Amélioration du processus de vérification des architectures générées à l'aide d'outils de synthèse de haut-niveau

Aurelien Ribon

▶ **To cite this version:**

Aurelien Ribon. Amélioration du processus de vérification des architectures générées à l'aide d'outils de synthèse de haut-niveau. Micro et nanotechnologies/Microélectronique. Université Bordeaux 1, 2012. Français. NNT : 2012BOR14719 . tel-01264247

HAL Id: tel-01264247
https://hal.science/tel-01264247

Submitted on 29 Jan 2016

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

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

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ BORDEAUX 1

ÉCOLE DOCTORALE DES SCIENCES PHYSIQUES ET DE L'INGÉNIEUR

Par Aurélien RIBON

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ ÉLECTRONIQUE

AMÉLIORATION DU PROCESSUS DE VÉRIFICATION DES ARCHITECTURES GÉNÉRÉES À L'AIDE D'OUTILS DE SYNTHÈSE DE HAUT-NIVEAU

Directeurs de recherche : MM. DALLET Dominique et LE GAL Bertrand

Soutenue le 17 décembre 2012

Devant la commission d'examen formée de :

MME. ANGHEL L.	Professeur des Universités à l'Université de Grenoble	Rapporteur
M. BOSSUET L.	Maître de Conférences (HDR) à l'Université de Saint Etienne	Examinateur
M. CASSEAU E.	Professeur des Universités à l'Université de Rennes I	Examinateur
M. DALLET D.	Professeur des Universités à l'Institut Polytechnique de Bordeaux	Examinateur
M. JÉGO C.	Professeur des Universités à l'Institut Polytechnique de Bordeaux	Examinateur
M. LAGADEC L.	Professeur à l'ENSTA Bretagne	Rapporteur
M. LE GAL B.	Maître de Conférences à l'Institut Polytechnique de Bordeaux	Examinateur

Résumé

L'impressionnante augmentation de la complexité des systèmes matériels durant les trois dernières décennies a bouleversé toutes les étapes de leur cycle de développement. Les méthodes de conception ont connu et connaissent encore des changements majeurs, nécessaires au maintien d'un temps de mise sur le marché, contraint par la compétition internationale, toujours plus court.

Des systèmes plus complexes sont aussi synonyme d'erreurs potentielles plus nombreuses, de cas particuliers plus difficiles à déceler, et de simulations plus longues et coûteuses. La vérification des systèmes matériels nécessite de plus en plus de ressources, et est depuis longtemps le poste de dépense principal dans la conception de circuits. La complexité des systèmes augmentant, le coût d'une erreur non détectée jusqu'à la mise en production est devenu prohibitif. Ainsi, le processus de vérification est divisé en nombreuses étapes intervenant à tous les moments de la conception : mesure de l'exécution des modèles comportementaux, simulation des descriptions RTL, analyse formelle des algorithmes, utilisation d'assertions, etc. Les méthodes de vérification ont beaucoup évolué avec le temps, afin de suivre les avancées des méthodes de conception. Certaines méthodes comme la vérification par assertions ont pris une ampleur telle qu'elles se sont imposées comme incontournables pour détecter au plus tôt les erreurs et leurs sources.

Ainsi, les travaux présentés dans ce manuscrit ciblent l'amélioration du processus de vérification par assertions, afin d'offrir un important gain de temps aux concepteurs. Deux contributions distinctes sont détaillées. La première s'intéresse à la transformation des assertions booléennes contenues dans une description algorithmique en assertions temporelles équivalentes dans la description RTL générée par synthèse de haut-niveau (*High-Level Synthesis*, HLS). Les assertions ainsi transformées sont alors utilisables pendant les simulations des architectures générées. La deuxième contribution, quant à elle, a pour contexte la vérification des systèmes matériels en fonctionnement réel. Elle détaille le processus de synthèse d'un gestionnaire d'erreurs matériel, dont le rôle est de sauvegarder et d'archiver le contexte d'exécution courant lorsqu'une erreur est détectée. Il est ainsi plus facile de déterminer la cause d'une erreur et de remonter à sa source. Les erreurs et leurs contextes sont archivées sous forme de rapports dans une mémoire accessible par le système ou le concepteur. Le fonctionnement d'un circuit peut ainsi être analysé à tout moment, sans nécessiter de sonde ou d'analyseur logique intégré.

Mots-clés : architecture, assertion, debug, erreur, HLS, moniteur, rapport, vérification

Abstract

Improvement of the Verification Process of Architectures Generated by High-Level Synthesis Tools

The fast growing complexity of hardware circuits, during the last three decades, has changed every step of their development cycle. Design methods evolved a lot, and this evolution was necessary to cope with an always shorter time-to-market, mainly driven by the international competition.

An increased complexity also means more errors, harder to find corner-cases, and more long and expensive simulations. The verification of hardware systems requires more and more resources, and is the main cost factor of the whole development of a circuit. Since the complexity of any system increases, the cost of an error undetected until the foundry step became prohibitive. Therefore, the verification process is divided between multiple steps involved at every moment of the design process : comparison of models behavior, simulation of RTL descriptions, formal analysis of algorithms, assertions usage, etc. The verification methodologies evolved a lot, in order to follow the progress of design methods. Some methods like the Assertion-Based Verification became so important that they are now widely adopted among the developers community, providing near-source error detection.

Thus, the work described here aims at improving the assertion-based verification process, in order to offer a consequent timing improvment to designers. Two contributions are detailed. The first one deals with the transformation of Boolean assertions found in algorithmic descriptions into equivalent temporal assertions in the RTL description generated by high-level synthesis (HLS) methodologies. Therefore, the assertions are usable during the simulation process of the generated architectures. The second contribution targets the verification of hardware systems in real-time. It details the synthesis process of a hardware error manager, which has to save and serialize the execution context when an error is detected. Thus, it is easier to understand the cause of an error and to find its source. The errors and their contexts are serialized as reports in a memory readable by the system or directly by the designer. The behavior of a circuit can be analyzed without requiring any probe or integrated logic analyzer.

Keywords : architecture, assertion, debug, error, HLS, monitor, report, verification

Laboratoire de l'Intégration du Matériau au Système - IMS - UMR 5218
351 Cours de la Libération
33405 Talence CEDEX
France

Sommaire

Chapitre 1

Contexte et motivations

Sommaire

L'augmentation de la capacité d'intégration des circuits a permis le développement de systèmes de plus en plus complexes. De cette complexité sont nés des besoins conséquents en terme de méthodes de conception et de vérification.

Ce chapitre présente le contexte dans lequel s'inscrit cette thèse, ainsi que les motivations qui ont principalement dirigé les recherches poursuivies. Plusieurs notions importantes y sont présentées, amenant à la problématique principale.

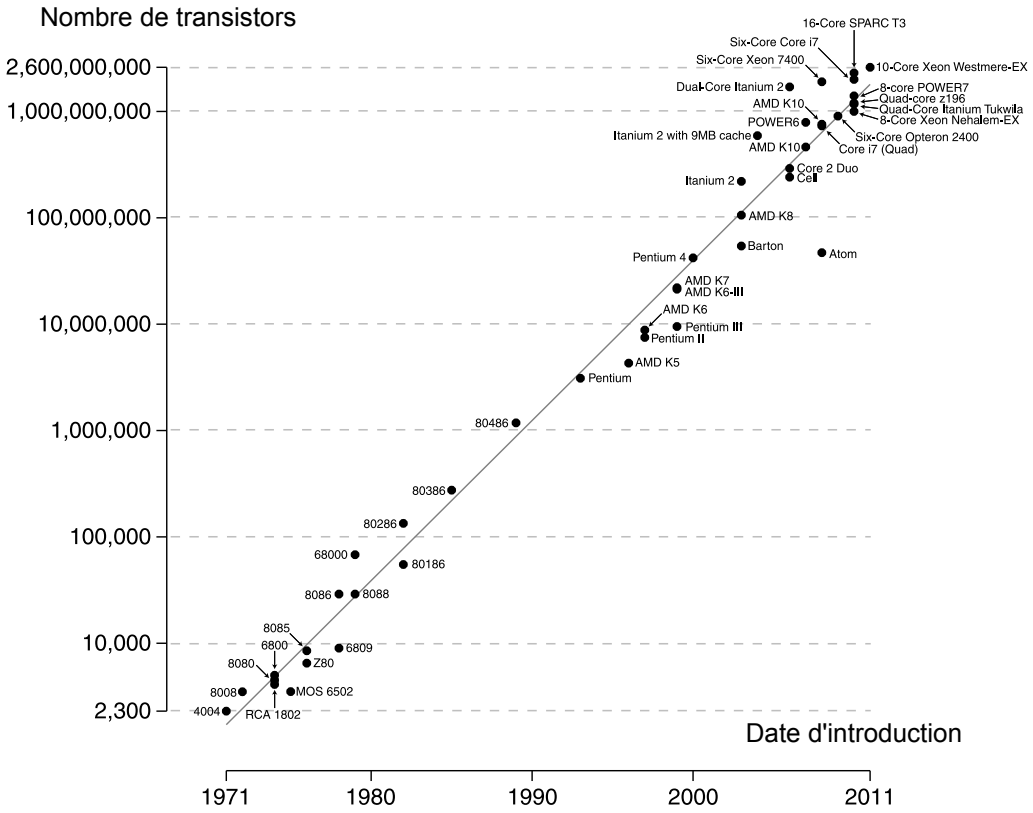

FIGURE 1 – La loi de Moore, appliquée aux microprocesseurs

1 Introduction

Le premier microprocesseur, l'*Intel 4004* [159], a été introduit en 1971 et était composé de 2300 transistors. L'un des derniers processeurs d'Intel (introduit en 2011), le *Xeon Westmere-EX* [165], dépasse largement les deux milliards de transistors avec la présence de dix coeurs de calcul. Cette augmentation de la complexité des circuits est possible grâce à la réduction constante de la finesse de gravure des circuits. En effet, cette dernière est passée de la dizaine de micromètres à 22 nanomètres en quarante ans [83]. L'augmentation continue de la capacité d'intégration des circuits lors de ces dernières décennies a suivi la «loi de Moore» [167], énoncée en 1965 et précisée en 1975. L'ingénieur à son origine, Gordon E. Moore alors en poste chez *Fairchild Semiconductor*, fut l'un des trois fondateurs d'*Intel*. Cette «loi» empirique illustrée en figure 1 imaginait à l'époque l'évolution des capacités d'intégration en nombre de transistors. Elle postule que le nombre de transistors des microprocesseurs double tous les deux ans. La loi de Moore fut étonnamment juste jusqu'à présent. Le tableau 1 est une retranscription de la feuille de route [5] d'Intel concernant la finesse de gravure et la capacité d'intégration à atteindre pour les dix prochaines années. Cette estimation suit toujours la loi de Moore, en faisant doubler la capacité d'intégration tous les deux ans.

Technology Outlook								
High Volume Manufacturing	2008	2010	2012	2014	2016	2018	2020	2022
Technology Node (nm)	45	32	22	16	11	8	6	4
Integration Capacity (BT)	8	16	32	64	128	256	512	1024

TABLE 1 – Feuille de route d'*Intel* concernant la finesse de gravure et la capacité d'intégration de ses circuits (en milliards de transistors)

1.1 Évolution des applications

L'évolution des capacités d'intégration a pour moteur principal l'évolution des applications, elle-même menée par les attentes et les besoins des utilisateurs. Les changements opérés dans les systèmes de téléphonie mobile en sont les meilleurs exemples. Au début uniquement prévus pour recevoir et transmettre la voix des usagers, ils permettent désormais le visionnage de vidéos haute-définition et l'exécution de jeux 3D complexes. Les besoins grandissants des utilisateurs dynamisent l'innovation et se trouvent en même temps amplifiés par cette innovation. Cela mène les développeurs à concevoir des applications toujours plus puissantes, et les fondeurs à augmenter encore les capacités d'intégration de leurs circuits pour répondre à ces besoins. Les limites physiques de la taille des téléphones, qui doivent servir le confort de l'utilisateur, imposent des contraintes immuables aux constructeurs, alors que les applications à embarquer dans un seul appareil sont de plus en plus nombreuses et complexes. De même, la progression plus lente des capacités des batteries oblige les concepteurs à réduire de plus en plus la consommation des circuits pour maintenir une durée de vie nomade acceptable.

