Université du Québec
INRS(Centre Énergie Matériaux Télécommunications)

La vérification automatique basée sur un modèle mathématique de réalisations physiques de circuits numériques

par
Ouiza Dahmoune

Thèse présentée
pour l'obtention
du grade de Philosophiæ Doctor (Ph.D.)
en télécommunications

Jury d'évaluation:

Président du jury et examinateur interne	Jean-Charles Grégoire, INRS(Centre Énergie Matériaux Télécommunications)
Examinateur externe	Christopher Fuhrman, Université du Québec, ÉTS Montréal
Examinateur externe	Otmane Ait Mohamed, Concordia University
Directeur de recherche et examinateur interne	Robert de B. Johnston, INRS(Centre Énergie Matériaux Télécommunications)

© Ouiza Dahmoune, 2012.

IEEE Copyright Notice:
http://web.mac.com/bob_celeste/Dahmoune/IEEECopyright.pdf

RÉSUMÉ

Le travail effectué dans le cadre de cette thèse vise la réduction du coût et du temps de vérification d'un circuit numérique ainsi que l'augmentation de la couverture de test. Notre hypothèse est que nous pouvons appliquer une des méthodes de vérification utilisée traditionnellement pour des modèles abstraits, à la version physique du circuit. Ce travail consiste donc en une évaluation expérimentale de notre hypothèse. Toutes nos expériences ont été réalisées sur des prototypes synthétisés sur des cartes FPGAs (Field-Programmable Gate Array). Les résultats de ce travail peuvent être résumés comme suit :

1. Une démonstration de la faisabilité de notre approche : elle étend le vérificateur de modèles TLC (Temporal Logic Checker) pour vérifier la conformité d'une implémentation FPGA par rapport à la spécification mathématique exprimée en TLA+, le langage de spécification "natif" de TLC, il est basé sur TLA (Temporal Logic of Actions). Ce point a fait l'objet de deux publications : [34, 60].

2. Lors de nos travaux nous avons été vite confrontés à un problème de goulot d'étranglement causé par les délais de communication. Nous avons implanté *un analyseur d'accessibilité hardware* (Hardware Reachability Analyser, HRA). Celui-ci est un circuit spécialisé qui permet l'analyse d'accessibilité du circuit physique alors que le vérificateur TLC analyse celle de l'implémentation de référence. Ce point a aussi fait l'objet de deux publications : [33, 36].
   Notons que l'analyseur ainsi que le circuit analysé sont sur un même FPGA. Ceci nous facilite la controlabilité ainsi que la visibilité du circuit d'une part et nous permet d'accélérer l'analyse d'autre part.

3. Le troisième et dernier résultat de notre travail consiste en un ensemble de mesures expérimentales sur différentes plates-formes. Elles sont illustrées par des tables pour les 2 exemples du contrôleur d'ascenseur et du micro-contrôleur à 8 bits, PicoBlaze de Xilinx.

Nous avons donc développé des outils qui peuvent vérifier la conformité d'une réalisation physique d'un circuit numérique par rapport à sa spécification abstraite. Nous voulions que celle-ci réponde à l'ensemble des critères suivants :

- Qu'elle se fasse d'une façon concise en termes mathématiques pour décrire le comportement du circuit, l'ensemble des contraintes auxquelles ce dernier doit obéir et la liste de propriétés qui assurent son bon fonctionnement.
- Le comportement dans le domaine abstrait est utilisé, par la suite, comme référence à laquelle est comparé le comportement de la réalisation physique. Le processus de validation consiste à démontrer un homomorphisme entre entités (états et transitions) des deux mondes abstrait et physique.
- L'automatisation de la génération de l'ensemble des états, des actions et des entrées (des paramètres). Chaque n-tuple (état, action, param1, param2, ...) est soumis à chacune des implémentations abstraite et physique et les sorties respectives sont comparées automatiquement pour déterminer la correspondance.
- L'automatisation de la génération de l'ensemble des bancs d'essais et des moniteurs qui se chargent respectivement de la soumission des n-tuples et de la lecture des résultats correspondants. Le vérificateur se charge aussi lui même du test de conformité.

Le premier avantage de cette approche est qu'elle associe directement le comportement du circuit physique à celui du modèle mathématique de référence. Elle permet aussi une expansion relativement exhaustive tout en réduisant le temps de vérification et en plus de rehausser le niveau d'abstraction, elle élimine la dure tâche d'apprentissage d'un nouveau paradigme de spécification.

Étudiante **Directeur de recherche**

REMERCIEMENTS

Je tiens à remercier très chaleureusement M. Robert deB. Johnston, professeur à l'INRS-EMT, qui a bien voulu me prendre sous son aile et m'encadrer. C'était pour moi un honneur et un grand privilège que de partager avec lui durant mes études doctorales. Je ne le remercierais jamais assez pour tous les enseignements et les conseils qu'il m'a prodigués.

Je remercie M. Jean-Charles Grégoire, professeur à l'INRS-EMT, de m'honorer en présidant le jury de ma thèse. Je remercie également Mrs. Christopher Fuhrman et Otmane Ait Mohamed, professeurs respectivement à l'ETS Montréal et à Concordia University, pour avoir accepté de juger mon travail en qualité d'examinateurs externes.

Je tiens également à remercier mon ex-collègue Mme Baya Oussena, Chercheuse à l'Université Johannes Gutenberg-University Mainz en Allemagne pour m'avoir lu lors de la rédaction de mes 2 premiers articles. Également un grand merci à mon cher mari, Chabane Tibiche, Analyste en Bio-Informatique au CNRC Montréal, qui m'a beaucoup aidée et soutenue sur tous les plans autant des côtés financier et moral que technique. Il m'a été d'un grand support durant toute la durée de cette thèse en plus de m'avoir lue tout au long de mes rédactions. Je ne dois pas oublier mes enfants Rayan, Ianis et Massyl. Même si c'est pour eux que j'ai choisi d'étudier à temps partiel, ils ont constitué tout le long de ces études une bonne raison pour moi de m'accrocher et de croire en ma réussite.

TABLE DES MATIÈRES

RÉSUMÉ iii

REMERCIEMENTS v

TABLE DES MATIÈRES vii

LISTE DES TABLEAUX xiii

LISTE DES FIGURES xv

LISTE DES APPENDICES xvii

LISTE DES SIGLES xix

DÉDICACE xxi

Partie I xxiii

CHAPITRE 1 : INTRODUCTION 1

1.1 Motivation 1
1.2 La vérification de modèle versus réalisation 4
1.3 Objectifs. 6
1.4 L'approche utilisée 7
1.5 Structure du document 8

CHAPITRE 2 : ÉTAT DE L'ART 11

2.1 La vérification de systèmes numériques 12
2.1.1 Le cycle de développement d'un circuit 12
2.1.2 Qu'est ce qu'on vérifie ? 12
2.1.3 Comment on vérifie ? 13
2.2 La vérification de modèles 18
2.3 La vérification du circuit synthétisé 20

2.4 Les architectures FPGA et Les plates-formes cibles 21
2.5 Conclusion 22

CHAPITRE 3 : LES CONCEPTS MATHÉMATIQUES **23**
3.1 La logique temporelle 24
3.2 Le verificateur TLC 25
3.2.1 Spécification d'un comportement 26
3.2.2 Spécification de propriétés 27
3.2.3 Le concept de raffinement 27
3.2.4 Une Simplification : Comportement Déterministe 29
3.3 Conclusion 29

CHAPITRE 4 : TEST D'IMPLÉMENTATIONS PHYSIQUES **31**
4.1 Survol des articles 31
4.2 Établissement de la connexion 33
4.2.1 Description de notre méthodologie 33
4.2.2 Des points d'accès logiques 36
4.2.3 Des points d'accès physique 37
4.3 Amélioration des performances 37
4.3.1 Le lien de communication et le goulot d'étranglement 37
4.3.2 Le mécanisme d'anticipation 38
4.3.3 Un analyseur d'accessibilité implanté en matériel 38
4.4 Conclusion 39

CHAPITRE 5 : RÉSULTATS, CONCLUSION ET PERSPECTIVES **41**
5.1 Résultats 41
5.2 Conclusion 43
5.3 Perspectives 47

Partie II **49**

CHAPITRE 6 : ARTICLE 1 . . . . **51**

RÉSUMÉ **53**

6.1 Abstract 53

6.2 Introduction 53

6.3 State of the art 55

6.3.1 Formal methods 56

6.3.2 Raising the level of abstraction in the HDL 56

6.3.3 Automating testing : test-benches 56

6.4 Motivation : Drawbacks of HDL Simulation 57

6.5 Applying Reachability Analysis to Physical Implementations 58

6.5.1 Raising the Level of Abstraction 59

6.5.2 The TLC Model-Checker 59

6.5.3 Over-riding the Model-Checker's native model. 60

6.5.4 Refinement criteria. 61

6.6 Proposed Configuration and Design Flow 62

6.7 Case study : An elevator controller 63

6.7.1 The TLA+ Model Domain. 63

6.7.2 The Java Interface 65

6.7.3 The External Implementation 65

6.8 Communication Bottleneck. 65

6.9 Preliminary Results 66

6.10 Conclusion 67

CHAPITRE 7 : ARTICLE 2 . . . . **69**

RÉSUMÉ **71**

7.1 Abstract 71

7.2 Introduction 71

7.2.1 Summary 71

7.2.2 Background and Motivation 72

7.2.3 General Approach ("MRAPT") 73

7.3 Theoretical Framework 74

7.3.1 Rationale for TLA+ Tools 74

7.3.2 State-Transition Systems and Model-Checking 75
7.3.3 Invariance and Refinement Properties 75
7.3.4 Reachability Analysis 76
7.3.5 Verifying a Processor's Instruction Set 77
7.4 Testbench Configuration 78
7.4.1 Main Model Domain Entities (TLA+) 78
7.4.2 Main Hardware Implementation Entities (VHDL) 79
7.5 The PicoBlaze Micro-Controller 81
7.5.1 State Space 81
7.5.2 Instruction Set 81
7.5.3 Instruction Semantics 82
7.6 Mathematical Reference Model (TLA+) 82
7.6.1 PicoBlaze Instruction Set Architecture (ISA) 83
7.6.2 Abstract Reference Implementation (ARI) 84
7.6.3 Test Suites 84
7.6.4 The Refinement Checker 85
7.7 Hardware Implementation (VHDL) 85
7.7.1 PicoBlaze Top-Level Entity 85
7.7.2 PicoBlaze VHDL Design Hierarchy 86
7.8 Experiments 87
7.8.1 Test Suites 87
7.8.2 Results 87
7.9 Conclusion 88

CHAPITRE 8 : ARTICLE 3 . . . . **89**

RÉSUMÉ **91**
8.1 Abstract 91
8.2 Introduction 92
8.3 State of the art 93
8.4 Connecting TLC to an FPGA Prototype 95
8.4.1 The State Space for Elevator in the abstract world (TLA+) 95

8.4.2 The State Space for Elevator in the concrete world (VHDL) . . . 96
8.4.3 The State Space for Elevator in the connecting world (JAVA) . . . 97
8.5 Communication Bottleneck. 98
8.6 Software Look-Ahead (Predictive) Caches 99
8.6.1 The No Cache (NC) model 100
8.6.2 The Single State Cache (SSC) 100
8.6.3 The Multi-States Cache (MSC) 100
8.6.4 The Multi-Levels Cache (MLC) 101
8.6.5 The One-Shot Cache (OSC) 101
8.7 A Hardware Reachability Analyzer 104
8.8 Hardware Look-Ahead (Predictive) Caches 104
8.9 Some Results 105
8.10 Conclusion 108

CHAPITRE 9 : ARTICLE 4 . . . . **109**

RÉSUMÉ **111**
9.1 Abstract 111
9.2 Introduction 111
9.3 State of the art 113
9.4 Why A Hardware Reachability Analysis ? 114
9.5 The Reachability Analysis Process 115
9.6 The Graph Expander (Grex) 116
9.6.1 The detailed Grex structure 116
9.7 Where and When ERAIC intervenes in the Design Flow 121
9.8 How to connect the post-silicon implementation to ERAIC 122
9.9 Conclusion 122

LISTE DE RÉFÉRENCES . . . . **lxxi**

LISTE DES TABLEAUX

5.I Quelques temps de vérification 43

6.I Some verification times 67

7.I 3 Test Suites P1, P2 and P3 87

7.II The 3 Test Suites TLA+ imp. Observations 87

7.III The 3 Test Suites FPGA imp. Observations 88

8.I The Return-Trip Time (RTT) 99

8.II Some verification times 107

8.III A factor of 4 times improvement 107

LISTE DES FIGURES

1.1 La configuration générale 8

2.1 Les différents niveaux de vérification de circuits 14

4.1 La chaîne de vérification du contrôleur d'ascenseur 34
4.2 Le diagramme d'homomorphisme entre les domaines R (Réalisation) et S (Spécification) 35

7.1 Model Domain Entities 79
7.2 Physical Implementation Domain Entities 80

8.1 The Verification Process 93
8.2 The Communication Link Components 96
8.3 Software Caches Performances for the Elevator Controller One with (8886 states,36977 transitions) and Invariant Type(T) versus Invariants Type & Refinement(TR) 102
8.4 Software Caches Performances for the Elevator Controller Two with (184938 states, 527821 transitions) and Invariant Type (T) versus Invariants Type & Refinement (TR) 103
8.5 Hardware Reachability Analyzer Data Paths and Operators 106
8.6 Reachability Analyzer Virtual Memory Mapping 106

9.1 The Verification Process 112
9.2 WBGrex Data Paths and Operators 117
9.3 Reachability Analyser Data Paths and Operators 119
9.4 Reachability Analyser Virtual Memory Mapping 119

I.1 WBGrexTop Entity and Architecture xxxv
I.2 TestManager(WBGrexEntity) Architecture xxxvi
I.3 Grex Data Flow xxxviii
I.4 Grex Operators Architecture xxxix
I.5 Graph Expander Architecture xl
I.6 Grex Operators Data Paths xli

I.7 Action Generator Architecture xlii
I.8 Current State Reader Architecture xlii
I.9 A Successor Generator (SGEN) Architecture xliii
I.10 A Successor State Writer (SSW) Architecture xliii
I.11 Read/Write BLOCK Architecture xliv
I.12 Linked searching List Architecture xlvi
I.13 Binary searching List Architecture xlvii
I.14 A Hashtable Reader/Writer (HTRW) Architecture xlviii
I.15 Tansition Table Writer Architecture xlviii
I.16 Grex Control Flowchart xlix
I.17 Round-Robin 6-User Mutex Arbiter Architecture l
I.18 6-Master, 2-Slave Wishbone Network (INTERCONN6x2) Architecture li
I.19 6-Master, 2-Slave INTERCONN6x2 wrapper (WBNet6x2) Architecture lii

III.1 Le diagramme des classes, vue globale lxv
III.2 L'API Java lxvi
III.3 L'API Java (suite 1) lxvii
III.4 L'API Java (suite 2) lxvii
III.5 L'API Java (suite 3) lxviii
III.6 L'API Java (suite 4) lxviii

LISTE DES APPENDICES

Appendice I : Un analyseur d’accessibilité en matériel . . . . **xxv**
I.1 Introduction xxv
I.2 L’algorithme d’analyseur d’accessibilité xxvi
I.2.1 Le Pseudo-Code xxvii
I.2.2 Une Implémentation Structurée en C. xxvii
I.3 Pourquoi une analyse d’accessibilité physique ? xxxiii
I.4 L’analyseur d’accessibilité dans le contexte physique xxxiv
I.5 Les différents composants de l’analyseur d’accessibilité physique ? . xxxiv
I.5.1 Les structures de données xxxvii
I.5.2 Les Opérateurs du Graph Expander xxxvii
I.5.3 Le Contrôleur Grex xlv
I.5.4 Le Réseau d’interconnexion xlv

Appendice II : Quelques Modules TLA+ . . . . **liii**
II.1 Spécification des charges (Requirements) liii
II.2 L’Implémentation de Référence lvi
II.3 Le critère de Raffinement lx

Appendice III : L’interface Java . . . . **lxiii**
III.1 Le module de sur-définition (over-riding) Java lxiii
III.2 Le diagramme des classes, vue globale lxv

Appendice IV : Sommaire de la méthodologie MRAPT . . . . **lxix**

LISTE DES SIGLES

ATEs	Automatic Test Equipment
ATPG	Automated Test-Pattern Generation
BIST	Built-in Self Test
BDD	Binary Decision Diagrams
CTL	Computation Tree Logic
EDA	Electronic Design Automation
ERAIC	Embedded Reachability Analyzer And Invariant Checker
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
HOL	Higher Order Logic
INRS	Institut National de la Recherche Scientifique
IPs	Intellectual properties
JNI	Java Native Interface
LTL	Linear-time Temporal Logic
OBDD	Ordered Binary Decision Diagrams
Promela	PROcess MEta-LAnguage
PVS	Prototype Verification System
SAT	Boolean Satisfiability Testing
SMV	Symbolic Model Checker
SPIN	Simple Promela INterpreter
TLA	Temporal Logic of Actions
TLC	Temporal Logic Checker
TLMs	Transaction Level Modeling

(Dédicace) Je dédie cette thèse à mon très cher mari, à mes enfants Rayan, Ianis et Massyl, sans oublier tous ceux et celles qui ont contribué de près ou de loin à ma réussite. Je la dédie aussi à ma mère adoptive, mes 2 parents biologiques et mon frère Ali. Je ne vais pas oublier tous ceux et celles qui ont lutté et qui luttent encore pour la reconnaissante de l'identité Kabyle, ma chère identité.

Partie I

CHAPITRE 1

INTRODUCTION

1.1 Motivation

Pour s'assurer du bon fonctionnement d'un circuit numérique, on procède à sa vérification. Celle-ci est répétée à différents niveaux du développement depuis sa conception jusqu'à son utilisation. Du fait que les méthodes de vérification existantes ne couvrent pas la totalité de l'espace des états du circuit physique, d'une quelconque manière, il y a des chances que des erreurs ne soient détectées que lors de l'utilisation. Les travaux sur la vérification de circuits remontent aux années 80 mais le problème devient de plus en plus complexe et coûteux et les défis de plus en plus grands à mesure que le circuit devient de plus en plus dense (on parle de milliards de transistors sur une même puce). Certaines estimations affirment que la vérification constitue 70% du temps et coût du cycle de développement d'un circuit [26, 44, 52, 91–93].

Par ailleurs, l'utilisation de circuits numériques s'est généralisée à tous les aspects de la vie quotidienne, même à des applications très critiques : le système de santé, les finances, le nucléaire, l'aviation, l'exploration des planètes etc. Des conséquences d'une défaillance quelconque dans une puce peuvent être très graves voire désastreuses et catastrophiques. Des exigences quant au bon fonctionnement de la puce s'avèrent donc être de plus en plus grandes. Il ne serait pas tolérable qu'un système mettant en jeu notre sécurité physique ou financière soit défaillant. Par conséquent, assurer **l'absence erreur** devient la principale motivation et constitue un défi majeur.

Suite à la complexité grandissante des circuits, on s'est mis à automatiser les processus de conception et de vérification et à développer de nombreux outils et méthodologies EDA (Electronic Design Automation) [41]. La vérification de modèles, pour la forme abstraite de circuits, s'avère être efficace et tend à être largement acceptée et utilisée dans l'industrie [15]. Malgré une multitudes de stratégies et d'outils pour améliorer le processus de vérification des systèmes physiques, les défis demeurent très grands.
Dans [26, 94], on affirme que de nouvelles stratégies innovatrices sont absolument nécessaires. À notre sens, notre méthodologie en constitue une. Elle étend la vérification de

modèles, qui est une méthode traditionnelle de spécification et de vérification formelle utilisée jusque là pour la forme abstraite de circuits, au domaine des systèmes physiques. De nombreuses initiatives disent utiliser ou rapprocher ces méthodes des systèmes physiques par une liaison indirecte. Plusieurs recueillent des informations, lors de la validation, qu'ils soumettent au modèle abstrait dans le but d'aider à déboguer l'implémentation physique. C'est le cas quand on s'intéresse aux traces des instructions lors de l'exécution d'un processeur [5, 94, 102] ou aux échanges entre le processeur et son environnement extérieur telles que les mémoires ou les caches [46]. Dans notre approche, nous réalisons une connexion directe entre l'implémentation abstraite et la réalisation physique en utilisant le concept de *raffinement* d'une part et le prototypage sur FPGA d'autre part. Nous transposons, par une fonction mathématique, la totalité de l'exploration du domaine physique au domaine abstrait, pour démontrer leur équivalence.

Dans la vérification traditionnelle, la simulation et la vérification formelle (démonstrateur de théorèmes, vérificateur de modèles et vérificateur d'équivalence) ou semi-formelle considèrent juste la réalisation abstraite du circuit ; elles sont, de cette façon, en mesure de prouver l'existence d'erreur mais elles ne peuvent pas garantir leur inexistence. L'ensemble de propriétés qu'un système doit avoir sont écrites sous forme de spécification qui constitue la référence à laquelle l'implémentation modèle est comparée. Un circuit est donc (1) modélisé et vérifié dans l'abstrait et (2), réalisé physiquement et testé. Cette séparation en deux phases pose la question : qu'est-ce qui vérifie la version physique et comment ? Selon [115] les deux procédés de vérification et de testing sont complètement indépendants et différents.

Les Préalables

Ayant au préalable effectué un travail sur l'utilisation de langages fonctionnels [31] pour la spécification de circuits numériques, nous avions déjà en tête que ces langages sont plus appropriés. En effet, la description d'un circuit en termes d'un ensemble de fonctions au sens mathématiques, liées entre elles de façon à ce que les sorties des unes constituent des entrées pour d'autres est non seulement d'apparence très naturelle mais aussi conceptuellement plus fidèle et plus adéquate. Cette schématisation illustre très clairement le fait qu'une fonction peut être en entrée à une autre fonction et c'est ce qu'un langage fonctionnel permet d'exprimer fidèlement ; une fonction pouvant retourner une autre fonction ou être passée en paramètre à une autre fonction sans l'utilisation de variable intermédiaire.

TLA+ est non seulement un langage fonctionnel muni d'une sémantique et d'une capacité d'expression très puissantes, sa syntaxe est également basée essentiellement sur le langage mathématique standard. La lecture d'un circuit exprimé dans ce langage est très facile pourvu qu'on ait une bonne base en mathématique et qu'on soit assez à l'aise avec le concept de récursivité.

Un autre préalable qui a beaucoup joué pour que nous nous retrouvions dans le contexte de ce travail est notre apprentissage sur la synthèse comportementale ainsi que celui sur la simulation et la synthèse de circuits numériques. Nous avions eu l'opportunité de réaliser combien est laborieux et très coûteux en temps le processus de synthèse d'un circuit sur FPGA. Pour une couverture, très loin d'être exhaustive, on peut passer des journées entières, voire des semaines pour simuler un circuit dans l'espoir de localiser l'origine d'un dysfonctionnement si ce dernier est détecté. Si tel est le cas pour des petits exemples de travaux pratiques, on peut facilement imaginer ce que ça peut être pour des circuits complexes.

Notre hypothèse de départ est que cette méthodologie, si elle est appliquée aux systèmes physiques, permettra, en plus de considérer le circuit concrètement, les avantages suivants :

- Nous épargner les inconvénients de la simulation qui sont essentiellement la lenteur et l'impossibilité d'une couverture optimale ; la simulation nous permet d'atteindre un certain degré de confiance en notre circuit mais jamais une confiance totale. L'avantage qu'offrait la simulation, par la possibilité de refaire autant de fois possible une vérification, est offert dans notre cas par la reconfigurabilité du circuit.

- Relever le niveau d'abstraction tout en facilitant et en renforçant l'étape de spécification. Le comportement physique est comparé, au plus haut niveau d'abstraction, au comportement abstrait.

- Automatiser la génération des bancs d'essais ainsi que des vecteurs de tests et du même coup les faire assumer par le vérificateur lui même au même titre que la tâche de comparaison au comportement physique.

- Appliquer l'analyse d'accessibilité au circuit physique. Notons que pour atteindre des performances maximales, nous avons implémenté l'analyse d'accessibilité du circuit physique en hardware. Nous avons donc deux analyses d'accessibilité abstraite et

physique et leurs résultats respectifs sont comparés dans le monde abstrait via une fonction mathématique. Cette dernière doit démontrer un homomorphisme entre les deux systèmes pour prouver leur conformité. Nous nous offrons, de cette manière, la possibilité de finir l'analyse d'accessibilité physique pour entamer l'abstraite et d'éliminer complètement l'attente du vérificateur des résultats de calculs du circuit physique. Cette façon de faire est innovatrice, elle ouvre la voie à toute une combinaison d'approches pour remédier au problème d'explosion d'états par l'agrandissement de l'espace occupé par le graphe d'expansion. Nous pouvons citer le parallélisme, l'utilisation de mémoires externes, la représentation symbolique et la compression de l'espace des états.

1.2 La vérification de modèle versus réalisation

Comme son nom l'indique la vérification de modèle (model checking) comme SPIN [53, 54], TLC [75, 76], ou SMV [84, 85], analyse un modèle, relevant du monde abstrait. Or le passage de ce dernier à la réalisation physique n'est pas exempt d'erreurs ; il exigerait nécessairement d'une part la validité de la modélisation et d'autre part la fidélité de la réalisation physique par rapport au modèle validé. En d'autres termes, en cas d'erreurs dans l'implémentation physique, leur origine pourrait être liée à l'implémentation elle-même ou à la modélisation.

La distinction entre les domaines abstrait et physique est fondamentale dans la plupart des branches d'ingénierie. Prenons l'exemple d'un scientifique qui étudie un phénomène naturel : À partir du phénomène observé, il construit un modèle mathématique qu'il validera par rapport au phénomène réel. D'autre part un ingénieur commence par une spécification abstraite du système telle une description textuelle des contraintes, construit une réalisation physique dont il vérifiera la conformité par rapport à la spécification. La différence entre ces deux cas réside dans le fait que le scientifique dispose d'une référence absolue qu'est la nature ; si son modèle ne prédit pas convenablement le comportement du phénomène il peut directement en déduire que le modèle n'est pas valide. La décision n'est pas aussi facile à trancher pour l'ingénieur, du fait que l'erreur peut provenir aussi bien du modèle que de l'implémentation.

Pour parvenir à garantir l'absence d'erreur dans le circuit physique par la vérification de

modèles, il est nécessaire d'étendre le procédé de vérification à la réalisation physique. Ceci constitue donc notre hypothèse de départ ; plutôt que de se contenter de vérifier seulement le modèle, notre approche vérifie la conformité du comportement du circuit physique par rapport à celui du modèle abstrait. Cette vérification se fait par une interconnexion entre le vérificateur et le circuit. Dans notre application les tests sont re-dirigés au système externe et le vérificateur génère automatiquement des bancs d'essais. Ces derniers constituent un support commun à la réalisation physique, par le mécanisme de sur-définition, ainsi qu'au modèle. Le vérificateur compare les deux résultats obtenus en se servant du concept de raffinement.
Au tout début de notre inscription à ce projet de recherche, des liens vérificateur-cible existaient pour les plates-formes suivantes :

- MIPS R2000/R300 : Un simulateur d'architecture RISC à 32 bits.
- PIC18F452 : Un micro-contrôleur à 8 bits via une ligne série RS232-C.
- Des implémentions Java.

Ces liens visaient seulement la vérification de systèmes logiciels, et nous voulions traiter des systèmes matériels, c.à.d des circuits numériques (hardware). Alors, nous avons proposé l'extension de cette technologie aux architectures FPGA et, afin d'évaluer notre méthodologie, nous avons examiné des réalisations électroniques sur les plates-formes suivantes :

- Altéra Cyclone EP1C6 : Un FPGA intégrant le processeur NIOS via une ligne série RS232-C.
- Altéra ACEX 10K : Un FPGA dit Morph-IC via un lien USB1.1.
- Altéra Cyclone EP1C12 : Un FPGA dit Easy FPGA via un lien USB1.1.
- Xilinx SEB3 avec un outil Quick USB via un lien USB2.0.
- Opal Kelly XEM3001 avec SRAM via un lien USB2.0.
- Digilent NEXYS2 avec SRAM via un lien USB2.0.
- Digilent ATLYS avec SRAM via un lien USB2.0.

1.3 Objectifs

Notre objectif est de réduire les coûts liés à la vérification, ainsi que d'augmenter leur qualité (couverture). Nous visons le développement de techniques basées sur un modèle mathématique pour tester le fonctionnement de réalisations physiques ; nous voulons offrir la possibilité d'appliquer le procédé de vérification de modèles de réalisations abstraites pour une vérification formelle de réalisations physiques. Ceci a nécessité dans une première étape, l'établissement d'un lien physique entre le vérificateur et le circuit. Dans une deuxième étape, nous avons développé des stratégies pour optimiser l'utilisation du lien de communication et réduire le temps de verification.

Nous nous sommes donc assignés ces deux objectifs majeurs et pour les atteindre nous avons combiné les techniques suivantes :

- Développement d'un circuit spécialisé pour mieux supporter la vérification sur la plate-forme cible.
- Utilisation de la technologie USB2.0 haute-vitesse .
- Développement d'algorithmes prédictifs pour résoudre le problème lié à la latence du lien USB.
- Développement d'un analyseur d'accessibilité en hardware

Nous visons :

- Une vérification potentiellement exhaustive
- Un temps de vérification réduit
- Éliminer la corvée de créations de bancs d'essai (test benches) et des vecteurs de tests comme cela se fait dans le contexte des simulations VHDL.
- Se servir d'une modélisation mathématique permettant de définir avec précision les contraintes et des propriétés à un haut niveau d'abstraction.

1.4 L'approche utilisée

L'idée de base dans ce projet est de comparer de façon automatique le comportement d'une implémentation physique par rapport à un modèle mathématique de référence. Les éléments clés qui en ressortent sont :

1. L'automatisation complète du processus de vérification, le vérificateur se charge de la génération des états et de toutes les actions possibles pour chacun d'eux, de les soumettre et de récupérer les résultats correspondants des deux implémentations abstraite et physique.
2. La comparaison des résultats obtenus pour prouver un homomorphisme en cas de conformité ou pour générer une trace de non conformité dans le cas contraire.
3. L'implémentation physique : la vérification concerne la réalisation physique et non un modèle.
4. Le modèle de référence : une description précises des contraintes indépendemment des détails de la réalisation physique.

Ce projet a nécessité deux systèmes interconnectés (voir figure 1.1) répartis sur deux plates-formes distinctes :

1. Un PC hôte. Celui-ci héberge la partie logique de notre système qui est structurée en deux sous-composantes :
   - Une composante intégrant le vérificateur TLC et le modèle TLA+ du système à vérifier
   - Une interface Java qui permet de relier le modèle TLC à l'implémentation physique (voir figure communication components).
2. La carte FPGA. Celle-ci héberge la partie physique de notre système qui est structurée en trois sous composantes :
   - l'implémentation physique de notre système.
   - Un "TAP" (Test Acces Point) ou point d'accès au test.
   - L'analyseur d'accessibilité implémenté en hardware (voir Chapitre 9).

3. Un lien de communication reliant le PC à la plate-forme cible permettant au vérificateur de contrôler et d'observer le circuit sous test (voir Chapitre 8).

1.5 Structure du document

Ce document est structurée en 2 parties. La première est composée de cinq chapitres. Après ce chapitre d'introduction et de mise en contexte du sujet de notre thèse, nous avons consacré le chapitre 2 pour l'état de l'art. Le chapitre 3 présente l'ensemble des notions utilisées en TLA pour la spécification d'un système. Le chapitre 4 est consacré à notre méthodologie. L'objectif de ce chapitre est de détailler notre méthodologie. Comme la majeure partie de cette matière a été publiée ou est en voie de l'être, pour des fins de non duplication du contenu, nous y pointerons souvent aux chapitres de la deuxième partie de ce document. Ce chapitre est conçu pour les compléter quand c'est nécessaire. Nous finissons cette partie par le chapitre 5 pour conclure et résumer les points essentiels de notre travail.
La deuxième partie regroupe sous forme de chapitres nos quatre articles publiés. Le chapitre 6 explique notre démarche sur la base d'un exemple simple du contrôleur d'ascenseur. Le chapitre 7 présente l'application de notre méthodologie pour la vérification d'un microprocesseur physique. Le chapitre 8 expose les différentes approches que nous avons mises en place pour résoudre le problème du goulot d'étranglement du lien de communication. Le chapitre 9 illustre le fonctionnement et la structure de notre *un analyseur d'accessibilité physique* qui nous permet d'atteindre la meilleure performance. Celui-ci, vu son importance dans notre méthodologie, est aussi détaillé dans l'annexe I. Les annexes II et III présentent respectivement des exemples de modules TLA+ et de classes Java pour le cas

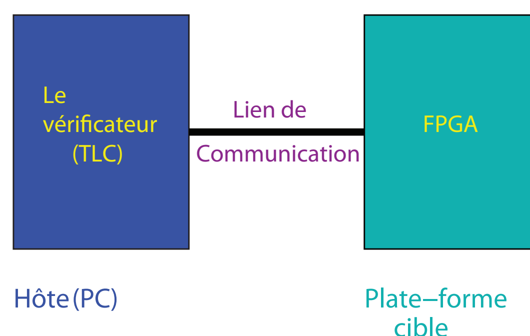


Figure 1.1 – La configuration générale

du micro-contrôleur PicoBlaze qui est décrit en détails dans [25]. L'annexe IV présente un petit sommaire de notre méthodologie.

CHAPITRE 2

ÉTAT DE L'ART

Le champ de la vérification de systèmes étant très vaste, elle est abordée selon plusieurs orientations et à différents niveaux dans la chaîne de développement. On a considéré par exemple le formalisme de spécification ; [48] illustre pourquoi la logique d'ordre supérieur constitue un bon formalisme pour la spécification et la vérification de systèmes physiques. Dans cette direction, plusieurs travaux [27, 55, 97, 104] ont considéré la compilation et la synthèse de circuits à partir de langages fonctionnels. Beaucoup d'autres travaux ont considérés d'autres orientations telle celle de relever le niveau d'abstraction et des efforts de standardisation sont encore en cours [3, 41, 93].

En traitant de fiabilité de programmes, Dijkstra [40] utilise le mot mécanisme au lieu de programme et exprime son sentiment que les considérations concernant un programme sont aussi bien appropriées et applicables au matériel. Ses principales conclusions sont en premier lieu : considérer la structure interne du mécanisme et non le traiter comme une boite noire. Il souligne aussi qu'on peut utiliser le testing (débogage) de programme pour montrer la présence d'erreurs mais jamais leur absence.

Certains se sont intéressés à développer des outils de débogages automatiques du code HDL source afin de localiser l'erreur [105].

Des travaux donc touchant à ce thème sont innombrables [115]. L'importance de la vérification de matériel n'est donc plus à démontrer, elle constitue une préoccupation majeure. Le but ultime est de parvenir au niveau **ZÉRO erreur** mais toute les tentatives à ce jour ne sont parvenues qu'à minimiser ces erreurs tout en réduisant le temps, les efforts ainsi que le coût du processus de conception d'un circuit électronique. Nous allons nous consacrer, dans le cadre de cette thèse, à l'application de la vérification de modèles (Model checking) aux systèmes numériques.

2.1 La vérification de systèmes numériques

2.1.1 Le cycle de développement d'un circuit

Avant d'atteindre la phase d'un produit livrable, le circuit est dans une première étape décrit dans un texte illustrant ses fonctionnalités et l'ensemble des contraintes et propriétés qui l'accompagnent. Dans un souci d'augmenter la productivité, de simplifier et de mieux maîtriser le circuit, vue sa grande complexité, la tendance a été et est de spécifier le circuit à un très haut niveau d'abstraction [18, 93] afin d'éliminer le maximum des détails de bas niveau.

La spécification initiale peut être validée en vue de s'assurer qu'elle respecte l'ensemble des contraintes. On procède par la suite au raffinement de cette spécification pour obtenir une autre spécification d'un niveau plus bas en introduisant plus de détails. Ce processus peut être répété un certain nombre de fois jusqu'à la réalisation physique. La spécification de niveau i constitue une réalisation par rapport à celle du niveau $i - 1$. À chaque étape, la réalisation résultat a besoin d'être validée par rapport à sa spécification.

2.1.2 Qu'est ce qu'on vérifie ?

La vérification d'un circuit consiste à prouver la conformité de sa réalisation (logique ou physique) par rapport à sa spécification. Celle-ci peut être partielle ou globale et la qualité de la réalisation en dépend directement du fait qu'elle réalise la spécification. On a dénombré plusieurs concepts de conformité [23]. Les propriétés souvent recherchées sont celles de “liveness” et “safety” auquel cas la vérification consisterait donc à prouver leur véracité.

2.1.2.1 Qu'est ce qu'on vérifie dans la réalisation

Tout circuit numérique est conçu pour assurer une fonction au sens mathématique du terme : pour les mêmes entrées nous devons toujours avoir les mêmes sorties. Si nous voulons un circuit optimisé, nous devons le réaliser de façon à ce qu'il délivre des sorties en un temps minimal et qu'il occupe un espace minimal. On peut aussi être intéressé à ce que notre circuit consomme un minimum d'énergie. Dans le cadre de notre thèse, nous ne nous intéressons qu'à l'aspect fonction mais notre méthodologie permet de détecter tout dysfonctionnement lié aux trois raisons suivantes :

- Une formulation incorrecte de la fonction.
- Des erreurs dans les connexions physiques.
- Des problèmes liés au timing telle que la non synchronisation entre composants.

On pourrait aussi avancer que notre approche permet de détecter toute erreur, peu importe son origine du moment que nous visons l'exploration complète de l'espace des états du circuit physique.

2.1.3 Comment on vérifie ?

La façon de vérifier dépend directement de la forme du circuit considèree (voir Figure 2.1), celui-ci pouvant être dans sa forme abstraite (pre-silicon) ou dans sa forme physique (post-silicon). Nous survolons donc dans ce qui suit les différentes formes de vérification.

2.1.3.1 Le test connu plutôt sous le nom "testing"

Ce procédé consiste à vérifier la réalisation physique, il peut être déterministe ou aléatoire. Il constitue le **premier** procédé de vérification pour lequel il y a eu développement de stratégies d'amélioration telles la génération de test patterns [2] et le développement des ATPG (Automated Test-Pattern Generation) [4, 67, 71].
Avec la densité grandissante des circuits et l'avènement de la technologie multi-couches, l'accès directe aux points d'accès (broche ou "pin"), pour des fins de test, est devenu très limité. Pour y remédier, on a eu recours à l'approche BIST (Built-in Self Test) qui consiste à intégrer aux circuits une logique dédiée au test. Le groupe JETAG (Joint European Test Access Group) a initié la standardisation de la solution Boundary-Scan (BS) qui consiste en un registre à décalage série à insérer aux entrées et sorties d'un circuit à tester. On parle dans ce cas de la méthodologie DFT (Design For Test) (voir chapitre 17 de [52]). D'autres standards comme STIL et CTL ainsi que d'autres outils comme ATEs (Automatic Test Equipment) ont été développés.

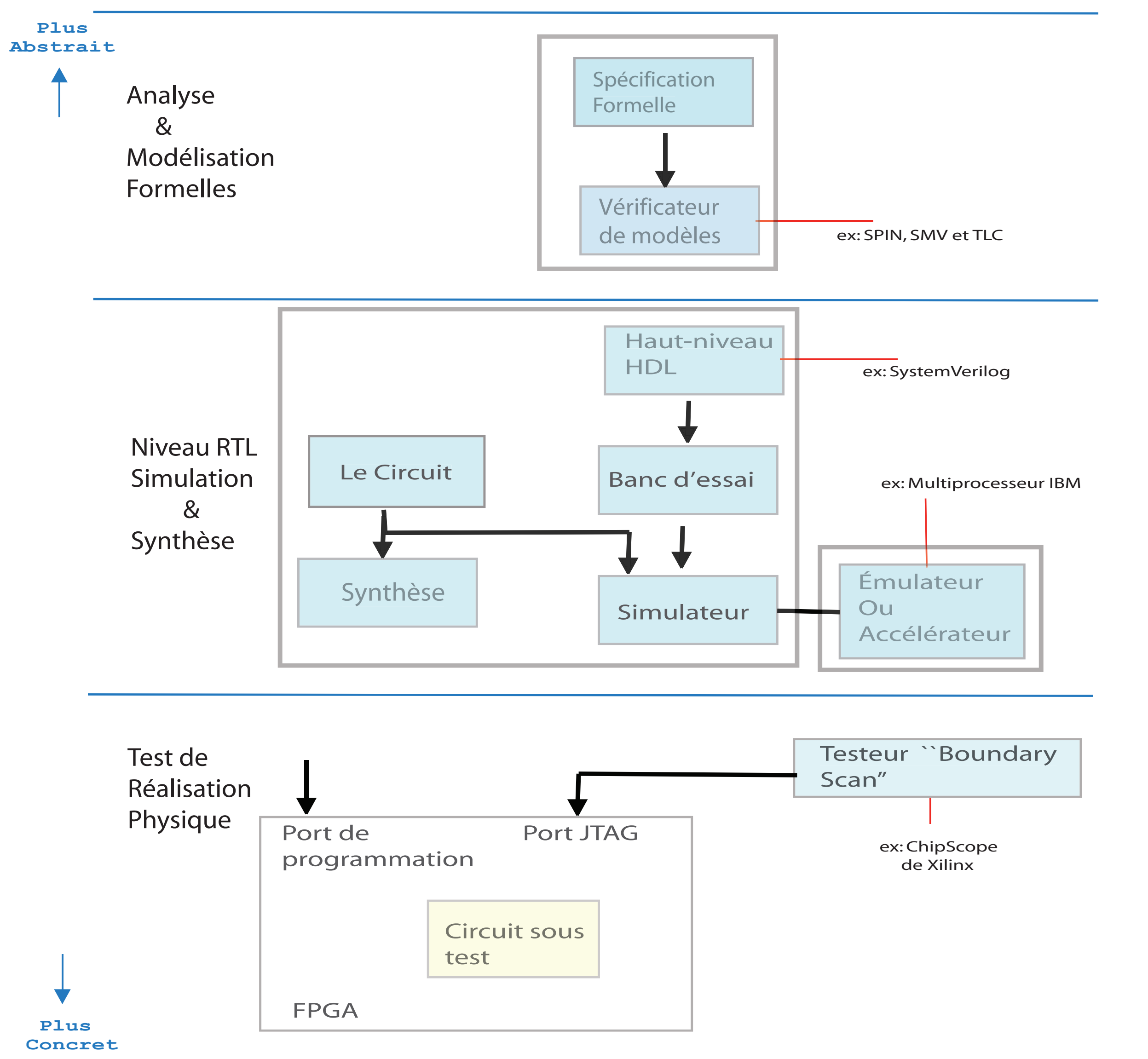


Figure 2.1 – Les différents niveaux de vérification de circuits

2.1.3.2 La simulation augmentée d'accélération et d'émulation

Le **deuxième** grand procédé qui a été très utilisé dans l'industrie est la simulation. Elle consiste à vérifier le circuit au niveau logique. Le circuit est modélisé sous forme d'un programme informatique avec des caractéristiques spécifiques au circuits. Dans ce contexte aussi, plusieurs outils ont été développés [115] : des langages HDL (Hardware description Language) comme VHDL et Verilog, HVLs (Hardware Verification Languages) comme e et Open Vera, des outils de simulations [38, 42] avec des générations dirigées ou pseudo-aléatoires de tests, des ATPG [4, 71] etc.

La première méthodologie de vérification automatique chez IBM [38] a été développée pendant les années 80, elle est basée sur la simulation et ces principaux objectifs étaient :

- Un niveau de spécification (RTL)
- Une Séparation de l'aspect fonctionnel du temporel
- Une simulation orientée cycle d'horloge
- Une analyse de couverture basée sur les propriétés du langage de spécification
- Une preuve formelle d'équivalence entre la spécification RTL et la réalisation au niveau portes logiques.

La méthodologie de vérification fonctionnelle décrite dans [42] parle d'entretenir un plan de vérification en utilisant l'enregistrement des tests, un processus de vérification progressif et le suivi de l'activité de débogage par rapport aux deux propriétés temps et espace. Avec un générateur de programmes tests, on introduit deux concepts TK (Testing Knowledge) et DF (Directives Files) pour remédier à la faille de la vérification qui se contentait des tests commerciaux et les tests par applications majeures. On se sert de métriques de couverture (temps et espace) pour s'assurer que le processus de simulation ne soit pas concentré sur une partie du design et ne pas considérer d'autres parties.

Les accélérateurs et émulateurs matériels [12, 81] sont utilisés dans cette dernière décade pour accélérer la vérification par simulation. L'accélérateur utilise des cartes ASICs [16, 107] ou FPGAs [45, 115] qui contiennent un certain nombre de processeurs logiques et des mémoires locales, la représentation HDL du design est compilée dans le code machine, qui est plus tard distribué parmi les divers processeurs. L'émulateur [66, 79, 96] utilise des

cartes FPGAs sur lesquelles la représentation HDL du circuit est en partie ou en totalité [78], synthétisée en un prototype. Un problème majeur [78, 95] avec ces systèmes de vérification, que nous tentons entre autres de résoudre, est le goulot d'étranglement causé par la communication entre les deux parties logicielle et matérielle.

La simulation via émulation constitue le seul outil, à notre connaissance, qui offre, à la fois, un haut niveau de modélisation, et une modélisation sur plate-forme physique. Comme il est susmentionné, son premier objectif est d'accélérer le processus de simulation ; elle demeure, comme est le cas de la vérification formelle, juste en mesure de prouver l'existence d'erreur mais elle ne peut pas garantir leur inexistence. Les émulateurs sont des machines spécialisées (dont le prix est très élevé) qui se servent de prototypages sur FPGAs pour accélérer le processus de simulation.

2.1.3.3 La vérification formelle

Le **troisième** grand procédé qui est beaucoup plus attrayant dans l'industrie ces dernières années, est la vérification formelle [17, 49, 65, 86, 90, 108]. Il est plus consistant et son outil de preuve est basé sur les mathématiques. La vérification fonctionnelle [115] ou logique peut avoir la forme d'une vérification formelle. Il existe deux grandes catégories de vérification formelle :

1. Vérification de modèle et la logique temporelle [9, 10, 113, 119]. Les outils traduisent la spécification initiale dans une représentation intermédiaire ; les vérificateurs de modèles modélisent le système sous forme d'automates à états finis et vérifient, de façon automatique, la véracité des propriétés pour chacun des états accessibles. Elle est souvent confrontée au problème d'explosion d'états. Des techniques, telles que SAT (Boolean Satisfiability Testing) et BDD (Binary Decision Diagrams) et autres [70], ont été développées pour alléger ce problème [44].

