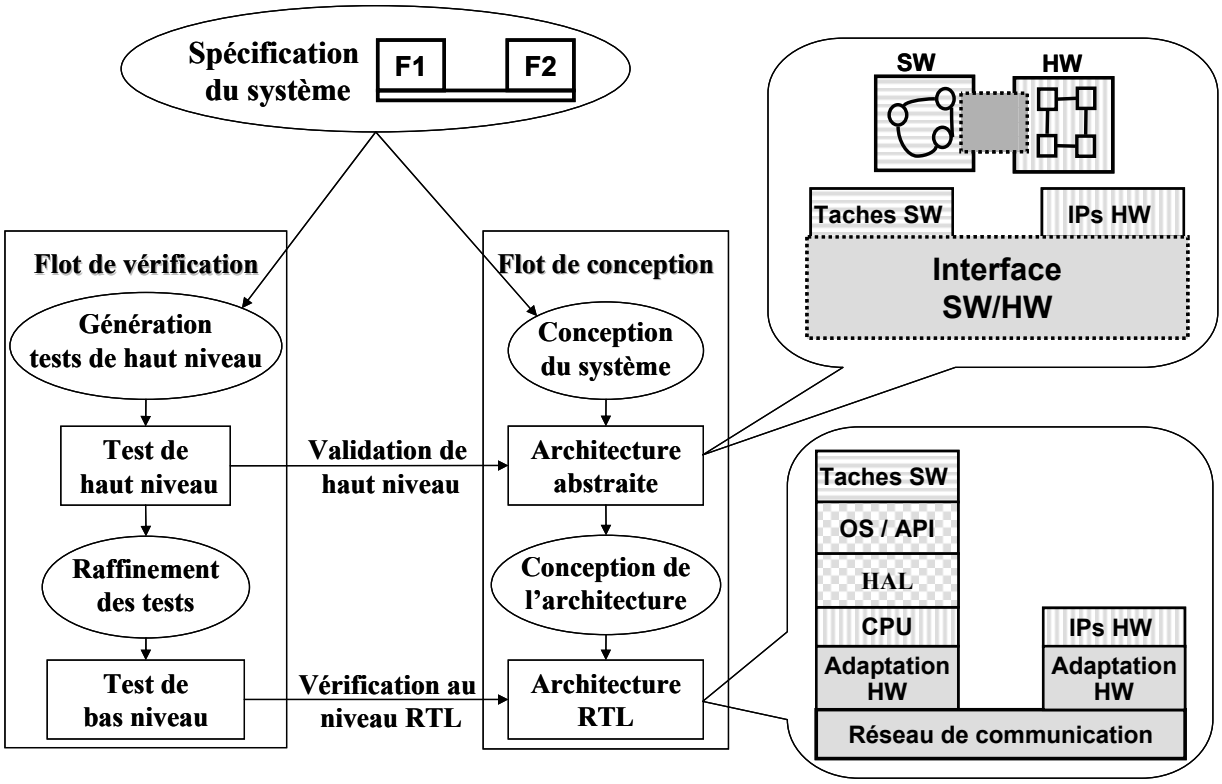


Figure 5 : Représentation de l'architecture d'un système monopuce dans le flot de conception et de vérification idéal

Comme le montre la Figure 5, la conception d'un système débute toujours par la spécification de celui-ci. La spécification peut être réalisée au travers d'un modèle, mais elle est le plus souvent réalisée de manière manuscrite. La figure présente ici une spécification qui est représentée sous la forme d'un système possédant deux fonctionnalités communicant entre elles. La première étape consiste à réaliser le partitionnement logiciel / matériel. La deuxième étape consiste ensuite à réaliser un modèle abstrait de cette architecture. Ce modèle comprend les différentes parties du logiciel de l'application et les différents composants matériels. Ce niveau d'abstraction est essentiellement obtenu par l'abstraction de la communication. Ce modèle abstrait se compose de trois parties : le logiciel, le matériel et une interface logicielle / matérielle. En générale, le modèle abstrait est réalisé au niveau transactionnel. Ce modèle peut par exemple être décrit en TLM (Transaction Level Modeling), qui est une extension de SystemC.

Pour aboutir à la réalisation RTL du système, il est nécessaire de passer par le raffinement de l'interface logicielle / matérielle. Cette réalisation au niveau RTL se compose des nombreuses couches d'adaptation logicielle et matérielle résultant de ce raffinement.

Ainsi, le raffinement de la partie logicielle introduit plusieurs couches logicielles : API (Application Programming Interface), OS (Operating System) et HAL (Hardware Abstaraction Level), mais aussi une couche matérielle : le CPU. Enfin, les dernières couches introduites par le raffinement constituent l'interconnexion qui se compose d'un réseau de communication et le cas échéant d'une couche d'adaptation du composant (HW IP ou CPU) au protocole de communication.

La présentation de la conception des systèmes monopuces met évidence la complexité de ces systèmes. Elle permet également de montrer l'importance croissante de la part du logiciel.

III. Problématiques de la vérification des systèmes monopuces

La présentation de la composition et de la conception des systèmes monopuces permet de mettre en évidence la complexité de ces systèmes. Tout d'abord les systèmes sur puce sont des systèmes de taille considérable. Ils atteignent des tailles de l'ordre de quelques 10 M de portes logiques. De plus cette complexité est également liée au fait que ces systèmes sont des assemblages de composants qui sont souvent hétérogènes. Il en résulte que la vérification est confrontée aux problèmes de comment assurer l'interconnexion des composants et leurs interactions.

La taille et la complexité de ces systèmes rendent leur vérification de plus en plus longue. La vérification représente actuellement environ 70% du temps de conception. Le temps nécessaire à cette tâche impacte ainsi le temps de développement du système et est confronté aux contraintes de temps de mise sur le marché du produit.

Pour la vérification des systèmes monopuces, les hypothèses suivantes sont en général réalisées :

- Les processeurs et les blocs matériels (IP) du système sont supposés validés ;
- Les processeurs du système sont utilisés avec des programmes de test logiciel pour effectuer la vérification du système.

Avec ces hypothèses et par l'étude des méthodes de vérification (celles-ci sont présentées dans le chapitre 2) qui utilisent les processeurs pour générer les stimuli, on

remarque qu'il existe un paradoxe. Ce paradoxe est issu de la différence entre le comportement du logiciel et du matériel. La partie matérielle du système utilise le parallélisme alors que les programmes de test logiciel exécutés sur les processeurs du système sont séquentiels et de bas niveau. Ce paradoxe montre immédiatement la difficulté de concevoir de manière rapide et efficace des programmes qui permettent la synchronisation entre le logiciel et le matériel. Un exemple simple pour illustrer cette difficulté est la gestion des interruptions et des leurs priorités.

Les problèmes liés à la vérification des SoCs peuvent ainsi être résumés en quatre points :

- La complexité et la taille des systèmes du fait de l'assemblage de parties logicielles et matérielles ;
- Le temps important pris pour effectuer la vérification ;
- La nécessité de vérifier le bon fonctionnement de l'interconnexion et des interactions entre les composants, c'est la vérification de l'intégration ;
- Le besoin de vérification tout au long de la conception du système.

IV. Objectifs et Contributions

Cette section présente les objectifs qui ont été définis et les contributions qui ont été réalisées.

IV.A. Objectifs

A partir des différents problèmes rencontrés dans la vérification des systèmes sur puce, plusieurs objectifs ont été définis :

- Maîtriser la complexité d'un système monopuce pour la vérification ;
- Définir une méthode systématique pour la validation globale d'un système monopuce ;
- Définir une méthode de vérification de l'intégration qui :

- Intègre une solution au paradoxe du parallélisme matériel et du logiciel séquentiel.
- S'applique de la spécification jusqu'à la réalisation RTL.

IV.B. Contributions

Les contributions réalisées au cours de ces travaux de recherches sont aux nombres de trois :

- Proposition d'une méthodologie de vérification globale et systématique pour les SoCs ;
- Définition d'un flot de vérification de l'intégration qui se base sur les processeurs du système et sur l'exécution par ceux-ci de programmes de test. Ce flot se décompose en 2 étapes :
  - Génération de programmes de test de haut niveau ;
  - Raffinement des programmes de test de haut niveau.
- L'utilisation de programmes de test de haut niveau basés sur une API et reposant sur un système d'exploitation pour permettre la synchronisation logicielle/matérielle ;
- L'application de la méthode proposée sur un système industriel.

V. Conclusion

Ce premier chapitre a permis de décrire les systèmes sur puce, leur conception et les problèmes rencontrés pour effectuer leur vérification. Il a ainsi été mis en évidence que la vérification constitue une tâche critique dans le processus de conception, en effet environ 70% du temps de conception est affecté à la vérification. De plus, la description des problèmes qui sont rencontrés lors de la vérification permet de définir les objectifs. Ces objectifs ont pour but principal de réduire le temps pris par la vérification.

Chapitre 2 : LA DIVERSITE DES OUTILS ET METHODES DE VERIFICATION

I. INTRODUCTION 28

II. GENERALITES SUR LA VERIFICATION DES SYSTEMES NUMERIQUES 29

II.A. DEFINITIONS 29
II.B. LES DEUX TECHNIQUES DE VERIFICATION 31

III. LES CRITERES D'ANALYSE ET D'EVALUATION DES METHODES DE VERIFICATION 31

III.A. LES CRITERES D'ANALYSE 32
III.A.1. Le type du système sous test 33
III.A.2. La modélisation du système 33
III.A.3. Le type de vecteurs de test 33
III.A.4. Le modèle d'exécution 34
III.B. LES CRITERES D'EVALUATION 34
III.B.1. Taux de couverture 35
III.B.2. Coût d'utilisation et de mise en œuvre 35
III.B.3. Limitations 36

IV. LA VERIFICATION DES BLOCS MATERIELS 36

IV.A. LA VERIFICATION FORMELLE 36
IV.B. LA VERIFICATION DYNAMIQUE 38
I.A.1. L'environnement de test 38
I.A.2. Les langages spécifiques pour la vérification du matériel (HVL : Hardware Verification Language) 41

V. LA VERIFICATION DES PROCESSEURS 42

V.A. LA VERIFICATION DYNAMIQUE BASEE SUR L'UTILISATION D'UN ENVIRONNEMENT DE TEST 42
V.B. LA VERIFICATION DYNAMIQUE BASEE SUR LES PROGRAMMES DE TEST 43

VI. LA VERIFICATION DES SYSTEMES MONOPUCES 46

VI.A. LA VERIFICATION DYNAMIQUE BASEE SUR L'UTILISATION D'UN ENVIRONNEMENT DE TEST 46
VI.B. LA VERIFICATION DYNAMIQUE BASEE SUR LES PROGRAMMES DE TEST 47
VI.B.1. X-Gen 47
VI.B.2. La TLV (Top Level Validation) 50
VI.B.3. La méthode de vérification du PIAGET 53

VII. CONCLUSION 56

I. Introduction

Ce chapitre présente le domaine de la vérification des systèmes numériques. Il permet ainsi de définir les termes employés et de définir le cadre de cette thèse. De ce fait, cette partie présente l'état de l'art sur les outils et méthodes de vérification et plus particulièrement ceux s'appuyant sur la simulation. Ce choix est motivé par le fait que les méthodes formelles ne permettent pas actuellement de réaliser la vérification d'un système monopuce complet. Ces techniques considérées comme complémentaires pour certaines parties de la vérification font l'objet d'une brève description. Elles sont très bien adaptées à la vérification de blocs matériels de petite taille. Les techniques de vérification basées sur la simulation sont ainsi indispensables pour réaliser la vérification de systèmes de taille importante comme le sont les processeurs et les systèmes monopuces. Il existe une très grande diversité de méthodes et outils de vérification basés sur la simulation. Car il existe d'une part différentes approches suivant le type de circuit sous test (bloc matériel, processeur, système monopuce) et d'autre part différentes techniques employées pour stimuler le système (vecteurs de test).

Ce chapitre réalise ainsi un inventaire des méthodes et outils de vérification en tenant compte du type du système sous test. L'état de l'art présenté dans ce chapitre est effectué au travers de critères d'analyse et d'évaluation définis pour les méthodes de vérification qui sont :

- **Critères d'analyse** : le type de système sous test, la modélisation du système, le type de vecteurs de test et le modèle d'exécution ;
- **Critères d'évaluation** : le taux de couverture, le coût d'utilisation et de mise en œuvre et les limitations.

La première section introduit la vérification des systèmes numériques. La seconde section définit les critères d'analyse et d'évaluation utilisés pour caractériser les outils et méthodes de vérification. Les sections trois, quatre et cinq présentent respectivement l'état de l'art pour la vérification des blocs matériels, des processeurs et des systèmes monopuces. Enfin, la conclusion de ce chapitre donne un résumé des différents outils et méthode de vérification au travers des différents critères définis dans ce chapitre.

II. Généralités sur la vérification des systèmes numériques

Cette section introduit la vérification des systèmes numériques et définit les différentes étapes de vérification qu'il est nécessaire d'effectuer lors de la conception d'un tel système.

II.A. Définitions

Dans la vérification, on distingue différentes terminologies suivant le type d'utilisation. En effet, la méthodologie et la finalité sont différentes. Ainsi lors de la conception d'un circuit, il y a plusieurs phases. Cela débute toujours par la spécification du circuit. Puis, cette spécification est traduite pour donner lieu à une série de réalisations pour finalement aboutir à la réalisation physique du circuit, ce flot est présenté sur la Figure 6.

On peut ainsi classer les tâches de vérification en trois catégories correspondant aux différentes phases de conception :

- La vérification d'une réalisation par rapport à la spécification initiale (manuscrite), dans ce cas on parlera de **validation**.
- La vérification d'un raffinement, ce qui correspond à vérifier la conformité de la nouvelle réalisation par rapport au modèle de référence. Dans ce cas on parlera de **vérification** ou de **déboguage**.
- La vérification de la réalisation physique, dans ce cas on parlera de **test**.

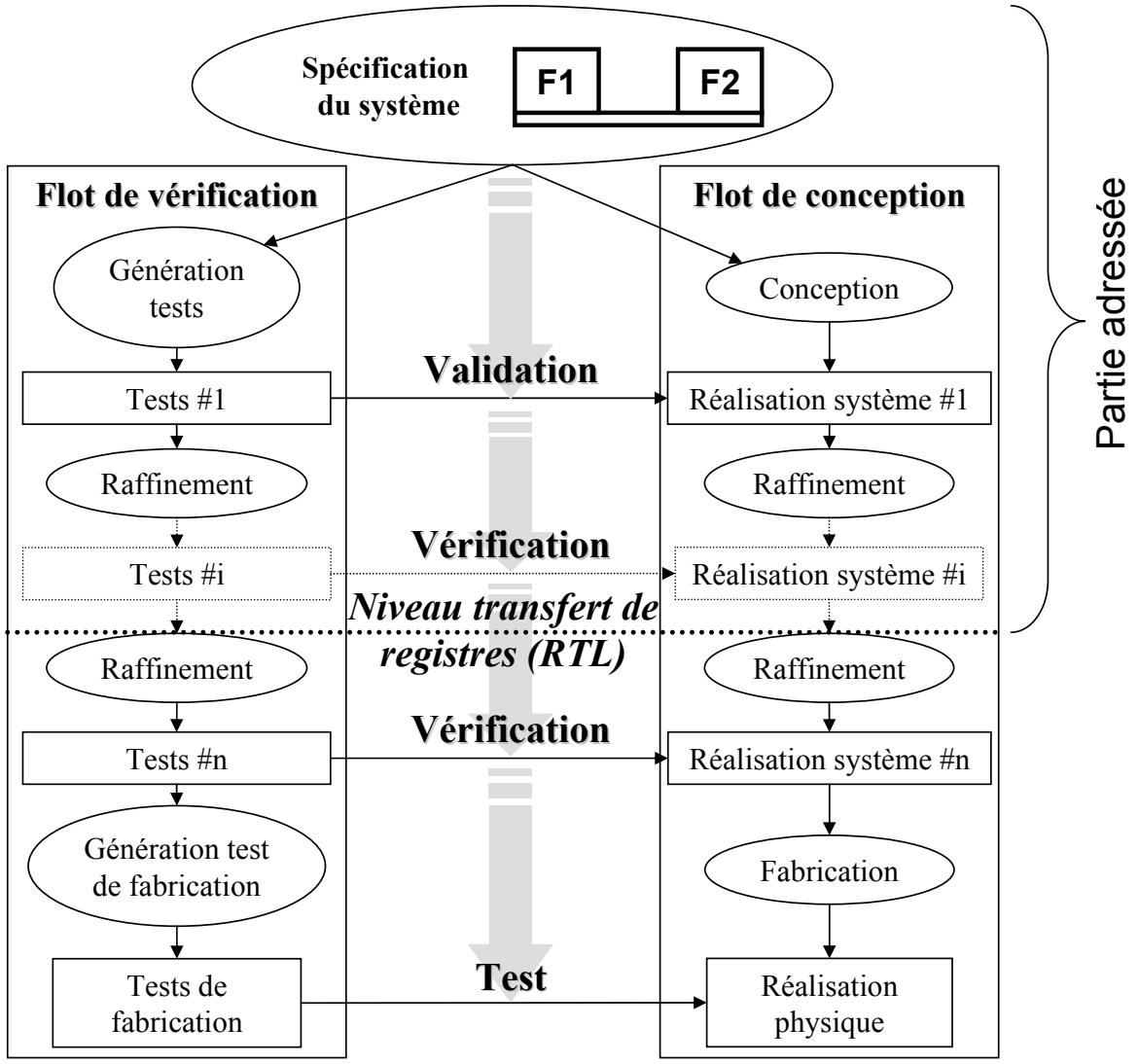


Figure 6 : Présentation générale du flot de conception et de vérification des systèmes électroniques

Dans la suite de ce mémoire, nous nous intéressons plus particulièrement à la validation et à la vérification, le test n'étant pas abordé. De plus, parmi les différents niveaux d'abstraction des réalisations, la vérification du raffinement entre le niveau RTL et la réalisation physique (layout) est bien maîtrisée. Aussi, dans la suite de ce document, les méthodes de vérification présentées ciblent plus précisément la vérification d'un système, de sa spécification au niveau RTL.

II.B. Les deux techniques de vérification

Dans les techniques de vérification, on distingue deux méthodes. La distinction se fait sur le modèle d'exécution. Ces modèles sont :

- Le **modèle formel** : la vérification formelle peut elle-même être décomposée en trois méthodes :
    - La vérification d'équivalence permettant de démontrer l'équivalence entre deux modèles du même système qui peuvent être au même niveau d'abstraction ou à des niveaux différents.
    - La vérification de propriétés permettant de vérifier si le circuit respecte les propriétés qui ont été définies à partir de la spécification du système.
    - La preuve de théorème démontrant mathématiquement que le système réalise une fonction donnée.
- Le **modèle dynamique ou par simulation**. Dans la vérification dynamique, la vérification est effectuée par simulation. Dans la méthode dynamique on peut différencier deux types d'approche. La première consiste à injecter les vecteurs de test à partir de l'environnement. La seconde consiste à utiliser les composants programmables du système pour exécuter des programmes de test logiciel, ce sont ces programmes qui représentent les vecteurs de test.

III. Les critères d'analyse et d'évaluation des méthodes de vérification

Cette section a pour but de décrire les critères d'analyse et d'évaluation qui sont utilisés pour caractériser les outils et méthodes de vérification. Etant donné que dans ce mémoire nous ciblons la vérification des systèmes monopuces, ces critères sont plus particulièrement destinés à l'étude des méthodes dynamiques. Un critère important est celui du modèle d'exécution : statique ou dynamique. Ainsi, dans le cas de la vérification statique, le critère du vecteur de test n'est pas significatif, ce concept n'étant pas présent.

III.A. Les critères d'analyse

Cette section présente les critères qui permettent d'analyser les différentes méthodes de vérification existantes. Au travers du flot de conception et de vérification, la Figure 7 présente les principaux critères retenus pour l'analyse de ces méthodes. Chaque critère fait ainsi référence à un élément de base de la méthode de vérification et cela quelle que soit la phase de vérification. Ces quatre critères permettent ainsi de caractériser une méthode de vérification dans sa globalité :

- Le type de système sous test,
- La modélisation du système (pour permettre la génération des tests),
- Le type des vecteurs de test utilisés,
- Le modèle d'exécution de la méthode de vérification.

On remarquera que la figure tient compte de la possibilité d'avoir plusieurs niveaux d'abstraction.

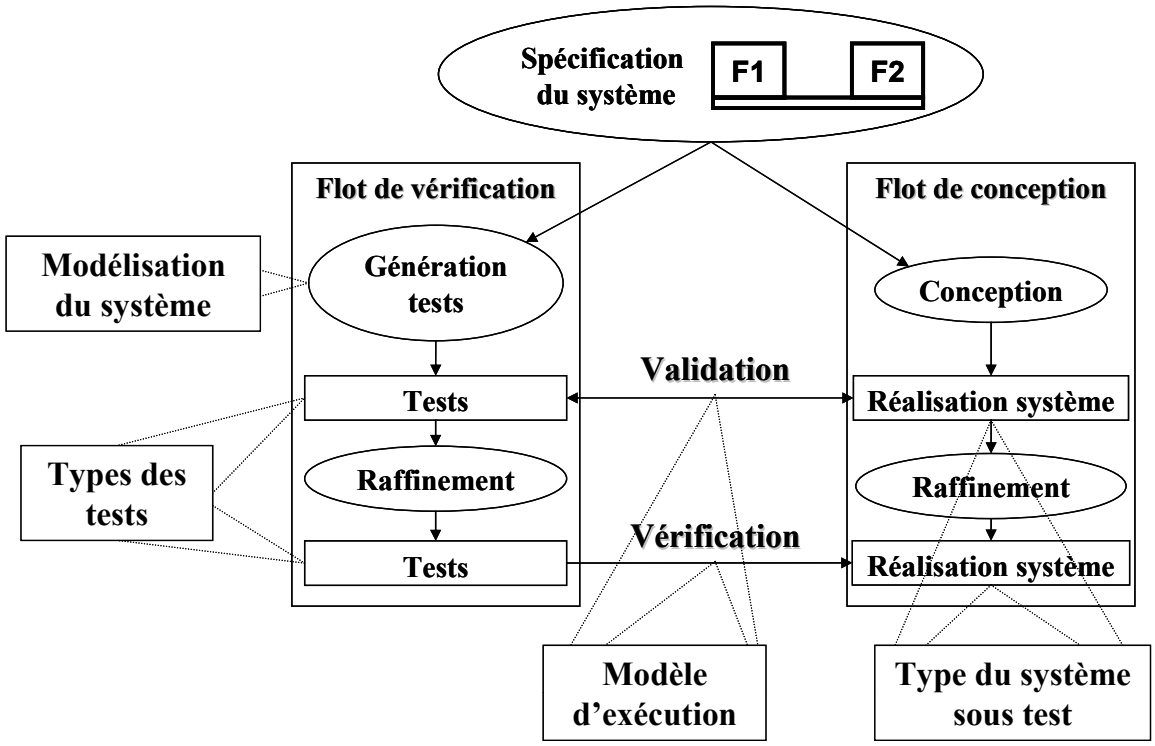


Figure 7 : Les critères d'analyse des méthodes de vérification

III.A.1. Le type du système sous test

La méthode de vérification dépend du type de circuit pouvant être adressé par la méthode. On définit trois types de circuit [41] :

- Bloc matériel (HW IP),
- Processeur,
- Système monopuce (SoC), Système multiprocesseur monopuce (MPSoC), Réseaux sur puce (NoC)).

III.A.2. La modélisation du système

La modélisation du système est la représentation utilisée pour permettre la définition du système dans le but d'en réaliser la vérification. Des exemples concrets de modélisation du système sont donnés lors de la présentation de méthodes de vérification. Parmi les modélisations rencontrées dans la vérification on peut citer par exemple les machines d'état finis.

III.A.3. Le type de vecteurs de test

Le type des vecteurs de test est un critère utilisé pour les méthodes dynamiques. En effet, les méthodes formelles n'utilisent pas de vecteurs de test et s'attachent à être exhaustives. Ainsi, pour la vérification dynamique, les vecteurs de test sont les stimuli appliqués au système lors de la simulation. On peut distinguer deux types de vecteurs de test : les stimuli générés par un environnement de test (Figure 8 - a) et les programmes de test exécutés par les composants programmables du système (Figure 8 - b). Chacun de ces deux types peut être modélisé à différents niveaux d'abstraction.

Dans le cas d'un environnement de test, les vecteurs de test peuvent être décrits à différents niveaux. Ils sont soit décrits directement au niveau RTL ou soit décrits à un niveau d'abstraction plus élevé, par exemple le niveau transactionnel.

Pour les vecteurs de test réalisés par des programmes de test, on peut distinguer également deux types suivant le niveau d'abstraction. On appellera programmes de test de bas

niveau, ceux réalisés aux niveaux ISA ou HAL et on appellera programmes de test de haut niveau ceux réalisés au niveau OS/API. Plus de détails sur la distinction entre les programmes de test de haut niveau et de bas niveau pourront être trouvés dans le chapitre 4.

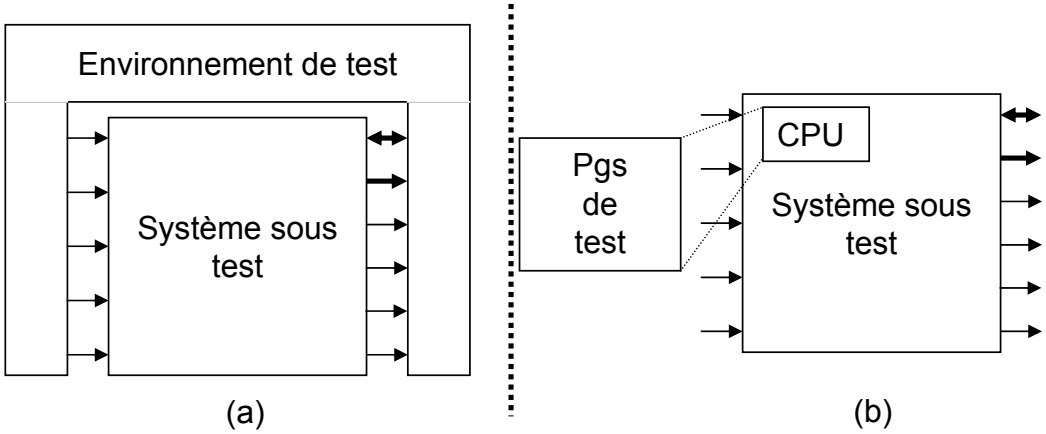


Figure 8 : Représentation pour la vérification dynamique, des deux types d'élément générant les stimuli : (a) Environnement de test ; (b) Programmes de test

III.A.4. Le modèle d'exécution

Le modèle d'exécution définit comment se déroule l'interaction entre les vecteurs de test et le modèle du système. Le modèle d'exécution peut ainsi être vu comme étant la manière avec laquelle est réalisée la vérification. On distingue deux modèles d'exécution qui ont été définis auparavant :

- Le modèle formel ou statique ;
- Le modèle dynamique.

III.B. Les critères d'évaluation

L'efficacité d'une méthode de vérification peut être mesurée en fonction :

- Du taux de couverture de la méthode,
- Du coût d'utilisation et de mise en œuvre,
- Des limitations.

III.B.1. Taux de couverture

Le taux de couverture permet de juger la pertinence des vecteurs de test employés donc la méthode. Dans la définition du taux de couverture, on distingue trois types :

- Le taux de couverture du code mesure le niveau avec lequel le code HDL du circuit a été exercé (par ex : en terme d'instructions, de branchements…) ;
- Le taux de couverture sur une machine d'état en terme d'états et de transitions entre les différents états du système ;
- Le taux de couverture fonctionnel permet de définir un ensemble de fonctionnalités du circuit que l'on souhaite vérifier (ex : protocoles, cas d'erreur, propriétés temporelles…).

D'autre part, on remarquera que la métrique du taux de couverture du code HDL est obtenue automatiquement par l'intermédiaire d'outils spécifiques (vNavigator) [15] ou des simulateurs (VCS [16], NC-SIM [17]). Il est déterminé par instrumentation du code HDL.

En ce qui concerne le taux de couverture d'une machine d'état, cette fonctionnalité peut être également offerte par les outils cités précédemment, mais sous certaines contraintes de description HDL du circuit.

Le taux de couverture fonctionnelle est défini par la personne réalisant la vérification. Cette métrique est présente dans des outils de vérification qui seront évoqués par la suite, mais elle peut également être réalisée par l'intermédiaire d'assertions. Ces techniques permettent également d'effectuer un taux de couverture d'une machine d'état. Cependant, la mise en place d'une métrique sur le taux de couverture fonctionnelle constitue un travail fastidieux qu'il faut maintenir. Cependant, il existe dans la littérature des travaux permettant de mettre en œuvre un taux de couverture fonctionnelle à partir d'une spécification exécutable du système [19][20].

III.B.2. Coût d'utilisation et de mise en œuvre

Le critère du coût d'utilisation permet d'évaluer l'effort devant être mis en œuvre pour mettre en place la méthode de vérification. Ce critère repose sur le temps lié à la construction

des différents modèles et sur la complexité d'utilisation de la méthode. Ce coût d'utilisation sera également d'autant plus faible que la méthode est automatisée.

III.B.3. Limitations

Ces différentes méthodes présentent un certain nombre de limitations qui peuvent être du type :

- Capacité de déboguage ;
- Taille du circuit adressable ;
- Contraintes d'utilisation (maintenance, réutilisation) ;
- Ressources nécessaires (puissance de calcul)…

IV. La vérification des blocs matériels

Cette section décrit les différents outils et les différentes méthodes employés pour réaliser la vérification des blocs matériels. Pour ce type de circuit, il existe un grand nombre de méthodes formelles et dynamiques que nous présenterons ici. La partie sur les méthodes dynamiques présente la technique qui consiste à utiliser un environnement de test pour réaliser la vérification. Cette technique ne cible pas exclusivement les blocs matériels, elle est également utilisée pour les processeurs et les systèmes sur puce. Elle fait l'objet d'une description détaillée uniquement dans cette section.

IV.A. La vérification formelle

La vérification formelle peut se décomposer en trois catégories [1] qui sont :

- **La vérification d'équivalence** : elle consiste à montrer l'équivalence entre deux réalisations d'un circuit au même niveau d'abstraction ou entre le circuit raffiné et le circuit décrit à un niveau d'abstraction plus élevé. Cette méthode est largement utilisée pour vérifier l'équivalence entre le résultat de la synthèse d'un circuit avec son modèle RTL. Formality [2] (Synopsys) est un exemple d'outil proposant cette

méthode de vérification d'équivalence. Ce type de vérification se base sur la modélisation des circuits combinatoires sous la forme de diagramme de décision binaire (BDD : Binary Decision Diagram). L'équivalence est démontrée par comparaison des BDDs de chacun des deux modèles. Pour les circuits séquentiels, l'équivalence est démontrée par comparaison des machines d'états finis (FSM : Finite State Machine).

- **La vérification de propriétés** : cette méthode se base sur la modélisation du système sous forme de machine d'états finis ainsi que d'un ensemble de propriétés décrites dans un langage logico-temporel que doit respecter le circuit par exemple en utilisant le langage PSL(Property Specification Language) [12]. Ce langage repose ainsi principalement sur le langage SUGAR [11], développé originellement par IBM dans le cadre de la vérification formelle pour la démonstration de propriétés. Vérifier qu'un circuit respecte une certaine propriété consiste à montrer que quel que soit l'ensemble des combinaisons possibles en entrée du circuit et quel que soit le chemin d'exécution, la propriété est respectée. Cette opération est réalisée en parcourant l'espace d'états de la machine d'états finis et en déterminant si la propriété est vérifiée ou non. Cependant, cette méthode souffre de limitations liées à la taille de l'espace d'états du circuit qui croit exponentiellement avec le nombre de variables d'état. Ainsi, les outils actuels permettent d'adresser des circuits ayant quelques milliers de variables d'états. Parmi les outils commerciaux proposant la vérification de propriétés on peut citer RuleBase [3] (IBM), Magelan [4] (Synopsys) et Insicive Formal Verifier (Cadence) [8]. Un exemple de vérification formelle utilisant conjointement la vérification de théorèmes ainsi que la vérification de propriétés est donnée dans [21].
- **La démonstration de théorème** : cette méthode présente le problème de la vérification sous la forme d'un théorème. Ensuite un ensemble d'axiomes est utilisé pour construire la preuve du théorème. La preuve de théorème nécessite souvent la démonstration de résultats intermédiaires et n'est pas complètement automatisée. La démonstration de théorèmes est actuellement une tâche très ardue et réservée aux experts. On peut également noter que la preuve de théorèmes est appliquée à la vérification de blocs matériels comme les unités de calcul en virgule flottante embarquées dans les processeurs [21].

Ces différentes méthodes utilisent toutes un modèle formel du circuit (BDD, FSM, théorème). Enfin en cas de bogue, ces méthodes déterminent en général un scénario permettant de reproduire le bogue en simulation. Ces méthodes ne nécessitent pas l'utilisation de vecteur de test afin d'achever la vérification, elles sont exhaustives. Malheureusement, la complexité et la taille des systèmes actuels rendent ces méthodes inapplicables à tous les types de circuit et particulièrement les systèmes sur puce. Ces méthodes sont cependant très efficaces pour les blocs matériels de taille moyenne.

IV.B. La vérification dynamique

La vérification dynamique des blocs matériels (IP HW) s'effectue principalement par l'utilisation d'un environnement de test. Cet environnement permet de vérifier les sorties et de générer les stimuli d'entrée adéquats. Parmi les outils et méthodes basés sur cette technique on peut citer : Vera (Synopsys) [5], Specman (Cadence, ex-Verisity) [6] ou SystemC Verification Library [7], cette liste n'étant pas exhaustive. On remarquera qu'il est possible de réaliser l'environnement de test en utilisant un langage HDL. La Figure 9 donne la représentation du flot de vérification basé sur l'environnement de test. On remarquera que cette méthode n'est pas uniquement destinée à la vérification des blocs matériels, mais couvre l'ensemble des systèmes numériques considérés ici. Cette section décrit, dans un premier temps, cette technique de vérification puis, les spécificités des outils. Les langages sont discutés dans un deuxième temps.

I.A.1. L'ENVIRONNEMENT DE TEST

L'environnement a comme objet d'exercer le circuit sous test par l'intermédiaire des entrées/sorties de celui-ci. La communication entre l'environnement de test et le circuit peut être décrite à un niveau d'abstraction plus élevé que celui des signaux. En général, elle est définie au niveau transactionnel, le but étant de maîtriser la complexité. Par exemple, si l'on considère un composant utilisant un protocole de communication donnée, la définition des stimuli ne s'effectue pas au niveau des signaux de l'interface par un mécanisme produisant une séquence de signaux d'entrée, mais au niveau des transactions. Dans le cas d'un protocole

« poignée de mains », une transaction correspond par exemple à l'écriture d'une donnée. C'est l'environnement qui effectue le raffinement de ces transactions en respectant le protocole. La modélisation d'un environnement de test pour achever la vérification peut ainsi se représenter sous la forme d'un système producteur – consommateur.

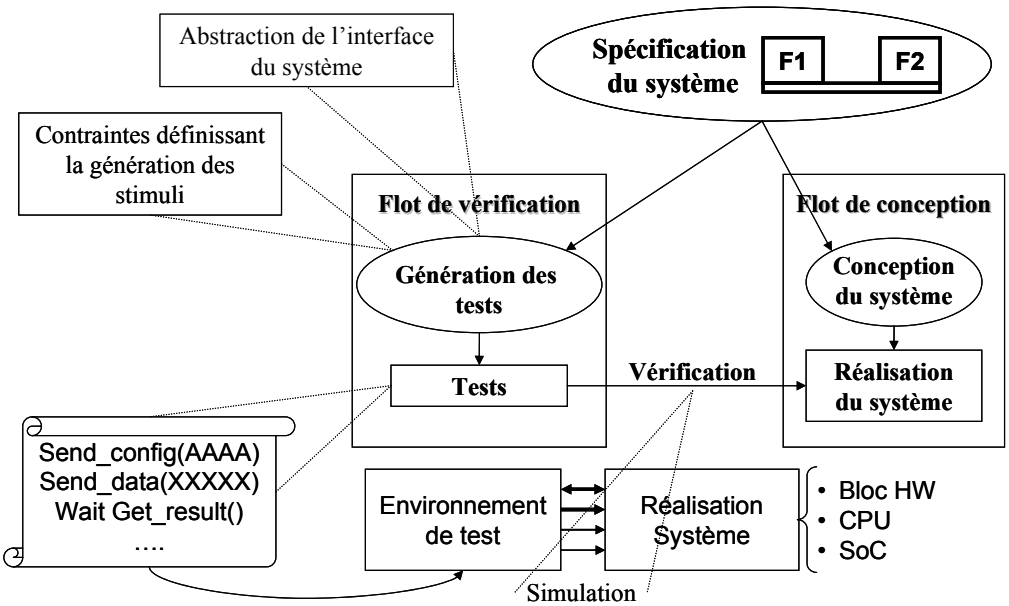


Figure 9 : Représentation du flot de vérification basée sur l'utilisation d'un environnement de test dans le flot de vérification et conception des systèmes électroniques

Pour réaliser la description de l'environnement de test, chaque outil dispose d'un langage spécifiquement dédié à la vérification. Ces langages font partie des langages de vérification du matériel (HVL : Hardware Verification Language). Ils permettent la réalisation d'environnements de test complexes grâce aux primitives qu'ils offrent (cf section suivante). Ils intègrent également des primitives permettant la définition d'assertions. Cependant, ces langages ont été conçus dans le but d'effectuer la vérification de réalisations matérielles. Ces approches sont de ce fait limitées dans le cas de la vérification des systèmes monopuces, vérification qui nécessite aussi la prise en compte de la partie logicielle du système.

Les différentes fonctionnalités que l'on peut trouver dans un environnement de test sont au nombre de trois :

- **La génération des stimuli**. Afin d'exercer le circuit sous test, l'environnement dispose d'un mécanisme de génération de stimuli qui peut se faire de manière dirigée, contrainte ou aléatoire [23]. La génération des stimuli de manière dirigée se fait

encore dans certain cas manuellement, alors que la génération aléatoire est, dans la majorité des cas, basée sur la définition de contraintes résolues par un solveur générant aléatoirement un ensemble de transactions respectant ces contraintes. Les transactions sont ensuite raffinées par l'environnement de test pour être appliquées au système.

- **La vérification automatique des sorties**. Elle permet de vérifier automatiquement les résultats produits par le circuit sous test. Cette vérification est rendue possible par l'utilisation de deux techniques distinctes. La première consiste à mémoriser les sorties du circuit sous test et à les comparer à des résultats prédéterminés soit par l'intermédiaire d'un modèle de référence soit par le concepteur. La seconde consiste à créer des moniteurs qui scrutent les valeurs des signaux d'entrées et qui calculent les valeurs de sorties afin de les comparer aux sorties du circuit sous test [21].

- **La collecte d'informations sur le taux de couverture**. La plupart des outils basés sur l'environnement de test offrent une fonctionnalité intéressante pour améliorer la qualité de la vérification à savoir le recueil d'informations sur le taux de couverture. Suivant les cas, cette métrique peut être obtenue directement par l'intermediaire de l'outil ou bien par l'utilisation d'autres utilitaires specifiques. Les taux de couverture employés sont :

  - le taux de couverture du code HDL,
  - le taux de couverture des transitions et des états de la machine d'état finis du circuit,
  - le taux de couverture fonctionnel définissant des fonctionnalités du circuit devant être exercées.

  Il est ainsi possible d'obtenir un bon taux de couverture quel que soit le type considéré.