Toutes ces contraintes ont rapidement poussé l'avènement des systèmes sur puce, ou *System on Chip* (SoC). Ils intègrent de très nombreux composants, autrefois indépendants les uns des autres, au sein d'un même circuit. Processeur, mémoire et accélérateurs matériels s'y trouvent réunis, complétés par de multiples périphériques intégrés. Par exemple, la famille de SoC Omap [108] du fondeur *Texas Instruments* (cf. figure 2) intègre deux coeurs de processeur Cortex A9, un processeur graphique dédié (GPU), un processeur de traitement de signal, des coeurs de cryptographie, plusieurs zones de mémoire ainsi que la quasi totalité des types de contrôleur de communication (USB, I^2C, SPI et même HDMI). Une seule puce suffit pour un système complet, là ou il en fallait une multitude quelques années auparavant. Et ce type de système intégré se retrouve à bas coût dans un grand nombre de téléphones portables, propulsant leur fonction de simples appareils de télécommunication à des ensembles multimédia complets. Décodage de vidéo full-HD, jeux 3D disposant des dernières technologies de *shaders*, sortie audio 5.1, sortie TV, lecture de cartes SD, photographie, enregistrement de vidéos... les applications les plus complexes s'insèrent désormais dans des systèmes à la base uniquement prévus pour le transfert de voix et de courts blocs de texte.

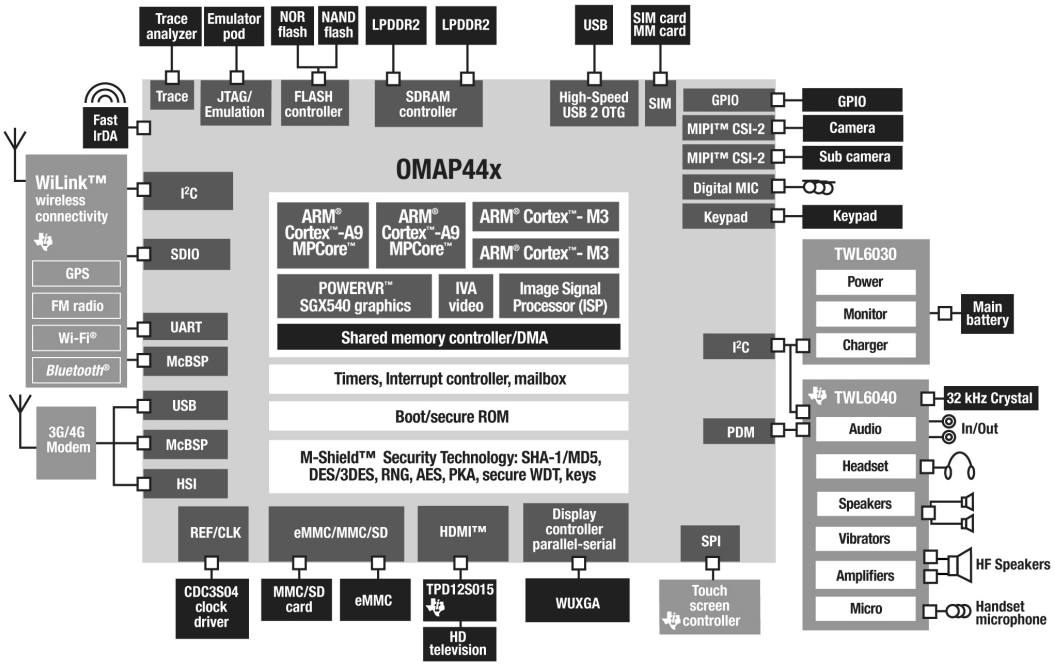

FIGURE 2 – Architecture du SoC OMAP44x de Texas Instruments

Pourtant, si les systèmes ont des ambitions que rien ne semble arrêter, leur conception devient très complexe. Or, les fenêtres de mise sur le marché des produits, uniquement articulées par la concurrence des entreprises, ne sont pas pour autant plus grandes maintenant qu'il y a dix ans. Les possibilités offertes par la technologie doublant tous les deux ans, les équipes de conception devraient alors aussi théoriquement doubler au même rythme. Comme cela est loin d'être viable, ou même possible, ce sont alors les méthodes de conception et de vérification qui sont concernées, et qui connaissent simultanément une évolution continue.

La figure 3 présente une partie des évolutions de ces méthodes au cours du temps. Ces innovations adoptées au fur et à mesure par l'industrie permettent la conception de systèmes de plus en plus complexes. Sans cet ajout constant de nouvelles techniques et de nouveaux outils, le coût de développement de ces systèmes exploserait en quelques années (coût représenté sur la figure par la ligne en pointillés).

1.2 Évolution des méthodes de conception

La pression du *time-to-market*, couplé à la complexité des systèmes à concevoir, ont rapidement rendu l'évolution des méthodes de conception nécessaire. Concevoir directement un circuit au niveau physique à partir d'un cahier des charges et espérer tenir les délais n'est

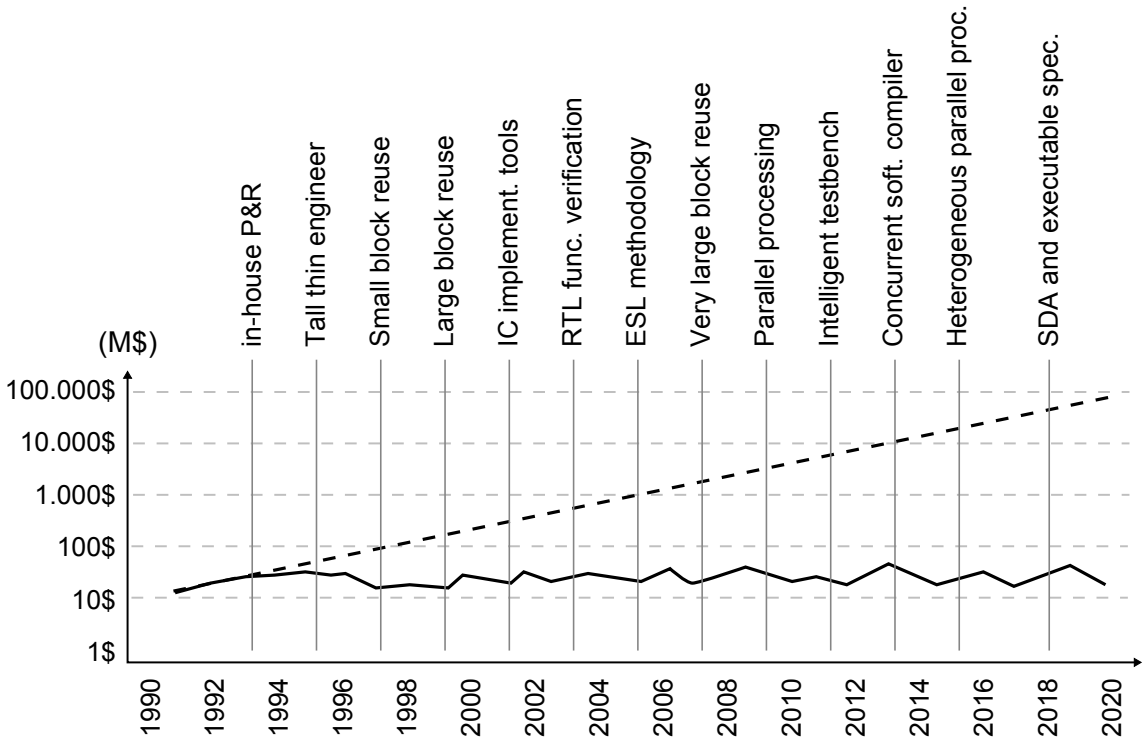

FIGURE 3 – Évolution des méthodes de conception et de vérification

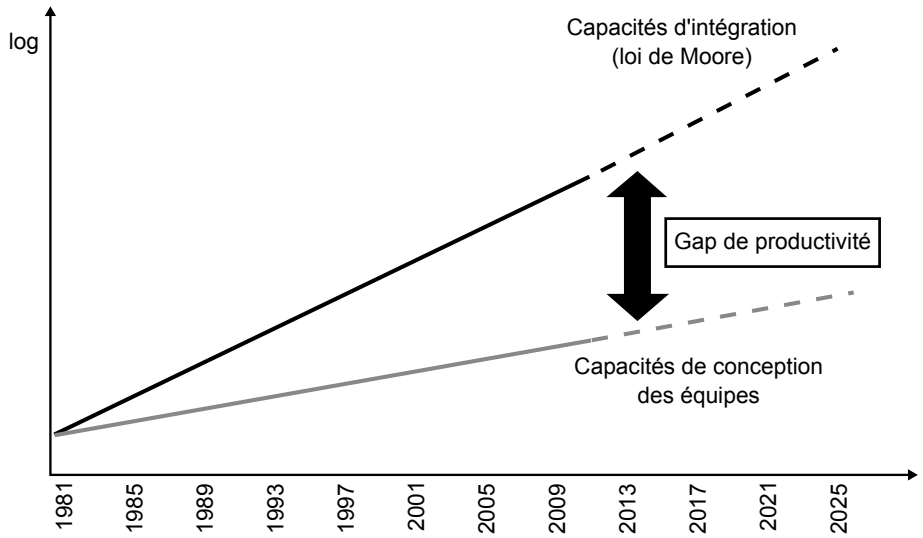

FIGURE 4 – Gap de productivité (ITRS 2011 [152])

plus envisageable depuis longtemps. Comme le montre la figure 4, la capacité d'intégration offerte par la technologie augmente bien plus vite que les capacités de développement des équipes de conception. Ce gap de productivité ne peut être comblé que par l'évolution des méthodes et des outils de conception.

L'évolution la plus visible des méthodes de conception consiste en l'augmentation du niveau d'abstraction pour la modélisation des systèmes. Ainsi, la conception d'un circuit ou

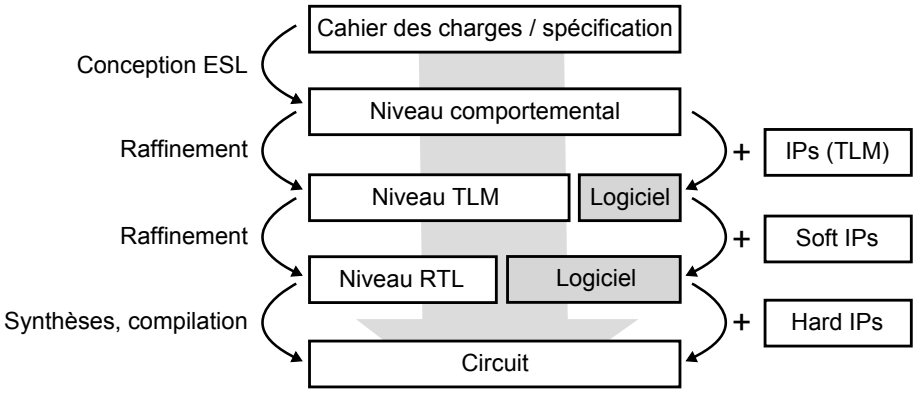

FIGURE 5 – Flot de conception (simplifié)

d'une application est désormais divisée en différents niveaux d'abstraction, illustrés schématiquement sur la figure 5. La description comportementale modélise au niveau le plus abstrait les algorithmes principaux, au contraire de la description matérielle qui précise les opérations effectuées à chaque cycle d'horloge. On parle de méthode de développement au niveau système (*Eletronic System-Level (ESL) design*). Entre le niveau comportemental et le niveau matériel, on trouve depuis quelques années le niveau TLM (*Transaction Level Model*) [69, 146]. Ce niveau d'abstraction, standardisé en 2011 par IEEE [133], permet une transition plus facile du modèle algorithmique au modèle matériel en découpant cette transition en plusieurs sous-niveaux d'abstraction. Les choix d'implémentation des différents algorithmes et des communications entre composants se font au fur et à mesure des passages d'un niveau d'abstraction à un autre, et non plus d'un seul coup lors de l'implémentation directe du circuit à partir de sa description comportementale.