2. Les prouveurs de théorèmes [43, 51, 63] tel PVS (Prototype Verification System) [57, 100] et HOL (Higher Order Logic) [24, 47] sont des vérificateurs semi-automatiques et représentent le système sous forme de formules mathématiques et procèdent par des inférences guidées par l'humain en se servant d'un ensembles de théorèmes et d'axiomes. Dans [57], on s'est servi d'une étape supplémentaire pour confirmer la validité de la vérification : La spécification PVS est traduite en Verilog et est implé-

mentée et testée, de façon aléatoire, sur FPGA.

Plusieurs travaux ont considéré la combinaison de plusieurs techniques de vérification [19] comme les deux outils vérificateur de modèles et prouveurs de théorèmes tel Forte [64, 111], ou prouveur de théorèmes et simulateur [111] ou autres combinaisons [24, 62, 109, 118].

Souvent les outils sont complémentaires d'où l'idée de développement d'outils unificateurs permettant extensibilité et flexibilité [116]. D'autres dans cette même direction que nous soutenons, tel que le langage TLA+ [73, 75], étant basé sur un des formalismes des plus universel, a de grandes prédispositions du fait qu'il est basé sur les mathématiques ppréalablement bien connues de tout programmeur.

On appelle vérification semi-formelle la combinaison de la simulation et la vérification formelle, on parle aussi dans ce cas de vérification par assertions.

2.1.3.4 Le niveau d'abstraction et le degré d'automatisation

Un haut niveau d'abstraction permet d'une part la maîtrise du système, de procéder par élimination des erreurs et d'accélérer la vérification. C'est une des stratégies qui a pour but d'automatiser certaines tâches qualifiées de banales, comme codifier les OP-codes d'instructions quand on écrit un "test case" pour vérifier le fonctionnement d'un microprocesseur et donc développer un composant logiciel qui s'occupe de traduire les mnémoniques vers les codes correspondants. Rehausser le niveau d'abstraction a constitué donc une des orientations majeures. Dans ce contexte nous avons les TLMs (Transaction level modeling) [22], SystemC [21, 37] , SystemVerilog [3, 18].

2.1.3.5 Notre contribution

Les outils existant utilisant des vérificateurs de modèle pour la vérification de circuits n'ont jamais dépassé le niveau RTL, en allant bien entendu vers la forme physique du circuit ; VIS [20] considère des spécifications RTL synthétisables, des outils font la traduction [28] de verilog ou VHDL vers SMV, d'autres se contentent de vérifier à un niveau supérieur au niveau RTL [28, 56]. Notre méthode tente d'étendre la vérification au niveau le plus bas et de compléter chacune des méthodes décrites ci-dessus en offrant les 4 éléments essentiels suivants :

1. Automatisation,
2. Construction d’une preuve rigoureuse basée sur des mathématiques,
3. Modélisation à un haut niveau d’abstraction,
4. Vérification de circuits physiques.

Soulignons que, dans notre cas, le vérificateur se sert des résultats de calcul du circuit synthétisé et non d’une quelconque de ses implémentations à un niveau supérieur.

2.2 La vérification de modèles

D’après la définition de [88], la vérification automatique de modèles (Model Checking) est une technique automatique de vérification de systèmes réactifs ; ces derniers étant en interaction continuelle avec leur environnement (les circuit séquentiels, les protocoles de communication). Elle spécifie le système par :

1. Un certain nombre de propriétés souvent exprimées en formules de la logique temporelle propositionnelle.
2. Son comportement est souvent modélisé sous forme de graphe de transitions dont les noeuds sont les états et les arcs sont des transitions.

Le vérificateur de modèle a pour rôle soit d’affirmer que le système satisfait les propriétés sans aucune erreur ou dans le cas contraire de délivrer un contre exemple sous forme d’une trace d’exécution du modèle pour laquelle une ou plusieurs propriétés ne sont pas satisfaites.

À l’origine, c’est une technique développée en 1981 par Clarke et Allen Emerson et au même moment et de façon indépendante par Joseph Sifakis, d’autres variantes sont développées par la suite.

Les premiers vérificateurs étaient confrontés au problème connu sous le nom explosion d’états [84] quand il s’agit de vérifier des systèmes assez larges. Vers la fin des années 80, est apparue une nouvelle façon plus concise de représenter les transitions dite OBDD (Ordered Binary Decision Diagrams).

La vérification originale jumelée à la technique des OBDD est appelée symbolic model checking. Cette dernière technique s'est avérée insuffisante dans bien des cas et on a pensé encore à développer d'autres. Les auteurs ont écrit plusieurs articles traitant de deux autres techniques : la composition [58] qui exploite la structure souvent hiérarchique des systèmes qui les rend vérifiables de facon progressive et l'abstraction qui consiste, pour n'exprimer que l'idée, à réduire les domaines de valeurs des variables d'états.

Dans le but de nous faire une idée assez claire sur les vérificateurs automatiques de modèles nous nous sommes intéressés à l'étude et à la comparaison de trois vérificateurs automatiques de modèles (models checkers) :

- TLC est un vérificateur explicite ; il est parmi les plus récents et son utilisation remonte au début des années 2000. Son langage de spécification est TLA+ [75].
- SPIN (Simple Promela INterpreter) est un des premiers vérificateurs explicites, il est très populaire et son utilisation remonte au début des années 80. Son langage de spécification est Promela (PROcess MEta-LAnguage) [53, 54].
- SMV (Symbolic Model Checker) [84, 85] est un des premiers vérificateurs symboliques, il est aussi populaire et son utilisation remonte au début des années 90.

Nous avons considérés 3 critères de comparaison : le temps de vérification, l'espace de vérification (en terme de nombres d'états) et l'expressivité de leur langage de modélisation.

Notre comparaison est faite par rapport à deux études de cas : l'algorithme d'exclusion mutuelle de Peterson et un contrôleur d'ascenseur. Ce qui en est ressorti est qu'on a eu rapidement (N=5) le problème d'explosion d'états avec l'outil SPIN. Le processus de vérification est très lent avec TLC, par contre, il est rapide avec SMV.

Un des points importants que nous avons eu à considérer dans cette comparaison est l'expressivité des trois outils de vérification automatique. Nous avons pu voir à travers les descriptions et les exemples considérés, que les outils SPIN et SMV utilise chacun un langage de modélisation très proche de point de vue syntaxique mais très loin du point de vue sémantique, des langages de programmation procéduraux.

Le concepteur de systèmes est donc amené à apprendre un autre langage de programmation avec une toute nouvelle sémantique, pour pouvoir se servir de l'outil de vérification. À travers notre tentative de modélisation du contrôleur d'ascenseur nous avons réalisé

combien il est difficile de décrire un comportement dans la sémantique des vérificateurs automatiques SPIN et SMV.

Avec l'outil TLC, comme on peut aussi le voir, une spécification TLA+ est simplement un ensemble d'expressions mathématiques dans un formalisme très familier et universel (connu de toute personne ayant fait les études supérieures en genie). Avec la nature fonctionnel du langage TLA+ nous saisissons le fait qu'une spécification TLA+ nécessite beaucoup moins d'encodage, elle est plus compact et concise.

2.3 La vérification du circuit synthétisé

Les outils existants qui utilisent la vérification de modèles pour des circuits matériels ne vont pas au delà de la forme synthétisable du circuit. Parmi ceux qui considère ce plus bas niveau d'abstraction, nous pouvons citer VIS [1] et FormalCheck [14] ;

Dans notre cas, nous avons relié notre vérificateur directement au circuit réel, le vérificateur utilise directement et dès le départ tous les résultats du calcul du circuit synthétisé pour la détection d'éventuels dysfonctionnements. Nous faisons ainsi l'extension de la vérification de modèle abstrait, à celle de la version physique du circuit [34]. Nous offrons la possibilité d'expansion partielle ou totale du graphe d'états du circuit physique.

Notre méthodologie présente une nouvelle approche qui se base sur une contrôlabilité et une visibilité complètes sans se limiter seulement aux dysfonctionnements fonctionnels (logiques). Notons que la nature de notre approche permet de faire d'une pierre plusieurs coups ; même des erreurs non fonctionnelles seront détectées. En effet du fait que nous considérons la totalité des états accessibles, au cas où un de ces états est atteint suite à une erreur de timing ou des connexions dans le circuit, le vérificateur va détecter qu'aucun état ne lui correspond dans l'implémentation abstraite et il va générer la trace en donnant le chemin qui a mené à cet état. Nous considérons l'état du circuit (IUT) explicitement pour le test de conformité ; pour chaque transition possible il y a calcul de l'éventuel successeur. Le but ultime de notre méthodologie est de surmonter des limites fondamentales de la simulation en offrant, d'un côté, une expansion exhaustive de l'espace des états en considérant toutes les entrées possible pour chacun des états, et, d'un autre côté, une assurance que chaque action activée pour chacun des états accessibles de l'implémentation physique est aussi activèe dans l'état correspondant dans l'implémentation de référence et que les états successeurs respectifs dans les 2 implémentations correspondent.

2.4 Les architectures FPGA et Les plates-formes cibles

Comme le nom l'indique, ce sont des architectures reconfigurables par programmation. Une architecture FPGA consiste en une matrice de portes logiques (Gate Array) qu'on peut lier et relier à notre guise. Nous pouvons personnaliser notre circuit et le reconfigurer autant de fois que l'on souhaite. Des outils accompagnants l'achat d'une carte FPGA permettent d'écrire des programmes VHDL ou VERILOG qu'on peut simuler et par la suite synthétiser sous formes de circuits physiques.

Le processus est répété jusqu'à obtention d'un résultat satisfaisant. Il permet de se définir un ensemble de variables d'états dont nous pouvons visualiser l'évolution au cycle d'horloge. C'est un procédé assez précis mais très lent. Pour l'ajout d'une variable d'état par exemple, il faut aller au programme source la rajouter explicitement, recompiler puis simuler à nouveau et par la suite synthétiser. Dans le cas d'une application un peu complexe, on peut devoir répéter ces étapes des milliers de fois. Il est clair qu'il s'agit là d'une méthode qui est très utile de point de vue pédagogique mais très loin d'être pratique pour le développement d'applications complexes.

Nous avons developpé plusieurs exemples de plus simple à plus complexe (les exemples des tutoriels Altera et Xilinx, des exercices et un travail pratique (le jeu démineur) [30] réalisés dans le cadre du cours [77], le processeur MIPS, et deux versions d'un contrôleur d'ascenseur. Le jeu démineur a fait l'objet d'une extension dans le but de le vérifier avec TLC [32] ; il a constitué notre point de départ pour ce présent projet de doctorat. Nous avons expérimenté le long et très coûteux processus de simulation et de synthèse.

Il existe deux grands constructeur de cartes FPGA : Xilinx [117] et Altera [8]. Nous avons considéré les plates-formes suivantes :

- La carte Altera University Program 3 (UP3), elle est équipée d'un Cyclone EP1C6 d'Altéra à 6K éléments logiques et munie d'une RAM interne de 20 blocs de 4Kx1 bit, elle intègre le processeur NIOS et un SRAM 64Kx16. La vérification sur cette carte se fait via une ligne série RS232-C .

- La carte Morph-IC est équipée d'un FPGA ACEX 10K d'Altéra. La vérification sur cette carte se fait via un lien USB1.1.

- La carte Easy FPGA est équipée d'un Cyclone EP1C12 d'Altéra. La vérification sur

cette carte se fait via un lien USB1.1.

- La carte Spartan 3 Experimenter's Board (SEB3) de Xilinx, équipée d'un FPGA XC3S400 à 400K portes logiques, d'un SRAM 128Kx8. La vérification sur cette carte se fait par un lien USB1.1 via un adaptateur AC/DC.

- La carte XEM3001 d'Opal Kelly(OK), elle est équipée d'un FPGA XC3S400 à 400K portes logiques de Xilinx et d'un SRAM 256Kx16-bit SRAM. La vérification sur cette carte se fait via un lien USB2.0.

- La carte NEXYS2 de Digilent, elle est équipée d'un Spartan-3E à 1200K portes logiques et de 16Mbytes de SDRAM et 16Mbytes de Flash ROM. La vérification sur cette carte se fait via un lien USB2.0.

- La carte ATLYS de Digilent, elle est équipée d'un Spartan 6 LX45 et de 128Mbytes x 16 bits DDR2 et d'une horloge de 500MHz. La vérification sur cette carte se fait via un lien USB2.0.

Ces cartes répondent à nos besoins en ressources, selon nos études de cas, tout en rentrant dans nos capacités financières.

2.5 Conclusion

De toutes les références que nous avons pu consulter, nous concluons que notre méthodologie constitue une première tentative de liaison directe entre la vérification de modèle et la vérification d'un circuit réel. Nous avons réalisé l'éxtension de la vérification par l'outil TLC aux prototypes sur architectures FPGA. Le vérificateur teste traditionnellement des modèles abstraits internes exprimés dans son propre langage ”native”, dans notre application les tests sont dirigés vers le système externe sous test. Cette approche utilise le vérificateur de modèle pour générer automatiquement des bancs d'essai, les états du système à vérifier ainsi que ses entrées. Il les applique au modèle ainsi qu'à la réalisation physique, puis évalue la conformité entre leurs résultats respectifs.

CHAPITRE 3

LES CONCEPTS MATHÉMATIQUES

TLA+ est un langage de spécification qui étend la logique temporelle TLA (Temporal Logic of Actions) développée par Leslie Lamport [74]. TLA est une logique temporelle linéaire dite LTL (voir la section 3.1), ses axiomes et règles de déduction sont décrits à differents endroits, comme [74] et [75]. Le site web sur TLA+ [73] contient des liens vers de nombreux articles sur TLA et TLA+.

Nous allons dans ce qui suit couvrir de façon progressive l'ensemble des notions utilisées en TLA+ pour une spécification (description écrite) de systèmes concurrents. Comme on peut le voir, son outil de base sont des mathématiques connues de tous, il ajoute à TLA les concepts suivants :

- La logique de premier ordre, incluant : la logique propositionnelle, la quantification de premier ordre, et l'égalité ; voir [87].
- La théorie des ensembles, basée sur la théorie ZFC (Zermelo-Fraenkel-Choice) ; voir par exemple [50]. Notons que la définition d'une fonction en TLA+ n'est pas basée sur la notion d'ordre telle que celle de ZFC ou d'autres formulations comme NBG, or Morse-Kelly.
- La théorie des nombres, basée sur les axiomes Peano : les nombres naturels et les entiers.
- Une puissante capacité de définition basée sur le lambda calcul [61], incluant une facilité d'ordre supérieur pour le passage d'opérateurs comme arguments.

En plus de son choix de formalisation pour les mathématiques traditionnelles [75], Leslie Lamport s'est inspiré des langages de programmation pour offrir l'option de modularisation lors de l'écriture d'une spécification, voir le chapitre 17 dans [75]. Celle-ci étant la description comportementale (logique ou fonctionnelle) d'un système.

Pour décrire un système, les scientifiques se servaient d'un système d'équations pour déterminer comment son état évolue dans le temps. Formellement, on définit un comportement comme étant une séquence d'états. Voici quelques notions utilisées lors d'une spécification d'un comportement :

- Variable : c'est tout simplement un identificateur declaré par le mot clé VARIABLE. Elle est non typée.

- État : On parle de l'état d'un système, c'est une association entre un ensemble de variables et un ensemble de valeurs.

- Prédicat : Soient s un état, f une fonction, f(s) est dite une fonction d'état. Si la valeur de f(s) est booléenne, f sera dite prédicat d'état.

- Action : C'est une relation entre un état et son successeur. Elle matérialise la transition du système d'un état s vers un état successeur s'. Son activation se spécifie par une relation binaire $A(s, s')$. Une Action A est activée (*enabled*) dans l'état s ssi[1], il existe un état s' pour lequel $A(s, s')$ est vrai (*true*).

3.1 La logique temporelle

Étant donné un système à transitions, la logique temporelle constitue un langage d'expression formelle de propriétés telles : Y auraient-ils des états indésirables dans le système ? Comme des états puits, d'où l'on sort plus, ce qui correspond à une situation d'interblocage. Il peut s'agir aussi d'états de violation d'exclusion mutuelle pour des modèles de systèmes à ressources partagées. Y aurait-il un comportement du système ou un certain état désirable n'est jamais atteint ou une certaine action jamais exécutée ? Il pourrait s'agir d'une situation de famine "livelocks", comme un processus n'accédant jamais à la section critique pendant que d'autres processus du système y accèdent. Y auraient-ils certains états initiaux atteignables de n'importe quel autre état ? Autrement dit, le système est il réinitialisable ? Si nous reprenons la définition de [59] la logique propositionnelle est un outil nous permettant de raisonner sur les circuits combinatoires et la logique temporelle constitue un outil nous permettant de raisonner sur les circuits séquentiels. Ces derniers rajoutent une autre dimension par rapport aux circuits combinatoires qu'est le **temps** du fait que les sorties d'un circuit séquentiel ne dépendent pas seulement des entrées mais aussi de l'état précèdent du système qu'il décrit. En passant du domaine combinatoire à celui du séquentiel nous introduisons deux nouveaux paramètres très connus dans le monde de l'informatique : le **temps** et la **mémoire**.

[1]"ssi" est une abréviation de l'équivalence logique : si et seulement si

On parle souvent de deux types de logiques temporelles étudiées en détails dans [59] :

- La logique LTL (Linear-time Temporal Logic) permet d'exprimer une propriété sur une séquence d'unités de temps, contrairement à la logique CTL elle considère un seul temps futur.

- Considérée aussi appartenir au type "branching-time temporal logics", la logique CTL (Computation Tree Logic) se base sur la vérification de propriétés à travers l'exploration de chemins d'un graphe d'états. Elle combine les opérateurs de la LTL avec les opérateurs existentiel (G : tous les temps et F : certains temps) et universel (E : il existe) de la logique mathématique.
  Exemple : soit p une propriété, L'opérateur EG : il existe un chemin pour lequel p est vraie pour tous les temps futurs. L'opérateur EF : il existe un chemin pour lequel p est vraie pour certains temps futur. L'opérateur AG : pour tous les chemins, p est vraie pour tous les temps futurs. L'opérateur AF : pour tous les chemins p est vraie pour certains temps futurs.

Il existe une classification de propriétés, nous avons : Les propriétés de survie (Liveness) qui expriment que la propriété soit vraie à certains temps dans le futur (selon [59] quelque chose de bon va possiblement se produire) et les propriétés de sûreté (Safety) qui expriment que la propriété soit vraie tout le temps (toujours selon [59] : quelque chose de mauvais ne va jamais se produire).

3.2 Le verificateur TLC

Le vérificateur TLC de Leslie Lamport constitue le noyau de la trousse d'outils TLA+ qu'on peut se procurer sur [73]. C'est un programme Java prenant en entrée des spécification TLA+.

TLA+ est un langage fonctionnel d'ordre supérieur du fait qu'il offre le concept de fonction d'ordre supérieur basée sur les expressions lambda qui constitue un de ses facteurs de puissance. Il a l'habileté de pouvoir passer une fonction en paramètre à une autre fonction et de pouvoir retourner une fonction. Il permet d'exprimer les concepts mathématiques standards : la logique propositionnelle, la logique du premier ordre, la théorie des ensembles et la théorie des nombres. Il inclut en partie la logique temporelle linéaire (LTL). Sa syntaxe est très proche des notations mathématiques standards.

3.2.1 Spécification d'un comportement

En général, un système peut choisir parmi plusieurs actions simples $A_1, A_2, \ldots, A_N$ dont l'ensemble se décrit par une action composée souvent nommée $Next$, définie par la disjonction des A_i (voir aussi l'exemple à la fin de la section 7.6.4) :

$$Next(s, s') \triangleq \bigvee_{i=1}^{N} A_i(s, s')$$

Si nous définissons les successeurs de l'état s via l'action A par $Succ_A(s) \triangleq \{s'|A(s, s')\}$ et nous dénotons l'activation de l'action A par $Enb_A(s)$ qui est un prédicat d'état, nous avons

$$Enb_A(s) \Leftrightarrow Succ_A(s) \neq \emptyset$$

et

$$A(s, s') \Leftrightarrow s' \in Succ_A(s)$$

L'ensemble des états initiaux définit les conditions initiales du système, il est specifié par un prédicat d'état souvent appelé $Init(s)$.

Un comportement est défini comme étant une séquence infinie d'états. TLA+ spécifie les comportements par des formules temporelles. Une forme commune de formule temporelle établit qu'une certaine propriété définie par un prédicat d'état P est toujours vraie pour un comportement, ce qui implique que P est vrai pour chacun des états de la séquence. La syntaxe correspondante est $\Box P$.

Soit $vars$ dénotant le tuple $< v_1, v_2, \ldots >$ des variables dont chaque association à un tuple de valeurs nous définit un des états du système. Étant données ces variables $vars$, un prédicat d'états initiaux et une action composée $Next$ pour un système donné, ses comportements sont définis comme satisfaisant une formule temporelle souvent dénotée $Spec$:

$$Spec \triangleq Init \wedge \Box[Next]_{vars}$$

De façon informelle, cela signifie qu'un comportement satisfait $Spec$ ssi $Init$ est vrai pour le premier état de la séquence et toute paire d'états consécutifs dans la séquence satisfait la relation $Next$ ou les deux états sont égaux.

3.2.2 Spécification de propriétés

Les propriétés d'un système sont aussi définies par des formules temporelles. Un système décrit par la formule $Spec$ satisfait la propriété exprimée par la formule $Property$ ssi $Spec \Rightarrow Property$. Ce qui signifie que tout comportement satisfaisant $Spec$ satisfait aussi $Property$.

Nous pouvons avoir plusieurs sortes de propriétés, la plus importante est l'invariant. Il est défini comme étant un prédicat d'état toujours vrai pour un système en question. Soit $Inv(s)$ un prédicat d'état, il est invariant d'un système spécifié par $Spec$ ssi $Spec \Rightarrow \Box Inv$. Une implication directe montre que Inv est un invariant du système spécifié par $Init \wedge \Box[Next]_{vars}$ ssi (1) $Init(s) \Rightarrow Inv(s)$ et (2) $Inv(s) \wedge Next(s, s') \Rightarrow Inv(s')$. Un invariant particulier constituant un élément crucial pour tout système est l'invariant *Type* qui caractérise le domaine des valeurs possibles des variables d'état. Comme exemple, si nous avons le prédicat $s \in Naturals$ est un invariant, il signifie que la valeur associée à la variable s dans chacun des états du système est toujours un Naturel et jamais autre chose d'autre tel qu'un nombre réel ou une chaîne de caractères.

3.2.3 Le concept de raffinement

Comme nous l'avons déjà mentionné dans la section 1.2, nous utilisons le raffinement pour prouver qu'une implémentation reproduit fidèlement sa spécification. La spécification constitue une formalisation abstraite de l'ensemble des contraintes qu'un système doit satisfaire. Il y a généralement différentes façons de vérifier la conformité d'une implémentation par rapport à sa spécification. Nous procédons par comparaison entre les deux en utilisant une forme simplifiée du concept de *raffinement* de TLA.

Le raffinement constitue un moyen pour comparer des comportements de deux systèmes différents étant donnée une relation ψ entre les deux espaces d'états correspondants. Supposons deux systèmes spécifiés en termes de modules TLA+ S et R, et que chacun définisse une action appelée A. TLA+ permet de distinguer les termes définis dans les différents modules en faisant précéder les noms des termes par le nom du module suivi d'un point d'exclamation. De cette manière les deux versions de l'action A peuvent être référées par $S!A$ et $R!A$ respectivement ; la première dénote la relation binaire $S!A(s, s')$ dans l'espace des états du système S, alors que la deuxième dénote la relation binaire $R!A(r, r')$

dans l'espace des états du système R. De façon similaire, si les deux modules définissent chacun un prédicat d'états initiaux $Init$, Les deux versions peuvent être référées par $S!Init$ et $R!Init$.

Posons ψ l'association de l'espace des états du système R à l'espace des états du système S. ψ est alors dite un raffinement (*refinement mapping*) pour l'action A ssi :

$$R!A(r,r') \Rightarrow S!A(\psi(r),\psi(r'))$$

Plus souvent, on dit que le système R *raffine* le, ou est un raffinement du, système S par la relation ψ ssi : (1) ψ est un raffinement pour chaque action de R et

$$(2)R!Init(r) \Rightarrow S!Init(\psi(r))$$

Algébriquement, une relation de raffinement est un type d'homomorphisme d'un système à l'autre. $\psi(r)$ est appelée le S-état s *correspondant* au R-état r. En d'autres mots si R raffine S, alors à chaque fois qu'une action est activée entre les états r et r' dans le système R, une action similaire est activée entre les états correspondants dans le système S, et l'état correspondant à tout état initial R-état est un état initial S-état.

Par une induction directe, nous pouvons voir que si R raffine S et r est un R-état atteignable à partir d'un état initial spécifié par $R!Init$, alors $\psi(r)$ est un S-état atteignable à partir d'un état initial spécifié par $S!Init$. De façon similaire, l'image par ψ de l'ensemble des états R-atteignables est un sous ensemble des états S-atteignables. Une implication importante est que tous les S-états correspondants aux R-états atteignables satisfont automatiquement tout invariant du système S.

Par l'ensemble des implications mentionnées précédemment, on assure que tout changement d'état dans le système R a son correspondant, par la fonction mathématique ψ, dans le système S. Comme exemple, voici la définition de cette fonction pour le raffinement du micro-contrôleur Picoblaze donnée dans la section 7.6.4 :

```
psi(r) == [
  Reg      |-> r.Core.DM.Reg ,
  RAM      |-> r.Core.DM.SCRAM ,
  InPort   |-> r.IO.InPort ,
  OutPort  |-> r.IO.OutPort ,
  PC       |-> r.PCtrl.PC ,
  SP       |-> r.PCtrl.Stk.SP ,
```

```
Stack   |-> r.PCtrl.Stk.Space ,
C       |-> r.Core.FL.C ,
Z       |-> r.Core.FL.Z ,
IE      |-> r.Core.FL.IE ]
```

Cet exemple présente l'implémentation en TLA+ de la fonction ψ qui associe l'état s à l'état r (voir la Figure 4.2).

3.2.4 Une Simplification : Comportement Déterministe

De manière générale, plusieurs états successeurs s' peuvent satisfaire l' Action $A(s,s')$ pour un état donné s ; l'ensemble $Succ_A(s)$ peut être arbitrairement large. Si $Succ_A(s)$ contient au plus un élément pour tous les états s, A est dite fonctionnelle ou *déterministe*. Dans ce cas, nous avons $Enb_A(s) \Rightarrow Succ_A(s) = \{Opn_A(s)\}$, où $Opn_A(s)$ est *la fonction successeur* de A. Nous avons aussi

$$A(s,s') \Leftrightarrow Enb_A(s) \wedge s' = Opn_A(s)$$

Un système est déterministe si toutes ses actions le sont.

Si les Actions $S!A, R!A$ dans deux systèmes S, R sont toutes les deux déterministes, nous pouvons alors simplifier le critère de raffinement. On peut montrer que la condition $R!A(r,r') \Rightarrow S!A(\psi(r),\psi(r'))$ est équivalente à :

$$R!Enb_A(r) \Rightarrow S!Enb_A(\psi(r)) \wedge \psi(R!Opn_A(r)) = S!Opn_A(\psi(r))$$

3.3 Conclusion

TLC est un logiciel qui évalue des formules exprimées dans un sous-ensemble de TLA+ étant données certaines conditions telles qu'un espace d'états fini. Bien que TLA+ soit conçu pour le raisonnement sur des systèmes concurrents et TLC soit décrit comme vérificateur de modèles, TLA+ est beaucoup plus général du moment qu'il permet l'expression de la majorité des mathématiques standards et discrètes d'une part. D'autre part, étant un langage fonctionnel, il est très approprié à la description d'un circuit en termes d'un ensemble de fonctions au sens mathématiques.

CHAPITRE 4

TEST D'IMPLÉMENTATIONS PHYSIQUES

Notre objectif est l'extension de la vérification automatique au *testing d'implémentations physiques* par comparaison automatique à une implementation de référence valide. Le modèle abstrait vérifié, à la façon traditionnelle, est connecté physiquement à l'implémentation externe. Il sert de référence à laquelle cette dernière va être comparée. Dans le cadre de cette thèse nous considérons exclusivement des réalisations sous formes de prototypes sur des plates-formes FPGAs.

Ainsi, avec cette approche, le vérificateur de modèles constitue aussi une station de testing automatique de circuits réels au lieu de servir uniquement de vérificateur de leur modèle abstrait. En gros, nous avons bâti le lien de communication approprié en juxtaposant une chaîne d'éléments (Figure 4.1). Comme expliqué dans [35], le lien établit une connexion physique et un raffinement (*refinement mapping*) entre la spécification abstraite et la réalisation physique. Tout ce qui peut se produire dans le domaine physique doit être traduit et prouvé conforme à ce qui se produit dans le domaine abstrait. Dans notre procédé final, pour le maximum de performances, nous réalisons une double expansion du graphe d'accessibilité, une du côté matériel (Implémentation réelle) et une autre du côté logiciel (implémentation abstraite). Le rôle du vérificateur est d'assurer la conformité entre les 2 expansions. En effet, toute transition et tout effet généré de cette transition dans l'expansion physique doit avoir la transition et l'effet correspondants dans l'expansion abstraite.

4.1 Survol des articles

Comme nous l'avons déjà dit dans le chapitre Introduction, l'objectif de ce chapitre est de présenter en détails notre méthodologie. La majeure partie de sa matière a été publiée et le reste est en voie de l'être. La deuxième partie de cette thèse étant consacrée à ces publications et pour éviter la duplication du contenu, nous nous référerons souvent à ces chapitres que nous compléterons là où c'est nécessaire.

Ce chapitre constitue donc un complément au contenu des articles. La première étape de notre travail a consisté en une démonstration de la faisabilité de notre approche. Rappelons que celle-ci étend le vérificateur de modèles TLC pour vérifier la conformité d'une im-

plémentation FPGA par rapport à la spécification mathématique exprimée en TLA+. Nous avons développé plusieurs exemples d'implémentation et de vérification : le jeu démineur, le microprocesseur MIPS, deux versions du contrôleur d'ascenseur et le microcontrôleur picoblaze. Ce point a fait l'objet de deux publications :

- [34] : "Applying Model-Checking to Post-Silicon-Verification : Bridging the Specification-Realisation Gap", cet article explique notre démarche sur la base d'un exemple simple du contrôleur d'ascenseur.
- [60] : "Overview of Applying Reachability Analysis to Verifying a Physical Microprocessor", cet article explique notre démarche sur la base d'un exemple plus complexe qui est le micro-contrôleur à 8 bits, PicoBlaze de Xilinx.

Lors de la démonstration de notre démarche, nous avons été confrontés au problème du goulot d'étranglement causé par les délais de communication entre la réalisation physique et la réalisation abstraite. Après des tentatives de réduction de ce délai par l'implantation d'algorithmes d'anticipation des invocations du vérificateur de modèles, nous avons implanté *un analyseur d'accessibilité matériel*. Celui-ci est un circuit spécialisé qui permet d'éliminer complètement ce delai de communication. Il procède à l'analyse d'accessibilité de l'implémentation physique alors que le vérificateur TLC analyse celle de l'implémentation de référence. Cette partie de notre travail a aussi fait l'objet de deux publications :

- [36] : "Model Checker to FPGA Prototype Commmunication Bottleneck Issue", il illustre, d'une part, le pourquoi du goulot d'étranglement par un modèle mathématique que nous avons élaboré sur la base de différentes mesures expérimentales. D'autres parts, il expose les différentes approches que nous avons mises en place pour résoudre ce problème.
- [33] : "An Embedded Reachability Analyzer And Invariant Checker (ERAIC)", ce papier illustre le fonctionnement et la structure de notre *analyseur d'accessibilité matériel*. Il démontre aussi que celui-ci permet d'atteindre une meilleure performance comparé aux méthodes par anticipation via des caches (voir 4.3.2).

Ces publications présentent aussi quelques résultats préliminaires. Nous avons prévu un autre papier [35] : "Connecting a Model-Checker to an FPGA Prototype : Bridging the Specification-Post-Silicon Gap". Notre objectif est de fusionner, en rajoutant plus de détails,

les contenus des 4 papiers précédents et de rajouter essentiellement la partie concernant la communication entre les deux domaines abstrait et physique. Il aborde en détails les concepts de raffinement et de sur-définiton d'opérateurs.

4.2 Établissement de la connexion

La première étape a consisté à établir une connexion entre le vérificateur et l'implémenation physique (prototype FPGA). Le but était de permettre au vérificateur de soumettre un état et une action au circuit et d'obtenir une réponse. Celle-ci est l'état successeur si l'action est activée ou l'indication que l'action ne peut être activée dans l'état soumis. Le vérificateur obtient de la même manière une réponse de l'implémentation de référence et décide de la conformité entre les deux. Dans un tel cas, le processus d'expansion, de questionnement et de test de conformité se poursuit, dans le cas contraire, un contre exemple, sous forme d'une trace d'exécution, est construit et le processus de vérification s'arrête. Cette étape a permis d'asseoir la base de notre méthodologie, de déterminer et de mettre en place tous les éléments nécessaires (voir la Figure 4.1, pour l'exemple du contrôleur d'ascenseur). Quelques uns des morceaux de cette chaîne ont fait l'objet de publications que nous avons reproduites pour constituer les chapitres de la deuxième partie de ce document. Ce chapitre va permettre d'introduire et de détailler, si nécessaire, chacun de ces morceaux.

4.2.1 Description de notre méthodologie

Pour la description de notre méthodologie, nous avons dédié le chapitre 6. Celui-ci illustre comment se fait la connexion d'une implémentation physique au vérificateur de modèles pour réaliser son analyse d'accessibilité. Pour mettre au point cette méthodologie nous avons eu à utiliser différents mécanismes qu'on peut énumérer comme suit :

4.2.1.1 Le raffinement

Le concept de raffinement constitue un autre mécanisme essentiel pour notre méthodologie. Nous l'avons déjà décrit dans 3.2.3. Dans la Figure 4.1, il est représenté par le module TLA+ *Refinement Criteria* et sont rôle est de définir une fonction mathématique ψ qui permet de traduire ou convertir les résultats obtenus de l'implémentation physique au format de représentation de l'implémentation de référence, dans le but de permettre au

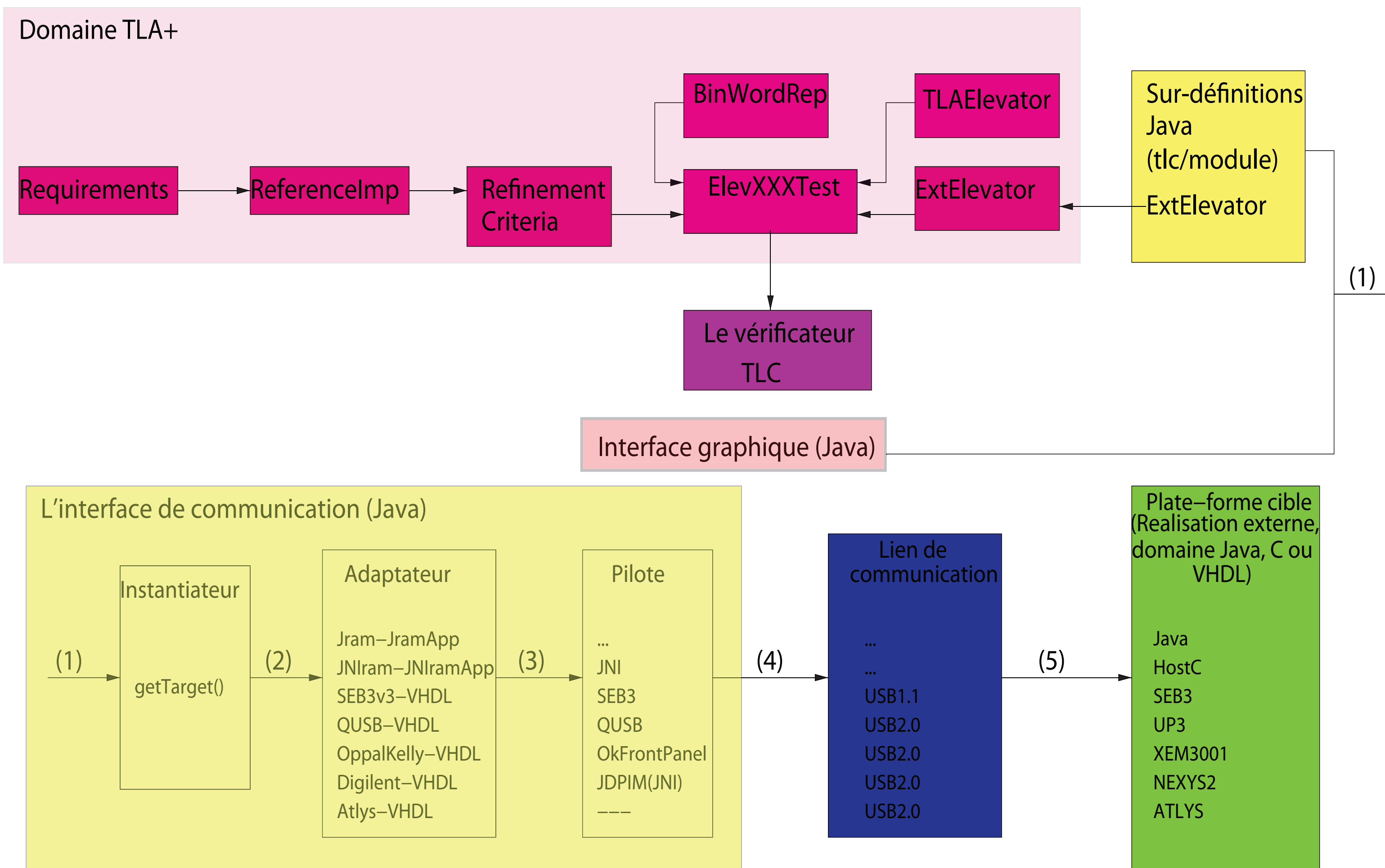


Figure 4.1 – La chaîne de vérification du contrôleur d'ascenseur

vérificateur de les comparer et vérifier leur conformité.

Dans ce qui suit, S-système est le domaine de la spécification “abstraite” alors que R-système est le domaine de la réalisation. Nous considérons la relation de raffinement ψ pour permettre une interprétation dans les termes du système S des états du système R. Comme l'illustre le diagramme de la Figure 4.2, cette relation est commutative.

Exemple : Considérons le circuit pour l'addition binaire dont l'état est spécifié par un tableau x de bascules $x[i], i = 1, \ldots, N$ où la valeur de chaque $x[i]$ est soit 0 ou 1 ; en termes TLA+, $x \in [1..N \rightarrow 0..1]$. Une relation de raffinement appropriée, associant le tableau binaire aux nombres naturels, est alors $\psi(x) \triangleq \sum_{i=1}^{N} 2^{i-1} x[i]$.

Définition : Un système dont le domaine est R est une implémentation correcte d'un autre dont le domaine est S ssi R raffine S par une relation de raffinement appropriée ψ.

En continuant l'exemple ci-dessus, supposons que le circuit dispose de l'opération inc, associant des tableaux binaires à des tableaux binaires, en incrémentant la valeur représentée par x. Une action TLA+ correspondante $Incr$ peut être définit comme $Incr(x, x') \triangleq x < 2^N - 1 \wedge x' = inc(x)$. Le circuit est correct ssi $\psi(inc(x)) = \psi(x) + 1$.

4.2.1.2 Sur-définition des conditions d'activation et des transitions

TLC intègre un mécanisme général qui permet de dédier un module TLA+ qu'on peut sur-définir (*override*) par un programme Java. Pour des raisons de performances, TLC utilise déjà ce mécanisme pour les types Integers, Sequences, etc. En principe, tout ce que ça requiert est de nommer adéquatement le programme Java ainsi que les méthodes que celui-ci surdéfinit et de le placer dans le répertoire *tlc/module/*. Dans nos études de cas, nous

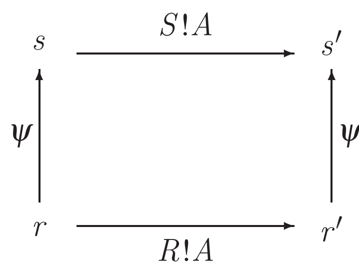


Figure 4.2 – Le diagramme d'homomorphisme entre les domaines R (Réalisation) et S (Spécification)

dédions un module TLA+ pour définir les conditions d'activation et les transitions. Lors de la connexion, nous remplaçons ce module par le programme Java qui communique directement avec l'implémentation physique cible. Celle-ci réalise concrètement les opérations correspondantes. Sur la Figure 4.1, le module TLA+ *ExtElevator* contient seulement les prototypes des conditions d'activation et des transitions qui sont surdéfinies en Java dans le fichier d'extension qui porte le même nom et qui est placé sous le répertoire */tlc/module*. Ce répertoire contient aussi un ensemble de classes prédéfinies qui aident à la traduction des structures voire des données entre les deux mondes TLA+ et Java. Par extension de ce mécanisme, nous pouvons dire que le modèle TLA+ est surdéfini (*over-ridden*) par l'implémentation physique .

4.2.1.3 Connexion du monde TLA+ au monde VHDL

Le but est de permettre au vérificateur TLC de considérer les résultats de calcul de la réalisation physique plutôt que ceux d'une réalisation abstraite. Un lien physique est donc nécessaire. Nous avons eu a expérimenter différents types de lien dépendamment de la plate-forme cible utilisée. Le choix de cette dernière nous a été dicté par des contraintes budgétaire d'une part et par le besoin du moment d'une autre part. Nous avons donc expérimenté entre autres les liens série RS232, USB1.1 et USB2.0.

Après avoir assuré la possibilité d'accès à la palte-forme physique, est venue la question concernant le type et le format de l'information à échanger. TLC nous permet de faire l'analyse d'accessibilité de l'implémentation abstraite de référence. Il procède de façon récursive sur l'ensemble des états, à chaque itération, il lui soumet un état déjà atteint et une action possible sur cet état. La réponse de l'implémentation abstraite est soit l'état successeur si l'action a été activée ou un indicateur de non acitivabilité dans le cas contraire. Notre besoin est donc de permettre à TLC d'interagir de façon similaire avec la réalisation physique.

4.2.2 Des points d'accès logiques

Il constitue le premier maillon et il est consacré à la connexion du monde Java à la carte FPGA. Chaque carte FPGA est accompagnée de sa propre interface permettant l'accès au broches ("pins") du FPGA. Ces dernières sont disposées à la manière des cartes mémoires ; un ensemble de broches servent d'adresses à d'autres qui, elles, sont utilisées pour véhiculer

les données. Un autre ensemble de broches est dédié pour le contrôle du trafic telle que la synchronisation entre émetteur et récepteur.

L'utilisation de certaines cartes nous est simplifiée par le fait que la partie logicielle de l'interface est déjà en Java. Pour d'autres qui sont implémentées en C, nous avions à rajouter une couche supérieure en Java en utilisant les outils JNI (Java Native Interface) pour invoquer les primitives de leur bibliothèques DLL.

4.2.3 Des points d'accès physique

Il constitue le deuxième maillon et il permet d'une part d'organiser la communication avec le circuit synthétisé et d'autre part de le relier aux broches du FPGA. La communication avec le circuit synthétisé est organisée de façon à offrir controlabilité et visibilité ; nous avons développé des opérateurs mutateurs et accesseurs contenant respectivement les mots "Set" et "Get". Les premiers nous permettent de mettre le circuit dans un état donné, de lui soumettre l'action à exécuter ainsi que l'ensemble des données nécessaires (paramètres). Les seconds nous permettent, une fois le calcul terminé, de lire l'état successeur quand l'action soumise est exécutée ou un indicateur de non activabilité dans le cas contraire.

4.3 Amélioration des performances

À ses débuts, notre approche fonctionnait sous le contrôle direct du vérificateur TLC ; c'était à ce dernier que revenait la tâche de construire les n-uplets (état, action, paramètres). Nous avons vite été confrontés à un goulot d'étranglement ; la lourdeur de la communication était tellement problématique que toute l'approche était remise en cause.

4.3.1 Le lien de communication et le goulot d'étranglement

Le chapitre 8 est dédié au problème du goulot d'étranglement auquel nous avons été confronté dès l'établissement du lien de communication entre le vérificateur TLC et le prototype sur FPGA. Lors de cette première implantation, TLC invoque un calcul pour chaque état atteint, partant d'un état initial, et pour chaque action possible. Pour chaque calcul, un paquet est envoyé et un autre est reçu en guise de réponse ; si la condition d'activation est remplie pour l'action en question dans un tel état, l'état successeur correspondant est calculé puis retourné, sinon la réponse est juste un indicateur de condition d'activation non

remplie.
Nous avons mesuré le temps de communication sur différentes plates-formes FPGA et nous avons constaté que cette dernière était très gourmande. Pour pallier à cette lenteur, nous avons élaboré un modèle mathématique qui suggère des envois groupés (voir la table 8.I).

4.3.2 Le mécanisme d'anticipation

Au lieu d'invoquer le calcul pour juste un couple (état, action) placé en un seul paquet, nous regroupons plusieurs paquets dans chaque envoi dans les deux directions. Le premier envoi consiste à récupérer tous les états successeurs immédiats accessibles à partir de l'état initial, nous anticipons ainsi le calcul pour toutes les autres actions possibles à partir de l'état initial. Quand TLC invoque le calcul pour le même état initial mais pour une deuxième action ou les suivantes, le résultat de calcul est déjà disponible. Comme on peut le comprendre, il y a nécessité de sauvegarder en mémoire les calculs non encore invoqués par TLC du côté logiciel. De façon similaire, il y a nécessité d'une mémoire côté matériel. Dans le deuxième envoi et les subséquents nous plaçons autant de paquets que permettent la capacité du lien de communication et la disponibilité de calculs à soumettre à l'implémentation physique. Chaque nouvel état retourné par la réalisation physique doit lui être retourné, à un moment donné, pour les fins de son expansion. Pour ce mécanisme d'anticipation, nous avons donc réalisé des systèmes de mémoires logicielles et une extension de la mémoire physique. Nous avons aussi élaboré et expérimenté différents algorithmes de gestion de ces mémoires et des communications par leur biais.