La méthode dynamique basée sur l'environnement de test nécessite suivant la complexité du circuit sous test un effort plus ou moins important pour l'implémentation de l'environnement. Ainsi, lorsque le bloc matériel se décompose lui-même en sous blocs, la méthodologie consiste alors à réaliser, pour chacun des blocs et pour le système, des environnements de test appropriés. Un exemple de vérification de ce type est donné dans [24]. L'approche utilisée dans ce cas est : « diviser pour conquérir ». Elle s'applique de la même manière pour les processeurs et les systèmes monopuces.

La vérification de circuits intégrants plusieurs blocs matériels emploie plusieurs environnements de test qu'il est alors nécessaire de maintenir en conformité avec les différents blocs. Aussi, lors de changements architecturaux importants du système, l'adaptation des environnements de test peut présenter un effort important. Cependant, en décomposant ainsi la vérification, il est possible de réutiliser certaines parties de l'environnement des composants comme les moniteurs dans l'environnement du système. Cela permet pendant le test du système d'obtenir une meilleure observabilité et de vérifier le comportement des différents blocs. L'utilisation de cette approche offre la possibilité de recueillir des informations sur le taux de couverture de chaque bloc du circuit et de faciliter le déboguage.

I.A.2. LES LANGAGES SPECIFIQUES POUR LA VERIFICATION DU MATERIEL (HVL : HARDWARE VERIFICATION LANGUAGE)

Les outils et méthodes basés sur l'environnement de test exploitent des langages spécifiques. Les plus couramment utilisés dans l'industrie sont le langage *e* et OpenVera [14]. Ces langages reposent sur un modèle orienté objet qui permet de réaliser l'abstraction et de faciliter la réutilisation. Ils offrent les constructions standard de langage de programmation comme le C++, et comme ceux des langages de description du matériel (HDL). Ils possèdent en outre des primitives spécifiques qui permettent la définition des contraintes. Enfin, ils intègrent un ensemble de primitives exclusivement destinées à la vérification : taux de couverture et vérification de propriétés temporelles (assertions).

Pour la spécification d'assertion, il existe de nombreux autres langages dédiés comme par exemple le langage PSL (Properties Specification Language) [12] qui est également utilisé en vérification formelle. Ces langages ont par ailleurs un pouvoir d'expression plus important permettant la définition de propriétés plus complexes.

On remarque ainsi qu'il existe de nombreux langages utilisés dans le cadre de la vérification. Au cours de ces dernières années, on assiste, de la part des fournisseurs d'outils de conception, à l'intégration de toutes ces fonctionnalités dans un même outil. Cette convergence s'est faite en même temps qu'ont été développés des nouveaux langages comme SystemVerilog [13]. Ce nouveau langage standardisé regroupe dans un même ensemble les

langages Verilog, OpenVera et OVA (OpenVera Assertion). Un exemple d'application des assertions pour la vérification est donné dans [25].

L'intégration de toutes ces fonctionnalités dans un même environnement permet d'utiliser différentes approches pour la vérification. Les plates-formes Discovery [9] et Incisive [10] permettent l'interaction entre la méthode formelle de démonstration de propriétés et la méthode dynamique par simulation. Il est ainsi possible de réutiliser les propriétés (assertions) pour mesurer le taux de couverture fonctionnelle ou encore pour définir des contraintes.

V. La vérification des processeurs

Les principaux outils et méthodes de vérification pour les processeurs présentés dans la littérature sont basés sur l'utilisation de programmes de test. Cependant, les outils commerciaux basés sur l'utilisation d'un environnement de test sont également utilisés comme cela a été précisé dans la section précédente. La vérification des processeurs est reconnue pour être une tâche très longue et fastidieuse. Les processeurs sont des systèmes dont la complexité et la taille ne cessent de croître, ainsi comme le décrit [27] et [28] les bogues sont d'origines très diverses et leur nombre augmente avec la complexité et la taille des systèmes considérés.

V.A. La vérification dynamique basée sur l'utilisation d'un environnement de test

La vérification des processeurs par l'utilisation d'un environnement de test ne fait pas l'objet de publication. Cette méthode est comparable à celle présentée dans la section décrivant la vérification des blocs matériels, un processeur étant en effet un assemblage de composants matériels ayant des interfaces bien définies. La vérification est donc réalisée pour chacun des blocs puis l'ensemble du système est testé de la même manière qu'un assemblage de blocs matériels.

Cependant, étant donnée la présence spécifique d'un jeu d'instruction, une partie logicielle est nécessaire pour réaliser la vérification (programmes de test). Tout comme pour

la méthode basée sur la génération de programmes de test, l'emploi d'un environnement de test nécessite l'utilisation des programmes de test. Ils peuvent être générés de manière aléatoire ou dirigée, ou encore écrits manuellement. Les outils commerciaux comme Specman ou Vera permettent également la modélisation du jeu d'instruction dans leur langage HVL. Ensuite, la définition de contraintes autorise l'utilisation du même mécanisme de génération que celui utilisé pour les transactions. Il est ainsi possible de générer des programmes de test de bas niveau. Ces programmes (flux d'instructions) sont chargés en mémoire afin d'être exécutés. Les résultats obtenus sont ensuite comparés aux résultats produits par le modèle de référence.

Toutes les fonctionnalités et les limitations liées à la méthode basée sur l'environnement de test pour effectuer la vérification des processeurs sont les mêmes que celles des blocs matériels.

V.B. La vérification dynamique basée sur les programmes de test

La vérification dynamique des processeurs basée sur l'utilisation de programmes de test nécessite toujours de réaliser la vérification séparée de chacun des blocs constituant le processeur. Parmi ces méthodes, on trouve principalement dans la littérature l'outil Genesys d'IBM qui a évolué en Genesys-Pro. Cet outil est présenté dans [29], [30] [31] et la Figure 10 donne une représentation de son flot toujours suivant le modèle de conception et de vérification idéale. Il existe d'autres générateurs de programme de test ciblant la vérification des processeurs. Un autre exemple de générateur est RAVEN [40].

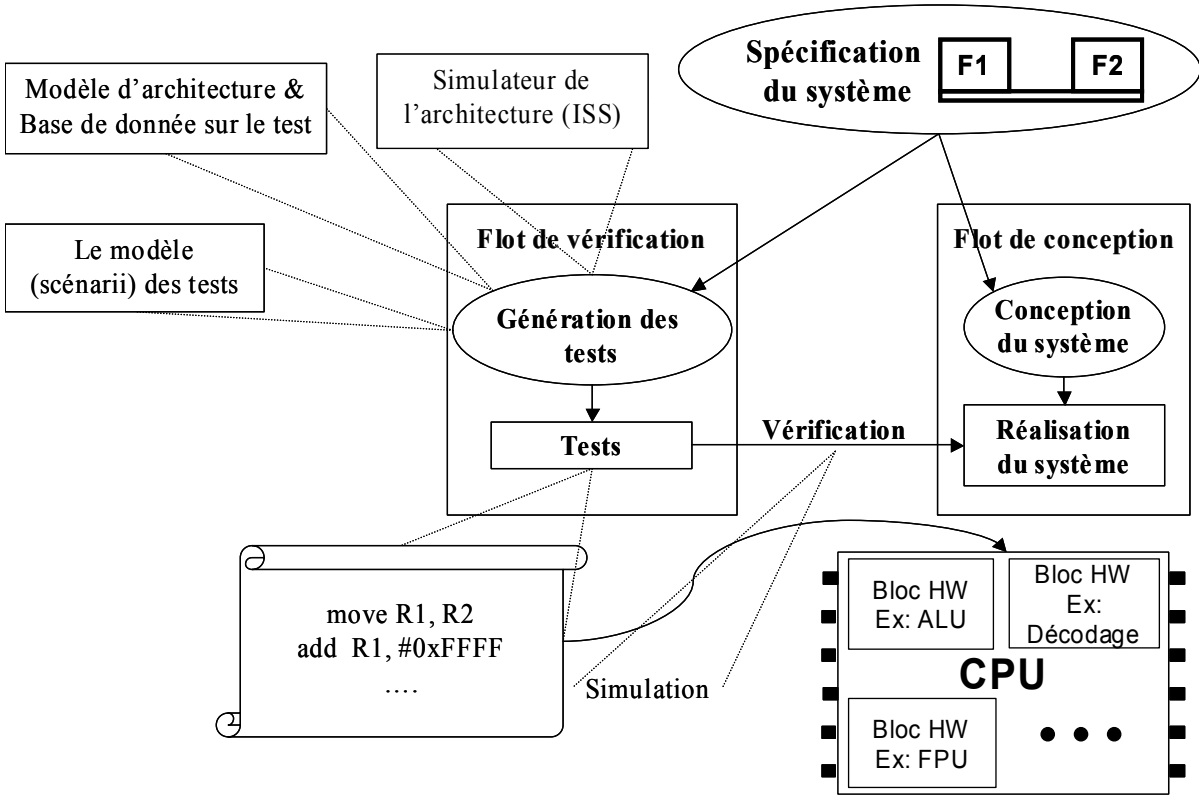


Figure 10 : Représentation du flot Genesys-Pro dans le flot de vérification et de conception des systèmes électroniques

Comme le montre la Figure 10, l'outil Genesys-Pro se décompose en quatre parties :

- **La modélisation de l'architecture et la base de donnée sur le test :** la modélisation du processeur correspond à la description du jeu d'instruction. Pour chaque instruction les ressources et les types de données associés sont décrits. Chaque instruction faisant partie du jeu d'instruction du processeur est ainsi modélisée sous forme d'arbre contenant le format (adressage, types de donnée) et la sémantique de l'instruction. Cette modélisation intègre des connaissances liées à l'expérience de l'utilisateur (base de donnée sur le test). Elles permettent par la suite lors de la génération des programmes de test de biaiser les valeurs pour atteindre des cas critiques pour la vérification. Ainsi la génération aléatoire devient dirigée et cela permet de cibler des cas bien précis difficilement atteignables de manière purement aléatoire. La spécification des tests est réalisée par l'intermédiaire de l'interface utilisateur. Cette modélisation permet au générateur de programmes de test d'être indépendant du modèle.

- **Le simulateur de l'architecture** correspond à un modèle de haut niveau du processeur, c'est un simulateur du jeu d'instruction (ISS : Instruction Set Simulator). Son but est d'exécuter les programmes de test afin de prédire le résultat des opérations.
- **Les modèles (scénarii) de test** permettent de diriger la génération des tests. La génération des programmes de test s'effectue par un langage qui permet de spécifier des scénarii de test. Ce langage se compose de quatre éléments qui sont :
    - Les instructions de base qui représentent les opérations que le processeur doit réaliser pendant le test, par exemple les instructions Load, Store ;
    - Les instructions de séquencement et de contrôle de la génération des tests, de type séquentiel, boucle ;
    - Les constructions des langages de programmation standard comme la définition, l'affectation de variables ;
    - Les instructions permettant de définir les contraintes
- **Les tests** ainsi obtenus représentent une suite d'instructions formant le programme de test. Les programmes générés le sont au niveau assembleur.

Ainsi à partir du modèle du processeur, des tests et du simulateur d'architecture, le générateur va être capable de produire des programmes de test permettant la vérification du processeur. Le mécanisme de génération est principalement basé sur un solveur de contraintes permettant de déterminer les valeurs des paramètres des instructions à partir du modèle du système et du scénario de test. Cette structure permet d'utiliser le même générateur quel que soit le type de processeur, indépendamment de son architecture. Cet outil fait également l'objet de nombreuses applications dans la vérification de différents processeurs. Des exemples de ces applications sont donnés dans [30], [32], [33].

De nombreux autres travaux ont également été menés pour proposer des méthodes et outils permettant d'adresser la vérification complète des processeurs. Ainsi les méthodes proposées dans [34], [35] utilisent des programmes de test et décrivent leurs méthodes qui sont principalement basées sur la génération automatique. D'autre part, [36] présente une étude de la vérification d'un processeur par l'utilisation d'applications réelles, comme les systèmes d'exploitation. Enfin de nombreux travaux ont également été menés pour adresser des points spécifiques et complexes de la vérification des processeurs. Ces études sont

présentées dans [37] et [38], elles adressent plus particulièrement la vérification des pipelines dans les processeurs. Enfin, [39] propose une méthode de génération dynamique des programmes, c'est-à-dire que la génération des instructions se fait en connaissant l'état dans lequel se trouve le processeur. Cela permet ainsi la génération de programmes adressant des cas spécifiques, mais aussi d'éviter des cas non encore supportés par le système.

L'avantage de ce type de méthode basée sur l'utilisation de programmes de test est la simplicité de la modélisation qui est faite du processeur. Cette modélisation étant indépendante de son architecture, elle ne nécessite que la description précise du jeu d'instruction.

VI. La vérification des systèmes monopuces

Les systèmes sur puce présentent des similarités avec les processeurs. Les méthodes de vérification basées sur l'utilisation d'un environnement de test ne seront donc pas détaillées dans cette section. Seront cependant exposées les méthodes spécifiques à la vérification des systèmes monopuces, basées sur l'utilisation de programmes de tests.

VI.A. La vérification dynamique basée sur l'utilisation d'un environnement de test

La vérification des systèmes monopuces basée sur l'utilisation d'un environnement de test possède les mêmes caractéristiques et limitations que dans le cas des processeurs et des blocs matériels. Une méthodologie permettant de réaliser la vérification de tels systèmes est donnée dans [42]. Proposée par Verisity, elle se base sur l'utilisation d'environnements de test.

D'autre part, une caractéristique de ces systèmes qui est la réutilisation d'IPs, a fait émerger l'idée de réutilisation des composants de vérification. Ils sont nommés eVC (e Verification Component) pour Verisity (Cadence) et vIP (verification IP) pour Synopsys. Ils sont particulièrement bien adaptés aux composants ayant des interfaces standard (protocole de bus…). Cependant, avec la complexité croissante de tels systèmes, l'environnement de test est

difficile à mettre en place et à maintenir. A noter que cette méthode n'intègre pas la partie logicielle du système.

VI.B. La vérification dynamique basée sur les programmes de test

Pour la vérification dynamique des systèmes monopuces, il existe deux méthodes se basant sur l'utilisation de programmes de test : X-Gen (IBM) [43] issue des recherches antérieures comme le montre [45] et la TLV (Top Level Validation d'Esterel Technologies) [46]. Ces deux méthodes, décrites ci-dessous, adressent la vérification des interactions entre composants du système et non pas le comportement des composants. Ces méthodes supposent que les composants sont vérifiés au préalable. En complément de l'étude bibliographique, une troisième méthode de vérification est présentée. Elle décrit la méthode de vérification employée et conçue par l'équipe TV pour effectuer la vérification du système PIAGET. Certaines parties de la vérification seront décrites plus précisément dans le chapitre 3.

Outre les méthodes basées sur X-Gen et la TLV, ils existent d'autres travaux qui font état de méthodes de vérification des systèmes sur puce. Ainsi on peut trouver dans [47] une méthode adaptant l'outil Genesys pour réaliser la vérification d'un système multiprocesseur. Des exemples de méthodes de vérification appliquées dans l'industrie sur des systèmes monopuces sont également présentés dans [48] et [49].

Il est encore fréquent, dans l'industrie, que la vérification de l'intégration soit effectuée par l'intermédiaire de programmes de test écrits manuellement [48] ou générés par un programme spécifique paramétrant simplement des configurations différentes. Cette méthode se base principalement sur la définition d'un plan de test décrivant les interactions devant être vérifiées. Cette méthode est très longue et fastidieuse car manuelle et parce qu'elle implique d'avoir une connaissance suffisante du système pour l'écriture des programmes afin de permettre la vérification des cas limites.

VI.B.1. X-Gen

La Figure 11 présente le flot de la méthode de vérification s'appuyant sur l'outil X-Gen. Il se base sur un modèle spécifique du système et rend ainsi le générateur indépendant de

celui-ci. D'autre part, cette méthode possède un langage spécifique de description des tests, description qui se fait par des fichiers de requêtes.

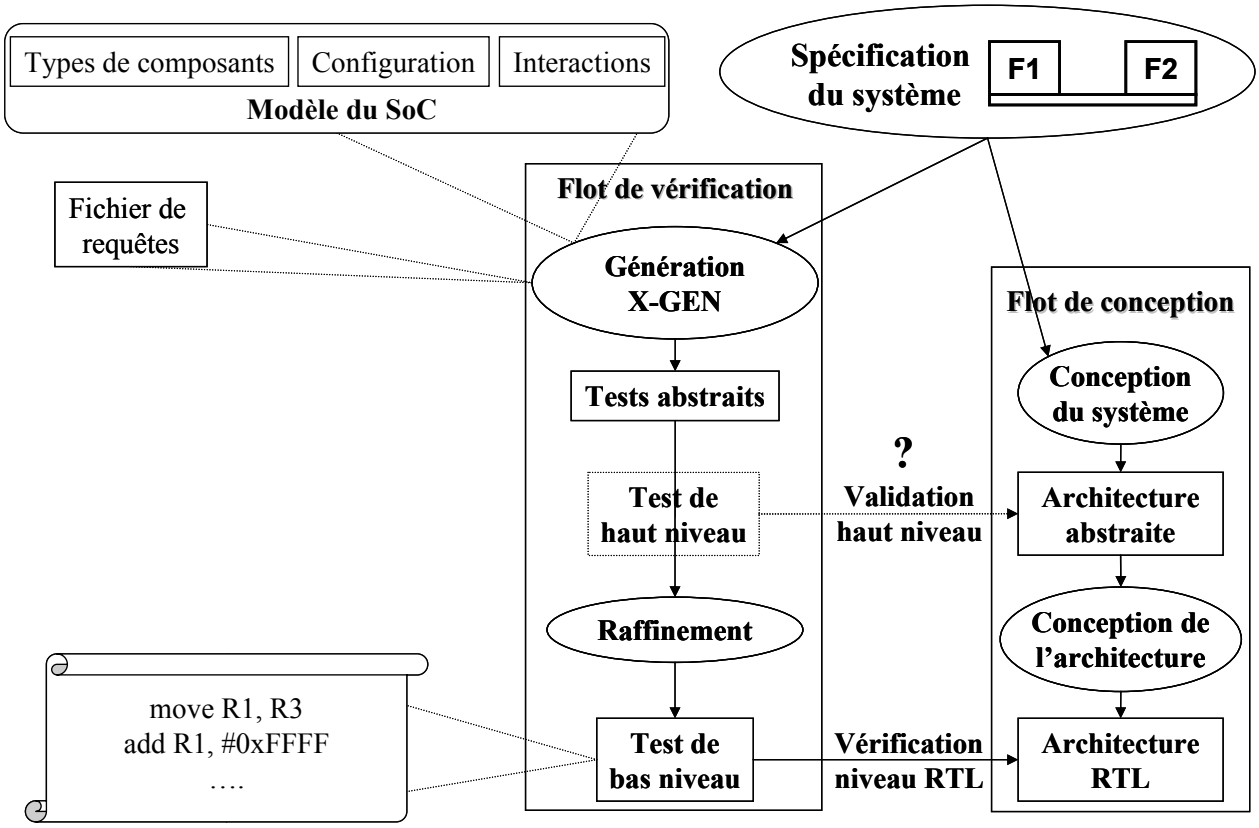


Figure 11 : Flot de vérification des systèmes monopuces : X-Gen (IBM) dans le flot de vérification et conception idéal

Comme le montre la Figure 11, la méthode se décompose en quatre éléments principaux qui sont :

- **La modélisation du système**. Elle se décompose en trois parties qui sont la modélisation des composants, les interactions et la configuration.
    - La modélisation du type de composants correspond à la description des composants matériels. Le modèle est d'abord décrit par des ports qui se composent d'un ensemble de propriétés définissant l'interface (adresse, donnée, priorité d'interruption…). Ensuite intervient la définition de ses ressources internes (registres…) qui peuvent influer sur le comportement. Enfin, la modélisation intègre un modèle du comportement du composant qui est décrit par un ensemble de contraintes liant les ports et les ressources internes.

- Les interactions permettent de décrire comment les différents composants du système interagissent. Par exemple, le transfert de données par un DMA (Direct Memory Access) sera décrit en trois étapes : le processeur initialise le DMA, ce dernier effectue le transfert et indique finalement au processeur la fin de l'opération par une interruption.
- La dernière étape de la modélisation est la configuration du système. Elle correspond à l'instantiation des composants du système et à la description de leurs interconnexions.

On remarquera que cette méthode ne permet pas de modéliser la partie logicielle du système.

- **Les fichiers de requêtes**. Ces fichiers permettent de définir les scénarios de test. Ils sont décrits dans un langage spécifique [44] qui permet de regrouper des interactions, briques de base, en utilisant par exemple une instruction de haut niveau qui permet de générer x fois la même interaction. Les interactions spécifient quels sont les acteurs (composants) impliqués pour une transaction donnée. La relation d'ordre des interactions peut également être spécifiée (séquentiel, Rendez-Vous...). Enfin, il est possible de biaiser la génération des tests par l'utilisation d'une base de donnée de connaissances sur le test qui est incluse à la fois dans le modèle et dans le générateur.
- **Le générateur de programmes de test abstraits**. La génération se fait en deux étapes, le parcours des instructions de haut niveau et des interactions définies dans le fichier de requête. Chaque interaction donne ensuite lieu à la génération d'un test abstrait. Suivant le fichier de requête, le générateur peut choisir aléatoirement les acteurs et les valeurs affectés aux différentes propriétés. Cette génération utilise un solveur de contraintes.
- **Le raffinement des tests abstraits en tests concrets**. Le raffinement correspond à la traduction du test abstrait en programme de test logiciel de bas niveau (assembleur) et en commande de contrôle pour le simulateur. La génération de commande pour le simulateur permet de réaliser le séquencement des transactions. Cette traduction réalise également l'affectation des données et variables aux différents registres du processeur.

X-Gen propose ainsi une méthodologie pour réaliser la vérification des interactions dans un système monopuce. Il permet de part la modélisation d'avoir un mécanisme de génération

indépendant du système et autorise la réutilisation des modèles de composants. Enfin, la méthode n'est pas destinée à effectuer la vérification du comportement des composants et suppose la vérification indépendante de chacun d'entre eux. Cette méthode cible ainsi plus particulièrement la vérification de l'aspect communication du système. Elle offre la possibilité, par la construction de son langage de description des tests, d'obtenir par exemple des scénarii complexes de trafic sur le réseau de communication dans lesquelles se produisent des collisions entre les transactions.

VI.B.2. La TLV (Top Level Validation)

La Figure 12 présente le flot de la méthode de vérification basée sur la TLV. Cette méthode présente l'intérêt d'utiliser une représentation formelle du système et de permettre la modélisation du logiciel.

La Figure 12 présente aussi par l'intermédiaire du flot de vérification et de conception idéal les principaux éléments de cette méthodologie qui sont la modélisation des composants et du système. Le modèle global est ensuite utilisé par le générateur pour produire des tests de haut niveau constitués d'une séquence de transactions. Enfin une librairie de transacteurs permet de raffiner ces séquences pour obtenir des tests de bas niveau décrits en langage C. Ces principaux éléments sont présentés dans la suite.

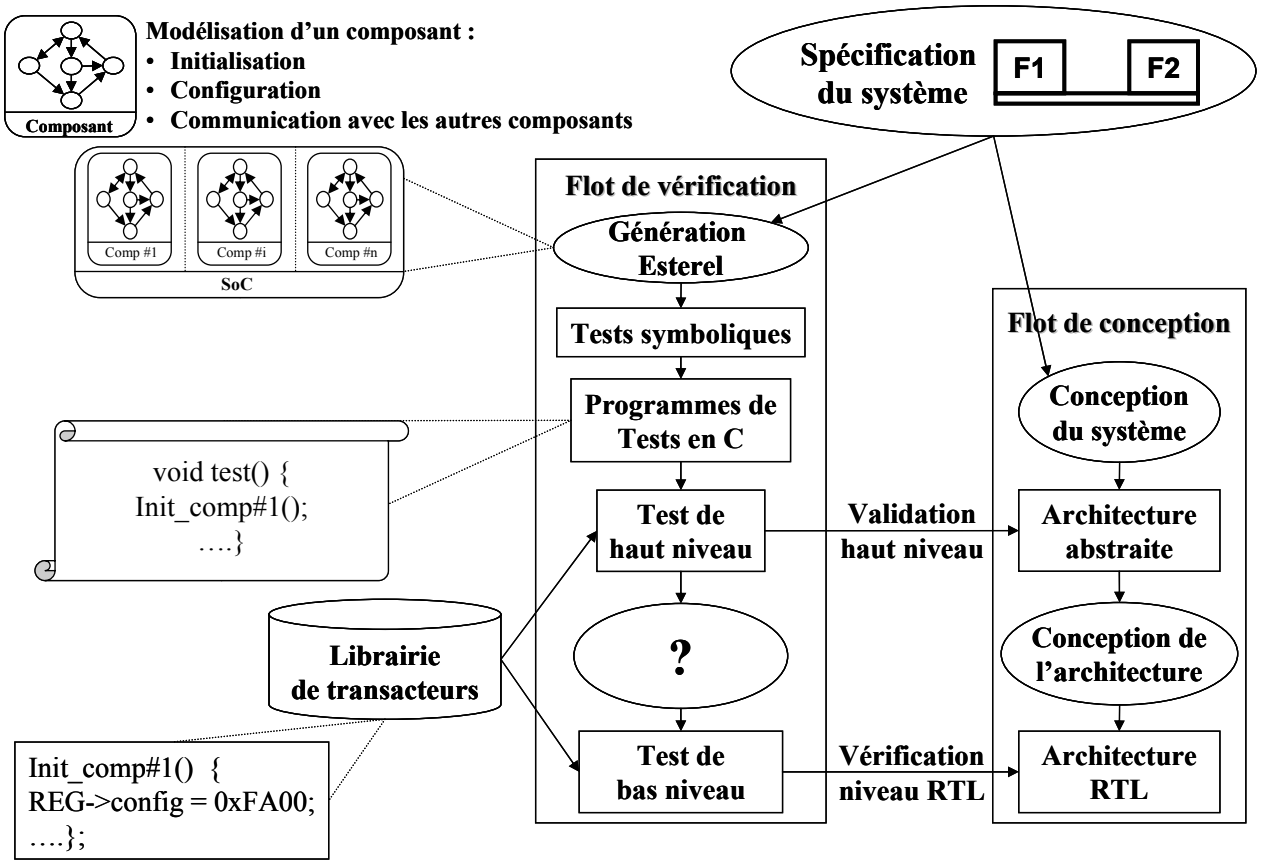


Figure 12 : Flot de vérification des systèmes monopuces : TLV (Top Level Validation : Esterel Techonologies) dans le flot de conception et de vérification idéal

Les principaux éléments de ce flot sont :

- **La modélisation des composants et du système** qui s'appuie sur un modèle formel. Ces modèles sont décrits dans le langage Esterel sous la forme de machine d'états finis hiérarchique et concurrente (HFSM : Hierarchical Finite State Machine). Le modèle HFSM de chaque composant ne comprend que la modélisation de l'initialisation, la configuration et la communication avec les autres composants. La communication entre composants s'effectue sous la forme d'évènements. Ce modèle n'est en aucun cas un modèle du comportement du composant. Ainsi les entrées/sorties de ces modèles représentent des commandes symboliques (évènements) et modélisent les interactions. Le modèle du système rassemble l'ensemble des modèles de chacun des composants et permet également de contraindre l'environnement afin d'éviter la génération de tests non valides.

- **La génération des tests symboliques**. La première phase consiste à énumérer toutes les séquences d'entrées qui couvrent le comportement du système en tenant compte des contraintes de l'environnement. Ce sont en effet les contraintes définies dans le modèle de l'environnement qui permettent de cibler la génération des tests. Les contraintes sont une représentation du scénario de test. L'énumération des séquences d'entrée se base sur les BDDs et leurs parcours pour couvrir tous les états possibles de la FSM globale. Par simulation, ces séquences d'entrées produisent les séquences de sorties associées. Ces sorties correspondent aux commandes de contrôle de différents composants du système et constituent les tests symboliques.
- **Le raffinement des tests symboliques**. Le raffinement consiste à rendre les tests symboliques exécutables. Il est réalisé en transformant les séquences de sortie en appels de fonction correspondante. Le programme ainsi obtenu est séquentiel. Pour une transaction donnée, la fonction réalise les actions concrètes au niveau du système. Ces fonctions sont rassemblées dans une bibliothèque de transacteurs construite manuellement. Ce sont typiquement des fonctions C/C++ exécutées sur un processeur ou un DSP, ou des commandes de contrôle de BFM (Bus Functional Model). Les BFMs désignent ici des composants de l'environnement de test. Il est important de noter que le développement de la librairie des transacteurs est laissé à l'utilisateur et comporte pour l'essentiel des fonctions qui réalisent la configuration des registres des composants. Elle nécessite cependant aussi des fonctions de synchronisation fastidieuses à développer, étant donné le bas niveau d'abstraction utilisé pour le logiciel (HAL).

La TLV est donc une méthodologie permettant la vérification des interactions dans un système sur puce. Comme pour X-Gen, l'hypothèse de base faite pour appliquer cette méthode est la vérification indépendante de chacun des composants. Cette méthode permet, de part la modélisation formelle, la génération de tests réalisant la vérification de toutes les interactions définies par le modèle. Cette exhaustivité peut poser un problème de modélisation : le nombre de scénarii risque de devenir rapidement trop important pour être simulé, suivant la granularité de modélisation des interactions. Cette méthode autorise la réutilisation de la modélisation des composants. Le langage Esterel et la modélisation utilisée permettent également à cette méthode d'offrir la possibilité de modéliser le logiciel. Mais, contrairement à X-Gen l'utilisation de cette méthode pour exercer le système n'est pas suffisante pour permettre la vérification complète du réseau de communication. Comme nous

le verrons plus tard, l'utilisation de programmes de test pour exercer un réseau de communication ne permet pas d'obtenir suffisamment de contrôlabilité. Il est ainsi difficile, contrairement à X-Gen, de réaliser des collisions dans le trafic produit pour vérifier le réseau de communication. Enfin, la TLV nécessite pour le raffinement des tests symboliques, l'écriture d'une bibliothèque de transacteurs complexes, et n'est donc pas complètement automatisée.

VI.B.3. La méthode de vérification du PIAGET

Cette partie décrit succinctement la méthodologie employée pour effectuer la vérification du système PIAGET au niveau RTL. Pour ce projet, la vérification consiste à :

- Développer des programmes de test en C ou assembleur qui seront exécutés sur le microprocesseur pour valider l'intégration (interconnexions et fonctionnalités des composants). Les résultats des tests C ainsi développés sont comparés automatiquement aux résultats prédéterminés (comparaison de l'état de la mémoire). Ces résultats sont soit issus d'un modèle de référence, soit directement définis par le concepteur du test.
- D'autres méthodes sont utilisées en parallèle pour la vérification de points plus spécifiques :
  - Vérification formelle pour les interfaces matérielles
  - Vérification dynamique par utilisation d'un environnement de test spécifique pour la validation du réseau de communication

Ces différentes méthodes sont expliquées dans les sections suivantes. La vérification de l'intégration correspond dans ce cas à la vérification des fonctionnalités des différents composants dans le système ainsi que la vérification de leurs interconnexions.

VI.B.3.a. La vérification de l'intégration

La vérification de l'intégration des composants correspond à valider les fonctionnalités du composant dans le système. Elle s'effectue dans l'environnement système du bloc par l'utilisation de programmes de test C écrits manuellement et exécutés par le processeur. La

vérification du composant seul dans un environnement dédié n'est pas considérée ici, cette partie étant indépendante du système.

Lors de la vérification des composants, on peut classer les tests suivant la complexité. Ainsi, la vérification débute toujours par la vérification des accès aux différents registres du composant. Cela consiste à vérifier le contenu des registres après le reset du système, puis à valider l'accès en lecture et en écriture pour chacun des champs des différents registres. Cette phase est automatisée, un outil permet la génération des programmes de test logiciel à partir d'une description des registres. Une fois l'accès aux registres validé, c'est au travers de la configuration de ces mêmes registres que sont vérifiées les fonctionnalités du composant. Cela correspond à l'écriture de programmes de test en C configurant ces mêmes registres afin d'exercer les différentes fonctionnalités offertes par le composant. Cependant, il est aussi nécessaire de définir le contexte de l'environnement, cela correspond à l'écriture d'un environnement de test capable d'interagir avec le système. La vérification du résultat du test se fait en récupérant la sortie du composant et en la comparant aux résultats prédéterminés par un modèle de haut niveau du système lorsqu'il est disponible.

En résumé, la vérification des composants consiste à valider :

- L'accès aux registres ;
- La génération d'interruptions ;
- L'interconnexion du composant dans le système ;
- Les fonctionnalités de base qui correspondent aux différents modes de fonctionnement (contrôlés par registres). Pour ces différents modes, il est nécessaire de prendre en considération le type du composant : flot de données ou contrôle. Pour le test des composants de type flot de données, il est nécessaire, en plus des différents modes de fonctionnement, de considérer les différents formats de donnée en entrée pour effectuer une vérification complète.

Une fois la vérification de l'intégration de chacun des composants achevée, vient la vérification de l'intégration des sous-systèmes et du système. Un sous-système est un ensemble de composants qui réalisent ensemble une fonction du système. La réalisation de cette tache s'effectue également par l'emploi de programmes de test logiciel de bas niveau. Elle consiste à faire fonctionner simultanément plusieurs composants et ainsi vérifier que les

interactions entre ces composants s'effectuent correctement. On remarquera que cette méthode exploite une approche hiérarchique dans la complexité des tests.

VI.B.3.b. La vérification du réseau de communication

Un point important de la vérification du système PIAGET est la vérification du réseau de communication. Ce réseau est construit à l'aide de plusieurs bus et possède un grand nombre de ports de communication. Il implémente différents algorithmes de gestion d'accès au réseau qu'il est également nécessaire de tester. La vérification du réseau de communication est principalement basée sur l'utilisation d'un environnement de test dédié, même si celui-ci est également exercé par les différents tests de composants ou du système. Cet environnement s'appuie sur l'outil Specman et plus particulièrement sur l'utilisation des composants de vérification génériques définis au sein de STMicroelectronics dans l'environnement CATG (Checker And Test Generator). Ainsi, l'extraction des différents composants correspondant au réseau de communication et leur lien avec les composants de vérification permettent d'achever la vérification. Cette approche de la vérification basée sur l'utilisation d'un environnement de test spécifique est nécessaire de par :

- La spécificité du réseau de communication pour un système donné ;
- La contrôlabilité réduite des flux circulant sur ce réseau par la seule utilisation de programmes de test logiciel.

Il a ainsi été développé un outil permettant la génération d'un environnement de test spécifique pour la vérification du réseau de communication. Cet outil fait également partie des contributions qui ont été réalisées au sein de STMicroelectronics. Elle est détaillée dans le chapitre 3.

De plus, la vérification du réseau de communication comprend également une part de vérification formelle (vérification de propriétés), destinée à valider les différents types d'arbitrage utilisés dans le réseau.

VI.B.3.c. La vérification des interfaces matérielles

Dans le processus de vérification de l'intégration, il est également nécessaire de vérifier la conformité au protocole de communication des interfaces matérielles entre le réseau de communication et les composants. Cette vérification est réalisée par l'intermédiaire de Rulebase avec un environnement nommé COPS (Checker Of Protocol System). Le mécanisme de vérification utilisé est la vérification de propriétés. Ainsi, pour le protocole de communication un ensemble de propriétés est défini, permettant de s'assurer que le composant ne bloque pas le réseau de communication par des erreurs de protocole. A l'instar de la vérification du réseau de communication, l'utilisation de programme de test ne permet pas de réaliser la vérification complète de l'interface matérielle car la contrôlabilité est réduite. Cette partie fait également l'objet d'une description plus complète dans le chapitre 3.

VII. Conclusion

Cet état de l'art sur la vérification des systèmes numériques montre bien la diversité des outils et méthodes existants. A travers la présentation de la vérification des blocs matériels, nous avons montré que la vérification dynamique et la vérification formelle sont complémentaires. Dans [26] est présentée une évaluation entre ces deux méthodes. Cette expérience met en évidence que le temps de mise en place de la méthode formelle est plus long, cependant elle permet d'effectuer une vérification complète. Le nombre d'erreurs détectées est ainsi plus important que dans le cas de la vérification dynamique. L'idéal serait donc de réaliser la vérification des systèmes par l'application de méthodes formelles. Malheureusement, ces méthodes restent confrontées à la fois à la complexité du circuit adressé et aux contraintes de temps imposées, rendant à l'heure actuelle impossible leur application unique à la vérification d'un système monopuce.

Le tableau ci-dessous présente un résumé de l'analyse des principaux outils et méthodes de vérification. Il décrit ainsi pour les méthodes dynamiques les critères d'analyse et d'évaluation définis dans ce chapitre.

	Critères d'analyse			Critères d'évaluation		
Méthodes / Outils	Type de circuit	Modélisation employée	Vecteurs de test	Taux de couverture	Coût d'utilisation	Limitations
Env de test (Specman ,Vera,…)	Bloc HW	Producteur - consommateur	Stimuli générés par l'env. de test	Bonne couverture de la FSM (Etats et transitions)	Construction et maintenance de l'env	Stimuli non exhaustifs
	CPU			Bon taux de couverture du jeu d'instruction		Complexité de mise en place Stimuli non exhaustifs
	Système monopuce			Bon taux de couverture pour les interactions		Pas de vérification du SW Complexité de mise en place Stimuli non exhaustifs
Genesys-Pro (IBM)	CPU	Basée sur le jeu d'instruction du processeur	Stimuli générés par des pgs de test en assembleur	Très bon taux de couverture du jeu d'instruction	Modélisation du jeu d'instruction	Stimuli non exhaustifs
X-Gen (IBM)	Système monopuce	Modélisation des composants et de leurs interactions	Stimuli générés par des pgs de test logiciel en assembleur	Taux de couverture pour les interactions défini par le modèle	Modélisation du système	Pas de vérification du SW Pas de vérification de la fonctionnalité des composants dans le système
TLV (Esterel		Interactions modélisées sous forme de FSM	Stimuli générés par des programmes de test logiciel de haut niveau (HAL)	Taux de couverture pour les interactions défini par le modèle	Modélisation du système Ecriture de la librairie des transacteurs	Pas de vérification du réseau de communication Vérification du SW non complète (HAL uniquement) Limitation due au modèle formel
Ecriture manuelle des pgs de test		Plan de test	Stimuli générés par des programmes de test logiciel (HAL)	Taux de couverture pour les interactions défini à partir du plan de test	Toutes les tâches sont manuelles	Vérification du réseau de communication Vérification du SW non complète (HAL uniquement) Stimuli non exhaustifs

Tableau 1 : Tableau récapitulatif sur les outils et méthodes de vérification dynamique

Au travers de ce tableau, on peut faire plusieurs constatations :

- La vérification dynamique par un environnement de test est très générale et s'applique à tous les types de circuit. Elle est cependant la mieux adaptée à la vérification des blocs matériels et reste également applicable aux processeurs. Elle ne permet cependant pas de prendre en compte la partie logicielle dans le cas des systèmes monopuces. Par ailleurs, plus la taille du système augmente et plus le temps de conception et de maintenance de l'environnement augmente.
- Il existe des méthodes qui exploitent et ciblent les caractéristiques de certains types de systèmes. C'est le cas de la vérification des processeurs avec leur jeu d'instruction et des systèmes monopuces avec les programmes de test logiciel.
- On remarque que la vérification des systèmes monopuces s'effectue par l'utilisation de programmes de test logiciel de bas niveau (ISA, HAL). La Figure 13 donne, sur une représentation en couche d'un système monopuce, les parties vérifiées par les trois méthodes présentées dans ce chapitre. Sur cette figure, les flèches représentent les interactions et leur niveau. Il apparaît que ces méthodes ne permettent donc pas d'intégrer dans la vérification la partie logicielle du système. D'autre part, certaines parties matérielles du système ne sont pas complètement vérifiées comme par exemple le réseau de communication et les interfaces, excepté avec la méthode X-Gen. Enfin, on remarquera que le niveau d'abstraction utilisé pour les programmes de test ne permet pas de mettre en oeuvre aisément des mécanismes de synchronisation logicielle / matérielle sophistiqués.