Toutefois, plus le niveau d'abstraction est bas, et plus les contraintes de conception sont importantes. Ainsi, le raffinement d'un niveau à l'autre prend un temps de plus en plus conséquent lorsque l'on s'approche de la description matérielle. De ce fait, le flot de conception d'un système a connu et connaît encore de nombreuses évolutions. Pour accélérer le développement, certaines sections d'un système sont implémentées de façon logicielle, et sont exécutées sur un co-processeur intégré au design. Cette technique s'appelle la conception conjointe, ou *co-design* [54, 58, 176, 175]. Les choix de partitionnement entre matériel et logiciel sont fait au fur et à mesure du raffinement. Une autre innovation consiste à réutiliser des composants virtuels, ou propriétés intellectuelles (*Intellectual Property* IP) [45, 64, 71, 164, 84]. Ces blocs matériels conçus pour une tâche spécifique sont réutilisés dans de multiples circuits. Ils sont disponibles à différents niveaux d'abstraction, comme montré sur la figure 5. Cela permet un gain de temps très important. Dans la figure 3, on constate que les IPs prennent une place de plus en plus importante. Finalement, la pression du *time-to-market* pousse aussi les équipes de conception à automatiser de plus en plus les étapes de raffinement. La synthèse de haut-niveau – *High-Level Synthesis* (HLS) [34, 47] – permet d'automatiser quasi-intégralement le processus de raffinement d'une description comportementale en une description matérielle. Il est ainsi possible de générer plusieurs

circuits différents en fonction de contraintes d'intégration spécifiées par l'utilisateur, telles que la latence du circuit, son débit ou sa surface.

Abstraction, réutilisation et automatisation sont détaillées dans la section 2.

1.3 Évolution des méthodes de vérification

La conception d'un circuit passe par une phase de vérification nécessaire afin de s'assurer du bon respect des spécifications imposées par le cahier des charges. Il est souvent admis dans l'industrie que la vérification consomme plus de 70% du temps total de développement d'un circuit [120]. Si cette estimation doit être nuancée [10, 150], il est certain que plus la complexité des applications augmente, plus les besoins de vérification sont conséquents. Les méthodes de conception évoluant très rapidement, elles entraînent en même temps l'apparition de leur équivalentes dans le domaine de la vérification. Et l'importance donnée aux phases de vérification oblige les équipes de développement à adopter ces nouvelles mesures au fur et à mesure, pour rester compétitif. Comme le souligne Mark Olen, spécialiste en vérification fonctionnelle chez Mentor Graphics, «*if you haven't adopted it yet, your competitors probably have*» [153].

Il existe deux classes pour les méthodes de vérification : fonctionnelle et formelle [19, 177]. La vérification fonctionnelle d'un système se base essentiellement sur sa simulation. Le développement des systèmes au niveau comportemental autorise une simulation très rapide, grâce à une modélisation performante des sections de code les plus critiques. Cependant, plus le système est raffiné vers sa description matérielle, et plus les temps de simulation augmentent. Par conséquent, les ressources allouées à la vérification se déplacent de plus en fonction vers le niveau ESL. Au niveau matériel, les développeurs ne peuvent souvent plus créer les stimuli de simulation à la main pour les circuits les plus complexes, et ont recours à l'automatisation des testbenchs. Les stimuli sont générés aléatoirement pour couvrir le maximum de cas possibles. De plus, le modèle algorithmique est testé en même temps que le modèle matériel, avec le même banc de test, pour en comparer les sorties.

Mais la vérification d'un circuit est désormais aussi formelle. Les langages d'assertions permettent de décrire l'intégralité d'une spécification de façon purement formelle, sous forme de séquences logiques à respecter. Seulement, ces techniques sont encore très lentes et trop complexes pour être appliquées à des systèmes complets [177]. Couplées aux méthodes de vérification fonctionnelle, elles permettent toutefois une amélioration conséquente des capacités de vérification des équipes d'ingénieurs.

Les sections suivantes de ce chapitre présentent les concepts autour desquels s'articulent les travaux de cette thèse. Le flot de conception d'une application est détaillé, ainsi que différentes méthodes de vérification actuelles. Les notions de synthèse de haut-niveau et

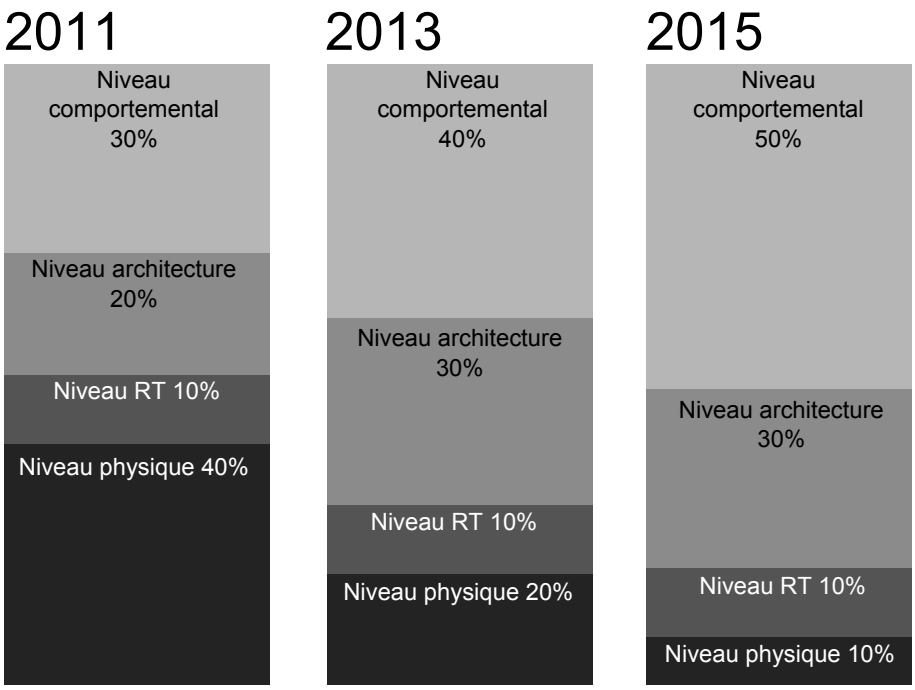

FIGURE 6 – Évolution des rôles des niveaux d'abstraction dans le processus de réduction de la consommation (ITRS 2011)

de vérification par assertion, piliers de ces travaux, sont aussi illustrés.

2 Flot de conception

La conception de circuits complexes passe désormais par de nombreuses étapes nécessaires afin de faciliter au maximum les choix fondamentaux pour l'implantation sur silicium, et d'éviter ainsi des changements de dernière minute qui peuvent s'avérer extrêmement coûteux. Ces étapes commencent toujours par une description haut niveau de la fonctionnalité qui est ensuite petit à petit raffinée vers une description matérielle. C'est le principe de l'*Electronic System Level* (ESL) *design*.

La figure 6 est issue du rapport ITRS 2011 [152]. Elle montre l'importance que prennent les modèles d'abstraction au cours des années. Ici, il est question de l'utilisation de ces modèles pour essayer de réduire la consommation des circuits. On constate que l'optimisation se fait de plus en plus au niveau système, au détriment du niveau physique qui devient trop complexe à manipuler. Ce déplacement de l'intérêt des équipes de conception des niveaux les plus bas vers les niveaux comportementaux ne se limite bien sûr pas à l'amélioration de la consommation des circuits, mais permet d'accroître la productivité des développeurs de façon globale.

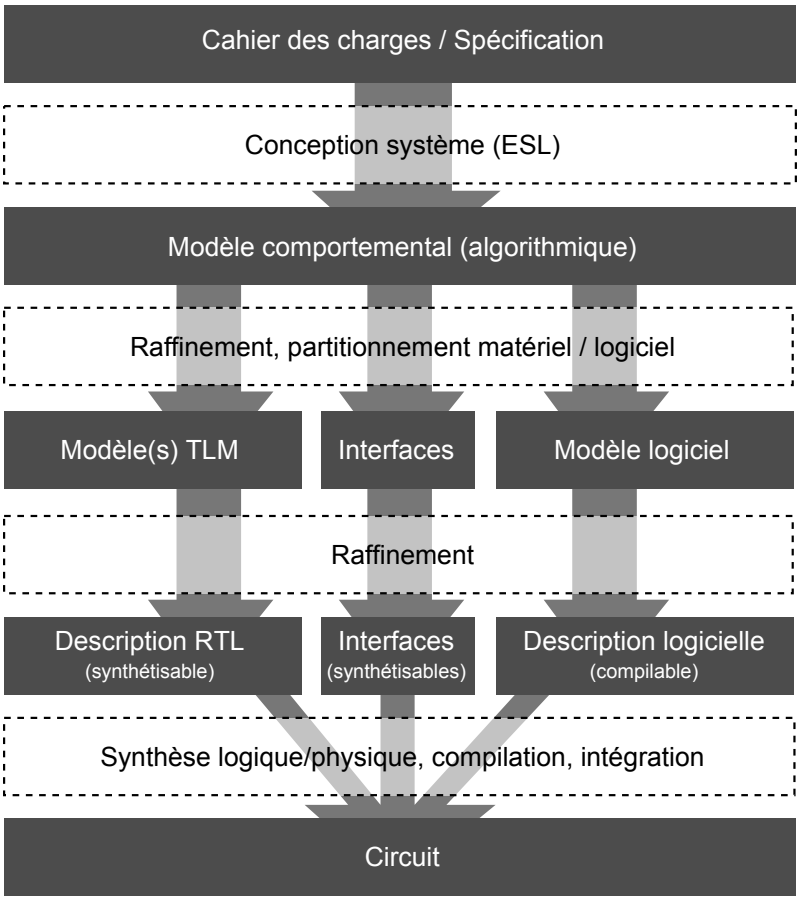

FIGURE 7 – Étapes de raffinement dans un flot de conception classique

La figure 7 présente les différentes étapes de développement d'une application pour circuit matériel. Concevoir un circuit commence nécessairement par l'établissement d'un cahier des charges décrivant les fonctionnalités requises de l'application. Suit alors la réalisation algorithmique de ce cahier des charges. Partant de là, la méthode de développement TLM (*Transaction Layer Model*), très employée dans l'industrie de nos jours, propose de raffiner ce modèle comportemental en plusieurs sous-modèles de plus en plus proches du niveau matériel final. Enfin, les concepteurs peuvent développer la description matérielle au niveau RTL.

Cette visibilité à différents niveaux d'abstraction a une conséquence sur les méthodes de conception conjointe. Le partitionnement matériel/logiciel se fait progressivement au cours des étapes de raffinement. Toutefois, la complexité croissante des applications rend aussi les choix de partitionnement de plus en plus complexes. Ainsi, ce partitionnement se voit de plus en plus automatisé. Les descriptions comportementales des applications sont analysées par un outil pour y détecter les sections critiques les plus susceptibles de bénéficier d'une accélération matérielle. Les espaces de solutions à explorer étant très vastes, plusieurs travaux [166, 103, 169] se concentrent sur l'optimisation de cette exploration de solutions.

Une fois cette description simulée sur ordinateur et validée, l'implantation sur silicium se fait au travers d'outils de synthèse logique. Une *netlist* est d'abord générée à partir de la description RTL. Elle décrit les interconnections nécessaires entre des instances de composants matériels de base (transistor, résistance, condensateur ou même puce électronique). Une phase de *placement-routage* est ensuite nécessaire pour répondre aux contraintes dictées par le matériel, liées aux entrées-sorties notamment.

2.1 Le modèle comportemental

Ce modèle, souvent nommé description fonctionnelle ou modèle algorithmique, permet de vérifier et valider le déroulement des opérations au niveau comportemental et fonctionnel, et de s'assurer ainsi que l'algorithme imaginé remplit la fonction attendue par l'application. La plupart des langages de programmation logicielle permettent de concevoir la description algorithmique d'une application. Les plus utilisés sont MATLAB [6], utilisé dans la plupart des applications de traitement du signal, le langage C [88], populaire car connu de la grande majorité des électroniciens, et le C++ [171], pour son approche modulaire également de mise dans les descriptions RTL. Une description comportementale de l'application peut être relativement simple et d'un niveau très abstrait, comme une description MATLAB d'un calcul de FFT (*Fast Fourier Transform*), ou beaucoup plus complexe comme l'implémentation C++ de la norme d'encodage vidéo H.264.