4.3.3 Un analyseur d'accessibilité implanté en matériel

Nous lui avons dédié le chapitre 9 et l'annexe I. Il nous permet de réaliser la meilleure performance. Son but est d'éliminer complètement le temps d'attente du vérificateur TLC. Le graphe d'expansion est anticipé en entier. Notons que de cette manière nous obtenons deux expansions indépendantes, la première réalisée par le vérificateur lui même sur l'implémentation de référence et la deuxième par cet analyseur sur l'implémentation physique.

4.4 Conclusion

Notre approche de vérification procède en deux étapes (voir Figure 8.1). L'objet de ce chapitre constitue la deuxième étape qui consiste à connecter TLC au système sous test dans le but de lui permettre de comparer entre le comportement d'une implémentation physique à celui de l'implémentation de référence. Pour ce faire, nous avons mis en place une chaîne d'éléments (voir Figure 8.2). Après avoir démontré la faisabilité de notre approche, nous avons développé différentes approches afin d'améliorer les performances en termes de temps de vérification et de grandeur de systèmes à vérifier. Après essai du mécanisme d'anticipation, nous avons eu à implanter des algorithmes pour réaliser l'analyse d'accessibilité en matériel d'une part et gérer des mémoires caches en matériel et en logiciel d'autre part afin de contenir le graphe d'expansion.

CHAPITRE 5

RÉSULTATS, CONCLUSION ET PERSPECTIVES

Ce travail consiste donc en une évaluation expérimentale, Il réalise l'extension d'une méthodologie de vérification automatique de systèmes logiciels à des systèmes matériels. Toutes nos expériences ont été réalisées sur des prototypes synthétisés sur des cartes FPGAs. Nous avons fait en sorte qu'elles soient indépendantes de cette technologie.

Le but ultime est de parvenir au message "`No error was found`" de TLC. À ce point, nous pouvons conclure que : (1) d'une part l'implémentation de référence satisfait toutes les contraintes et propriétés et (2) d'autre part, l'implémentation physique raffine bel et bien celle de référence.

5.1 Résultats

Les résultats de ce travail peuvent être résumés comme suit :

1. Une démonstration de la faisabilité de notre approche : elle étend le vérificateur de modèles TLC pour vérifier la conformité d'une implémentation FPGA par rapport à la spécification mathématique exprimée en TLA+, le langage de spécification "Natif" à TLC, il est basé sur TLA (Temporal Logic of Actions). Nous avons développé plusieurs exemples d'implémentation et de vérification : le jeu démineur, le microprocesseur MIPS, deux versions du contrôleur d'ascenseur et le microcontrôleur picoblaze. Ce point a fait l'objet de deux publications : [34, 60].

2. Lors de nos travaux nous avons été vite confrontés à un problème de goulot d'étranglement causé par les délais de communication. Nous avons implanté *un analyseur d'accessibilité hardware* (Hardware Reachability Analyser, HRA). Celui-ci est un circuit spécialisé qui permet l'analyse d'accessibilité du circuit physique alors que le vérificateur TLC analyse celle de l'implémentation de référence. Ce point a aussi fait l'objet de deux publications : [33, 36].
   Notons que l'analyseur ainsi que le circuit analysé sont sur un même FPGA. Ceci nous facilite la controlabilité ainsi que la visibilité du circuit d'une part et nous permet d'accélérer l'analyse d'autre part.

3. Le troisième et dernier résultat de notre travail consiste en un ensemble de mesures expérimentales sur différentes plates-formes. Elles sont illustrées par les tables et figures pour les 2 exemples du contrôleur d'ascenseur et du micro-contrôleur à 8 bits, PicoBlaze de Xilinx. Les figures 8.3 et 8.4 illustrent les performances obtenues avec nos différents modèles de caches logiciels pour deux versions (simple et complexe) du contrôleur d'ascenseur.

   La table 5.I donne les temps de vérification (en secondes(s)) pour le cas du contrôleur d'ascenseur à $184,938$ états, $4,623,450$ tests[1] et $526,284$ transitions. Nous avons considéré différentes configurations selon l'implémentation, la plate-forme, le lien de communication, les caches logicielle et matérielle (SRAM).

   Pour les caches logicielles, nous avons cinq versions : la sans cache (No Cache : NC), la version à un état (Single-State Cache : SSC), la cache à plusieurs états (Multi-State Cache : MSC), la cache à plusieurs niveaux (Multi-Level Cache : MLC) et une cache remplie d'un seul coup (One-Shot Cache : OSC) ou contenant la totalité du graphe d'expansion.

   Nous avons aussi une configuration où l'analyseur d'accessibilité est personnalisé en Java (Java Custom Reachability Analyzer : JCRA) et les transferts sont soient multiples (Multiple Transfers : MT) ou singuliers (Single Transfer : ST).

   La dernière configuration est celle où l'analyseur d'accessibilité est implanté en matériel (Embedded Reachability Analyzer And Invariant Checker : ERAIC) et donc synthétisé sur FPGA.

   Le résultat principal qui ressort de la table 5.I est que l'expansion, pour le contrôleur d'ascenseur, avec ERAIC tout seul (non connecté à TLC) nous permet d'atteindre une performance d'un ordre de grandeur variant de 3 à 4 par rapport à une simulation HDL. Par ailleurs pour plus de performance encore, nous avons expérimenté le passage de la plate-forme NEXYS2 à l'ATLYS, avec une mémoire dynamique plus large et plus rapide, une plus grande fréquence d'horloge et un bus de données et adresses plus large. Nous avons remplacé la recherche linéaire dans la table de hachage par une binaire. Nous obtenons une expansion 4 fois plus rapide. Pour ce contrôleur d'ascen-

[1] Il y a un test pour chaque action dans chaque état, mais seules les actions remplissant les conditions d'activation aboutissent à des transitions

Implémentation	Temps(s)	Configuration
Implémentation TLA+	170	TLC + TLA+
Implémentation Java	102	TLC + Java
Implémentation C	110	TLC + C
Simulation	7 020	VHDL Model
XEM3001	10,000	TLC + NC + USB2.0 + FPGA + SRAM
XEM3001	645	TLC + SSC + USB2.0 + FPGA + SRAM
XEM3001	114	TLC + MSC + USB2.0 + FPGA + SRAM
XEM3001	105	TLC + MLC + USB2.0 + FPGA + SRAM
XEM3001	17	JCRA + MT + USB2.0 + FPGA + SRAM
NEXYS2	198	TLC + OSC + USB2.0 + FPGA + SRAM + ERAIC
NEXYS2	11	JCRA + ST + USB2.0 + FPGA + SRAM + ERAIC
NEXYS2	3.3	FPGA + SRAM + ERAIC + recherche linéaire
ATLYS	0.8	FPGA + DDR2 + ERAIC + recherche binaire

Tableau 5.I – Quelques temps de vérification

seur, nous avons une couverture exhaustive en $0.8s$.

Pour d'autres types de circuits comme un microprocesseur, nous pouvons organiser une vérification séparée pour chaque instruction en prenant la précaution de générer la totalité des états dès le départ. Pour le micro-contrôleur PicoBlaze, nous présentons quelques résultats avec les tables 7.I, 7.II et 7.III. Le principal résultat avec le cas de PicoBlaze est la démonstration que notre méthodologie peut s'étendre à des circuits plus complexes tel qu'un microprocesseur.

5.2 Conclusion

Notre objectif était d'étendre la vérification par l'outil TLC aux circuits synthétisés sur des puces FPGA. Jusqu'à une certaine grandeur assez élevée du circuit, nous voulons assurer une couverture exhaustive tout en réduisant le temps de vérification. Nous considérons que les méthodes existantes pour résoudre le problème d'explosion d'états, telles que la décomposition du circuit et la représentation symbolique des états, sont utilisables, dans notre méthodologie, pour des circuits de très grande complexité.

En gros, nous avons rajouté deux gros maillons à la chaîne de vérification qui a constitué notre point de départ. Rappelons que celle-ci a été mise en place par le professeur Robert DeB. Johnston pour la vérification de systèmes logiciels implémentés essentiellement en C

ou en Java. Elle exploite un mécanisme dit de surdéfiniton offert par TLC pour exécuter des opérateurs implémentés en Java plutôt que ceux dans son langage “native” auxquels on ne substitue que leur prototype TLA+.

Le premier maillon est consacré à la connexion du monde Java à la carte FPGA. Il consiste en un ensemble de points d’accès logiques 4.2.2. Le deuxième maillon est consacré à la connexion sur la carte FPGA du circuit sous test à l’interface vers le domaine Java. Il consiste en un ensemble de points d’accès physiques 4.2.3.

Nous avons, par la suite, été amené, dans un premier temps, à améliorer les performances 4.3 en terme de temps de communication par des systèmes de caches au niveau matériel et logiciel ; nous avons besoin, de part et d’autre du lien de communication, de mémoriser les paquets avant leur envoi. Au niveau de la machine hôte, nous avons développé trois algorithmes dont l’objectif est d’exploiter au maximum la capacité du lien de communication utilisé. Notre procédé a été progressif, nous avons donc commencé par emonter une mémoire cache à un niveau. Pour le cas du contrôleur d’ascenseur il est question de faire un transfert vers le circuit pour le calcul des 24 successeurs éventuels d’un état et les mémoriser en cache jusqu’à ce que TLC commande un calcul pour un nouvel état. Nous avons développé, dans un deuxième temps, deux variantes de caches, une a plusieurs états dite “multi-states” et une autre à deux niveaux. Leur principe de base est identique : on envoie pour autant d’états que le lien de communication permet. La taille du paquet, en terme du nombre d’états pour lesquels on demande un calcul, est déterminée à l’avance selon sa bande passante et la taille d’un envoi pour un état. La stratégie consiste à exploiter au maximum la capacité du lien à chacun des transferts.

L’avantage de ce système est qu’on effectue un seul et unique transfert pour un même état. Quand on invoque le calcul pour l’état désigné par TLC, nous lui joignons le nombre maximal d’états déjà disponibles et non encore soumis au circuit physique pour anticiper leur calcul. Les deux variantes se distinguent par la façon dont est organisée la cache, dans la “multi-states”, tous les états sont au même niveau et la recherche de l’état invoqué par TLC se fait parmi tous les états disponibles à chacune de ces invocations. Nous avons jugé intéressant d’éliminer les recherches multiples de l’état pour chacune des actions.

Dans l’approche à 2 niveaux, nous entretenons une table de transitions pour l’état courant. À chaque première invocation d’un état, cette table est mise à jour à partir de la cache du deuxième niveau, ou après communication avec l’implémentation physique. Pour

l'exemple du contrôleur d'ascenseur, chaque état est invoqué 24 fois pour les 24 actions possibles. Suite à la première invocation, la table de transitions du premier niveau va contenir les 24 transitions possibles et lors des invocations suivantes, l'état successeur éventuel ou la décision de non activabilité y est directement retournée au vérificateur.

En plus des caches au niveau de la machine hôte, nous avions eu le même besoin de mémoriser les paquets du côté du circuit physique. À chacun des transferts vers le circuit, est retourné un transfert de même longueur. Nous avons donc mis en place un contrôleur pour écrire les calculs dans une mémoire externe à la puce FPGA avant de les envoyer vers la machine hôte.

D'après les résultats des essais (voir les Figures 8.3 et 8.4) que nous avons fait, les résultats sont très satisfaisants. Comme on peut s'y attendre, ils le sont encore davantage avec les lien USB2.2 à plus haute vitesse.

Nous sommes passés par la suite à vouloir éliminer complètement le temps d'attente du vérificateur. Nous avons éliminé le contrôle direct de celui-ci sur le circuit physique ; ce n'est plus à celui-ci qu'incombe l'action de déterminer le n-tuple à être considéré par le circuit physique. Nous n'attendons donc plus les invocations du vérificateur pour les envoyer vers le FPGA. Après avoir expérimenté l'analyseur d'accessibilité en Java, nous sommes amenés à conclure que c'est intéressant de supprimer complètement les envois vers la carte FPGA. Nous avons donc implémenté l'analyseur d'accessibilité du circuit physique en matériel, nous l'avons appelé ERAIC (Embedded Reachability Analyzer And Invariant Checker).
Avec ERAIC, le matériel anticipe, à vitesse maximale et de façon autonome, la totalité du graphe d'expansion du circuit physique. Il opère ainsi indépendamment du vérificateur de modèle. Le rôle de ce dernier dans ce cas est de faire l'analyse d'accessibilité de l'implémentation de référence en comparant entre les deux graphes d'expansion correspondants pour déterminer la conformité du circuit physique par rapport à l'implémentation de référence. En plus de procéder à l'analyse d'accessibilité, nous offrons aussi la possibilité de vérifier certains invariants en matériel. Les assertions peuvent être ainsi vérifiées sur la machine hôte par le vérificateur de modèle, ou pour certains types d'invariants, sur la plateforme physique. À notre connaissance, nous sommes les premiers à connecter "l'assertional model checking" à l'exploration d'états d'un circuit synthétisé.

ERAIC offre la possibilité de choisir l'état à partir duquel commence l'expansion, ceci

ouvre la voie pour considérer la vérification par décomposition dans le cas de circuits très complexes. On peut aussi exploiter le parallélisme, les BDDs pour la représentation symbolique des états et des combinaisons matériel/logiciel. ERAIC est conçu de façon à être indépendant de toutes technologie. Son mécanisme est basé sur la technologie Wishbone [99] de sorte à être flexible et portable.

Pour compléter notre travail, nous avons élaboré plusieurs exemples : nous avons implémenté et vérifié le jeu démineur, le microprocesseur MIPS, deux versions du contrôleur d'ascenseur et le micro-contrôleur PicoBlaze.

Notre apport par ce travail est donc multiple :

1. La vérification du circuit physique au lieu de juste son modèle abstrait.

2. Nous permettons une exploration exhaustive de l'espace des états. Ceci est, à notre sens, faisable en combinaison avec les méthodes déjà utilisées pour la résolution du problème d'explosion d'états dans le monde abstrait. Nous appliquons le principe soutenu par Dijkstra disant que tous ce qui s'applique à un programme s'applique à un circuit. Nous avons la décomposition de systèmes, la représentation symbolique de l'état du système, les méthodes SAT et la compression de données. Sans oublier que nous ne sommes plus à l'ère d'un espace mémoire presque illimité

3. Une contrôlabilité et une visibilité ; nous avons un mecanisme qui nous permet de spécifier une valeur à l'état, à une action choisie et de récupérer le résultat qui est l'éventuel état successeur ou l'indicateur de non activabilité de l'action considérée.

4. Nous assurons l'inexistence d'erreur du fait que l'espace des états et des actions est exploré dans sa totalité. Notons que cette méthodologie est très adéquate au domaine d'IPs (Intellectual properties) pour s'assurer que celles-ci sont exemptes d'erreurs.

5. Nous nous servons des mathématiques dont la puissance n'est plus à démontrer, pour le raffinement et la spécification.

6. La tâche de spécification se trouve facilitée du fait que nous utilisons un langage basé sur les mathématiques standards.

7. L'utilisation des mathématiques standards intégrées dans un langage fonctionnel assure une puissance prouvée de notre système.

Nous sommes très satisfaits des résultats. Une des configurations complète une vérification exhaustive du contrôleur d'ascenseur en 3 secondes vs. 2 heures par une simulation VHDL équivalente. Notons que pour permettre cette comparaison, la simulation intègre aussi la version comportementale de notre analyseur d'accessibilité implantée en VHDL.

5.3 Perspectives

On peut imaginer plusieurs suites possibles au travail que nous avons effectué dans le cadre de cette thèse.

1. Améliorer davantage les performances
   - Utilisation de BDDs et d'algorithme de compression de données
   - Exploiter le parallélisme au niveaux matériel et logiciel. On peut penser à utiliser le multi-threading de TLC et un circuit implémenté sur 2 ou plusieurs FPGAs.
   - Accélérer le vérificateur par des extensions Java
   - Étudier la compatibilité de notre approche avec d'autres vérificateurs de modèles ou de manière générale avec d'autres méthodes de vérification formelle.
   - Implémenter le test de conformité en matériel
   - Essayer et comparer différentes combinaisons matériel/logiciel,
   - Considérer l'idée d'une réalisation d'un vérificateur matériel. On peut imaginer TLC rouler sous forme de circuit numérique et communiquer de très près avec le circuit à vérifier.
2. Automatiser le processus d'intégration de système dans notre environnement.
3. Réaliser le cas ou ERAIC serait sur un FPGA séparé.
4. Automatiser la traduction de l'implémentation de rréférence vers un circuit synthétisable.
5. Expérimenter notre approche à très grande échelle, au niveau industriel.
6. Étudier et expérimenter la faisabilité de notre approche avec les circuits ASICs.

7. Considérer l'utilisation de port JTAG pour prévenir les cas d'insuffisances éventuelles du nombre de pins.

8. Comme outil pédagogique pour introduction au domaine TLA+ en conjonction avec la synthèse et la vérification de circuits numériques.

Il serait envisageable aussi d'intégrer la norme JTAG pour rendre la méthodologie applicable pour des circuits présentant un très grand nombre d'E/S d'une part et permettre l'application de la méthodologie vers les autres technologies de circuit logiques telles ASICs et VLSI (Full Custom).

Deux autres ouvertures très possibles seraient de déplacer le maximum d'intelligence de vérification vers le côté matériel et implanter le protocole JTAG sur un support physique de plus grande performance tel que USB.

Partie II

CHAPITRE 6

ARTICLE 1

Ouiza Dahmoune and Robert De B. Johnston, **“Applying Model-Checking to Post-Silicon-Verification : Bridging the Specification-Realisation Gap”**, © [2010] IEEE. Reprinted, with permission, from [34].

RÉSUMÉ

Nous présentons une nouvelle méthodologie pour la vérification formelle de circuits numériques “Concrets”. Nous appliquons la vérification de modèles (model checking) au prototype FPGA (Field Programmable Gate Array) du circuit. Nous somme en mesure d’établir un pont fonctionnel entre les comportements du circuit dans le monde de spécification, modélisation et analyse abstraites, avec ceux dans le domaine post-silicium.

Nous suggérons que, comparée aux techniques conventionnelles de développement de circuits physiques, cette approche 1) peut opérer à plus haut niveau d’abstraction, 2) offre un test automatique de circuits physiques au lieu de juste leur modèle, 3) réduit ou élimine les simulations lentes et laborieuses, 4) est à des ordres de grandeur plus rapide qu’un simulateur HDL (Hardware Description Language), et, 5) convient parfaitement au co-développement matériel/logiciel.

6.1 Abstract

We present a new methodology for Formal Verification of “Concrete” Digital Circuits. We apply Model checking to a Field Programmable Gate Array (FPGA)-based prototype of the circuit. We are able to establish or demonstrate a working bridge between circuit behaviors in the abstract world of specification, modelling and analysis, and the ones in the post-silicon domain.

We suggest that, compared to conventional hardware design techniques, this approach 1) can operate at a much higher level of abstraction, 2) offers automated testing of the physical device in contrast to just a model, 3) reduces or eliminates laborious and time-consuming simulations, 4) is orders of magnitude faster than an HDL (Hardware Description Language) simulator, and, 5) is very suitable for Hardware/Software co-design.

6.2 Introduction

According to some experts, the verification process is the most time-consuming in the overall process of circuit development. This project aims at reducing that figure in the context of digital hardware design. We are interested in a system’s overall behavior, or function, without regard to certain implementation details such as timing and area consi-

derations. By ‘verification’ we mean comparing the behavior of an implementation with that of a specification, or, a set of requirements. Approaches to verification include 1) formal modelling/analysis, 2) simulation of proposed designs, and 3) physical device testing/validation.

We assume that the conventional hardware development process consists of the following phases :

1. formal and/or transaction level modelling and analysis
2. HDL coding of the desired device and eventually SAT based property checking (semi-formal)
3. HDL simulation enhanced with Hardware emulation or/and acceleration
4. synthesis and physical implementation with formal equivalence checking and eventually assertion based verification
5. industry has used Boundary-Scan Testing, to test physical devices once implemented in silicon.

These phases seem to be completely separate from each other. We can conclude that traditionally, a circuit is at first modeled and verified in its abstract form, and then, physically implemented and tested. The two processes of verification and testing are independent and different. In this context, we suggest replacing, or augmenting, an HDL simulation phase with a direct connection between the abstract world of modelling and the concrete one of physical implementation. We do not suggest eliminating an HDL simulation for purposes of timing analysis ; we do suggest following a timing analysis by a functional analysis for which an HDL simulator is unnecessary.

It’s important here to underline that almost all traditional formal verification tools verify only a *model* and not the physical implementation itself. In this sense, a model checker resembles any other CAD program that predicts physical behavior based on some theoretical model. A starting point for the present article is this distinct connection between an “abstract” modelling world and the “physical” implementation world. Some works, such as [6, 112], refer to “Verifying a Synthesized Implementation”, but, in our understanding this is limited to a non-physical, pre-silicon level.

In brief, we suggest replacing the HDL simulator by a model-checker linked to a physical implementation, possibly on an FPGA, of the desired system. On the one hand, this raises the level of abstraction to that of the model-checker's native language, and, on the other hand, extends the verification "downwards" to a concrete implementation in silicon.

One feature of the approach we suggest is a separation of the verification process into two steps : (1) showing that an abstract Reference Implementation satisfies all the Invariance (formal assertions) and Progress properties required in its specifications (requirements), and (2) showing that a concrete Implementation is a refinement, as defined by Refinement Criteria (see below), of that Reference Implementation. An implication of (2) is that a concrete implementation cannot produce behaviors not allowed by its requirements specification ; this means that if the implementation produces an error, then the specifications are wrong or incomplete. This addresses the question of how to differentiate between a logical flaw in the specifications and an error in the physical realization. The document outline is as follows. Section 6.3 overviews related works. Section 6.4 describes the drawbacks of HDL Simulation. Section 6.5 illustrates the application of reachability analysis to physical implementations. Section 6.6 presents the proposed configuration and design flow in our methodology. Section 6.7 presents an elevator controller case study. Section 6.8 introduces the major problem we met after establishing the connection between the implementation and its mathematical specifications. Section 6.9 presents the preliminary results. We conclude in section 6.10.

6.3 State of the art

As it's illustrated in [89], many works have been done in pre-silicon verification and/or post-silicon validation. This last one is because of the incompleteness of the one before ; post-silicon validation aims to determine what's wrong when a failure is detected [69, 101, 114]. The first work we met pointing, from a theoretical perspective, to post-silicon verification is [46]. There are 2 other papers [39, 72] entitled with "post-silicon verification", but for our comprehension, they rather deal with post-silicon validation. The closest works we met trying to bridge the gap between the two domains [94] are [5, 103, 108]. We suggest that our methodology, instead of bridging the two worlds, extends pre-silicon verification to post-silicon verification, and represents a new and unique way based on full controllability and visibility without being limited to functional (logical) malfunctions.

Since the beginning of system design, many solutions have been developed to automate and help the verification process :

6.3.1 Formal methods

In the last decade, hardware formal verification and validation are attracting more attention due to their consistency based on a mathematical proof, high level of abstraction and, most importantly, the potentially exhaustive exploration of the reachable state space. On one hand we have Model checking with temporal logic which offers a completely automated assertional verification but must deal with the problem of state explosion. Much work, like Boolean Satisfiability (SAT) or Binary Decision Diagrams (BDD) techniques and others, has been done to alleviate this problem. On the other hand, we have theorem provers, which are assertional and don't have this problem but are not completely automated. Other works consider combining techniques.

A potential shortcoming of existing tools, for hardware model checking, is that results may not carry forward into post-silicon flows ; The lowest level they consider is synthesizable Register Transfer Level (RTL). We can cite VIS [1] and FormalCheck [14] Model Checkers ; others consider higher levels.

6.3.2 Raising the level of abstraction in the HDL

One direction has been to specify the circuit at the highest level of abstraction to eliminate the maximum of low level details and thus raise designer productivity. This is a major research orientation ; we can cite Transaction Level Modeling (TLM), SystemC, Open Verification Methodology (OVM), Property Specification Language (PSL) and SystemVerilog.

6.3.3 Automating testing : test-benches

Automating the process of testing has been and is still a big challenge ; in the context of functional simulation, much work has been done to develop test-benches (HDLs like VHDL and Verilog, HVLs : High-level Verification Languages like e and Open Vera, ATPG : Automated Test-Pattern Generation) and to extend them with assertions(ABV : Assertion-Based Verification) which are dynamically evaluated in hardware assisted simulation like Hardware emulation and acceleration approaches. Hardware debugging approaches are also used

in the context of FPGA-Based prototyping.

6.4 Motivation : Drawbacks of HDL Simulation

The verification task is an important and inherent part of any system's development. Industry has mostly relied on simulation to verify HDL designs. HDL simulation requires :

1. an HDL description of the *device to be implemented*, the Implementation Under Test (IUT) ;
2. an HDL description of a mechanism, usually known as a *test-bench*, to send stimuli to, and receive responses from, the IUT ;
3. a test plan, or *test script*, which defines sequences of acceptable (stimulus,response) pairs (transactions) ;
4. a *simulator*, or simulation engine, to execute the test scripts.

The simulation approach has several drawbacks. The first, and most obvious, is that a simulator is *slow*, even with hardware-based acceleration and emulation. Our experiments show a three to four orders of magnitude difference. HDL simulation was invented at a time when there was no alternative to the expense and delay of silicon fabrication ; nowadays, modern FPGA technology offers the capability of creating a silicon implementation at a tiny fraction of the previous cost and time lag. This suggests that we could dramatically reduce the cost and time-to-test by verifying the physical implementation directly ; this would require a communication link, analogous to a Joint Test Access Group (JTAG) link, between tester and implementation. Reducing the time-to-test can also lead to increasing test coverage.

The second drawback concerns the need for an HDL test-bench, which takes the form of an HDL program to exercise and monitor the HDL description of an IUT during simulation. Some reports state that designing the test-bench can be at least as complex and laborious as designing the IUT itself.

The third drawback deals with the need to define HDL test sequences, or scripts, with which to drive the test-bench. This has important scalability issues and can not only be laborious, but worse, gives rise to questions of objectivity and coverage, or completeness. Ideally, we would like a fully automated and objective way of generating test patterns with clearly-defined coverage.

6.5 Applying Reachability Analysis to Physical Implementations

At the core of a model checker is a reachability algorithm which computes the set of all states reachable from some initial one(s). It is a loop which works by selecting some state s, typically from a queue or stack, to expand, i.e., examine for possible transitions. Successor states s' are computed for each enabled, or "executable", transition. If a successor state has been visited before, it is discarded. Otherwise, it is tested for required invariance properties, and, if OK, it is placed on the queue or stack for future expansion, or if not, the process terminates with producing a counterexample. When all successor states of a given s have been processed, the loop starts over. The process stops when there are no more unexpanded states.

Let $Actions = \{A, B, \ldots\}$ be a finite set of actions, and suppose that each action can have some parameter $x \in X$ associated with it. Here, X is an arbitrary finite set. Then, let $Enb_{A,x}(s)$ denote the predicate, or boolean-valued function, which specifies whether action A, with parameter x, is enabled in state s. In TLA+, $Enb_{A,x}$ is called an Enabling Condition. Let $Succ_{A,x}(s)$ denote the set of succesor states of s under the action $A(x)$; $Enb_{A,x}(s)$ is equivalent to the statement $Succ_{A,x}(s) \neq \emptyset$.

To simplify, we consider only deterministic behaviors. A system is termed *deterministic* iff[1], for any choice of A or x, $Succ_{A,x}(s)$ is either empty or a singleton $\{Opn_{A,x}(s)\}$, where $Opn_{A,x}$ is a transition function which maps a state s to its unique successor. In this case, a breadth-first version of the above reachability algorithm could be expressed in pseudo-code as :

```
ModelState s0 = InitState_x() ;
enter(s0, queue) ;
while (existUnexpandedStates(queue)) {
   ModelState s = nextStateToExpand(queue) ;
   forall A in Actions: forall x in X: {
    if (Enb_A,x(s)) {
           ModelState s’ = Opn_A,x(s) ;
           if (!contained(s’,queue)) {
             if (satisfiesInvariants(s’))
                enter(s’, queue);
```

[1] "iff" abbreviates logical equivalence : if and only if

```
            else HALT ;
          } // end if !contained
      } // end if enabled
  } // end forall
  setExpanded(s, TRUE) ;
} // end while
```

As far as we know, our work is the first attempt to connect assertional model checking to post-silicon hardware testing. It's worth underlining the main difference between simulation and reachability analysis. In the latter case, the state of the IUT is considered explicitly for property conformance and for each possible transition the process calculates the eventual successor. The promise of reachability analysis is to surmount the fundamental limitation of simulation by, on one hand, offering an exhaustive space traversal and considering all possible inputs at every state, and, on an other hand, assuring that every action enabled in any reachable state of the implementation is also enabled in the corresponding state of the reference, and that the successors correspond.

6.5.1 Raising the Level of Abstraction

Based on the precision and expressive power of mathematics, we suggest that the TLA+ language [73] is an excellent formalism for specifying physical system properties [15] at a very high level of abstraction. TLA+ is a high-level language which expresses most of standard mathematics : Propositional Logic, First-Order Logic with Equality, Set Theory including Functions, Relations, and Number Theory (Integers). TLA+ extends TLA which is part of Linear Temporal Logic, and includes some higher-order extensions (operator arguments), and a powerful definitional facility based on the lambda calculus. The syntax closely resembles normal mathematical notation. It follows that one of the most powerful features of TLA+ is that it is *abstract* in the sense that it is independent of particular implementation languages or technologies.

6.5.2 The TLC Model-Checker

The main verification tool used here is Leslie Lamport's TLC Model Checker. It is a Java program which accepts mathematical specifications written in TLA+ and, assuming

the model is finite, verifies formulas identified therein by first performing a reachability analysis.

6.5.3 Over-riding the Model-Checker's native model

A model checker computes functions using internal procedures to process state-transition information expressed in its native language. The basic idea here is to make the model checker use an *external implementation* itself to generate these quantities. TLC has an especially simple mechanism for replacing calls to native functions by calls to general, external, Java programs ; it is simple in that it does not require recompilation of TLC.

The approach involves a separation and a division of labour between model checker and target implementation : the model checker determines the testing sequence, records results, and verifies the behavior, while the target implementation provides the enabling conditions and performs state transitions. The model checker can now be viewed as an automated test system for verifying external implementations.

To make the model checker use an external implementation to provide the quantities of interest, we assume it must be able to :

1. set the state of the external implementation to a given value (state controllability),
2. get, i.e., read or observe, the implementation's state (state visibility), and
3. activate, or invoke, the implementation's procedures for initializing itself, determining enabling conditions, and making transitions to successor states.

We also have to be able to convert back and forth between state representations, often radically different, in the two systems.

So we postulate that the model checking program has to have available to it the following functions :

1. `setState(z)` - sets the implementation's state to the value z.
2. `getState()` - returns the value of the implementation's state.
3. `execute("procname",x)` - invokes the implementation's procedure called `"procname"` with argument `x`.

4. `phi(s)` - maps a state `s` represented in the model checker domain into its representation in the implementation domain.

5. `psi(z)` - maps a state `z` represented in the implementation domain into its representation in the model checker domain.

6. `truthValue(z)` - maps implementation state `z` into a Boolean value corresponding to the result returned by invoking a predicate on the implementation [2].

The constraint on `phi` and `psi` is that, for all Model States `s`, `psi(phi(s)) = s`.

Given these functions, we can rewrite (over-ride) the three functions $InitState_x()$, $Enb_A, x(s)$, and $Opn_A, x(s)$ of the pseudo-code description in order to use the external implementation as follows :

```
ModelState InitState_x() {
  execute("Init", x) ;
  return psi(getState()) ;
}

Boolean Enb_A,x(ModelState s) {
  setState(phi(s)) ;
  execute("Enb_A", x) ;
  return truthValue(getState()) ;
}

ModelState Opn_A,x(s) {
  setState(phi(s)) ;
  execute("Opn_A", x) ;
  return psi(getState()) ;
}
```

6.5.4 Refinement criteria

The approach here recognizes that there is generally an infinite variety of ways in which to realise, or implement, a system specification. The question is how to prove that some

[2] Strictly speaking, this function isn't necessary, but it simplifies the explanation.

proposed implementation correctly realises that specification. We use a simplified form of the TLA concept of *refinement* to do this.

Refinement provides a way to compare the behaviors of two different systems given a *refinement mapping* ψ between their state spaces.

6.6 Proposed Configuration and Design Flow

The proposed configuration requires a Host Computer which contains the Abstract Model descriptions, the Verifier software (the TLC Model Checker) and a java communication interface, a Target Platform which contains the actual IUT wishbone compatible core expander, and a Communication Link between Host and Target. Note that the Abstract descriptions and the concrete implementations could be developed separately by respective specialists. Our methodology suggests to put a mechanism connecting them to make the verifier compare their behaviors.

As a short description of our proposed design flow, it consists of the following steps :

- Identify the observable quantities of the desired system ; these are functions (projections) of its state s.
- Identify the actions which define a system's behavior including any parameters they may require.
- Express the requirements, i.e., desired properties, in terms of the observables. We'll call the result of these first three steps the "Requirements Specification".
- Develop an (abstract) reference implementation, and verify that it satisfies the above requirements. The result will be called the "Reference Implementation".
- Express the conditions an implementation must satisfy in order to be a refinement of the Reference Implementation. These will be called the "Refinement Criteria".
- Develop a post-silicon(Concrete) Implementation and perform timing and space analysis.
- Put the connection mechanism between the 2 implementations.
- Verify that the Concrete Implementation refines the Reference Implementation.

6.7 Case study : An elevator controller

We aimed to illustrate our approach by developing and verifying 2 case studies : an elevator controller and a Picoblaze 8-bit Microcontroller.

We developed and “Synthesized” a VHDL implementation that we verified by comparing its behavior, as implemented on an FPGA, with the abstract reference model. We can inject some bugs in the circuit and see how easy and fast it is to find the states which are affected. In what follows, we’ll identify the different parts of the verification chain for an elevator controller.

6.7.1 The TLA+ Model Domain

6.7.1.1 The Requirements Specification

This module expresses the requirements that an elevator controller implementation should satisfy. The module is *instantiated*, i.e., imported, by the Reference Implementation. It specifies what an implementation must provide :

- constant values, eg., the number N_e of floors
- the observable output functions of state
- the Initial Conditions predicate
- a Type Invariant
- implementations of the Enabling Conditions and State Transformations
- the Invariant properties to be satisfied
- the Liveness properties to be satisfied

It also defines the behavioral specifications, $Spec$, for an elevator controller.

6.7.1.2 A Reference Implementation

This module provides an abstract reference implementation of an elevator controller. It is *abstract* in the sense that it is independent of any “real” implementation mechanism. The state space, for example, is defined in terms of purely mathematical concepts such as

numbers, sets, and symbols (strings). It is a *reference* implementation in the sense that other, more concrete, implementations are compared to it; it is the "gold standard" of elevator behavior.

This module instantiates the above requirements module. TLC verifies that it satisfies all the criteria, such as Invariants and Liveness properties, specified in the latter module. Verification of this module demonstrates existence of a feasible solution.

6.7.1.3 A Refinement Criterion for Implementations

This module is similar to requirements but instead of directly defining the safety and liveness properties for an implementation, it defines the criteria an implementation must satisfy in order to be a *refinement* of the Reference Implementation. In particular, a candidate implementation must supply the definition of a refinement mapping ψ, mapping its state space into that of the Reference Implementation, to be used during its verification. Successful verification proves that ψ is an homomorphism from the concrete system into the reference one. This module is instantiated by all "concrete" implementations.

6.7.1.4 A bridge to the external domain

This module imports an instance of the Implementation to be tested as well as an instance of the Refinement Criteria. This is the bridge between modelling world and physical implementation world where the two are compared.

6.7.1.5 An interface to the external domain

This is a TLA+ module containing the function prototypes for a Java program which over-rides normal TLC execution. It is like a C header file or Java interface class which presents function signatures. The Java program, which is next in the chain, implements those functions. TLC has a mechanism where one can over-ride a TLA module by a Java program simply by placing the Java class in the special directory tlc/module. TLC automatically checks this directory to see if an over-riding class exists for any TLA+ module. This directory also contains a collection of utility classes to simplify the translation between TLC values and Java data.

6.7.2 The Java Interface

6.7.2.1 The Java over-ride module

This is an interface between the TLC world and the communication link to an external realisation. It contains Java implementations for the operators defined in the abstract TLA+ world. Other than an initialisation operation, the operators typically have an Enabling Condition part, which returns a Boolean value, and a Successor part, which returns a state value. The computation of these values is carried out on the target platform ; arguments and return values are exchanged over the communication link. To do this, the over-ride module communicates with a Link Adapter module which may choose a Cache algorithm to use (see section 6.8).

6.7.2.2 The Adapter

This interface adapts to the form of communication with the outside world. It's in this part of the chain that the choice of target platform as well as communication libraries (RS-232, USB1.1,USB2.0, Ethernet,..) is made.

6.7.3 The External Implementation

Whatever the Target Platform, it consists of an interface part and an implementation part. We have, for example, an interface developed for the OpalKelly XEM3001 card and an other one for the Digilent NEXYS2 card, both of which are USB2.0-enabled FPGAs with on-board RAM. To evaluate the way we can apply our methodology to other circuits, we developed an other simpler case study which is the Arithmetic Logic Unit (ALU) part of the Picoblaze. After separately developing the TLA+ and VHDL parts, we had essentially in a first step to define the ALU state and, in a second step, to integrate the ALU design to our verification system in the VHDL side. We are repeating the process for a more consistent and significant example (the Xilinx Picoblaze Microcontroller).

6.8 Communication Bottleneck

The main problem we met after establishing the connection was the delay introduced by the latency of the communication link. We had to develop strategies to overcome this

issue by bundling multiple tests into a single transfer.

We have improved the performance in terms of verification times by :

- Hardware and software look-ahead (predictive) caches : we have developed three algorithms designed to exploit the bandwidth of the communication link. The advantage of this system is that it anticipates TLC's future needs and obtains them automatically via transfers which are as large as possible and hence reduce the effect of link latency.

- In a following step, we implemented the process of state expansion of the reachability analysis in Hardware, we aimed at a universal core expander with a Wishbone compatible structure. In this way the 2 parts don't communicate directly, the hardware part anticipates for the overall graph expansion.

6.9 Preliminary Results

The original objective of demonstrating how a model checker could be extended to an implementation verifier has been achieved. The goal is to get the "No error was found" message from the model checker. At this point, the main conclusions are that (1) the Reference Implementation satisfies all its requirements and (2) the Physical Implementation is a refinement of the Reference one. In case of any problem, we expect to get an error trace.

We have performed actual measurements on different FPGA platforms. To overcome very high USB latency times, much of the work involved, on one hand, building predictive cache memories on the FPGA's. On the other hand, we implemented the reachability analysis in Hardware. In this way, the model checker and the IUT don't communicate directly, the Hardware extensions construct the overall graph expansion.

The table 6.I resumes some verification times(in seconds(s)) for the elevator controller with different configurations considering the Implementation (Imp.), the platform, the communication link, the Software (No Cache(NC), Single-State Cache(SSC), Multi-State Cache(MSC), One-Shot Cache(OSC), a Java Custom Reachability Analyzer(JCRA), Multiple Transfers(MT) or Single Transfer(ST)) vs. the Hardware ((SRAM) memory or/and an Embedded Hardware Reachability Analyzer(EHRA)). At present, we are working on reducing the timing (198 s) of the NEXYS2-TLC-ERHA configuration to get it below the

Imp.	Time(s)	Note
TLAModel	170	TLC + TLA+ Imp.
Simulator	7 020	VHDL Model
XEM3001	10,000	TLC + NC + USB2.0 + FPGA + SRAM
XEM3001	645	TLC + SSC + USB2.0 + FPGA + SRAM
XEM3001	114	TLC + MSC + USB2.0 + FPGA + SRAM
XEM3001	17	JCRA + MT + USB2.0 + FPGA + SRAM
NEXYS2	198	TLC + OSC + USB2.0 + FPGA + SRAM + EHRA
NEXYS2	11	JCRA + ST + USB2.0 + FPGA + SRAM + EHRA
NEXYS2	3.3	FPGA + SRAM + EHRA alone

Tableau 6.I – Some verification times

XEM3001-TLC one (114 s). This is feasible because without TLC we got respectively 11 s and 17 s and 3.3 s for the pure EHRA configuration.

6.10 Conclusion

We extend formal verification to checking the overall behavior of post-silicon implementation. We use the reachability algorithm of a model checker which computes the set of all states reachable from some initial one(s) and tests a real system(eg, a prototype on FPGA) by repeatedly invoking functions, and verifying that each new state encountered satisfies an invariant property(formal assertion). In the approach we propose here, calculation of these functions is carried out by the target implementation itself, *not an abstract model*. The basic idea was to create, on one hand, a communication path or "test harness", on an other hand, a translation mechanism, between the model checker and the implementation under verification. This mechanism relies on full state controllability and observability. Note that for more performance, flexibility and the ability to verify complex systems, we implement the reachability analysis in hardware [33].

To complete our research, we suggest to, on one hand, demonstrate the possibility to apply our methodology to more complex circuits. On an other hand, we will show more

concretely how our methodology is more effective and faster than simulation and how, in the absence of simulation, this will point out the path and the state where the missing physical to abstract states correspondence happened if any. More details about defining reference implementation and refinement criteria are necessary, this will require an other paper.

In conclusion, we believe the results are encouraging enough to extend the work to more substantial problems, to other model checking-based approaches and to a variety of implementation platforms.

CHAPITRE 7

ARTICLE 2

Robert De B. Johnston and Ouiza Dahmoune, **“Overview of Applying Reachability Analysis to Verifying a Physical Microprocessor”**, © [2011] IEEE. Reprinted, with permission, from [60].

RÉSUMÉ

Ce papier illustre l'application d'un vérificateur de modèles pour la vérification fonctionnelle d'une implémentation physique d'un microprocesseur. Les objectifs sont (1) tester à la vitesse du matériel, (2) automatiser la tâche de création de bancs d'essai, et (3) relever le niveau d'abstraction pour la spécification.

Le vérificateur de modèles TLC de L. Lamport's est utilisé pour vérifier une version personnalisée du micro-contrôleur (Micro-Controller Unit : MCU) PicoBlaze de Xilinx ; le MCU est spécifié en VHDL et implémenté sur un FPGA (Field Programmable Gate Array).

La vérification expérimentale consiste de montrer que le comportement du circuit physique est conforme à la spécification mathématique de son jeu d'instruction (Instruction Set Architecture : ISA), pour un sous ensemble sélectionné d'instructions et d'arguments.

7.1 Abstract

This paper outlines the application of a Model-Checker to the functional verification of a microprocessor's physical electronic implementation. The goals are (1) testing at hardware speed, (2) reduced labor of testbench creation, and (3) a raised level of abstraction for the specifications.

L. Lamport's TLC Model-Checker is used to verify a custom version of the Xilinx PicoBlaze Micro-Controller Unit (MCU) ; the MCU is specified in VHDL and implemented on a Field Programmable Gate Array (FPGA).

Experimental verification consists of showing that the physical device behavior conforms to a mathematical specification of its Instruction Set Architecture (ISA), for selected subsets of instructions and arguments.

7.2 Introduction

7.2.1 Summary

The overall goal is reduced time and increased economy in the testing/verification phases of digital system design and implementation. This paper presents an experimental evaluation of a novel verification framework based on the application of model-checking

technology to verifying *physical* digital systems. In this framework, the physical system itself is used to provide the state-transition information required by the Model-Checker.

The experiment reported is the design, implementation in hardware, and verification, of a custom version of the Xilinx PicoBlaze 8-bit Micro-Controller Unit (MCU). This version of the Xilinx device includes a Test Access Port (TAP) to provides state controllability and observability ; this is the Device Under Test (DUT). It is defined using the Hardware Description Language (HDL) VHDL and realized physically on a Xilinx Field Programmable Gate Array (FPGA). L. Lamport's TLA+ Tools provide (1) the specification language TLA+ and (2) Model-Checker TLC to drive the verification.

Experimental results are presented for several different PicoBlaze Test Suites. They show that, in general, the verification times for the (TLC + Physical Implementation) combinations are no more than twice those of the (TLC + Abstract TLA+ Implementation) combinations.

To accelerate state-transition graph generation, a custom Hardware Reachability Analyzer (HRA) circuit [33], called "Grex", is also implemented on the FPGA Platform. Within memory limits, it performs a breadth-first graph expansion of the DUT's state-transition system starting from a given power-up state. It can also perform on-the-fly correctness checks, such as verification of Invariants. A major result is that the HRA performs an analysis comparable to the Model-Checker's *three to four orders of magnitude* (around 2,500 times) faster. It executes about 1.5 million tests per second. This leads to different avenues for further development.

After presenting some background information, this document gives a brief overview of the Model-Checker modules and the hardware components, and then presents experimental results.

7.2.2 Background and Motivation

Conventional hardware verification methods include :

1. Theorem Proving (HOL, Isabelle, PVS, Larch)
2. Model-Checking (SPIN, NuSMV, TLC)
3. HDL Simulation (GHDL, ModelSim)

4. IEEE 1149 Boundary-Scan Technology (JTAG)

Of these, only the JTAG method tests physical hardware ; it requires a communication link, typically USB, between a host computer and the device under test (DUT). The other three methods treat software abstractions, or models, of the device. The first two express models in high-level terms. For example, the native language of the TLC Model-Checker [75] is TLA+, which expresses most of standard mathematics in a familiar notation.

Drawbacks of HDL simulation include :

1. Does not test *physical* implementations
2. HDL Simulation is *slow*
3. Test Coverage is *limited*
4. Construction of an ad hoc HDL simulation testbench is *laborious*
5. Correctness requirements are expressed in the HDL, not in higher-level, *technology-independent*, terms

7.2.3 General Approach (“MRAPT”)

As reported in [34], the basic idea is to apply the analytical capabilities of Model-Checker technology to the functional verification of a physical DUT implemented on some external platform such as an FPGA. In fact, the DUT could be implemented with any desired digital technology. The approach bypasses the HDL Simulator, but requires a Communication Link between Model-Checker and DUT, and a specification of the Correctness Criteria.