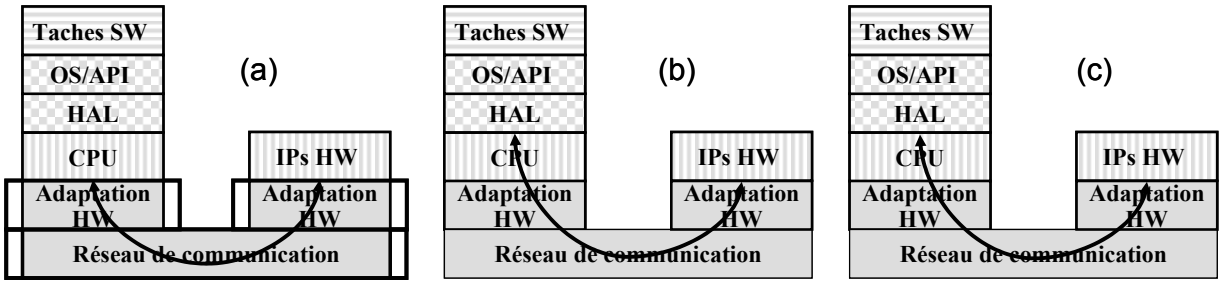


Figure 13 : Représentation des parties vérifiées dans un système monopuce par les méthodes (a) X-Gen, (b) TLV et (c) écriture manuelle de programmes de test

La suite de ce mémoire propose une méthodologie permettant la validation globale d'un système monopuce. Plusieurs approches sont utilisées pour répondre aux besoins de la vérification et prendre en compte toutes les parties du système.

Chapitre 3 : PROPOSITION D'UNE METHODOLOGIE DE VALIDATION GLOBALE POUR LES SYSTEMES MONOPUCES

I. INTRODUCTION 62

II. LES INTERFACES LOGICIELLES / MATERIELLES DANS LE CADRE DE LA CONCEPTION ET DE LA VERIFICATION DES SYSTEMES MONOPUCES 62

III. DESCRIPTION DU FLOT DE VERIFICATION GLOBAL ET L'IMPORTANCE DES INTERFACES LOGICIELLES / MATERIELLES 66

III.A. LA DECOMPOSITION DU FLOT DE VALIDATION GLOBAL 66
III.B. LA VERIFICATION DES INTERFACES HW : METHODE FORMELLE 67
III.C. LA VERIFICATION DU RESEAU DE COMMUNICATION : METHODE DYNAMIQUE UTILISANT UN ENVIRONNEMENT DE TEST SPECIFIQUE 69
III.D. LA VERIFICATION DE L'INTEGRATION : METHODE UTILISANT DES PROGRAMMES DE TEST DE HAUT NIVEAU 72

IV. TECHNIQUES DE BASE POUR LA VERIFICATION DE L'INTEGRATION PAR DES PROGRAMMES DE TEST LOGICIEL DE HAUT NIVEAU 73

IV.A. DESCRIPTION DE L'ENVIRONNEMENT ROSES 73
IV.B. LE MODELE DE REPRESENTATION DES SYSTEMES : COLIF 75
IV.B.1. Une description structurelle modulaire 75
IV.B.2. Les différents niveaux d'abstraction 77
IV.B.3. Composants virtuels et ports hiérarchiques 77
IV.C. PRESENTATION DU GENERATEUR DE SYSTEME D'EXPLOITATION : ASOG 78
IV.C.1. Concept : le ciblage logiciel 78
IV.C.2. Les mécanismes d'assemblage 78
IV.C.3. Structure de la bibliothèque 78
IV.C.4. Le flot de génération automatique de systèmes d'exploitation 80

V. CONCLUSION 81

I. Introduction

Ce chapitre présente une proposition de méthodologie de vérification globale pour les systèmes monopuces. Cette proposition est le fruit de l'expérience industrielle acquise au cours de cette thèse et de l'analyse des différentes méthodes et outils de vérification existants qui ont été présentés dans le chapitre précédent. Afin d'expliquer la méthodologie, il est important de décrire les interfaces logicielles / matérielles. L'existence des interfaces est primordiale pour expliquer le découpage de la méthodologie. Celle qui est proposée se décompose en trois parties :

- Vérification de l'intégration ;
- Vérification des interfaces matérielles ;
- Vérification du réseau de communication.

Ce chapitre s'achève par la présentation de l'environnement ROSES qui constitue la base pour le développement de la méthode de vérification de l'intégration. Cet environnement est le fruit des travaux du groupe SLS du laboratoire TIMA [50],[51],[52],[53].

II. Les interfaces logicielles / matérielles dans le cadre de la conception et de la vérification des systèmes monopuces

La Figure 14 présente l'évolution de l'architecture d'un système sur puce dans le flot de conception et de vérification idéal. Cette évolution est décrite de la phase de spécification du système à la réalisation du système au niveau RTL. Elle montre ainsi les changements de l'architecture pendant le raffinement, c'est-à-dire l'apparition des interfaces logicielles / matérielles.

Comme le montre la Figure 14, la conception d'un système débute toujours par la spécification. La première étape consiste à réaliser le partitionnement logiciel / matériel. La deuxième étape consiste ensuite à réaliser un modèle abstrait de cette architecture. Il est, par exemple, réalisé en SystemC au niveau TLM (Transaction Level Modeling). Ce modèle comprend les différentes parties logicielles de l'application et les différents composants matériels. Ce niveau d'abstraction est obtenu par l'abstraction de la communication. Ce

modèle abstrait se compose de trois parties : le logiciel, le matériel et une interface logicielle / matérielle.

Finalement, pour aboutir à la réalisation RTL du système, il faut réaliser le raffinement de l'interface logicielle / matérielle. Ce raffinement fait apparaître des nombreuses couches d'adaptation logicielle et matérielle. Ainsi, il introduit :

- Plusieurs couches logicielles : API, OS et HAL qui réalisent l'adaptation du logiciel applicatif au processeur ;
- Mais aussi des couches matérielles : CPU, réseau de communication et interfaces matérielles qui réalisent l'adaptation des composants (HW ou CPU) au protocole du réseau de communication.

On observe ainsi lors du raffinement du modèle abstrait en architecture RTL qu'il est nécessaire de vérifier l'ensemble des couches qui en résultent.

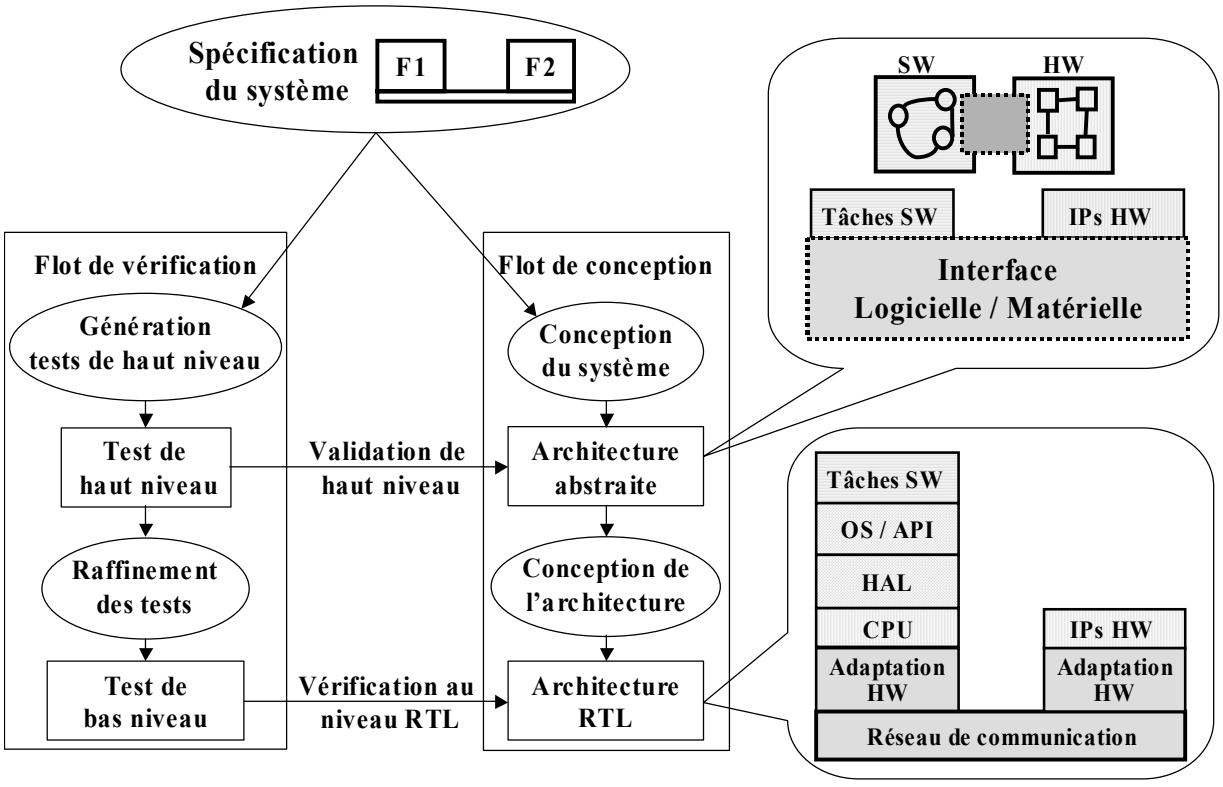


Figure 14 : Représentation de l'architecture d'un système monopuce dans le flot de conception et de vérification idéal

La présentation de la conception des systèmes monopuces met en évidence la complexité de ces systèmes. Elle permet également de montrer l'importance croissante de la part du logiciel dans de tels systèmes.

A partir de la présentation de l'évolution de l'architecture des systèmes monopuces dans les différentes phases de conception (Figure 14), on remarque que le point clé du passage de la réalisation du niveau abstrait au niveau RTL est le raffinement de l'interface logicielle / matérielle. La Figure 15 présente cette interface et montre de manière détaillée la complexité de ses différentes couches.

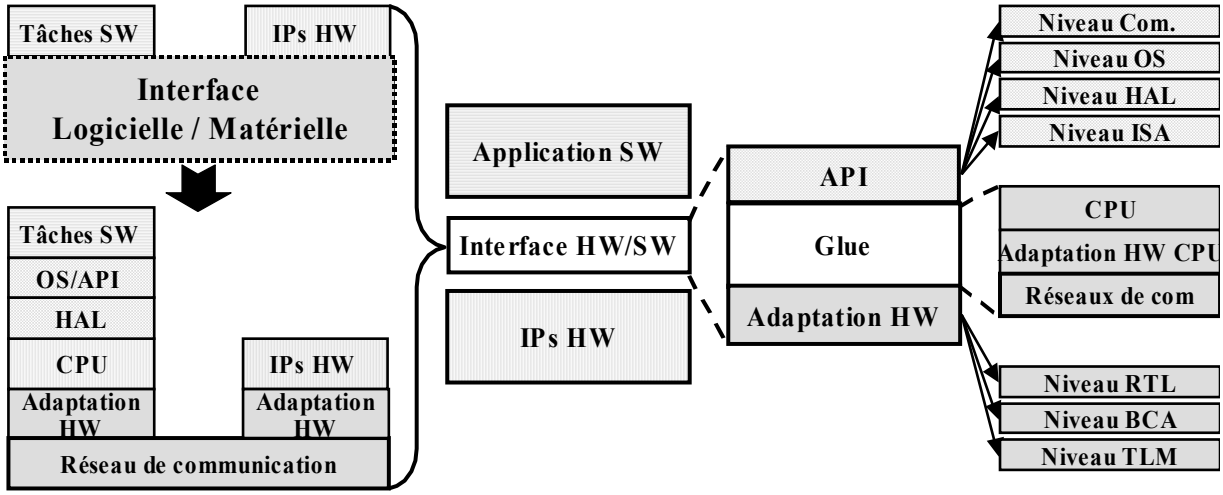


Figure 15 : Représentation de la composition de l'interface logicielle / matérielle

La Figure 15 montre que cette interface se divise en trois ensembles :

- **L'interface pour la partie logicielle du système** : elle se compose de couches logicielles et matérielles. L'application étant construite sur une API (Application Programming Interface), il est possible de distinguer plusieurs niveaux d'abstraction pour cette API :
    - Communication
    - OS: Operating System
    - HAL : Hardware Abstraction Layer
    - ISA : Instruction Set Architecture

De plus, on remarquera que quelque soit le niveau d'abstraction de l'API, les niveaux inférieurs sont nécessaires pour réaliser l'interface. Sous la couche ISA, il y a deux couches d'adaptation matérielle. En effet, le processeur est nécessaire à l'exécution du logiciel. De plus, suivant le réseau de communication, l'utilisation d'une couche d'adaptation du protocole du processeur au protocole du réseau de communication peut se révéler indispensable.

- **L'interface pour les composants matériels** : elle se compose uniquement d'une couche d'adaptation du protocole du composant matériel au protocole du réseau de communication.
- **Le réseau de communication** : il permet de réaliser les échanges de données entre les différents composants du système. Ce réseau de communication est généralement spécifique au système. Il existe un grand nombre de réseaux de communication pouvant être réalisés par :
  - Un ensemble de connexion point à point ;
  - Un ensemble de bus tel que AMBA-AHB™, STBus™, Sonics™…
  - Un réseau sur puce (NoC : Network on Chip).

La vérification des systèmes monopuces est ainsi confrontée aux problèmes suivants :

- La vérification des interfaces matérielles ;
- La vérification du réseau de communication ;
- La vérification des interactions entre le matériel (IP HW) et le logiciel (Application SW) : la vérification de l'intégration. Cette étape nécessite également la validation de la couche d'adaptation logicielle et en particulier le HAL (Hardware Abstraction Layer).

III. Description du flot de vérification global et l'importance des interfaces logicielles / matérielles

Cette section décrit le flot de vérification global proposé. La première sous-partie présente et justifie le découpage du flot de vérification. Les parties suivantes se focalisent chacune sur des étapes de la méthodologie de vérification.

III.A. La décomposition du flot de validation global

Un système sur puce peut se représenter sous forme de pile informatique. La Figure 16 permet de re-préciser les couches logicielles et matérielles qu'il est nécessaire de vérifier.

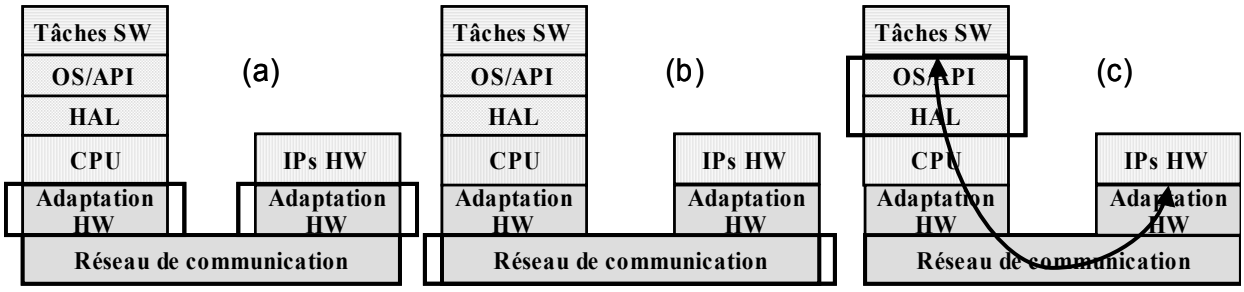


Figure 16 : Représentation de la vérification sur le modèle en couche d'un système monopuces : (a) vérification de l'interface matérielle, (b) vérification du réseau de communication, (c) vérification de l'intégration et la validation de l'interface logicielle

Pour la vérification, l'utilisation de programmes de test logiciel n'est pas suffisante pour effectuer la vérification complète de toutes les couches. Ainsi, la vérification du réseau de communication et celle des interfaces matérielles ne sont pas réalisables facilement et efficacement par des programmes de test logiciel. En effet, la contrôlabilité de ces couches à partir du logiciel est limitée. Il est par exemple difficile dans le cas du réseau de communication d'obtenir des scénarii dans lesquels le trafic généré par les composants permet de créer des conflits pour l'accès au réseau. De même pour les interfaces matérielles il est difficile de créer des conditions permettant de saturer facilement les FIFOs que contiennent en générale ces interfaces. Il est donc nécessaire pour achever la vérification de toutes les

couches de réaliser un découpage de la vérification et d'utiliser des méthodes appropriées. Le découpage proposé est le suivant :

- La vérification des interfaces matérielles (Figure 16 a) utilisant une méthode formelle : la vérification de propriétés
- La vérification du réseau de communication (Figure 16 b) réalisé par simulation et basée sur l'utilisation d'un environnement de test spécifique
- La vérification des interconnexions et des interactions entre les composants du système, également appelée vérification de l'intégration (Figure 16 c). Celle-ci est réalisée par simulation, elle emploie des programmes de test logiciel de haut niveau. Elle nécessite cependant la validation de la couche logicielle d'abstraction du matériel.

Ces trois méthodes sont présentées dans les sections suivantes.

III.B. La vérification des interfaces HW : méthode formelle

La conception des systèmes sur puce se base en général sur l'utilisation d'un réseau de communication utilisant un protocole bien défini, par exemple : STBus™, AMBA™, Sonics™,… Chaque composant doit, pour être intégré dans le système, se conformer au protocole du réseau de communication. Ceci nécessite dans la plupart des cas, l'utilisation d'une interface matérielle réalisant l'adaptation du protocole du composant à celui du réseau.

Etant donné la standardisation des protocoles de communication, il est possible pour chacun d'entre eux de définir un ensemble de propriétés devant êtres vérifiées par l'interface. Elles permettent la vérification de la conformité de l'interface avec le protocole de communication.

Cette méthode est préférable à l'utilisation de programmes de test logiciel exécutés par les processeurs du système. En effet, cette méthode manque de contrôlabilité, c'est à dire qu'il est très difficile voire impossible en utilisant un programme de test d'atteindre les cas limites de l'interface. Ce n'est pas le cas dans les méthodes formelles puisqu'elles sont exhaustives.

Il est ainsi possible dans le cas de la vérification des interfaces matérielles de disposer d'un ensemble de propriétés prédéfinies permettant de dériver de façon automatisée des propriétés adaptées à la cible. Aussi au cours de cette thèse il m'a été donné de mettre en

œuvre une telle méthode. Cette méthode utilise le logiciel RuleBase (IBM) qui est un outil de preuve formelle basé sur la vérification de propriétés. Il a ainsi été conçu au sein de STMicroelectronics un outil nommé COPS (Checker Of Protocol System). Cet outil est destiné à la vérification des interfaces matérielles utilisant le protocole de communication STBus™.

Le fonctionnement de cet outil est illustré sur la Figure 17. Il permet pour les composants s'interfaçant à un réseau de communication STBus, de générer un ensemble de propriétés définies dans le langage PSL. Pour réaliser cette opération, l'outil analyse la description RTL du composant pour en identifier le type et déterminer le nom des signaux de son interface. Il existe trois types de protocole STBus : T1, T2, T3 et deux catégories d'interface : initiateur, cible qui sont plus généralement connus dans la littérature sous l'appellation : maître, esclave. A partir d'une librairie de propriétés génériques et de ses paramètres, le générateur spécialise l'ensemble des propriétés devant êtres vérifiées par l'interface. En sortie du générateur, on obtient un fichier qui contient l'ensemble de propriétés adaptées à l'interface. Celui-ci est ensuite utilisé par l'outil de preuve formelle (RuleBase) pour achever la vérification.

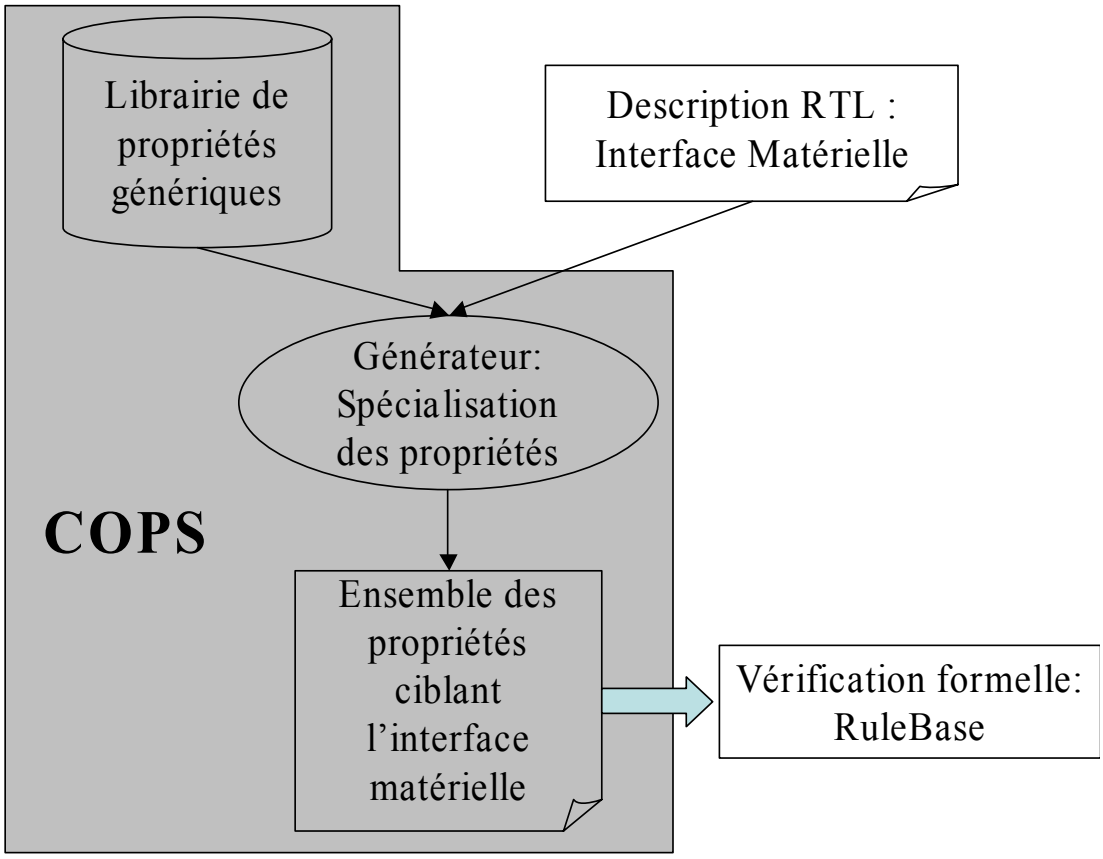


Figure 17 : Représentation du flot de génération de propriétés pour la vérification des interfaces matérielles basé sur le protocole STBus (COPS)

Pour finir d'illustrer cette méthode, voici un exemple très simple de propriété. Tout d'abord, cette propriété peut être spécifiée de manière textuelle : « Lorsqu'une requête est faite, celle-ci est maintenue tant qu'elle n'a pas été acquittée ». Ce qui se traduit en PSL de la manière suivante :

« ASSERT ALWAYS ((signal_request & !signal_grant) → next signal_request); ».

III.C. La vérification du réseau de communication : méthode dynamique utilisant un environnement de test spécifique

Comme cela vient d'être précisé pour la vérification des interfaces matérielles, le réseau de communication se base sur des protocoles existants et bien définis. Cependant avec la complexité des systèmes et la multitude de composants interconnectés (dans le cas du système

PIAGET, il y a: 47 initiateurs et 48 cibles), la taille et la complexité de tels réseaux ne cessent de croître. Ce réseau de communication se compose de plusieurs bus.

Les méthodes de vérification formelle sont donc difficilement applicables. En effet, les méthodes formelles sont limitées par la taille du système sous test. En d'autre terme la vérification formelle est confrontée au problème de l'explosion combinatoire. Il est de ce fait préférable d'utiliser une méthode basée sur la simulation.

Ainsi dans le cadre du projet PIAGET, l'expérience de la vérification de ce réseau a montré que le problème posé est de réaliser la vérification de l'intégration de ces différents bus dans le système. Dans ce contexte, un nouvel outil permettant de réaliser la vérification de l'intégration d'un réseau de communication a été développé. Celui-ci permet de générer automatiquement un environnement de test, construit autour de la réalisation RTL du système (en VHDL dans notre cas). Il permet d'exercer le réseau de communication dans son ensemble et dans la configuration de la réalisation finale.

L'utilisation de cette méthodologie a été privilégiée car elle offre une contrôlabilité beaucoup plus fine que dans le cas des programmes de test. D'autre part, cette approche s'appuie sur un outil réalisé au sein de STMicroelectronics, CATG (Check And Test Generator). Cet outil est destiné à la vérification d'une réalisation STBus. Il permet de construire un environnement de test générique lors de la conception d'un seul bus. L'environnement ainsi généré repose sur le logiciel Specman et est décrit dans le langage e. La structure de cet environnement se compose d'un ensemble de composants de vérification génériques également connu sous l'appellation vIP (verification IP) ou eVC (e Verification Component). Ils existent plusieurs type d'eVC, chacun correspondant à un type d'interface STBus. De la même manière que dans le cas de la vérification des interfaces matérielles, il en existe six. Les six eVCs sont issues d'une combinaison entre le type du protocole (T1, T2, T3) et du type de l'interface (initiateur, cible).

L'outil de génération d'un environnement de test spécifique pour la vérification d'un réseau de communication basé sur le STBus est présenté sur la Figure 18. Il a pour but de générer un environnement de test qui respecte l'intégration du réseau de communication. Les contraintes d'intégration sont principalement liées à l'assemblage de plusieurs réseaux STBus et à la souplesse offerte au niveau de l'interface, certains signaux de l'interface n'étant pas obligatoires et d'autres pas nécessaires suivant les opérations que réalise le composant. Un exemple simple est celui d'un composant qui réalise toutes ses opérations de la même

manière : envoi de deux mots par transfert. Dans ce cas les signaux codant le mode de transfert peuvent être simplifiés (fixés).

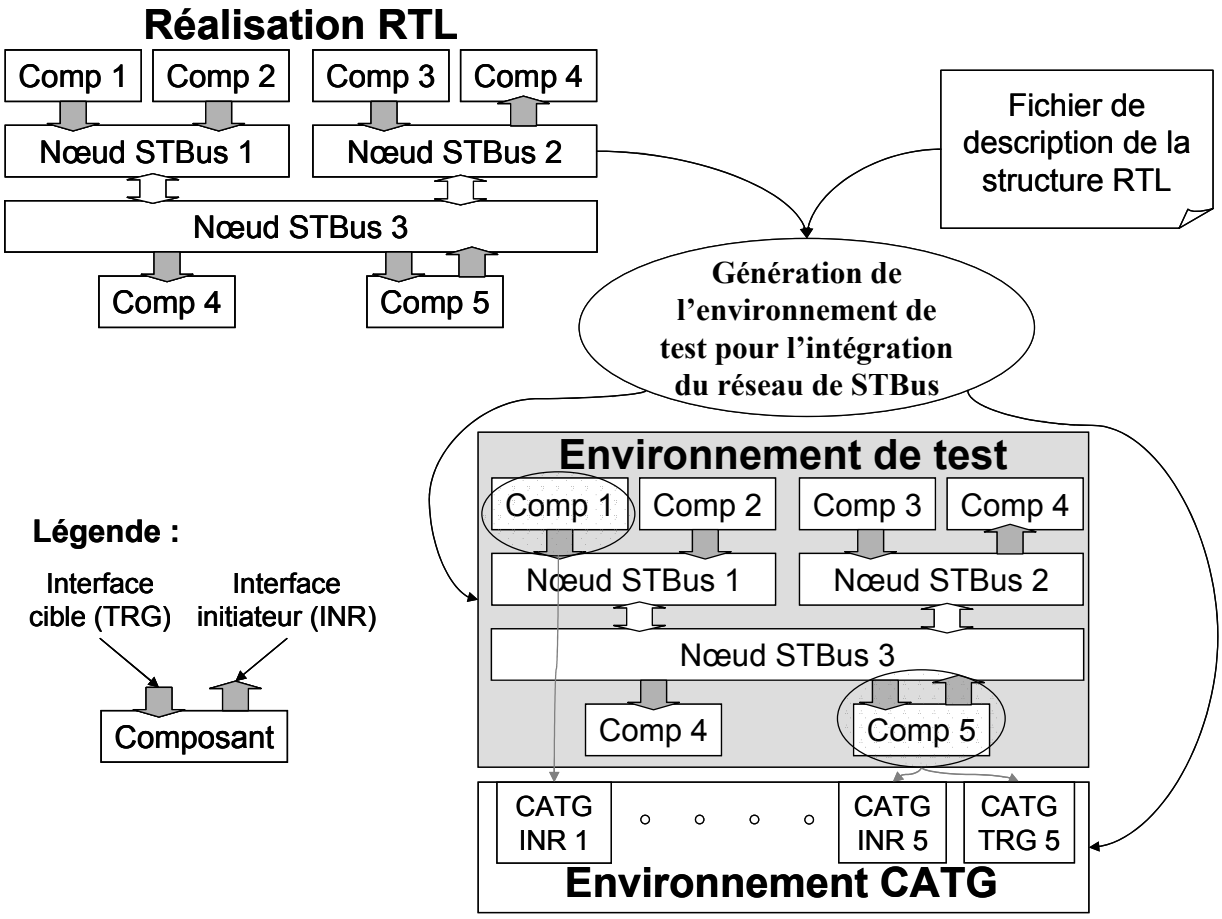


Figure 18 : Représentation schématique de la génération d'un environnement de test spécifique pour l'intégration d'un réseau de communication (STBus)

Pour réaliser la génération de l'environnement de test, l'outil prend en entrée la réalisation RTL du système et un fichier de description de la structure du RTL. Ce fichier contient des informations sur la configuration du système et les interfaces de chacun des composants faisant l'objet d'un remplacement de l'interface par l'eVC approprié. A partir de ces éléments, le générateur produit l'ensemble de l'environnement de test en RTL pour permettre l'interaction avec l'environnement de test CATG - Specman.

La génération de l'environnement s'effectue en deux étapes :

- L'analyse de la réalisation RTL et la construction de la partie RTL de l'environnement : cette étape comprend une phase de reconnaissance de l'interface

du composant à remplacer. Vient ensuite la génération de l'environnement qui consiste à substituer le comportement des composants par un ensemble de signaux pour permettre l'interconnexion de l'interface avec l'environnement Specman – CATG. Pour finir, l'outil génère la configuration de l'ensemble du système et de son environnement.

- La génération de la partie de l'environnement Specman – CATG : cette étape consiste à créer l'ensemble de fichier de configuration de l'environnement CATG. Elle s'effectue par l'instantiation des eVCs et la spécification des signaux d'interconnexion avec la réalisation RTL.

De cette manière, l'environnement de test construit prend en compte les spécificités des différentes interfaces substituées par des eVCs. L'outil permet ainsi d'obtenir un environnement sur mesure et de manière systématique, étant donné qu'il est possible en procédant de cette façon de remplacer n'importe quelle interface STBus du système. Enfin, il automatise la création de l'environnement de test qui reste une tâche fastidieuse. Si cette tâche était réalisée manuellement, elle serait propice aux erreurs. En effet, elle nécessite l'analyse minutieuse des signaux de l'interface des différents composants pour en extraire les signaux utilisés pour la connexion avec le réseau de communication. Puis il est nécessaire d'écrire du code HDL permettant de réaliser le lien avec l'environnement CATG.

III.D. La vérification de l'intégration : méthode utilisant des programmes de test de haut niveau

Le principe de base de cette méthode de vérification réside dans l'utilisation de programmes de test logiciel de haut niveau exécutés par le(s) processeur(s) du système. Cette solution repose sur l'emploi d'un système d'exploitation spécifique qui permet d'interfacer les programmes avec le(s) processeur(s). Un intérêt immédiat de l'utilisation d'un système d'exploitation est la possibilité de réaliser les programmes qui exploitent des mécanismes de synchronisation sophistiqués. Cette partie constitue la principale contribution de cette thèse et fait l'objet d'une présentation complète dans le chapitre 4.

Cette méthode de vérification introduit le concept de programmes de test logiciel de haut niveau. Ce nouveau concept, comparé à l'existant qui se base sur des programmes de test logiciel de bas niveau, repose sur l'utilisation d'une API pour la spécification du test, elle

même basée sur un système d'exploitation généré automatiquement. Il devient ainsi possible de définir des programmes de test qui mettent en œuvre des mécanismes de synchronisation complexes :

- Synchronisation entre tâches logicielles (sémaphore, signaux…) ;
- Synchronisation logicielle / matérielle, comme le traitement des interruptions avec gestion des priorités.

L'introduction de la synchronisation logicielle / matérielle permet ainsi de mieux appréhender la vérification des systèmes monopuces qui intègrent un grand nombre de composants matériels s'exécutant parallèlement. Cette méthode permet également de réaliser l'abstraction de la synchronisation et des spécificités matérielles à travers l'utilisation d'une API.

IV. Techniques de base pour la vérification de l'intégration par des programmes de test logiciel de haut niveau

Cette partie comprend la description de l'environnement ROSES qui a été réalisé au laboratoire TIMA par l'équipe SLS. Cet environnement est destiné à faciliter la conception des systèmes sur puce. Dans le cadre de cette thèse, il constitue un point de départ pour la méthodologie de vérification de l'intégration définie par la suite. Cette section décrit les éléments de ce flot qui sont utilisés d'une part pour la modélisation du système, mais aussi pour réaliser le raffinement automatique des programmes de test logiciel de haut niveau.

IV.A. Description de l'environnement ROSES

Le flot de conception du groupe SLS, nommé ROSES, permet la conception de systèmes monopuces à partir de composants hétérogènes. Il permet l'utilisation de plusieurs processeurs mais aussi de blocs matériels. Il résout les problèmes liés à la conception des systèmes complexes par la génération automatique d'interfaces logicielles/matérielles. La conception des parties logicielles et matérielles est conjointe. De plus, il permet une validation à plusieurs niveaux d'abstraction par simulation ainsi que la réalisation automatique d'un système à partir d'une spécification de haut niveau.

Dans un premier temps, le flot permet de générer un système d'exploitation spécifique pour chacun des processeurs, c'est à dire que chaque processeur possède son propre OS pour gérer la communication et la synchronisation de ses tâches. Il n'est pas possible d'obtenir un OS global gérant l'ensemble des processeurs. Cette hypothèse n'est pas très contraignante dans le cas des systèmes sur puce dédiés. En effet, la plupart des processeurs sont dédiés à des fonctions particulières conçues pour agir de manière indépendante.

Le principe le plus important dans ce flot est la conception d'un système complet par assemblages d'éléments. Que ce soit pour composer les parties logicielles des interfaces ou les parties matérielles, mais aussi pour le développement de modèles de simulation, la technique reste la même et consiste en un assemblage d'éléments de bibliothèque. On retrouve cette technique à différents niveaux : le système à concevoir sera un assemblage de composants (processeurs, IP) liés par des interfaces de communication. Ces interfaces seront le résultat d'assemblages de parties logicielles et matérielles qui auront été obtenus par composition d'éléments de bibliothèque.

Le flot ROSES est présenté sur la Figure 19. Il débute par une description de haut niveau du système qui comporte les paramètres nécessaires au raffinement de cette architecture abstraite. Cette description est réalisée dans le langage VADeL (Virtual Architecture Description Language) qui est une extension du langage SystemC. A partir de cette description, on obtient le modèle Colif [50] du système qui correspond à l'architecture du système et comprend les paramètres relatifs à chaque composant. Ensuite, le modèle de représentation Colif est utilisé par les différents outils qui constituent l'environnement ROSES pour réaliser les actions suivantes :

- **CosimX** [52] permet de réaliser des modèles de simulation du système ;
- **ASOG** [51] (Application-Specific OS Generation) permet la génération de l'interface logicielle, en d'autre terme le raffinement du logiciel applicatif par la génération d'un système d'exploitation spécifique ;
- **ASAG** [53] (Application-Specific Architecture Generation) permet la génération de l'interface matérielle.

La suite de la description insiste sur la présentation du modèle de représentation COLIF et sur le générateur de systèmes d'exploitation. Ces éléments constituent la base de la méthode de vérification présentée dans le chapitre 4.

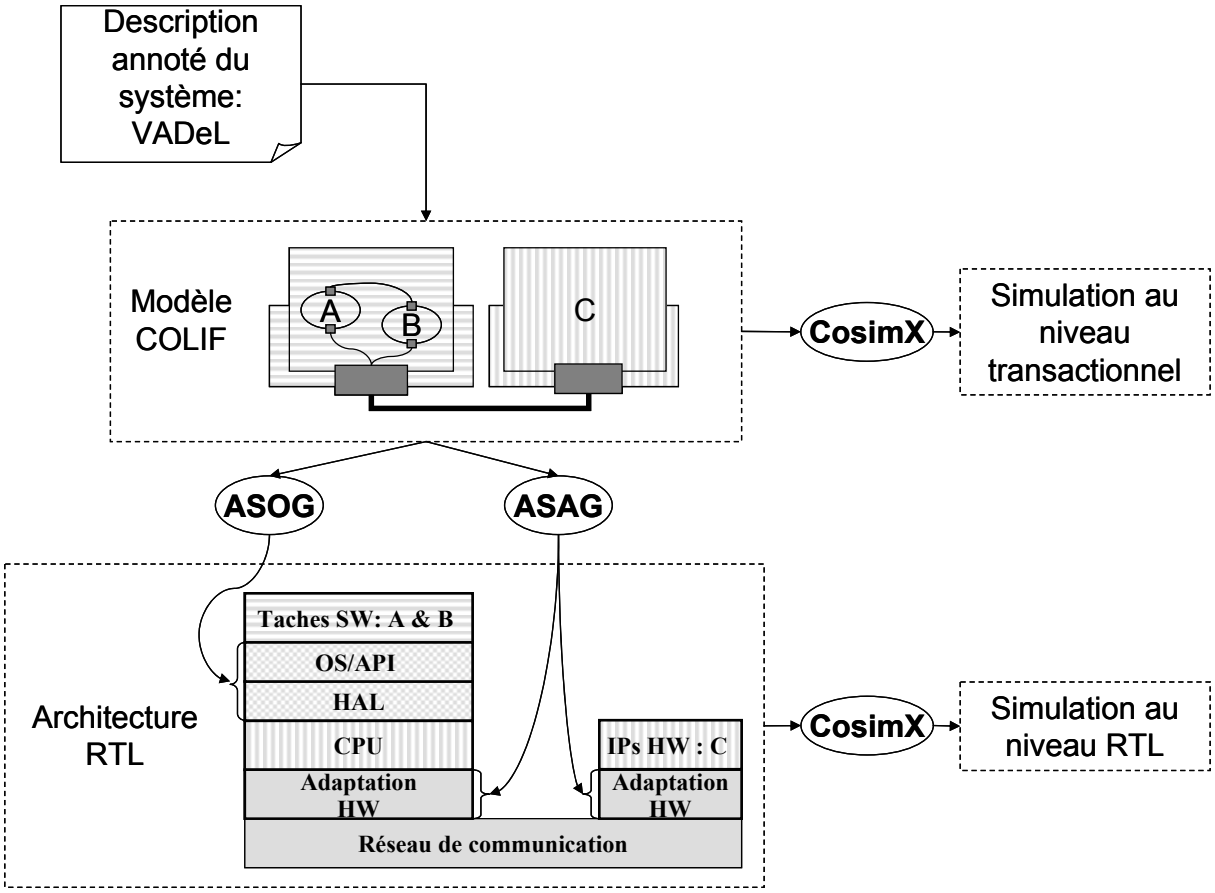


Figure 19 : Représentation du flot de conception des systèmes monopuces : ROSES du groupe SLS du laboratoire TIMA

IV.B. Le modèle de représentation des systèmes : Colif

Le flot utilise un modèle de description commun à tous les outils nommé Colif. C'est un modèle de description de systèmes hétérogènes permettant de définir des modules logiciels et matériels. Avec Colif, la description de la structure du système est indépendante de celle du comportement. Cela permet de modéliser les systèmes à partir de composants hétérogènes par leur niveau d'abstraction, leur langage utilisé ou encore leur type d'implémentation (ex : logicielle ou matérielle).