L'intérêt premier du modèle comportemental est de permettre une vérification fonctionnelle très rapide de l'application, car uniquement basée sur une modélisation très abstraite. Le modèle comportemental permet aussi de décrire plus formellement les interactions entre les composants que ce qui est possible au niveau matériel. Ainsi, il est à la base de méthodes de vérification formelle qui comparent l'implémentation matérielle à son modèle, comme l'*equivalence checking* [158, 92, 42].

2.2 Le modèle TLM (*Transaction Layer Model*)

Jusqu'aux années 1990-2000, l'étape suivant la description comportementale consistait à raffiner cette dernière directement en description matérielle prête à être implantée sur silicium. Ce saut important entre les deux niveaux d'abstraction a longtemps contribué à la grande difficulté de conception d'applications matérielles complexes. Toutefois, l'introduction d'une nouvelle méthode de raffinement, la méthode TLM [69, 146] a permis de découper ce saut d'abstraction en plusieurs étapes intermédiaires, facilitant d'autant la conception du circuit final et la détection d'erreurs au cours du développement. Ce modèle permet de descendre moins brusquement vers la description RTL, ce qui rend cette transition plus simple, autorisant ainsi la conception d'applications beaucoup plus complexes

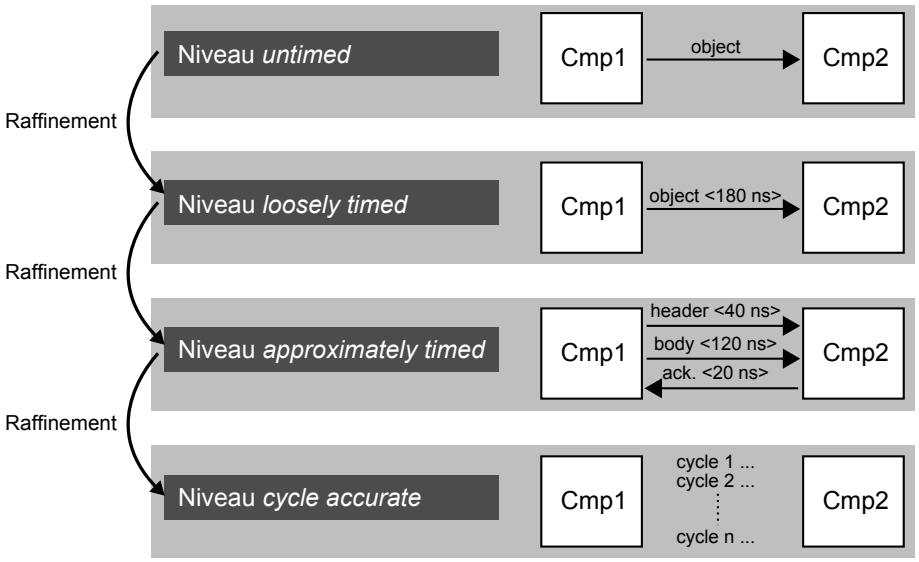

FIGURE 8 – Les différents niveaux d'abstraction du modèle TLM

qu'auparavant. De plus, son objectif est aussi de permettre un gain de temps important lors des phases de vérification, grâce à la modélisation des opérations et des transferts de données entre composants, de façon plus ou moins fine selon le niveau d'abstraction souhaité.

Le modèle TLM est divisé en plusieurs niveaux d'abstraction, liés au degré de précision temporelle des opérations et des échanges de données, et illustrés dans la figure 8 :

Untimed : Très proche de la description comportementale, la notion de temps n'est pas présente dans l'application. Toutes les opérations complexes (cosinus, fft) sont modélisées de façon algorithmique. Les échanges de données entre les composants se font par appel de fonction. L'intérêt principal réside dans la découpe de l'application en différents composants et canaux de communications. Le partitionnement logiciel/matériel est ainsi plus facile à appréhender et peut se faire à ce niveau.

Loosely timed : Les temps nécessaires aux échanges de données entre composants sont spécifiés pour chaque transaction individuelle. Cela permet d'avoir une idée approximative des goulots d'étranglements tout en maintenant une grande vitesse de simulation. Ce qui importe dans les échanges de données n'est pas leur déroulement précis, mais quelles sont les données échangées et comment celles-ci circulent dans l'application.

Approximately timed : Les transactions sont scindées en plusieurs parties, et chacune spécifie son propre temps d'exécution. Les protocoles de communication peuvent être implémentés de façon plus fine et l'accent peut être mis sur la façon dont les données sont échangées plus que sur les données elles-mêmes.

Cycle accurate : Précédant le niveau RTL, toutes les transactions et les opérations sont modélisées au cycle d'horloge près. Toutefois, de nombreux détails d'implémentation peuvent être omis tout en précisant le nombre de cycles d'horloge nécessaires à leur fonctionnement. Cela permet une modélisation orientée performances de certaines parties du système afin de conserver un temps de vérification bien plus efficace qu'au niveau RTL.

Toute la puissance du modèle TLM réside dans la compatibilité d'un niveau d'abstraction avec les autres. Cela permet de raffiner certains composants de l'application, tout en pouvant continuer à les tester au sein de l'ensemble du système. Les possibilités offertes par le modèle TLM ont contribué à son adoption rapide par beaucoup d'entreprises. Les langages de description les plus utilisés pour modéliser des applications en TLM sont le *SystemC* (IEEE 1666 [81]) et de façon moins prononcée le *SystemVerilog* (IEEE 1800 [79]). SystemC est issu d'une initiative du consortium d'entreprises OSCI [122], aujourd'hui fusionné avec la société Accelera [122]. Accelera est elle à l'origine de SystemVerilog. L'idée principale qui se trouve derrière le développement de ces deux langages est de fournir aux développeurs une interface de programmation (API) commune pour le développement de leurs modèles, afin de faciliter (voire de permettre) l'interfaçage de plusieurs modèles entre eux.

2.3 Le modèle RTL (*Register Transfert Level*)

Dernier modèle d'abstraction avant la synthèse logique, le modèle RTL décrit les interconnexions nécessaires entre les briques matérielles de base qui constituent le circuit final, et est couramment implémenté avec les langages de description *VHDL* (IEEE 1076 [80]) et *Verilog* (IEEE 1364-2005 [78]).

Une description matérielle au niveau RTL peut elle-même avoir différents niveaux d'abstraction. Les premiers outils de synthèse logique traduisaient directement la description en briques de base, et avaient donc besoin d'une spécification au niveau de la porte logique, voire du transistor. Cependant, les progrès considérables effectués par ces outils depuis les deux dernières décennies ont permis aux descriptions de devenir de plus en plus abstraites et éloignées des composants matériels primaires. Les ressources comme les opérateurs arithmétiques peuvent être inférées à partir de simples signes d'addition ou de multiplication présents dans la description matérielle, et elles sont souvent automatiquement partagées entre les composants en ayant besoin. Du fait de ces progrès rapides, les langages de description RTL ne s'utilisent désormais que très rarement au niveau de la porte logique, et permettent la construction d'architectures complexes de façon bien plus aisée qu'auparavant. Ils contribuent ainsi à la complexification des applications actuelles.

2.4 Évolution : blocs matériels réutilisables

La notion de composants virtuels, ou *Intellectual Properties* (IPs), constitue le changement majeur de ces dernières décennies dans le monde de l'électronique numérique, aussi bien pour la conception de circuits que leur vérification.

La réutilisation d'une description fonctionnelle est à la base de la programmation de logiciels, mais n'est arrivée dans le développement matériel que très tardivement. En programmation logicielle, des bibliothèques de fonctions pré-compilées réalisant les traitements des plus élémentaires (opérations mathématiques de base *libm* [136]) aux plus complexes (encodage/décodage vidéo *libav/ffmpeg* [117], crytopgraphie *openSSL* [90], calcul massivement parallèle *openMP* [104]) sont proposées aux développeurs. Cela permet d'éviter d'avoir à «réinventer la roue» sans cesse. Deux évolutions expliquent cette approche. Premièrement, la forte augmentation des performances en calcul brut des processeurs permet au programmeurs de ne pas avoir à parfaitement optimiser une description logicielle pour en obtenir une exécution rapide. Deuxièmement, les progrès considérables des compilateurs octroient à ces derniers la faculté d'adapter une description logicielle à un processeur spécifique sans changement au niveau de cette description. De ce fait, il est tout à fait possible de fournir des implémentations d'algorithmes génériques sous la forme de bibliothèque à lier à un projet.

Les composants réutilisables peuvent être classés en trois catégories : *Hard IPs*, *Firm IPs* et *Soft IPs* [163].

Hard IP : Ce type de composant est fourni sous forme de description physique binaire (*layout*) inaltérable et dédiée à une technologie matérielle définie. Cela permet au fournisseur de proposer un composant très optimisé, mais aucune personnalisation n'est possible. La structure interne du composant et son placement-routage sont déterminés à l'avance. Certaines sociétés comme ARM [124] ou MIPS [138] vendent leurs processeurs de cette façon. Cela leur évite d'avoir à fondre leurs composants, ce qui se retrouve au niveau du prix de vente. L'intérêt d'une hard IP est de pouvoir être intégrée (dans le cas des ASIC) directement au sein d'un layout plus grand, ce qui est impossible lorsque le composant est vendu fondu sur silicium.

Firm IP : Plus flexibles, les firm IPs correspondent à des netlists qu'il est possible de modifier avant synthèse physique et placement-routage. Le comportement du composant reste souvent trop complexe à déterminer pour être modifié, mais l'IP peut être intégrée au sein d'un système plus simplement : les contraintes de placement-routage sont libres.

Soft IP : Version HDL synthétisable du composant. Les soft IPs offrent un maximum de flexibilité et ne dépendent généralement pas d'une technologie précise. Une même

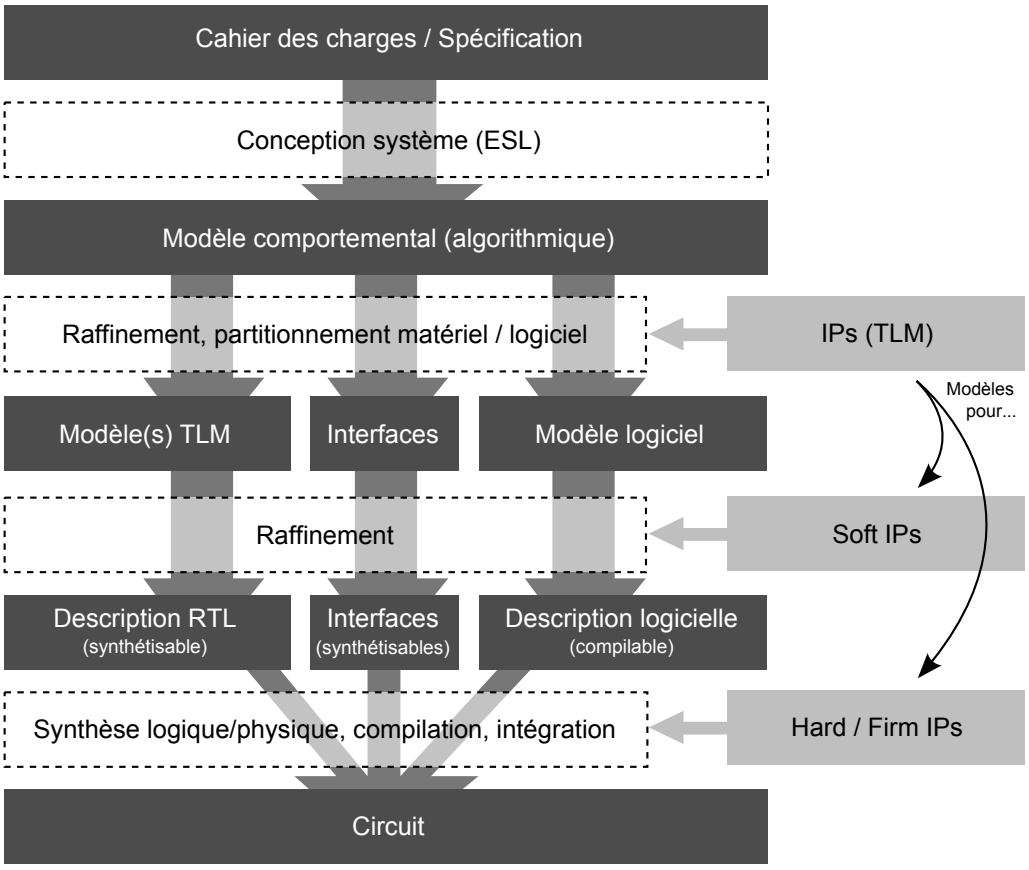

FIGURE 9 – Composants virtuels et flot de conception

IP peut être intégrée aussi bien dans un ASIC que dans un FPGA. Toutefois, il n'existe aucun moyen de protéger ce genre de composant par une compilation ou quelque autre moyen, et c'est donc son code source qui doit être vendu directement. Le fournisseur livrant ainsi les détails de son implémentation, le prix du composant s'en ressent fortement à la hausse.