Compared to HDL simulation, this approach’s potential advantages include :

1. as with JTAG technology, the physical implementation itself is tested, not just an HDL model
2. tests execute at hardware speed, which is typically several orders of magnitude faster than HDL simulation

3. time-consuming construction of an *ad hoc* HDL testbench is largely eliminated by using an existing Model-Checker. In particular, the steps of (1) defining test sequences, (2) applying the test sequences (stimuli), (3) recording test results (responses), and (4) analyzing the results for correctness, are all automatically carried out by the Model-Checker. The user need not implement any of these steps. The user, however, must specify an Abstract Reference Implementation to define correct behavior.

4. specifications are expressed in the high-level native language of the Model-Checker. This includes the Correctness Criteria, i.e., Abstract Reference Implementation.

5. Model-Checker behavior is common and well-understood. Several Model-Checker software packages are freely available and extensible.

Technical challenges include (1) Link Latency and Bandwidth limitations and (2) Memory limitations.

The proposed approach will be called “Model-Referenced Automated Physical Testing", or just *MRAPT* .

7.3 Theoretical Framework

This section follows the development of the Temporal Logic of Actions (TLA), the version of Linear Temporal Logic (LTL) developed by L. Lamport. This Logic is supported by a collection of software tools, the TLA+ Tools.

7.3.1 Rationale for TLA+ Tools

Lamport’s TLA+ is a formal language that extends the Temporal Logic of Actions (TLA) with standard mathematics : First-Order Logic with Equality, Set Theory, Numbers, and some Linear Temporal Logic (LTL). It has a powerful definitional capability, including recursion, resembling the λ-calculus. TLC is a program that evaluates, or verifies, formulas expressed in TLA+ ; in particular, it has a Model-Checking capability. These TLA+ tools are used because

1. they are existing, portable, Java programs and are well-documented.

2. TLA+ syntax closely resembles standard mathematical notation; it is thus familiar and easy to use.

3. since TLA+ can express most of standard mathematics, it is far more general and concise than the specialized languages found in other Model-Checkers.

4. TLC is easily extensible via an over-ride mechanism whereby a TLA+ Module can be replaced by an arbitrary Java implementation *without* recompiling TLC source code.

7.3.2 State-Transition Systems and Model-Checking

A State Transition System (STS) $\mathscr{T}$ includes a set of States S, a set of Initial States $I \subseteq S$, and a set of Action Identifiers A. A binary predicate, or relation, $Act(s,t)$ on S is termed an Action; for each $a \in A$ there is an Action $Act_a(s,t)$. The set I is specified by its characteristic predicate $Init_{\mathscr{T}}(s)$.

In what follows, the sets S and A are assumed to be finite.

The Action $Next_{\mathscr{T}}(s,t)$ is defined as the disjunction over A of the Act_a : $Next_{\mathscr{T}}(s,t) \triangleq \bigvee_{a \in A} Act_a(s,t)$. If $Act_a(s,t)$ is an Action, then $[Act_a(s,t)]_t$ denotes the Action $Act_a(s,t) \vee (t = s)$.

A Behavior σ over S is an infinite sequence $\sigma = \langle \sigma_0, \sigma_1, \ldots \rangle$ of states $\sigma_i \in S$, i.e., $\sigma \in S^\infty$. In LTL, properties of a State Transition System $\mathscr{T}$ are specified by Temporal Formulas involving Behaviors and Temporal Operators such as $\Box$ (always) and $\Diamond$ (eventually).

The Behavior Specification of an STS $\mathscr{T}$ is the Temporal Formula $Spec_{\mathscr{T}} \triangleq Init_{\mathscr{T}} \wedge \Box[Next_{\mathscr{T}}]_v$. Then, for example, stating that a Behavior σ satisfies its specification in $\mathscr{T}$ is expressed by asserting $\sigma \models Spec_{\mathscr{T}}$. Stating that, if a Behavior σ satisfies the formula $Spec_{\mathscr{T}}$, then the State predicate P is always satisfied, i.e., is true in every state, is expressed as $\sigma \models Spec_{\mathscr{T}} \Rightarrow \Box P$.

A Model-Checker is an algorithm, or program, which, for given STS $\mathscr{T}$ and Temporal Formula F, computes the predicate $\forall \sigma \in S^\infty : \sigma \models Spec_{\mathscr{T}} \Rightarrow F$. This formula will be abbreviated $\models_{\mathscr{T}} F$, and states that $\mathscr{T}$ is a Model of F.

7.3.3 Invariance and Refinement Properties

The following defines the two properties used in the sequel.

7.3.3.1 Invariance

An Invariant of $\mathscr{T}$ is a State predicate Inv such that $\models_{\mathscr{T}} \Box Inv$.

7.3.3.2 Refinement

Refinement provides a way to compare two STS's. It will be used here to verify conformance, i.e., that a concrete implementation conforms to an abstract implementation known to be correct. The following is a simplified description.

Let $\mathscr{X},\mathscr{Y}$ be two STS's with behavior specifications $Spec_{\mathscr{X}}, Spec_{\mathscr{Y}}$ respectively. Let ψ be a map $\psi : S_Y \to S_X$; ψ is extended to the map $\psi : S_Y^\infty \to S_X^\infty$ where $\psi(\sigma) \triangleq \langle \psi(\sigma_0), \psi(\sigma_1), \ldots \rangle$. Then system $\mathscr{Y}$ Refines, or is a Refinement of, $\mathscr{X}$ with respect to the Refinement Map ψ iff $\sigma \models Spec_Y \Rightarrow \psi(\sigma) \models Spec_X$, noted $\mathscr{Y} \trianglerighteq_\psi \mathscr{X}$. This will be the case, for example, if $Init_Y(y) \Rightarrow Init_X(\psi(y))$ and $Next_Y(y, y') \Rightarrow Next_X(\psi(y), \psi(y'))$.

7.3.4 Reachability Analysis

Computing $\models_{\mathscr{T}} F$ for an arbitrary formula F requires the State-Transition Graph T, and T is usually not known in its entirety. However, it may be sufficient to compute only those states that are *reachable* from I. This is Reachability Analysis, and Model-Checkers typically incorporate some form of Reachability Analyzer.

Given $a \in A, s \in S$, the a-successors of s is the set $succ(a,s) \triangleq \{t \in S \mid Act_a(s,t)\}$. Action a is said to be Deterministic iff $succ(a,s)$ is either empty or a singleton. The successors of a state $s \in S$ is the union of all its a-successors : $succ(s) \triangleq \bigcup_{a\in A} succ(a,s)$. The Successors of a subset $X \subseteq S$ is the set $Succ(X) \triangleq \bigcup_{s\in X} succ(s) = \bigcup_{s\in X} \bigcup_{a\in A} succ(a,s)$. $Succ$ is thus a function $Succ : \wp(S) \to \wp(S)$, where $\wp(S)$ is the powerset of S.

Let $R(X) \triangleq X \cup Succ(X)$, for any subset $X \subseteq S$. $R(X)$ is the set of states reachable in at most one step from X. R is a continuous function on the complete lattice $\wp(S)$. By the Tarski-Knaster Theorem, $R(X)$ has a least fixpoint $R^*(X)$, given by $\bigcup_{n=0}^{\infty} R^n(X)$, which is the set of all states reachable from an itial set X. A Reachability Analyzer is any mechanism that computes $R^*(I)$. It is assumed that a Model-Checker program contains such a mechanism. Section 7.4 outlines a hardware implementation.

Reachability analysis requires a DUT V that provides a set of initial states $V!I$ and a function $V!succ(a,s)$. The basic idea here is that these quantities can be provided by a *phy-*

sical DUT external to the Analyzer, given a suitable communication link between Analyzer and DUT. The Analyzer thus verifies an actual implementation, not just an abstract model of it. This implies controllability and observability of the DUT.

7.3.5 Verifying a Processor's Instruction Set

A Processor consists of a State Space S, an Initial State $s_0 \in S$, and an Instruction Set $\mathscr{I} = \bigcup_{i=1}^{N} T_i \times A_i$, where the T_i are disjoint sets, called Instruction Types, of Instruction Names. The A_i are Argument Sets. An element of $\mathscr{I}$ will be noted f_a, where for some i, $f \in T_i$ and $a \in A_i$; the set A_i associated with T_i will be noted $Args(T_i)$. Since the T_i are disjoint, a name f uniquely determines its Type $typ(f) = T_i$ and hence Argument Set $args(f) = Args(typ(f))$. Typically, S has components giving the state of General-Purpose Registers, RAM, Status Flags, Program Counter, Stack, etc.

The Interpretation, or Semantics, $\mu(f_a)$ of an instruction f_a is a function $\mu(f_a) : S \to S$ which gives the processor state resulting from execution of f_a in state $s \in S$. The semantics $\mu(f_a)(s)$ is defined by the processor's Instruction Set Architecture (ISA). For example, the semantics of $ADD_I_{(1,42)}$ might be defined as the function which adds 42 to register 1, sets Carry and Zero flags appropriately, and increments a Program Counter. The execution function $exec$ is defined by $exec(f, a, s) \triangleq \mu(f_a)(s)$.

Let $Ref = (S, s_0, \mathscr{I}, exec)$ be the processor defined by an ISA as above ; this is the Reference Implementation. Let $Imp = (S, s_0, \mathscr{I}, exec)$ be any other processor over the same Instruction Set $\mathscr{I}$, and let $\psi : Imp!S \to Ref!S$ be a refinement map. Then Imp will be a Correct Implementation of Ref iff, for every s reachable from $Imp!s_0$, it is true that $\forall f_a \in \mathscr{I} : \psi(Imp!\ exec(f, a, s)) = Ref!\ exec(f, a, \psi(s))$.

In general, an instruction set $\mathscr{I}$ will be huge and impossible to test exhaustively at every reachable state of a processor. To make physical verification feasible, it will be necessary to restrict attention to subsets of $\mathscr{I}$. Given an instruction set $\mathscr{I} = \bigcup_{i=1}^{N} T_i \times A_i$, an ISA Test Set (ITS) Σ is a pair of subsets $\langle T_i', A_i' \rangle$ where $T_i' \subseteq T_i$ and $A_i' \subseteq A_i$ for some i. The projections T_i', A_i' will be noted $ins(\Sigma), args(\Sigma)$ respectively. Then Imp is Correct with respect to an ITS $\Sigma = \langle T', A' \rangle$ iff, at each reachable s, the above formula holds $\forall f \in T' :$ $\forall a \in A'$.

An ISA Test Suite is a sequence $\langle \Sigma_1, \ldots, \Sigma_N \rangle$ of ITS's ; Imp is Correct wrt. an ISA Test

Suite iff $\forall i \in 1..N$, Imp is Correct wrt. [1] Σ_i.

7.4 Testbench Configuration

The test system consists of

1. a Host Computer, which runs the Model-Checker program TLC. TLC operates on a set of TLA+ Modules, collectively called the Model Domain, whose structure is outlined in section 7.4.1. One of the these modules is the Abstract Reference Implementation.

2. a Target Platform, which is a Digilent ATLYS circuit board containing a Xilinx Spartan-6 FPGA on which the DUT is implemented. The board also has a Micron 128MB DDR2 RAM and a Cypress CY68013A USB interface device connected to the FPGA. The Target Platform components are collectively called the Hardware Implementation Domain, whose structure is outlined in section 7.4.2. One of these components is the actual Physical Implementation, or DUT.

3. a Communication Link, which connects the Host Computer to the Target Platform. It is a high-speed USB2.0 link and has software and hardware components on both Host Computer and Target Platform. The Communication Link is highly technology-dependent, and will not be described here.

7.4.1 Main Model Domain Entities (TLA+)

The main Model Domain entities are the following TLA+ modules (see Figure 7.1) :

7.4.1.1 Abstract Reference Implementation (ARI)

this module contains the reference implementation against which other, more concrete, implementations are compared. The ARI is an implementation known to be correct. Another implementation is considered correct iff it Refines this one. The ARI thus expresses the correctness criteria.

[1] wrt. abbreviates “with respect to”

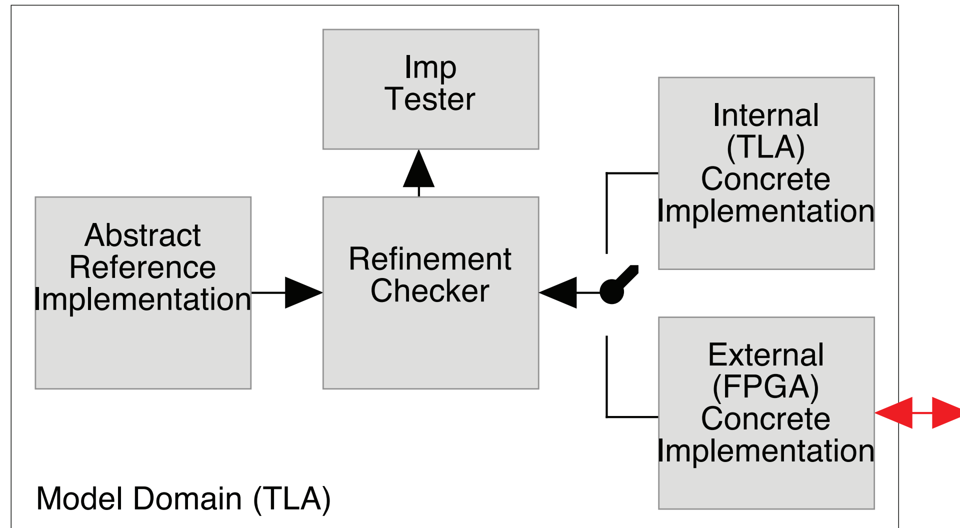


Figure 7.1 – Model Domain Entities

7.4.1.2 Refinement Checker

defines the mechanism for comparing an arbitrary implementation with the ARI. It specifies the behavior of TLC's Reachability Analyzer, including Invariance and Refinement.

7.4.1.3 Internal and External Implementations

the system can contain one or more Implementation modules, which can be Internal or External. An Internal Implementation is a normal one expressed in TLA+. An External Implementation is one which is over-ridden by a Java module that accesses the "real" implementation. In the present configuration, the latter, an FPGA implementation, is accessed via a USB Communication Link.

7.4.1.4 Implementation Tester

this is the top-level module which invokes and interconnects the others.

7.4.2 Main Hardware Implementation Entities (VHDL)

The FPGA circuitry is described in VHDL and consists of (1) the *Target Device* entity, which contains the DUT, and (2) a *Test Manager* entity. They are both implemented on the same FPGA for convenience. See Figure 7.2.

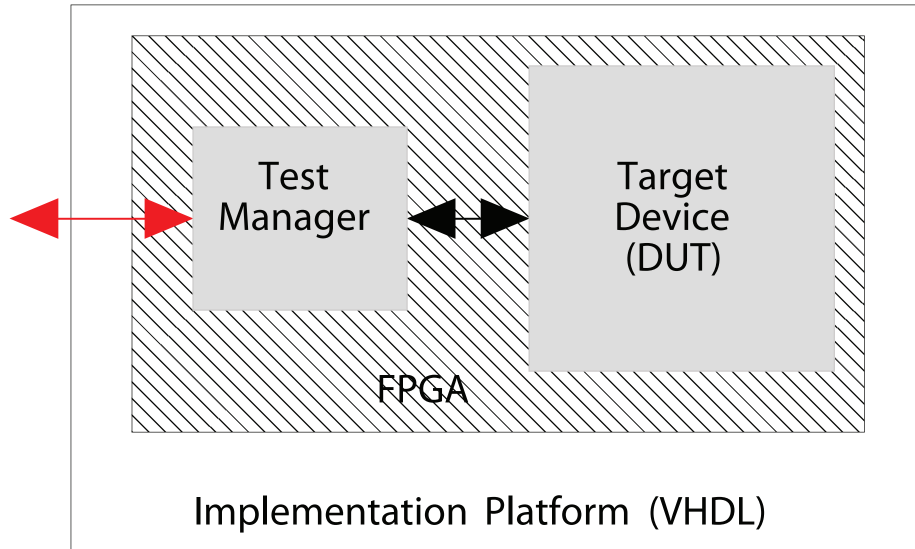


Figure 7.2 – Physical Implementation Domain Entities

The Target Device is a general VHDL wrapper for a DUT implementation, and has the declaration :

```
ENTITY TargetDevice IS  -- Compute succ(a,s)
  PORT (
    ---------------------------
    clock     : IN  STD_LOGIC ;  -- eg, @ 200 MHz
    reset     : IN  STD_LOGIC ;  -- global reset
    test_auto : IN  STD_LOGIC ;
    ---------------------------
    -- Test Access Port (TAP)
    ---------------------------
    start_succ : IN  STD_LOGIC ; -- start pulse
    act_in     : IN  Action ;    -- action a
    state_in   : IN  State  ;    -- current state s
    ------------
    succ_done  : OUT STD_LOGIC ; -- done flag
    act_echo   : OUT Action ;    -- echo of a
    state_out  : OUT State  ;    -- state succ(a,s)
    act_enb    : OUT STD_LOGIC   -- 1 iff Enabled(a,s)
    ---------------------------
  ) ;
END TargetDevice ;
```

The Test Manager is the interface between Target Device, Communication Link, and off-chip RAM. At its core is a Hardware Reachability Analyzer (HRA), or state-graph expander, called Grex, which anticipates the information required by the Model-Checker. It

performs a high-speed reachability expansion of the DUT and returns accumulated results in a limited number of Link transactions. It can also perform Invariance and Refinement tests "on-the-fly". Given enough RAM, Grex will compute the entire state-transition graph.

Grex is described more fully in [33]. Briefly, it consists of a number of Grex Operators which communicate with external RAM via a multiplexed Wishbone [99] bus WB. A Wishbone interconnect allows Master devices to read and write Slave devices via DMA-like packet exchanges. RAM address space is divided into

1. the state-expansion queue XQ,
2. a state hash table HT,
3. the output transition table TT

Finally, operations are sequenced by the GrexControl entity, implemented by a Finite-State Machine (FSM). It drives the expansion at each reachable state through the list of Actions specified in an ActionTable.

7.5 The PicoBlaze Micro-Controller

The PicoBlaze MCU is described in detail in the Xilinx PicoBlaze User Guide UG129 [25]. A brief description follows.

7.5.1 State Space

PicoBlaze is an 8-bit MCU with 16 General-Purpose Registers, 64 RAM Locations, 5 Status Flags, a 10-bit Program Counter, a 5-bit Stack Pointer, and a 31-deep Call Stack. The Status Flags are 1-bit values including Carry Bit, Zero Indicator, Interrupt Enable, and the two Carry, Zero bits which are saved on interrupt. A PicoBlaze MCU can address up to 256 Input Ports and up to 256 Output Ports.

7.5.2 Instruction Set

To describe the Instruction Set, let $RegId$ be the set of register identifiers, $Datum$ be the set of 8-bit data, $PgmAddr$ be the set of program addresses, and $Bit \triangleq \{0, 1\}$. PicoBlaze has 6 different Instruction Types T with argument sets $Args(T)$ as follows :

1. Register-Immediate (RegImm) : Args = $RegId \times Datum$

2. Register-Register (RegReg) : Args = $RegId \times RegId$

3. Shift-Rotate (Shift) : Args = $RegId$

4. Branch (Branch) : Args = $PgmAddr$

5. InterruptEnb (OneBit) : Args = Bit

6. No-Argument (NoArg) : Args = $\emptyset$

Each Type T is defined below by the set of instructions it contains. The names are slightly modified from those given in [25] by the addition of suffixes and a uniform syntax.

```
RegImm == {ADD_I, ADDCY_I, SUB_I, SUBCY_I,
           LOAD_I, AND_I, OR_I, XOR_I,
           FETCH_DIR, STORE_DIR,
           INPUT_DIR, OUTPUT_DIR}
RegReg == {ADD_R, ADDCY_R, SUB_R, SUBCY_R,
           LOAD_R, AND_R, OR_R, XOR_R,
           FETCH_IND, STORE_IND,
           INPUT_IND, OUTPUT_IND}
Shift  == {RL, SL0, SL1, SLA, SLX,
           RR, SR0, SR1, SRA, SRX }
Branch == {JUMP_U, JUMP_C, JUMP_NC, JUMP_Z, JUMP_NZ,
           CALL_U, CALL_C, JUMP_NC, CALL_Z, CALL_NZ}
OneBit == {SETIENB, RETURNI}
NoArg  == {RETURN_U, RETURN_C, RETURN_NC,
           RETURN_Z, RETURN_NZ, RESET, INTERRUPT}
```

7.5.3 Instruction Semantics

Definitions of the instructions, the ISA, are given in [25], using a combination of natural language and pseudo-code. A mathematical version, using TLA+, is outlined in Section 7.6 below.

7.6 Mathematical Reference Model (TLA+)

This section presents TLA+ fragments to illustrate how the PicoBlaze Reference Model is expressed. It is necessarily brief and incomplete ; it is included to convey the definitional style of TLA+ and to suggest plausibility of the model. Letting `..x..` denote an expression involving the variable `x`, some TLA+ syntax conventions are :

- Operators are defined with the syntax `op(x,y,..) == ..x..y..`
- `[A -> B]` is the set of all functions from set `A` to set `B`
- `m..n` denotes the set of integers between `m` and `n`
- `[i \in X|-> ..i..]` denotes the function defined on domain `X` whose value at `i` is given by `..i..`
- `g == [f EXCEPT ![y] = a]` is the function defined by `g[x]`=IF x=y THEN a ELSE f[x].

General definitions include :

```
Array(L,T) == [0..(L-1) -> T]
InitArray(L,val) == [i \in 0..(L-1)|-> val]
Bit   == {0,1}
BitVector(L) == Array(L,Bit)
Byte  == BitVector(8)
Zero  == InitArray(8,0)
```

7.6.1 PicoBlaze Instruction Set Architecture (ISA)

To conserve space, it is convenient to define reduced PicoBlaze configurations. A PicoBlaze configuration is therefore parameterized by the constants : 1) Nreg : no. of General-Purpose Registers, 2) Nram : size of RAM, 3) Ninp : number of Input Ports, 4) Nout : number of Output Ports, 5) Nstk : size of Call Stack, and 6) Npgm : size of Program Memory. Given the above constants, the State Space of the abstract model is defined by the set `AbstractPBStateSpace` of records :

```
PgmAddress == 0..(Npgm - 1)
StkAddress == 0..Nstk

AbstractPBStateSpace  == [
  Reg     : Array(Nreg, Byte) ,
  RAM     : Array(Nram, Byte) ,
  InPort  : Array(Ninp, Byte) ,
  OutPort : Array(Nout, Byte) ,
  Stack   : Array(Nstk, PgmAddress) ,
  PC      : PgmAddress ,
  SP      : StkAddress ,
  C       : Bit,
  Z       : Bit,
  IE      : Bit ] (* No saved sC,sZ *)

isTyped(s) == s \in AbstractPBStateSpace
```

The following illustrates some utility operators that are useful in defining the instruction semantics. PB denotes an MCU state in the above state space.

```
incPC(PB)   == [PB EXCEPT !.PC = @ + 1]
clearPC(PB) == [PB EXCEPT !.PC = 0]
setReg(sX,byte,PB) ==
  incPC([PB EXCEPT !.Reg[sX] = byte ])
```

Instruction semantics are then defined by constructs like :

```
LOAD_I(sX,kk,PB)  == setReg(sX, kk, PB)
LOAD_R(sX,sY,PB)  == setReg(sX, PB.Reg[sY], PB)
```

7.6.2 Abstract Reference Implementation (ARI)

The ARI defines the composite function $exec(f, a, s)$ in two steps. The first dispatches $exec$ to a type-specific execution function appropriate for the instruction arguments :

```
exec(f,a,s) ==
  CASE f \in RegImm   -> execRegImm(f, a[1], a[2], s)
    [] f \in RegReg   -> execRegReg(f, a[1], a[2], s)
    [] f \in Shift    -> execShift (f, a[1], s)
    ...
```

The second step dispatches execution to the semantic definition of the specific instruction f, for example :

```
execRegImm(f,rX,byte, s) ==
  CASE f =
    ...
    [] f = LOAD_I_f   -> LOAD_I(rX, byte, s)
    [] f = AND_I_f    -> AND_I (rX, byte, s)
    [] f = ADD_I_f    -> ADD_I (rX, byte, s)
    ...
```

7.6.3 Test Suites

Test Suites are defined in TLA+ using instruction names and numerical immediate data. An example is :

```
TestSuite ==
   LET Regs      == 0..1
       ImmData   == {0,1,128}
       RegImmIns == {"LOAD_I", "COMPARE_I"}
       RegRegIns == {"LOAD_R", "OR_R", "ADD_R", "COMPARE_R" }
       ShiftIns  == {"SL0", "SL1" }
   IN  <<
       (* newITS(T,A) constructs an ITS as defined above*)
       newITS(RegImmIns, <<Regs, ImmData>> ),
       newITS(RegRegIns, <<Regs, Regs>>    ),
       newITS(ShiftIns , <<Regs>>          ) >>
```

7.6.4 The Refinement Checker

(This sub-section assumes a bit more knowledge of TLA+.)
The Refinement correctness criterion is expressed by the formula :

```
Refines(f,args,s,f_s) ==
  LET psi_f_s == psi(f_s)
      f_psi_s == REF!clearPC(REF!exec(f,args, psi(s)))
  IN  f_psi_s = psi_f_s
```

The only system Action is :

```
verify(f,args,s) ==
  LET f_s == clearPC(exec(f,args,s))
  IN  /\ TLCSet(1,f_s)
      /\ s' = TLCGet(1)
      /\ Refines(f,args,s,TLCGet(1))
```

Behavior is specified by :

```
vars == <<pbsys>>
Init ==
   /\ pbsys = pup_state
   /\ psi(pup_state) = REF!pup_state
Next ==
   \E i \in DOMAIN TestSuite :
   \E f \in TestSuite[i].ins_set :
   \E a \in toArgs(TestSuite[i].arg_seq) :
   verify(f,a,pbsys)

Spec == Init /\ [][Next]_vars
```

7.7 Hardware Implementation (VHDL)

7.7.1 PicoBlaze Top-Level Entity

The top-level PicoBlaze Processor entity designed here differs from the Xilinx version in the following ways :

- it has a Test Access Port for writing and reading its state.
- it does not (yet) use I/O strobes or interrupt signals.
- the design is hierarchical rather than monolithic.

Its declaration in VHDL is :

```
ENTITY Processor IS
  PORT (
    -------------------------
    clock : IN  STD_LOGIC    ;
    reset : IN  STD_LOGIC    ;
    -------------------------
    --  I/O Port ID & Data
    -------------------------
    port_id  : OUT ALUData   ;
    in_port  : IN  ALUData   ;
    out_port : OUT ALUData   ;
    -------------------------
    instr    : IN  Instruction ;
    PC       : OUT PgmAddress  ;
    -------------------------
    -- Test Access Port (TAP)
    --------------------------
    load_st   : IN  STD_LOGIC ;
    procst_in : IN  ProcState ;
    procst_out: OUT ProcState
    --------------------------
  ) ;
END Processor ;
```

7.7.2 PicoBlaze VHDL Design Hierarchy

The above PicoBlaze Processor entity is realized by a hierarchy of VHDL modules :

- IDECODE : instruction decoder
- PBCORE : PicoBlaze core
  - DATAMEMORY : GP registers and scratchpad RAM
  - OPERATORS : data transformation
    - ALU : arithmetic/logic Unit
      - ADDERSUBTRACTER
      - LOGICUNIT
    - SHIFTERN : shifter unit
    - ODDPARITY : parity generator
  - FLAGS : status flags C, Z, IE, sC, and sZ
- PGMCONTROL : program counter and call stack

– CALLSTACK : stack pointer and stack space

7.8 Experiments

7.8.1 Test Suites

The following 3 Test Suites P1,P2,P3 were tested (see table 7.I) :

Suite	Instructions	$\times$	Arguments
P1	{SL0, SL1} {LOAD_I, COMP_I} {ADD_R, COMP_R}		$\{0,1\}$ $\{0,1,128\}$ $\{0,1\} \times \{0,1\}$
P2	{SL0, SL1}		$\{0,1\}$
P3	{SL0, SL1} {ADD_R, COMP_R}		$\{0,1\}$ $\{0,1\} \times \{0,1\}$

Tableau 7.I – 3 Test Suites P1, P2 and P3

7.8.2 Results

Verification of an internal TLA+ implementation yields no errors and the following observations (see table 7.II) :

Test Suite	No. of States Reached	No. of Instr's Tested	TLC + TLA+ Imp Time
P1	198,388	6,348,417	3,055 s
P2	132,093	528,373	214 s
P3	197,118	2,365,417	1,606 s

Tableau 7.II – The 3 Test Suites TLA+ imp. Observations

Verification of the FPGA implementation yields no errors and the following observations (see table 7.III) :

Test Suite	No. of States Reached	No. of Instr's Tested	TLC + FPGA Imp Time	Grex Time
P1	198,388	6,348,417	> 2 h	4.329 s
P2	132,093	528,373	740 s	0.422 s
P3	197,118	2,365,417	3,486 s	1.797 s

Tableau 7.III – The 3 Test Suites FPGA imp. Observations

7.9 Conclusion

The main results are

1. the PicoBlaze design is correct wrt. the Test Suites used.

2. a demonstration that the MRAPT methodology is scalable to the verification of a small microprocessor. Coverage is user-controllable through the definition of Test Suites.

3. the Hardware Reachability Analyzer (HRA) is orders of magnitude faster than the software Model-Checker.

4. since the HRA is so fast, it does *not* accelerate Model-Checker execution ; execution time is overwhelmingly due to the Model-Checker software, especially computation of the refinement map.

Possible avenues for further work include

1. compressing the hardware Transition Table (TT) by BDD's in order to accommodate larger systems and perhaps lead to compatibility with Model-Checkers other than TLC.

2. accelerating the Model-Checker and/or its Java extensions.

CHAPITRE 8

ARTICLE 3

Ouiza Dahmoune and Robert De B. Johnston, **“Model Checker to FPGA Prototype Commmunication Bottleneck Issue”**, © [2011] IEEE. Reprinted, with permission, from [36].

RÉSUMÉ

Le principal problème que nous avons rencontré, quand nous avons appliqué le vérificateur de modèles (Model Checker) TLC pour la vérification d'un prototype FPGA (Field Programmable Gate Array) d'un circuit [34], est lié aux larges délais introduits par la latence du lien de communication. Nous avons effectué des mesures expérimentales sur différentes plates-formes FPGA, et nous avons pu élaboré un modèle ou un ensemble de formules mathématiques pour le lien de communication. Ces dernières suggèrent la concaténation de plusieurs paquets dans un seul transfert pour surmonter le problème de goulot d'étranglement. Pour ce faire, nous avions à anticiper les futurs besoins de TLC's pour les lui obtenir automatiquement via des transferts aussi larges que possible et donc réduire l'effet de la latence du lien. À ce propos nous avons mis en place des structures de mémoires caches logicielles et matérielles pour mémoriser les paquets échangés entre TLC et l'implémentation cible. Nous avions aussi eu à développer des stratégies pour plus de performance en améliorant l'organisation de et l'accessibilité à ces mémoires.

ERAIC (Embedded Reachability Analyzer And Invariant Checker) [33], est une partie essentielle dans notre nouvelle méthodologie de vérification formelle de circuits numériques "Concrets". Quant il est ccombiné avec les mémoires, ERAIC élimine les delais introduits par le lien de communication. Son mécanisme assure une totale controllabilité et observabilité de l'état du circuit, et offre plus de performance, flexibilité, portabilité, en plus de la possibilité de vérification d'invariants sur le circuit sous test (Implementation Under Test : IUT) avant sa soumission au vérificateur de modèle.

8.1 Abstract

The main problem we met, when applying the TLC Model Checker to the verification of a Field Programmable Gate Array (FPGA)-based prototype [34], was the large delay introduced by the latency of the communication link. We have performed actual measurements on different FPGA platforms, and from these measurements we could elaborate a model or a set of mathematical formulas for the communication link. These suggested that we had to combine multiple packets in a single transfer to overcome the bottleneck issue. To do this, we had to anticipate TLC's future needs and obtain them automatically via transfers which are as large as possible and hence reduce the effect of link latency. For this purpose we made

software and hardware memory (RAM) structures to buffer the packets going between TLC and the target implementation. We also had to develop strategies for more performance by improving these memories's organization and accessibility.

An Embedded Reachability Analyzer And Invariant Checker (ERAIC) [33], part of our new methodology for Formal Verification of "Concrete" Digital Circuits, is essential. When combined with the memories, the ERAIC essentially eliminated the communication overhead. The mechanism relies on full state controllability and observability, and offers more performance, flexibility, portability, and furthermore, the possibility of checking invariants on the Implementation Under Test (IUT) before submitting it to the model checker.

8.2 Introduction

According to some experts, in the overall process of circuit development, the verification phase consumes the most of the time. This paper presents a part of a project aiming at reducing that figure in the context of digital hardware design. We are interested on a system's overall behavior, or functionality, without taking into account certain details such as timing and area. By 'verification' we mean comparing the behavior of an implementation with that of a specification, or, a set of requirements.

We are developing a new methodology for Formal Verification of "Concrete" Digital Circuits. We apply Model Checking to an FPGA-based prototype of the circuit. It is an automated, assertion-based verification of physical implementations ; we used Lamport's TLA+ tools which offer powerful mechanisms for high level specification. The work brings some advantages of formal specification and verification closer to engineering practice.

One feature of our approach consists of implementing the verification process in two steps (Figure 8.1) : first, we show that an abstract Reference Implementation satisfies all the Invariance (formal assertions) and Progress properties required in its specifications (requirements), and second, we show that a concrete Implementation is a refinement, as defined by Refinement Criteria, of that Reference Implementation. An implication of the second step is that the expansion process operates on the reference implementation and on the concrete one and this last one cannot produce behaviors not allowed by its requirements specification ; this means that if the implementation produces an error, then the specifications are wrong or incomplete. This addresses the question of how to differentiate between a logical flaw in the specifications and an error in the physical realization.

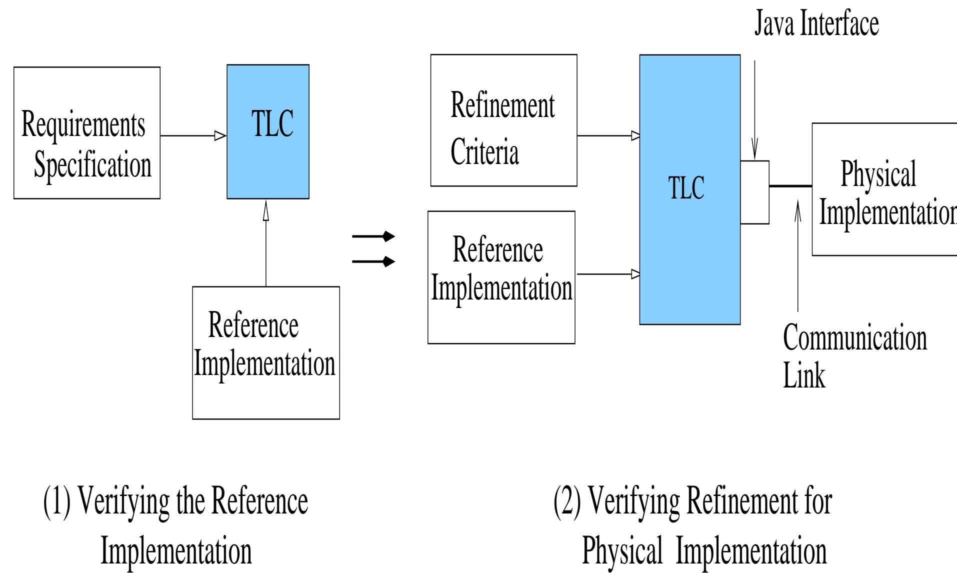


Figure 8.1 – The Verification Process

We established a USB connection between circuit behaviors at a mathematical level of abstraction and the ones in the physical domain [34]. But this was not enough, we had to develop strategies to find a cure for the catastrophic effect of the USB scheduling delays and improve the communication performance. The purpose of this paper is to present these strategies and their performance.

The document outline is as follows. Section 8.3 overviews related work. Section 8.4 explains the process of connecting the TLC Model Checker to an FPGA Prototype, which is a basic step in this methodology. Section 8.5 describes the major problem we met after establishing the connection between the implementation and its mathematical specifications. Section 8.6 presents different ways of organizing and manipulating the data being exchanged between the abstract and the physical worlds. Section 8.7 motivates why a hardware reachability analyzer. Section 8.8 presents the way of bundling multiple tests/results into a single transfer. Section 8.9 presents the preliminary results. We conclude in section 8.10.

8.3 State of the art

The work we are presenting here is in the context of applying model checking to the functional verification of a physical Device Under Test (DUT) implemented on some external platform such as an FPGA [34]. It deals with the communication overhead introduced

by the model checker to FPGA prototype connection. A similar problem is referred to in the context of conventional hardware verification solutions, for example, hardware-assisted acceleration and emulation, and FPGA-based prototype boards for hardware emulation, co-emulation, or co-simulation. We found very interesting articles [11–13, 80] helping to distinguish between these solutions. FPGA-based prototype boards constitute an alternative to hardware accelerator and emulator limitations. They offer high performance, are inexpensive and thus accessible to all, and are more extensible.

Note that the above conventional hardware-assisted verification solutions are directly related to software simulation ; the design under test and a part of the test-bench is accelerated or emulated. In other words, a part of the simulation model is implemented in hardware to get more performance and alleviate the simulation slowness. As we can expect, the communication overhead is considered from the simulation perspective ; the attempt is to reduce the amount of data being exchanged between the software and the hardware by moving the maximum of the communicating parts to the hardware [7, 68, 80–83, 110].

"FPGA-based prototype boards are not usually considered to be a viable alternative because they lack the capability to link with a workstation and they don't provide a sufficient level of visibility into the design required for debugging." said Howard Mao in [80]. Our work shows that these two problems can be overcome.

The aim of our methodology is overtaking the fundamental limitations of hardware-assisted simulation, on one hand, by offering an exhaustive space traversal and considering all possible inputs at every state, and, on an other hand, by assuring that every action enabled in any reachable state of the implementation is also enabled in the corresponding state of the reference, and that the successors correspond.

Note that an exhaustive space traversal assumes combination of different techniques to conserve space. We can consider modularising the system, reducing the state's memory space by testing for example a configuration with 4 bits registers instead of 8 bits ones and finally consider testing in a list of successive passes ; for each pass reduce the reachable system state space by testing a subset of actions. An other important point to consider is the complete state space for each action.

8.4 Connecting TLC to an FPGA Prototype

The Figure 8.2 illustrates the chain of components constituting a communication link. Concretely, it consists of a list of connected elements from the Model side, in the software part, to the target physical device, in the hardware part. We consider the refinement concept as a part of a pre-connection mechanism as it translates forward into (versus backward from) between the concrete world and the abstract one.

8.4.1 The State Space for Elevator in the abstract world (TLA+)

```
(* Parameters *)

CONSTANTS Ne,   (* Number of floors     *)
          W     (* Word-length in bits  *)

LOCAL INSTANCE Words WITH N <- W
(*WORD   == [1..N -> BIT] & Bit   == {0,1}*)

(* Visualisation *)
(*          | W |         | Ne |          | 4  | 3  | 2  | 1  |
            |------------------------------------------------|
STATUS = 0 | 0   ...   0 | 0    ...    0 |LOCK|OPEN|ASCD|STOP|
            |             |               |-BIT|-BIT|-BIT|-BIT|
            |------------------------------------------------|
TOP    = 1 | 0   ...   0 |                                    |
            |------------------------------------------------|
FLOOR  = 2 | 0   ...   0 |                                    |
            |------------------------------------------------|
UPBUTT = 3 | 0   ...   0 |                                    |
            |------------------------------------------------|
DNBUTT = 4 | 0   ...   0 |                                    |
            |------------------------------------------------|
DESTS  = 5 | 0   ...   0 |                                    |
            |------------------------------------------------|
*)
(* Symbolic Constants *)
(* Indices in the 6-WORD state array *)
STATUS == 0
TOP    == 1
FLOOR  == 2
UPBUTT == 3
DNBUTT == 4
DESTS  == 5

REGID  == {STATUS, TOP, FLOOR, UPBUTT, DNBUTT, DESTS}
```

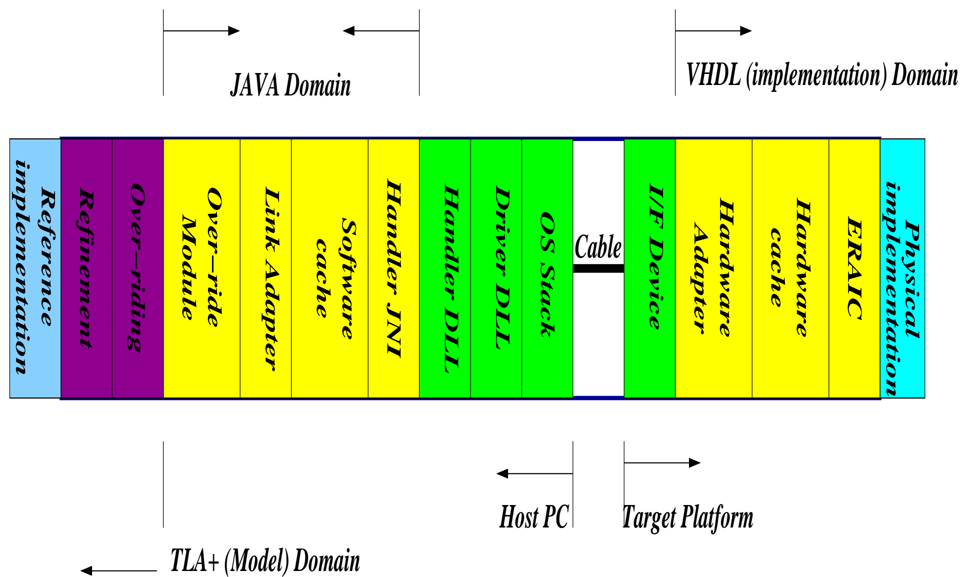


Figure 8.2 – The Communication Link Components

```
(* Bit positions in the STATUS word *)
STOP_BIT == 1
ASCD_BIT == 2
OPEN_BIT == 3
LOCK_BIT == 4

STAT  == {STOP_BIT, ASCD_BIT, OPEN_BIT, LOCK_BIT}
```

Given the above parameters and constants, the State Space of the abstract model is defined by the set `AbstractElevStateSpace` of records :

```
AbstractElevStateSpace == [REGID -> WORD]
```

8.4.2 The State Space for Elevator in the concrete world (VHDL)

This is the corresponding VHDL for ElevatorCircuit1's state :

```
TYPE SystemState IS RECORD
    -- Controller Process:
    stop, ascd, dopen, lock : STD_LOGIC ;
    top, floor, UPbutt, DNbutt, dest :
     STD_LOGIC_VECTOR(7 DOWNTO 0) ;
END RECORD ;
```

The top-level Elevator entity has a Test Access Port for writing and reading its state.

```
TYPE ActionID IS (
    UpDemand, DnDemand, DestChoice, Open, Close,
    Stop, GoUp, GoDown,InitState,NullAct
```

```
  ) ;

ENTITY Elev_Core IS
 PORT (
  action:  IN  ActionID ;      -- Action selector
  -- Action Parameters
  param1:   IN  STD_LOGIC_VECTOR(7 DOWNTO 0) ;
  param2:   IN  STD_LOGIC_VECTOR(7 DOWNTO 0) ;
  state:   IN  SystemState ;   -- Current State
  -- Value of selected  Enabled Condition
  isEnb:   OUT STD_LOGIC;
  succ:    OUT SystemState    -- Successor State
  ) ;
END Elev_Core ;
```

An action could be, for example, in the floor indicated by param1, ask to go Down to the floor indicated by param2.

8.4.3 The State Space for Elevator in the connecting world (JAVA)

In the Java connecting interface the elevator state is described as follow ;

```
public class ElevState implements ElevConstants {
   // CONSTANTS
   static final int NOBS = 6 ;
   // Length of byte[] representation.
   // VARIABLES
   byte status  = (byte)0x01 ;
   byte top     = (byte)0x08 ;
   byte floor   = (byte)0x01 ;
   byte DNbutts = (byte)0x00 ;
   byte UPbutts = (byte)0x00 ;
   byte dests   = (byte)0x00 ;

   // CONSTRUCTOR(S)
   public ElevState() {// defaults to initial state.
   } // end constructor
   public ElevState(byte[] array, int offset) {
     status  = array[offset+0] ;
     top     = array[offset+1] ;
     floor   = array[offset+2] ;
     UPbutts = array[offset+3] ;
     DNbutts = array[offset+4] ;
     dests   = array[offset+5] ;
   } // end constructor
   ...}
```

8.5 Communication Bottleneck

The main problem we met after establishing the connection was the delay introduced by the latency of the communication link. We have performed actual measurements on different FPGA platforms (see a few of them in table 8.I). From these measurements we could elaborate the following model to characterize the USB Channel :

We assume that a USB channel is characterized by : (1) a fixed Scheduling Delay T_{sd} (seconds, s), and (2) a Transfer Rate B_r (Bytes per second, Bps). Then the time $T_x(L)$, in seconds, to send or receive a sequence of L Bytes is :

$$T_x(L) = T_{sd} + L/B_r$$

In the following we deal with Packets of size P_s Bytes and with transfers of N_{ppx} Packets concatenated together into a single large transfer of length $N_{ppx} * P_s$ Bytes.

Let the Packet Transfer Time T_p be defined by

$$T_p = P_s/B_r$$

and the Total Time T_t to transfer N_{ppx} packets by

$$T_t = T_x(N_{ppx} * P_s)$$

then,

$$T_t = T_{sd} + (N_{ppx} * P_s)/B_r$$

and the Effective Byte-Rate (EBR = total number of Bytes transferred / total time), after some substitutions :

$$EBR(N_{ppx}) = B_r/(1 + (T_{sd}/T_p)/N_{ppx})$$

Let

$$N_{ppd} = T_{sd}/T_p$$

so we can write

$$EBR(N_{ppx}) = B_r/(1+N_{ppd}/N_{ppx})$$

Let the total time to send and then receive N_{ppx} packets

$$T_t = 2 * T_x(N_{ppx} * P_s)$$

$$T_t = 2 * (T_{sd} + N_{ppx} * T_p)$$

then the average time to send and receive one packet, T_t/N_{ppx}, also known as the Return Trip Time, is

$$RTT(N_{ppx}) = 2 * T_p * (1+N_{ppd}/N_{ppx})$$

Implementation	**Tsd(ms)**	**Br(MBps)**	**Ps**	**Nppx**	**RTT(us)**
SEB3	7.5	0.225	8	25	672
DLP-FPGA	7.5	1.0	12	5K	27
UP3/QuickUSB	0.43	30.8	8	1	861
XEM3001	1.28	25.9	400	1.3K	2.895

Tableau 8.I – The Return-Trip Time (RTT)

In our context, the RTT is the average communication delay to send a single (action,state) pair to the target platform and receive the result ; it is the average communication time to perform 1 test.