IV.B.1. Une description structurelle modulaire

La description Colif d'un système est un ensemble d'objets de trois types : les modules, les ports et les nets [50]. Un objet, quel que soit son type, est composé de deux parties : une interface (entity) et un contenu (content). L' «entity » permet la connexion avec les autres objets. Le «content » fournit soit une référence à un comportement, soit des instances d'autres objets. Cette possibilité d'instancier des éléments au sein d'un objet permet le développement de modules, ports et nets hiérarchiques : de cette façon, dans un même module, on peut instancier une structure complète de modules, ports et nets.

Le module représente un composant matériel ou logiciel. Son «entity » donne son type et les ports par lesquels il communique. Son « content » peut être caché (black box), contenir des références à un comportement (ex : fichier C) ou un sous-système de modules, ports et nets.

Le port représente un point de communication pour un module. Son «content » est composé de deux parties : l'une est en relation avec l'intérieur du module et l'autre avec l'extérieur. Ces deux parties peuvent être cachées, contenir des références à un comportement ou des instanciations de ports.

Le net représente une connexion entre plusieurs ports. De la même façon, un net peut contenir des références à un comportement ou contenir d'autres nets.

Dans la Figure 20, deux modules A et B communiquent entre eux via des ports (petits carrés) et un net. Ils contiennent tous les deux un sous-ensemble de modules, ports et nets. Dans A nous retrouvons deux modules représentant des tâches. Dans B les modules représentent des blocs existants dont on connaît uniquement les entrées et la façon de communiquer.

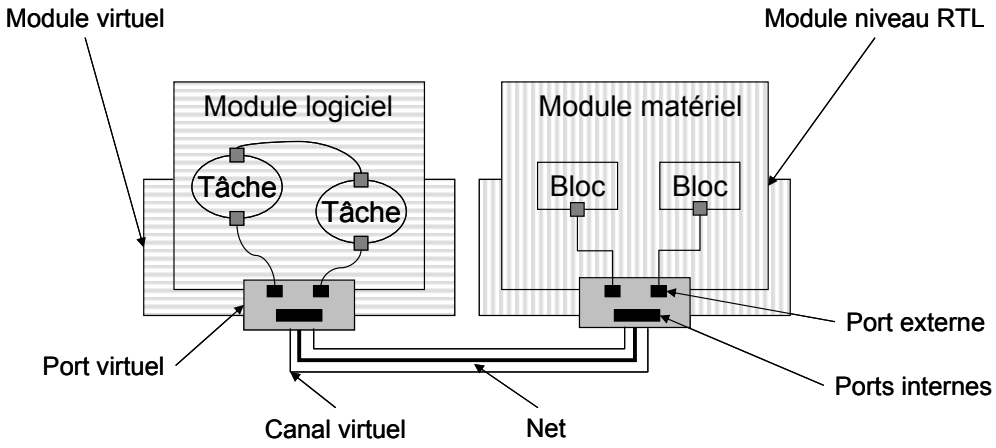


Figure 20 : Illustration de la représentation Colif d'un système

IV.B.2. Les différents niveaux d'abstraction

Colif permet pour l'instant de distinguer plusieurs niveaux d'abstraction. Dans ce travail, seulement deux niveaux sont utilisés :

Le niveau transactionnel est le niveau spécifique à la communication par des fils abstraits, englobant des protocoles de niveau pilote (driver). Un modèle de ce niveau implique par conséquent le choix d'un protocole de communication et de la topologie des interconnexions.

Le niveau RTL : la communication est réalisée par des fils et des bus physiques. La granularité de l'unité de temps devient le cycle d'horloge et les primitives de communication sont «set/reset » sur des ports et l'attente d'un nouveau cycle d'horloge.

IV.B.3. Composants virtuels et ports hiérarchiques

Pour permettre la connexion de composants de niveaux d'abstraction différents, Colif utilise le concept de composants virtuels.

Un composant virtuel possède deux interfaces de communication : une interne et une externe. L'interface interne est adaptée au niveau d'abstraction du module enveloppé. L'interface externe est adaptée au niveau d'abstraction des canaux connectés au module virtuel. Pour communiquer, l'interface interne possède des ports internes et l'interface externe des ports externes. Pour envelopper ces ports dans un même protocole de communication, on les rassemble dans un port hiérarchique appelé port virtuel. Un port virtuel modélise un protocole. De la même façon deux ports virtuels sont connectés par des canaux virtuels : ce sont des canaux abstraits modélisant les protocoles utilisés.

IV.C. Présentation du générateur de système d'exploitation : ASOG

Cet outil a été développé par Lovic Gauthier dans le cadre de sa thèse. Ce paragraphe présente les concepts de base, les mécanismes, la structure et la composition de la bibliothèque. Enfin le flot de génération automatique d'un système d'exploitation sera décrit en détail avec la présentation des entrées et sorties ainsi que les principales étapes de la génération.

IV.C.1. Concept : le ciblage logiciel

Le ciblage logiciel consiste à produire une description exécutable (pour une architecture donnée) d'une application logicielle, à partir d'une ou plusieurs descriptions de haut niveau de cette dernière, ne contenant pas de détails d'implémentation. L'outil de génération automatique d'un système d'exploitation permet un ciblage logiciel puisqu'il fourni la couche logicielle permettant l'exécution de l'application sur l'architecture cible.

IV.C.2. Les mécanismes d'assemblage

La génération du code du système d'exploitation consiste en l'assemblage et en la paramétrisation de morceaux de code d'une bibliothèque. Les morceaux de code sont encapsulés dans des macros. Leur expansion avec des paramètres adéquats permet la génération du code du système d'exploitation. Seuls les éléments nécessaires à l'application sont sélectionnés pour être expansés.

IV.C.3. Structure de la bibliothèque

La bibliothèque est composée de deux parties : un ensemble de fichiers en langage macro et un fichier de description des dépendances entre les éléments.

IV.C.3.a. Les objets et la description de leurs dépendances

Le langage de description de dépendance utilisé a été développé au sein du groupe. C'est un langage de description structurelle nommé LiDeL (pour « Library Description Langage »). Il est composé d'un ensemble de structures de donnée manipulées par des API.

Les concepts de base pour la description se basent sur trois concepts fondamentaux donnant chacun une vue différente du système d'exploitation : l'élément, le service et l'implémentation.

L'**élément** représente une partie de l'OS, c'est une brique de base. Cependant cette notion ne correspond pas directement à un bout de code généré. Ce terme désigne un type de composant non spécialisé (implémentation) au sens ou il n'est pas forcement dédié à une architecture particulière.

Le **service** représente une fonctionnalité du système. C'est une notion abstraite qui permet de diviser et de structurer le comportement fonctionnel de l'OS. Les services sont fournis par des éléments, mais un élément peut aussi requérir un service.

L'**implémentation** représente une réalisation particulière d'un comportement. Un élément peut posséder plusieurs implémentations. Les implémentations désignent des objets concrets : à chaque implémentation correspond une portion de code générique de l'OS.

La description de la bibliothèque est composée d'un ensemble d'objets. Ils correspondent à différents types de structures de données différenciées par le type d'information qu'elles apportent. Les trois concepts présentés précédemment constituent les principaux objets de la bibliothèque.

IV.C.3.b. Les éléments de macro

Les éléments de macro sont des fichiers écrits en langage de macro. Une fois assemblés et expansés, ils forment le code de l'OS.

Le langage utilisé est appelé Rive, c'est un langage développé par Lovic Gauthier[51]. Il offre à peu près le même pouvoir d'expression que le langage m4 [18]. Il a été préféré à ce dernier car il est plus facile à écrire. Ce langage s'accompagne d'un outil du même nom qui prend en entrée des fichiers texte écrits en Rive, des paramètres mais aussi des macros. Le

fichier de sortie est un fichier texte. Ce type de langage et d'outils permet d'écrire n'importe quel type de code puisqu'il se contente d'éditer du texte. L'avantage de la description d'élément en macro est d'avoir des éléments génériques : selon les paramètres utilisés, le code expansé ne sera pas le même.

IV.C.4. Le flot de génération automatique de systèmes d'exploitation

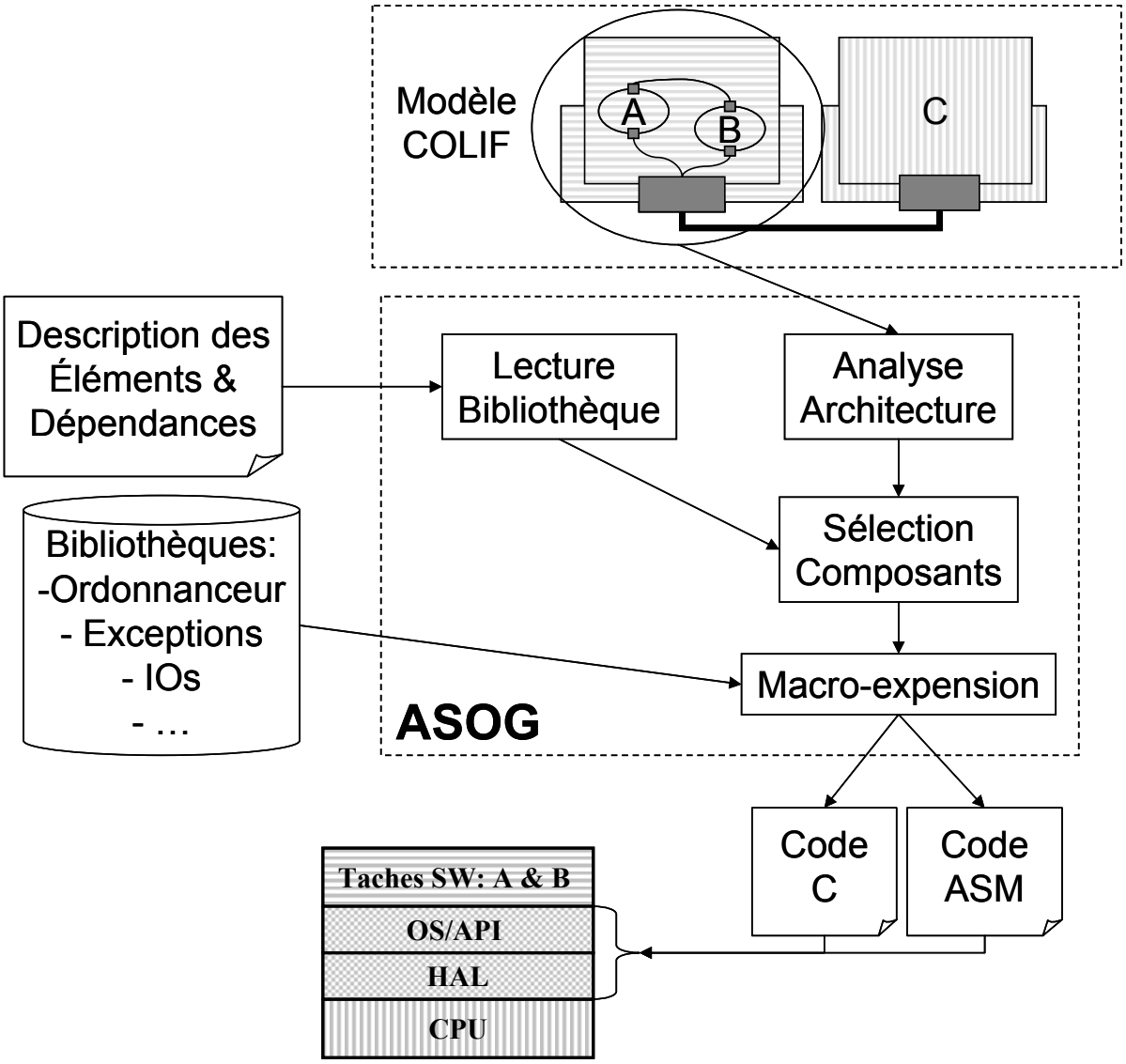


Figure 21 : Représentation du flot du générateur de systèmes d'exploitation : ASOG

Le flot prend en entrée une description du système en COLIF, le code des tâches, la description des dépendances de la bibliothèque en LiDeL (un langage développé en interne) et l'ensemble des éléments en langage de macro. La description COLIF contient les besoins de services pour l'application et tous les paramètres pour l'implémentation.

Le flot produit en sortie le code du système d'exploitation adapté à l'architecture et à l'application. Ce code est constitué de plusieurs fichiers pouvant être de divers langages comme le C le C++ ou les langages d'assemblage. Il produit aussi les fichiers de compilation : ce sont les fichiers qui permettent l'automatisation de la compilation. Ces fichiers sont dans le format Makefile, compréhensible par le programme d'automatisation de commandes make. Enfin, le flot produit en sortie des comptes rendus sur les opérations effectuées : liste des actions effectuées et erreurs survenues durant le ciblage.

V. Conclusion

Ce chapitre a présenté la méthodologie de validation globale qui a été proposée pour la vérification des systèmes sur puce. Elle se base sur la décomposition du système de telle manière que des méthodes spécifiques puissent être appliquées. Cette décomposition permet ainsi de bénéficier d'une méthode automatisée et appropriée pour la vérification des interfaces matérielles et du réseau de communication. En complément, une nouvelle méthode de vérification de l'intégration des systèmes monopuces a été proposée pour répondre aux limitations des méthodes actuelles. Afin de décrire cette méthode, la présentation de l'environnement de conception ROSES a été effectuée. Certains éléments comme le modèle de représentation des systèmes et le générateur de systèmes d'exploitation de l'environnement ROSES sont utilisés pour mettre en œuvre la méthode de vérification de l'intégration.

Chapitre 4 : PROPOSITION D'UNE METHODE DE VERIFICATION DE L'INTEGRATION POUR LES SYSTEMES MONOPUCES

I. INTRODUCTION 84

II. PRESENTATION DE LA METHODE DE VERIFICATION DE L'INTEGRATION 84

III. PROPOSITION D'UNE METHODE DE GENERATION DES PROGRAMMES DE TEST LOGICIEL DE HAUT NIVEAU 86

III.A. LA MODELISATION DES COMPOSANTS ET LE CONCEPT DE SERVICES OFFERTS ET SERVICES REQUIS 88
III.B. LA MODELISATION DU SYSTEME 90
III.B.1. Présentation de la modélisation des composants 91
III.B.2. Présentation des éléments de description d'un composant 92
III.B.3. Présentation de la structure du modèle d'un système 95
III.B.4. Résumé des points clé de la modélisation 97
III.C. LES PROGRAMMES DE TEST LOGICIEL DE HAUT NIVEAU 98
III.D. LA GENERATION DES PROGRAMMES DE TEST LOGICIEL 102
III.D.1. Les programmes de test de bas niveau pour le test des registres 104
III.D.2. Les programmes de test de haut niveau 104

IV. LE RAFFINEMENT DES PROGRAMMES DE TEST LOGICIEL DE HAUT NIVEAU : LE GENERATEUR DE SYSTEME D'EXPLOITATION 107

IV.A. LES PROGRAMMES DE TEST DE HAUT NIVEAU ET LA SPECIFICATION DES PARAMETRES DE MODELISATION DE LA PARTIE LOGICIELLE ET DE L'API 109
IV.B. LA BIBLIOTHEQUE DES ELEMENTS DU SYSTEME D'EXPLOITATION 111
IV.C. LA GENERATION AUTOMATIQUE DU SYSTEME D'EXPLOITATION ET LES PROGRAMMES DE TEST DE BAS NIVEAU 115

V. CONCLUSION 116

I. Introduction

Ce chapitre présente la principale contribution de cette thèse. Cette contribution est la définition d'une méthode de vérification de l'intégration pour les systèmes monopuces. Ainsi dans un premier temps, ce chapitre présente le flot complet de cette méthode. Ensuite, les deux parties de ce flot sont décrites :

- La génération des programmes de test logiciel de haut niveau ;
- Le raffinement des programmes de test logiciel de haut niveau.

Cette approche permet, de par l'utilisation de programmes de test logiciel de haut niveau, de proposer une solution à la synchronisation logicielle / matérielle. C'est un système d'exploitation spécifique qui autorise la mise en place de ces mécanismes de synchronisation.

II. Présentation de la méthode de vérification de l'intégration

Cette section décrit une nouvelle méthode pour la validation des SoCs présentée sur la Figure 22. Elle s'appuie sur une hypothèse forte : les éléments programmables (CPU, DSP) contenus dans le système sont supposés déjà vérifiés. D'autre part, ce sont ces éléments programmables qui vont permettre d'exercer, de stimuler le système. De même, la vérification des blocs matériels (IP HW) est supposée réalisée. La vérification des SoCs est ainsi appelée vérification de l'intégration : elle correspond à la vérification des interactions entre les différents composants du système et de leurs interconnexions.

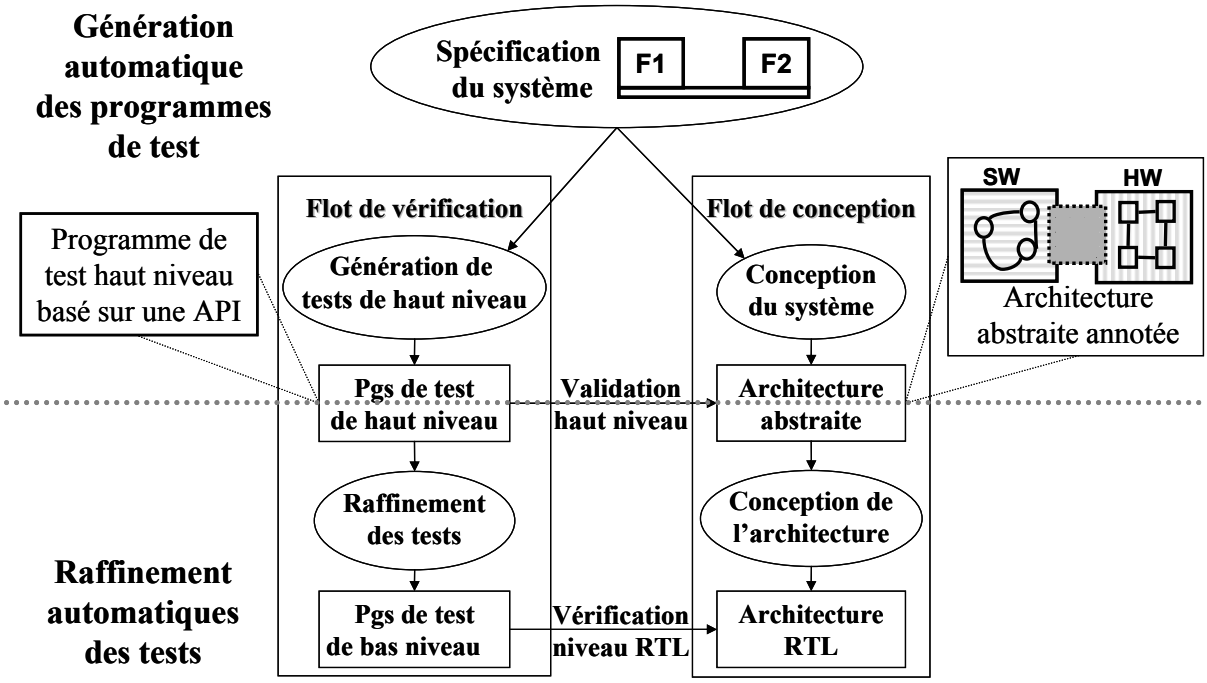


Figure 22: Présentation du flot de la méthode de vérification proposée et illustrée sur le flot de conception et de vérification idéal

La Figure 22 présente le flot de vérification proposé dans le contexte du flot de conception et de vérification idéal. Ce flot commence par la spécification du système, à partir de laquelle débute en parallèle le flot de conception et de vérification. Ainsi, pendant la conception d'un modèle de haut niveau (par ex : niveau TLM pour Transaction Level Modeling), il y a la génération de programmes de test logiciel de haut niveau. Ces programmes de test de haut niveau permettent la validation de cette architecture abstraite. Cette étape permet le développement en avance de phase de la suite de tests sur une plateforme de haut niveau, avec l'avantage d'avoir des temps de simulation plus rapides que lorsqu'elle est réalisée au niveau RTL. Dans un deuxième temps, l'architecture abstraite est raffinée pour obtenir l'architecture RTL. A ce niveau du flot de conception, correspond le raffinement de programmes de test de haut niveau afin de réaliser la vérification de l'architecture RTL. De cette manière, il est possible d'effectuer la vérification du système tout au long de sa conception.

Finalement, le flot de vérification peut se décomposer en deux étapes fondamentales décrites dans la suite. Ces étapes sont:

1. La génération automatique des programmes de test logiciel de haut niveau basés sur une API (Application Programming Interface).
2. La génération automatique d'un système d'exploitation spécifique pour l'exécution des programmes de test logiciel de haut niveau : c'est la phase de raffinement des tests.

III. Proposition d'une méthode de génération des programmes de test logiciel de haut niveau

Cette section présente la méthode de génération des programmes de test logiciel de haut niveau. Cette partie du flot n'est actuellement pas réalisée, il s'agit d'une proposition de mise en œuvre, présentée plus en détails dans la Figure 23.

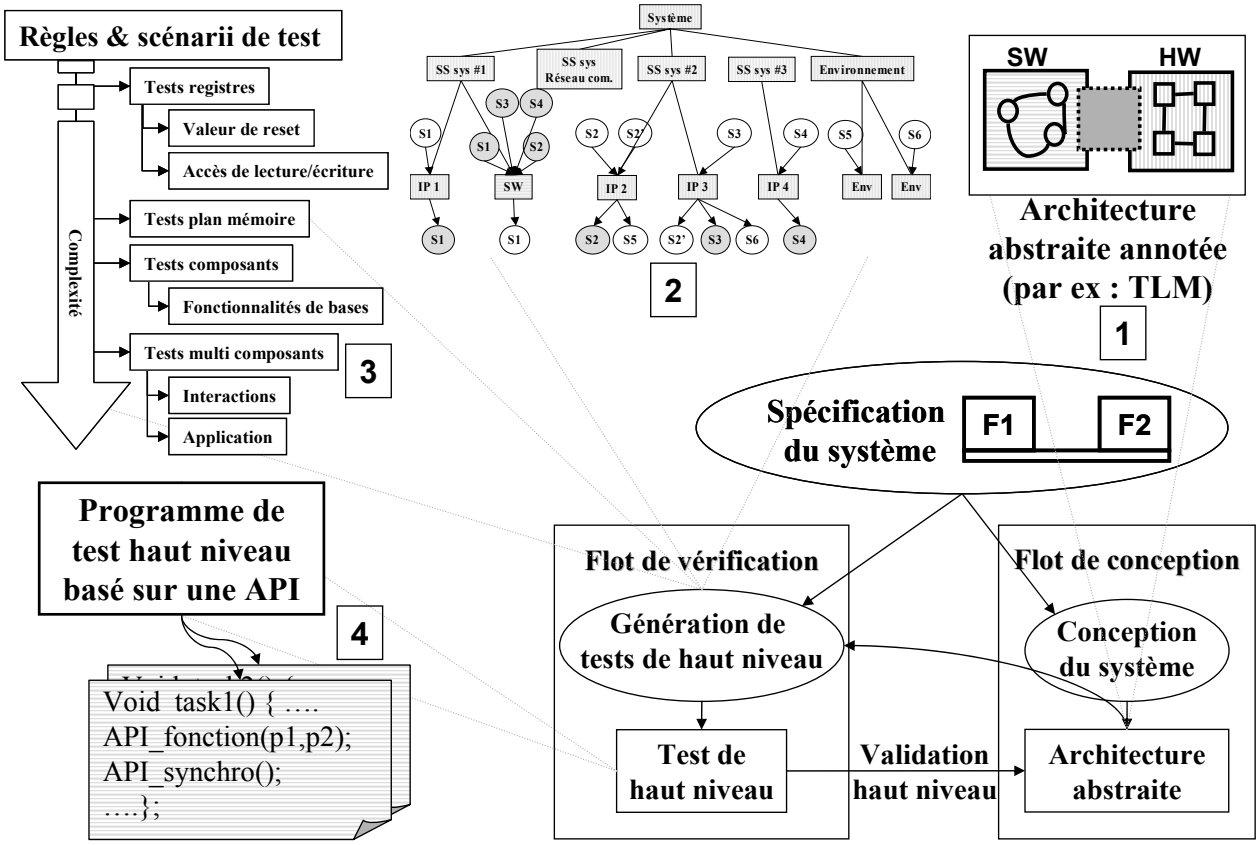


Figure 23 : Présentation du flot de génération des programmes de test logiciel de haut niveau basés sur une API.

Ce flot prend en entrée un modèle de haut niveau du système (l'architecture abstraite sur la figure) qui décrit par l'intermédiaire d'annotations les caractéristiques du système : plan mémoire, registres, fonctionnalités des composants, interactions… Le générateur construit à partir de l'architecture abstraite un modèle du système qui intègre la modélisation des fonctionnalités et de leurs dépendances par l'intermédiaire de services offerts et de services requis. Avec ce modèle du système et d'un ensemble de règles de test, le générateur de programmes de test logiciel va produire des programmes de test logiciel de haut niveau qui sont basés sur une API. Celle-ci repose sur un système d'exploitation et comprend un ensemble de fonctions qui permettent la configuration et le contrôle des différents composants du système.

Les éléments clé de ce flot sont les suivants :

- Un modèle de haut niveau (architecture abstraite : Figure 23 - 1) définissant le profil des programmes de tests ;
- Le modèle du système (Figure 23 - 2) construit à partir de l'architecture abstraite qui se décompose en trois parties :
    - La modélisation des fonctionnalités des composants et de leurs dépendances basées sur le concept de services offerts et de services requis ;
    - La modélisation des ressources, de la configuration et des entrées / sorties d'un composant ;
    - La modélisation de la structure du système.
- La génération des programmes de test de haut niveau réalisé par l'application de règles de test (Figure 23 - 3) ;
- Les programmes de test de haut niveau basés sur une API (Figure 23 - 4).

Ces différents éléments font l'objet d'une description détaillée dans les prochaines sections.

III.A. La modélisation des composants et le concept de services offerts et services requis

Avant de présenter la modélisation du système, cette section présente le concept de services offerts et de services requis. Ce concept est appliqué à la modélisation d'un système monopuce pour la vérification de l'intégration. Cependant, le concept de service n'est pas nouveau dans le cadre de la modélisation. Il a été défini pour la modélisation de la communication dans les applications réseaux [56]. Il est, en outre, utilisé au sein du groupe SLS pour réaliser la modélisation et permettre la génération des interfaces logicielles / matérielles [54][55].

De ce fait, il est important de préciser ce que représente un service dans notre cas. Un service représente une fonctionnalité d'un composant (ou d'un ensemble de composants), de telle sorte que la configuration des paramètres du composant (respectivement de l'ensemble des composants) puisse être déterminée. Un service est ainsi attaché à un élément et hérite de ces paramètres.

La modélisation d'un système, dans le cadre de sa vérification, par l'intermédiaire de service permet d'obtenir une représentation de l'ensemble des fonctionnalités qu'offre le système. En d'autre terme, cela représente les éléments devant être vérifiés. Dans cette modélisation, on distingue différents types de service. La première distinction se fait entre service offert et service requis :

- Un **service offert** représente une fonctionnalité que fourni le composant. Ce type de service caractérise ainsi les fonctionnalités du système qui sont à exercer. En d'autre terme, les services offerts peuvent être vus comme une représentation du plan de test d'un composant dans le système ;
- Un **service requis** représente une interaction. Il définit la dépendance du composant pour une fonctionnalité d'un autre composant. Ainsi, pour réaliser une fonctionnalité, un composant peut nécessiter une fonctionnalité qui est offerte par un autre composant.

De plus, pour réaliser la représentation des fonctionnalités, nous introduisons dans la modélisation des services la notion de hiérarchie. Cette notion est importante, car elle permet de représenter sous la forme d'un arbre de services l'ensemble des fonctionnalités d'un composant. Cette modélisation hiérarchique des services correspond au groupement des

fonctionnalités de telle manière que la valeur des paramètres soit définie hiérarchiquement. Ainsi pour chaque nœud de l'arbre certains paramètres sont définis puis hérités par les fils. En d'autre terme, les feuilles de l'arbre des services définissent l'ensemble des paramètres du composant.

La Figure 24 permet d'illustrer la modélisation du concept de services offerts et de services requis (a). Elle présente également un exemple de modélisation d'un composant issue du système PIAGET (b). Ce composant fait l'objet de l'étude de cas présentée dans le chapitre 5. Ce composant (HDPP : High Definition Pixel Port) réalise l'analyse de différents types de flux vidéo numérique, la modélisation donnée ici est simplifiée, elle ne couvre pas l'ensemble des fonctionnalités.

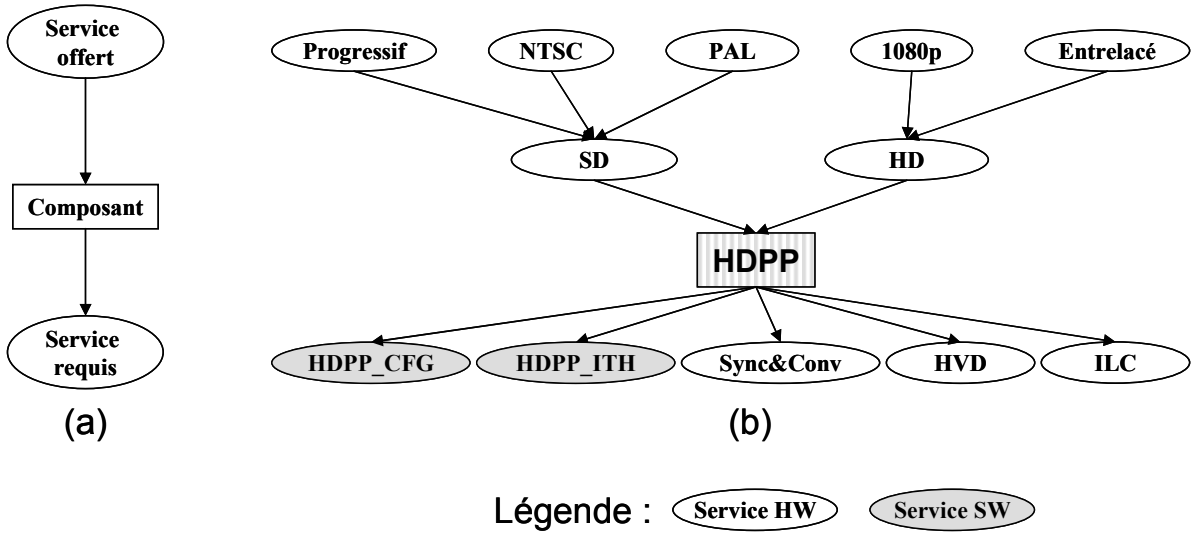


Figure 24 : Représentation de la modélisation des services (a) et illustration par un exemple d'un composant issue du système PIAGET (b)

Ce composant offre différents services, il est ainsi capable d'analyser des flux vidéo numérisés dans des formats HD (Haute Définition) et SD (Définition Standard). Cette distinction est la première étape dans la spécification des services et des paramètres de configuration. Chacun des deux types de flux vidéo est ensuite divisé suivant les formats supportés et nécessitent une configuration différente des paramètres du composant. Cette étape est réalisée par la hiérarchie des services. Des services fils sont donc attachés aux services HD et SD, par exemple dans le cas du HDPP, le service SD possède les services NTSC, PAL, progressif en tant que fils. Pour chacun de ces formats, certains paramètres du composant sont différents. On remarque également que ce composant requiert plusieurs

services pour accomplir sa fonction. Deux d'entre eux sont des services logiciels correspondant à des fonctions de l'API qui permettent la configuration du composant et la gestion du traitement de ses interruptions. Les deux autres services sont eux de types matériels et représentent les interactions avec d'autres composants.

Il est de ce fait possible de classer les services selon qu'ils soient logiciels ou matériels. Cette distinction est importante, car dans notre méthode, nous utilisons le(s) processeur(s) du système pour effectuer la vérification. Ainsi, lorsque l'on substitue le logiciel applicatif du système par du logiciel de test pour exercer le système, les services logiciels forment une API de vérification qui permet de configurer et de contrôler les différentes fonctionnalités du système. Enfin, dans le cas des systèmes multiprocesseurs, on peut souhaiter ne pas remplacer certaines parties logicielles du système. Celles-ci peuvent alors être modélisées de la même manière qu'un composant matériel.

Mais la description des fonctionnalités sous la forme de services n'est pas suffisante. Il est également nécessaire comme cela vient d'être montré de définir un ensemble de paramètres pour compléter la modélisation d'un composant. Cet ensemble de paramètres permet d'abstraire les entrées/sorties et la configuration d'un composant. La description de ces paramètres est donnée dans la section suivante qui décrit la modélisation du système.

III.B. La modélisation du système

La modélisation du système s'appuie sur un modèle orienté objet qui peut être réalisé aisément à partir de Colif [50] présenté dans le chapitre 3. La modélisation d'un système se décompose ainsi en trois parties décrites dans la suite et qui sont :

- La modélisation des composants ;
- La modélisation des ressources, de la configuration et des entrées / sorties d'un composant ;
- La modélisation de la structure du système.

III.B.1. Présentation de la modélisation des composants

La modélisation d'un système sur puce requiert d'une part la capacité de modéliser les parties matérielles, mais aussi les parties logicielles. Cependant, comme cela a déjà été mentionné, la méthode de vérification de l'intégration qui est présentée ici est basée sur l'utilisation des processeurs (CPU, DSP) du système pour effectuer la vérification. En d'autre terme, la modélisation du logiciel intègre une partie spécifique pour la modélisation des programmes de test de haut niveau. La Figure 25 donne une représentation de la modélisation des composants matériels et logiciels.

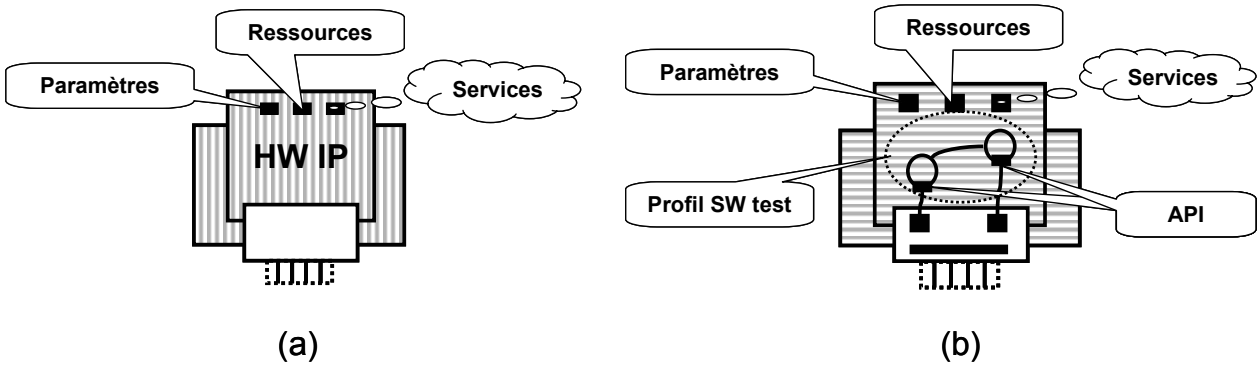


Figure 25 : Présentation de la modélisation d'un composant matériel (a) et d'un composant logiciel (b)

Comme le montre la Figure 25, le modèle d'un composant matériel et celui d'un composant logiciel possèdent des éléments de modélisation partagés : services, paramètres, ressources. En addition on trouve pour un composant logiciel deux nouveaux éléments qui sont spécifiques à la vérification. Ils permettent la définition des éléments nécessaires d'une part à la génération des programmes de test de haut niveau et d'autre part à leurs raffinements. Voici, pour le matériel et le logiciel, plus de détails sur ces éléments :

- Le modèle d'un composant matériel comprend :
    - Une description des ressources du composant : les registres ;
    - Un ensemble de paramètres se décomposant en deux types :

        - Les paramètres de modélisation du composant. Ces paramètres réalisent l'abstraction des données d'entrées/sorties et définissent ainsi les données échangées lors des interactions avec d'autres composants ;
        - Les paramètres de configuration du composant. Ces paramètres permettent de définir la configuration du composant et sont directement reliés aux paramètres des fonctions de l'API de vérification ;
    - La modélisation des fonctionnalités et des interactions du composant par la spécification de services, comme montré dans la section précédente.
- Le modèle d'un composant logiciel comprend les mêmes éléments que le modèle matériel ci-dessus et en plus des éléments spécifiques à la vérification :
    - *La définition de l'API de vérification utilisée pour construire les programmes de test et pour réaliser la génération du système d'exploitation ;*
    - *Un modèle du profil des programmes de test qui est substitué au logiciel applicatif pour la vérification. Cette partie est présentée plus en détail dans la section décrivant les programmes de test de haut niveau.*

La modélisation permet ainsi pour les composants logiciels de réaliser ou non la substitution du logiciel applicatif par des programmes de test. Ces représentations utilisées pour décrire les composants matériels et logiciels sont les briques de base pour construction du modèle du système.

III.B.2. Présentation des éléments de description d'un composant

Pour réaliser la modélisation d'un composant en vue d'en réaliser la vérification par la méthode présentée ici, il est nécessaire de décrire les trois types d'éléments (ressources, entrées /sorties et configuration) qui sont utilisés.

III.B.2.a. La définition des ressources d'un composant ou la description des registres

Le premier type de paramètres qui est utilisé pour caractériser un composant est celui décrivant les ressources du composant. La définition des ressources correspond à la définition des registres du composant.

Cette spécification des ressources du composant réalise la définition pour chaque registre :

- Du nom et de l'adresse ;
- Des champs et du type d'accès (lecture / écriture).

Un tel format a déjà été développé au sein de l'équipe TV. Une représentation de ce format de description est donnée sous la forme d'une pseudo grammaire sur la Figure 26. A partir de cette description, un outil a été développé et permet la génération de programme de test de bas niveau permettant de vérifier tous les types d'accès aux registres. Ce formalisme peut être intégré directement dans le cadre de notre modélisation, ainsi que la génération de ce type de test.

AXIOME → REGISTER_DEF
REGISTER_DEF → **Register** "nom_registre " SIZE_DEF
SIZE_DEF → **Size** " taille " OFFSET_DEF
OFFSET_DEF → **Offset** " offset " BITFIELD_DEF
BITFIELD_DEF → **Bitfield** BITFIELD_TYPE
BITFIELD_TYPE → {" nom_champ " | - }:"bit_index" :"bit_index"
{BITFIELD_DEF * | INITVAL_DEF}
INITVAL_DEF → **Initialvalue** " valeur_reset " ACCESS_DEF
ACCESS_DEF → **Accesstype** ACCESS_TYPE
ACCESS_TYPE → { **RO** | **RW** } REGISTER_DEF *

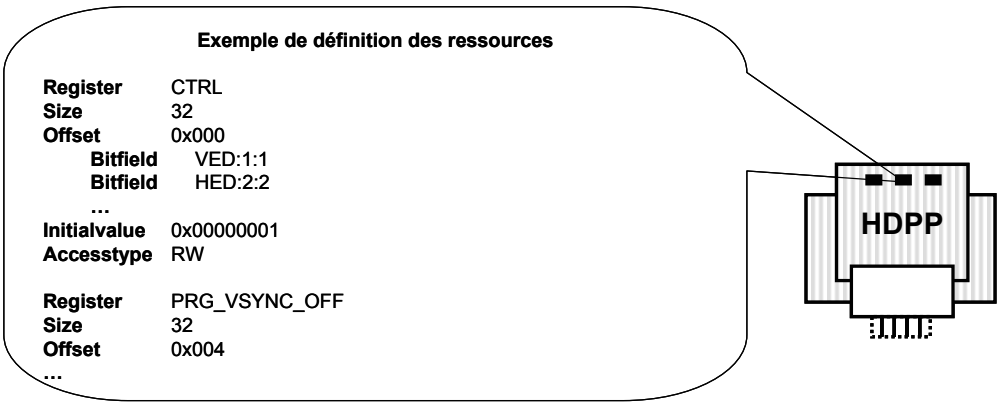


Figure 26 : Présentation de la définition des registres sous la forme d'une grammaire et exemple de description

III.B.2.b. Présentation des paramètres de modélisation d'un composant et leurs relations avec la modélisation des services.