La figure 9 montre l'utilisation de ces composants virtuels au sein d'un flot de conception d'un circuit. Le modèle TLM contribue pour beaucoup au développement de blocs matériels pré-conçus. Il est désormais systématique pour un vendeur d'IP de fournir un modèle de son composant compatible avec les différents niveaux d'abstraction TLM. Cela permet d'intégrer le composant directement dans le modèle du système, facilitant ainsi la simulation de ce dernier. Construisant sur ce principe, l'*Open Core Protocol* [168, 39, 181, 20] est récemment né de l'initiative *Open Core Protocol International Partnership* (OCP-IP) [140], regroupant plusieurs leaders de l'industrie. Il se base sur le modèle TLM pour proposer aux développeurs une interface de communication universelle pour les blocs réutilisables, elle aussi différenciée en plusieurs niveaux d'abstraction.

Les composants virtuels sont très utilisés dans le monde industriel pour accélérer le développement de circuits. Il existe de nombreuses sociétés spécialisées dans la vente de blocs

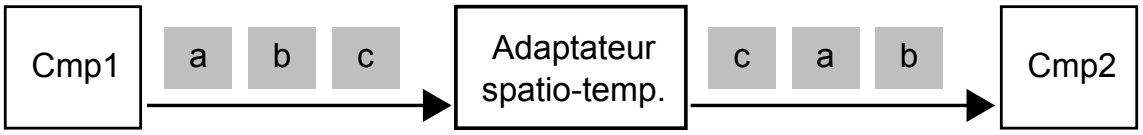

FIGURE 10 – Composants virtuels et adaptateurs spatio-temporels

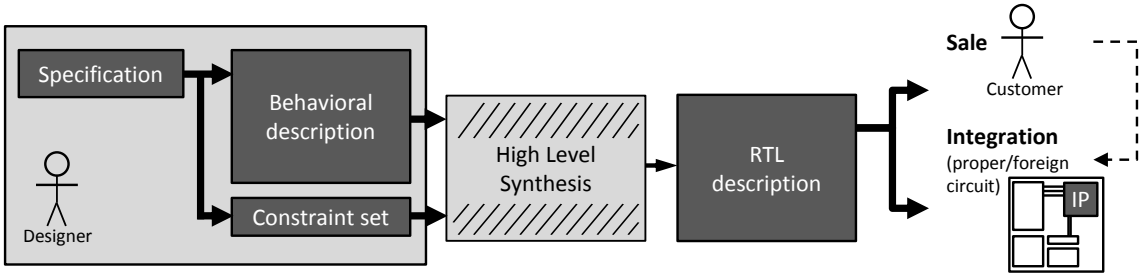

FIGURE 11 – Flot de conception avec synthèse HLS

réutilisables dans de nombreux domaines. Toutefois, les IPs ont pour principal problème d'offrir un fonctionnement fixe et souvent peu paramétrable. Une IP de décodeur d'image JPEG peut par exemple produire les valeurs de quatre pixels en parallèle alors que le système qui y est connecté ne peut les récupérer qu'une par une. Ce genre de différence d'interface nécessite l'utilisation d'un wrapper d'adaptation, ou adaptateur spatio-temporel, comme illustré dans la figure 10. Les données échangées entre l'IP et le système sont mémorisées et réordonnées si nécessaire, afin de satisfaire les contraintes des deux parties. Cela peut induire de très gros coûts matériels en terme de mémorisation. L'optimisation de ces adaptateurs fait l'objet de plusieurs travaux [45, 37, 36, 77, 9].

2.5 Évolution : synthèse de haut-niveau

La synthèse de haut niveau [34, 43, 44, 47, 49, 109], ou synthèse HLS (*High-Level Synthesis*), a pour ambition d'automatiser dans la mesure du possible les phases de raffinement menant de la description fonctionnelle d'une application à sa description RTL. Une synthèse HLS se base sur la description comportementale de l'application, et utilise un jeu de contraintes spécifiées par le développeur pour générer automatiquement une description matérielle répondant à ces contraintes (cf. figure 11).

Les contraintes permettent de spécifier la cible matérielle et sa technologie, mais aussi et surtout l'interface entrées/sorties désirée. Cela comprend entre autre les dates de disponibilité des variables d'entrée, les dates de production souhaitées pour les résultats, leur ordre et les registres où les écrire. En variant uniquement ces contraintes, il est possible de privilégier la taille de l'architecture générée, sa latence ou sa consommation électrique par exemple. Ainsi, le principal intérêt de la synthèse HLS est de pouvoir générer de multiples architectures dédiées à une cible matérielle et un environnement précis à partir d'une seule

et unique description fonctionnelle. Cette particularité est fondamentale pour la génération de composants virtuels. En effet, la synthèse HLS permet de produire des composants parfaitement adaptés aux contraintes des utilisateurs de ces composants, sans aucun besoin d'adaptateur spatio-temporel pour adapter les données produites par le système au composant, et vice-versa. De plus, les descriptions matérielles générées automatiquement sont très peu compréhensibles car optimisées et non pensées pour être lues. De ce fait, vendre un composant généré par HLS sous forme de soft IP ne dévoile pas pour autant l'algorithme implémenté, ce qui est un avantage fort dans l'industrie.

Les outils de synthèse HLS fournis par les principaux acteurs de l'industrie électronique sont *Catapult-C* [128] de *Mentor Graphics* (cédé à *Calypto Design Systems* en août 2011), *Synphony-C-Compiler* [145] de *Synopsis* et dans une moindre mesure *C-to-Silicon* [125] de *Cadence Design Systems*. Plusieurs outils académiques (souvent open-source) ont été développés à des fins de recherche, comme *Gaut* [46] de l'Université de Bretagne Sud / Lab-STICC, *GraphLab* [101] de l'Université de Bordeaux / IMS ou *HerculeS* de l'Université de Thessaloniki (Grèce).

Une synthèse HLS commence comme toute synthèse par une compilation de la description fonctionnelle en modèle interne à l'outil de synthèse. Cela se fait par une analyse lexicale (avec un *lexeur*) puis sémantique (avec un *parseur*) de la description fonctionnelle fournie. Le lexeur découpe la description en mots (*tokens*) associés à un type (identifiant, nombre, chaîne de caractères, etc). Le parseur quand à lui utilise ce découpage pour créer la hiérarchie de la description (unité de compilation, fonctions, déclarations, variables, etc). Le modèle abstrait qui en résulte s'appelle un arbre AST (*Abstract Syntax Tree*) [156]. La phase de compilation est basée sur la théorie des compilateurs [2], et utilise les mêmes méthodes d'optimisation que celles proposées par les compilateurs de logiciels. Le code mort (fonctions/variables non utilisées) est supprimé, certains appels de fonctions sont mis en ligne et les calculs déterminés sont remplacés par leurs résultats. Des outils HLS comme *C-to-Verilog* [126] utilisent le compilateur open-source universel *LLVM* [100, 98, 99, 135]. Ce même compilateur est utilisé par *Apple* pour produire des exécutables pour *MacOS* [99].

Un outil HLS effectue alors généralement les tâches suivantes :

– Pendant la phase de **sélection**, les types de ressources matérielles nécessaires aux opérations à effectuer sont choisies à partir des bibliothèques d'opérateurs disponibles. Les contraintes spécifiées (latence, surface et technologie) sont prises en compte dans ce choix.

– L'**ordonnancement** consiste à affecter une date d'exécution à chaque opération de l'application en prenant en compte les latences induites par les ressources sélectionnées.

– L'**allocation** des ressources se base sur le nombre d'opérateurs de même type qui vont

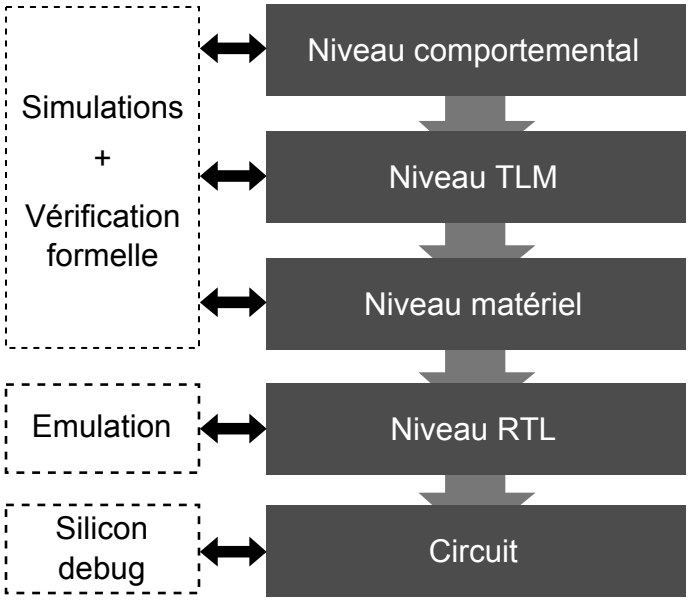

FIGURE 12 – Importance de la vérification dans un flot de conception

être utilisés à la même date. Chaque opérateur est réutilisé au cours de l'exécution de l'application.

– Finalement, l'**assignation** associe à chaque opération sa ressource matérielle correspondante.

3 Méthodes de vérification

La vérification d'une application est un point fondamental de son développement. Elle intervient à tous les niveaux d'abstraction, comme le montre la figure 12.

Chaque niveau de raffinement depuis la description comportementale permet d'avancer vers le modèle RTL, tout en affinant progressivement les choix d'implémentation nécessaires (type de bus de communication, protocole utilisé). Surtout, chaque niveau permet de vérifier que le raffinement est correct et que la description reste parfaitement fonctionnelle. Mais plus on descend dans les étapes du raffinement de l'application, et plus les temps de vérification sont longs et contraignants. Cela accroît fortement le coût de la correction d'une erreur (cf. figure 13), et repousse d'autant plus la date de mise en production du circuit.

Deux types de vérification bien distincts existent. D'une part, la vérification fonctionnelle, qui s'attache au comportement en exécution ou simulation de l'application par rapport à sa spécification. D'autre part, la vérification formelle, qui essaie de prouver à la compilation

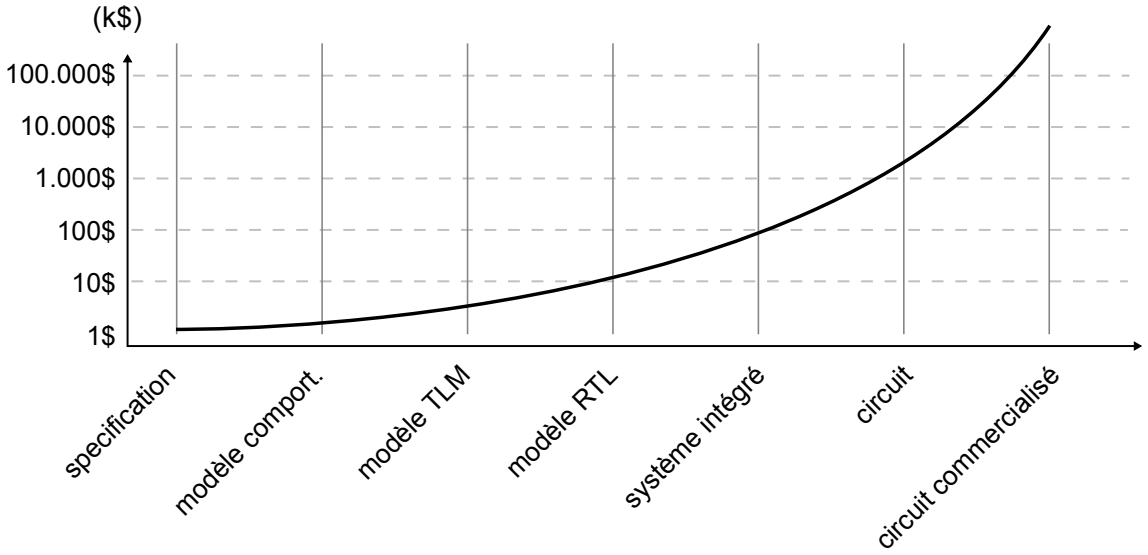

FIGURE 13 – Coût d'une erreur en fonction du niveau de raffinement

qu'une description matérielle est correcte et exempte d'erreur. Très complémentaires, elles font toutes deux l'objet de recherches continues. Les sections ci-après sont loin d'êtres exhaustives sur le sujet, mais fournissent des notions sur l'évolution des méthodes de vérification fonctionnelles et formelles.