These formulas suggested that we had to bundle multiple packets into a single transfer to overcome the bottleneck issue. We had to anticipate TLC's future needs and obtain them automatically via transfers which are as large as possible and hence reduce the effect of link latency. For this purpose we needed, on one hand, software and hardware memory structures to contain the packets going forward and the others coming backward between TLC and the target implementation. On an other hand, we had to develop strategies for more performance by improving the memories's organization and accessibility.

8.6 Software Look-Ahead (Predictive) Caches

In its present configuration, our verification system offers the ability to choose among different ways of organizing and manipulating the data being exchanged between the abs-

tract and the physical worlds.

8.6.1 The No Cache (NC) model

In this model, there is no cache ; TLC invokes a calculation to every test (a single (state,action) pair). In each state there are many possible actions. Each possible pair is sent in its own packet to the target platform. As we can expect, this has a catastrophic effect. Apart from the fact that it constituted a starting point for our methodology, it was clear that we had to develop alternatives to obtain better performance. We have developed three algorithms to exploit the bandwidth of the communication link. The alternatives consist of bundling multiple packets into a single transfer by using a cache memory.

8.6.2 The Single State Cache (SSC)

This is the one level cache. We use a transition table structure “Transtab” ; it contains, for a single current state being treated by the Model Checker (MC), all the possible transitions. Each time the MC submits a state, we check if it is the current state for which the possible transitions are already in Transtab. In this case the result is directly returned from Transtab. Otherwise, the newly invoked state becomes the current state for which a packet is constructed to be sent the target platform. This one will return, for each action, a successor state with which Transtab is updated.

8.6.3 The Multi-States Cache (MSC)

To consider as many states as the sending buffer permits, we use 2 separated structures for the treated and the non treated states. The $Cache[DUTState, transits]$ structure is a hash table where, for each state(DUTState) already treated, we construct a transition table containing the (action,successor) pairs. For each cycle, we construct a packet by taking Xmax states from a structure containing the non- treated states called noTreatedQueue. The target platform will return the eventual successors for each state. The non-treated successors are inserted into noTreatedQueue, and the treated corresponding ones pairs (state, transits) are inserted into the structure Cache.

8.6.4 The Multi-Levels Cache (MLC)

This is an other way to consider as many states as the sending buffer allows. We use a second level structure called $TreatedQueue[DUTState, Byte[][]]$. This is a hash table, where each state (DUTState) already treated has its successors table. When the MC submits a state, we check if it is the current state for which the possible transitions are already in Transtab(SSC). In this case, the result is directly returned from Transtab. Otherwise, we see if the newly invoked state is already treated (it is in TreatedQueue), we have just to update Transtab from TreatedQueue (second level). In case the newly invoked state is not already treated, a packet (block) to send to the target (physical circuit) is constructed as follows : we start by putting the newly invoked state, and, to use the maximum capacity of the sending buffer, we add the non-treated successors of the treated states.

The following figures 8.3 and 8.4 illustrate the performances obtained with these different software cache models for two elevator controller prototypes. The second controller has more functionalities than the first one.

8.6.5 The One-Shot Cache (OSC)

To eliminate completely the waiting time by the MC for the target platform, we implemented the One-Shot Cache (OSC). In this case, every state requested by the MC is already in the software cache. The cache will contain all the accessible states. In other words, all states are ready to be treated and at any time the state to expand is either as the current state, for which the possible transitions are already in Transtab, or in the cache structure and all what is needed to be done is just to update Transtab.

Here, the process of anticipating the successor calculations is generalized to the expansion of the whole state graph. In this case, the hardware expansion is not activated by the MC ; it proceeds autonomously and terminates the whole state graph expansion of the hardware circuit before the MC begins the initial state expansion of the reference implementation. We implemented such state expansion in the hardware reachability analyzer, described in [33], independently of the MC.

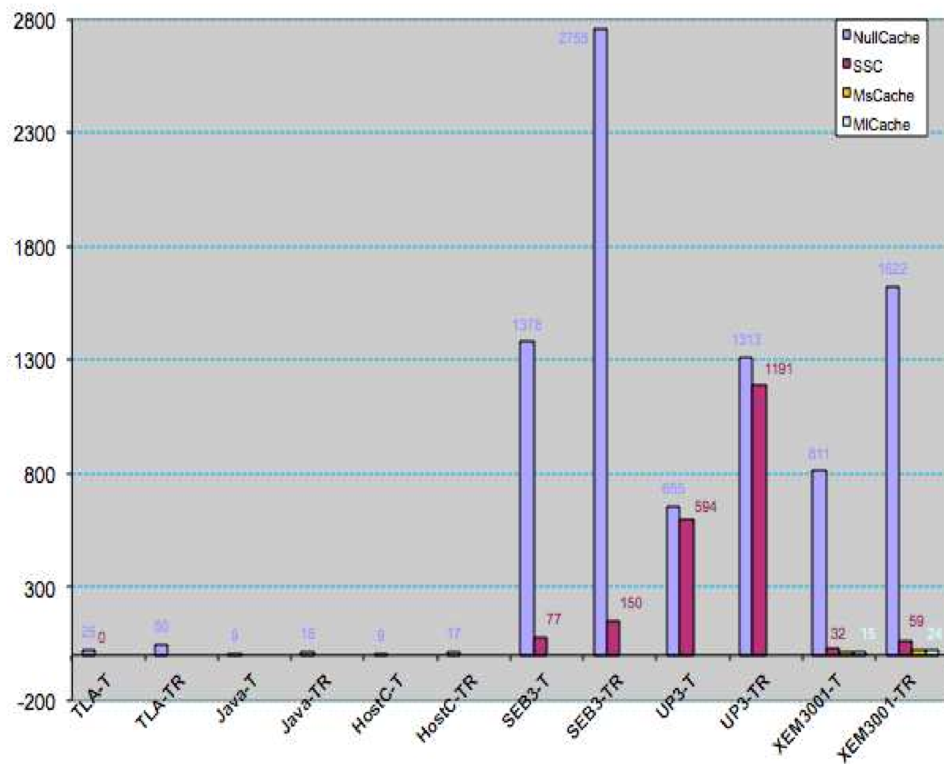


Figure 8.3 – Software Caches Performances for the Elevator Controller One with (8886 states,36977 transitions) and Invariant Type(T) versus Invariants Type & Refinement(TR)

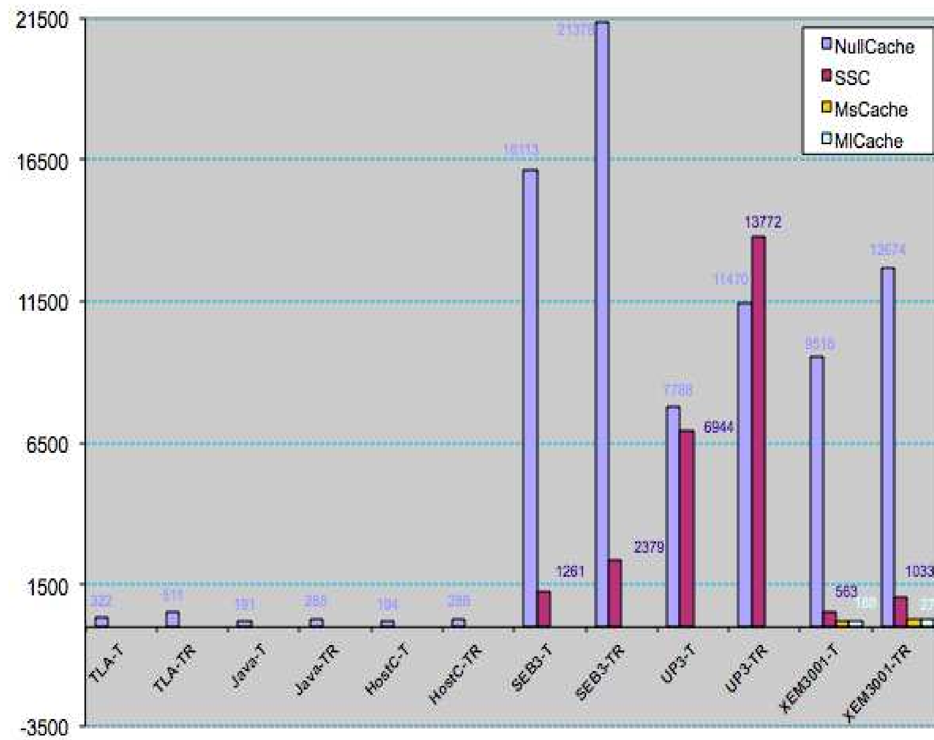


Figure 8.4 – Software Caches Performances for the Elevator Controller Two with (184938 states, 527821 transitions) and Invariant Type (T) versus Invariants Type & Refinement (TR)

8.7 A Hardware Reachability Analyzer

The Hardware Reachability Analyzer is the most important component of the strategies we have developed. It presents the following advantages :

- The hardware expansion process is autonomous, independent of the MC.
- Its function is essentially to generate the state graph expansion, which could be submitted totally to the MC, and/or be used in hardware to have invariants verified "on the fly".
- The hardware part anticipates the overall graph expansion.
- Runs at the hardware speed.
- We aimed at a universal core expander with a Wishbone compatible structure. In this way, it is portable to other implementation platforms.
- It is applicable to any circuit-creating technology.
- It offers the possibility to start the expansion from any state.
- Opens new avenues to consider very complex circuits by exploiting parallelism and hardware/software combinations.
- It has the ability of checking some invariants and then offering the possibility of integrity tests of the IUT before submitting it to the MC.

8.8 Hardware Look-Ahead (Predictive) Caches

Among the strategies we had to develop to overcome the communication bottleneck issue is bundling multiple tests into a single transfer backward and forward. To provide this, from the physical side, we had to use two physical memories, 2 Wishbone Slave Memories. These are structured to contain a hash-table, an expansion queue and a transition table, which are produced by ERAIC ; they are consecutively mapped to the 2 wishbone slave memories via wishbone addresses. As illustrated on Figure 8.5, we used 3 kinds of data structures :

1. An Expansion Queue (XQ) which is a list of linked lists. Each linked list is pointed from an entry in the Hash-table HT ; it contains the expanded and unexpanded states for which the hash function attributes a same value.

2. A Hash-table (HT) : To accelerate the process of state search, we define a hash function to determine, for each state, an index to an entry in HT. All states having the same function code are linked in XQ, where the first one is pointed to by the first field in the HT entry. The second field records the number of states in the linked list ("hash bucket").

3. A Transition Table (TT) which contains, for every expanded state, its set of (enabled action, successor state) pairs.

In the present implementation, these virtual data structures are consecutively mapped to two physical memories via wishbone addresses (see Figure 8.6).

8.9 Some Results

We have performed actual measurements on different FPGA platforms. To overcome very high USB latency times, much of the work involved, on one hand, building predictive cache memories on the FPGA's and on the other hand, implementing the reachability analysis in hardware. In this way, the MC and the IUT don't communicate directly ; the hardware extensions construct the overall graph expansion.

The table 8.II summarizes some verification times(in seconds (s)) for the Elevator Controller application in different configurations considering the implementation (Imp.), the platform, the communication link, the Software (No Cache (NC), Single-State Cache (SSC), Multi-State Cache (MSC), One-Shot Cache (OSC), a Java Custom Reachability Analyzer (JCRA), Multiple Transfers (MT) or Single Transfer (ST)) vs. the Hardware ((SRAM) memory or/and an embedded Hardware Reachability Analyzer (HRA)).

The main result for now is that a Hardware Reachability Analyzer, when in stand-alone mode, is 3 to 4 orders of magnitude faster than an equivalent HDL simulation (8775 times in the above example).

To extend the work to more substantial problems, we implemented and verified an 8-bit microprocessor's physical electronic implementation, the Xilinx PicoBlaze Micro-Controller Unit (MCU)[60]. Experimental verification consists of showing that the physical

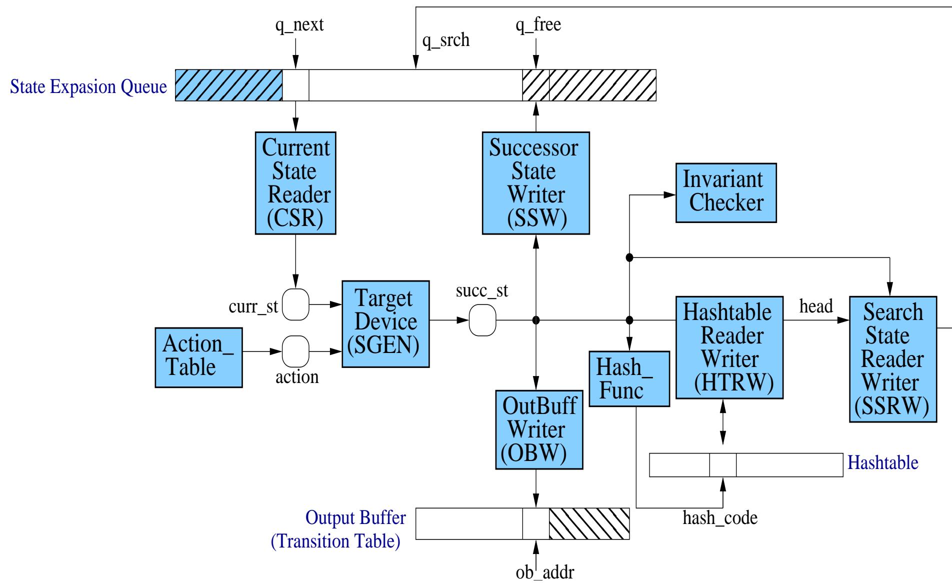


Figure 8.5 – Hardware Reachability Analyzer Data Paths and Operators

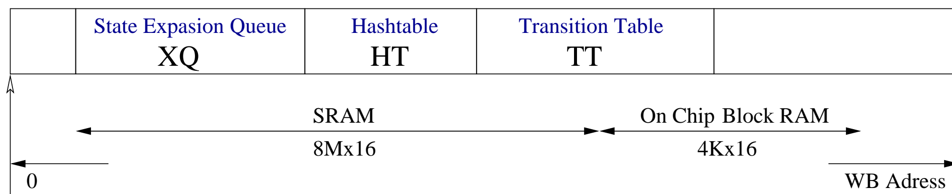


Figure 8.6 – Reachability Analyzer Virtual Memory Mapping

MCU behavior conforms to a mathematical specification of its Instruction Set Architecture (ISA), for selected subsets of instructions and arguments. We considered using a higher-capacity FPGA offering a larger and faster RAM, a higher clock rate and wider bus width. We replaced the linear hash table search algorithm with a binary one. For the Elevator Controller (see table 8.III) we got a factor of 4 times improvement considering the expansion time.

For each state, there is a test for every action, but only actions which are enabled lead

Imp.	**Time(s)**	**Note**
TLAModel	170	TLC + TLA+ Imp.
Simulator	7 020	VHDL Model
XEM3001	10,000	TLC + NC + USB2.0 + FPGA + SRAM
XEM3001	645	TLC + SSC + USB2.0 + FPGA + SRAM
XEM3001	114	TLC + MSC + USB2.0 + FPGA + SRAM
XEM3001	17	JCRA + MT + USB2.0 + FPGA + SRAM
NEXYS2	198	TLC + OSC + USB2.0 + FPGA + SRAM + EHRA
NEXYS2	11	JCRA + ST + USB2.0 + FPGA + SRAM + EHRA
NEXYS2	3.3	FPGA + SRAM + ERAIC + alone with linear search
ATLYS	0.8	FPGA + DDR2 + ERAIC + alone with binary search

Tableau 8.II – Some verification times

Board **Project**	**NEXYS2** 4-floor	**ATLYS** 4-floor	**ATLYS** 5-floor
No. Actions	25	25	28
No. States	184,938	184,938	2,439,937
No. Transitions	526,284	526,284	7,014,688
No. Tests	4,623,450	4,623,450	68,318,208
Execution Time (ms)	3,300	812	17,203

Tableau 8.III – A factor of 4 times improvement

to transitions.

Another future development to consider is to try to improve the MC performance by making it run on an embedded processor or, even better, implementing it in hardware. We could also consider TLC's multi-threading and parallelism concepts, to speed up its computation.

8.10 Conclusion

As far as we know, our work is the first attempt to connect assertional model checking to physical circuit state exploration. Note that the assertions can be expressed and verified on a host computer or the physical circuit. This framework is intended not only to raise the level of abstraction or accelerate the expansion process but also and above all to verify the concrete implementation where the high level specification is directly connected to the physical realization.

To reduce the communication overhead, we implemented software and hardware caches. We aimed also to completely eliminate the overhead by adding ERAIC. In this way, the hardware anticipates, at maximum speed and autonomously, the overall graph expansion. ERAIC offers the possibility to start the expansion from any state, and opens new avenues to consider very complex circuits by exploiting parallelism and hardware/software combinations. Its mechanism does, however, rely on state controllability and observability of the target device.

CHAPITRE 9

ARTICLE 4

Ouiza Dahmoune and Robert De B. Johnston, **“An Embedded Reachability Analyzer And Invariant Checker (ERAIC)”**, © [2010] IEEE. Reprinted, with permission, from [33].

RÉSUMÉ

ERAIC (Embedded Reachability Analyzer And Invariant Checker) est un composant essentiel dans notre nouvelle méthodologie de vérification formelle de circuits numériques "Concrets". Nous appliquons la vérification de modèles (model checking) au prototype FPGA (Field Programmable Gate Array) du circuit [34].

Au noyau d'ERAIC, nous avons le processus d'expansion d'états de l'analyse d'accessibilité opéré en matériel. Il a une structure universelle compatible Wishbone. Son mécanisme assure une totale controllabilité and observabilité de l'état du circuit, et offre plus de performance, flexibilité, portabilité, en plus de la possibilité de vérification d'invariants sur le circuit sous test (Implementation Under Test : IUT) avant sa soumission au vérificateur de modèle.

9.1 Abstract

ERAIC (Embedded Reachability Analyzer And Invariant Checker) is an essential component in our new methodology for Formal Verification of "Concrete" Digital Circuits. We apply Model checking to a Field Programmable Gate Array (FPGA)-based prototype of the circuit [34].

At the core of ERAIC is the process of state expansion of the reachability analysis in Hardware. We aimed at a universal core expander with a Wishbone compatible structure. Its mechanism relies on full state controllability and observability offering more performance, flexibility, portability, and furthermore, the possibility of checking invariants on the Implementation Under Test (IUT) before submitting it to the model checker.

9.2 Introduction

According to some experts, the verification process is the most time-consuming in the overall process of circuit development. This project aims at reducing that figure in the context of digital hardware design. We are interested in a system's overall behavior, or function, without regard to certain implementation details such as timing and area considerations. By 'verification' we mean comparing the behavior of an implementation with that of a specification, or, a set of requirements.

Traditionally, a circuit is at first modeled and verified in its abstract form, and then, physically implemented and tested. The two processes of verification and testing are independent and different. A starting point for the present article is this distinct connection between an "abstract" modelling world and the "physical" implementation world. In this context, we suggest replacing, or augmenting, an HDL simulation phase with a direct connection between the abstract world of modelling and the concrete one of physical implementation. We do not suggest eliminating an HDL simulation for purposes of timing analysis ; we do suggest following a timing analysis by a functional analysis for which an HDL simulator is unnecessary. At a certain degree of maturity of the design we can alternate between Model checking and simulation. The first to detect an error in a particular state and path, the second to help in the correction process.

To our knowledge, our work presents a new approach extending the verification "downwards" to a concrete implementation in silicon. One feature of this approach is a separation of the verification process into two steps (Figure 9.1) : (1) showing that an abstract Reference Implementation satisfies all the Invariance (formal assertions) and Progress properties required in its specifications (requirements), and (2) showing that a concrete Implementation is a refinement, as defined by Refinement Criteria, of that Reference Implementation. An implication of (2) is that a concrete implementation cannot produce behaviors not allowed by its requirements specification ; this means that if the implementation produces an error, then the specifications are wrong or incomplete. This addresses the question of how to differentiate between a logical flaw in the specifications and an error in the physical realization.

We extend formal verification to the overall behavior of post-silicon implementation.

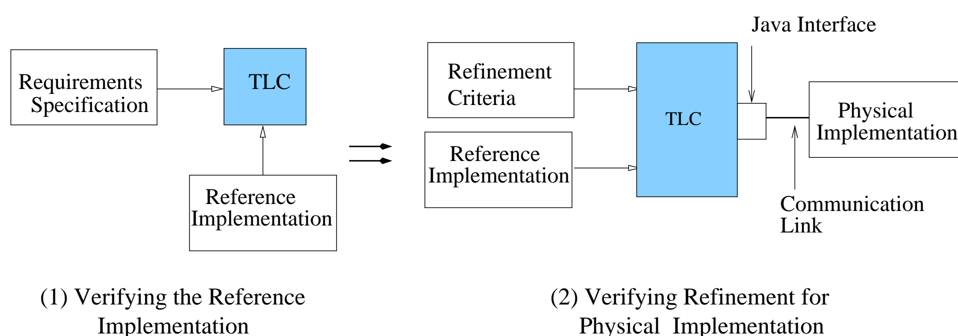


Figure 9.1 – The Verification Process

We use the reachability algorithm of a model checker which computes the set of all states reachable from some initial one(s) and tests a real system(to date, a prototype on Field Programmable Gate Array (FPGA)) by repeatedly invoking functions, and verifying that each new state encountered satisfies an invariant property(formal assertion). In this approach, calculation of these functions is carried out by the target implementation itself, *not an abstract model*. The basic technical requirement was thus to create an interconnection, or a communication link, between the model checker and the IUT. In the context of this paper the communication link is indirect ; an instance of the reachability algorithm is implemented in hardware, it anticipates the overall graph expansion which is submitted in a next step to the model checker.

The document outline is as follows. Section 9.3 overviews related works. Section 9.4 explains why a hardware reachability analysis. Section 9.5 describes the reachability analysis process. Section 9.6 illustrates the graph expander core. Section 9.7 shows where and when ERAIC intervenes in the design flow. Section 9.8 presents how to connect the post-silicon implementation to ERAIC. We conclude in section 9.9.

9.3 State of the art

Traditionally, state space exploration of a design is mostly considered in the context of the abstract world of modelling. In the last decade, hardware formal verification [17, 49, 86, 90, 108] is attracting more attention due to its consistency based on a mathematical proof, high level of abstraction and, most importantly, the potentially exhaustive exploration of the reachable state space. On one hand we have Model checking with temporal logic which offers a completely automated assertional verification but must deal with the problem of state explosion. Much work, like Boolean Satisfiability (SAT) or Binary Decision Diagrams (BDD) techniques and others, has been done to alleviate this problem. On the other hand, we have theorem provers, which are assertional and don't have this problem but are not completely automated. Other works consider combining techniques.

A potential shortcoming of existing tools, for hardware model checking, is that results may not carry forward into post-silicon flows ; The lowest level they consider is synthesizable Register Transfer Level (RTL). We can cite VIS [1] and FormalCheck [14] Model Checkers ; others consider higher levels.

In our case we consider extending pre-silicon verification to post-silicon verification

[34]. We suggest that our methodology represents a new and unique way based on full controllability and visibility without being limited to functional (logical) malfunctions. It worth to underline that our ultimate goal is to realize in hardware a state space traversal of a hardware (post-silicon) circuit.

9.4 Why A Hardware Reachability Analysis ?

The main problem we met after establishing the connection, between the model checker and the IUT, was a delay introduced by the structure of the communication link. To overcome this communication bottleneck issue, we implemented the process of state expansion of the reachability analysis in hardware. It is the best among the strategies we developed ; it presents the following advantages :

- The hardware expansion process is autonomous, independent of the model checker.
- Its function is essentially to generate the graph expansion (see section 9.5) which could be submitted totally to the model checker or be used in hardware to get certain invariants verified on the fly.
- The hardware part anticipates the overall graph expansion.
- Runs at the hardware speed.
- We aimed at a universal core expander with a Wishbone compatible structure. In this way, it is portable to any implementation platform.
- It is applicable to any circuit.
- It offers the possibility to start the expansion from any state.
- Opens new avenues to consider very complex circuits by exploiting parallelism and hardware/software combinations.
- It has the ability of checking some invariants and then offering the possibility of integrity tests of the IUT before submitting it to the model checker.

9.5 The Reachability Analysis Process

The reachability algorithm computes the set of all states reachable from some initial one(s). It is a loop which works by selecting some state s, typically from a queue or stack, to expand, i.e., examine for possible transitions. Successor states s' are computed for each enabled, or "executable", transition. If a successor state has been visited before, it is discarded. Otherwise, it is tested for required invariance properties, and, if OK, it is placed on the queue or stack for future expansion. When all successor states of a given s have been processed, the loop starts over. The process stops when there are no more unexpanded states.

Let $Actions = \{A, B, \ldots\}$ be a finite set of actions, and suppose that each action can have some parameter $x \in X$ associated with it. Here, X is an arbitrary finite set. Then, let $Enb_{A,x}(s)$ denote the predicate, or boolean-valued function, which specifies whether action A, with parameter x, is enabled in state s. Let $Succ_{A,x}(s)$ denote the set of succesor states of s under the action $A(x)$.

To simplify, we consider only deterministic behaviors. A system is termed *deterministic* iff[1], for any choice of A or x, $Succ_{A,x}(s)$ is either empty or a singleton $\{Opn_{A,x}(s)\}$, where $Opn_{A,x}$ is a transition function which maps a state s to its unique successor. In this case, a breadth-first version of the above reachability algorithm could be expressed in pseudo-code as :

```
ModelState s0 = InitState_x();
enter(s0, queue) ;
while (existUnexpandedStates(queue)) {
   ModelState s = nextStateToExpand(queue) ;
   forall A in Actions: forall x in X: {
    if (Enb_A,x(s)) {
           ModelState s' = Opn_A,x(s) ;
           if (!contained(s',queue)) {
             if (satisfiesInvariants(s'))
                enter(s', queue);
             else HALT ;
           } // end if !contained
```

[1] "iff" abbreviates logical equivalence : if and only if

```
        } // end if enabled
    } // end forall
    setExpanded(s, TRUE) ;
} // end while
```

9.6 The Graph Expander (Grex)

At the core of our hardware reachability analyzer is a state transition Graph expander (Grex). It returns the graph expanded in a transition table containing for each state its set of (executable action, successor sate) couples. Grex is universal in the sense that it does not necessarily use manufacturer-specific primitive components such as LUT's (Look Up Tables). An effort has been made to isolate board and manufacturer specifics in the peripherals while leaving the core algorithm specifics-independent. Grex is globally structured on the two following parts (see Figure 9.2) :

1. The Grex operators : Some of them are connected to the Wishbone interconnection network ; they involve block ("DMA") transfers between memory and local registers.

2. The Grex Control : It defines the signals required to sequence the state-transition graph expansion process in the WBGrex system. It uses a standard Moore Finite-State Machine.

In the following, we will give a short description of the other Grex parts as illustrated in Figure 9.2.

9.6.1 The detailed Grex structure

9.6.1.1 A Wishbone Interconnection Network

To offer more flexibility and mainly portability and extensibility, we choose a Wishbone (WB) [98, 99, 106] interconnect network. It supports 7 WB masters and 2 WB slaves. Its 'unibus' architecture is that of a shared (multiplexed) bus which is arbitrated by a round-robin scheduler ; only one master can be granted access to the bus at one time, i.e., the bus is a mutually- exclusive resource. However, all masters may compete independently and simultaneously for access to the bus.

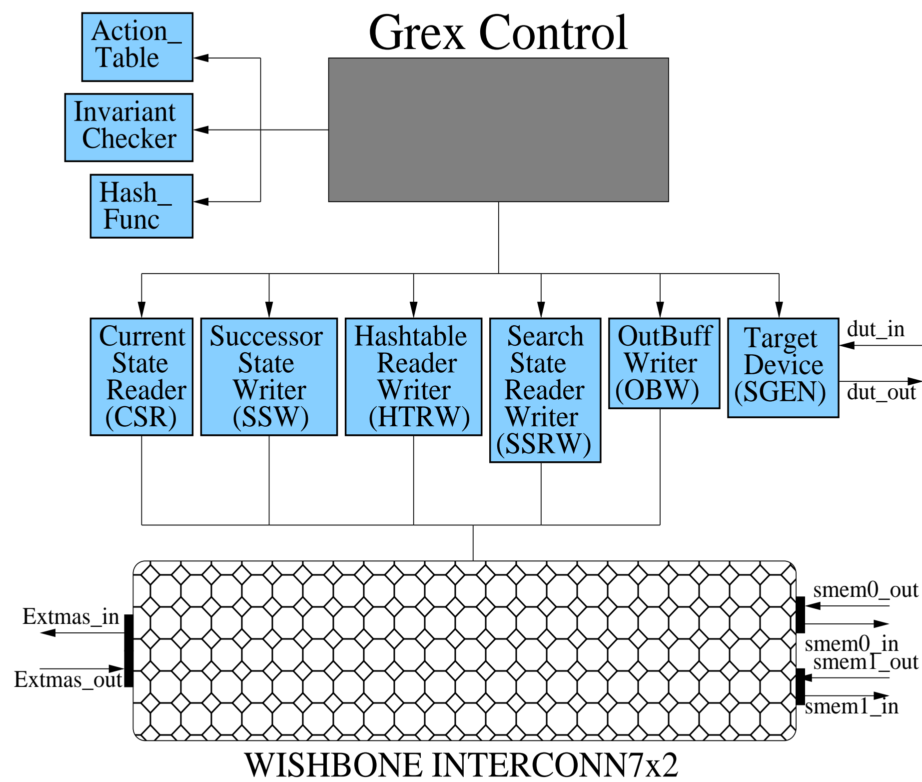


Figure 9.2 – WBGrex Data Paths and Operators

9.6.1.2 2 Wishbone Slave Memory ports

The memory structure and organisation constitutes one part of our solution to the problem of scalability. As it's illustrated by Figure 9.3, we use 3 kind of data structures :

1. A Hash-Table (HT) : To accelerate the process of state search, we define a hash function to determine an index to each state entry in HT. All states having the same entry are linked in XQ and the first one is pointed by the first field in the HT entry. The second field gives the number of states in the linked list.

2. An expansion queue (XQ) which is a list of linked lists. Each linked list is pointed from an entry in HT, it contains the expanded and unexpanded states for which the hash function attributes a same value.

3. A Transition Table (TT) which contains for each expanded state its set of (executed action, successor sate) couples.

In the present implementation, these virtual data structures are consecutively mapped to two physical memories via wishbone addresses (seeFigure 9.4).

9.6.1.3 A Wishbone External Master Port (XMAS)

This I/O port is used by a host computer for initialising and reading the transition table.

9.6.1.4 A Current State Reader (CSR)

The XQ structure is accessed sequentially. A pointer "q_next" (see Figure 9.3)is used to get the block representing the current state to expand. CSR returns a Link State Pair (LSP).

9.6.1.5 A Successor Generator (SGEN)

This is a proxy for the IUT. With the (action, state) structure, it invokes IUT to get the successor state in case the action is enabled.

9.6.1.6 A Successor State Writer (SSW)

The first free space in XQ is pointed by "q_free"(see Figure 9.3). It is used by SSW to insert the newly reached state.

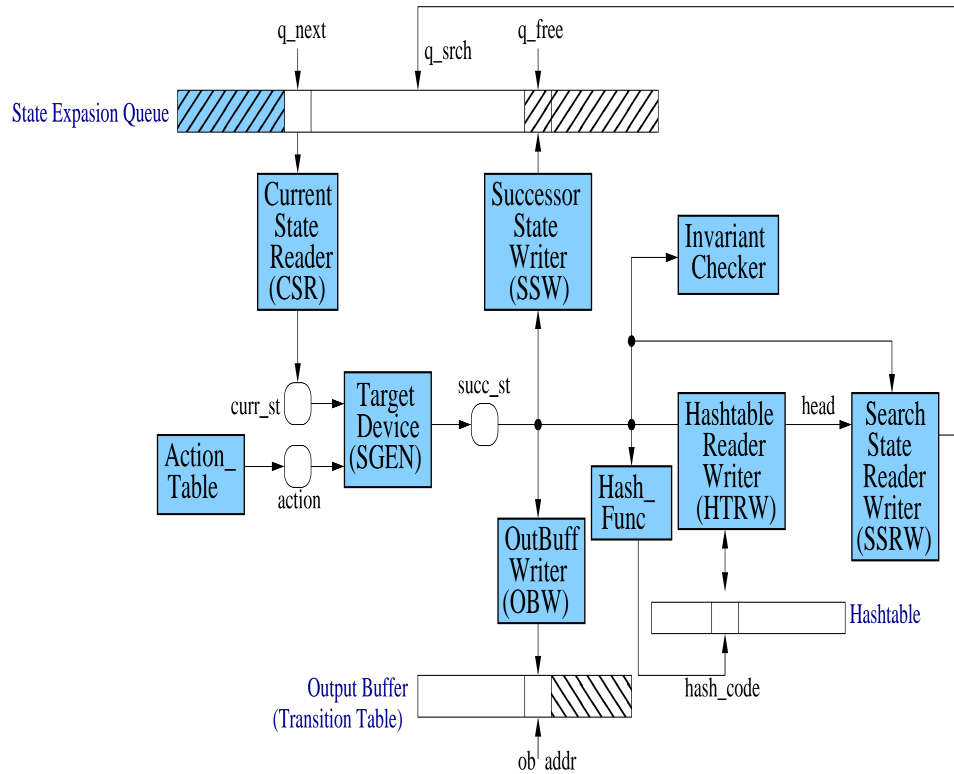


Figure 9.3 – Reachability Analyser Data Paths and Operators

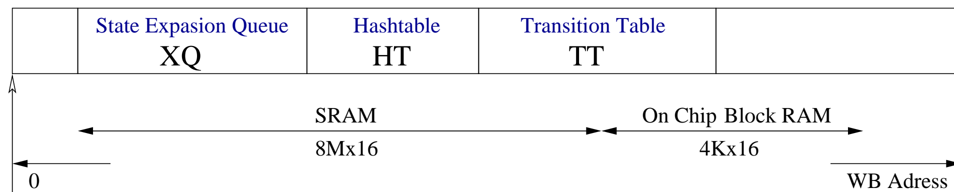


Figure 9.4 – Reachability Analyser Virtual Memory Mapping

9.6.1.7 A Hashtable Reader/Writer (HTRW)

Sets of states in XQ form linked lists in which the first element is pointed by “head" (see Figure 9.3). To return “head", HTRW use “h_code", given by the hash function (see Section 9.6.1.11) for each state. For each newly reached state, HTRW read its entry in the hash table if its linked list does already exist ; otherwise a new entry is inserted. Each entry contains the “head” field and a counter field representing the size of the linked list.

9.6.1.8 A Search State Reader/Writer (SSRW)

To determine if a state is already in XQ, SSRW is invoked to read and compares it successively to its linked list elements. It uses “q_srch" (see Figure 9.3) to get the pointed XQ element. In case it is the first reach to the state, this one should be linked to the last element of the linked list which is than read and written.

9.6.1.9 An Output Buffer Writer (OBW)

Each newly calculated couple (state,action) is inserted in the transition table. This one is submited in a next step to the model checker.

9.6.1.10 An Action Generator

With the action number submitted by the Grex Control part, it returns the action to execute for the current state.

9.6.1.11 A Hash Function

For each state, it calculate an index for its entry in the hash table (see item 1 in section 9.6.1.2).

9.6.1.12 An Invariant Checker

For each reached state we have the ability to check on the fly if it satisfies some invariants expressed in hardware.

9.7 Where and When ERAIC intervenes in the Design Flow

The proposed configuration requires a host computer which contains the abstract model descriptions, the verifier software (the TLC Model Checker) and a Java communication interface, a target platform which contains the actual IUT connected to a wishbone compatible core expander, and a communication link between Host and Target. Note that the abstract descriptions and the concrete implementations could be developed separately by respective specialists. Our methodology suggests putting a mechanism connecting them to make the verifier compare their behaviors.

To precisely situate the ERAIC in this configuration, let's put it as a bold point in this short descriptive summary of the design flow. It consists of :

1. Identify the observable quantities of the desired system ; these are functions (projections) of its state s.

2. Identify the actions which define a system's behavior including any parameters they may require.

3. Express the requirements, i.e., desired properties, in terms of the observables. We'll call the result of these first three steps the "Requirements Specification".

4. Develop an (abstract) reference implementation, and verify that it satisfies the above requirements. The result will be called the "Reference Implementation".

5. Express the conditions an implementation must satisfy in order to be a refinement of the Reference Implementation. These will be called the "Refinement Criteria".

6. Develop a post-silicon(Concrete) Implementation.

7. **Connect the post-silicon implementation to ERAIC**.

8. Put the connection mechanism between the 2 implementations.

9. Verify that the Concrete Implementation refines the Reference Implementation.

9.8 How to connect the post-silicon implementation to ERAIC

To connect the 2 parts, ERAIC integrates a connection mechanism ; all we have to do is to adapt it to the circuit of ones interest, it consist of :

1. Instantiate each of the 2 parts and connect them to allow the ERAIC to submit a state and an action to the IUT which will return the successor state if the action is enabled.

2. We consider the state of the IUT explicitly for property conformance. The (state,action) structure in ERAIC is represented as an array of words ; a translation mechanism is needed to encode and decode states and actions.

9.9 Conclusion

As far as we know, our work is the first attempt to connect assertional model checking to post-silicon hardware state exploration. Note that the assertions can be expressed and verified on a host computer or, for limited types of invariants, on silicon. With ERAIC, the hardware anticipates, at maximum speed and autonomously, the overall graph expansion.

ERAIC offers the possibility to start the expansion from any state, and opens new avenues to consider very complex circuits by exploiting parallelism and hardware/software combinations. Its mechanism relies on full state controllability and observability offering more performance, flexibility, portability, and furthermore, the possibility of checking invariants on the IUT before submitting it to the model checker.

To complete our research we will focus in the future on two different aspects. First, we will attempt to apply our methodology to more complex circuits and than we will try to illustrate how concretely our methodology can be more efficient and more faster than just relying on simulation. We believe that this extend will be in a great help for further development.

Results are very encouraging. At the moment, ERAIC, applied to the elevator controller, completes an exhaustive verification in 3.3 s vs. 7 020 s for an equivalent VHDL simulation.

Appendice I

Un analyseur d’accessibilité en matériel

I.1 Introduction

Les grandes lignes de cette annexe ont déjà fait l’objet de la publication [33]. ERAIC (Embedded Reachability Analyzer And Invariant Checker) constitue un composant essentiel dans notre méthodologie. À son noyau, nous avons le processus d’expansion d’états pour l’analyse d’accessibilité de l’implémentation physique. Notons qu’au début de notre projet nous visions l’utilisation du processus d’expansion d’états du vérificateur TLC. Il s’agissait donc d’invoquer TLC pour la construction des "stimuli"s nécessaires (état, action, paramètres) pour mettre l’implémentation physique dans un état et faire en sorte que celle-ci, stimulée par l’action et les paramètres, retourne l’état successeur éventuel. Avec ERAIC, une copie de ce processus d’expansion est implémentée physiquement. Nous en présenterons dans ce qui suit les détails.

Au noyau de ERAIC, un réseau d’interconnexion dont la structure est compatible Wishbone [99]. Il relie un ensemble d’opérateurs à des mémoires physiques d’où est récupérée le graphe d’expansion. ERAIC utilise un mécanisme de contrôle et d’observation en donnant la possibilité de mettre le circuit dans un état parmi ces états possibles, de lui spécifier une action et un ensemble de paramètres possibles, d’invoquer son exécution et par la suite récupérer et observer son état successeur éventuel, du même coup l’état en question peut être vérifié par rapport a sa possible satisfaction d’invariant définit en matériel. Ce mécanisme permet :

- Plus de performance parce qu’il est le plus proche de la réalisation physique à vérifier
- Plus de flexibilité par le fait qu’il donne la possibilité de démarrer l’expansion à partir de n’importe quel état
- Plus de portabilité du fait qu’il peut être mis en place sur n’importe quelle plate-forme et pour n’importe quel circuit
- La possibilité de vérification d’invariants dans l’implémentation physique avant de de soumettre le graphe d’expansion au vérificateur de modèles

.

ERAIC constitue donc l'implémentation physique de l'algorithme d'analyse d'accessibilité du vérificateur de modèles TLC. Il calcule l'ensemble de tous les états accessibles à partir d'un certain état initial. Il vérifie un système physique (un prototype sur FPGA) en répétant l'invocation de fonctions, et en vérifiant que chaque nouvel état rencontré satisfait un certain invariant.
Soulignons que dans cette approche, le calcul de ces fonctions se fait par l'implémentation physique, et non par *le model abstrait*. Notons toutefois le besoin de se servir d'un outil de haut niveau d'abstraction pour la description d'un certain type d'invariants. Ce qui justifie la nécessité de créer une interconnexion, ou un lien de communication, entre le vérificateur de modèles et l'implémentation physique. Dans le contexte d'analyse d'accessibilité physique, le lien de communication est indirect ; un circuit physique anticipe la totalité du graphe d'expansion qui est utilisé, dans une étape suivante, pour répondre aux invocations du vérificateur de modèles.

Traditionnellement, l'exploration de l'espace des états d'un système est plutôt considérée dans le contexte abstrait de modélisation. Nous avons rencontré un seul travail implémentant un analyseur d'accessibilité de Réseaux de Pétri sur FPGA [29], or un réseau de Pétri ne représente qu'un modèle, de ce fait, le but ainsi que l'approche sont différents.

I.2 L'algorithme d'analyseur d'accessibilité

C'est un algorithme récursif qui, à partir d'un état désigné comme état initial, et à chaque état rencontré durant son exécution, répète l'opération de calcul, de l'état successeur éventuel du système pour chaque valeur donnée du couple (action, liste de paramètres). Il utilise une table $(acttab[NACTS])$ contenant toutes les actions qu'un système peut faire étant dans un de ses états accessibles. Les états à explorer sont insérés au fur et à mesure dans une file $(XQ[])$.

Pour chaque état ainsi exploré, l'algorithme retourne une table dont les entrées sont des paires (action, successeur éventuel ou un indicateur de non activabilité de l'action), la première paire de cette table correspond à celle d'une action nulle qui comme on peut facilement le comprendre permet au système de rester dans le même état et du même coup sert d'indicateur de début d'exploration de l'état. La juxtaposition de l'ensemble des tables ainsi obtenues forme la table de transitions $(TT[])$.

Un état successeur est d'abord vérifié s'il satisfait l'invariant physique. Dans un cas contraire, l'algorithme d'expansion s'arrête et retourne l'état courant du système d'expansion et la trace de l'expansion faite jusqu'à cet état, dans la table de transitions $(TT[])$. Notons que de cette manière nous sommes en mesure de corriger certaines erreurs sans devoir passer par TLC.

Dans une deuxième étape l'algorithme vérifie si l'état successeur a été déjà exploré auquel cas il est tout simplement ignoré. Dans cette étape, l'algorithme effectue une recherche de l'état dans deux structures HT et XQ. En effet, pour accélérer cette recherche, nous utilisons une table de hachage $(HT[])$. Une fonction de hachage détermine, en fonction de la valeur binaire de l'état, l'entrée de l'état dans HT. Un élément de HT pointe une liste chaînée dans XQ et donne la taille de celle-ci ; tous les états évalués par la fonction de hachage à une même valeur seront mis dans une même liste. Notons que quand la fonction de hachage retourne une valeur pour laquelle une entrée n'existe pas encore dans HT, l'algorithme déduit directement que l'état en question a été inexistant auparavant et donc il est a insérer dans les 2 structures HT et XQ.

En dernière étape, si l'état n'a pas été rencontré, un maillon lui correspondant est inséré XQ et son entrée est mise à jour dans HT.