En plus de la définition des registres d'un composant, il est nécessaire d'en modéliser les principales caractéristiques. Cette modélisation est réalisée par l'intermédiaire de paramètres qui abstraient le fonctionnement. Ces paramètres peuvent être divisés en deux ensembles :

- Les paramètres spécifiques au système ;
- Les paramètres spécifiques au composant.

Les paramètres relatifs au système permettent de définir si le composant possède une zone mémoire qui lui est allouée. Ces paramètres sont nécessaires pour vérifier la cohérence du plan mémoire spécifié pour le système. Ils sont génériques pour tous les composants. Il existe deux paramètres de ce type, le premier est noté « memory_map ». Il permet ainsi de spécifier l'espace mémoire accessible au composant. Un second paramètre est attaché à cette description, il est noté « memory_map_service ». Il permet de spécifier le ou les services qui mettent en œuvre les accès mémoires.

Les paramètres relatifs aux composants sont de deux types :

- Les paramètres d'entrés/sorties réalisent l'abstraction du type de données que manipule le composant ;
- Les paramètres de configuration définissent le mode de fonctionnement du composant.

Comme cela a été évoqué lors de la description des services, les services et les paramètres sont liés. Les paramètres spécifiés pour un composant sont ainsi hérités par les services. Au moment de la spécification des paramètres du composant, il n'est pas nécessaire de définir les valeurs respectives de chaque paramètre. Aussi, certaines valeurs de ces paramètres sont différentes voire non définies d'un service à l'autre. La seule contrainte de modélisation fixée ici est que tous les paramètres du composant soient définis pour les feuilles de l'arbre de service.

Ces deux types de paramètres permettent pour un service donné de définir le mode de fonctionnement du composant. Les paramètres de configuration sont ensuite utilisés dans les programmes de test de haut niveau. Ils constituent un ensemble des paramètres, utilisé lors de

l'appel des fonctions de l'API de vérification. Ils permettent au travers de l'API la configuration, la gestion et le contrôle du composant pendant le test.

Si l'on considère maintenant le composant HDPP utilisé pour l'illustration du concept de services, la Figure 27 présente un exemple de la modélisation de celui-ci par les paramètres et leurs relations avec la modélisation des services offerts. On notera que ce composant ne dispose pas d'un accès mémoire, d'où l'absence de paramètres spécifiques au système.

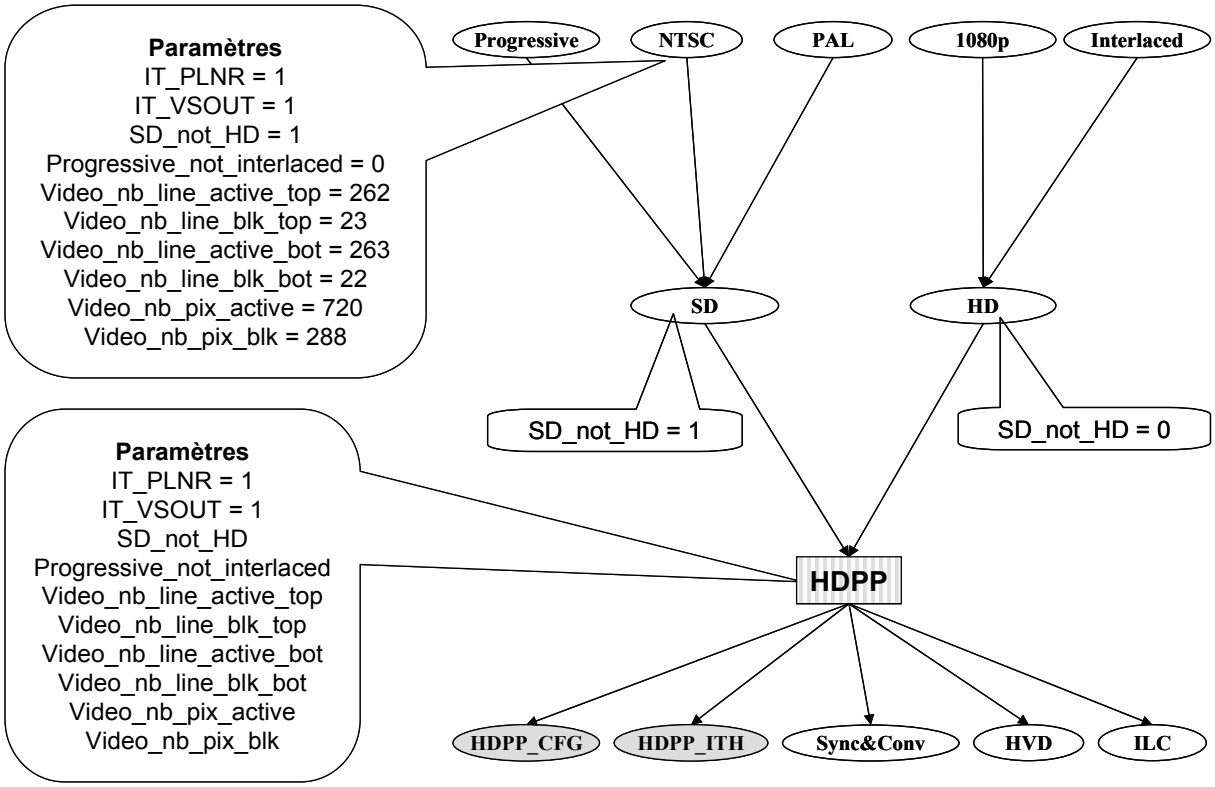


Figure 27 : Exemple de modélisation pour le composant HDPP des paramètres d'abstraction des entrées/sorties et de la configuration

Comme cela apparaît sur la Figure 27, les paramètres « IT_* » spécifient quelles interruptions du composant sont activées lors du test. Les autres paramètres constituent la définition du flux vidéo d'entrée et sont utilisés pour définir la configuration du composant.

III.B.3. Présentation de la structure du modèle d'un système

Il est maintenant nécessaire pour la vérification de capturer la structure du système. La principale caractéristique de cette modélisation est un modèle hiérarchique. Il permet d'effectuer le découpage et l'organisation du système en sous systèmes. Un sous système est un ensemble de composants qui réalisent ensemble une fonctionnalité du système.

La Figure 28 donne une illustration de la modélisation d'un système. Sur cet exemple, le système est composé de quatre sous systèmes :

- Un réseau de communication ;
- Un sous système (#1) avec un processeur (SW) et un composant matériel (IP 1) par exemple une mémoire ;
- Un sous système (#2) avec deux composants matériels (IP 2 & 3) réalisant un traitement quelconque ;
- Un sous système (#3) avec un composant matériel (IP 4) réalisant également un traitement quelconque.

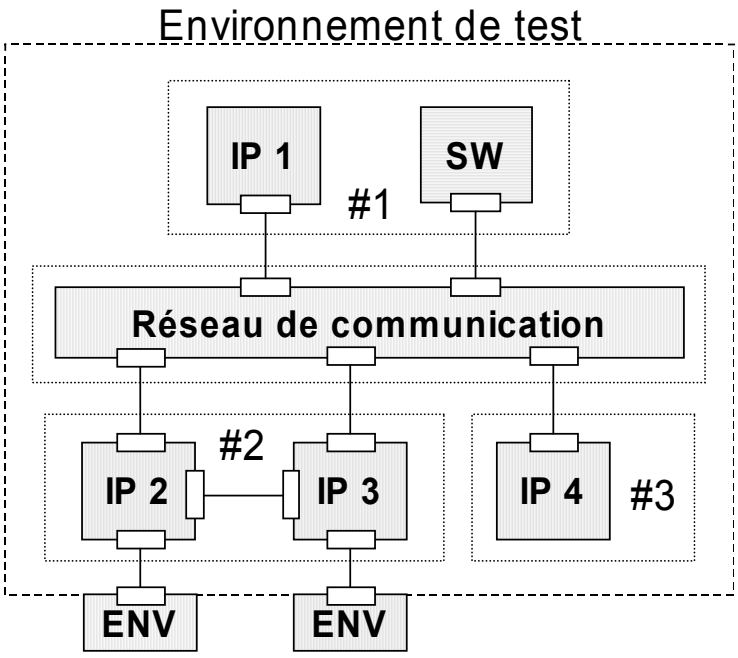


Figure 28 : Illustration de la modélisation d'un système

En outre, lors de la modélisation, nous représentons les blocs réalisant l'environnement de test. En effet, un composant du système disposant d'une interface externe peut requérir des services offerts par les éléments réalisants l'environnement du système. Dans cet exemple, la

représentation du système n'intègre pas la modélisation des paramètres ni celle des services qui sont représentés sur la Figure 29. Cependant, elle donne la représentation informatique d'une telle structure et un exemple de représentation de services symboliques. Elle montre la structure et les relations entre les composants par l'intermédiaire des services. Pour simplifier l'exemple, seuls les services attachés aux composants sont donnés. Un exemple concret est donné dans le chapitre 5.

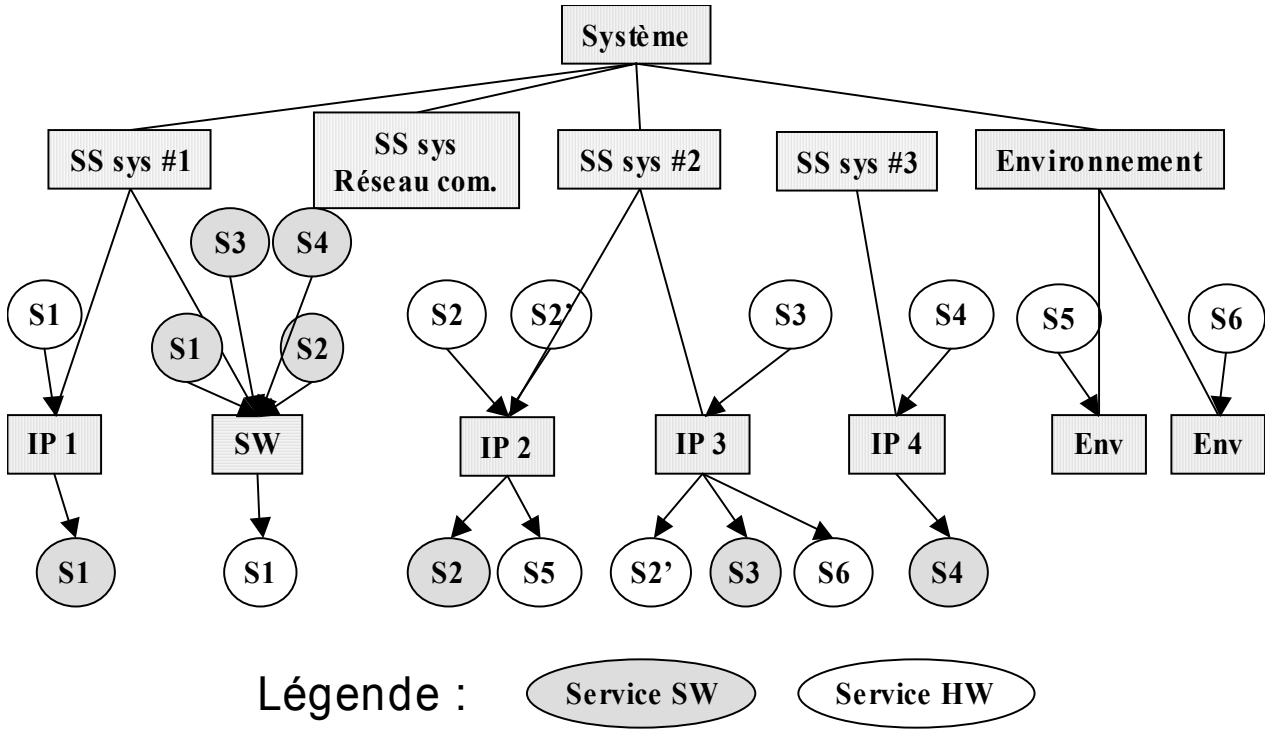


Figure 29 : Illustration de la représentation informatique de la structure du système de la Figure 28

III.B.4. Résumé des points clé de la modélisation

La modélisation hiérarchique du système et l'utilisation du concept de services pour représenter les fonctionnalités des composants permettent de réaliser la représentation des différents niveaux de vérification de l'intégration (composant, sous système, système). D'autre part, la modélisation des composants matériels et logiciels permet de définir:

- Les différents modes de fonctionnement du composant que l'on souhaite vérifier: en d'autre terme une représentation du plan de test pour chacun des composants du système ;
- Les interactions entre les composants par les dépendances de service ;
- Les ressources des composants par l'utilisation d'une architecture abstraite annotée.

Enfin la partie spécifique de modélisation des composants logiciels permet de définir pour les programmes de test :

- L'API de vérification utilisée pour réaliser la gestion et la configuration des composants du système. Cette API est construite sur un système d'exploitation ce qui permet d'abstraire la communication et le matériel ;
- La modélisation de la structure du programme logiciel de test. Nous appellerons cette modélisation le profil des programmes de test présenté dans la section suivante.

III.C. Les programmes de test logiciel de haut niveau

Un programme de test de haut niveau est un programme de test construit sur une API qui repose elle-même sur un système d'exploitation. Cette définition permet d'illustrer sur la Figure 30 la différence entre un programme de test de haut niveau et un programme de test de bas niveau.

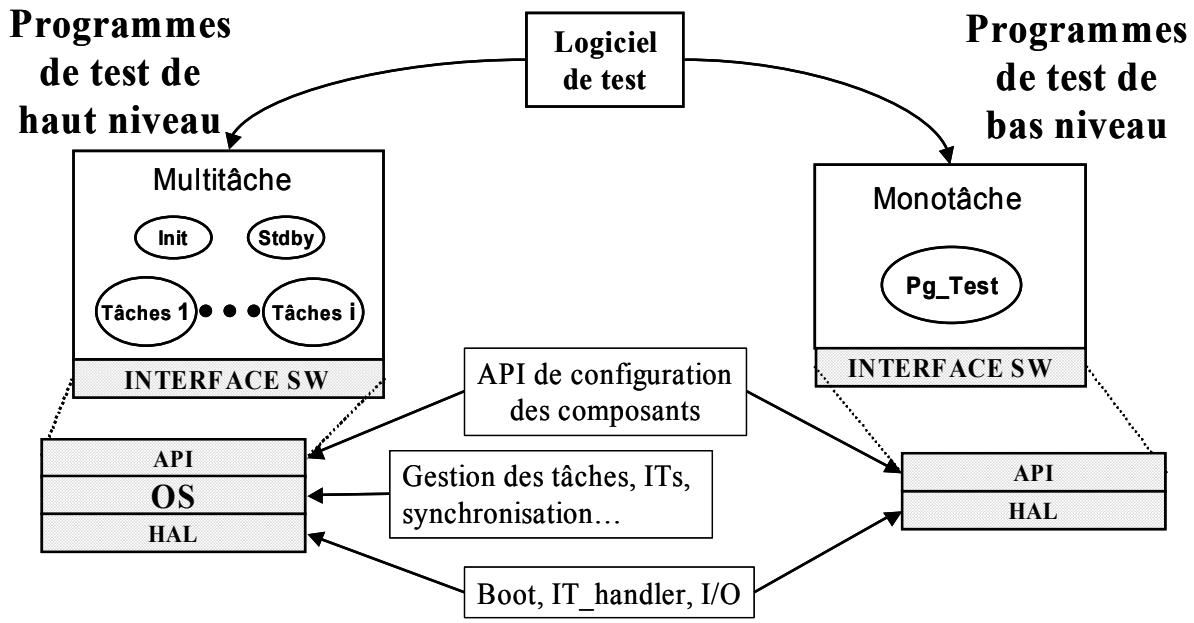


Figure 30 : Illustration de la différence entre un programme de test de haut niveau et un programme de test de bas niveau

Cette figure montre les éléments apportés par un programme de test de haut niveau :

- Un test peut être constitué de plusieurs tâches ;
- L'interface logicielle se compose d'un système d'exploitation.

Avec cette structure, un programme de test de haut niveau intègre, par l'abstraction de l'API, des mécanismes de synchronisation complexes. Ils permettent de mettre en œuvre, d'un point de vue du logiciel, la gestion du parallélisme du matériel. Il est par exemple possible de contrôler le parallélisme du matériel par la gestion des interruptions et de leurs priorités.

La Figure 31 (a) donne un exemple de la structure d'un programme de test permettant la mise en œuvre de la synchronisation. Cette structure sera détaillée dans la suite par la Figure 32. Cet exemple s'appuie sur le système ayant été utilisé pour la présentation de la modélisation (Figure 28 et Figure 29). Chacun de ces composants matériels s'exécute de manière concurrente et est synchronisé avec le logiciel par des interruptions. La gestion de ces composants est réalisée par l'attribution de priorité aux interruptions.

Cette méthode de description des programmes permet de simuler le parallélisme matériel. Pour montrer cette synchronisation, la Figure 31 (b) présente également un exemple d'un profil d'exécution du logiciel de test, les priorités étant fixées arbitrairement et définies

sur la figure. Au travers de cet exemple, on montre comment le système d'exploitation gère les interruptions et les niveaux de priorités. Sur la figure, le temps s'écoulant vers le bas, on observe plusieurs interruptions qui se produisent à des intervalles non fixes. Ces interruptions engendrent de la part du système d'exploitation l'ordonnancement des tâches. L'ordonnancement s'effectue suivant les priorités. Il permet ainsi de réaliser les opérations nécessaires au contrôle et à la gestion des différents composants. Par exemple, lorsque l'interruption IT1 du composant IP2 arrive, le système d'exploitation va démarrer l'exécution de la tâche T_IP2 contrôlant ce composant et ainsi de suite en fonction des interruptions détectées et des priorités.

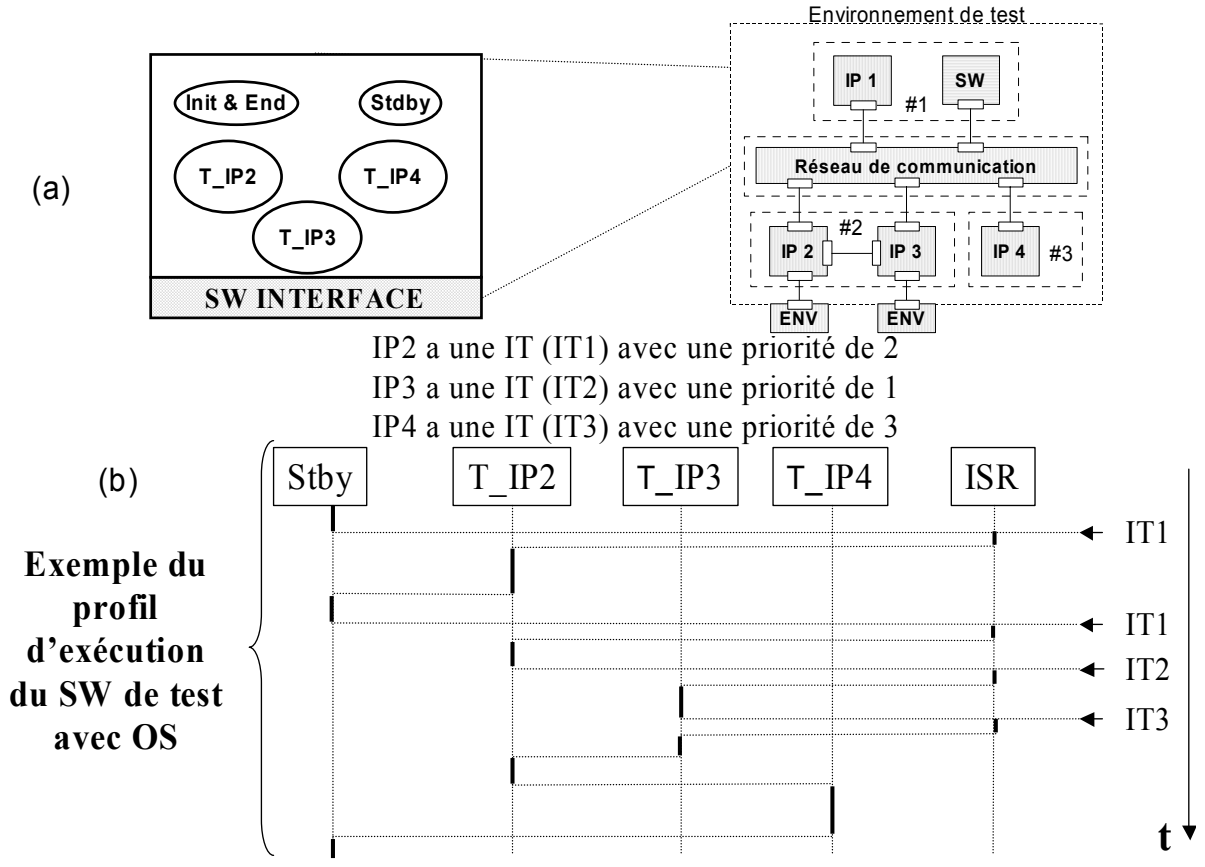


Figure 31 : Exemple de la structure d'un programme de test de haut niveau (a) et de la mise en œuvre la synchronisation par un exemple d'exécution du logiciel de test (b)

La génération des programmes de test de haut niveau nécessite donc la description du profil de ce logiciel de test. En effet, la génération des tests s'effectue par assemblage de tâches logicielles, elles-mêmes composées d'un assemblage d'appels de fonctions de l'API.

La spécification du profil des programmes de test consiste à réaliser le découpage des programmes de test en tâches logicielles. Chaque tâche est destinée à gérer et contrôler une partie du matériel utilisant le même mécanisme de synchronisation. Si on considère l'exemple précédent du système et des trois composants matériels, la figure détaille la modélisation du profil des programmes de test et donne un exemple du contenu d'une tâche. La modélisation employée ici se conforme au modèle Colif présenté dans le chapitre 3. Sur la Figure 32, le profil des programmes de test se compose de cinq tâches :

- Init & End : tâche réalisant l'initialisation du système et permettant de mettre fin au test ;
- T_IPx : tâches réalisant le contrôle et la gestion des composants IPx ;
- Stdby : tâche inactive du système.

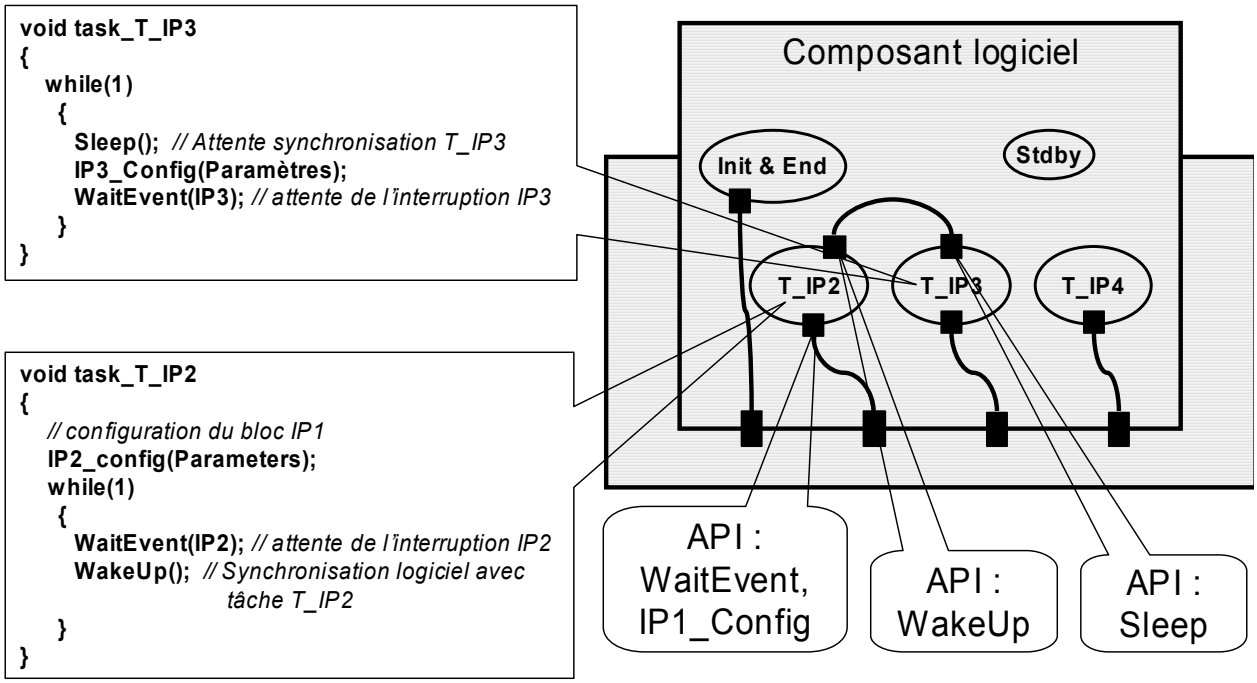


Figure 32 : Exemple de profil des programmes de test de haut niveau et de la structure d'une tâche

La Figure 32 donne une représentation de la modélisation des programmes de test de haut niveau. Sur cette figure on distingue plusieurs éléments :

- Le profil logiciel représente la structure des programmes de test de haut niveau qui est constituée d'un ensemble de tâches et qui définit la communication et la synchronisation.
- La spécification de l'API utilisée par chacune des tâches ;
- Un exemple du contenu des tâches. Ceci permet de montrer qu'une tâche se compose principalement d'appels de fonctions de l'API. Dans cet exemple on distingue deux types de fonctions : celle permettant la gestion des composants et celle réalisant les mécanismes de synchronisation.

III.D. La génération des programmes de test logiciel

Pour réaliser la génération des programmes de test, notre approche est systématique. Elle se base sur la définition de règles de test génériques. La définition des ce règles est réalisée de manière hiérarchique permettant ainsi une approche croissante de la complexité des tests. Cette hiérarchie est réalisée par le découpage des tests en quatre familles (Figure 33).

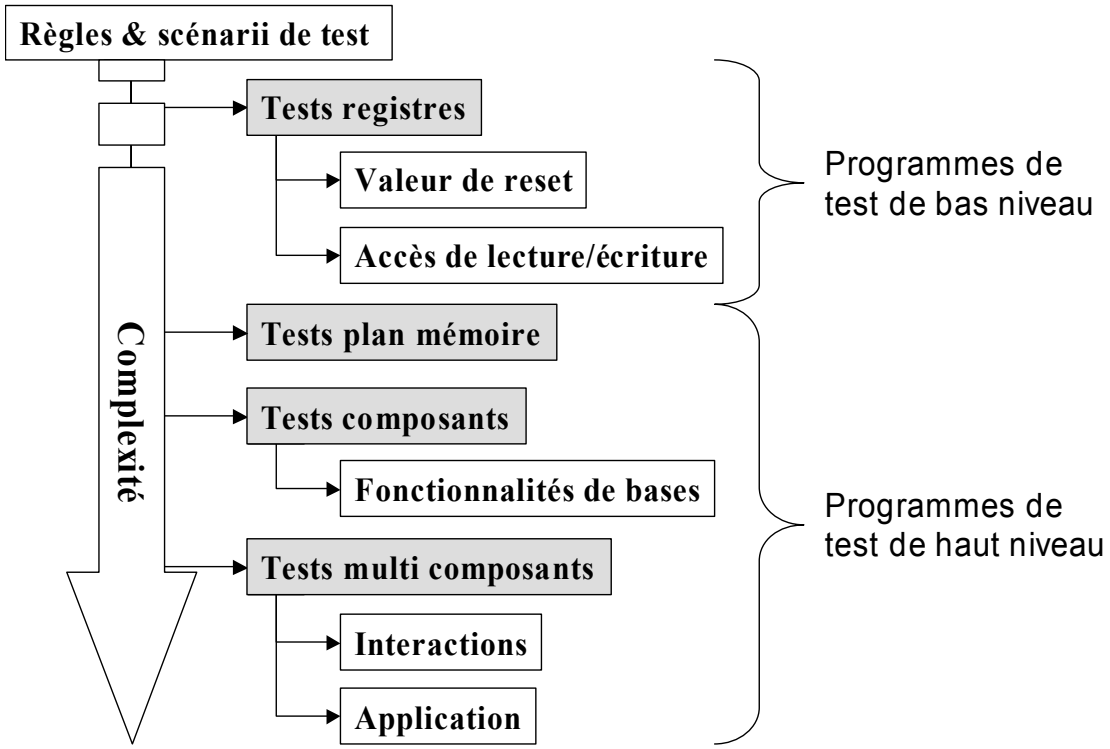


Figure 33 : Présentation des règles de test

Les quatre familles (parties grisées) de règles de test sont :

- Tests des registres ;
- Test du plan mémoire ;
- Tests des composants ;
- Tests multi-composants.

La représentation utilisée peut être vue comme une modélisation de haut niveau de la structure du plan de test. Les éléments nécessaires à la vérification de l'intégration des composants dans le système y sont adressés. Ces familles peuvent ensuite être divisées en deux ensembles suivant le type de programmes de test utilisé :

- Programmes de test de bas niveau :
  - Le test des registres ;
- Programmes de test de haut niveau :
  - Le test du plan mémoire ;

- Le test des fonctionnalités d'un composant ;
- Le test des interactions ou multi-composants.

Les règles de test sont appliquées dans l'ordre cité ci-dessus, cette manière de procéder permet une approche hiérarchique de la complexité des tests. Elle devient de ce fait systématique. Cependant, la vérification des registres ne requiert pas l'utilisation des programmes de test de haut niveau alors qu'ils sont utilisés pour les autres types de test. Cette distinction est importante pour la génération des tests. Les sections suivantes décrivent les différents tests générés suivant qu'ils sont de haut ou de bas niveau.

III.D.1. Les programmes de test de bas niveau pour le test des registres

La première étape de la vérification dans le flot proposé consiste à tester les registres de chaque composant du système. Dans cette étape est réalisée la génération de programmes de test permettant la première phase de la vérification de l'intégration des composants. Ces programmes ont pour but de vérifier les accès aux registres des composants. Les points ciblés sont :

- La vérification des valeurs de reset ;
- La vérification de l'adresse ;
- La vérification de tous les types d'accès (lecture, écriture…) aux différents champs des registres.

La génération de cette famille de programme de test s'effectue à partir de la description des ressources. Les programmes sont de bas niveau et réalisent séquentiellement des opérations de lecture et d'écriture dans les registres. Ces accès sont réalisés suivant la spécification des champs du registre et de leurs types d'accès.

III.D.2. Les programmes de test de haut niveau

Cette section détaille les trois autres familles de test. Celles-ci utilisent des programmes de test de haut niveau et ne se basent de ce fait plus sur le même mécanisme de génération. La

section décrit dans un premier temps les différentes parties de la vérification qui sont ciblées par chacune des familles :

- Test du plan mémoire ;
- Test des fonctionnalités d'un composant ;
- Test des interactions ou multi-composants.

Dans un deuxième temps, elle détaille le mécanisme de génération des programmes de test de haut niveau.

III.D.2.a. Le test du plan mémoire

La deuxième famille de règles de test consiste à vérifier la spécification du plan mémoire. Cette étape réalise, pour chaque composant possédant une zone mémoire allouée, la génération de programmes de test de telle manière que le composant effectue différents accès mémoires dans la zone allouée et en dehors. La génération de ce type de programme de test s'effectue à partir des paramètres « memory_map » et « memory_map_service ». Le premier paramètre spécifie la zone mémoire accessible et le second spécifie un service mettant en œuvre les accès mémoire du composant.

III.D.2.b. Le test des fonctionnalités d'un composant

La troisième famille de règles de tests cible la vérification de l'intégration des composants. Ces programmes de test permettent de mettre en œuvre les fonctionnalités du composant dans le système. Ils vérifient si l'intégration du composant dans le système n'impacte pas ses fonctionnalités. La génération de ces programmes de test est basée sur la modélisation des services offerts du composant ciblé. Ainsi pour chaque service offert par le composant, un programme de test de haut niveau est généré. Ce programme met ainsi en œuvre la fonctionnalité correspondant au service sélectionné.

III.D.2.c. Le test des interactions ou test multi-composants

La dernière famille de règles de test permet de considérer la vérification des interactions entre composant et/ou sous-système. Ce type de test s'approche des cas de fonctionnement réel du système. La génération de ces scénarii de test se base également sur des programmes de test de haut niveau. Elle est réalisée à partir de la spécification des services. Cependant, elle ne s'applique plus au service d'un seul composant mais aux services offerts par des sous-systèmes complets. C'est à ce niveau qu'intervient la hiérarchie de modélisation du système. En effet, un service peut être défini pour un sous système. Il permet alors de définir les services pour l'ensemble des composants du sous-système.

III.D.2.d. La génération des programmes de test de haut niveau

La génération des programmes de test de haut niveau débute par l'application d'une des règles de test présentées précédemment. La génération se base ensuite sur la modélisation du système et des services. La Figure 34 présente le flot associé à travers un exemple. La génération s'effectue en trois étapes:

1. **Sélection de la règle de test à appliquer**. Sur l'exemple, la règle appliquée cible la vérification de l'intégration d'un composant et fait partie de la famille « tests composants, fonctionnalités de base ». Ici, elle s'applique sur le composant IP 3 et cible le service S3 associé.
2. **Construction du modèle du système à partir de l'architecture abstraite annotée**. Le générateur détermine l'ensemble des composants intervenants dans le scénario de test. Cette étape est réalisée par l'analyse des dépendances de services qui sont représentées sur la figure par des flèches en pointillé. Cette étape permet également de déterminer l'ensemble des paramètres des composants intervenants dans le scénario. Dans l'exemple présenté, l'analyse débute au service S3.
3. **Configuration du profil du programme de test**. Il est défini par la modélisation du composant logiciel dans l'architecture abstraite. Cette étape permet de construire le profil en sélectionnant les tâches logicielles impliquées dans le scénario. La sélection des tâches est effectuée à partir de l'ensemble des

composants intervenants dans le scénario. La définition des fonctions de l'API utilisées s'effectue en plus de la sélection des tâches. La définition de l'API correspond à l'ensemble des services offerts par le logiciel de test.

En sortie du générateur, on obtient le profil du programme de test spécialisé pour le scénario induit par la règle de test. En plus, de ce profil, on obtient également le code des différentes tâches, elles aussi spécialisées pour le scénario.

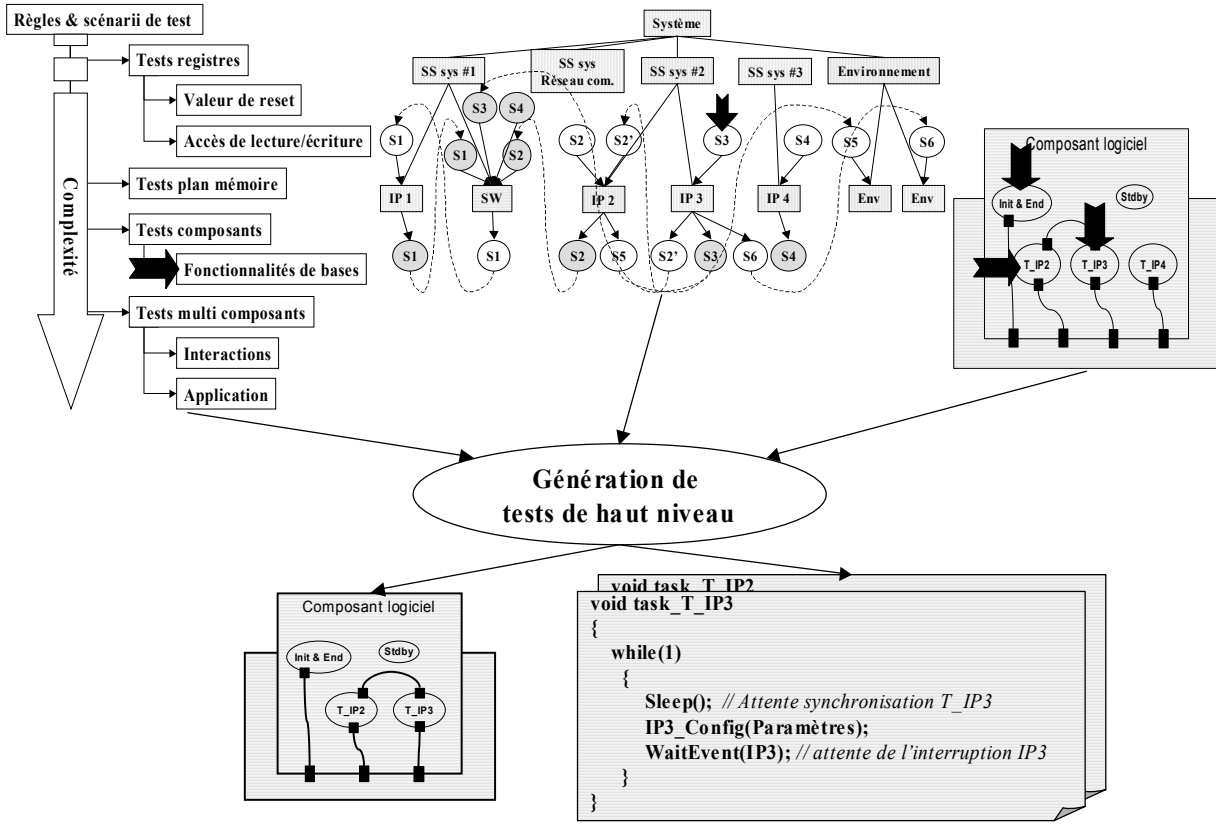


Figure 34 : Le flot de génération détaillé des programmes de test de haut niveau

IV. Le raffinement des programmes de test logiciel de haut niveau : le générateur de système d'exploitation

Cette section présente la seconde partie du flot de vérification proposé. Elle consiste à réaliser le raffinement des programmes de test logiciel de haut niveau basés sur une API. Cette étape est nécessaire pour permettre d'interfacer les programmes de test avec l'architecture RTL et ainsi réaliser la vérification de celle-ci.

Le raffinement de programmes de test logiciel de haut niveau est effectué par un générateur de système d'exploitation. Ce générateur s'appelle ASOG (décrit au chapitre 3). Il est issu des travaux réalisés par le groupe SLS du laboratoire TIMA.

La Figure 35 illustre le mécanisme de raffinement. Celui-ci nécessite la modélisation de la partie logicielle. Le modèle correspond, pour le générateur, à la spécification des paramètres de génération et à la définition de l'API. A partir de ce modèle et du code des différentes tâches constituant le programme de test, le générateur de système d'exploitation va produire les couches logicielles intermédiaires, nécessaires pour interfacer le programme de test avec l'architecture RTL.