3.1 Vérification fonctionnelle

La métrique principale de la vérification fonctionnelle est la «couverture de code», ou *code-coverage*. Elle évalue la quantité de code source testé lors des simulations de l'application. Dans un banc de test, un stimulus fourni à une application produit un résultat en déclenchant divers états internes à l'application. Le but principal de la couverture de code est de tendre vers 100% des états possibles testés. Pour un SoC, cela représente vite des millions d'états, et donc des millions de stimuli. L'automatisation de la génération des stimuli a rapidement été nécessaire. Afin d'obtenir la meilleure couverture possible, les ingénieurs ont de plus en plus recours aux tests dits «aléatoires et contraints» (*constrained randomized tests*) [18, 130], par opposition aux tests plus traditionnels dont les vecteurs sont élaborés à la main et précisément ciblés. Cela permet de détecter des problèmes insoupçonnés dus à des effets de bord, et évite d'avoir à créer manuellement les vecteur de test pour chaque cas.

Toutefois, cette méthode reste insuffisante pour les très gros systèmes, du fait de la couverture à obtenir. L'aléatoire a cela de particulier qu'un état peut être testé des dizaines de fois avant qu'un deuxième n'obtienne son stimuli correspondant. Pour palier à ce problème, les testbenchs ont évolués ces dernières années avec l'*Intelligent Verification*, ou *Intelligent TestBench Automation* (iTBA) [153, 74, 55]. Les données des modèles compor-

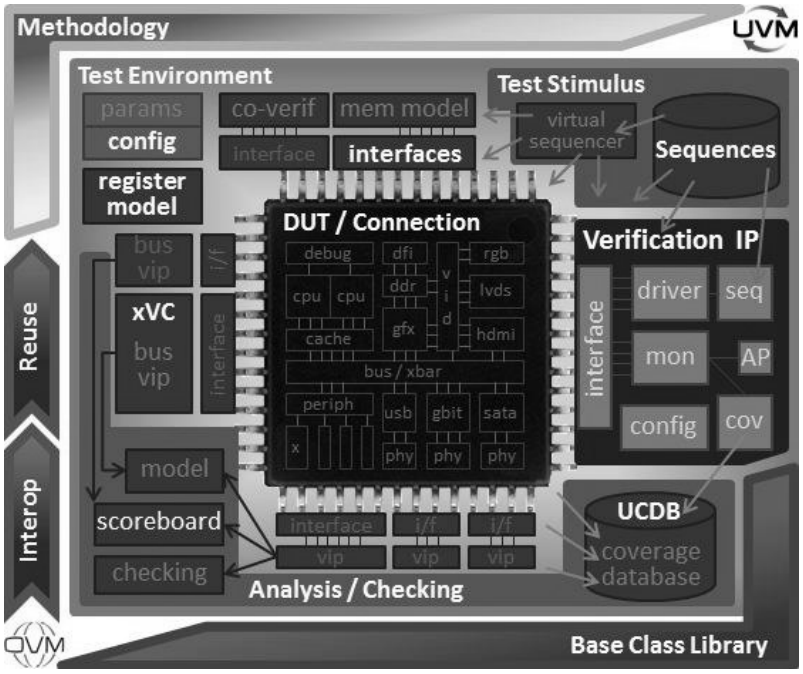

FIGURE 14 – Vérification d'une architecture par la méthode OVM/UVM

tementaux sont utilisées conjointement avec les tests existants pour générer des vecteurs de test adaptés à la description matérielle. Si le caractère aléatoire des vecteurs est supprimé, la génération d'un ensemble de vecteurs couvrant l'intégralité du code d'une description matérielle prend 10 à 100 fois moins de temps qu'avec des vecteurs aléatoires. 2012 a vu le début de l'adoption de cette méthode dans des outils commerciaux comme *Questa* [142], de Mentor Graphics [137].

Mentor Graphics est d'ailleurs l'artisan principal (avec *Cadence Design Systems* [127]) de l'*Open Verification Methodology* (OVM) [147, 72], en passe d'être standardisée par IEEE sous le nom *Universal Verification Methodology* (UVM) [162]. OVM/UVM (cf. figure 14) représente la définition standardisée la plus aboutie décrivant l'intégralité du processus de vérification d'une description RTL d'un système complexe (tel un SoC). Elle est le témoignage de l'évolution conséquente des mentalités sur l'importance de la vérification dans un flot de conception.

Pour accélérer toujours plus le processus de vérification, les efforts sont de plus en plus portés vers les modèles plus abstraits. Ces derniers sont beaucoup plus rapides à simuler, et offrent une vision proche de l'implémentation RTL. L'intérêt majeur du modèle TLM est de permettre la simulation de l'application tout en modélisant de façon très performante certains goulots d'étranglements. L'intégration de code RTL dans un système de niveau TLM est maintenant permise par les simulateurs, et offre au composant RTL à tester un environnement complètement fonctionnel mais rapide à simuler. Inversement, la vérification du modèle RTL se fait aussi de plus en plus aux niveaux plus bas, directement

sur l'implantation FPGA ou ASIC :

L'émulation des ASIC sur FPGA consiste à implanter la description matérielle d'un circuit dans un FPGA. Cela permet de pouvoir tester l'application en conditions réelles, et avec une vitesse d'exécution bien supérieure à la simulation. Toutefois, ce que l'on gagne en vitesse se perd au niveau de la facilité d'analyse des signaux, qui nécessite ici des outils de surveillance pour récupérer des chronogrammes d'activité lors de certains événements définis. La société *Xilinx* [148] fournit pour ses FPGA l'outil *ChipScope Pro* [129] à cette fin, similaire à *SignalTap II Logic Analyzer* [3] de la société *Altera* [123].

La vérification sur silicium (*post-silicon validation*) est la phase de vérification la plus redoutée. Le circuit est analysé une fois fondu sur silicium, en conditions réelles. Cette phase nécessaire permet de s'assurer du bon fonctionnement de l'application juste avant sa production en masse et sa mise sur le marché. Le circuit fonctionne ici à vitesse maximale, et peut être connecté à son environnement d'utilisation final. Toutefois, les coûts liés à la production d'un circuit sur silicium ont pour résultat la concentration des ressources dans les phases de vérification en amont, sur les modèles et par émulation du design sur FPGA, afin de minimiser la probabilité qu'une erreur perdure jusqu'à cette phase de vérification. De plus, la complexité des applications actuelles et les fenêtres de mise sur le marché (souvent très courtes) ne permettent plus de vérifier le circuit au niveau électrique, au bénéfice de la seule vérification fonctionnelle. La très grande difficulté d'observation des signaux pousse les développeurs à consacrer plus de temps aux vérifications en amont, au niveau des simulation des modèles et de l'émulation sur FPGA.

Néanmoins, la vérification des architectures matérielles ne peut plus toujours se faire de façon exhaustive, et les équipes de vérification, pressées par le *time-to-market*, doivent faire la part entre les fonctionnalités à tester complètement et celles considérées comme secondaires. Il n'est pas rare qu'un circuit, une fois commercialisé, voit son architecture modifiée au cours de sa durée de vie suite à la découverte d'un bogue, exigeant un nouveau passage en fonderie (on parle de *re-spin* du circuit). De ce fait, de nombreux circuits, SoC en premiers, embarquent dans leur version commerciale toute la connectique nécessaire pour continuer à déboguer le système une fois déployé (port JTAG, sorties de lecture de messages d'erreurs) et continue à mémoriser les erreurs potentielles qui surviennent. Cette méthode, le *in-field monitoring* [25], permet d'analyser le circuit en conditions réelles et pendant une très longue période de fonctionnement. De par sa nature, elle ne permet pas de corriger les problèmes avant la mise sur le marché, mais seulement à posteriori.

3.2 Vérification formelle

Les difficultés liées aux méthodes de vérification fonctionnelle de la description matérielle – simulation, émulation et vérification sur silicium – ont rapidement poussé les techniques de vérification formelle sur le devant de la scène. Longtemps restées un domaine de recherche académique, elles sont de plus en plus utilisées dans l'industrie, surtout dans le domaine du développement matériel (le logiciel ayant une meilleure tolérance aux erreurs subsistant dans un produit final). Au delà de la vérification fonctionnelle, qui s'attache à vérifier le circuit en fonctionnement, la vérification formelle se concentre sur l'analyse statique de la description de l'application. Elle a pour objectif la démonstration mathématique de la conformité d'une description, matérielle ou comportementale, par rapport à sa spécification. Cette méthode de vérification des circuits et des algorithmes s'avère très puissante, car entièrement automatisée et non soumise à l'erreur humaine. Deux grandes catégories d'outils se distinguent. D'une part, le *Model Checking* [40, 17], qui consiste à explorer tous les états possibles d'une l'application et à tester les transitions avec une spécification temporelle décrivant les transitions permises. D'autre part, de façon complémentaire, le *Theorem Proving* [180], qui raisonne sur un modèle purement mathématique de l'application, et essaie de prouver de façon plus ou moins automatisée sa validité.

L'introduction de langages dédiés aux propriétés temporelles d'une description, comme le PSL (*Property Specification Language*) [82], a permis d'augmenter grandement l'utilisation des méthodes formelles pour analyser les descriptions matérielles. Ce type de langage permet de décrire entièrement une application sous forme d'assertions. Ces assertions définissent ici des conditions booléennes ou des séquences temporelles à vérifier. Par exemple, le listing 1.1 montre une assertion exprimée en PSL ciblant la vérification du signal de remplissage d'une FIFO. La séquence décrit le comportement suivant : «si la FIFO est presque pleine, alors toute écriture de donnée doit entraîner le signal de remplissage complet». Grâce à ce type de possibilité, des techniques formelles comme l'*equivalence Checking* [158, 92, 42] essaient de prouver directement qu'une description matérielle correspond parfaitement à son modèle comportemental.

Listing 1.1– Exemple de propriété PSL

```
1 fifoFullTest : assert always {
2     fifo_almost_full |=> fifo_push |-> fifo_full;
3 } @(rising_edge(clk));
```

Les propriétés temporelles d'une application sont aussi exploitables en vérification fonctionnelle. En effet, les simulateurs vérifient qu'elles sont respectées à chaque cycle d'horloge simulé. De même, de nombreux travaux (décrits dans le chapitre suivant) s'attachent à transformer ces propriétés temporelles en moniteurs synthétisables afin de fournir un service similaire à celui des simulateurs mais directement sur le circuit, en émulation ou

silicon-debug. Ces possibilités ont créé un très fort engouement pour les langages d'assertions. Le langage de description HDL SystemVerilog s'est enrichi d'une bibliothèque dédiée aux propriétés formelles (*SystemVerilog Assertions*, SVA [32, 53]). L'*Open Verification Library* (OVL) [141, 97] est une initiative d'Accelera pour fournir une même syntaxe de description de propriétés à la majorité des langages HDL existants (VHDL, Verilog, SystemVerilog, et même PSL). La vérification d'une description RTL par l'utilisation des assertions se nomme l'*Assertion Based Verification* (ABV) [63, 62, 107, 174, 61]. Apparue récemment, cette technique de vérification par assertions est devenue l'une des méthodes de vérification les plus populaires et les plus utilisées.