I.2.1 Le Pseudo-Code

```
DUTState s0 := InitState_x() ;
enter(s0, queue) ;
while (moreUnexpandedStates, queue) {
    DUTState s := nextStateToExpand(queue) ;
    forall a in Action , p in Param {
        if (Enabled_a,p(s)) {
            DUTState s' := Successor_a,p(s) ;
            if (satisfiesInvariants(s')) {
                if (!contained(s', queue)) {
                    enter(s', queue) ;
                } // end if !contained
            } else HALT ;
        } // end if enabled
    } // end forall
    markExpanded(s, TRUE) ;
} // end while
```

I.2.2 Une Implémentation Structurée en C

```
/*************************************************************************************
```

```
file: GrexEngine.c
dire: MRAPT/GrexDoc/GrexAlgorithm/StructuredCImp/
*************************************************************************************/
#include "GrexParams.h"
#include "SimpleDevice.h"
/* -------------------------------------------- */
/*          Constants & Types                   */
/* -------------------------------------------- */
/*
typedef struct { ? }  Action ;
Action  RESET    ;    // = { ? } ;
Action  NULL_ACT ;    // = { ? } ;

#define NACTS 10
typedef unsigned int  ActIndex ; // 0..(NACTS-1 )
Action  actab[NACTS] ;   // = { ? } ;

typedef struct { ? }  State  ;
State   NULL_DUT ;
State   InitDUTState ;

typedef struct {
    State    sst ;
    BOOLEAN  enb ;}   DUTSucc ;  */
/* ------------------------------------------- */
/*                  Functions                  */
/* ------------------------------------------- */
/*
DUTSucc  dut(Action a, State s) ;
HashCode hash(State s) ;
BOOLEAN  correct(State s) ;
BOOLEAN  equals(State s1, State s2) ;  */

/************************************************************************************/
/*                               Auxiliary Types                                     */
/************************************************************************************/

typedef struct {
    Pointer ptr  ;
    State   st   ;
} LSP ;
LSP LSP0 = {NULL_PTR, NULL_DUT} ;

typedef struct {
    Pointer head ;
    int     size ;
} HTE ;
HTE HTE0 = {NULL_PTR, 0} ;
```

```
typedef struct {
    Action  act  ;
    State   sst  ;
} ASP ;
ASP ASP0 = {NULL_ACT, NULL_DUT} ;

typedef unsigned int
  PgmLoc ;  // = 0..99

/************************************************************************************/
/*                                  GrexState Type                                  */
/************************************************************************************/

typedef struct {
    PgmLoc    pc         ;
    LSP       XQ[XQ_LEN] ;
    Pointer   xq_next    ;
    Pointer   xq_free    ;
    State     curr_st    ;
    ActIndex  actnum     ;
    Action    act_in     ;
    State     succ_st    ;
    BOOLEAN   act_enb    ;
    BOOLEAN   inv_ok     ;
    HashCode  h_code     ;
    HTE       HT[HT_LEN] ;
    HTE       ht_elt     ;
    BOOLEAN   found      ;
    Pointer   xq_srch    ;
    LSP       tst_elt    ;
    TTIndex   tt_free    ;
    ASP       TT[TT_LEN] ;
} GrexState ;

/************************************************************************************/
/*                       GrexState Predicates  (Boolean-valued Methods)             */
/************************************************************************************/

BOOLEAN queue_empty  (GrexState *gs) { return (gs-> xq_next == gs-> xq_free) ;}

BOOLEAN queue_full   (GrexState *gs) { return (gs-> xq_free == XQ_LEN) ;}

BOOLEAN ttable_full  (GrexState *gs) { return (gs-> tt_free == TT_LEN) ;}

BOOLEAN invar_err    (GrexState *gs) { return (gs-> inv_ok == FALSE) ;}

BOOLEAN stop_cond    (GrexState *gs) {
   return ( queue_empty(gs) ||
            queue_full (gs) ||
```

```
            ttable_full(gs) ||
            invar_err  (gs) ) ;}

BOOLEAN bucket_empty (GrexState *gs) { return (gs-> ht_elt.head == NULL_PTR) ;}

BOOLEAN more_acts    (GrexState *gs) { return (gs-> actnum < NACTS) ;}

BOOLEAN null_act     (GrexState *gs) { return (gs-> act_in == NULL_ACT) ;}

BOOLEAN act_disabled (GrexState *gs) {
   return ( (gs-> act_enb == FALSE)    &&
            (gs-> act_in  != NULL_ACT) ) ;}

BOOLEAN state_match  (GrexState *gs) { return equals(gs-> tst_elt.st, gs-> succ_st) ;}

BOOLEAN endof_list   (GrexState *gs) { return (gs-> tst_elt.ptr == NULL_PTR) ;}

BOOLEAN succ_found   (GrexState *gs) { return gs-> found ;}

/*****************************************************************************************/
/*                       GrexState Operators (State Transformations)                    */
/*****************************************************************************************/
/* Simple Operators */

void initAddresses(GrexState *gs) {
   gs-> xq_next = 1 ;
   gs-> xq_free = 1 ;
   gs-> tt_free = 0 ;
   gs-> actnum  = 0 ;}

void initHashtable(GrexState *gs) {
   int h ;
   for(h = 0; h < HT_LEN; h++) gs-> HT[h] = HTE0 ;}

void resetDUT(GrexState *gs)   { gs-> act_in = RESET ;}

void initActGen(GrexState *gs) { gs-> actnum = 0 ;}

void nextActGen(GrexState *gs) { gs-> actnum = gs->actnum + 1 ;}

void getAction(GrexState *gs)  { gs-> act_in = actab[gs-> actnum] ;}

void getDUTSucc(GrexState *gs) {
   DUTSucc dut_res = dut(gs-> act_in, gs-> curr_st) ;
   gs-> succ_st = dut_res.sst ;
   gs-> act_enb = dut_res.enb ;}

void verifySucc(GrexState *gs) { gs-> inv_ok = correct(gs-> succ_st) ;}
```

```
void computeHash(GrexState *gs) {gs-> h_code = hash(gs-> curr_st) ;}

void getHashEntry(GrexState *gs) {gs-> ht_elt = HT[gs-> h_code) ;}

void updHashEntry(GrexState *gs) {
   HTE hte = gs-> ht_elt ;
   hte.size = hte_size + 1 ;
   if (bucket_empty(gs)) hte.head = gs-> xq_free ;
    gs-> HT[gs-> h_code] = hte ;}

void getCurrState(GrexState *gs) {
   gs-> curr_st = gs-> XQ[gs-> xq_next] ;
   gs-> xq_next = gs-> xq_next + 1 ;}

void putTransit(GrexState *gs) {
   gs-> TT[gs-> tt_free] = {gs-> act_in, gs-> succ_st} ;}

void initSearch(GrexState *gs) {
   gs-> found   = FALSE ;
   gs-> xq_srch = gs-> ht_elt.head ;}

void getTestElt(GrexState *gs) { gs-> tst_elt = gs-> XQ[gs-> xq_srch] ;}

void followLink(GrexState *gs) { gs-> xq_srch = gs-> tst_elt.ptr ;}

void updateLink(GrexState *gs) {
   LSP lsp = gs-> tst_elt ;
   lsp.ptr = gs-> xq_free ;
   gs-> XQ[gs-> xq_srch] = lsp ;}

void putSuccState(GrexState *gs) {
   gs-> XQ[gs-> xq_free] = {NULL_PTR, gs-> succ_st} ;
   gs-> xq_free = gs-> xq_free + 1 ;}

/*-------------------------------------------------------------------------------------*/
/* Composite Operators */

void initGrex(GrexState *gs) {
   initAddresses(gs) ;
   initHashtable(gs) ;
   resetDUT(gs)      ;
   getDUTSucc(gs)    ;
   verifySucc(gs)    ;
   if (!invar_err(gs))  {
      computeHash(gs)   ;
      getHashEntry(gs)  ;
      updHashEntry(gs)  ;
      putSuccState(gs)  ;
   } // end if !invar_err
```

```
} // end initGrex

void doSuccSearch(GrexState *gs) {
/* search_loop */
while (TRUE) {
getTestElt(gs) ;
if (state_match(gs)) {
setFound(gs) ;
break ;
}  else {
if (endof_list(gs)) break ;
else followLink(gs) ;
} // end if match else
} // end while
/* end search_loop */
} // end doSuccSearch

void RunGrex(GrexState *gs) {
   /* initialize */
   initGrex(gs)  ;

   /* main_loop */
   while (!stop_cond(gs)) {
      getCurrState(gs)  ;

      /* action_loop */
      for (initActGen(gs) ;
       more_acts(gs)  ;
   nextActGen(gs) ) {
         getAction(gs)  ;
         getDUTSucc(gs) ;
         if (!act_disabled(gs)) {
    if (ttable_full(gs)) break ;
            putTransit(gs)   ;
            verifySucc(gs)   ;
            if (!invar_err(gs)) {
               computeHash(gs)  ;
               getHashEntry(gs) ;
               if (!bucket_empty(gs)) {
                  initSearch(gs) ;
  doSuccSearch(gs) ; // search_loop
               } // end if !bucket_empty
               if (!succ_found(gs)) {
      updHashEntry(gs) ;
  if (!bucket_empty(gs))
   updateLink(gs) ;
  if (queue_full(gs)) break ;
  putSuccState(gs) ;
               } // end if !succ_found
```

```
        } else break ;  // end if !invar_err
      } // end if !action_disabled
    } // end action_loop

  } // end main_loop

} // end RunGrex
/*****************************************************************************************/
```

I.3 Pourquoi une analyse d'accessibilité physique ?

Le principal problème que nous avons rencontré après interconnexion du vérificateur de modèles à la réalisation physique, était un délai introduit par la structure du lien de communication. Il engendrait un goulot d'étranglement qui faisait que l'attente du vérificateur de modèle était trop grande et donc le temps de vérification trop grand aussi. Pour solution, nous avons implémenté physiquement le processus d'expansion d'état de l'analyse d'accessibilité. Elle constitue la meilleure stratégie parmi celles que nous avons mises en place ; elle présente les avantages suivants :

- Le processus d'expansion matériel est autonome, indépendant du vérificateur de modèles.
- Sa fonction est essentiellement de générer le graphe d'expansion qui peut être soumis dans sa totalité au vérificateur de modèles ou utilisé directement en matériel pour vérifier certains invariants à la volée.
- La partie physique anticipe le graphe d'expansion dans sa totalité.
- S'exécute à la vitesse physique(maximale).
- Le noyau est universel avec une structure Wishbone. Il est portable à n'importe quelle plate-forme.
- Elle est applicable à n'importe quel circuit.
- Elle offre la possibilité d'une expansion à partir de n'importe quel état.
- Offre d'autres alternatives pour considérer des circuits plus complexes en exploitant le parallélisme et la combinaison codesign.

- Par la possibilité de vérifications d'invariants à la volée, elle offre la possibilité de tests d'intégrité du circuit avant sa soumission au vérificateur de modèles.

I.4 L'analyseur d'accessibilité dans le contexte physique

Comme on peut le voir sur la figure I.2, le composant qui réalise l'analyse d'accessibilité est au coeur de la partie physique (implémentée en VHDL) de notre système de vérification. Les modules VHDL sont structurés en une hierarchie au sommet de laquelle nous avons l'entité dite WBGrexTop (Wishbone Grex Top) (voir figure I.1). Son architecture englobe un module, nommé TestManager, et un autre qui est l'unité sous test. Le TestManager regroupe plusieurs fonctionnalités :

- Celle qui s'occupe de l'expansion du graphe de l'unité sous test, elle est incarnée par le module VHDL appelé GraphExpander.

- Celle qui s'occupe de la communication avec la machine hôte, elle est incarnée par le module VHDL appelé DigilentHostIF.

- Le contrôle et la visualisation, via la carte FPGA, sont assurés respectivement par deux modules GrexManControls et GrexQSSDObserver. Pour ce faire, on utilise les boutons, les lumières et les segments d'affichage sur la carte.

- Celle qui s'occupe de la communication avec les blocs mémoires, elle est incarnée par le modules VHDL appelé SpartanWBSMem.

- Celle qui s'occupe de la communication avec la mémoire externe au FPGA, elle est incarnée par le module VHDL appelé Nexys2WBSMem.

I.5 Les différents composants de l'analyseur d'accessibilité physique ?

L'implémentation de notre analyseur d'accessibilité physique a nécessité différentes structures de données (voir Figure I.3), différents opérateurs (voir Figure I.4), une machine à états qui génère différents signaux (voir Figure I.16) pour le contrôle des opérateurs et un réseau d'interconnexion reliant ces derniers aux mémoires physiques.

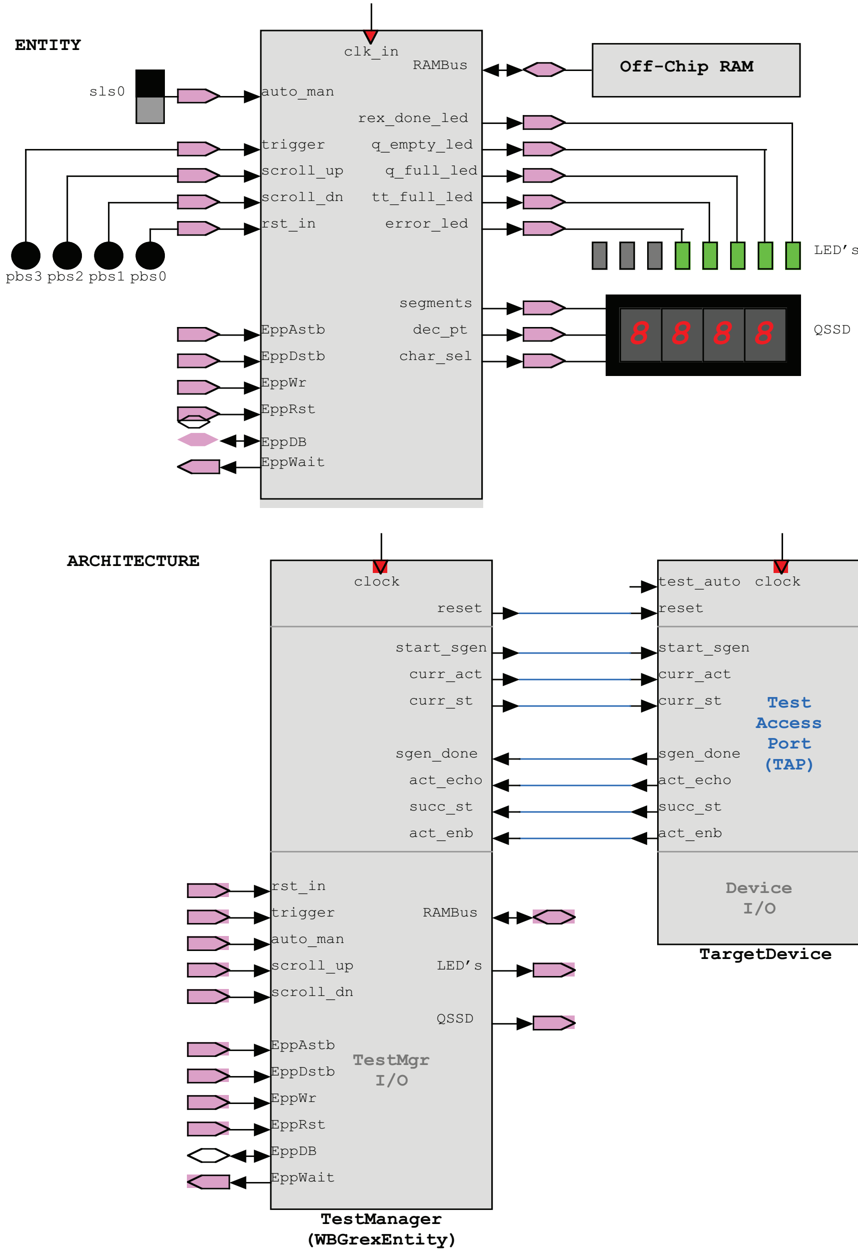


Figure I.1 – WBGrexTop Entity and Architecture

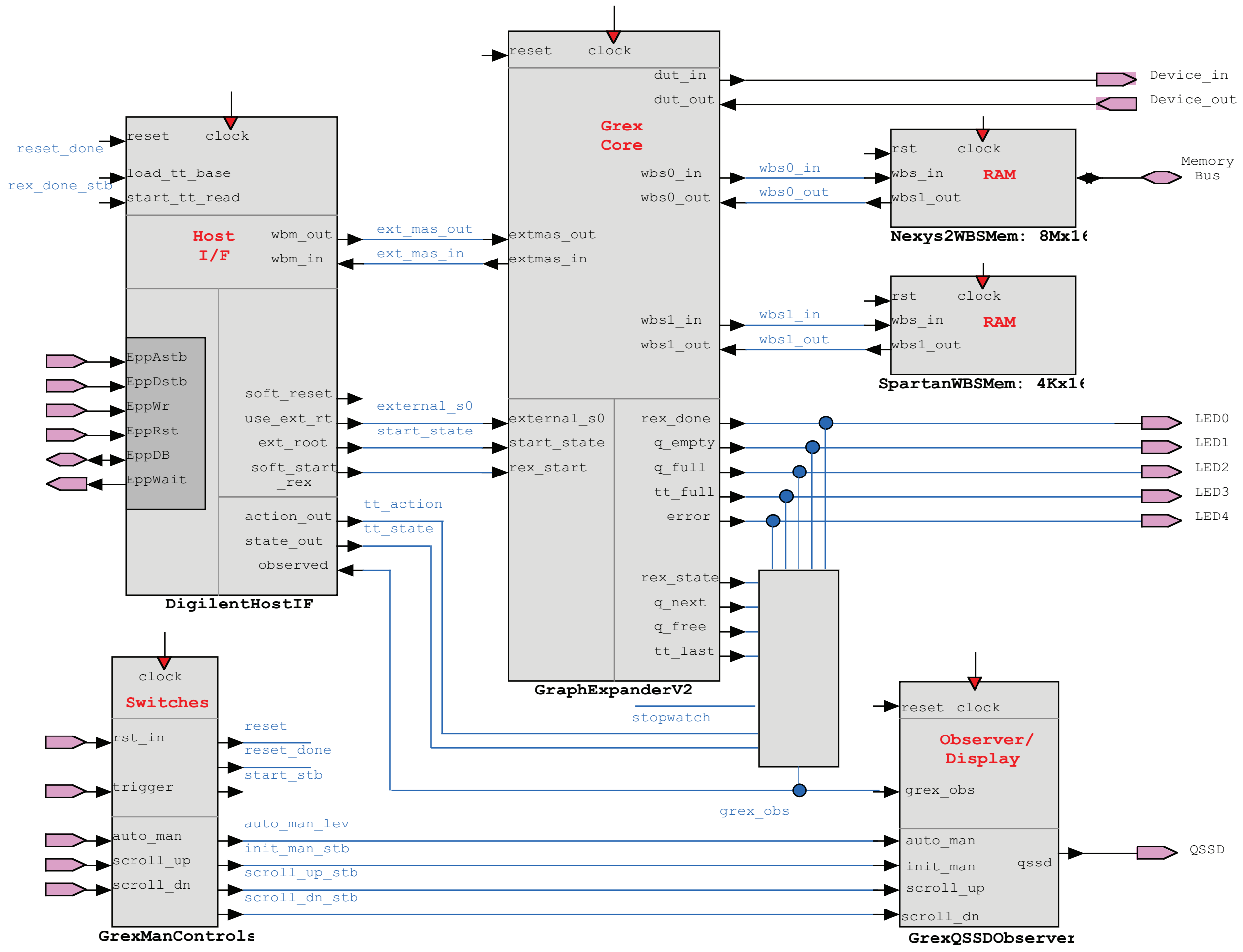


Figure I.2 – TestManager(WBGrexEntity) Architecture

I.5.1 Les structures de données

- Une table d'actions $(acttab[NACTS])$: C'est une table indexée qui contient l'ensemble des actions possibles que le système peut faire étant dans un de l'ensemble de ses états accessibles.

- Une file d'expansion $(XQ[])$, contient l'ensemble des états en attente d'expansion.

- Une table de hachage, contient une entrée pour chacun des états dans la file XQ. Chaque élément de cette table est un maillon à 2 champs, un pointeur vers la file d'expansion et la taille de la liste pointée.

- Une table de transitions $(TT[])$, contient pour chaque état, la table de ses successeurs éventuels.

I.5.2 Les Opérateurs du Graph Expander

Les connexions entre ces différents opérateurs sont illustrées par la Figure I.6.

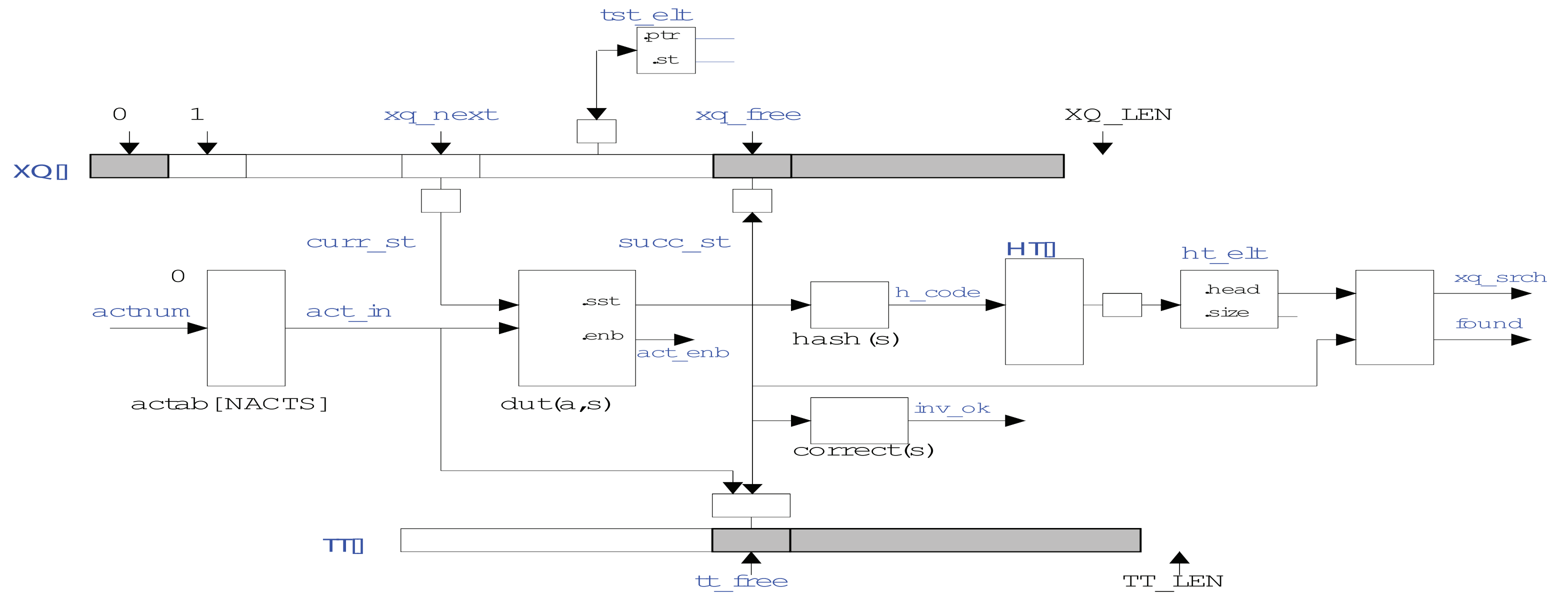
1. Le générateur d'action (Action Generator : ActionGen), détermine l'action à faire exécuter par le système sous test, voir Figure I.7.

2. Un lecteur de l'état courant (Current State Reader, CSR), permet de lire l'état en cours d'expansion à partir de la file XQ, voir Figure I.8.

3. Un générateur d'état successeur (A Successor Generator : SGEN), ce composant est juste un proxy qui permet de soumettre l'état, l'action et la liste de paramètres nécessaires à l'IUT, pour le calcul d'un éventuel état successeur, voir Figure I.9.

4. Le rédacteur d'état successeur (A Successor State Writer SSW). Chaque fois qu'un état est rencontré pour la première fois lors du processus d'expansion, ce module l'insère dans la file XQ, voir Figure I.10.

5. Lecteur/Rédacteur d'un bloc (Read/Write BLOCK : BLOCK_RW), il s'occupe de la lecture ou écriture d'un bloc en mémoire, voir Figure I.11.



C Data Types

```
Note: above variables are fields of GrexState.
Note: assume that 'include' files define:
        XQ_LEN, HT_LEN, TT_LEN, Pointer, HashCode, TT_Index
        Action, NACTS, ActIndex, actab, State

typedef unsigned int Pointer ;
typedef struct {Pointer ptr ; State st} LSP ;
typedef struct {Pointer head; int size} HTE ;
typedef struct {Action act ; State sst} ASP ;
typedef unsigned int PgmLoc ;

typedef struct {
  PgmLoc   pc          ;
  LSP      XQ[XQ_LEN]  ;
  Pointer  xq_next     ;
  Pointer  xq_free     ;
  State    curr_st     ;
  ActIndex actnum      ;
  State    succ_st     ;
  BOOLEAN  act_enb     ;
  BOOLEAN  inv_ok      ;
  HashCode h_code      ;
  HTE      HT[HT_LEN]  ;
  HTE      ht_elt      ;
  BOOLEAN  found       ;
  Pointer  xq_srch     ;
  LSP      tst_elt     ;
  TTIndex  tt_free     ;
  ASP      TT[TT_LEN}  ;
} GrexState ;
```

TLA+ Data Types

```
Pointer == 0..(XQ_LEN-1)

LSP == [
  ptr : Pointer,
  st  : State ]

HTE == [
  head : Pointer,
  size : Nat ]

ASP == [
  act  : Action,
  sst  : State ]

GrexState == [
  pc : PgmLoc,
  XQ : Array(XQ_LEN, LSP),
  xq_next : 0..XQ_LEN,
  xq_free : 0..XQ_LEN,
  curr_st : State,
  actnum  : 0..NACTS,
  act_in  : Action,
  succ_st : State,
  act_enb : BOOLEAN,
  inv_ok  : BOOLEAN,
  h_code  : 0..(HT_LEN-1),
  HT : Array(HT_LEN, HTE),
  ht_elt  : HTE,
  found   : BOOLEAN,
  xq_srch : Pointer,
  tst_lsp : LSP,
  tt_free : 0..TT_LEN,
  TT : Array(TT_LEN, ASP) ]
```

Figure I.3 – Grex Data Flow

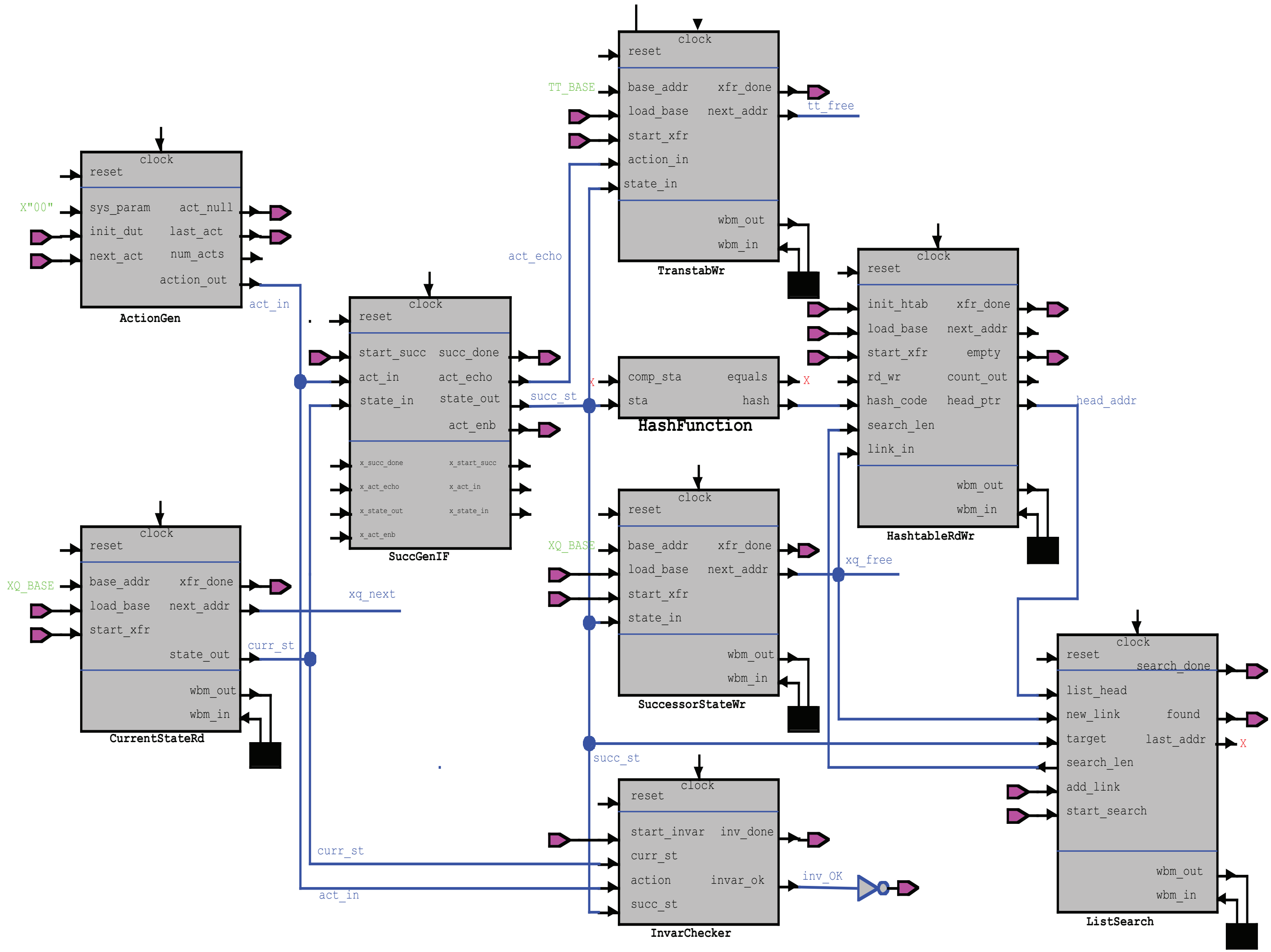


Figure I.4 – Grex Operators Architecture

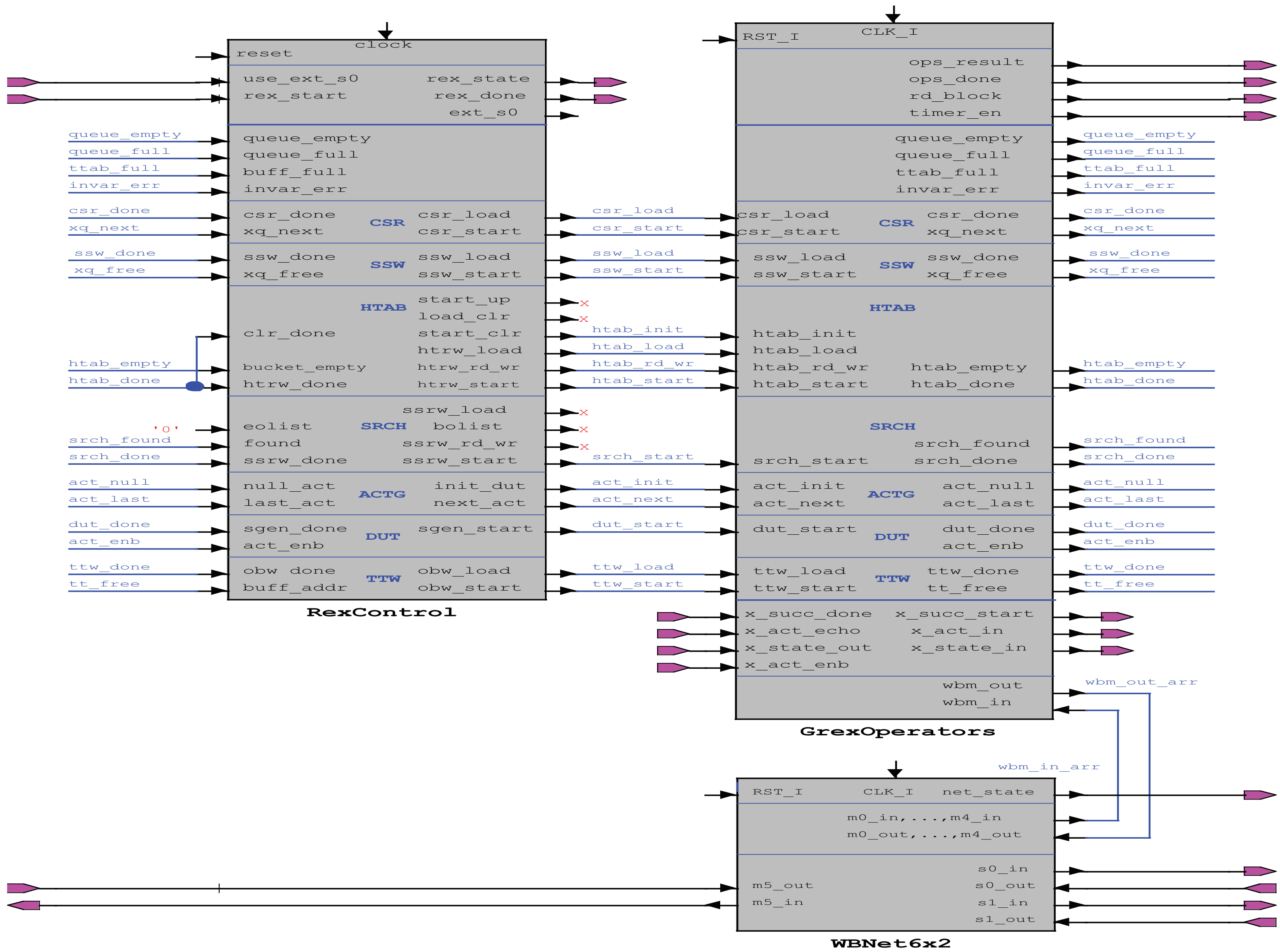


Figure I.5 – Graph Expander Architecture

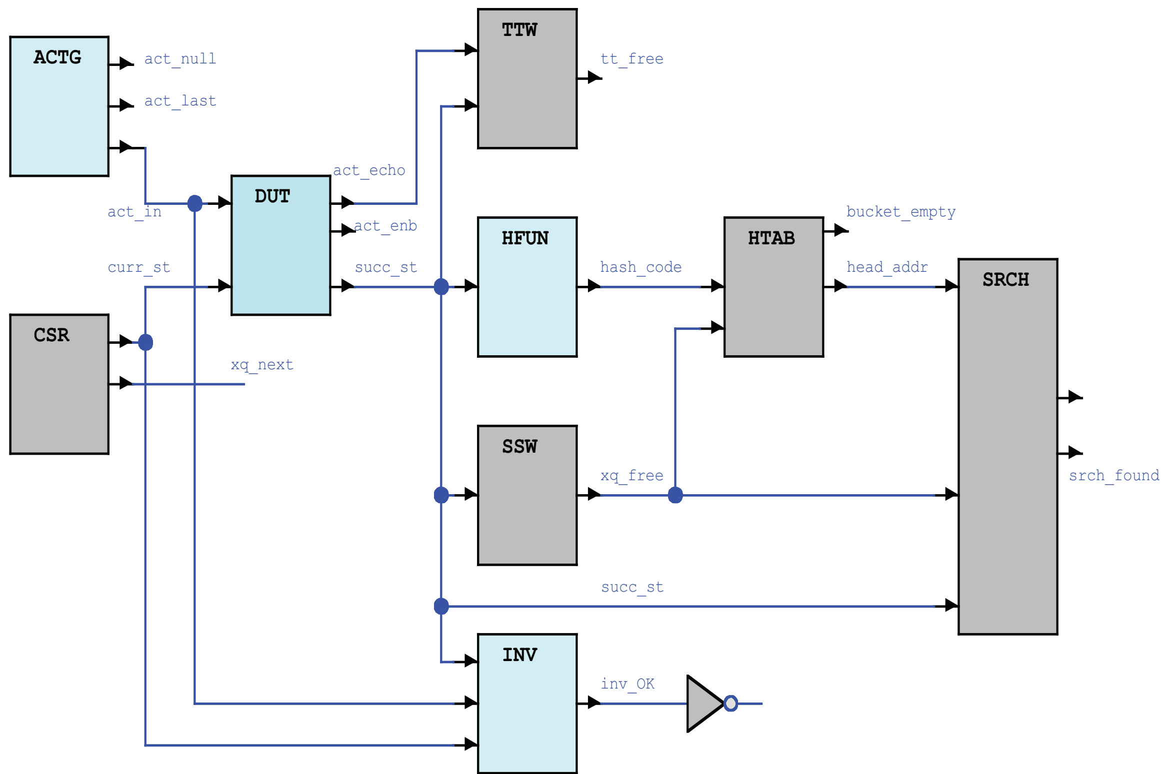


Figure I.6 – Grex Operators Data Paths

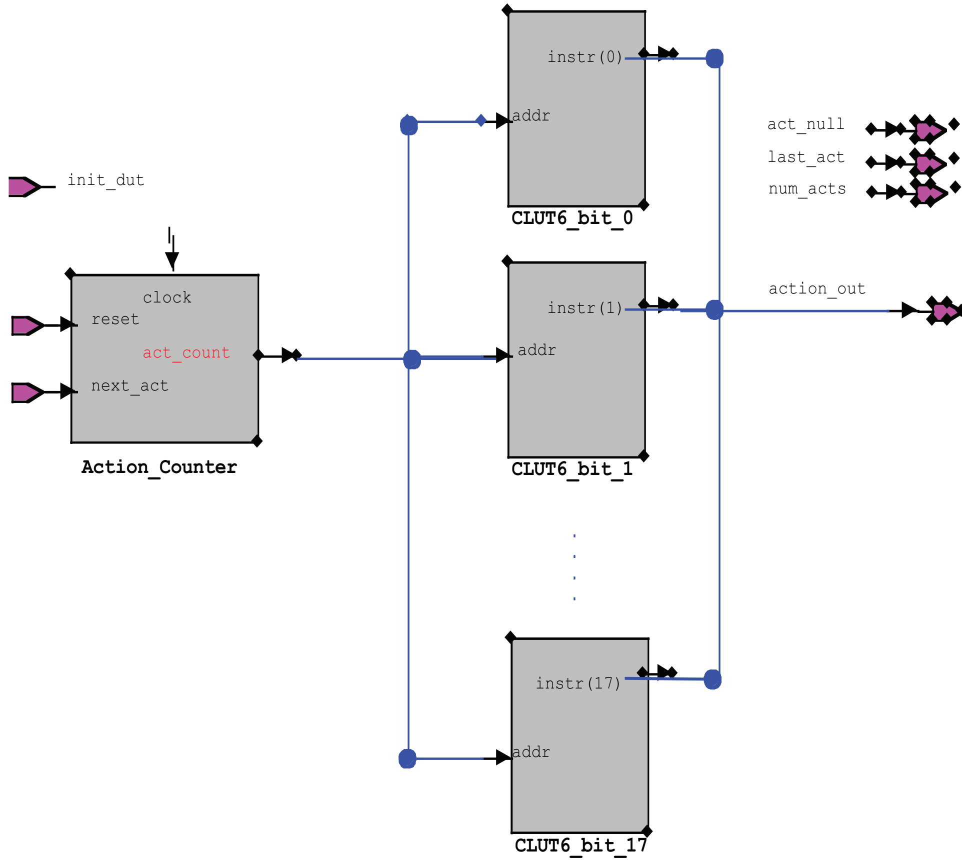


Figure I.7 – Action Generator Architecture

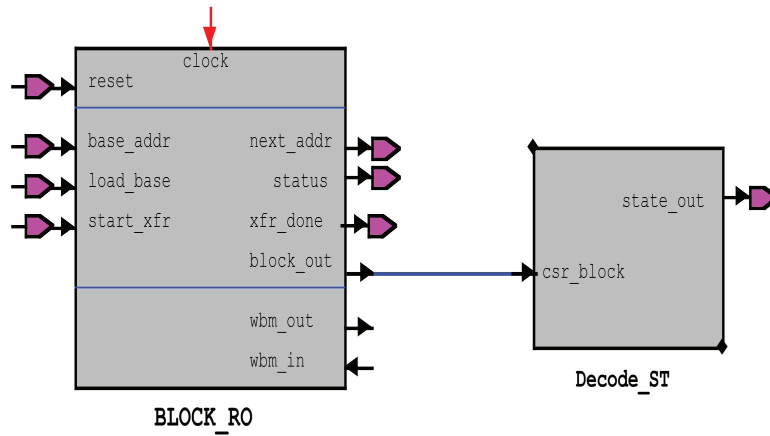


Figure I.8 – Current State Reader Architecture

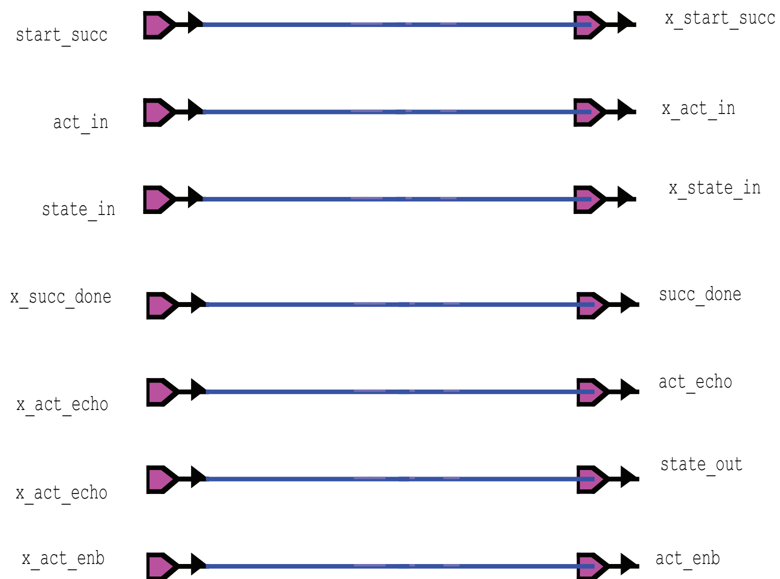


Figure I.9 – A Successor Generator (SGEN) Architecture

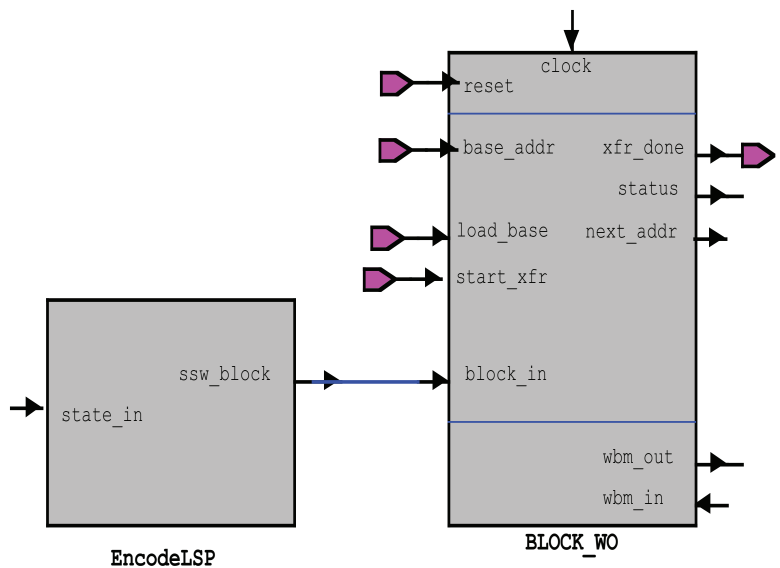


Figure I.10 – A Successor State Writer (SSW) Architecture

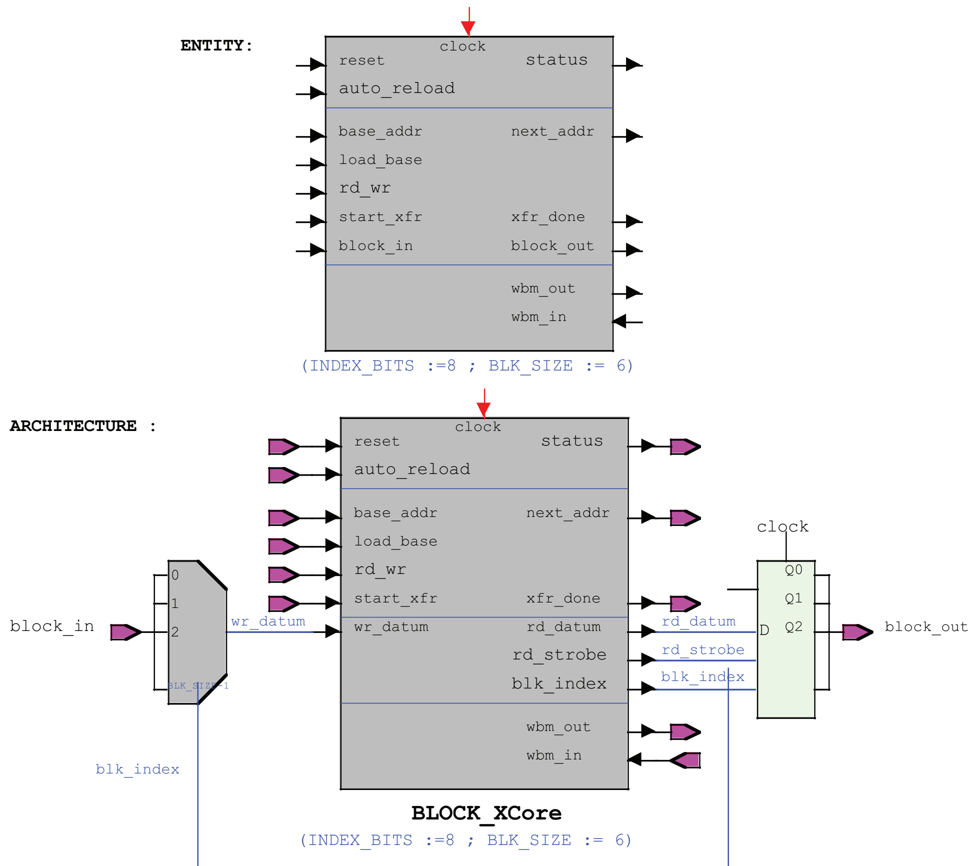


Figure I.11 – Read/Write BLOCK Architecture

6. Recherche dans une liste chaînée (Linked searching List : ListSearch_Linked) : Avant d'être inséré dans la file XQ, on vérifie d'abord si l'état n'y est pas déjà. Pour accélérer la recherche d'un état dans la file XQ, les états sont chaînés entre eux selon une fonction de hachage, voir Figure I.12.

7. Pour améliorer encore le processus de recherche, la file XQ est organisée en un arbre binaire, voir Figure I.13.

8. Lecteur/rédacteur dans la table de Hachage (A Hashtable Reader/Writer : HTRW), un nouvel état, avant d'être inséré dans la file XQ, on lui calcule une valeur par la fonction de hachage qui détermine son entrée dans la table de hachage. Cette entrée donne la tête de liste dans la file XQ ainsi que la taille de cette liste quand celle-ci n'est pas vide. Dans le cas d'une liste vide, il s'agit du premier élément dans une nouvelle liste à créer. La tête de cette dernière va constituer le premier champ et la valeur 1 le deuxième champ d'un maillon à insérer dans la table de Hachage, voir Figure I.14.

9. Rédacteur de table de transition (Tansition Table Writer : TranstabWr), s'occupe de l'écriture de la paire (action, successeur) dans la table de transition, voir Figure I.15.

10. La fonction de hachage (A Hash Function), pour chaque état, elle calcule son entrée dans la table de hachage.

11. Le vérificateur d'invariant(An Invariant Checker), pour chaque état nouvellement calculé, il détermine s'il satisfait un certain invariant définit en VHDL.

I.5.3 Le Contrôleur Grex

Grex (GRph EXpander) contrôle les différents opérateurs via une machine à états (FSM : Finite State Machine) dont nous donnons la structure, voir Figure I.16.