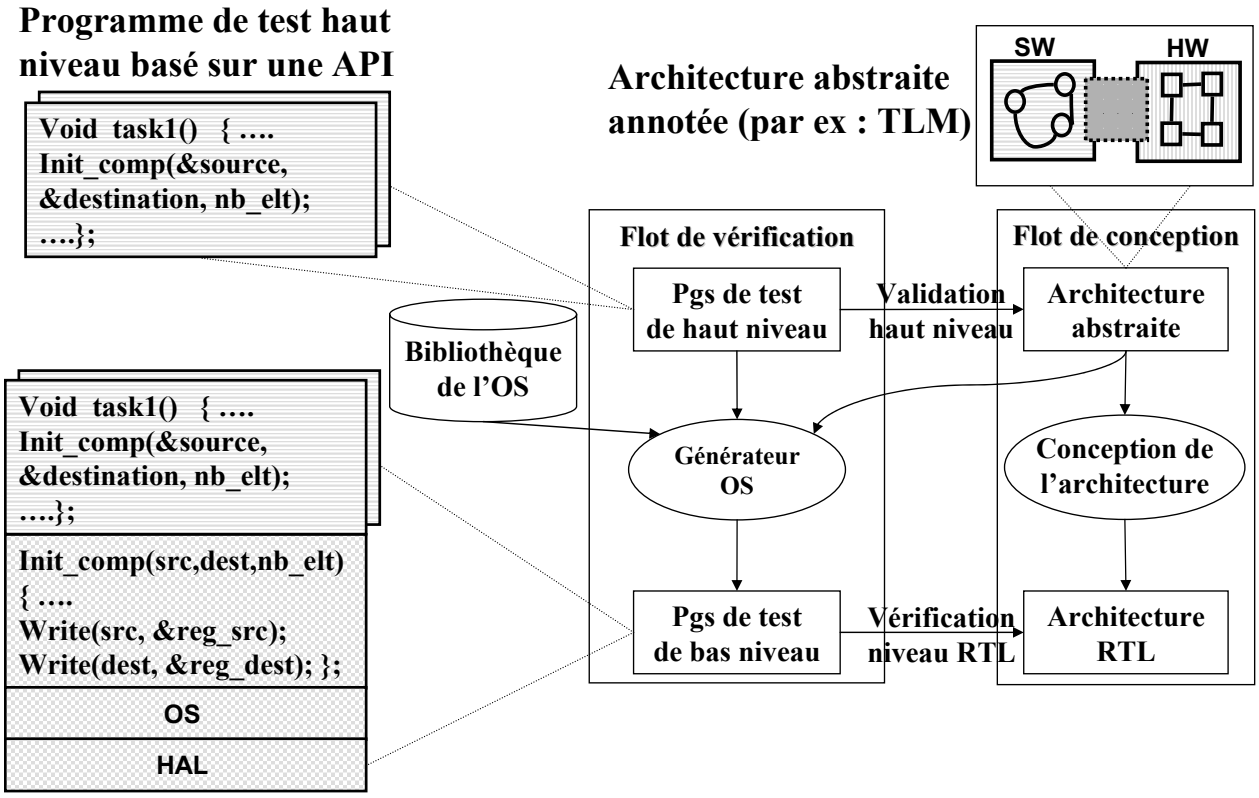


Figure 35 : Présentation du flot de raffinement des programmes de test logiciel de haut niveau

La Figure 35 présente ainsi les principaux éléments de ce flot qui sont :

- L'architecture abstraite contenant la modélisation du profil des programmes de test permettant de définir l'API et des paramètres de génération du système d'exploitation ;

- Le code des différentes tâches constituant le programme de test logiciels de haut niveau ;
- Un générateur de système d'exploitation spécifique (ASOG) ;
- La bibliothèque des éléments du système d'exploitation ;
- Les programmes de test de bas niveau.

La modélisation des programmes de test de haut niveau et le fonctionnement du générateur de système d'exploitation ayant déjà fait l'objet d'une présentation dans les chapitres précédents, cette section réalise la description des nouveaux éléments apparaissant dans le flot :

- La structure des programmes de test logiciel de haut niveau et de la spécification des paramètres de modélisation de la partie logicielle et l'API ;
- La bibliothèque des éléments du système d'exploitation. Ils constituent les briques de base utilisées lors de la génération du système d'exploitation ;
- La génération automatique du système d'exploitation et les programmes de test de bas niveau.

IV.A. Les programmes de test de haut niveau et la spécification des paramètres de modélisation de la partie logicielle et de l'API

La structure de représentation (profil) des programmes de test de haut niveau a été décrite précédemment. Cependant, pour compléter la modélisation, il faut préciser les paramètres permettant la génération du système d'exploitation. Ils sont présentés sur la Figure 36.

On retrouve la description du profil du logiciel de test et la structure d'une tâche. La figure présente également la spécification de l'API utilisée par le programme de test. Ces éléments ont déjà été décrits lors de la présentation du flot de génération des programmes de test de haut niveau.

Cependant, pour la génération du système d'exploitation, il est nécessaire de spécifier des paramètres de modélisation de la partie logicielle pour permettre le ciblage logiciel. Ces paramètres réalisent d'une part la définition de l'architecture cible :

- Type du processeur (AMR7, 68000, ST40, …).

Et d'autre part, la spécification des paramètres liés à la communication :

- Type de donnés (entier…)
- Taille du bus de donnée ;
- Numéro et priorité d'interruption ;
- Type du média de communication spécifiant le type de communication (par exemple : communication par un bus, communication par une FIFO matérielle, …)

Ces paramètres permettent de réaliser, lors de la génération, la sélection et la spécialisation des éléments de la bibliothèque pour produire un système d'exploitation spécifique. La bibliothèque est présentée dans la section suivante.

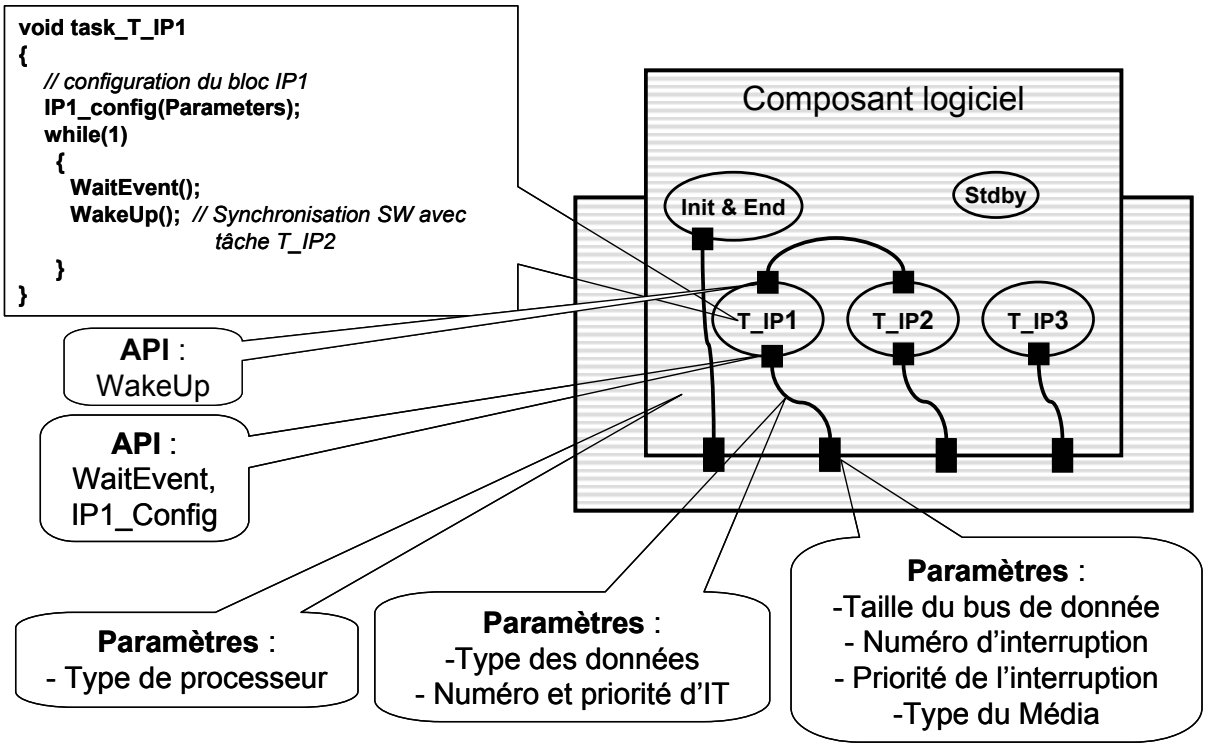


Figure 36 : Les programmes de test de haut niveau et la spécification de l'API et des paramètres de modélisation de la partie logicielle pour la génération du système d'exploitation.

IV.B. La bibliothèque des éléments du système d'exploitation

La génération de systèmes d'exploitation spécifiques s'effectue comme nous l'avons vu par la spécialisation et l'assemblage d'éléments de la bibliothèque. De ce fait, pour expliquer le raffinement des programmes de test de haut niveau, il est nécessaire de décrire la bibliothèque des éléments du système d'exploitation, élément central du flot de vérification. Elle intègre non seulement l'API de vérification définie pour réaliser le contrôle et la gestion des composants matériels du système, mais aussi l'API de communication et l'API de l'OS qui fournit des mécanismes de synchronisation.

Initialement, le générateur ASOG permettait le raffinement de la partie logicielle d'un système. La différence par rapport à la spécification des programmes de test est que la partie logicielle reposait uniquement sur une API de communication et de synchronisation. La Figure 37 présente la composition de la bibliothèque initiale. La bibliothèque est représentée ici par les services et les éléments qui la composent et définit les dépendances. Pour simplifier, l'intégralité de la bibliothèque n'est pas représentée.

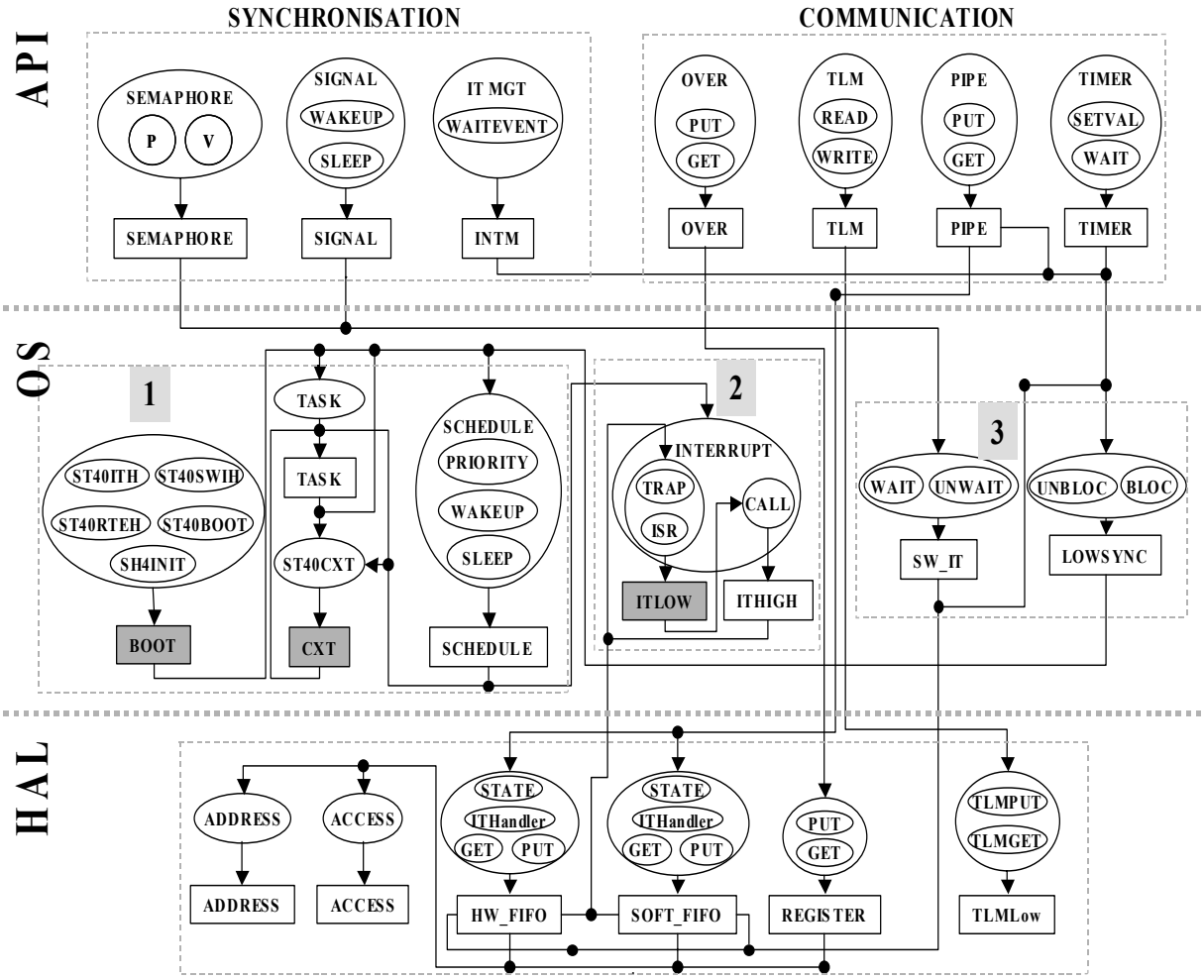


Figure 37 : La bibliothèque des éléments du système d'exploitation initiale

La Figure 37 présente un exemple de la structure de la bibliothèque des éléments du système d'exploitation. Dans cette figure, les éléments sont symbolisés par des rectangles, et les services par des formes arrondies. La hiérarchie des services est représentée par des formes arrondies imbriquées. Chaque flèche allant d'un service vers un élément indique que le service est fourni par l'élément. Chaque flèche allant d'un élément vers un service indique que l'élément requiert le service. En outre, les conventions suivantes ont été prises pour simplifier la représentation de la bibliothèque :

- Lorsque plusieurs flèches ont la même origine ou arrivent à la même destination, elles peuvent être connectées entre elles (la connexion est symbolisée par un rond noir) ;
- Lorsqu'une flèche part d'un ensemble de services ou le rejoint, elle est équivalente à plusieurs flèches rejoignant chacun des sous-services.

Finalement, cette bibliothèque peut se diviser en trois couches :

- La couche **HAL**. Elle réalise l'abstraction de bas niveau de la communication. Dans cette couche, on peut également inclure les éléments grisés du système d'exploitation qui sont spécifiques à l'architecture du processeur.
- La couche **OS**. Elle intègre les fonctionnalités du système d'exploitation que l'on peut diviser en trois parties :
    1. Le noyau du système qui comprend la définition des tâches et l'ordonnancement ;
    2. La gestion des interruptions ;
    3. La synchronisation.
- La couche **API** comprend deux parties : l'API de **communication** et l'API de **synchronisation**, utilisées pour construire le logiciel applicatif. Celles-ci sont décrites par les services qu'elles offrent et les éléments les réalisant. Parmi les éléments de l'API de communication, on notera que l'élément « PIPE » peut être implémenté par différents éléments suivant le type de média de communication : FIFO matérielle ou logicielle. D'autre part, l'élément « TLM » défini une communication par simple lecture/écriture sur un bus. Concernant l'API de synchronisation, outre les services classiques (sémaphore, signaux), le service « IT MGT » fourni par l'élément INTM permet de réaliser l'attente d'une interruption matérielle.

Comme nous l'avons vu, la méthode de vérification qui est proposée repose sur des programmes de test de haut niveau eux-mêmes basés sur une API de vérification. L'API présentée ci-dessus n'intègre que des services de communication et de synchronisation et n'est donc pas suffisante. Dans notre cas, L'API doit intégrer des services permettant de configurer et de contrôler les composants du système. La Figure 38 présente l'extension de cette bibliothèque par l'API de vérification. Pour éviter de surcharger cette figure, les APIs de communication et de synchronisation ont été supprimées.

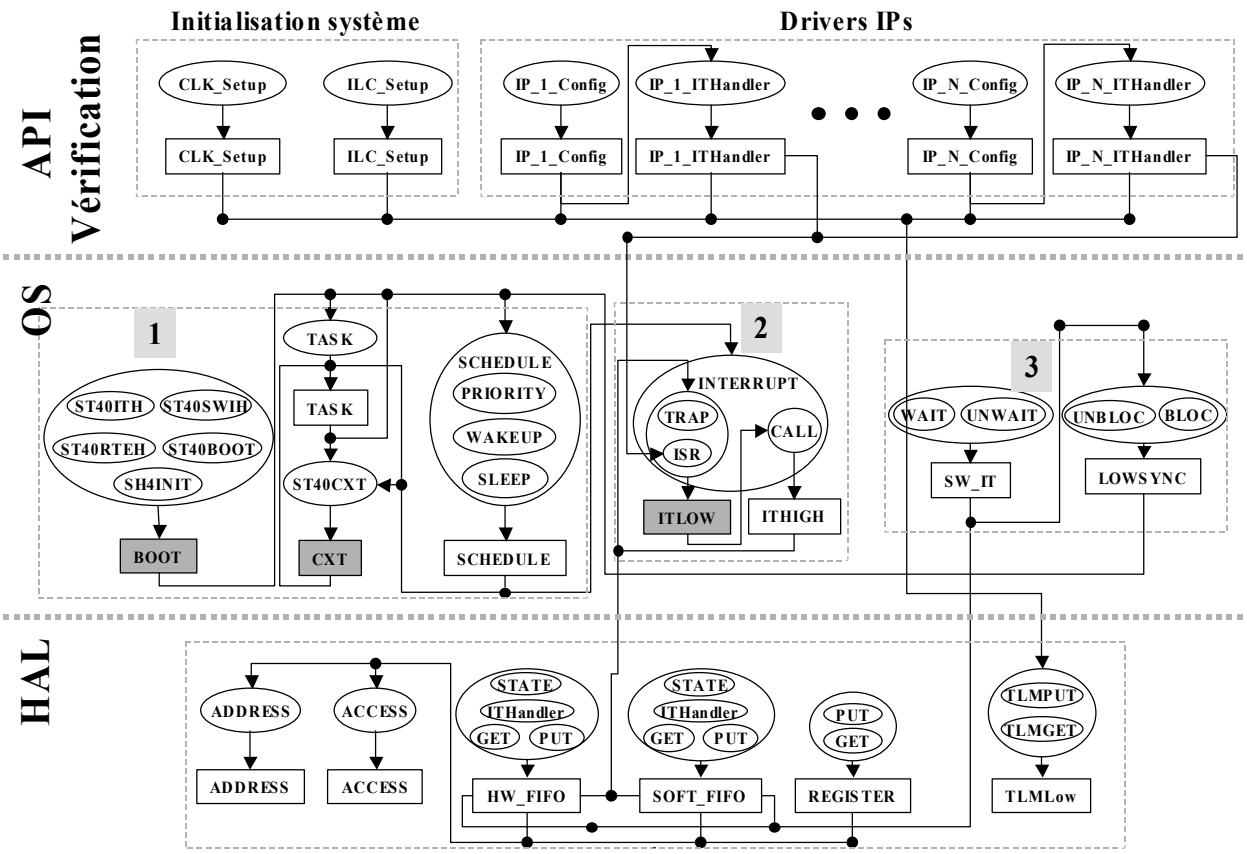


Figure 38 : La bibliothèque des éléments du système d'exploitation et l'ajout de l'API

L'approche utilisée pour étendre la bibliothèque du système d'exploitation est de construire une nouvelle couche que l'on appelle API de vérification et qui repose sur les couches déjà existantes. Un exemple de dépendance entre l'élément « IP_1 » et les services « Read/Write » de l'élément « TLMLow » permet d'illustrer la construction de l'API. Elle est constituée par des drivers de base de l'ensemble des composants présents dans le système. Ceci est illustré sur la figure par les services de l'API. Ainsi, si on considère par exemple le service « CLKGEN_Config », celui représente une fonction ou driver qui permet de configurer le composant contrôlant la génération des horloges d'un système donné. On procède ainsi pour chacun des composants du système (IP_1,…,IP_N). De plus, suivant les composants, la bibliothèque peut intégrer un service correspondant au traitement des interruptions par exemple « IP_1_Handler ». Ce service repose sur la couche OS et les éléments de gestion des interruptions.

IV.C. La génération automatique du système d'exploitation et les programmes de test de bas niveau

La génération du système d'exploitation s'effectue à l'aide d'ASOG. Ce générateur prend en entrée l'ensemble des éléments exposés dans les sections précédentes. On rappelle ces entrées :

- La modélisation de la partie logicielle correspondant à la définition du profil des programmes de test et de l'ensemble des paramètres qui le compose ;
- Le code des différentes tâches définissant le comportement du programme de test de haut niveau ;
- La bibliothèque des éléments constituant la description du système d'exploitation.

A partir de ces éléments le générateur va sélectionner et spécialiser les éléments de la bibliothèque pour produire un système d'exploitation spécifique. La génération de ce système permet de réaliser le raffinement des programmes de test de haut niveau illustré sur la Figure 39. Cette figure présente l'exemple d'une tâche d'un programme de test et la composition des couches générées par ASOG.

Cette tâche repose sur trois fonctions de l'API et de l'OS, les flèches illustrent les services correspondant dans la bibliothèque. ASOG construit ensuite à partir de ces services, le système d'exploitation. La figure illustre ainsi le raffinement du service « IP1_Config » qui repose sur les services « Read/Write » de l'élément « TLMLow ».

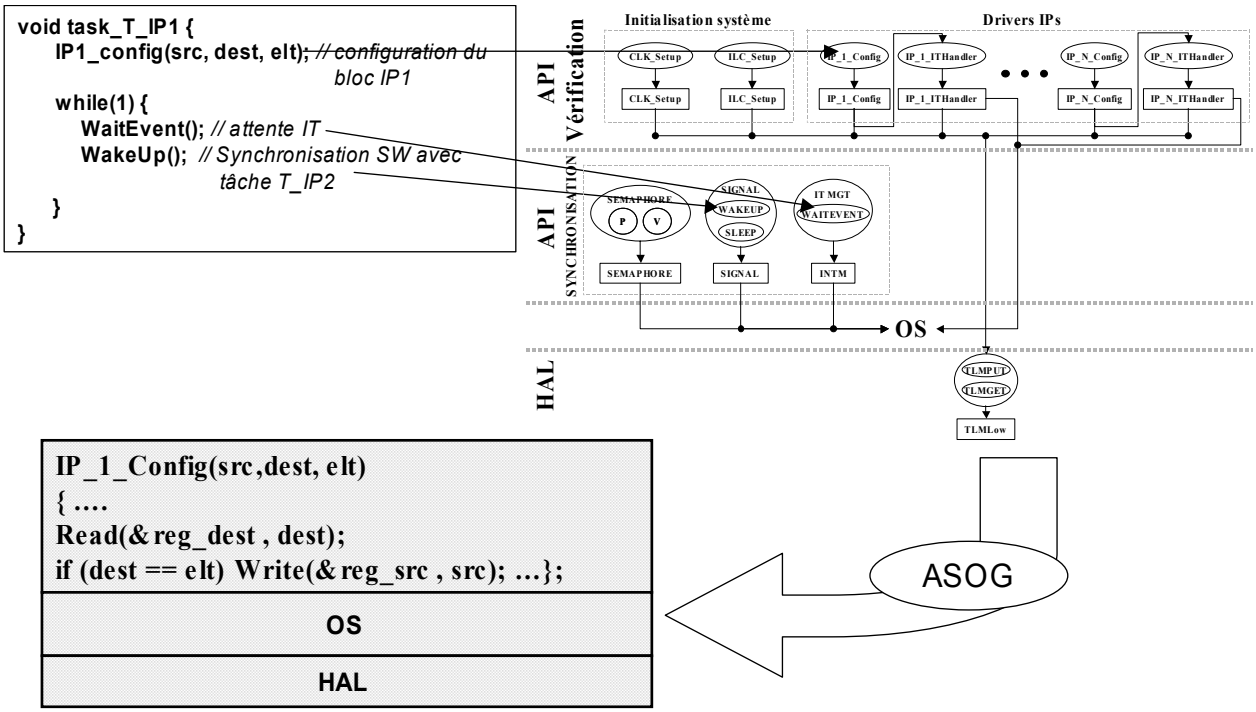


Figure 39 : Présentation du raffinement des programmes de test de haut niveau par la génération d'un système d'exploitation spécifique

V. Conclusion

Ce chapitre a présenté l'ensemble de la méthode qui est définie pour la vérification de l'intégration des systèmes monopuces. Elle s'effectue en deux étapes :

- Tout d'abord la génération systématique de programmes de test logiciel de haut niveau ;
- Puis le raffinement de ces programmes par la génération d'un système d'exploitation spécifique.

Cette méthode est rendue possible par la modélisation du système sous la forme de services offerts et de services requis. Cette modélisation intègre aussi bien la partie logicielle que la partie matérielle. Combinée à la définition de règles de test simple, elle permet de réaliser la capture du plan de test dans la méthode. De ce fait l'approche de génération des programmes de test devient systématique.

De plus, nous avons vu que les programmes de test générés sont constitués de tâches et sont construits sur une API. Cette API permet de réaliser l'abstraction des composants matériels du système et correspond ainsi à leurs drivers. En plus d'offrir l'abstraction des composants, l'API permet de définir des mécanismes de synchronisation sophistiqués entre, d'une part les tâches constituant le programme de test et d'autre part, entre les tâches et le matériel. Cette structure permet d'envisager la réutilisation de l'API.

Enfin, le raffinement des programmes de haut niveau s'effectue par la génération d'un système d'exploitation. Cette étape permet l'utilisation des programmes de test de haut niveau pour réaliser la vérification au niveau RTL d'un système qui peut ainsi être appliquée tout au long de la conception. De plus, le système d'exploitation permet la mise en œuvre des mécanismes de synchronisation lorsque l'on passe au niveau RTL. L'utilisation d'un tel système d'exploitation permet d'envisager une réduction du volume de code nécessaire pour réaliser la description des programmes de test.

Chapitre 5 : ETUDE DE CAS : APPLICATION DE LA METHODE SUR UN SYSTEME INDUSTRIEL

I. INTRODUCTION 120

II. LE SYSTEME PIAGET 120

II.A. GENERALITES 120
II.B. LE SOUS SYSTEME SELECTIONNE 122

III. L'APPLICATION DE LA METHODE 123

III.A. LA MODELISATION DU SOUS-SYSTEME 124
III.B. LES PROGRAMMES DE TEST DE HAUT NIVEAU 128
III.C. LA BIBLIOTHEQUE DU SYSTEME D'EXPLOITATION ET L'API DE VERIFICATION 131
III.D. LE RAFFINEMENT DES PROGRAMMES DE TEST LOGICIEL DE HAUT NIVEAU 133

IV. RESULTATS ET ANALYSES 135

IV.A. RESULTATS 136
IV.B. ANALYSES 137

V. CONCLUSION 138

I. Introduction

Ce chapitre présente l'étude de cas réalisée pour illustrer la méthode de vérification de l'intégration décrite au chapitre 4. Cette étude est effectuée sur un système industriel en cours de conception au sein de l'équipe TV de STMicroelectronics. Le système s'appelle PIAGET, il est destiné aux applications pour la télévision numérique haute définition.

La structure de ce chapitre se décompose en trois sections. La première section décrit le système sur lequel a été appliquée la méthode. La deuxième section présente les développements effectués pour son application. Etant donné que la génération des programmes n'est pas automatique, sont présentés les programmes de test de haut niveau réaliser pour cette étude de cas. Afin de permettre le raffinement automatique des programmes de test de haut niveau, cette section décrit également la bibliothèque du système d'exploitation. Elle s'achève par un exemple de programme de test raffiné. La dernière section présente les résultats obtenus et donne une analyse de l'application de cette méthode.

II. Le système Piaget

Cette section présente d'abord le système PIAGET dans son intégralité puis plus particulièrement le sous-ensemble sélectionné pour l'étude de cas.

II.A. Généralités

Le système monopuce PIAGET est destiné aux applications de télévision numérique haute définition. Il est constitué d'environ 10 millions de transistors. La Figure 40 donne un synoptique de la structure du PIAGET. On distingue sur celle-ci une frontière représentée par les pointillés, différenciant la partie purement numérique de la partie mixte analogique et numérique. Pour la suite nous ne considérons plus que la partie numérique du système.

Comme nous l'avons déjà précisé dans le chapitre 3, ce système intègre un réseau de communication complexe basé sur le STBus, composé de 47 initiateurs et 48 cibles. Ce réseau

de communication permet d'interconnecter de nombreux composants que l'on peut regrouper en plusieurs sous-systèmes :

- Un sous-système de contrôle et de configuration (CPU Sub-System)
- Un sous-système de traitement des flux vidéo des sorties (Video Pipe Sub-System),
- Un sous système de traitement des flux vidéo des sorties auxiliaires (Auxiliary Pipe Sub-System),
- Un sous-système de traitement des flux des entrées non compressées vidéo (Uncompressed Data Sub-System),
- Un sous-système de traitement des flux compressés (Compressed Data Sub-System),
- Un sous système de synchronisation (Synchronization Sub-System),
- Un sous système de communications externes du système (Comms Sub-System)
- Un sous système Blitter (Blitter Sub-System) : Accélérateur graphique.

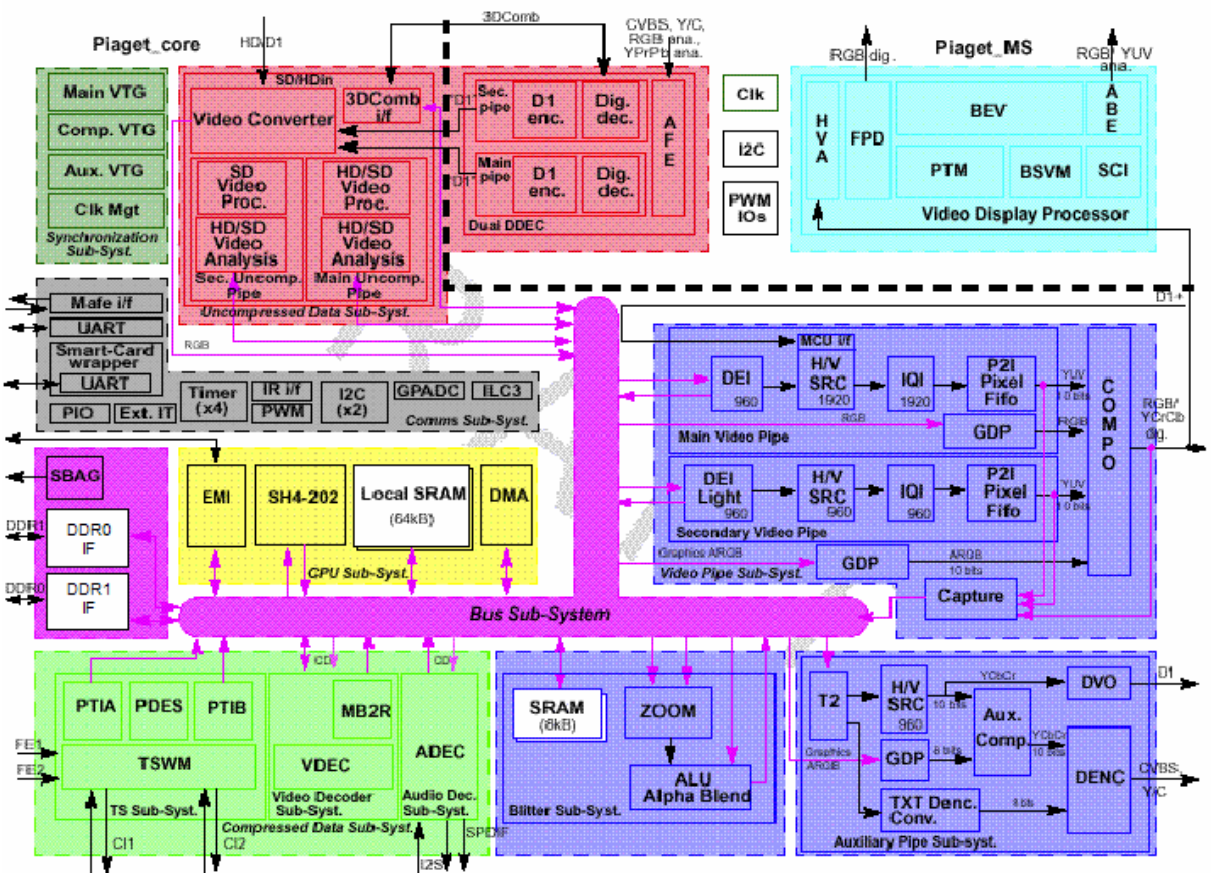


Figure 40 : Synoptique de la structure du système PIAGET

A travers l'ensemble de ces sous-systèmes, le PIAGET offre de nombreuses fonctionnalités parmi lesquelles on peut citer :

- L'acquisition et l'affichage en parallèle de deux flux vidéo ;
- Le traitement de flux vidéo MPEG 2 et de flux non compressés de type NTSC, HD 1080i…
- L'amélioration de la qualité de l'image par des traitements numériques complexes.

II.B. Le sous système sélectionné

Après avoir présenté le système PIAGET dans son ensemble, voici la partie du système qui a fait l'objet de l'étude de cas (Figure 41). Cette étude de cas a consisté à réaliser la vérification de l'intégration du sous-système « UNCOMPRESSED ». La validation de ce sous-système au niveau RTL est réalisée par l'utilisation des programmes de test de haut niveau écrits manuellement et ensuite raffinés.

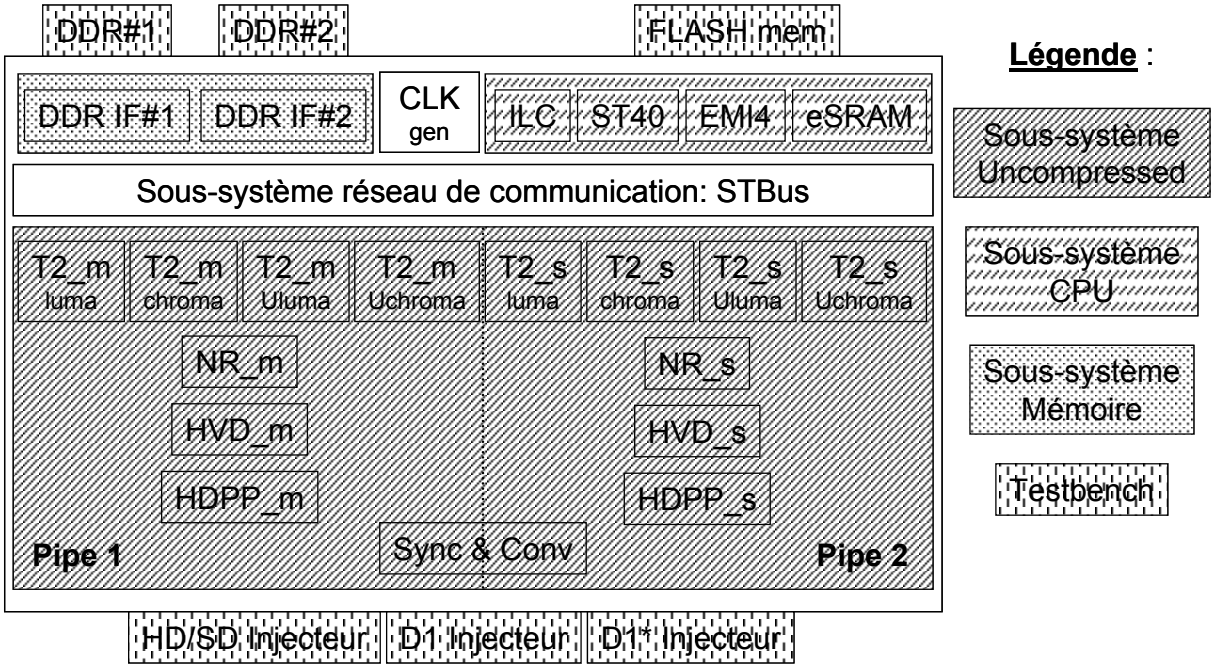


Figure 41 : Schéma du sous-ensemble sélectionné pour l'étude

La Figure 41 présente le schéma du sous-ensemble du système Piaget utilisé pour l'étude. Il se compose de six parties :

- Le sous-système UNCOMPRESSED qui réalise l'acquisition et le traitement d'un flux vidéo numérique non compressé. Il peut réaliser cette opération sur deux flux vidéo simultanément, d'où la présence de deux chemins de traitement identiques (Pipe 1 et Pipe 2). En amont, se trouve un composant permettant de réaliser la conversion et la re-synchronisation du flux vidéo numérique en entrée. Un chemin de traitement du flux vidéo se compose :
  - D'un élément réalisant l'analyse et le découpage du flux vidéo (HDPP) ;
  - D'un élément réalisant une décimation du flux (HVD) ;
  - D'un élément réalisant une réduction du bruit (NR) ;
  - Enfin de plusieurs interfaces STBus type T2 (T2), effectuant le transfert des données du flux vidéo en mémoire.
- Le sous-système CPU comprenant un processeur (ST40) avec sa mémoire dédiée (eSRAM), un contrôleur d'interruption (ILC) et une interface pour une mémoire Flash externe (EMI4) ;
- Le sous-système Mémoire constitué de deux interfaces pour les mémoires partagées du système dans lesquelles sont stockées les données vidéo ;
- Le sous-système Réseau de communication basé sur une interconnexion de plusieurs STBus et permettant la communication entre tous les sous-systèmes ;
- L'environnement de test qui simule l'environnement du système ;
- Un composant (CLK gen) configurant les horloges du système.

III. L'application de la méthode

Cette section décrit en quatre parties l'application de la méthode de vérification présentée au chapitre 4 :

- La modélisation du sous-système ;
- La description des programmes de test logiciel de haut niveau ;

- La bibliothèque du système d'exploitation ;
- La présentation des programmes de test raffinés.

Bien que la génération des programmes ne soit pas réalisée, la présentation de la modélisation du système permet de préciser les hypothèses utilisées pour appliquer la méthode. La modélisation définit ainsi les éléments du sous système qui sont adressés. Elle permet également de définir l'API employée pour réaliser la description des programmes de test de haut niveau.

III.A. La modélisation du sous-système

La Figure 42 présente la modélisation de la structure du système. A travers cette modélisation sont également représentés les services de base pour les composants qui ne sont pas considérés d'un point de vue de la vérification. Ces services sont cependant indispensables au fonctionnement du système et de l'Uncompressed. Ce modèle présente les hypothèses simplificatrices permettant d'adresser uniquement la vérification de l'intégration du sous-système Uncompressed. On notera également que la modélisation de ce sous-système a été simplifiée pour l'expérimentation et que le Pipe2, identique au Pipe1, n'est pas représenté sur la figure. De même, les services offerts par le composant logiciel (ST40) ne sont pas représentés sur la figure. Ces éléments sont cependant définis par les services requis par les composants matériels du système et sont grisés sur la figure.

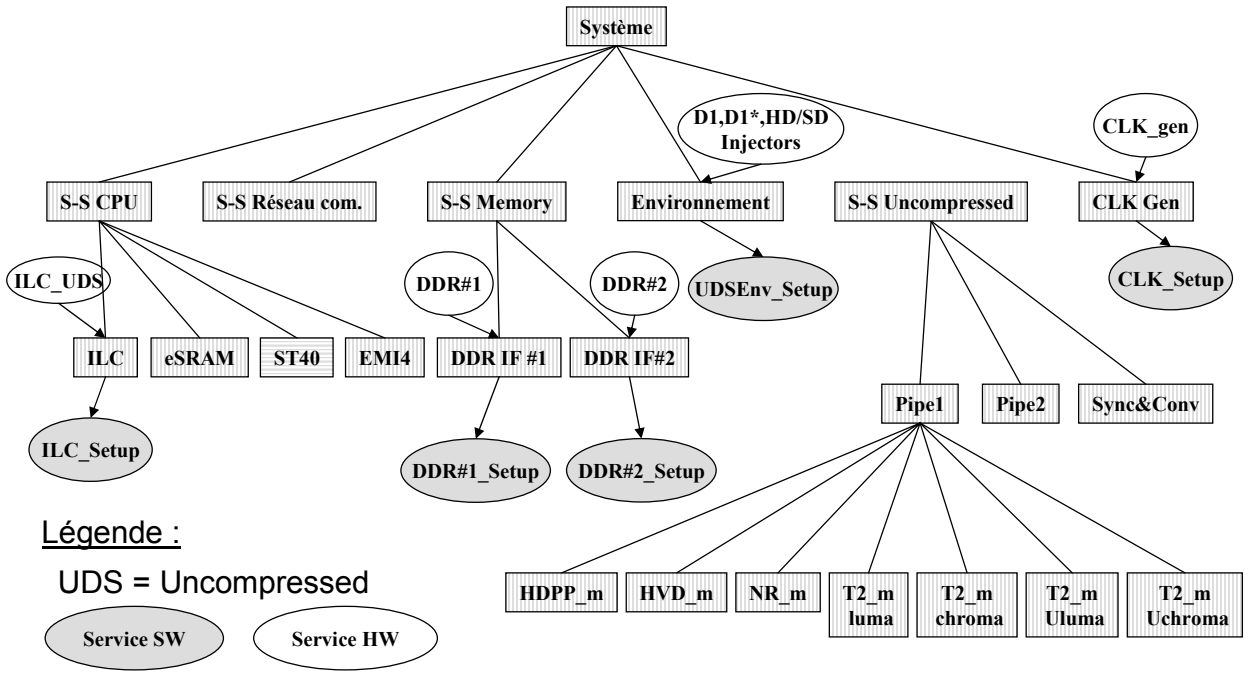


Figure 42 : Modélisation de la structure du sous-ensemble du PIAGET et des services pour les composants non adressés par la vérification

La Figure 42 présente les différents composants et leurs services nécessaires pour la vérification du système Uncompressed. On considère comme hypothèse que :

- La configuration des interfaces pour les DDRs est fixée ;
- La configuration du contrôleur d'interruption n'adresse que les interruptions issues du sous-système Uncompressed ;
- La configuration des horloges est réalisée pour le fonctionnement du sous-ensemble PIAGET présenté ici.