La vérification par assertions est aussi possible au niveau de la description comportementale. La notion de temps ayant disparu, les propriétés temporelles font place aux assertions purement booléennes, aussi appelées assertions de niveau système, ou assertions comportementales. La présence de ces assertions dans une description algorithmique permet d'en vérifier mathématiquement la validité. Les assertions comportementales font aussi très souvent partie intégrante du langage de description utilisé à haut-niveau. Elles sont ainsi compilables sous forme exécutable, et fonctionnent comme moniteurs au sein du programme. Dans cette configuration, une détection d'erreur entraîne l'arrêt immédiat du programme, et laisse l'utilisateur inspecter l'état de ce programme tel qu'il était au moment de l'erreur (contenu de la pile d'appels de fonctions, valeur des variables, etc.).

Les méthodes formelles, surtout aux niveaux d'abstraction les plus bas, ont de très grands besoins en temps d'analyse et en mémoire, du fait d'espaces de solutions parfois gigantesques, et ne peuvent pour le moment remplacer la vérification fonctionnelle du système complet. Néanmoins, plusieurs travaux se font actuellement dans ce domaine, ciblant notamment la vérification par assertions [56, 96, 106, 172].

4 Problématique

Les utilisateurs attendent de plus en plus de fonctionnalités d'un même système. Cela pousse les fondeurs à accroître encore la capacité d'intégration des circuits pour permettre à des systèmes de plus en plus complexes de voir le jour. Toutefois, cette augmentation continue de la complexité des systèmes accroît aussi les temps de conception et de vérification, et repousse les dates de mise sur le marché. Pour palier à cela et continuer à satisfaire un *time-to-market* paradoxalement de plus en plus court, trois méthodes principales ont émergé : **abstraction**, **réutilisation**, et **automatisation**.

Premièrement, l'abstraction des designs du niveau RTL au niveau comportemental permet aux développeurs d'avoir une meilleure compréhension des mécanismes internes au

système. Les choix d'implémentation des fonctionnalités abstraites (type de bus et protocole de communication, réalisation des opérations, etc) peuvent être raffinés petit à petit. Deuxièmement, la réutilisation de composants virtuels permet de ne pas avoir à les développer spécifiquement pour l'application. Ces IPs sont fournies comme étant vérifiées et fiables. Enfin, l'automatisation des méthodes de conception et de vérification permet un gain de temps substantiel lors du développement d'applications et de systèmes. Particulièrement, les techniques de synthèse de haut-niveau permettent dans certains cas de se passer du raffinement manuel du modèle comportemental vers le modèle RTL.

4.1 Problème général

Cependant, il existe une différence importante entre l'évolution des méthodes de conception et celle des méthodes de vérification. En effet, les méthodes de conception qui ont émergé ces dernières années se concentrent pour la plupart sur l'amélioration du raffinement progressif d'une spécification en modèle matériel. Les méthodes d'abstraction TLM cherchent à faciliter le raffinement en le découpant en plusieurs paliers, l'utilisation d'IPs permet de passer plus rapidement d'un niveau à l'autre sans avoir à tout coder à chaque transition, et les méthodes de synthèse HLS rendent possible le passage automatique de la description comportementale à la description matérielle. Au contraire, les méthodes de vérification se concentrent toutes sur un niveau d'abstraction spécifique, et ne sont pour la majorité d'entre elles pas directement portables d'un niveau à l'autre. Ainsi, pour chaque raffinement d'un modèle d'abstraction à un autre, les techniques de vérification changent, et les informations utilisées à un niveau ne sont alors plus utilisables au suivant. Par exemple, les assertions utilisées à haut niveau pour vérifier le niveau comportementale ne sont en aucun cas réutilisables pour la vérification de la description RTL, ou l'on doit définir des propriétés temporelles correspondant aux données présentes à ce niveau.

Ainsi, l'idée générale des travaux présentés dans ce manuscrit est la réutilisation des informations de vérification présentes dans les modèles comportementaux pour la vérification des modèles RTL. Plus précisément, c'est l'automatisation de la propagation de ces informations qui est visée, afin de fournir une aide supplémentaire au concepteur.

4.2 Contexte des travaux

Ces travaux ont pour contexte la synthèse HLS d'une description comportementale. Cette dernière permet de générer automatiquement une description RTL dédiée à une cible matérielle, sans vraiment nécessiter l'intervention du concepteur entre les étapes. De ce fait, le concepteur n'a pas la maîtrise du fonctionnement interne de l'architecture générée. Les techniques de vérification *white-box* – c'est à dire s'intégrant au sein du composant –

comme l'*Assertion-Based Verification* sont alors limitées aux comportements prévisibles du composant généré, tels que la gestion des entrées/sorties. Dans ce contexte, il devient alors particulièrement intéressant de pouvoir faire propager par l'outil de synthèse les assertions système de la description comportementale vers la description RTL. De plus, une partie des tests que l'on peut vouloir faire sur les entrés/sorties du composant sont souvent déjà formalisés dans les assertions de la description comportementale. Les réécrire dans un langage adapté à la description RTL est une perte de temps, rendue de plus compliquée par les décisions prises par l'outil HLS.

4.3 Solutions proposées

Solution 1 : Transformation d'assertions

Aussi fiable que soit le modèle comportemental, les méthodologies de synthèse HLS ne s'intéressent pas aux assertions – et autres définitions formelles – présentes dans un modèle comportemental. Certains outils de synthèse les ignorent, et d'autres les considèrent comme des appels de fonctions classiques, menant à une implémentation inadaptée de leur comportement.

Par conséquent, la vérification de l'architecture générée ne peut se baser sur les assertions déjà présentes dans le modèle comportemental de l'architecture. Les équipes de vérification sont donc contraintes à écrire ces assertions manuellement dans la description RTL générée. Or, les transformations choisies par la synthèse HLS pour passer du modèle comportemental à la description matérielle sont souvent loin d'être triviales. Par exemple, les zones de mémorisation des données (registres, RAM) sont réutilisées au cours du temps pour différentes données. Écrire les assertions RTL correspondant aux assertions comportementales peut rapidement devenir très compliqué et très long, donc très coûteux.

La méthodologie proposée dans la première partie de cette thèse adresse ce problème par la propagation des assertions algorithmiques dans la description RTL. Le processus de synthèse HLS est complété par l'analyse des assertions de la description comportementale et leur transformation en assertions RTL.

Solution 2 : Gestionnaire d'erreurs matériel

Les assertions RTL sont non-synthétisables, et donc inutilisables en émulation et silicon-debug. Plusieurs travaux de la littérature évoqués dans le chapitre suivant ont pour objectif la génération de moniteurs matériels synthétisables à partir de ces assertions. Ces moniteurs assurent la même fonction que les assertions : observer les changements d'état du

système et prévenir l'utilisateur en cas d'erreur, mais cette fois lorsque le circuit est en fonctionnement réel.

Toutefois, les outils de simulation gardent en mémoire l'intégralité des chronogrammes liés à l'ensemble des signaux du système. Le développeur peut donc à tout moment revenir au niveau d'une erreur détectée par une assertion pour analyser l'état des signaux tels qu'ils étaient à ce moment là. Cette analyse est souvent déterminante pour la compréhension de la cause de l'erreur. Or, les moniteurs matériels proposés par l'ensemble de la littérature exposent souvent seulement un signal de sortie booléen qui s'active lorsqu'une erreur est détectée. De ce fait, la cause de l'erreur peut demeurer difficile à appréhender en pratique.

Ainsi, nous proposons un gestionnaire d'erreurs synthétisable. Son rôle est de récupérer, au moment d'une erreur, l'état de variables de l'application définies par le concepteur afin de les mémoriser pour un accès ultérieur. Chaque erreur donne lieu à l'organisation des données relatives sous forme de rapport. Le gestionnaire est synthétisé automatiquement en fonction des informations apportées par l'outil de synthèse HLS.

5 Conclusion

La complexité croissante des systèmes pousse l'apparition de méthodes de conception et de vérification de plus en plus performantes. L'abstraction des modèles, la réutilisation des composants et l'automatisation des étapes de conception sont les principales réponses à cette complexité.

Les méthodes de conception les plus importantes sont aujourd'hui les blocs matériels réutilisables (IPs) et les techniques de synthèse de haut-niveau (HLS). Quant à la vérification, l'utilisation d'assertions (booléennes aux haut-niveaux d'abstraction et temporelles au niveau RTL) permet de détecter les erreurs au plus près de leurs sources. Toutefois, les assertions spécifiées à haut-niveau pour vérifier les modèles comportementaux ne sont pas conservées par les outils de synthèse HLS. Certains travaux de la littérature répondent en partie à ce problème en transformant les assertions booléennes en moniteurs matériels pour analyser les circuits en fonctionnement réel. Dans cette approche, les méthodes décrivent comment synthétiser les moniteurs, mais ne précisent pas comment permettre la récupération et l'analyse des signaux d'erreurs.

Ainsi, les travaux présentés dans cette thèse se concentrent sur ces deux points. En particulier, ils proposent une méthode pour la transformation des assertions booléennes spécifiées dans la description algorithmique d'une application en moniteurs matériels pour la simulation, afin de conserver le formalisme des assertions. Une deuxième méthode est proposée.

Elle cible la synthèse automatique d'un gestionnaire d'erreurs matériel dont le rôle est d'archiver les erreurs survenant dans un circuit en fonctionnement réel, ainsi que leurs contextes d'exécution.

Le chapitre suivant présente en détail la vérification par assertions, et propose un état de l'art sur les méthodes de transformations d'assertions existantes.

Chapitre 2

État de l'art

Sommaire

La vérification fonctionnelle est l'approche la plus couramment utilisée lors de la validation d'un système ou sous-système, et cela indépendamment du niveau d'abstraction. Cette approche nécessite à bas niveau le déploiement de techniques complexes comme l'émulation sur FPGA ou l'analyse sur silicium. L'utilisation d'assertions au sein des modèles améliore le taux de détection des erreurs et facilite leur correction. Les assertions sont utilisables du niveau système jusqu'aux niveaux de description les plus bas (ASM, C, RTL). Toutefois, leur niveau d'expressivité varie en fonction des langages et des niveaux d'abstraction. Elles sont de plus inutilisables en émulation et silicon-debug.

Ce chapitre développe le concept de vérification par assertions, notamment au niveau système, et détaille les travaux existants visant à les convertir en moniteurs matériels intégrés aux circuits.

1 Introduction

Dans un flot de conception traditionnel, la simulation du système permet d'inspecter point par point son fonctionnement, quel que soit le niveau de raffinement. Cette simulation est souvent aidée par l'inclusion d'assertions dans les descriptions du système. Ces assertions permettent d'exprimer formellement les spécifications du cahier des charges, mais aussi les contraintes liées aux choix d'implémentation. Les mécanismes d'assertions sont utilisés depuis le modèle comportemental du système jusqu'à sa description à bas niveau (C, RTL).

Bien entendu, plus le système à vérifier est complexe et plus les temps de simulation sont longs. Ainsi, pour les niveaux d'abstraction les plus bas (niveau RTL notamment), la complexité et la quantité des interactions entre composants ralentissent fortement ce processus de simulation. En effet, celui-ci doit garder en mémoire un nombre conséquent de paramètres (états courants de tous les signaux et des mémoires). La simulation d'un seul cycle d'horloge pouvant prendre un temps important, simuler le système (ou une sous-partie) sur une période significative peut alors prendre des jours, voire des mois. Ce constat a motivé le développement et la mise en oeuvre des méthodes d'émulation des designs sur FPGA. Ces dernières permettent de tester le système en conditions réelles, avec une fréquence d'horloge (et donc une vitesse d'exécution) relativement proche de celle du circuit final.