I.5.4 Le Réseau d'interconnexion

Il s'agit d'un bus Wishbone [99] arbitré (voir Figure I.17), multiplexant les opérateurs de Grex. Il leur permet d'agir en maîtres pour lire et/ou écrire des paquets dans des mémoires physiques qui elles sont configurées en esclaves (voir Figures I.19 et I.18).

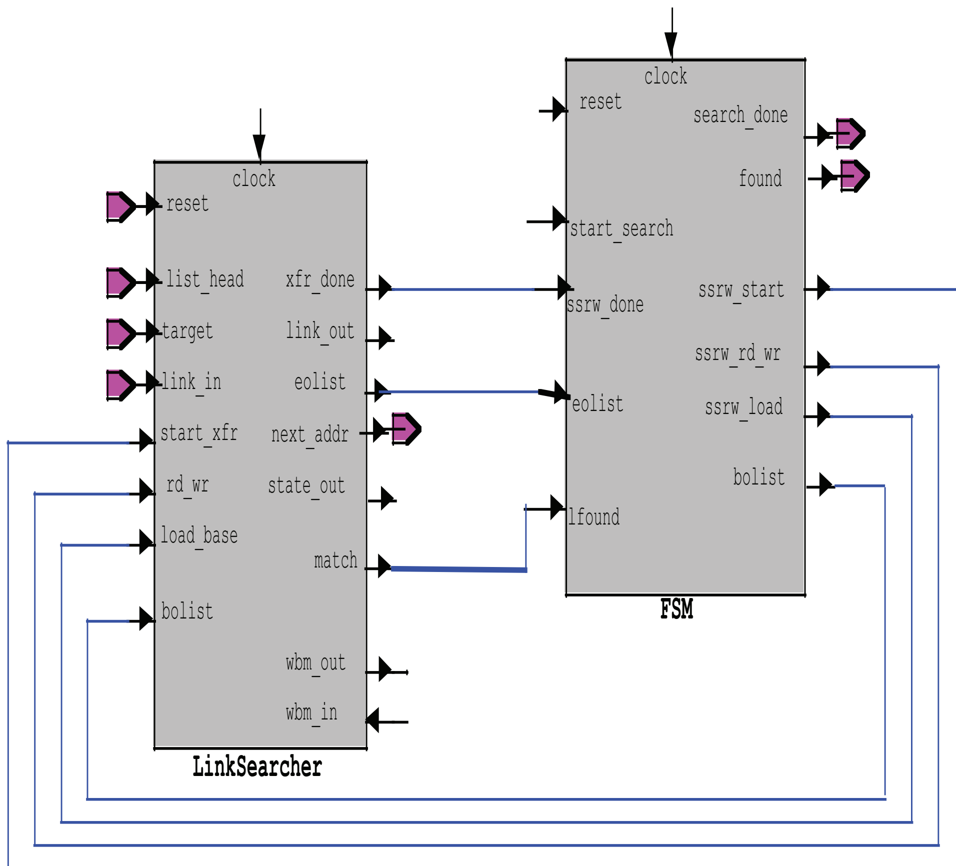


Figure I.12 – Linked searching List Architecture

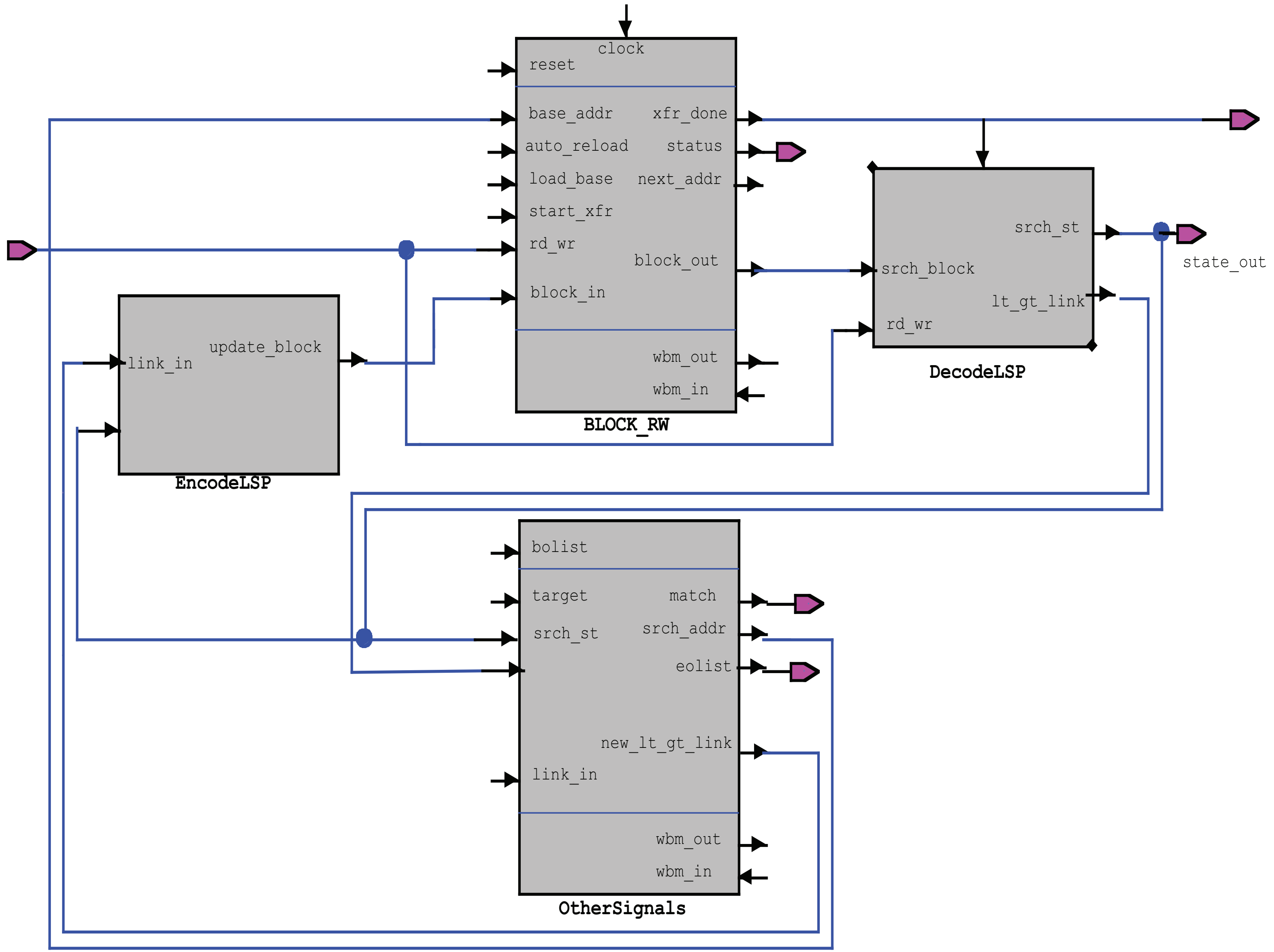


Figure I.13 – Binary searching List Architecture

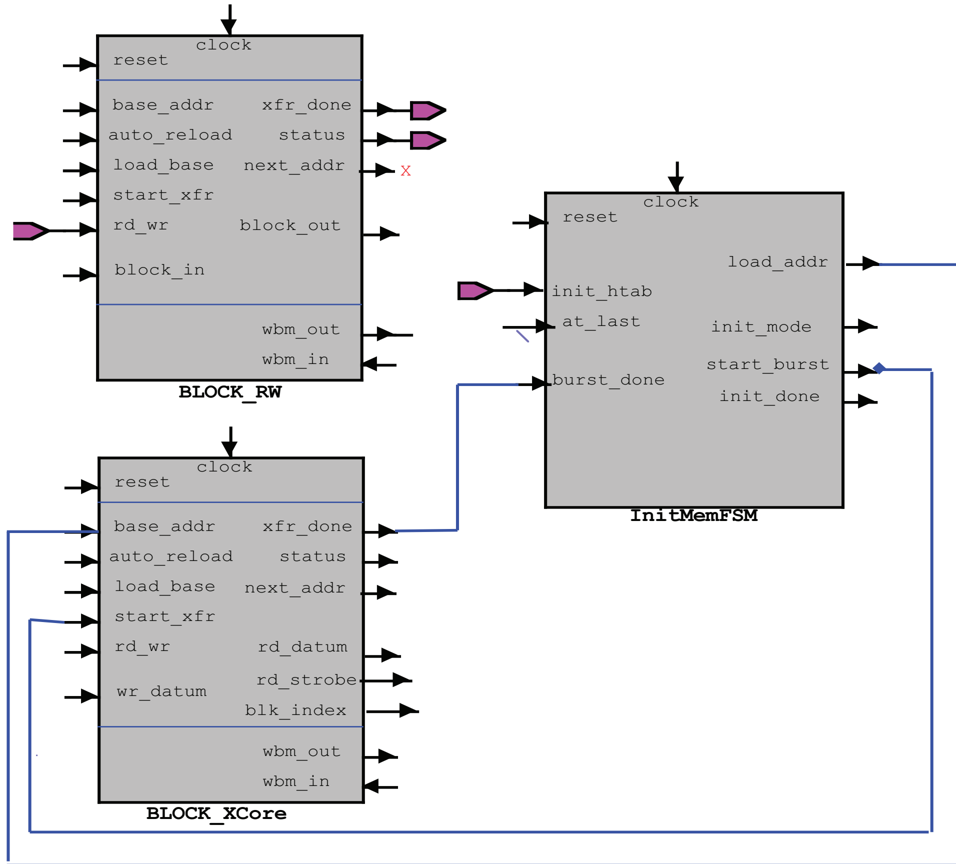


Figure I.14 – A Hashtable Reader/Writer (HTRW) Architecture

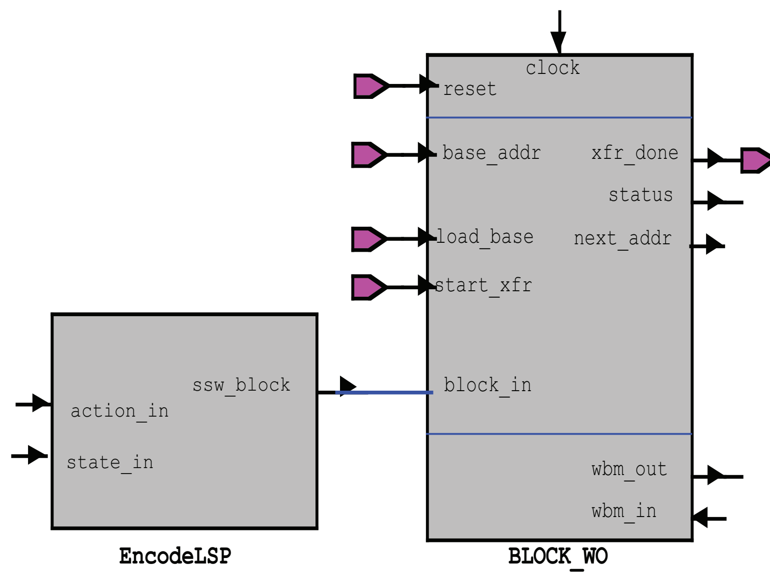


Figure I.15 – Tansition Table Writer Architecture

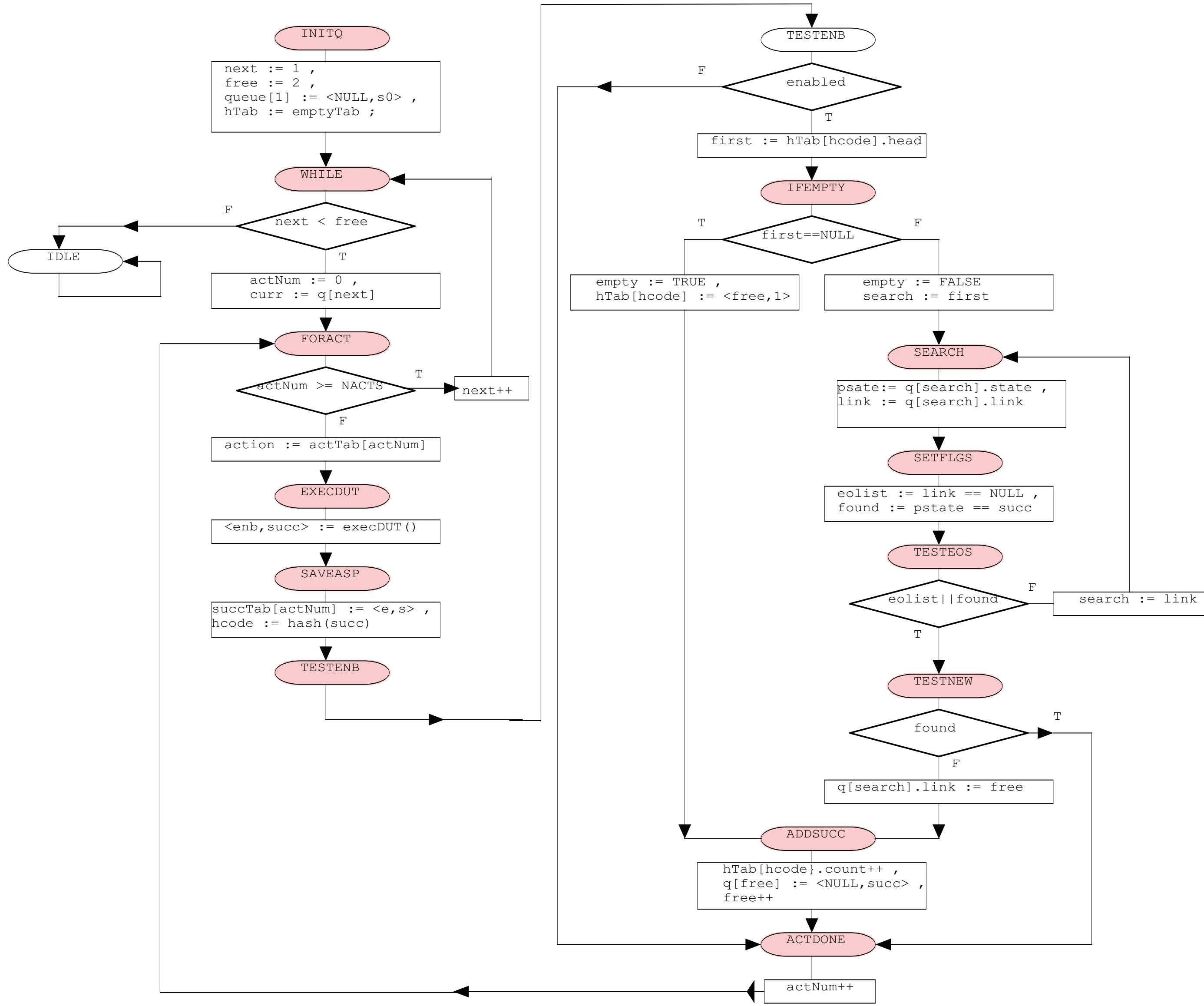


Figure I.16 – Grex Control Flowchart

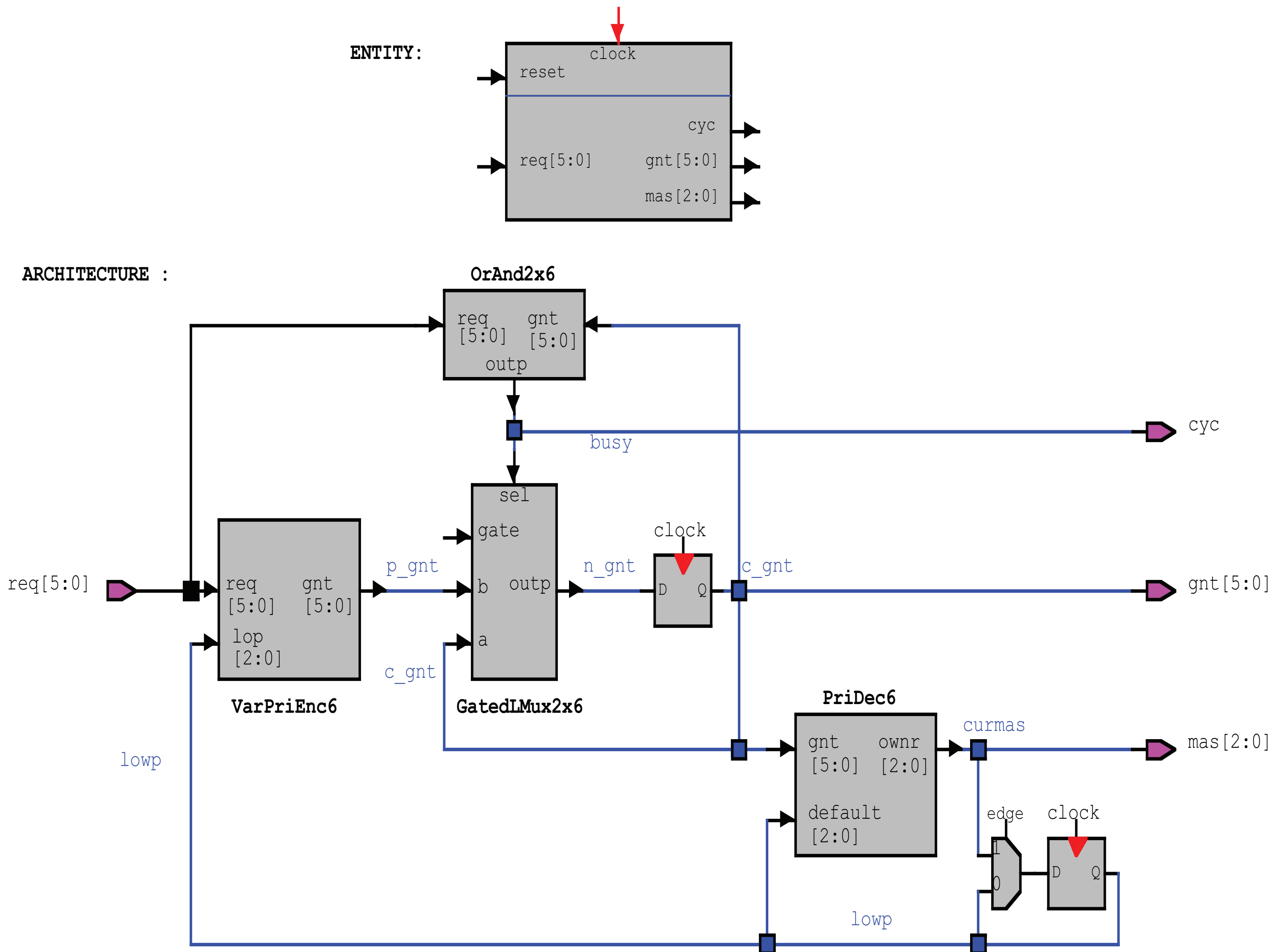


Figure I.17 – Round-Robin 6-User Mutex Arbiter Architecture

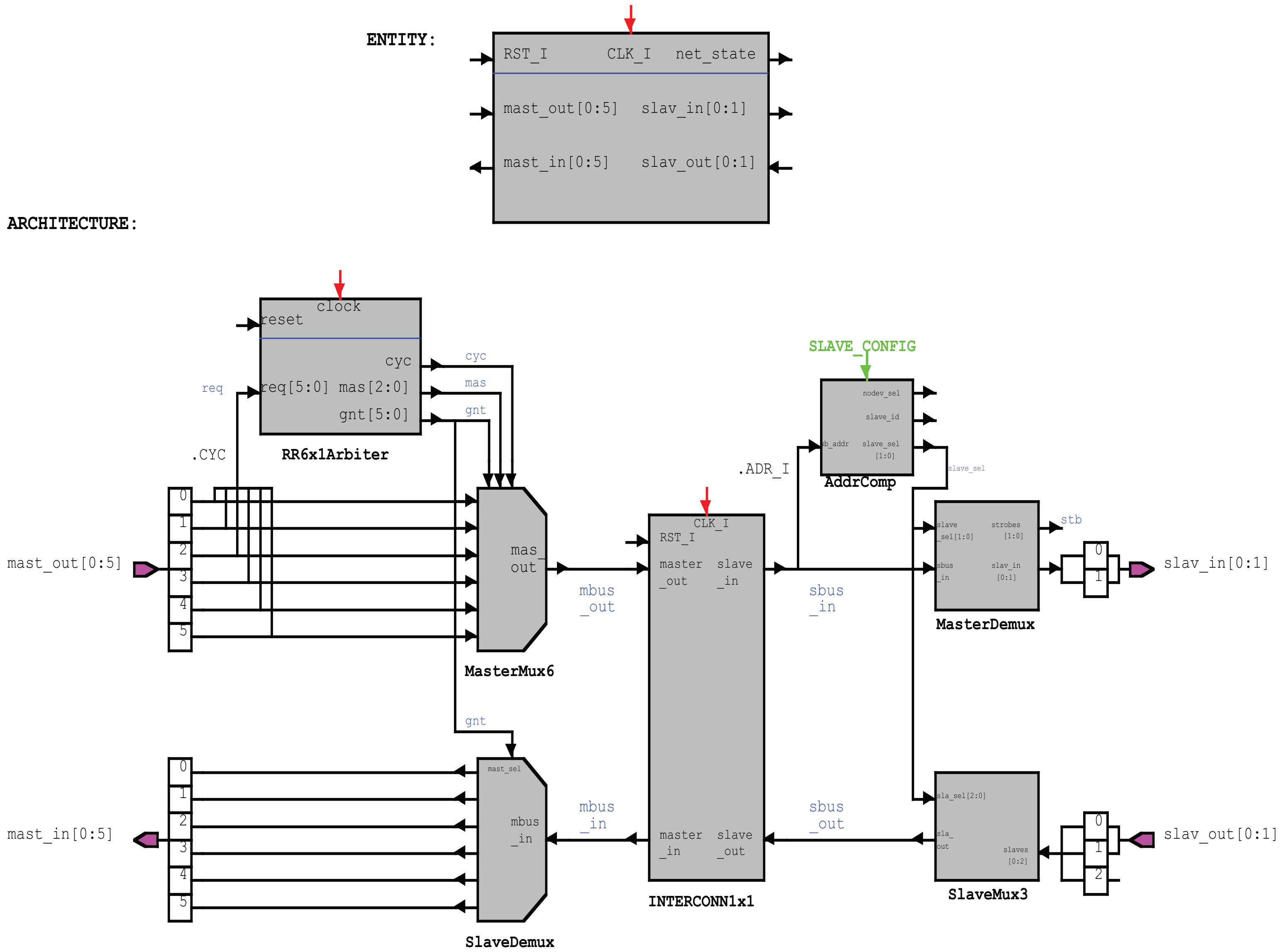


Figure I.18 – 6-Master, 2-Slave Wishbone Network (INTERCONN6x2) Architecture

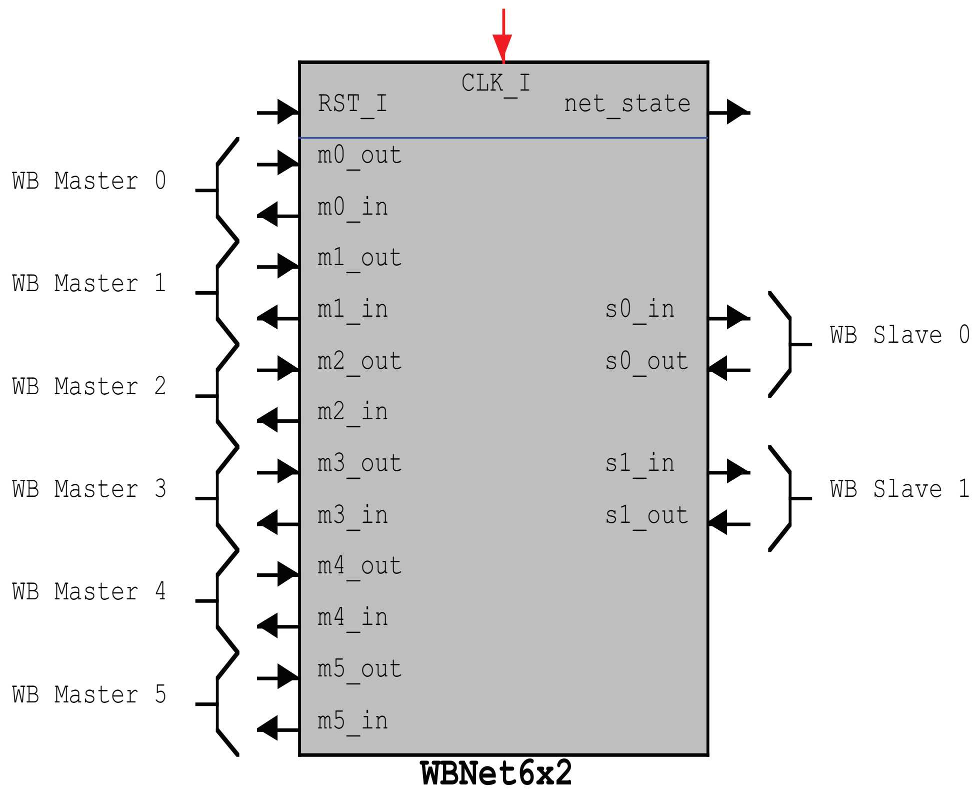


Figure I.19 – 6-Master, 2-Slave INTERCONN6x2 wrapper (WBNet6x2) Architecture

Appendice II

Quelques Modules TLA+

II.1 Spécification des charges (Requirements)

```
----------------------------------- MODULE Requirements  ---------------------------
\**************************************************************************************
\* `^ \textcolor{blue}{\textbf{PicoBlaze Requirements }}
\*
\* ^'
\**************************************************************************************
(* Libraries *)

LOCAL INSTANCE Naturals
LOCAL INSTANCE Sequences
LOCAL INSTANCE TLC
LOCAL INSTANCE BinaryNumbers

VARIABLE pb

--------------------------------------------------------------------------------------
\**************************************************************************************
\* `^ \textcolor{blue}{\textbf{Constants }}
\* ^'
\**************************************************************************************
CONSTANTS DWbyte, DWinstr                            (* Data Widths in bits    *)
CONSTANTS AWpgm, AWreg, AWram, AWstk, AWinp, AWout  (* Address Widths in bits *)
CONSTANTS Nreg, Nram, Nstk, Ninp, Nout              (* Actual Numerical sizes *)

RegID   == 0..(Nreg-1)
RAMAddr == 0..(Nram-1)
OutPort == 0..(Nout-1)

--------------------------------------------------------------------------------------
\**************************************************************************************
\* `^ \textcolor{blue}{\textbf{State Observables }}
\* ^'
\**************************************************************************************
CONSTANT Regs(_)
CONSTANT SCRAM(_)
CONSTANT OutPorts(_)
CONSTANT InPorts(_)
CONSTANT PgmCntr(_)
CONSTANT StackPtr(_)
CONSTANT PStack(_)
CONSTANT CarryBit(_)
```

```
CONSTANT ZeroBit(_)
CONSTANT IEnbBit(_)

--------------------------------------------------------------------------------
\*******************************************************************************
\* `^ \textcolor{blue}{\textbf{Instructions }}
\* ^'
\*******************************************************************************
CONSTANTS LOAD_I(_,_,_), LOAD_R(_,_,_)
CONSTANTS INPUT_DIR(_,_,_), INPUT_IND(_,_,_), FETCH_DIR(_,_,_), FETCH_IND(_,_,_)
CONSTANTS AND_I(_,_,_), AND_R(_,_,_)
CONSTANTS OR_I(_,_,_), OR_R(_,_,_), XOR_I(_,_,_), XOR_R(_,_,_)
CONSTANTS TEST_I(_,_,_), TEST_R(_,_,_)
CONSTANTS COMPARE_I(_,_,_), COMPARE_R(_,_,_)
CONSTANTS ADD_I(_,_,_), ADD_R(_,_,_), ADDCY_I(_,_,_), ADDCY_R(_,_,_)
CONSTANTS SUB_I(_,_,_), SUB_R(_,_,_), SUBCY_I(_,_,_), SUBCY_R(_,_,_)
CONSTANTS RL(_,_), SL0(_,_), SL1(_,_), SLA(_,_), SLX(_,_)
CONSTANTS RR(_,_), SR0(_,_), SR1(_,_), SRA(_,_), SRX(_,_)
CONSTANTS RETURN_U(_), RETURN_C(_), RETURN_NC(_), RETURN_Z(_), RETURN_NZ(_)
CONSTANTS OUTPUT_DIR(_,_,_), OUTPUT_IND(_,_,_), STORE_DIR(_,_,_), STORE_IND(_,_,_)
CONSTANTS JUMP(_,_), JUMP_C(_,_), JUMP_NC(_,_), JUMP_Z(_,_), JUMP_NZ(_,_)
CONSTANTS INTERRUPT_DIS(_), INTERRUPT_ENB(_)
CONSTANT  RESET(_)
(*
CONSTANTS RETINT_DIS(_), RETINT_ENB(_)
CONSTANT INTERRUPT(_)
*)
--------------------------------------------------------------------------------
\*******************************************************************************
\* `^ \textcolor{blue}{\textbf{Actions }}
\* ^'
\*******************************************************************************
CONSTANT clearPC(_)

LOAD_I_Act(rX,byte,s) == s' = clearPC(LOAD_I(rX,byte,s))
LOAD_R_Act(rX,rY,s)   == s' = clearPC(LOAD_R(rX,rY,s))

--------------------------------------------------------------------------------
\*******************************************************************************
\* `^ \textcolor{blue}{\textbf{Type and Initialization Predicates }}
\* ^'
\*******************************************************************************
CONSTANT PBTyped(_)

CONSTANT PBInit(_)
--------------------------------------------------------------------------------
\*******************************************************************************
\* `^ \textcolor{blue}{\textbf{Test Behavior Specification }}
\* ^'
```

```
\*******************************************************************************
RegSet   == 0..1
TestData == 0..255

toByte(n) == NatToBin(n, DWbyte)

vars == pb

Init == PBInit(pb)

Next ==
  \/ \E rX \in RegSet: \E n \in TestData: LOAD_I_Act(rX,toByte(n), pb)
  \/ \E rX,rY \in RegSet : LOAD_R_Act(rX,rY,pb)

Spec == Init /\ [][Next]_vars

-------------------------------------------------------------------------------
\*******************************************************************************
\* `^ \textcolor{blue}{\textbf{Properties }}
\* ^'
\*******************************************************************************
TypeInvar == PBTyped(pb)

===============================================================================
```

II.2 L’Implémentation de Référence

```
------------------------------- MODULE ReferenceImp  ------------------------------
EXTENDS ProcessorISA
\**********************************************************************************
\* `^ \textcolor{blue}{\textbf{PicoBlaze Abstract Reference Implementation }}
\* ^'
\**********************************************************************************
LOCAL INSTANCE BinaryNumbers

INSTANCE FunCodes

(* VARIABLE pb *) (* Must be PRESENT for Requirements; ABSENT for Refinement *)

-----------------------------------------------------------------------------------
\**********************************************************************************
\* `^ \textcolor{blue}{\textbf{State Observables }}
\* ^'
\**********************************************************************************

Regs(s)     == s.Reg
SCRAM(s)    == s.RAM
InPorts(s)  == s.InPort
OutPorts(s) == s.OutPort
PgmCntr(s)  == s.PC
StackPtr(s) == s.SP
PStack(s)   == s.Stack
CarryBit(s) == s.C
ZeroBit(s)  == s.Z
IEnbBit(s)  == s.IE

-----------------------------------------------------------------------------------
\**********************************************************************************
\* `^ \textcolor{blue}{\textbf{Implementation Interface }}
\* ^'
\**********************************************************************************

pup_state   == PB0
isTyped(s)  == PBTyped(s)
psi(s)      == s
prFormat(s) == formatPB(s)

(* clearPC(_) is defined in ProcessorISA; exec(_,_,_) is defined below. *)

-----------------------------------------------------------------------------------
\**********************************************************************************
\* `^ \textcolor{blue}{\textbf{Instruction Execution Dispatched by Type }}
\* ^'
\**********************************************************************************
```

```
(* Register-Immediate Instructions: rX \in RegID, byte \in Byte *)

execRegImm(f,rX,byte, s) ==
  CASE f = INPUT_DIR_f  -> INPUT_DIR (rX, BinToNat(byte), s)
    [] f = OUTPUT_DIR_f -> OUTPUT_DIR(rX, BinToNat(byte), s)
    [] f = FETCH_DIR_f  -> FETCH_DIR (rX, BinToNat(byte), s)
    [] f = STORE_DIR_f  -> STORE_DIR (rX, BinToNat(byte), s)
    [] f = LOAD_I_f     -> LOAD_I    (rX, byte, s)
    [] f = AND_I_f      -> AND_I     (rX, byte, s)
    [] f = OR_I_f       -> OR_I      (rX, byte, s)
    [] f = XOR_I_f      -> XOR_I     (rX, byte, s)
    [] f = ADD_I_f      -> ADD_I     (rX, byte, s)
    [] f = ADDCY_I_f    -> ADDCY_I   (rX, byte, s)
    [] f = SUB_I_f      -> SUB_I     (rX, byte, s)
    [] f = SUBCY_I_f    -> SUBCY_I   (rX, byte, s)
    [] f = TEST_I_f     -> TEST_I    (rX, byte, s)
    [] f = COMP_I_f     -> COMPARE_I (rX, byte, s)

----------------------------------------------------------------------------------
(* Register-Register Instructions: rX,rY \in RegID *)

execRegReg(f,rX,rY, s) ==
  CASE f = INPUT_IND_f  -> INPUT_IND (rX, rY, s)
    [] f = OUTPUT_IND_f -> OUTPUT_IND(rX, rY, s)
    [] f = FETCH_IND_f  -> FETCH_IND (rX, rY, s)
    [] f = STORE_IND_f  -> STORE_IND (rX, rY, s)
    [] f = LOAD_R_f     -> LOAD_R    (rX, rY, s)
    [] f = AND_R_f      -> AND_R     (rX, rY, s)
    [] f = OR_R_f       -> OR_R      (rX, rY, s)
    [] f = XOR_R_f      -> XOR_R     (rX, rY, s)
    [] f = ADD_R_f      -> ADD_R     (rX, rY, s)
    [] f = ADDCY_R_f    -> ADDCY_R   (rX, rY, s)
    [] f = SUB_R_f      -> SUB_R     (rX, rY, s)
    [] f = SUBCY_R_f    -> SUBCY_R   (rX, rY, s)
    [] f = TEST_R_f     -> TEST_R    (rX, rY, s)
    [] f = COMP_R_f     -> COMPARE_R (rX, rY, s)

----------------------------------------------------------------------------------
(* Shift & Rotate Instructions: rX \in RegID *)

execShift(f,rX, s) ==
  CASE f = RL_f   -> RL (rX, s)
    [] f = SL0_f  -> SL0(rX, s)
    [] f = SL1_f  -> SL1(rX, s)
    [] f = SLA_f  -> SLA(rX, s)
    [] f = SLX_f  -> SLX(rX, s)

    [] f = RR_f   -> RR (rX, s)
    [] f = SR0_f  -> SR0(rX, s)
```

```
    [] f = SR1_f  -> SR1(rX, s)
    [] f = SRA_f  -> SRA(rX, s)
    [] f = SRX_f  -> SRX(rX, s)

-----------------------------------------------------------------------------------
(* Branch Instructions: addr \in PgmAddress *)

execBranch(f,addr, s) ==
  CASE f = JUMP_U_f    -> JUMP   (addr, s)
    [] f = JUMP_C_f    -> JUMP_C (addr, s)
    [] f = JUMP_NC_f   -> JUMP_NC(addr, s)
    [] f = JUMP_Z_f    -> JUMP_Z (addr, s)
    [] f = JUMP_NZ_f   -> JUMP_NZ(addr, s)

    [] f = CALL_U_f    -> CALL_U (addr, s)
    [] f = CALL_C_f    -> CALL_C (addr, s)
    [] f = CALL_NC_f   -> CALL_NC(addr, s)
    [] f = CALL_Z_f    -> CALL_Z (addr, s)
    [] f = CALL_NZ_f   -> CALL_NZ(addr, s)

    [] f = RETURN_U_f  -> RETURN_U (s)
    [] f = RETURN_C_f  -> RETURN_C (s)
    [] f = RETURN_NC_f -> RETURN_NC(s)
    [] f = RETURN_Z_f  -> RETURN_Z (s)
    [] f = RETURN_NZ_f -> RETURN_NZ(s)

-----------------------------------------------------------------------------------
(* Single-Bit Instructions: bit \in Bit *)

execIntEnb(f,bit, s) ==
  CASE f = SETIE_f  -> IF bit = 1 THEN INTERRUPT_ENB(s)
                                  ELSE INTERRUPT_DIS(s)
    [] f = SETZ_f   -> SETZ(bit, s)

(*  [] f = RETINT_f -> IF bit = 1 THEN RETINT_ENB(s)
                                  ELSE RETINT_DIS(s) *)

-----------------------------------------------------------------------------------
(* Pseudo-Instructions *)

execPseudo(f, args, s) ==
  CASE f = NOP_f      -> NOP(s)
    [] f = NULLACT_f  -> NULLACT(s)
    [] f = RESET_f    -> RESET(s)

(*  [] f = INTERRUPT_f -> INTERRUPT(s) *)

-----------------------------------------------------------------------------------
\**********************************************************************************
```

```
\* `^ \textcolor{blue}{\textbf{Composite Instruction Execution }}
\* ^'
\*******************************************************************************

exec(f, args, s) ==
  CASE isRegImm(f)   -> execRegImm(f, args[1], args[2], s)  (* args = <rX,byte> *)
    [] isRegReg(f)   -> execRegReg(f, args[1], args[2], s)  (* args = <rX,rY>   *)
    [] isShift(f)    -> execShift (f, args[1], s)           (* args = <rX>      *)
    [] isRetCode(f)  -> execBranch(f,       0, s)           (* args = <>        *)
    [] isGoCode(f)   -> execBranch(f, args[1], s)           (* args = <addr>    *)
    [] isOneBit(f)   -> execIntEnb(f, args[1], s)           (* args = <bit>     *)
    [] isPseudo(f)   -> execPseudo(f, args,    s)

-------------------------------------------------------------------------------
\*******************************************************************************
\* `^ \textcolor{blue}{\textbf{Analysis }}
\* ^'
\*******************************************************************************

(* INSTANCE Requirements *) (* PRESENT for Requirements; ABSENT for Refinement *)

===============================================================================
```

II.3 Le critère de Raffinement

```
------------------------- MODULE RefinementCriteria --------------------------
\*****************************************************************************
\* `^ \textcolor{blue}{\textbf{CPU ISA Refinement Checker }}
\*
\*    This module compares the behavior of an external, either TLA or physical,
\* implementation
\* with that of an internal (Abstract) Reference implementation, instruction
\* by instruction. The comparison is based on the TLA concept of Refinement, a
\* form of homomorphism, or congruence, between Algebras. The resulting
\* Congruence criterion is described and defined below.
\* ^'
\*****************************************************************************
LOCAL INSTANCE Naturals
LOCAL INSTANCE Sequences
LOCAL INSTANCE TLC

LOCAL INSTANCE BinaryNumbers

VARIABLE pbsys
-------------------------------------------------------------------------------
\*****************************************************************************
\* `^ \textcolor{blue}{\textbf{Declare Quantities from ImpTester Environment}}
\* ^'
\*****************************************************************************
(* From the Implementation Under Test (IUT) : *)

CONSTANTS pup_state, exec(_,_,_), isTyped(_), clearPC(_), psi(_), prFormat(_)

(* From Test Specifications : *)

CONSTANT TestSuite, toArgs(_), showTestSuite(_,_)

-------------------------------------------------------------------------------
\*****************************************************************************
\* `^ \textcolor{blue}{\textbf{Get the Abstract Reference Implementation (ARI) }}
\* ^'
\*****************************************************************************
REF == INSTANCE ReferenceImp

-------------------------------------------------------------------------------
\*****************************************************************************
\* `^ \textcolor{blue}{\textbf{ Define Refinement (Congruence) Criterion }}
\* ^'
\*****************************************************************************
(*
`.   The predicate Congruent(f,args,s,f_s) is true iff the following diagram
  is commutative, where s denotes a processor state:
```

```
                              f_args(_)
                         s. -------------> . f_s = f_args(s)    <- IUT Domain
                          |                |
                   psi(_) |                | psi(_)
                          |                |
                          v                v
                   psi(s) . -------------> .                    <- REF Domain
                              f_args(_)

  that is, iff      psi(f_args(s)) = f_args(psi(s)) ,

  where f represents an instruction, args represents its arguments,
  and f_args(_) is the operator f_args : States -> States representing
  execution of instruction f with arguments args.
  f_s is the processor state resulting from executing
  instruction f_args while in processor state s: f_s = f_args(s).
  psi(_) is a Refinement Map which maps implementation states to reference
  model states.

  If this is true for all Reachable s, then psi(_) is an algebra homomorphism .'
*)
--------------------------------------------------------------------------------
(* Refinement Error Reporting and Diagnosis *)

Diagnose(f,args,s,f_s,psi_f_s,f_psi_s) ==
  LET instruct == REF!toAssembler(f,args)
  IN
  /\ PrintT("Refinement Error for instruction:  " \o instruct )
  /\ PrintT("IUT Parent State s:")
  /\ PrintT(prFormat(s))
  /\ PrintT("IUT Successor State f(s):")
  /\ PrintT(prFormat(f_s))
  /\ PrintT("Image of Parent State psi(s):")
  /\ PrintT( REF!formatPB(psi(s))  )
  /\ PrintT("Image of IUT Successor psi(f(s)):")
  /\ PrintT( REF!formatPB(psi_f_s) )
  /\ PrintT("Reference Successor f(psi(s)):")
  /\ PrintT( REF!formatPB(f_psi_s) )
  /\ REF!compareStates(instruct, f_psi_s, psi_f_s)  (* expected vs actual *)
  /\ FALSE

--------------------------------------------------------------------------------
(* Congruenece *)

Congruent(f,args,s,f_s) ==
  LET psi_f_s == psi(f_s)                              (* map   f_s   to REF *)
      f_psi_s == REF!clearPC(REF!exec(f,args, psi(s))) (*exec f_args in REF *)
    (*f_psi_s == REF!exec(f,args, psi(s))*)            (* exec f_args in REF *)
```

```
  IN  IF f_psi_s = psi_f_s  THEN TRUE                  (* test equals in REF *)
      ELSE Diagnose(f,args,s,f_s,psi_f_s,f_psi_s)

verify(f,args,s,f_s) == TRUE
  (*Assert(Congruent(f,args,s,f_s), "Congruence Failure")*)

-------------------------------------------------------------------------------
\******************************************************************************
\* `^ \textcolor{blue}{\textbf{ Define Refinement-Checker Behavior }}
\* ^'
\******************************************************************************
(* The Instruction Execution Action  (includes Refinement Check !!!) *)

verifyExec(f,args,s) ==
  LET f_s == clearPC(exec(f,args,s))  (* processor succ state = f_args(s) *)
(*LET f_s == exec(f,args,s)*)         (* processor succ state = f_args(s) *)
  IN  /\ TLCSet(1,f_s)                (* save f_s in TLC cell number 1    *)
      /\ s' = TLCGet(1)               (* assign f_s to successor state s' *)
      /\ verify(f,args,s,TLCGet(1))   (* check congruence criterion       *)

-------------------------------------------------------------------------------
(* Checker Behavior *)

vars == <<pbsys>>

Init ==
   /\ pbsys = pup_state
   /\ PrintT("------------------------------------------------------")
   /\ PrintT("Implementation Initial (Power-up) State s0 is:")
   /\ PrintT(prFormat(pup_state))
   /\ psi(pup_state) = REF!pup_state
   /\ showTestSuite("            Instruction Test Suite", TestSuite)

Next ==
   \E i \in DOMAIN TestSuite :
   \E f    \in TestSuite[i].ins_set :
   \E args \in toArgs(TestSuite[i].arg_seq) :
   verifyExec(f,args,pbsys)

Spec == Init /\ [][Next]_vars

-------------------------------------------------------------------------------
(* IUT Type Invariant *)

TypeInvar == isTyped(pbsys)

===============================================================================
```

Appendice III

L'interface Java

III.1 Le module de sur-définition (over-riding) Java

```
//file tlc2\module\ExtPicoBlazeImp.java
// PACKAGE
package tlc2.module ;

// IMPORTS

// TLA+ tools:
import util.Assert ;
import util.UniqueString ;
import tlc2.value.* ;
import tlc2.tool.*  ;

// GREX:
import mrapt.* ;            /* General MRAPT Grex Interfaces     */
import mrapt.digilent.* ;   /* Digilent Adept API for Comm Link  */
import pico.*  ;            /* PicoBlaze Project specifics       */

// CLASSES
/******************************************************************************/
public class ExtPicoBlazeImp extends Object implements ValueConstants {

  /*------------------------------------------------------------------------*/
  /*                              STATIC                                    */
  /*------------------------------------------------------------------------*/
  // CONSTANTS

  static final boolean DEBUG  = false ;

  // VARIABLES

  static int cfg_num = 0 ;
  static StateTransitionSystem sts = null ;  /* STS Interface */

  // INITIALIZATION

  // Instantiate STS Implementation:
  static {
    sts = new PicoSTS() ;         /* pico.PicoSTS  */
  }
  /*------------------------------------------------------------------------*/
  /*       IMPLEMENTATION OF OPERATOR PROTOTYPES IN ExtPicoBlazeImp.tla      */
```

```
  /*----------------------------------------------------------------------*/
  /**
   * init_state(cfg_id, rep_size)
   */
  public static Value init_state(Value cfg_id, Value rep_size) {
    int id_num = ((IntValue)cfg_id  ).val ; // decode cfg_id
    int nwords = ((IntValue)rep_size).val ; // decode rep_size
    cfg_num    = id_num ; // -> set static field
    int[] s0 = ((PicoState)sts.getInitialState()).toIntArray() ;
    s0[0] = cfg_num ;     // over-ride s0[0] with cfg_num
    BinWordArray cpu0 = new BinWordArray(s0, 16) ;
    return cpu0.toTLCValue() ;  // Convert Java int[] -> TLC value
  } // end method init_state

  /**
   * dut_exec(instr, cpu_st)
   */
  public static Value dut_exec(Value instr,  Value cpu_st) {
    BinWordArray ins  = new BinWordArray(instr)   ; // TLC value -> Java int[]
    BinWordArray cpu  = new BinWordArray(cpu_st)  ; // TLC value -> Java int[]
    PicoAction   act  = new PicoAction(ins.state) ; // int[] -> PicoAction
    PicoState    curr = new PicoState(cpu.state)  ; // int[] -> PicoState
    PicoState    succ = (PicoState)sts.getSuccessor(act,curr) ;
    BinWordArray next_cpu = new BinWordArray(succ.toIntArray(), 16) ;
    next_cpu.state[0] = cfg_num ;
    return next_cpu.toTLCValue() ; // Convert Java int[] -> TLC value
  } // end method dut_exec

} // end class ExtPicoBlazeImp
/************************************************************************/
// end file tlc2\module\ExtPicoBlazeImp.java
```

III.2 Le diagramme des classes, vue globale

Comme on peut le voir sur les figures III.3 et III.2, l'API Java est structurée de façon à regrouper tout ce qui est commun à toutes les applications dans le package "mrapt". Celui-ci permet de définir un système à états et transitions (State Transition System). Dans le package "mrapt", nous avons un sous package "digilent" qui permet de personnaliser un Grex (GRaph EXpander) selon les caractéristiques de la plate-forme cible. Le package "adept" nous offre les opérations de communications avec la plate-forme cible. Enfin un package "pico" dédié à l'application à vérifier qui est dans le cas de cette exemple celui du micro-contrôleur PicoBlaze.