Les modèles correspondants aux composants du sous-système Uncompressed sont présentés sur les figures ci-dessous. La Figure 43 présente les modèles pour le composant HDPP (a) et HVD (b). La description des services du composant HDPP a déjà été réalisée dans le chapitre 4. En ce qui concerne le composant HVD, les services offerts représentent les différentes combinaisons des facteurs de décimation horizontale (HD) et verticale (VD).

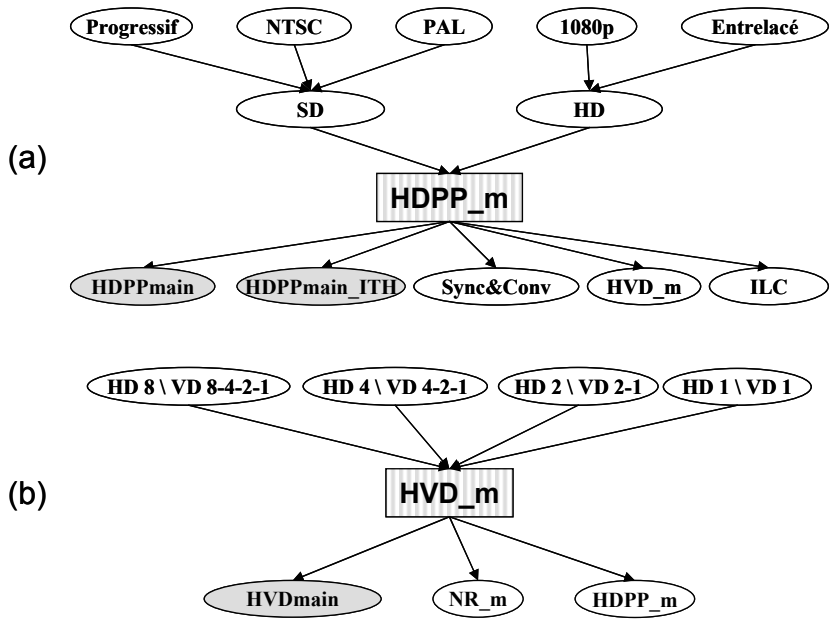


Figure 43 : Modélisation des services pour les composants : HDPP (a) et HVD (b)

La Figure 44 donne la représentation de la modélisation des composants NR (a) et T2 (b). On notera que pour les T2, seule la modélisation du composant T2LUMA est donnée, les autres modèles sont identiques au nom près. Pour le composant NR, les services offerts définissent les différents modes de traitement des données pouvant être réalisés pour réduire le bruit de l'image. Dans notre cas, aucun traitement ne sera effectué (service BYPASS), ce qui simplifie également notre étude. Les composants T2 permettent de réaliser le transfert des données en mémoires, ils ne disposent que d'un seul service offert comme le montre le modèle.

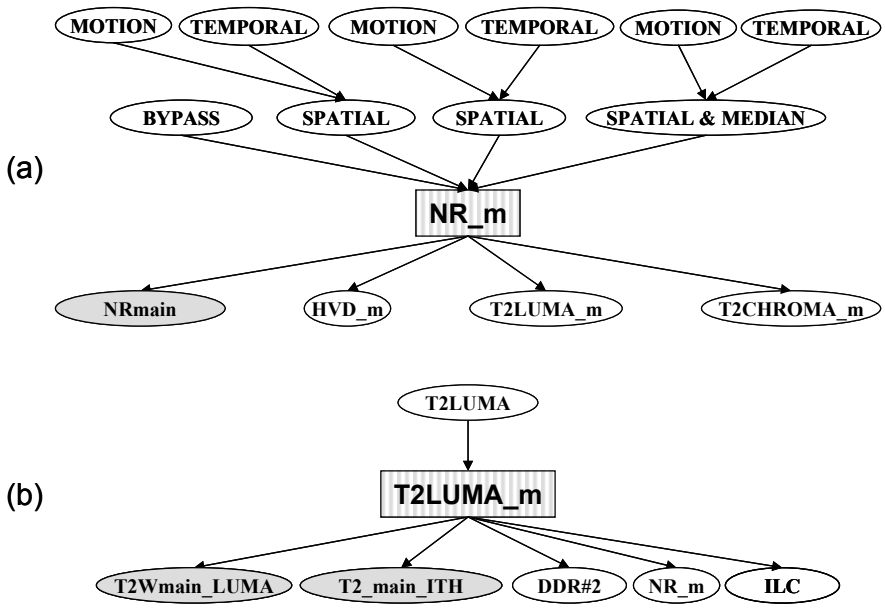


Figure 44 : Modélisation des services pour les composants : NR (a) et T2LUMA (b)

Enfin, pour finir la description des différents services associés aux composants du sous-système Uncompressed, la Figure 45 donne la modélisation du composant Sync&Conv. Ce composant réalise l'interface et le multiplexage des entrées vidéo (D1, D1*, HD/SD) avec les deux pipes (UDS_m, UDS_s) du sous-système Uncompressed. Les services offerts représentés sur cette figure montrent qu'il est possible de diriger chacun de ces flux vidéo vers l'un des pipes. Afin de simplifier la figure, ces combinaisons ne sont pas toutes données.

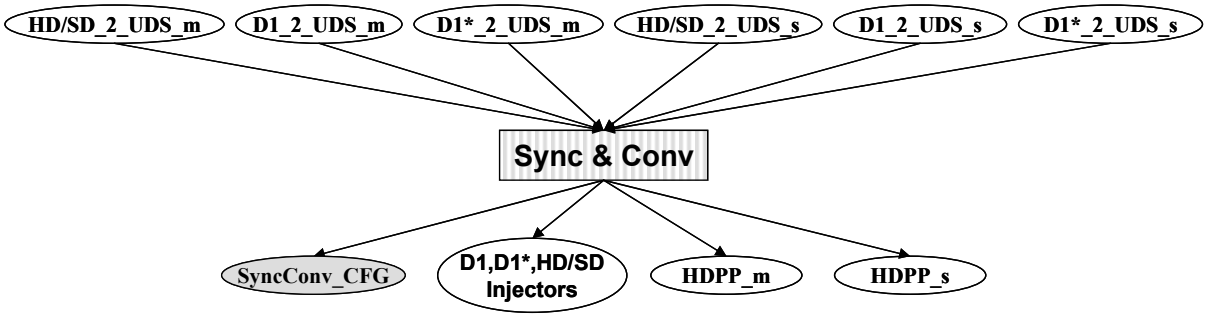


Figure 45 : La modélisation des services du composant Sync & Conv

Ces modèles permettent de présenter l'ensemble des services impliqués dans les scénarii de test. En d'autre terme ils définissent les fonctionnalités des composants mises en œuvre

ainsi que l'ensemble des fonctions de l'API de vérification utilisées. Ils mettent également en évidence les interactions entre les différents composants du système.

III.B. Les programmes de test de haut niveau

Lors de la réalisation de la vérification, six programmes de test de haut niveau ont été développés. Ces éléments n'ont pas pour vocation de couvrir l'ensemble de la vérification de l'intégration des composants du sous-système Uncompressed mais d'illustrer l'application de la méthode proposée. On distinguera deux ensembles ciblants :

- L'intégration du composant HDPP ;
- L'intégration du sous-système Uncompressed.

Le premier ensemble de tests adresse l'intégration du composant HDPP. Comme le défini le modèle, ce composant offre cinq types de services. Cinq programmes de test ont ainsi été réalisés. Ils permettent de configurer l'ensemble des composants du système de manière à réaliser la vérification du traitement des différents formats supportés : flux vidéo NTSC, PAL, Progressif, HD 1920x1080 entrelacé, HD 1080 progressif. La structure de ces programmes de test logiciel de haut niveau est donnée sur la Figure 46. Leur profil se compose de quatre tâches logicielles :

- Task_standby : tâche inactive ;
- Task_aInit : cette tâche réalise la configuration des horloges, des interfaces mémoire, du contrôleur d'interruption et de l'environnement de test. Elle permet de réaliser l'initialisation du système ;
- Task_b_uds_T2main : cette tâche réalise la configuration des interfaces T2. Elle est synchronisée avec le matériel par une interruption signifiant la fin du transfert de données en mémoire. Elle est également synchronisée avec la tâche Task_b_uds_main pour permettre la re-programmation des T2 pendant le déroulement du test ;
- Task_b_uds_main : cette tâche réalise la configuration des composants du pipe 1 (HDPP, HVD,…). Elle est synchronisée avec le matériel par une interruption

provenant du HDPP indiquant avec un délai programmable la fin du traitement d'une image, permettant ainsi la synchronisation pour la programmation des T2.

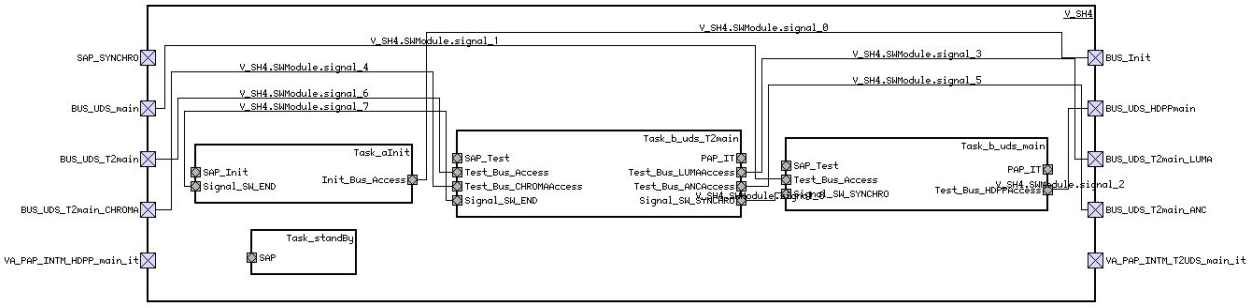


Figure 46 : Représentation Colif de la structure logicielle pour les tests adressant l'intégration du composant HDPP

Le second ensemble est constitué d'un test pour l'intégration du sous-système et met en œuvre les deux « pipes » : Flux vidéo HD 1920x1080 entrelacé sur le pipe 1 et un flux vidéo NTSC sur le pipe 2. La structure de ce programme de test composé de six tâches logicielles est donnée sur la Figure 47. Le test possède une structure similaire aux tests précédents. Deux tâches additionnelles pilotent ainsi le Pipe 2 de la même manière que celle présentée pour l'intégration du HDPP.

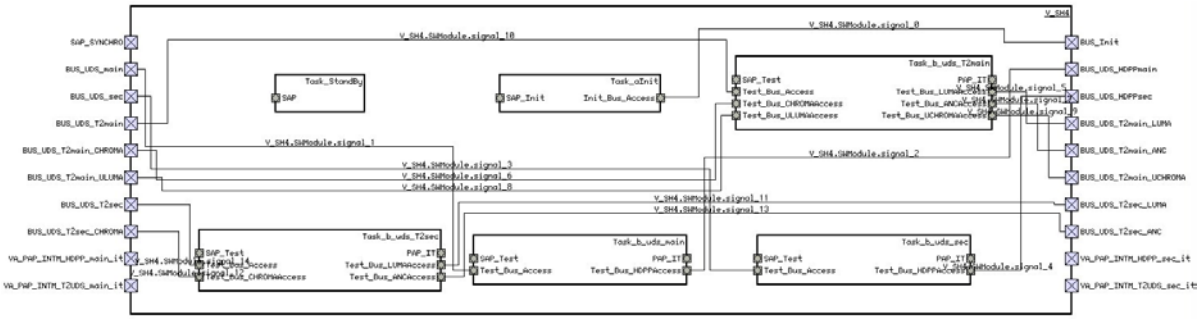


Figure 47 : Représentation Colif de la structure logicielle du test d'intégration pour le sous-système Uncompressed

Pour compléter la description des programmes de test de haut niveau, voici une illustration du code des tâches issu du programme ciblant l'intégration du HDPP pour un flux de type PAL. Cet exemple correspond à la structure logicielle constituée de quatre tâches

exposée Figure 46. Les figures suivantes illustrent le contenu des tâches : Task_aInit (Figure 48), Task_b_uds_T2main (Figure 49) et Task_b_uds_main (Figure 50).

```
void task_ainit::task_beh()
{
  Init_Bus_Access.CLKGENSetup();
  Init_Bus_Access.DDR2Setup();
  Init_Bus_Access.UDS_ENVSetup();
  Init_Bus_Access.ILCSetup_UDS();
  // Waiting the end of the test
  Signal_SW_END.Sleep();
  * END_TEST = 0x5555AAAA; // end the simulation
};
```

Figure 48 : Illustration du code de la tâche : task_aInit

```
void task_b_uds_t2main::task_beh()
{
   int cur_field = 0;
   while(cur_field < NB_FIELD )
     {
       Signal_SW_SYNCHRO.Sleep(); // Wait SW synchro to configure the T2
       cur_field++;
       if (cur_field != NB_FIELD)  // Avoid reprogramming the T2 while finishing the test
       {
          Test_Bus_LUMAAccess.T2Wmain_LUMA(LUMA_T2START[cur_field],…);
          Test_Bus_CHROMAAccess.T2Wmain_CHROMA(CHROMA_T2START[cur_field],…]);
          Test_Bus_ULUMAAccess.T2Wmain_ULUMA(ULUMA_T2START[cur_field],…);
          Test_Bus_UCHROMAAccess.T2Wmain_UCHROMA(UCHROMA_T2START[cur_field],…);
       }
       PAP_IT.WaitEvent(); // waiting end of transfert interrupt
     }
   // Test Finished ending the simulation -> back to Init Task
   Signal_SW_END.WakeUp();
   Signal_SW_END.Sleep();
};
```

Figure 49 : Illustration du code de la tâche : task_b_uds_T2main

```
void task_b_uds_main::task_beh()
{
   // Sync block config
   Test_Bus_Access.Sync_Config(1,0,0); // output1 -> hdsd to main in SD
   // HDPP block config
   Test_Bus_HDPPAccess.HDPPmain(720,288,50,51,23,22,0);
   // HVD block config
   Test_Bus_Access.HVDmain(0,0,0,0); // do nothing bypass
   // NR block config
   Test_Bus_Access.NRmain(75,720,1,0,0); // bypass
   // Start UDS Injector : HD/SD
   Test_Bus_Access.Write((int*)&ENV_UDS_HDSD->START,0x0001);
   while(1)
     {
        PAP_IT.WaitEvent();  // HDPP IT waiting
        Signal_SW_SYNCHRO.WakeUp(); // SW synchro to configure the T2
     }
};
```

Figure 50 : Illustration du code de la tâche : task_b_uds_main

Ces figures présentent le contenu des tâches logicielles. On constate qu'elles sont pour l'essentiel constituées d'appels système des fonctions de l'API de vérification et de l'OS pour la synchronisation. Ces fonctions sont représentées en gras sur les figures. On remarquera également que les différences entre les tests permettant la vérification de l'intégration du composant HDPP sont les valeurs de paramètres de ces mêmes fonctions. Une description des fonctions de l'API de vérification est donnée dans l'annexe 1.

III.C. La bibliothèque du système d'exploitation et l'API de vérification

Afin de permettre la mise en œuvre du raffinement des programmes de test, il a été nécessaire de réaliser une extension de la librairie du système d'exploitation, système qu'il a tout d'abord fallu porter sur le processeur ST40 (la bibliothèque avait en effet été initialement développée pour le processeur ARM7). La deuxième partie de l'extension consiste à intégrer à cette même bibliothèque les fonctions de l'API de vérification. En d'autres termes, il s'agit d'ajouter les différents drivers des composants du système sous test.

La Figure 51 donne une représentation de l'extension de la bibliothèque du système d'exploitation. On retrouve sur cette figure les deux extensions:

- Le portage : les éléments de la bibliothèque du système d'exploitation modifiés pour réaliser le portage sont : le boot, le changement de contexte et la gestion des ITs (parties grisées de la figure) ;
- L'API de vérification : tous les services et les éléments ne sont pas représentés. Les services associés aux composants NR, HVD n'y figure pas, de même, les services des T2 du Pipe 1 sont regroupés sous le service « T2MAIN ». D'autre part, les services et les éléments de l'API de vérification sont regroupés suivant leur sous-système respectif. On remarquera également que le découpage de la couche API permet une plus grande réutilisation du code. La configuration des HDPP (pipe 1 & 2) est identique, la seule différence réside dans l'adresse de registres de configuration. Dans ce cas, nous introduisons un service générique « 'HDDPconfig » qui s'applique aux deux composants et qui est requis par les éléments « HDPPmain & HDPPsec ». On notera que cette API de vérification repose principalement sur la couche HAL et l'élément « TLMLow », à l'exception des routines de traitement des interruptions par exemple « HDDPmainITH ». Celles-ci reposent sur la partie « INTERRUPT » de l'OS.

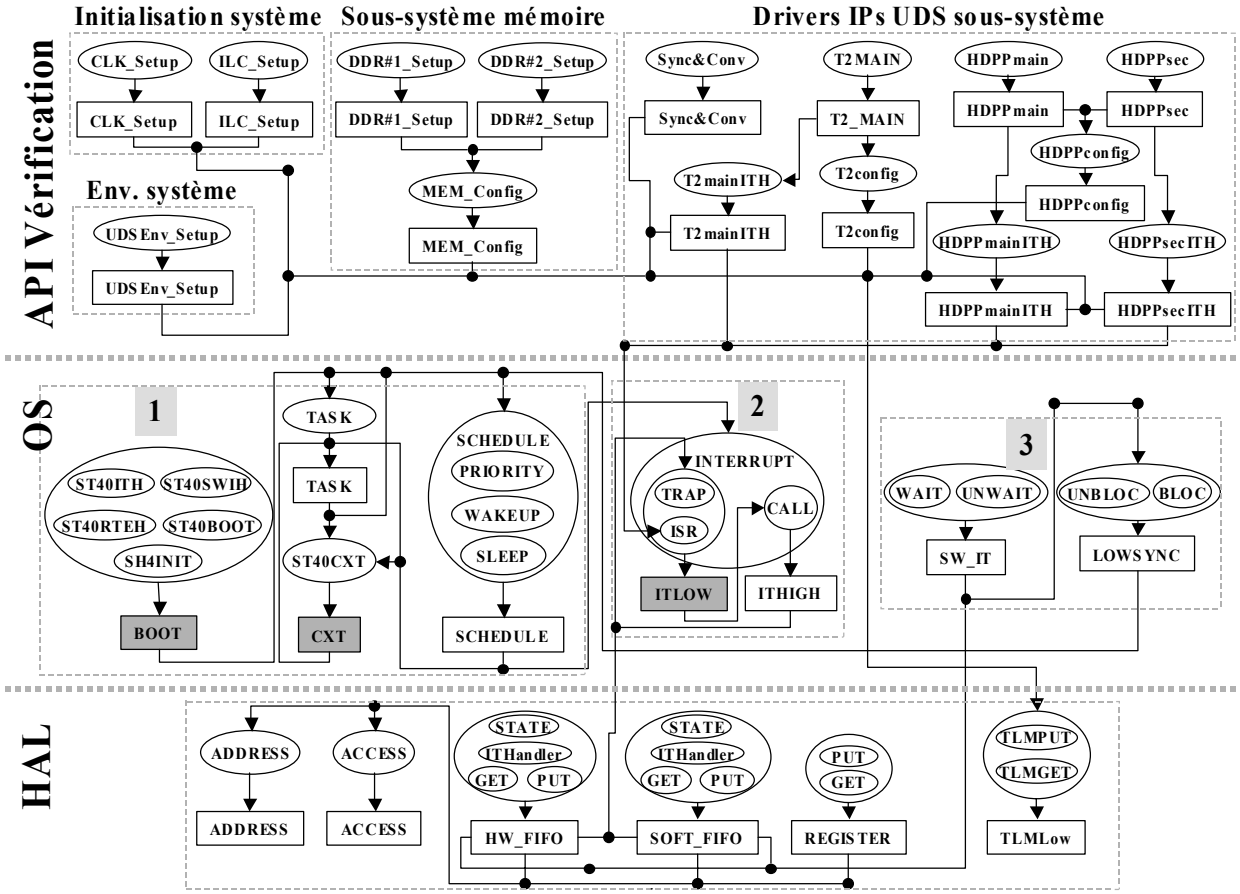


Figure 51 : Représentation de la bibliothèque du système d'exploitation étendue

Le système d'exploitation permet ainsi d'abstraire l'API de vérification du matériel. De ce fait, il accroît ses possibilités de réutilisation. Une description des fonctions de l'API de vérification est donnée dans l'annexe 1.

III.D. Le raffinement des programmes de test logiciel de haut niveau

La réalisation du raffinement des programmes de test de haut niveau par la génération d'un système d'exploitation spécifique s'effectue par l'intermédiaire de l'outil ASOG. Ce générateur nécessite la spécification de paramètres permettant la spécialisation des différents éléments de la bibliothèque. Ces paramètres sont définis sur les ports des tâches et du module logiciel. Ils spécifient les priorités d'interruption et les services sur lesquels repose le programme. Aussi, pour illustrer ces paramètres, considérons l'exemple du programme de test destiné à la vérification de l'intégration du composant HDPP. La Figure 52 donne une illustration des paramètres Colif de la tâche task_b_uds_T2main et un exemple des paramètres de spécification des services.

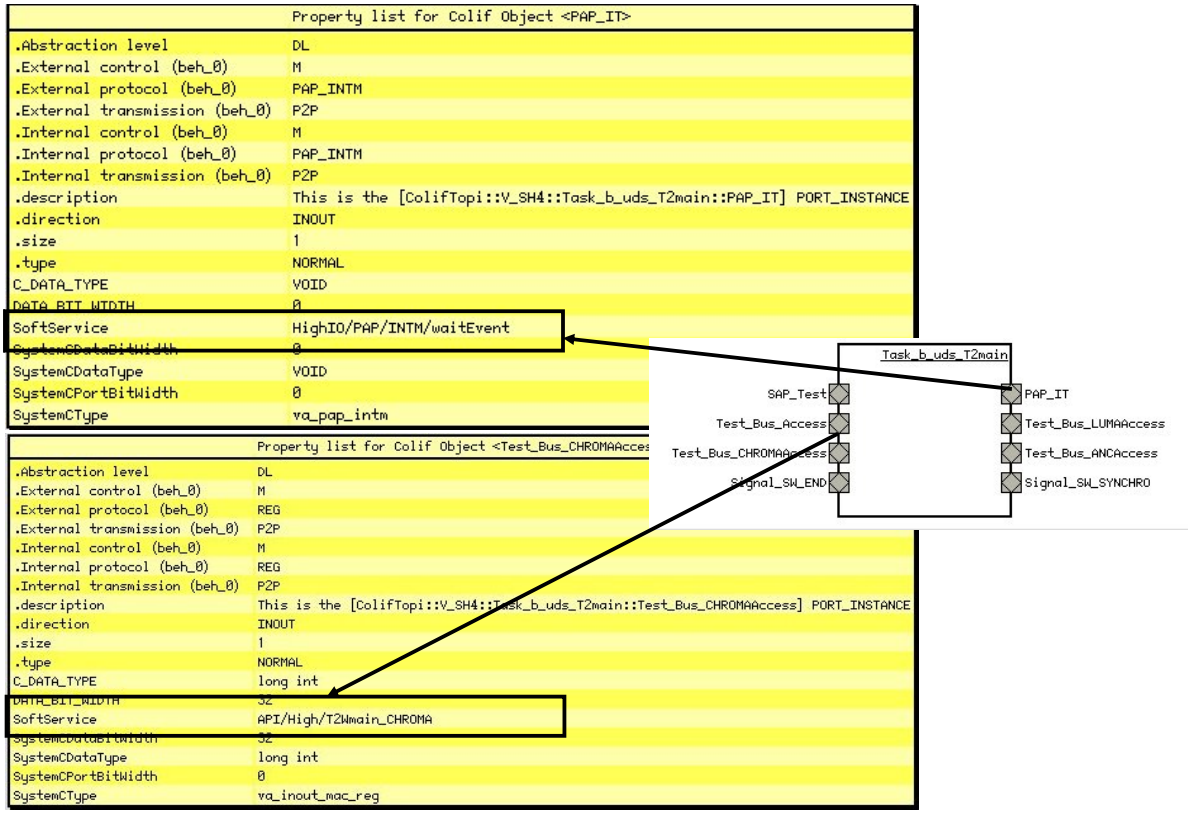
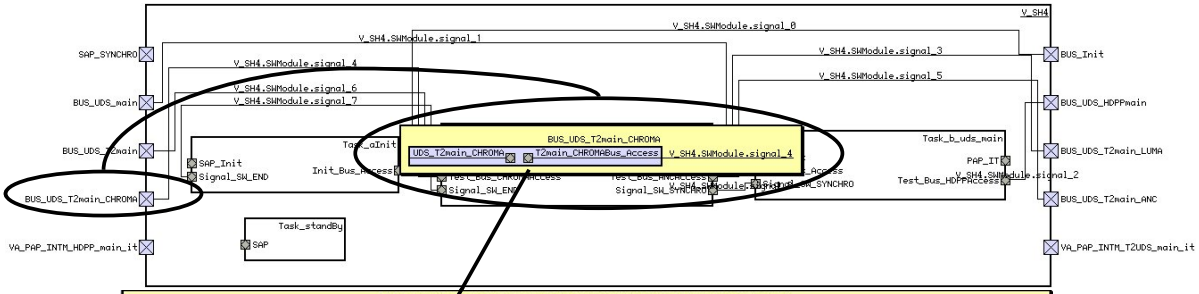


Figure 52 : Illustration des paramètres spécifiant les services pour la tâche task_b_uds_T2main

Les paramètres définissant les interruptions sont eux attachés aux ports du module logiciel. La Figure 53 présente le modèle Colif du port du module logiciel auquel est attaché le service permettant la configuration de la T2 Chroma. Ce port permet ainsi de définir les paramètres des interruptions attachées à ce composant et nécessaires au mécanisme de gestion des interruptions.



Property list for Colif Object <T2main_CHROMABus_Access>

.Abstraction level	DL
.External control (beh_0)	M
.External protocol (beh_0)	REG
.External transmission (beh_0)	P2P
.Internal control (beh_0)	M
.Internal protocol (beh_0)	REG
.Internal transmission (beh_0)	P2P
.description	this is the [ColifTopi::V_SH4::BUS_UDS_T2main_CHROMA::T2main_CHROMABus_Access] PORT_INSTANCE
.direction	INOUT
.side	INTERNAL
.size	1
.type	NORMAL
CHAN_PRIO	1
C_DATA_TYPE	long int
DATA_BIT_WIDTH	64
IT_LEVEL	10
IT_NUMBER	84
MASK_GET	2
MASK_PUT	1
SoftPortType	TLMLow
SystemCDataBitWidth	32
SystemCDataType	long int
SystemCPortBitWidth	0
SystemCType	va_inout_mac_reg

Figure 53 : Illustration des paramètres spécifiant les interruptions

A partir de la bibliothèque du système d'exploitation et des différents éléments de la modélisation des programmes de test de haut niveau, le générateur produit le code spécifique du système d'exploitation. La Figure 54 présente le code généré relatif à l'appel système réalisant la configuration du composant HDPP. Cet appel système est utilisé dans la tâche task_b_uds_main: « Test_Bus_HDPPAccess.**HDPPmain(720,288,50,51,23,22,0)** » (Figure 50). Il est réalisé par le système d'exploitation par la fonction « **HDPPmain_TLMLow_long_int** » qui appelle ensuite la fonction de configuration commune aux HDPP du pipe 1 et 2 : « **HDPP_Config_TLMLow_long_int** ».

```
void HDPPmain_TLMLow_long_int(int tlm_master_id, int NB_PIX, int NB_PIX_BLK, int NB_LINE_TOP, int
NB_LINE_BOT, int NB_LINE_BLK_TOP, int NB_LINE_BLK_BOT, int HD_NOT_SD){
    HDPP_Config_TLMLow_long_int(tlm_master_id,MAIN_HDPP,NB_PIX,NB_PIX_BLK,NB_LINE_TOP,N
B_LINE_BOT,NB_LINE_BLK_TOP,NB_LINE_BLK_BOT,HD_NOT_SD);
}
void HDPP_Config_TLMLow_long_int(int tlm_master_id, volatile struct s_uds_hdpp *p_HDPP_regs, int nb_pix,
int nb_pix_blk, int nb_line_top, int nb_line_bot, int nb_line_blk_top, int nb_line_blk_bot, int hd_not_sd){
    tlm_type tlm = tlm_long_int_TLMLow[tlm_master_id];
    int total_nb_lines_top = (nb_line_top + nb_line_blk_top)-1;
    int total_nb_lines_bot = (nb_line_bot + nb_line_blk_bot)-1;
    int total_nb_pix;
    total_nb_pix= (nb_pix + nb_pix_blk)-1;
    TLMLow_TLMPut_long_int(tlm.low,&p_HDPP_regs->VIDEO_VSTART_TOP,nb_line_blk_top);
    TLMLow_TLMPut_long_int(tlm.low,&p_HDPP_regs->VIDEO_VSTOP_TOP,total_nb_lines_top);
    TLMLow_TLMPut_long_int(tlm.low,&p_HDPP_regs->VIDEO_VSTART_BOT,nb_line_blk_bot);
    TLMLow_TLMPut_long_int(tlm.low,&p_HDPP_regs->VIDEO_VSTOP_BOT,total_nb_lines_bot);
    TLMLow_TLMPut_long_int(tlm.low,&p_HDPP_regs->VIDEO_HSTART,nb_pix_blk);
    TLMLow_TLMPut_long_int(tlm.low,&p_HDPP_regs->VIDEO_HSTOP,total_nb_pix);
    TLMLow_TLMPut_long_int(tlm.low,&p_HDPP_regs->IT_EN,0x8);
    TLMLow_TLMPut_long_int(tlm.low,&p_HDPP_regs->PRG_LINE_NR,(int) total_nb_lines_top/2 );
    if (!hd_not_sd) TLMLow_TLMPut_long_int(tlm.low,&p_HDPP_regs->CTRL,(0x1 <<
uds_hdppCTRLVIDEO_EAV_ENShift) | (hd_not_sd << uds_hdppCTRLHD_NOT_SDShift)); // this configures the
HDPP for SD interlaced (eg: NTSC, PAL) input flow
    else TLMLow_TLMPut_long_int(tlm.low,&p_HDPP_regs->CTRL,(0x1 << uds_hdppCTRLHEDShift)|(0x1 <<
uds_hdppCTRLVNFShift) | (0x1 << uds_hdppCTRLHD_NOT_SDShift)); // this configures the HDPP for HD 1080i
input flow...
    TLMLow_TLMPut_long_int(tlm.low,&p_HDPP_regs->PRG_VSYNC_OFF,(int) nb_line_blk_top/2);
}
```

Figure 54 : Illustration du code généré pour l'appel système permettant la configuration du composant HDPP

Sur cette figure, on note de nombreux appels à la fonction « TLMLow_TLMPut_long_int ». Cette fonction fait partie de l'API de communication offerte par la bibliothèque du système d'exploitation. Elle permet l'écriture d'une donnée à l'adresse spécifiée. Dans notre cas, les adresses correspondent aux adresses des registres du composant et permettent de réaliser sa configuration.

IV. Résultats et analyses

Cette section présente les résultats obtenus suite à la vérification de l'intégration du composant HDPP et du sous-système Uncompressed par la méthode proposée dans ce mémoire. Afin d'analyser ces résultats, ils sont comparés à la méthode de vérification employée par l'équipe TV. Elle consiste à écrire manuellement des programmes de test de bas niveau.

IV.A. Résultats

Dans cette expérimentation, plusieurs programmes de test logiciel de haut niveau ont ainsi été développés manuellement. Ils sont aux nombres de six et ont été décrits dans la section précédente. Ils ciblent l'intégration du composant HDPP et le sous système Uncompressed. En comparaison, la vérification complète de l'intégration des composants et du sous-système se compose de 113 tests.

Afin de réaliser cette vérification, il a donc été nécessaire de développer la bibliothèque du système d'exploitation. Ce développement correspond à l'API de vérification, constituée de 25 fonctions écrites dans le langage de macro. Ces fonctions correspondent en terme de développement à quelques 340 lignes de codes plus 100 lignes génériques que l'on retrouve pour chacune des fonctions. Il a ainsi été nécessaire de réaliser 440 lignes de code pour l'ensemble de la partie API de vérification de la bibliothèque du système d'exploitation.

Pour comparer le coût de développement des tests, une comparaison entre les programmes de haut niveau et les programmes de bas niveau (développés par l'équipe de vérification du Piaget) est donnée dans le Tableau 2. Ce tableau décrit le volume de code en nombre de lignes pour les programmes de test de haut et de bas niveau, pour l'intégration du composant HDPP et du sous système. Il contient également la taille de l'exécutable. On notera que le nombre de lignes de code est le nombre effectif, c'est dire sans commentaires. Il représente le nombre de lignes utiles du programme. Les chiffres donnés entre parenthèses pour les programmes de test de bas niveau correspondent à la partie du code qui est spécifique à la gestion des interruptions.

	Tests de l'intégration du composant HDPP		**Tests de l'intégration du sous-système Uncompressed**	
	Taille en lignes de code	Taille de l'exécutable (ko)	Taille en lignes de code	Taille de l'exécutable (ko)
Programmes de test de haut niveau	145	~28	180	~33
Programmes de test de bas niveau	200 à 290 (60 à 80)	9 à 10	340 à 450 (70 à 120)	12 à 13

Tableau 2 : Comparaison entre les programmes de test de haut niveau et de bas niveau

Au regard, de ce tableau on constate tout d'abord que le volume de code des programmes de test de bas niveau est donné sur un intervalle. Ceci est lié au nombre d'images que contient le flux vidéo. Il en est de même pour le code spécifique à la gestion des interruptions. En comparaison, les programmes de test de haut niveau ne sont pas impactés par le nombre d'images contenues dans le flux vidéo. De plus, il n'intègre pas de partie spécifique à la gestion des interruptions, celle-ci étant prise en charge par le système d'exploitation et définie dans la bibliothèque.

IV.B. Analyses

Au travers de cette étude de cas, on constate en comparant les programmes de test logiciel de haut niveau avec les programmes de test logiciel de bas niveau développés par l'équipe de vérification du Piaget que :

- Le volume de code développé pour les programmes de test peut être réduit de l'ordre de 30 à 50% ;

- La taille de l'exécutable augmente d'un facteur 3 pour les tests d'intégration de composants. Ce facteur diminue avec la complexité de tests, il est de 2,5 pour le test de l'intégration du sous-système. On remarquera qu'il est possible de diminuer ce facteur en optimisant l'API de vérification utilisée. Cependant ce point n'est pas critique étant donnée que l'application destinée au système est de taille plus importante. Le système d'exploitation à lui seul correspond à près de 30 ko;
- Enfin l'utilisation d'un système d'exploitation n'engendre qu'un surcoût inférieur à 5% en terme de temps de simulation. Ce surcoût est lié à l'initialisation statique des tâches en début de simulation.

Ainsi, l'expérimentation de la méthode montre tout d'abord sa faisabilité et l'impact négligeable lié à l'utilisation d'un système d'exploitation. Ensuite, elle permet de mettre en évidence la simplicité de développement des programmes de test de haut niveau. Le coût du développement de l'API de vérification dans la bibliothèque du système d'exploitation est quasiment équivalent à celui d'un seul test de bas niveau. La bibliothèque nécessite quelques 440 lignes de code pour l'API contre près de 300 pour un programme de test de bas niveau. De plus, sachant qu'il est nécessaire de développer de nombreux programmes, la maintenance devient vite conséquente, par exemple pour l'intégration du sous-système l'ensemble des tests représente près de 5000 lignes de codes. Aussi l'utilisation de cette API de vérification reposant sur un système d'exploitation permet de réduire le volume de code qu'il est nécessaire de maintenir. En outre, l'utilisation d'une API permet d'envisager sa réutilisation, lorsque les composants sont réutilisés dans d'autre système.

Enfin dans cette analyse, le taux de couverture du code HDL n'a pas été considéré. Les six programmes de test de haut niveau qui ont été développés dans cette expérimentation ne sont pas significatifs en comparaison de ceux développés par l'équipe de vérification du Piaget.

V. Conclusion

Ce chapitre a permis d'illustrer la mise en œuvre de la méthode de vérification de l'intégration qui a été proposée dans le chapitre 4. Il présente d'une part la modélisation du système et les programmes de test de haut niveau. D'autre part, il expose le développement de

la bibliothèque du système d'exploitation permettant de réaliser le raffinement automatique des programmes de test de haut niveau.

Cette expérimentation montre la faisabilité de la mise en œuvre de cette méthode de vérification de l'intégration. Il est également possible d'observer les bénéfices obtenus par l'utilisation d'un système d'exploitation :

- La modélisation du système et des composants permet la description du plan de test ;
- L'utilisation de programmes de test de haut niveau basés sur une API/OS permet de simplifier et de systématiser l'écriture des tests;
- La mise en œuvre de mécanismes sophistiqués de synchronisation logicielle/matérielle dans les tests est simplifiée et rendue possible par l'API et le système d'exploitation ;
- Le raffinement automatique des programmes de test de haut niveau par la génération automatique d'un système d'exploitation permet un gain de temps dans la maintenance des tests.

Cependant, l'application de cette méthode introduit deux limitations :

- L'accroissement de la complexité du déboguage des programmes de test qui est liée à l'utilisation d'un système d'exploitation. Pour en réduire l'importance, il est nécessaire d'utiliser des outils de déboguage logiciels efficaces. De plus, l'utilisation d'un model de haut niveau du système permet la validation plus rapide de l'OS ;
- L'augmentation du temps de simulation qui est négligeable et de l'ordre de 5%.

CONCLUSION ET PERSPECTIVES

Ce mémoire a permis dans un premier temps de présenter les problèmes de la vérification des systèmes monopuces. Les principaux problèmes rencontrés sont tout d'abord la complexité et la taille de ce type de système. Ils sont composés d'un assemblage complexe de composants logiciels et matériels (ces composants matériels étant supposés vérifiés de façon indépendante). La vérification de ces systèmes est ainsi confrontée au problème de la vérification de l'intégration : on doit valider l'interconnexion et les interactions entre les différents composants. Cependant, comme cela a été montré, ces systèmes se composent également de composants spécifiques qu'il est nécessaire de vérifier. Ce sont le réseau de communication et les interfaces matérielles réalisant l'adaptation du protocole de communication entre les composants et le réseau.

Dans un deuxième temps, ce mémoire dresse un inventaire non exhaustif des différents outils de vérification. Il rassemble cependant les principales méthodes de vérification classées suivant le type de circuit pouvant être adressé. A travers cette présentation, les avantages et les limitations de ces méthodes peuvent être définis. On remarque ainsi qu'il existe un grand nombre d'outils et de méthodes traitant des points spécifiques dans le domaine de la vérification des systèmes numériques. Cet inventaire permet également de mettre en évidence l'absence de solution unique. La problématique posée par les différents types de circuit (composant matériel, processeur, système monopuce) ne peut à l'heure actuelle être adressée dans son ensemble.