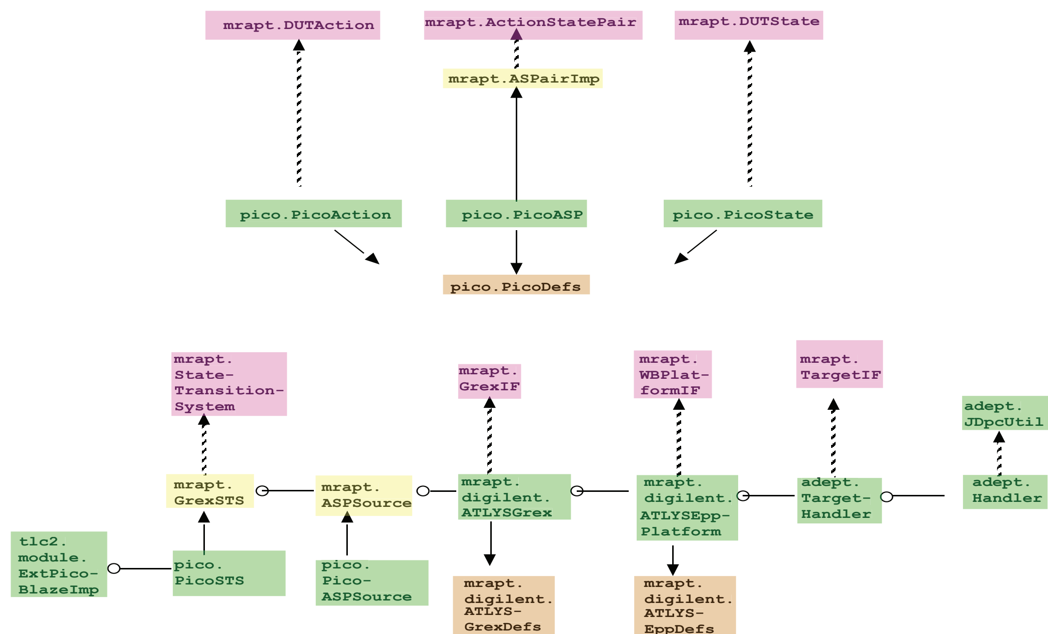


Figure III.1 – Le diagramme des classes, vue globale

Quelques classes

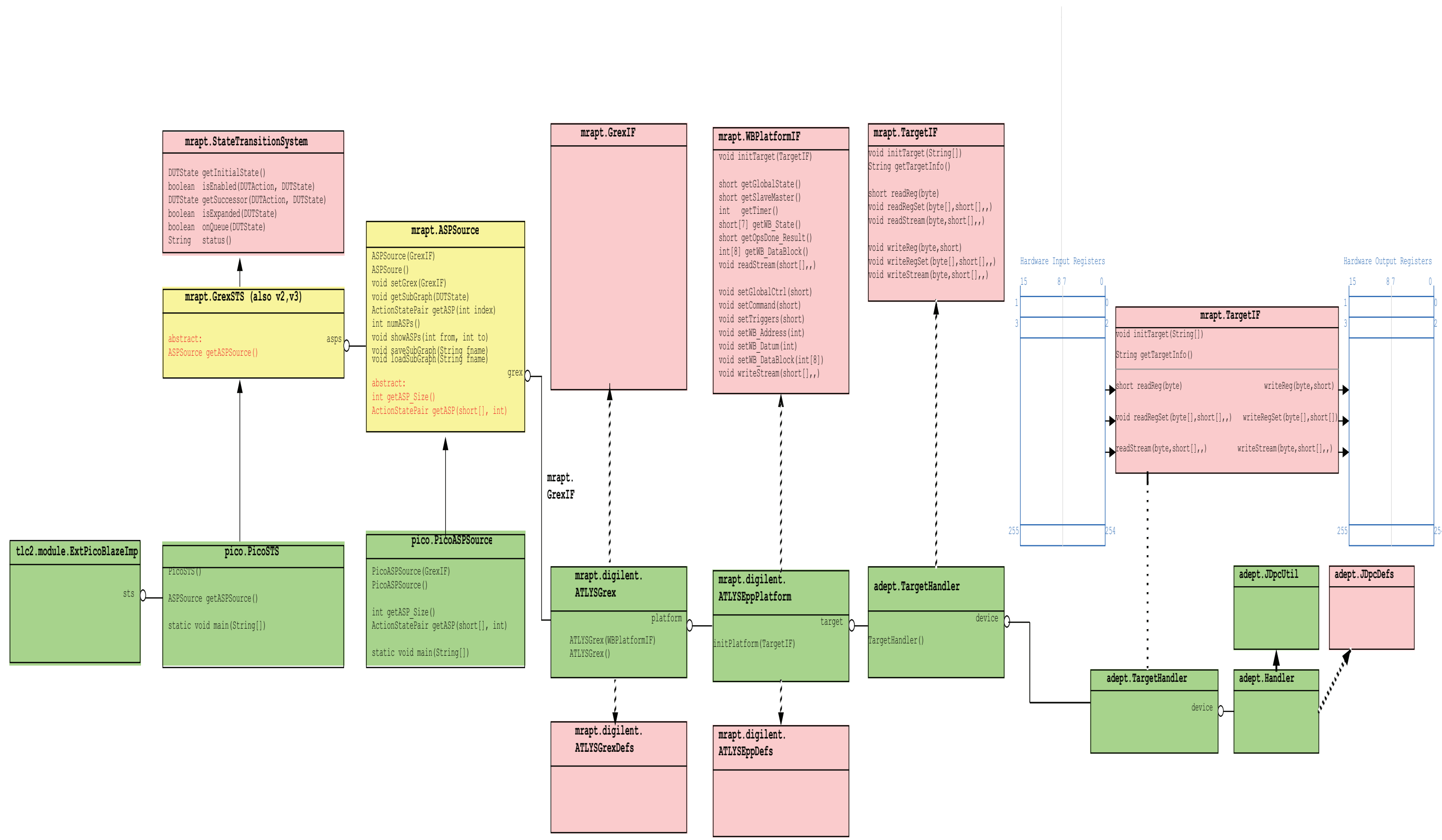


Figure III.2 – L'API Java

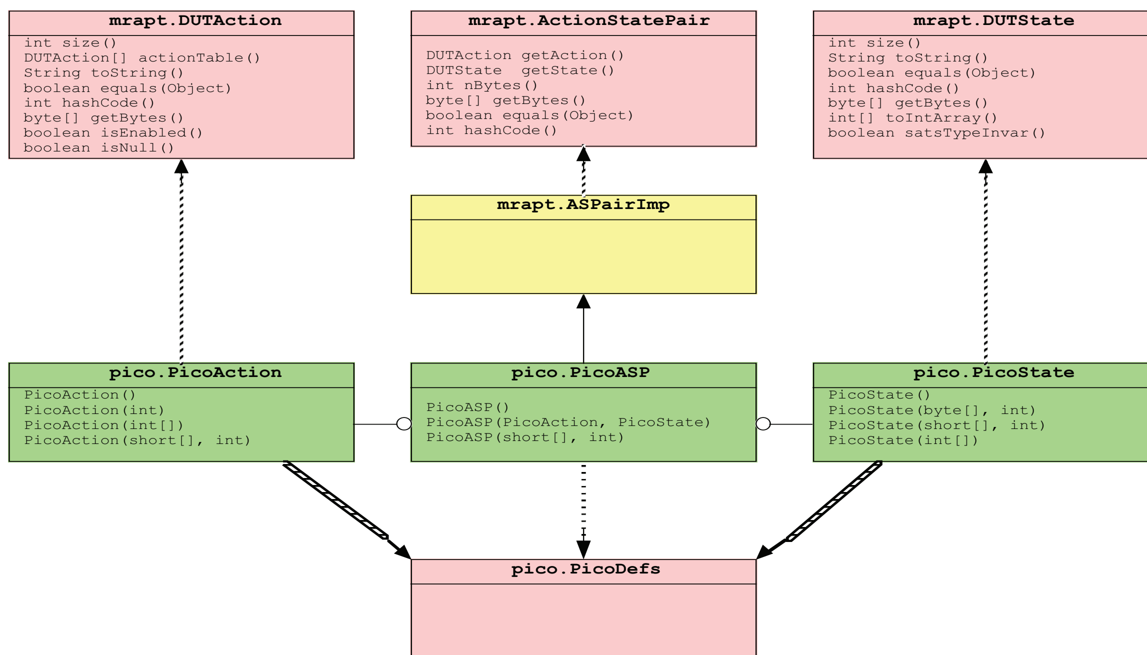


Figure III.3 – L'API Java (suite 1)

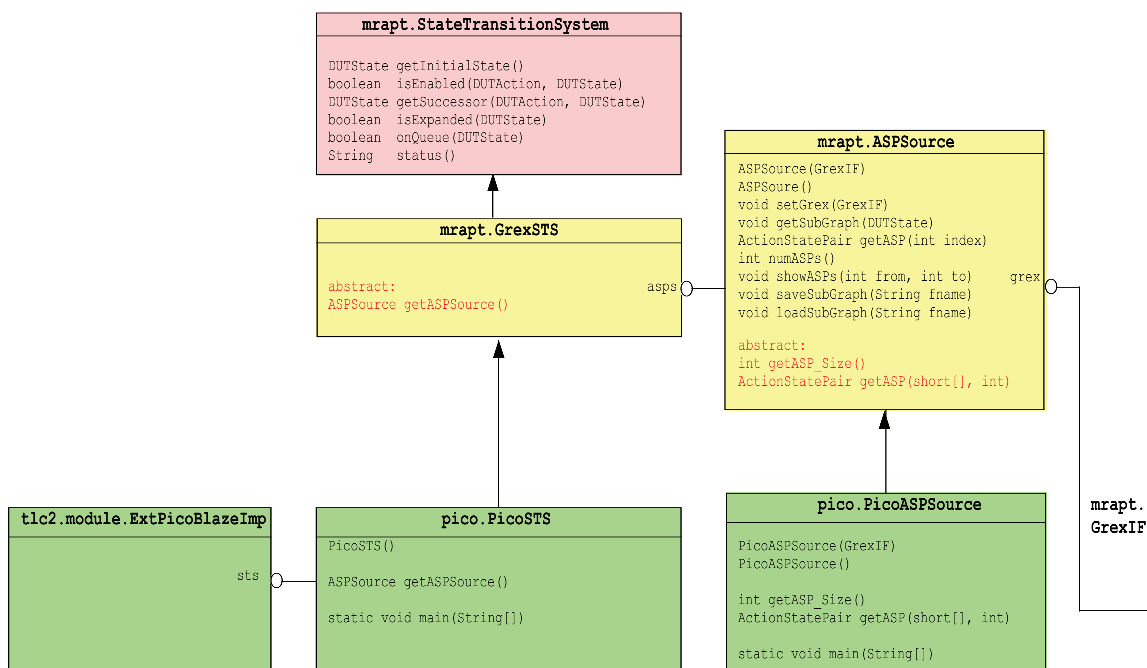


Figure III.4 – L'API Java (suite 2)

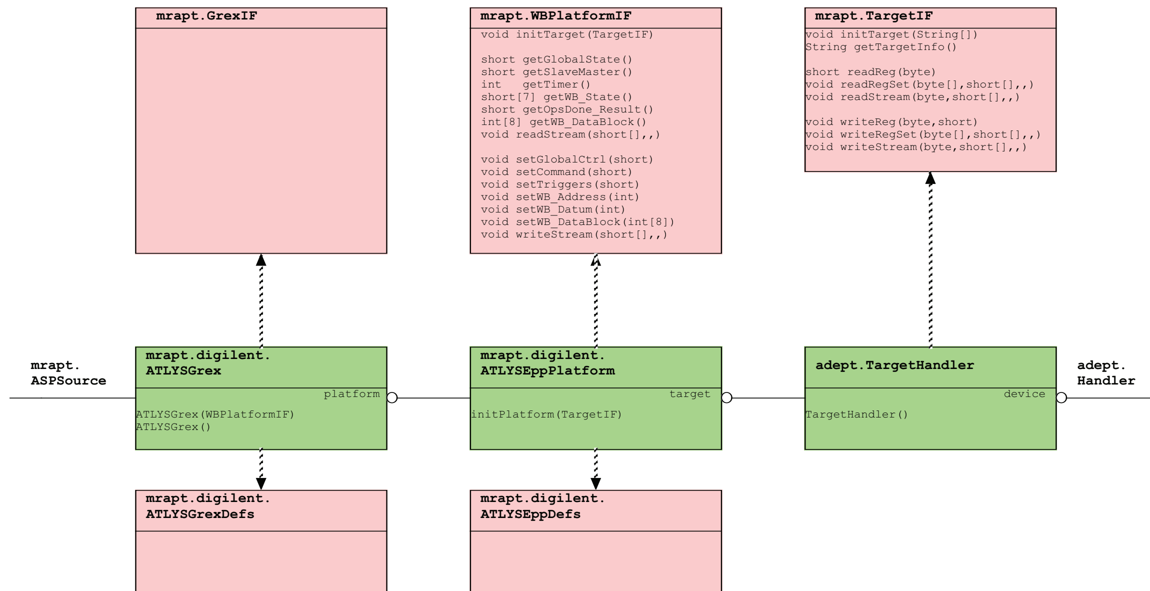


Figure III.5 – L'API Java (suite 3)

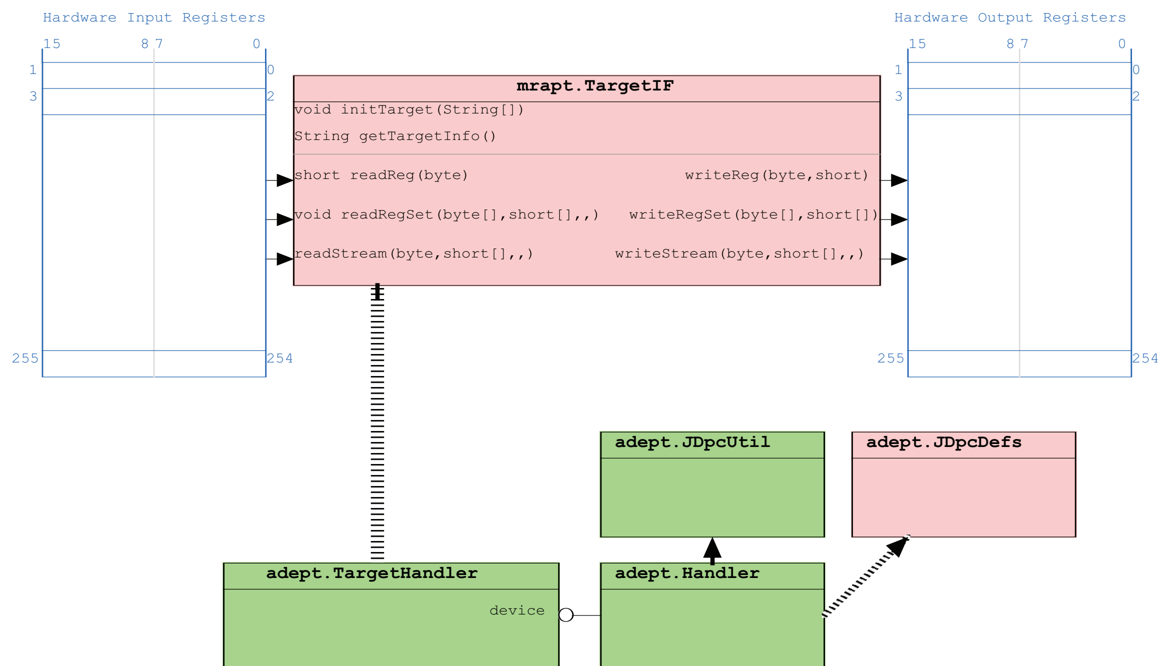


Figure III.6 – L'API Java (suite 4)

Appendice IV

Sommaire de la méthodologie MRAPT

Ce projet a nécessité deux systèmes inter-connectés répartis sur deux plates-formes distinctes :

1. Un PC hôte : celui-ci héberge la partie logique de notre système qui est structurée en deux sous composantes :

   - Une composante intégrant le vérificateur TLC et le modèle TLA+ du système à vérifier.
   - Une interface java qui permet de relier le vérificateur de modèles TLC à la réalisation physique.

2. La carte FPGA : celle-ci héberge la partie matérielle de notre système qui est structurée en deux sous composantes :

   - Une composante intégrant notre analyseur d'accessibilité implanté en matériel et la réalisation physique du système à vérifier.
   - Une interface VHDL qui permet de relier la partie matérielle de notre système à la machine hôte.

3. Un lien de communication reliant le PC à la plate-forme cible permettant contrôle et observation du circuit sous test.

Voici les étapes à suivre pour utiliser notre méthodologie afin de vérifier un système numérique :

1. Identifier les entités observables du système considéré ; ce sont des composantes définissant son état s. Pour l'exemple du contrôleur d'ascenseur nous avons l'étage courant de l'ascenseur, la direction de son mouvement, etc.

2. Identifier les actions qui définissent les changements d'états du système en incluant d'éventuels paramètres. Pour l'exemple du contrôleur d'ascenseur nous avons par exemple les actions monter et descendre.

3. Exprimer les propriétés et les contraintes désirées pour le système considéré. Pour l'exemple du contrôleur d'ascenseur nous avons la contrainte qui empêche l'ascenseur d'aller au delà du dernier étage. Jusqu'à ce point nous sommes au niveau de la spécification abstraite du système à vérifier.

4. Développer une implémentation de référence abstraite en TLA+, et vérifier qu'elle satisfait les contraintes et propriétés définies à l'étape spécification. Le résultat, à ce niveau, est appelé une implémentation de référence.

5. Exprimer les conditions qu'une implémentation doit satisfaire pour qu'elle soit un raffinement de implémentation de référence. Ceci constitue le critère de raffinement. Pour l'exemple du contrôleur d'ascenseur, une de ces conditions assure que chaque état de la réalisation physique est converti vers un état du domaine abstrait et celui-ci doit être égal à l'état de la réalisation de référence.

6. Développer une réalisation physique du système à vérifier. À ce stade ci du développement, le système à vérifier atteint un bon niveau de maturité et le passage de son expression abstraite à une expression matérielle est aussi facilité. En effet, il y a entre autres correspondance directe entre l'état abstrait et l'état physique du système à vérifier.

7. Connecter la réalisation physique à ERAIC. À ce niveau, quelques ajustements sont nécessaires dans certains modules d'interface VHDL pour répondre par exemple à la structure exacte de l'état du système à vérifier.

8. Connecter les deux implémentations abstraite et physique. C'est à ce niveau que sont développés les opérateurs de sur-définition en Java. C'est par le biais de ces derniers que TLC récupère les calculs réalisés par l'implémentation physique placés au préalable dans la mémoire cache logicielle sous forme d'un graphe de transition.

9. Vérifier que l'implémentation physique raffine l'implémentation de référence.

LISTE DE RÉFÉRENCES

[1] VIS - Verification Interacting with Synthesis. URL http://vlsi.colorado.edu/~vis/.

[2] El Mostapha Aboulhamid, Younès Karkouri et Eduard Cerny. On the generation of test patterns for multiple faults. *J. Electron. Test.*, 4(3):237–254, 1993. ISSN 0923-8174.

[3] Accellera. URL http://www.accellera.org/.

[4] Prathima Agrawal, Vishwani Agrawal et Sharad Seth. Generating Tests for Delay Faults in Nonscan Circuits. *IEEE Des. Test*, 10(1):20–28, 1993. ISSN 0740-7475.

[5] C. Ahlschlager et D. Wilkins. Using Magellan to Diagnose Post-Silicon Bugs. Rapport technique 3, Synopsys Verification Avenue Technical Bulletin, September 2004.

[6] Behzad Akbarpour, Amr T. Abdel-Hamid, Sofiène Tahar et John Harrison. Verifying a Synthesized Implementation of IEEE-754 Floating-Point Exponential Function using HOL. *The Computer Journal*, 2009.

[7] N. Alachiotis, S.A. Berger et A. Stamatakis. Efficient pc-fpga communication over gigabit ethernet. Dans *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pages 1727 –1734, 29 2010-july 1 2010.

[8] Altera. URL http://www.altera.com/.

[9] Roy Armoni, Limor Fix, Alon Flaisher, Rob Gerth, Boris Ginsburg, Tomer Kanza, Avner Landver, Sela Mador-Haim, Eli Singerman, Andreas Tiemeyer, Moshe Y. Vardi et Yael Zbar. The ForSpec Temporal Logic : A New Temporal Property-Specification Language. Dans *TACAS '02 : Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 296–211, London, UK, 2002. Springer-Verlag. ISBN 3-540-43419-4.

[10] Roy Armoni, Dmitry Korchemny, Moshe Y. Vardi et Yael Zbar. Deterministic Dynamic Monitors for Linear-Time Assertions. Dans *Formal Approaches to Software Testing and Runtime Verification*, pages 163–177. Springer Berlin / Heidelberg, 2006. ISBN 978-3-540-49699-1.

[11] Brian Bailey. Debugging FPGAs at full speed. *EE Times : EE Life Blogs*, 2010. URL `http://www.eetimes.com/electronics-blogs`.

[12] Brian Bailey. Emulator, accelerator, prototype what's the difference ? *EE Times : EE Life Blogs*, 2010. URL `http://www.eetimes.com/electronics-blogs`.

[13] Brian Bailey. To emulate or prototype ? *EE Times : EE Life Blogs*, 2010. URL `http://www.eetimes.com/electronics-blogs`.

[14] Leila Barakatain, Sofiène Tahar, Jean Lamarche et Jean-Marc Gendreau. Practical approaches to the verification of a telecom megacell using FormalCheck. Dans *GLSVLSI '01 : Proceedings of the 11th Great Lakes symposium on VLSI*, pages 1–6, New York, NY, USA, 2001. ACM. ISBN 1-58113-351-0.

[15] B. Batson et L. Lamport. High-level specifications : Lessons from industry. Dans *Proceedings of the First International Symposium on Formal Methods for Components and Objects*, Leiden, The Netherlands, 2003. URL `citeseer.ist.psu.edu/batson03highlevel.html`.

[16] Daniel K. Beece, George Deibert, Georgina Papp et Frank Villante. The IBM engineering verification engine. Dans *DAC '88 : Proceedings of the 25th ACM/IEEE conference on Design automation*, pages 218–224, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press. ISBN 0-8186-8864-5.

[17] Robert Beers. Pre-RTL formal verification : an intel experience. Dans *DAC '08 : Proceedings of the 45th annual conference on Design automation*, pages 806–811, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-115-6.

[18] Janick Bergeron, Eduard Cerny, Alan Hunter et Andy Nightingale. *Verification Methodology Manual for SystemVerilog*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005. ISBN 0387255389.

[19] Jayanta Bhadra, Magdy S. Abadir, Li-C. Wang et Sandip Ray. A Survey of Hybrid Techniques for Functional Verification. *IEEE Des. Test*, 24(2):112–122, 2007. ISSN 0740-7475.

[20] Robert K. Brayton, Gary D. Hachtel, Alberto Sangiovanni-Vincentelli1, Fabio Somenzi, Adnan Aziz, Szu Tsung Cheng, Stephen Edwards, Sunil Khatri, Yuji Kukimoto, Abelardo Pardo, Shaz Qadeer, Rajeev K. Ranjan, Shaker Sarwary, Thomas R. Staple, Gitanjali Swamy et Tiziano Villa. VIS : a system for verification and synthesis. Dans Springer Verlag, éditeur, *Pocceedings of the 8th International Conference om Computer Aided Verification CAV*, New Brunswick, NJ, USA, 1996.

[21] L. Cai, D. Gajski, P. Kritzinger et M. Olivares. Top-Down System Level Design Methodology Using SpecC, VCC and SystemC. Dans *DATE '02 : Proceedings of the conference on Design, automation and test in Europe*, page 1137, Washington, DC, USA, 2002. IEEE Computer Society.

[22] Lukai Cai et Daniel Gajski. Transaction level modeling : an overview. Dans *CODES+ISSS '03 : Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 19–24, New York, NY, USA, 2003. ACM. ISBN 1-58113-742-7.

[23] P. Camurati et P. Prinetto. Formal Verification of Hardware Correctness : Introduction and Survey of Current Research. *IEEE Computer*, 21(7):8 – 19, 1988.

[24] Michael J. C.Gordon, James Reynolds, Warren A. Hunt et Matt Kaufmann. An Integration of HOL and ACL2. Dans *FMCAD '06 : Proceedings of the Formal Methods in Computer Aided Design*, pages 153–160, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2707-8.

[25] Ken Chapman. UG129(v1.1) : PicoBlaze 8-bit Embedded Microcontroller User Guide. Rapport technique, June 10 2004. URL `http://www.xilinx.com/support/documentation`.

[26] Yen-Kuang Chen et S. Y. Kung. Trend and Challenge on System-on-a-Chip Designs. *J. Signal Process. Syst.*, 53(1-2):217–229, 2008. ISSN 1939-8018.

[27] Koen Claessen et Gordon J. Pace. An embedded language approach to teaching hardware compilation. *SIGPLAN Not.*, 37(12):35–46, 2002. ISSN 0362-1340.

[28] Edmund Clarke, Daniel Kroening et Karen Yorav. Behavioral consistency of C and verilog programs using bounded model checking. Dans *DAC '03 : Proceedings of*

the 40th conference on Design automation, pages 368–371, New York, NY, USA, 2003. ACM. ISBN 1-58113-688-9.

[29] György Csertán, István Majzik, András Pataricza et Susan Allmaier. Reachability Analysis of Petri-nets by FPGA Based Accelerators. Dans *Proc. Design and Diagnostics of Electronic Circuits and Systems Workshop (DDECS'98)*, pages 307 – 312, 1998. URL `mycite.omikk.bme.hu/doc/16325.pdf`.

[30] O. Dahmoune et A. Tsikhanovich. Le jeu du démineur adapté sur FPGA. Rapport technique, Université de Montreal (DIRO), April 2004.

[31] Ouiza Dahmoune. La synthèse de circuits par programmation fonctionnelle. Rapport technique, Université de Montreal (DIRO), Dec 2004.

[32] Ouiza Dahmoune. Le Co-processeur VHDL. Rapport technique, Institut National de Recherche Scientifique (INRS), Août 2005.

[33] Ouiza Dahmoune et Robert De B Johnston. An Embedded Reachability Analyzer And Invariant Checker (ERAIC). Dans *Proceedings of the 11th International Workshop on Microprocessor Test and Verification*, MTV '10, pages 47–50, Austin, TX, December 2010. IEEE Computer Society.

[34] Ouiza Dahmoune et Robert De B Johnston. Applying Model-Checking to Post-Silicon-Verification : Bridging the Specification-Realisation Gap . Dans *Proceedings of the International Conference on Reconfigurable Computing and FPGAs*, RECONFIG '10, pages 73–78, Cancun, Mexico, 2010. IEEE Computer Society.

[35] Ouiza Dahmoune et Robert De B Johnston. Connecting a Model-Checker to an FPGA Prototype : Bridging the Specification-Physical-Implementation Gap. Rapport technique, INRS(Centre Énergie Matériaux Télécommunications), May 2011.

[36] Ouiza Dahmoune et Robert De B Johnston. Model Checker to FPGA Prototype Commmunication Bottleneck Issue. Dans *Proceedings of the 12th International Workshop on Microprocessor Test and Verification*, MTV '11, pages 38–43, Austin, TX, December 2011. IEEE Computer Society.

[37] Markus Damm, Jan Haase, Christoph Grimm, Fernando Herrera et Eugenio Villar. Bridging MoCs in SystemC specifications of heterogeneous systems. *EURASIP J. Embedded Syst.*, 2008(3):1–16, 2008. ISSN 1687-3955.

[38] J. Darringer, E. Davidson, D. J. Hathaway, B. Koenemann, M. Lavin, J. K. Morrell, K. Rahmat, W. Roesner, E. Schanzenbach, G.Tellez et L. Trevillyan. EDA in IBM, Past, Present and Future. *IEEE Trans. Computer Aided Design.*, 19(12):1476–1497, 2000.

[39] Andrew DeOrio, Adam Bauserman et Valeria Bertacco. Post-silicon verification for cache coherence. Dans *ICCD*, pages 348–355, 2008.

[40] Edsger W. Dijkstra. Notes on Structured Programming. URL `http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF`. circulated privately, 1970.

[41] EDA. URL `http://www.eda.org/`.

[42] Laurent Fournier, Yaron Arbetman et Moshe Levinger. Functional verification methodology for microprocessors using the Genesys test-program generator. Dans *DATE '99 : Proceedings of the conference on Design, automation and test in Europe*, page 92, New York, NY, USA, 1999. ACM. ISBN 1-58113-121-6.

[43] Wiedijk Freek. *The Seventeen Provers of the World : Foreword by Dana S. Scott (Lecture Notes in Computer Science / Lecture Notes in Artificial Intelligence)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. ISBN 3540307044.

[44] Malay Ganai et Aarti Gupta. *SAT-Based Scalable Formal Verification Solutions (Series on Integrated Circuits and Systems)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007. ISBN 0387691669.

[45] V. Goncalves, J.T. de Sousa et F. Goncalves. A low-cost scalable pipelined reconfigurable architecture for simulation of digital circuits. Dans *Field Programmable Logic and Applications, 2005. International Conference on*, pages 481 – 486, 2005.

[46] G. Gopalakrishnan et C. Chou. The Post-silicon Verification Problem : Designing Limited Observability Checkers for Shared Memory Processors. Dans *Fifth Interna-*

tional Workshop on Designing Correct Circuits, pages 27–28. ETAPS 2004, March 2004.

[47] M. J. C. Gordon et T. F. Melham, éditeurs. *Introduction to HOL : a theorem proving environment for higher order logic*. Cambridge University Press, New York, NY, USA, 1993. ISBN 0-521-44189-7.

[48] M.J.C Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. Dans *Formal Aspects of VLSI Design*, pages 153–177, Amsterdam, 1986. eds., North-Holland Publishing.

[49] Aarti Gupta. Formal hardware verification methods : a survey. *Journal of Formal Methods in System Design*, 1(2-3):151–238, 1992. ISSN 0925-9856.

[50] Paul R. Halmos. *Naive Set Theory*. D. Van Nostrand Company Ltd., Princeton, N.J., 1960.

[51] John Harrison. Theorem Proving for Verification (Invited Tutorial). Dans *Proceedings of the 20th international conference on Computer Aided Verification*, pages 11–18, Berlin, Heidelberg, 2008. Springer-Verlag.

[52] Hamilton B. Carter ; Shankar G. Hemmady. *Metric Driven Design Verification : An Engineer's and Executive's Guide to First Pass Success*. Springer US, Secaucus, NJ, USA, 2007. ISBN 2007 ISBN 978-0-387-38151-0 (Print) 978-0-387-38152-7 (Online).

[53] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[54] Gerard J. Holzmann. *The SPIN Model Checker : Primer and Reference Manual*. Addison-Wesley, sep 2003.

[55] Paul Hudak, John Hughes, Simon Peyton Jones et Philip Wadler. A history of Haskell : being lazy with class. Dans *HOPL III : Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1–12–55, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-766-X.

[56] Naoki Iwasaki et Katsumi Wasaki. A Meta Hardware Description Language Melasy for Model-Checking Systems. Dans *ITNG '08 : Proceedings of the Fifth International Conference on Information Technology : New Generations*, pages 273–278, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3099-4.

[57] Christian Jacobi et Christoph Berg. Formal Verification of the VAMP Floating Point Unit. *Formal Methods in System Design*, 26(3):227266, 2005.

[58] Mohammad-Sadegh Jahanpour. *Compositional verification using interface recognizers/suppliers (irs)*. Thèse de doctorat, Universite de Montreal, Monreal, P.Q., Canada, Canada, 2001. Adviser-Eduard Cerny.

[59] Geert L.J.M. Janssen. *Logics for Digital Circuit Verification Theory, Algorithms, and Applications*. Thèse de doctorat, Eindhoven University, Netherlands, 1999. Adviser-Professor Jochen Jess.

[60] Robert De B Johnston et Ouiza Dahmoune. Overview of Applying Reachability Analysis to Verifying a Physical Microprocessor. Dans *in MTV'11 : Proceedings of the 12th International Workshop on Microprocessor Test and Verification*, MTV '11, pages 32–37, Austin, TX, December 2011. IEEE Computer Society.

[61] Peyton Jones et Simon L. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987. ISBN 013453333X.

[62] Jeffrey J. Joyce et Carl-Johan H. Seger. Linking BDD-based symbolic evaluation to interactive theorem-proving. Dans *DAC '93 : Proceedings of the 30th international conference on Design automation*, pages 469–474, New York, NY, USA, 1993. ACM. ISBN 0-89791-577-1.

[63] Reinhard Kahle. The Seventeen Provers of the World. *Studia Logica*, 87(2-3):369–374, 2007.

[64] Roope Kaivola et Mark Aagaard. Divider Circuit Verification with Model Checking and Theorem Proving. Dans *TPHOLs '00 : Proceedings of the 13th International Conference on Theorem Proving in Higher Order Logics*, pages 338–355, London, UK, 2000. Springer-Verlag. ISBN 3-540-67863-8.

[65] C. Kern et M. R. Greenstreet. Formal verification in hardware design : a survey. *ACM Trans. Des. Autom. Electron. Syst.*, 4(2):123–193, 1999. ISSN 1084-4309.

[66] Namseung Kim, Hoon Choi, Seungwang Lee, Seungwang Lee, In-Cheol Park et Chong-Min Kyung. Virtual Chip : Making Functional Models Work On Real Target Systems. Dans *In IEEE/ACM Design Automation Conference*, pages 170–173, 1998.

[67] Yong Chang Kim, Kewal K. Saluja et Vishwani D. Agrawal. Multiple Faults : Modeling, Simulation and Test. Dans *ASP-DAC '02 : Proceedings of the 2002 conference on Asia South Pacific design automation/VLSI Design*, page 592, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1441-3.

[68] Young-Il Kim, Wooseung Yang, Young-Su Kwon et Chong-Min Kyung. Communication-efficient hardware acceleration for fast functional simulation. Dans *DAC '04 : Proceedings of the 41st annual conference on Design automation*, pages 293–298, New York, NY, USA, 2004. ACM. ISBN 1-58113-828-8.

[69] Ho Fai Ko et Nicola Nicolici. Algorithms for state restoration and trace-signal selection for data acquisition in silicon debug. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 28(2):285–297, 2009. ISSN 0278-0070.

[70] Heon-Mo Koo et Prabhat Mishra. Functional test generation using design and property decomposition techniques. *ACM Trans. Embed. Comput. Syst.*, 8(4):1–33, 2009. ISSN 1539-9087.

[71] Peter A. Krauss, Andreas Ganz et Kurt J. Antreich. Distributed Test Pattern Generation for Stuck-At Faults in Sequential Circuits. *J. Electron. Test.*, 11(3):227–245, 1997. ISSN 0923-8174.

[72] Jai Kumar, Catherine Ahlschlager et Peter Isberg. Post-silicon verification methodology on Sun's UltraSPARC T2. Dans *HLDVT '07 : Proceedings of the 2007 IEEE International High Level Design Validation and Test Workshop*, page 47, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 978-1-4244-1480-2.

[73] Leslie Lamport. TLA - The Temporal Logic of Actions. URL `http://research.microsoft.com/users/lamport/tla/tla.html`.

[74] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, mai 1994. ISSN 0164-0925. URL http://doi.acm.org/10.1145/177492.177726.

[75] Leslie Lamport. *Specifying Systems : The TLA+ Language and Tools for Hardware and Software Engineers*. Addison Wesley, jul 2002.

[76] Leslie Lamport, John Matthews, Mark Tuttle et Yuan Yu. Specifying and verifying systems with TLA+. Dans *EW10 : Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 45–48, New York, NY, USA, 2002. ACM.

[77] Lasso. URL http://www.iro.umontreal.ca/~lablasso.

[78] Jae-Gon Lee et Chong-Min Kyung. PrePack : Predictive Packetizing Scheme for Reducing Channel Traffic in Transaction-Level Hardware/Software Co-Emulation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 25(10):1935 – 1949, 2006.

[79] Seungjong Lee, Ando Ki, In-Cheol Park et Chong-Min Kyung. Interface synthesis between software chip model and target board. *J. Syst. Archit.*, 48(1-3):49–57, 2002. ISSN 1383-7621.

[80] Howard Mao. A next-gen FPGA-based SoC verification platform. *EE Times : Design, Programmable Logic DesignLine*, 2010. URL http://www.eetimes.com/design/programmable-logic.

[81] I. Mavroidis et I. Papaefstathiou. Efficient testbench code synthesis for a hardware emulator system. Dans *DATE '07 : Proceedings of the conference on Design, automation and test in Europe*, pages 888–893, San Jose, CA, USA, 2007. EDA Consortium. ISBN 978-3-9810801-2-4.

[82] I. Mavroidis et I. Papaefstathiou. Accelerating hardware simulation : Testbench code emulation. Dans *ICECE Technology, 2008. FPT 2008. International Conference on*, pages 129 –136, dec. 2008.

[83] I Mavroidis et I Papaefstathiou. Accelerating Emulation and Providing Full Chip Observability and Controllability at Run-Time. *Design Test of Computers, IEEE*, PP (99):1, 2009. ISSN 0740-7475.

[84] K. L. McMillan. *Symbolic Model Checking, An approach to the state explosion problem*. Thèse de doctorat, Carnegie Mellon University, 1992.

[85] K. L. McMillan. The SMV Language. Rapport technique, Cadence Berkeley Labs, 2001. URL `http://santos.cis.ksu.edu/smv-doc/language/language.html`.

[86] B. Meenakshi. Formal verification. *Resonance*, 10(5):26–38, 2005.

[87] Elliott Mendelson. *Introduction to Mathematical Logic*. Chapman & Hall/CRC, 5th édition, 2009. ISBN 1584888768, 9781584888765.

[88] Stephan Merz. Model Checking : A Tutorial Overview. Dans *MOVEP*, pages 3–38, 2000. URL `citeseer.ist.psu.edu/merz00model.html`.

[89] Subhasish Mitra, Sanjit A. Seshia et Nicola Nicolici. Post-silicon validation opportunities, challenges and recent advances. Dans *DAC '10 : Proceedings of the 47th Design Automation Conference*, pages 12–17, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0002-5.

[90] A. Molina et O. Cadenas. Functional verification : approaches and challenges. *Latin American Applied Research*, 37(1):65–69, 2007. ISSN 0327-0793.

[91] Gabe Moretti, Tom Anderson, Janick Bergeron, Ashish Dixit, Peter Flake, Tim Hopes et Ramesh Narayanaswamy. Your core— my problem ? (panel session) : integration and verification of IP. Dans *DAC '01 : Proceedings of the 38th conference on Design automation*, pages 170–171, New York, NY, USA, 2001. ACM. ISBN 1-58113-297-2.

[92] Jayant Nagda. High Level Functional Verification Closure. Dans *ICCD '02 : Proceedings of the 2002 IEEE International Conference on Computer Design : VLSI in Computers and Processors (ICCD'02)*, page 91, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1700-5.

[93] Jayant Nagda. 'Smart' verification moves beyond SystemVerilog 3.0. *EE Times : Design News*, 2002. URL `http://www.eetimes.com/story/OEG20020927S0057`.

[94] A. Nahir, A. Ziv et al. Bridging Pre-Silicon Verification and Post-Silicon Validation. Dans *DAC '10 : Proceedings of the 47th Design Automation Conference*, pages 94–95, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0002-5.

[95] Yuichi Nakamura, Kouhei Hosokawa, Ichiro Kuroda, Ko Yoshikawa et Takeshi Yoshimura. A fast hardware/software co-verification method for system-on-a-chip by using a C/C++ simulator and FPGA emulator with shared register communication. Dans *DAC '04 : Proceedings of the 41st annual conference on Design automation*, pages 299–304, New York, NY, USA, 2004. ACM. ISBN 1-58113-828-8.

[96] Eriko Nurvitadhi, Jumnit Hong et Shih-Lien Lu. Active Cache Emulator. *Very Large Scale Integration (VLSI) Systems, IEEE transactions on*, 16(3):229 – 240, 2008.

[97] John J. O'Donnell. From Transistors to Computer Architecture : Teaching Functional Circuit Specification in Hydra. Dans *FPLE '95 : Proceedings of the First International Symposium on Functional Programming Languages in Education*, pages 195–214, London, UK, 1995. Springer-Verlag. ISBN 3-540-60675-0.

[98] OPENCORES.ORG. Technical Reference Manual : WISHBONE Public Domain Library for VHDL, 2001.

[99] *Specification for the : WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores, Rev :B.3*. OPENCORES.ORG, September 7 2002. URL `http://www.opencores.org/downloads/wbspec_b3.pdf`.

[100] Sam Owre, John M. Rushby et Natarajan Shankar. PVS : A Prototype Verification System. Dans *CADE-11 : Proceedings of the 11th International Conference on Automated Deduction*, pages 748–752, London, UK, 1992. Springer-Verlag. ISBN 3-540-55602-8.

[101] Sung-Boem Park, Ted Hong et Subhasish Mitra. Post-silicon bug localization in processors using instruction footprint recording and analysis (IFRA). *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 28(10):1545–1558, 2009. ISSN 0278-0070.

[102] Sung-Boem Park et Subhasish Mitra. Ifra : instruction footprint recording and analysis for post-silicon bug localization in processors. Dans *DAC '08 : Proceedings*

of the 45th annual Design Automation Conference, pages 373–378, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-115-6.

[103] Sung-Boem Park et Subhasish Mitra. Post-silicon bug localization for processors using IFRA. *Commun. ACM*, 53(2):106–113, 2010. ISSN 0001-0782.

[104] Sungwoo Park, Jinha Kim et Hyeonseung Im. Functional netlists. Dans *ICFP '08 : Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 353–366, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-919-7.

[105] B. Peischl et F. Wotawa. Automated Source-Level Error Localization in Hardware Designs. *IEEE Des. Test*, 23(1):8–19, 2006. ISSN 0740-7475.

[106] Patrick Pelgrims, Dries Driessens et Tom Tierens. Titel : Embedded Systeemontwerp op basis van Soft- en Hardcore FPGA's, 2003. URL `http://citeseerx.ist.psu.edu/viewdoc`.

[107] G.F. Pfister. The IBM Yorktown Simulation Engine. *Proc. IEEE*, 74(6):850 – 860, 1986.

[108] Sandip Ray et Warren A. Hunt. Connecting pre-silicon and post-silicon verification. Dans *FMCAD*, pages 160–163, 2009.

[109] Sandip Ray et Rob Sumners. Combining Theorem Proving with Model Checking through Predicate Abstraction. *IEEE Des. Test*, 24(2):132–139, 2007. ISSN 0740-7475.

[110] A.W. Ruan, Y.B. Liao, P. Li, Y.W. Wang et W.C. Li. A stream-mode based hw/sw co-emulation system for soc test and verification. Dans *Testing and Diagnosis, 2009. ICTD 2009. IEEE Circuits and Systems International Conference on*, pages 1 –4, april 2009.

[111] Carl-Johan H. Seger et Jeffrey J. Joyce. A Two-Level Formal Verification Methodology using HOL and COSMOS. Dans *CAV '91 : Proceedings of the 3rd International Workshop on Computer Aided Verification*, pages 299–309, London, UK, 1992. Springer-Verlag. ISBN 3-540-55179-4.

[112] Sanjit A. Seshia, Wenchao Li et Subhasish Mitra. Verification-guided soft error resilience. Dans *DATE '07 : Proceedings of the conference on Design, automation and test in Europe*, pages 1442–1447, San Jose, CA, USA, 2007. EDA Consortium. ISBN 978-3-9810801-2-4.

[113] Deian Tabakov, Moshe Y. Vardi, Gila Kamhi et Eli Singerman. A temporal language for SystemC. Dans *FMCAD '08 : Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*, pages 1–9, Piscataway, NJ, USA, 2008. IEEE Press. ISBN 978-1-4244-2735-2.

[114] Bart Vermeulen, Rolf Kühnis, Jeff Rearick, Neal Stollon et Gary Swoboda. Overview of Debug Standardization Activities. *IEEE Des. Test*, 25(3):258–267, 2008. ISSN 0740-7475.

[115] Bruce Wile, John Goss et Wolfgang Roesner. *Comprehensive Functional Verification*. Morgan Kaufman Publishers, Inc., San Francisco, CA, USA, 2005. ISBN 0-12-751803-1.

[116] Eric Van Wyk et Mats Per Erik Heimdahl. Flexibility in modeling languages and tools : a call to arms. *International Journal on Software Tools for Technology Transfer (STTT)*, 2009.

[117] Xilinx. URL `http://www.xilinx.com/`.

[118] Haiyan Xiong, Paul Curzon, Sofiène Tahar et Ann Blandford. Providing a formal linkage between MDG and HOL. *Form. Methods Syst. Des.*, 30(2):83–116, 2007. ISSN 0925-9856.

[119] Ying Xu, Xiaoyu Song, Eduard Cerny et Otmane Ait Mohamed. Model Checking for a First-Order Temporal Logic Using Multiway Decision Graphs (MDGs). *The Computer Journal*, 47(1):71–84, 2004. ISSN 0925-9856.