Après cet état de l'art, une méthodologie de validation globale a été proposée. Celle-ci consiste à découper la vérification des systèmes monopuces en trois parties :

- La vérification des interfaces matérielles ;
- La vérification du réseau de communication ;
- La vérification de l'intégration.

Elle exploite ainsi les avantages des méthodes existantes pour traiter deux points spécifiques de la vérification des systèmes sur puce. La vérification des interfaces matérielles s'effectue par une méthode formelle : la vérification de propriété. Et, la vérification du réseau de communication est réalisée par l'utilisation d'un environnement de test spécifique. Ce choix permet de par la standardisation de ces éléments d'automatiser leur vérification. Enfin, une nouvelle méthode de vérification est proposée pour compléter la méthodologie.

La méthode de vérification de l'intégration proposée dans ce mémoire est la principale contribution de cette thèse. Elle se décompose en deux étapes :

- **La génération automatique de programmes de test logiciel de haut niveau basés sur une API**. Cette partie n'est actuellement pas automatisée, mais fait l'objet d'une proposition de mise en œuvre. Elle permet de systématiser la génération des tests car elle capture le plan de test dans la modélisation du système et s'appuie sur des règles génériques pour réaliser la génération des scénarii de test. Cette approche générique et systématique permet d'appréhender de manière croissante la complexité des tests. Elle permet également de réaliser la vérification du modèle de haut niveau du système (par exemple TLM). Il est de ce fait possible de construire une suite de tests en avance de phase par rapport à la conception de l'architecture RTL du système.
- **Le raffinement des programmes de test logiciel de haut niveau par la génération automatique d'un système d'exploitation spécifique.** Cette partie du flot de la méthode de vérification permet de réaliser le raffinement des programmes de test de manière à effectuer la vérification de l'architecture RTL du système. Elle permet en outre par l'utilisation d'un système d'exploitation de proposer une solution au problème de la synchronisation entre le logiciel et le matériel. Mais, elle permet aussi d'intégrer dans la vérification, la partie logicielle du système. Enfin l'utilisation d'une API pour réaliser la description des programmes et d'un système d'exploitation généré automatiquement, permet de diminuer le temps pris pour effectuer la maintenance des tests.

Cette méthode de vérification est donc applicable tout au long de la conception d'un système, de la spécification à la réalisation RTL. Si l'on considère le modèle de pile informatique pour représenter un système monopuce, la Figure 55 permet d'illustrer les parties d'un système qui sont adressées par la méthodologie de vérification globale.

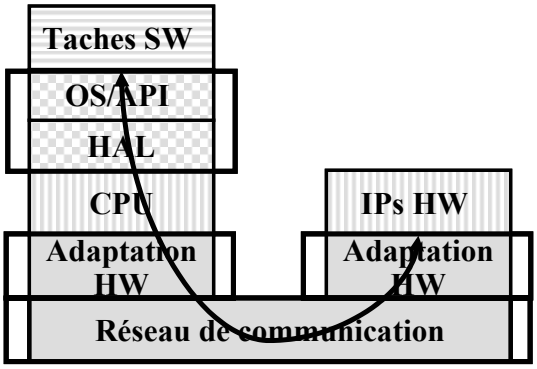


Figure 55 : Représentation des parties vérifiées dans un système monopuce par la méthodologie de validation globale proposée

Finalement, une expérimentation de la méthode de vérification de l'intégration sur un système industriel permet de mettre en évidence les bénéfices obtenus. Ils sont le fruit de la modélisation et de l'utilisation d'un système d'exploitation.

En ce qui concerne les perspectives à court terme, elles peuvent être résumées en deux points :

- L'automatisation de la méthode de génération des programmes de test de haut niveau et la formalisation de la représentation du plan de test et des règles de test;
- L'étude de métriques du taux de couverture pour permettre de quantifier la qualité de la vérification et des tests ;

Les perspectives à plus long terme pour la vérification par simulation des systèmes monopuces sont :

- L'intégration dans un modèle de programmation parallèle des programmes de test de haut niveau pour répondre aux besoins de vérification des systèmes multiprocesseurs.

Références

Vérification en général, outils commerciaux et manuels de référence

[1] Sangiovanni-Vincentelli, A; McGeer, P; Saldnaha,A: "Verification of Electronics Systems". 33rd Design Automation Conference, Las Vegas, 1996.

[2] Synopsys, Formality, Home page: http://www.synopsys.com/products/verification/verification.html

[3] IBM, RuleBase, Home page: http://www.haifa.il.ibm.com/projects/verification/RB_Homepage/index.html

[4] Synopsys, Magellan, Home page : http://www.synopsys.com/products/magellan/magellan.html

[5] Synopsys Vera, Home page: http://www.synopsis.com/products/vera/vera.html

[6] Verisity Specman Elite, Home page: http://www.verisity.com/products/specman.html

[7] SystemC Verification Library (TestBuilder), Home page: http://www.testbuilder.net/

[8] Cadence, Incisive Formal Verifier, Home page: http://www.cadence.com/products/functional_ver/incisive_formal_verifier

[9] Synopsys, Discorvery Verification Platefrom : http://www.synopsis.com/products/solutions/discovery_platform.html

[10] Cadence, Incisive Functional Verification Plateform : http://www.cadence.com/products/functional_ver/

[11] IBM, Sugar home page : http://www.haifa.il.ibm.com/projects/verification/sugar/

[12] PSL Language Reference Manual, http://www.haifa.il.ibm.com/projects/verification/sugar/

[13] SystemVerilog Language Reference Manual, http://www.eda.org/sv/SystemVerilog_3.1_final.pdf

[14] Synopsys, OpenVera, Home page : http://www.openvera.org/

[15] TransEDA, VN-cover Home page : http://www.transeda.com/products/vn-cover/overview.php

[16] Synopsys, VCS Home Page : http://www.synopsys.com/products/simulation/simulation.html

[17] Cadence NC-vhdl, NC-verilog, Home page : http://www.cadence.com/products/functional_ver

[18] GNU implementation of the UNIX macro processor

[19] Lemire, J.-F; Regimbal, S; Bois, G; Savaria, Y; Aboulhamid, E.-M; Baron, A: "Implementing e Assertion Checkers From An SDL Executable Specification", Proc. of Design and Verification Conference, San Jose, USA, 2003.

[20] Regimbal, S; Lemire, J.-F; Savaria, Y; Bois, G; Aboulhamid, E.-M; Baron, A: "Automating Functional Coverage Analysis Based On An Executable Specification", Proc. of the International Workshop on System-on-Chip for Real-Time Applications, Calgary, 2003

Vérification des blocs matériels et ASICs

[21] Kaivola, R; Narasimhan, N: "Formal Verification of the Pentium 4 Floating-Point Multiplier". Design Automation and Test in Europe, 2002.

[22] Iyer, V.V.: " Comparison of Verification Methodologies for Datapath Testing"; IEEE International Microprocessor Test and Verification, Austin, 2003

[23] Bartley, M; Galpin, D; Blackmore, T: "A Comparison of Three Techniques: Directed Testing, Pseudo-Random Testing and Property Checking", 39th Design Automation Conference, New Orleans, 2002.

[24]Evans, A; Silburt, A; Vrckovnik, G; Brown, T; Dufresne, M; Hall, G; Ho, T; Liu, Y: "Functional Verification of Large ASICs", 35th Design Automation Conference, San Francisco, 1998.

[25]Data, K, Das, P.P.: "Assertion Based Verification Using HDVL", VLSI Design, Mumbai (India), 2004

[26]Segev, E; Goldshlager, S; Miller, H; Shua, O; Sher, O; Greenberg, S: "Evaluating and Comparing Simulation Verification vs Formal Verification approach On Block Level Design", IEEE International Conference on Electronics, Circuits and Systems, Tel Aviv, 2004

Vérification des processeurs

[27]Bentley, B: "High Level Validation of Next-Generation Microprocessors". IEEE International High Level Design Validation and Test Workshop, Cannes, 2002.

[28]Bentley, B: "Validating the Intel Pentium4 Microprocessor", 38th Design Automation Conference, Las Vegas, 2001.

[29]Aharon, A; Goodman, D; Levinger, M; Lichtenstein, Y; Malka, Y; Metzger, C; Molcho, M; Shurek, G: "Test Program Generation for Functional Verification of PowerPC processor in IBM". 32nd Design Automation Conference, San Fransisco, 1995.

[30]Fournier, L; Albertman, Y; Levinger, M: "Functional Verification Methodology for Microprocessors Using the Genesys Test-Program Generator". Design Automation and Test in Europe, 1999.

[31]Adir, A; Almog, E; Fournier, L; Marcus, E; Rimon, M; Vinov, M; Ziv, A : "Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification", IEEE Design & Test of Computers, p 84-93, March-April 2004

[32]Casaubieilh, F; McIssac, A; Benjamin, M; Pogodalla, F; Rocheteau, F; Belhadj, M; Eggleton, J; Mas, G; Barrett, G; Berthet, C: "Functional Verification Methodology of Chameleon Processor". 33rd Design Automation Conference, Las Vegas, 1996.

[33]Maladain, D; Palmen, P; Taylor, P; Aharoni, M; Arbetman,Y: "An Effective Approach to Functional Verification of Processor Families". IEEE International High Level Design Validation and Test Workshop, Cannes, 2002.

[34]Taylor, S; Quinn, M; Brown, D; Dohm, N; Hildebrandt, S; Huggins, J; Ramey, C: "Functional Verification of a Multiple-issue, Out-of –Order, Superscalar Alpha Processor – the DEC Alpha 21264 Microprocessor". 35th Design Automation Conference, San Fransisco, 1998.

[35]Hosseini, A; Mavroidis, D; Konas,P: "Code Generation and Analysis for the Functional Verification of Microprocessors". 33rd Design Automation Conference, Las Vegas, 1996.

[36]Chang, Y-S; Lee, S; Park, I-C; Kyung C-M: "Verification of a Microprocessor Using Real World Applications". 36th Design Automation Conference, New Orleans, 1999.

[37]Van Campenhout, D; Mudge, T; Hayes, J: "High-Level Test Generation for Design Verification of Pipelined Microprocessors". 36th Design Automation Conference, New Orleans, 1999.

[38]Mishra, P; Dutt, N: "Automatic Functional Test Program Generation for Pipelined Processors using Model Checking". IEEE International High Level Design Validation and Test Workshop, Cannes, 2002.

[39]Adir, A; Emek, R; Marcus, E: "Adaptive Test Program Generation: Planning for the Unplaned". IEEE International High Level Design Validation and Test Workshop, Cannes, 2002.

[40]Obsidian, Raven Test Program Generator manual : http://www.obsidiansoftware.com/files/manual.pdf

Vérification des systèmes monopuces

[41]Albin, K: “Nuts and Bolts of Core and SoC Verification”. 38th Design Automation Conference, Las Vegas, 2001

[42]Mosensoson, G: “Practical Approaches to SOC Verification” In *Proceedings of DATE User Forum*, 2000.

[43]Emek, R; Jeager, I; Naveh, Y; Bergman, G; Aloni, G; Katz, Y; Farrkash, M; Dozoretz, I; Goldin, A: “X-GEN: a random test-case generator for systems and SoCs”. IEEE International High Level Design Validation and Test Workshop, Cannes, 2002.

[44]Emek, R; Naveh, Y: “Scheduling of Transactions for System-Level Test-Case Generation” IEEE International High Level Design Validation and Test Workshop, San Francisco, 2003.

[45]Geist, D; Biran, G; Arons, T; Slavkin, M; Nustov, Y; Farkas, M; Holtz, Karen; Long, A; King, D; Barret, S: “A methodology For the Verification of a “System on Chip””. 36th Design Automation Conference, New Orleans, 1999.

[46]Berry, G; Blanc, L; Bouali, A; Dormoy, J: “Top Level Validation of system-on-chip in Esterel Studio”. IEEE International High Level Design Validation and Test Workshop, Cannes, 2002.

[47]Adir, A; Shurek, G: “Generating concurrent test-programs with collisions for multi-processor verification”. IEEE International High Level Design Validation and Test Workshop, Cannes, 2002.

[48]Berthet, C: “Going Mobile: The Next Horizon For Multi-million Gate Designs in The Semi-Conductor Industry”, 39th Design Automation Conference, New Orleans, 2002.

[49]Mathys, Y; Châtelain, A: “Verification Strategy for Integration 3G Baseband SoC”, 40th Design Automation Conference, Anaheim, 2003.

Roses

[50]Cesario, W; Nicolescu, G; Gauthier, L; Lyonnard, D; Jerraya A.A: “Colif: a Multilevel Design Representation for Application-Specific Multiprocessor System-on-Chip Design”. In Proc. Twelfth IEEE International Workshop on Rapid System Prototyping, Jun 2001.

[51]Gauthier, L: “Génération de système d’exploitation pour le ciblage de logiciel multitâche sur des architectures multiprocesseurs hétérogènes dans le cadre des systèmes embarqués spécifiques”, Thèse de Doctorat, INPG, Spécialité Microélectronique, laboratoire TIMA, 2001

[52]Nicolescu, G: “Spécification et validation des systèmes hétérogènes embarqués”, Thèse de doctorat, INPG, Spécialité Microélectronique, laboratoire TIMA, 2002.

[53]Lyonnard, D: “Approche d’assemblage systématique d’éléments d’interface pour la génération d’architecture multiprocesseur”, Thèse de doctorat, INPG, Spécialité Microélectronique, laboratoire TIMA, 2003.

[54]Grasset, A; Rousseau, F; Jerraya, A.A: "Automatic Generation of Component Wrappers by Composition of Hardware Library Elements Starting from Communication Service Specification", RSP 2005, Montréal, 2005.

[55]Sarmento, A; Kriaa, L; Grasset, A; Youssef, W; Bouchhima, A; Rousseau, F; Cesario, W; Jerraya, A.A: "Service Dependency Graph, an Efficient Model for Hardware/Software Interfaces Modeling and Generation for SoC Design", International Conference on Hardware - Software Codesign and System Synthesis CODES-ISSS 2005, New York Metropolitan Area, USA, 2005.

[56]Zitterbart, M; Stiller, B; Tantwy, A.N: “A Model for Flexible High-Performance Communication Subsystems”, IEEE Journal on selected Areas in Communications, Vol 11 No 4, 1993.

Annexe 1: Description de l'API de vérification permettant la vérification du sous-système Uncompressed

Cette annexe présente les fonctions de l'API de vérification utilisées lors de l'étude de cas réalisée sur le système PIAGET. On distingue deux ensembles parmi les fonctions de l'API de vérification :

- Une API standard pour le système PIAGET, elle intègre les éléments nécessaires au fonctionnement du système ;
- Une API spécifique au sous-système Uncompressed.

I. L'API de vérification standard pour le système PIAGET

Cette partie de l'API de vérification a été conçue pour permettre :

- L'initialisation du système ;
- La configuration de l'environnement de test.

L'ensemble des fonctions de cette API est destiné à initialiser le système en vue de réaliser la vérification du sous-système Uncompressed.

I.1 L'API d'initialisation et de configuration du système

void CLKGENSetup_TLMLow_long_int(int tlm_master_id)

Cette fonction réalise la configuration du générateur d'horloge du système. La configuration de ce composant est définie pour initialiser l'ensemble des horloges du système nécessaire au fonctionnement du sous-système Uncompressed.

void DDR1Setup_TLMLow_long_int(int tlm_master_id) &

void DDR2Setup_TLMLow_long_int(int tlm_master_id)

Ces fonctions permettent l'initialisation et la configuration des interfaces mémoires contrôlant les deux DDR du système.

void ILCSetup_UDS_TLMLow_long_int(int tlm_master_id)

Cette fonction permet de configurer le contrôleur de niveau d'interruption, sa configuration ne comprend que la gestion du sous-système Uncompressed.

I.2 L'API de configuration de l'environnement de test

void UDS_ENVSetup_TLMLow_long_int(int tlm_master_id)

Cette fonction réalise la configuration des composants de l'environnement de test pour le sous-système Uncompressed.

II. L'API de vérification du sous-système Uncompressed

Cette partie de l'API de vérification a été spécifiquement conçue pour réaliser la vérification du sous-système Uncompressed.

II.1 La configuration du composant : Sync&Conv

void Sync_Config_TLMLow_long_int(int tlm_master_id, int output1, int output2, int SD_HD_RGB_input_type)

Cette fonction réalise la configuration du composant effectuant la synchronisation et la conversion des données provenant de l'environnement de test. Les paramètres de cette fonction sont :

- output1: définit quelle entrée du circuit est dirigée vers le pipe 1 de l'UDS
  - 0 - 1 : sélectionne l'entrée HD/SD
  - 2 : sélectionne l'entrée DDEC1
  - 3 : sélectionne l'entrée DDEC2
- output2: définit quelle entrée du circuit est dirigée vers le pipe 2 de l'UDS
  - 0 - 1 : sélectionne l'entrée HD/SD

- 2 : sélectionne l'entrée DDEC1
- 3 : sélectionne l'entrée DDEC2

- SD_HD_RGB_input_type: définit le type de données vidéo entrant dans le pipe 1
  - 0 : le type est YCrCb 4:2:2
  - 1 : le type est YCrCb 4:4:4
  - 2 : le type est SD RGB, activation de la conversion en YCrCb
  - 3 : le type est HD RGB, activation de la conversion en YCrCb

Ci-dessous le code de la fonction :

```
void Sync_Config_TLMLow_long_int(int tlm_master_id, int output1, int output2, int SD_HD_RGB_input_type) {
    tlm_type tlm = tlm_long_int_TLMLow[tlm_master_id];
    TLMLow_TLMPut_long_int(tlm.low,&SYNC->HD_MODE,(0x1 << uds_syncHD_MODEHDVAL_ENShift) | (0x1 <<
uds_syncHD_MODESW RESET_MODEShift));   // Set the strobe for HDSD input and SW reset
    switch(output1) {
        case 0: //HDSD to output1
        case 1: TLMLow_TLMPut_long_int(tlm.low,&SYNC->OUTPUT1,0x1);
            switch(SD_HD_RGB_input_type) {
                case 0: TLMLow_TLMPut_long_int(tlm.low,&CONV->CONFIG,uds_convCONFIGBYPASS);
                    break; // YcrCb 4:2:2
                case 1: break; // YcrCb 4:4:4
                case 2: TLMLow_TLMPut_long_int(tlm.low,&CONV->CONFIG,uds_convCONFIGRGB_NOT_YCRCB);
                    TLMLow_TLMPut_long_int(tlm.low,&CONV->CONFIG,uds_convCONFIGRGB_YCRCB_MATRIX_SEL);
                    break; // SD RGB
                case 3: TLMLow_TLMPut_long_int(tlm.low,&CONV->CONFIG,uds_convCONFIGRGB_NOT_YCRCB);
                    break; // HD RGB
            }
            break; //HDSD to output1
        case 2: TLMLow_TLMPut_long_int(tlm.low,&SYNC->OUTPUT1,0x2);break; //DDEC1 to output1
        case 3: TLMLow_TLMPut_long_int(tlm.low,&SYNC->OUTPUT1,0x3);break; //DDEC2 to output1
    }
    switch(output2) {
        case 0: //HDSD to output2
        case 1: TLMLow_TLMPut_long_int(tlm.low,&SYNC->OUTPUT2,0x1);break; //HDSD to output2
        case 2: TLMLow_TLMPut_long_int(tlm.low,&SYNC->OUTPUT2,0x2);break; //DDEC1 to output2
        case 3: TLMLow_TLMPut_long_int(tlm.low,&SYNC->OUTPUT2,0x3);break; //DDEC2 to output2
    }
    TLMLow_TLMPut_long_int(tlm.low,&SYNC->HD_MODE,uds_syncHD_MODEHDVAL_EN); // SW RESET
}
```

II.2 La configuration et la gestion du composant : HDPP (pipe 1 & pipe 2)

void HDPPmain_TLMLow_long_int(int tlm_master_id, int NB_PIX, int NB_PIX_BLK, int NB_LINE_TOP, int NB_LINE_BOT, int NB_LINE_BLK_TOP, int NB_LINE_BLK_BOT, int HD_NOT_SD)

void HDPPsec_TLMLow_long_int(int tlm_master_id, int NB_PIX, int NB_PIX_BLK, int NB_LINE_TOP, int NB_LINE_BOT, int NB_LINE_BLK_TOP, int NB_LINE_BLK_BOT, int HD_NOT_SD)

Ces fonctions sont destinées à la programmation des composants HDPP du pipe 1 et du pipe 2. Elles reposent sur la fonction HDPP_Config et permettent ainsi une meilleure réutilisation du code. On notera que chaque HDPP dispose d'une routine de traitement d'interruption.

Ci-dessous le code de la fonction HDPPmain :

```
void HDPPmain_TLMLow_long_int(int tlm_master_id, int NB_PIX, int
NB_PIX_BLK, int NB_LINE_TOP, int NB_LINE_BOT, int
NB_LINE_BLK_TOP, int NB_LINE_BLK_BOT, int HD_NOT_SD){

        HDPP_Config_TLMLow_long_int(tlm_master_id,MAIN_HDPP,NB_PIX,NB
_PIX_BLK,NB_LINE_TOP,NB_LINE_BOT,NB_LINE_BLK_TOP,NB_LINE_BL
K_BOT,HD_NOT_SD);

}
```

void HDPP_Config_TLMLow_long_int(int tlm_master_id, volatile struct s_uds_hdpp *p_HDPP_regs, int nb_pix, int nb_pix_blk, int nb_line_top, int nb_line_bot, int nb_line_blk_top, int nb_line_blk_bot, int hd_not_sd)

Cette fonction réalise la configuration d'un composant HDPP. Les valeurs possibles des paramètres de la fonction sont :

- p_HDPP_regs : définit l'adresse de base des registres du composant ;
- nb_pix : définit le nombre de pixels actifs d'une ligne de l'image ;
- nb_pix_blk : définit le nombre de pixels de blanking d'une ligne de l'image ;
- nb_line_top : définit le nombre de lignes actives d'une image de type TOP (cas des images entrelacées) ou définit le nombre de lignes actives d'une image de type progressive ;

- nb_line_bot : définit le nombre de ligne active d'une image de type BOTTOM, n'est pas utilisé dans le cas des images progressives ;
- nb_line_blk_top : définit le nombre de ligne de blanking d'une image de type TOP (cas des images entrelacées) ou définit le nombre de ligne de blanking d'une image de type progressive ;
- nb_line_blk_bot : définit le nombre de ligne de blanking d'une image de type BOTTOM, n'est pas utilisé dans le cas des images progressives ;
- hd_not_sd : ce paramètre définit si le type du format de l'image est HD ou SD
  - 0 : format SD :
  - 1 : format HD.

Ci-dessous le code de la fonction HDPP_Config :

```
void HDPP_Config_TLMLow_long_int(int tlm_master_id, volatile struct s_uds_hdpp *p_HDPP_regs, int nb_pix, int nb_pix_blk, int nb_line_top, int nb_line_bot, int nb_line_blk_top, int nb_line_blk_bot, int hd_not_sd){
    tlm_type tlm = tlm_long_int_TLMLow[tlm_master_id];
    int total_nb_lines_top = (nb_line_top + nb_line_blk_top)-1;
    int total_nb_lines_bot = (nb_line_bot + nb_line_blk_bot)-1;
    int total_nb_pix;
    total_nb_pix = (nb_pix + nb_pix_blk)-1;
    TLMLow_TLMPut_long_int(tlm.low,&p_HDPP_regs->VIDEO_VSTART_TOP,nb_line_blk_top);
    TLMLow_TLMPut_long_int(tlm.low,&p_HDPP_regs->VIDEO_VSTOP_TOP,total_nb_lines_top);
    TLMLow_TLMPut_long_int(tlm.low,&p_HDPP_regs->VIDEO_VSTART_BOT,nb_line_blk_bot);
    TLMLow_TLMPut_long_int(tlm.low,&p_HDPP_regs->VIDEO_VSTOP_BOT,total_nb_lines_bot);
    TLMLow_TLMPut_long_int(tlm.low,&p_HDPP_regs->VIDEO_HSTART,nb_pix_blk);
    TLMLow_TLMPut_long_int(tlm.low,&p_HDPP_regs->VIDEO_HSTOP,total_nb_pix);
    TLMLow_TLMPut_long_int(tlm.low,&p_HDPP_regs->IT_EN,0x8);
    TLMLow_TLMPut_long_int(tlm.low,&p_HDPP_regs->PRG_LINE_NR,(int) total_nb_lines_top/2 );
    if (!hd_not_sd) TLMLow_TLMPut_long_int(tlm.low,&p_HDPP_regs->CTRL,(0x1 << uds_hdppCTRLVIDEO_EAV_ENShift) | (hd_not_sd << uds_hdppCTRLHD_NOT_SDShift));// this configures the HDPP for SD interlaced (eg: NTSC, PAL) input flow
    else TLMLow_TLMPut_long_int(tlm.low,&p_HDPP_regs->CTRL,(0x1 << uds_hdppCTRLHEDShift)|(0x1 << uds_hdppCTRLVNFShift) | (0x1 << uds_hdppCTRLHD_NOT_SDShift)); // this configures the HDPP for HD 1080i input flow...
    TLMLow_TLMPut_long_int(tlm.low,&p_HDPP_regs->PRG_VSYNC_OFF,(int) nb_line_blk_top/2);
}
```

void HDPPmain_ithandler(int signum)

Ci-dessous le code de la routine de traitement de l'interruption pour le HDPP du pipe 1 :

```
void HDPPmain_ithandler(int signum) {
    int it_raised;
    /* Get the it status of the HDPP of the main UDS */
    it_raised = MAIN_HDPP->IT_STATUS;
    ILC->CLEAR_STATUS_2 = 0x00040000;
    MAIN_HDPP->IT_CLEAR = it_raised;
    /* UnBloc the task */
    __bloc_unbloc(signum);
}
```

II.3 API for HVD configuration

void HVDmain_TLMLow_long_int(int tlm_master_id, int V_DEC_FACTOR, int H_DEC_FACTOR, int OUTPUT1_SELECT, int OUTPUT2_SELECT)

void HVDsec_TLMLow_long_int(int tlm_master_id, int V_DEC_FACTOR, int H_DEC_FACTOR, int OUTPUT1_SELECT, int OUTPUT2_SELECT)

Ces fonctions sont destinées à la programmation des composants HVD du pipe 1 et du pipe 2. Elles reposent sur la fonction HVD_Config et permettent ainsi une meilleure réutilisation du code.

void HVD_Config_TLMLow_long_int(int tlm_master_id, volatile struct s_uds_hvd *p_HVD_regs, int V_dec_factor, int H_dec_factor, int output1_sel, int output2_sel)

Cette fonction réalise la configuration d'un composant HVD. Les valeurs possibles des paramètres de la fonction sont :

- p_VHD_regs : définit l'adresse de base des registres du composant ;
- V_dec_factor : facteur de décimation vertical, valeurs possibles 8,4,2,1, avec le facteur vertical <= facteur horizontal
- H_dec_factor : facteur de décimation horizontal, valeurs possibles 8,4,2,1

- output1_sel : permet de définir le traitement effectué sur la sortie 1 du composant, peut prendre les valeurs suivantes :
  - 0 : bypass ;
  - 1 : décimation ;
- output2_sel : idem output1_sel, mais pour la sortie 2 du composant.

Ci-dessous le code de la fonction HVD_Config :

```
#define  END_TEST ( (int*) 0x05FFFFF8)

void HVD_Config_TLMLow_long_int(int tlm_master_id, volatile struct s_uds_hvd
*p_HVD_regs, int V_dec_factor, int H_dec_factor, int output1_sel, int output2_sel){
    tlm_type tlm = tlm_long_int_TLMLow[tlm_master_id];
    if (H_dec_factor >= V_dec_factor)
    TLMLow_TLMPut_long_int(tlm.low,&p_HVD_regs->CTRL,((H_dec_factor <<
uds_hvdCTRLH_FORMATShift) | (output1_sel << uds_hvdCTRLOUT_SEL1Shift) |
(V_dec_factor << uds_hvdCTRLV_FORMATShift) | (output2_sel <<
uds_hvdCTRLOUT_SEL2Shift) ));
    else
    * END_TEST = 0x5555AAAA; // Stop simulation --> wrong programmation
}
```

II.4 API for NR configuration

void NRmain_TLMLow_long_int(int tlm_master_id, int FIELD_HEIGHT, int FIELD_WIDTH, int BYPASS_ON, int SPATIAL_MODE, int MOTION_MODE)

void NRsec_TLMLow_long_int(int tlm_master_id, int FIELD_HEIGHT, int FIELD_WIDTH, int BYPASS_ON, int SPATIAL_MODE, int MOTION_MODE)

Ces fonctions sont destinées à la programmation des composants NR du pipe 1 et du pipe 2. Elles reposent sur la fonction NR_Config et permettent ainsi une meilleure réutilisation du code.

void NR_Config_TLMLow_long_int(int tlm_master_id, volatile struct s_uds_nr *p_NR_regs, int FIELD_HEIGHT, int FIELD_WIDTH, int BYPASS_ON, int SPATIAL_MODE, int MOTION_MODE)

Cette fonction réalise la configuration d'un composant NR. Les valeurs possibles des paramètres de la fonction sont :

- p_NR_regs : définit l'adresse de base des registres du composant ;
- FIELD_HEIGHT : définit le nombre de lignes de l'image ;
- FIELD_WIDTH : définit le nombre de pixels par ligne ;
- BYPASS_ON : configure le composant en mode bypass, prend les valeurs suivantes :
  - 0 : le composant réalise un traitement ;
  - 1 : le composant est en bypass ;
- SPATIAL_MODE : définit le type de traitement appliqué aux images, il peut prendre les valeurs suivantes :
  - 0 : bypass ;
  - 1 : mode de filtrage faible ;
  - 2 : mode de filtrage fort ;
  - 3 : mode de filtrage fort + filtrage médian ;
- MOTION_MODE : ce mode n'est pas supporté.

Ci-dessous le code de la fonction NR_Config :

```
void NR_Config_TLMLow_long_int(int tlm_master_id, volatile struct s_uds_nr *p_NR_regs, int FIELD_HEIGHT, int FIELD_WIDTH, int BYPASS_ON, int SPATIAL_MODE, int MOTION_MODE){
    tlm_type tlm = tlm_long_int_TLMLow[tlm_master_id];
    int Dummy_reg_val;
    int NLE_THRESHOLD = (int) (FIELD_HEIGHT * FIELD_WIDTH * 0.28)/100;
    if (BYPASS_ON != 1){
        if (SPATIAL_MODE == 0) {
            TLMLow_TLMPut_long_int(tlm.low,&p_NR_regs->CTRL,(BYPASS_ON << uds_nrCTRLBYPASSShift) | (0x0 << uds_nrCTRLSPATIAL_ENABLEShift));
        }
        else if (SPATIAL_MODE == 1) // Weak filter mode{
            TLMLow_TLMPut_long_int(tlm.low,&p_NR_regs->CTRL,(BYPASS_ON << uds_nrCTRLBYPASSShift) | (0x1 << uds_nrCTRLSPATIAL_ENABLEShift));
        }
        else if (SPATIAL_MODE == 2) // Strong filter mode {
            TLMLow_TLMPut_long_int(tlm.low,&p_NR_regs->CTRL,(BYPASS_ON << uds_nrCTRLBYPASSShift) | (0x1 << uds_nrCTRLSPATIAL_ENABLEShift));
            TLMLow_TLMGet_long_int(tlm.low,&p_NR_regs->SPATIAL,Dummy_reg_val);
            TLMLow_TLMPut_long_int(tlm.low,&p_NR_regs->SPATIAL,Dummy_reg_val | (0x1 << uds_nrSPATIALSPATIAL_MODEShift));
        }
        else if (SPATIAL_MODE == 3) // Strong filter mode + median filter{
            TLMLow_TLMPut_long_int(tlm.low,&p_NR_regs->CTRL,(BYPASS_ON << uds_nrCTRLBYPASSShift) | (0x1 << uds_nrCTRLSPATIAL_ENABLEShift) | (0x1 << uds_nrCTRLMEDIAN_ENABLEShift));
            TLMLow_TLMGet_long_int(tlm.low,&p_NR_regs->SPATIAL,Dummy_reg_val);
            TLMLow_TLMPut_long_int(tlm.low,&p_NR_regs->SPATIAL,Dummy_reg_val | (0x1 << uds_nrSPATIALSPATIAL_MODEShift));
        }
        if (MOTION_MODE != 0){
            TLMLow_TLMGet_long_int(tlm.low,&p_NR_regs->MOTION,Dummy_reg_val);
            TLMLow_TLMPut_long_int(tlm.low,&p_NR_regs->MOTION,Dummy_reg_val | (0x0 << uds_nrMOTIONMOTION_BYPASSShift));
        }
        //else  not needed as the bypass is by default
    TLMLow_TLMPut_long_int(tlm.low,&p_NR_regs->NLE_THRESHOLD,(NLE_THRESHOLD << uds_nrNLE_THRESHOLDTHRESHOLDShift));
    }
    else TLMLow_TLMPut_long_int(tlm.low,&p_NR_regs->CTRL,(BYPASS_ON << uds_nrCTRLBYPASSShift));
}
```

II.5 API for T2W configuration

void T2Wmain_LUMA_TLMLow_long_int(int tlm_master_id, int Start_addr, int Pitch, int Max_PnL)

void T2Wmain_ULUMA_TLMLow_long_int(int tlm_master_id, int Start_addr, int Pitch, int Max_PnL)

void T2Wmain_CHROMA_TLMLow_long_int(int tlm_master_id, int Start_addr, int Pitch, int Max_PnL)

void T2Wmain_UCHROMA_TLMLow_long_int(int tlm_master_id, int Start_addr, int Pitch, int Max_PnL)

void T2Wmain_ANC_TLMLow_long_int(int tlm_master_id, int Start_addr, int Pitch, int Max_PnL)

void T2Wsec_LUMA_TLMLow_long_int(int tlm_master_id, int Start_addr, int Pitch, int Max_PnL)

void T2Wsec_CHROMA_TLMLow_long_int(int tlm_master_id, int Start_addr, int Pitch, int Max_PnL)

void T2Wsec_ANC_TLMLow_long_int(int tlm_master_id, int Start_addr, int Pitch, int Max_PnL)

Ces fonctions sont destinées à la programmation des composants T2 du pipe 1 et du pipe 2. Elles reposent sur la fonction T2_Config et permettent ainsi une meilleure réutilisation du code.

void T2_Config_TLMLow_long_int(int tlm_master_id, volatile struct s_uds_t2w_vid *p_T2_regs, int Start_addr, int Pitch, int Max_PnL)

Cette fonction réalise la configuration d'un composant T2. Les valeurs possibles des paramètres de la fonction sont :

- p_T2_regs : définit l'adresse de base des registres du composant ;
- Start_addr : adresse de destination ;
- Pitch : définit le nombre d'éléments par ligne pour respecter l'alignement mémoire ;
- Max_PnL : définit le nombre maximum de pixels par ligne, et de lignes par image.

Ci-dessous le code de la fonction T2_Config :

```
void T2_Config_TLMLow_long_int(int tlm_master_id, volatile struct s_uds_t2w_vid
*p_T2_regs, int Start_addr, int Pitch, int Max_PnL){
    tlm_type tlm = tlm_long_int_TLMLow[tlm_master_id];
    TLMLow_TLMPut_long_int(tlm.low,&p_T2_regs->CONF,uds_t2w_vidCONFEN_STBT2);
    TLMLow_TLMPut_long_int(tlm.low,&p_T2_regs->ADDGEN_START,Start_addr);
    TLMLow_TLMPut_long_int(tlm.low,&p_T2_regs->ADDGEN_PITCH,Pitch);
    TLMLow_TLMPut_long_int(tlm.low,&p_T2_regs->ADDGEN_SIZE,Max_PnL);
    TLMLow_TLMPut_long_int(tlm.low,&p_T2_regs->IT_MASK,0x2); // Set End of Transfert IT
}
```

void T2_UDS_main(int signum)

Ci-dessous le code de la routine de traitement de l'interruption pour les T2 du pipe 1 :

```
void T2_UDS_main(int signum) {
    int it_raised;
    ILC->CLEAR_STATUS_2 = 0x00100000;
    /* Get the it status of the T2 of the main UDS */
    if ((it_raised = MAIN_T2LUMA->IT_STATUS) != 0) {
        MAIN_T2LUMA->IT_CLEAR = it_raised;
    }
    if ((it_raised = MAIN_T2CHROMA->IT_STATUS) != 0) {
        MAIN_T2CHROMA->IT_CLEAR = it_raised;
    }
    if ((it_raised = MAIN_T2ANC->IT_STATUS) != 0) {
        MAIN_T2ANC->IT_CLEAR = it_raised;
    }
    if ((it_raised = MAIN_T2ULUMA->IT_STATUS) != 0) {
        MAIN_T2ULUMA->IT_CLEAR = it_raised;
    }
    if ((it_raised = MAIN_T2UCHROMA->IT_STATUS) != 0) {
        MAIN_T2UCHROMA->IT_CLEAR = it_raised;
    }
    /* UnBloc the task */
    __bloc_unbloc(84);
}
```

RESUME
Les technologies actuelles permettent l'intégration de nombreux composants sur une seule puce. Ces systèmes appelés systèmes monopuce (SoC) sont un assemblage hétérogène de composants logiciels et matériels. La pression pour la qualité et les délais de mise sur le marché font de la validation de ces systèmes un point clé (70% du temps de conception). La vérification de l'intégration des SoCs, réalisée par simulation, consiste à valider les fonctionnalités des composants et leurs interconnexions dans le système. Elle est couramment effectuée par l'exécution de programmes logiciels sur les processeurs embarqués. Ces programmes sont généralement conçus à bas niveau (assembleur, C) ce qui rend difficile la réalisation de scénarii de test complexes nécessitant des mécanismes de synchronisation sophistiqués. De plus, leur utilisation n'est pas suffisante pour effectuer la validation complète d'un système. Ainsi, les contributions permettant d'accélérer la validation sont : (1) la définition d'une méthodologie de validation utilisant plusieurs techniques de vérification adressant les problèmes spécifiques aux SoCs ; (2) la définition d'une nouvelle méthode de vérification de l'intégration s'appuyant sur des programmes de test logiciel de haut niveau reposant sur un système d'exploitation. Cette méthode a été validée sur un système monopuce industriel destiné aux applications de télévision numérique haute définition.

MOTS CLES
Système monopuce, validation, vérification de l'intégration, simulation, programmes de test logiciel de haut niveau, système d'exploitation

TITRE EN ANGLAIS
Global validation method for system on chip

ABSTRACT
Actual technologies facilitate integration of many components onto a single chip. These systems called system on chip (SoC) are a heterogeneous assembly of hardware and software components. As quality and time to market constraints of SoCs increase, validation becomes the key point (70% of the overall design process). Verification of the integration is done through simulation and consists to check component functionalities and interconnections in the system. It is often achieved by executing software programs on the embedded processors. Programs are generally designed at low level (assembly, C) which makes difficult to design complex test scenarios that need sophisticated synchronisation schemes. Furthermore, their use does not enable performing the complete system validation. The main contributions of this work for accelerating validation are: (1) the definition of a validation methodology using different verification techniques targeting specific SoC issues; (2) the definition of a new verification method of the integration based on high level software test programs using an operating system. This method was validated on an industrial SoC aimed at high definition television applications.

KEY WORDS
System on chip, validation, verification of the integration, simulation, high level software test program, operating system

INTITULE ET ADRESSE DU LABORATOIRE
Laboratoire TIMA (Techniques de l'Informatique et de la Microélectronique pour l'Architecture des ordinateurs)
46, avenue Félix Viallet 38 031 GRENOBLE Cedex – France

ISBN : 2-84813-084-9 ISBNE : 2-84813-084-9