Cependant, la vérification d'un circuit gravé sur silicium ne peut plus se faire par simulation, et nécessite des techniques très sophistiquées. Pour des finesses de gravure jusqu'à 0.5 µm, une station de micro-sondage [4] est utilisée. Associant un microscope et divers types de sondes, ce type d'appareil permet d'observer les états des signaux internes du circuit. Pour les technologies plus fines, on utilise des sondes ioniques focalisées (FIB) [4, 70], qui consistent en une chambre à vide avec un canon à particules. L'accès aux couches les plus basses du circuit est alors possible en injectant de l'iode par une aiguille très proche de la surface de la puce, sans endommager la structure du circuit. Ces techniques reposent sur des outils extrêmement coûteux, et sont loin d'être évidentes à mettre en oeuvre. Il est aussi difficile de faire le lien entre un fil conducteur du circuit et le signal correspondant du système au niveau RTL. De plus, les contraintes d'espace limitent l'observation de nombreux signaux simultanément. Il est alors nécessaire de stopper le circuit au moment voulu pour avoir accès aux différents signaux les uns après les autres.

Ainsi, au lieu de positionner des sondes au dessus d'un circuit pour l'observer, il est alors préférable de les intégrer directement dans le circuit. Ces sondes – ou *analyseurs logiques intégrés* – dédiés à la surveillance permanente des propriétés du système, sont alors utilisables en silicon-debug comme en émulation, sans nécessiter de modifications destructrices de la puce. Des analyseurs logiques intégrés sont proposés par les acteurs principaux du marché des FPGA. Toutefois, de nombreux travaux s'attachent depuis quelques années à

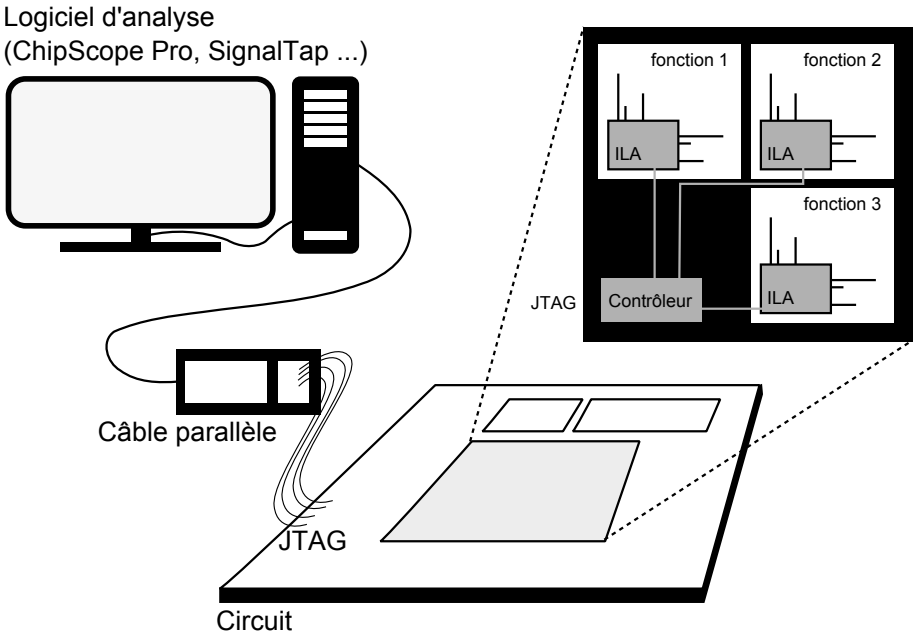

FIGURE 15 – Schéma de fonctionnement d'un analyseur logique intégré

simplifier la vérification par assertions aux niveaux d'abstraction les plus bas, y compris en émulation. Ces travaux visent à concevoir et à intégrer de manière automatique des moniteurs matériels à partir des spécifications des assertions issues des niveaux d'abstraction supérieurs. Ces travaux seront présentés plus en détail tout au long de ce chapitre.

Afin de simplifier la mise au point des circuits sur FPGA, l'incorporation de moniteurs basiques est facilitée dans les outils standards des principaux acteurs du marché. En effet, la mise en oeuvre de ces moniteurs est assurée par les outils *Chipscope Pro* [129] de *Xilinx*, *SignalTap II* [3] d'*Altera* et *Reveal* [143] de *Lattice*. Ces outils permettent la génération automatique d'une IP dédiée au monitoring de signaux, nommée couramment *Integrated Logic Analyser* (ILA). Cette IP, conçue tel un analyseur logique, est directement insérée dans la netlist du design au moment de la synthèse. Le monitoring est configurable par l'utilisateur. En effet, ce dernier peut choisir plusieurs déclencheurs (*triggers*) et la liste des signaux à enregistrer. Chaque trigger teste en permanence si sa condition est vérifiée ou non. Lorsqu'une condition est vérifiée, l'IP de monitoring stocke dans une mémoire tous les changements d'états des signaux à observer, pendant un nombre de coups d'horloge défini par l'utilisateur. Ces informations sont ensuite transmises vers un outil logiciel par une liaison JTAG.

Les analyseurs logiques intégrés, s'il sont très utiles pour la vérification d'une description RTL, ne font pas le lien entre les niveaux d'abstraction d'un flot de conception. En effet, ils ne font qu'enregistrer les changements d'état des signaux du circuit, et ne permettent aucune correspondance entre signaux RTL et variables définies au niveau comportemental.

De plus, la complexité des triggers varie de manière significative en fonction des outils. Ces derniers peuvent être basiques ($=$, $\neq$) ou plus complexes ($<$, $\leqslant$, combinés à des opérations logiques). Il est aussi souvent possible de tester une séquence (limitée) de ces triggers. Toutefois, le niveau d'expression des assertions est fortement réduit par rapport à ce qui peut être supporté aux niveaux d'abstraction supérieurs. Altera propose un langage de description des conditions des triggers [144] permettant une conception assez élaborée, mais il demeure loin des possibilités permises par les langages de propriétés comme le PSL [82].

De ce fait, utiliser les assertions spécifiées dans les autres niveaux d'abstraction pour définir les conditions des moniteurs intégrés permettrait l'unification des langages de vérification, offrant ainsi un gain de temps important. De nombreux travaux se concentrent depuis une dizaine d'années sur les assertions utilisées en vérification formelle et fonctionnelle pour synthétiser des moniteurs matériels. La vérification par assertion [63, 62, 107, 174, 61] est un procédé qui a pris une très grande importance parmi les différentes méthodes de vérification, et ce à tous les niveaux du flot de conception.

2 Vérification par assertions

Les assertions permettent de transcrire les spécifications d'un système et les choix d'implémentation de façon formelle directement au sein des modèles utilisés pour décrire le système. Elles contribuent à la fois à la vérification formelle et à la vérification fonctionnelle du système. D'une part, elles aident les outils de vérification formelle à effectuer l'analyse des modèles (par exemple une architecture de niveau RTL). D'autre part, elles agissent comme des moniteurs lors de la simulation du système, en signalant les divergences entre les hypothèses de fonctionnement et la réalité au moment où cela se produit. Cela permet une détection des erreurs au plus proche de leur source et évite d'avoir à attendre la validation des sorties de l'application pour détecter un problème éventuel.

La découverte d'erreurs à la fin de la simulation du modèle représente le pire cas de détection. En effet, l'erreur identifiée à la sortie du système peut provenir de n'importe quel élément du modèle participant directement ou indirectement à la production de la donnée en question. Ainsi, une erreur générée par une portion de code possédant un défaut peut parfois circuler longtemps au sein d'un système, à travers de nombreux composants, avant d'être finalement détectée. Remonter alors à la portion de code à l'origine de l'erreur est souvent complexe et très chronophage. Il est donc indispensable pour les concepteurs logiciel/matériel de détecter les erreurs au plus proche de leur source afin de pouvoir les corriger rapidement. Pour ce faire, l'usage des assertions au sein des descriptions logicielles et matérielles d'un système permet d'en accroître fortement la fiabilité.

L'introduction des langages de propriétés temporelles tels que le PSL a ouvert la voie à l'*Assertion-Based Verification* (ABV) [63] pour la vérification des modèles RTL, méthode de vérification rapidement devenue très populaire. Néanmoins, les assertions temporelles sont inefficaces pour vérifier les modèles de plus haut-niveau (comme le modèle comportemental) où la notion de temps est absente. A ce niveau d'abstraction, les assertions sont purement booléennes, et s'occupent principalement de comparaison entre primitives. La grande majorité des langages de programmation de niveau système (MATLAB, SystemC, C++, ...) disposent d'un mot-clé dédié aux assertions, ou d'une bibliothèque standard y donnant accès sous forme d'appel de fonction. Par exemple, le listing 2.1 montre un exemple de code C utilisant une assertion booléenne pour vérifier que la division effectuée est possible. Le listing 2.2 quant à lui présente un exemple typique d'utilisation d'assertions en PSL au niveau RTL. La séquence temporelle spécifie qu'après un signal *reset*, il faut que le signal x passe à 1, puis y pendant une période de 100 cycles, et qu'enfin le signal *sig_done* soit valide.

Listing 2.1– Utilisation d'assertions booléennes au niveau comportemental et RTL

```
1 int foo(int a, int b) {
2     int t1 = a + b;
3     assert(b != 0);
4     int t2 = a / b;
5     return t1 + t2;
6 }
```

Listing 2.2– Utilisation d'assertions temporelles au niveau RTL

```
1 sequence foo is {x; y[*1 to 100]; sig_done};
2 assert always reset |=> {foo} @(rising_edge(clk);
```

2.1 Les différents types d'assertions

Les assertions permettent d'introduire formellement dans le code les modèles des propriétés qui doivent être toujours vérifiées. Toutefois, en fonction de la nature des hypothèses, on distingue deux types d'assertions : assertions de spécification ou assertions d'implémentation. En effet, une même spécification peut être implémentée de différentes façons, en faisant différents choix.

D'une part, les assertions issues des spécifications sont relatives aux équipes de vérification. Elles traduisent formellement la spécification de l'application de façon non-ambiguë et concise. Dans le listing 2.3, l'assertion permet de s'assurer que le modèle fonctionnel du système est utilisé correctement : si a est négatif ou nul, b doit être nul, sinon, il doit être strictement positif.

Listing 2.3– Exemple d'assertion de spécification

```
1 int foo(int a, int b) {
2     assert(a <= 0 ? b == 0 : b > 0);
3
4     ...
5     ... // implémentation de la fonction
6     ...
7 }
```

Sans ce type de vérification préalable par assertion, il n'y a que deux possibilités :

– Soit le modèle et son implantation implémentent un mécanisme de détection d'erreur de type $if...then...else$. Ce genre de vérification a un coût matériel/logiciel car il est implémenté comme faisant partie du design. Le coût matériel associé est pénalisant si cette détection n'est utile que pendant le debug du circuit. Toutefois, le système restera robuste.

– Soit le modèle ne se préoccupe pas de faire la vérification de cette hypothèse car il considère que l'erreur n'arrivera pas. Cela sous-entend que le système englobant le modèle est bien conçu et cohérent vis-à-vis du cahier des charges de ce modèle. Cela évite tout surcoût inutile, mais si un défaut de conception génère une donnée d'entrée incorrecte, remonter à la source de l'erreur peut devenir très complexe.

L'utilisation d'assertions pour vérifier le respect des spécifications permet donc de faciliter la phase de debug (en simulation et vérification formelle), tout en évitant le surcoût matériel/logiciel d'une implémentation de la vérification dans le circuit (les assertions étant automatiquement supprimées par les outils lors de la synthèse logique ou compilation logicielle).

D'autre part, les assertions issues des choix de conception et de raffinement ne concernent que les équipes de conception. Ces équipes sont amenées à sélectionner et raffiner différents algorithmes pour une même fonctionnalité. Par exemple, un modèle comportemental implémentant la fonction racine carrée peut utiliser directement la fonction *sqrt()* correspondante dans le langage de description utilisé. Toutefois, en abaissant le niveau d'abstraction, les concepteurs doivent faire un choix quant à l'implémentation matérielle de cette fonction. Différentes solutions sont envisageables en fonction du niveau de précision et des performances souhaitées. L'article [151] propose 13 implémentations algorithmiques différentes pour cette fonction, chacune ayant ses avantages et défauts. Le choix d'une implémentation implique nécessairement plusieurs contraintes fonctionnelles qu'il faut pouvoir vérifier à l'exécution. L'utilisation d'assertions dans ce cadre-ci est alors directement liée à l'implémentation de la fonction, et non à sa spécification, donc nous avons affaire à une assertion d'implémentation.

Le listing 2.4 montre l'implémentation en SystemC d'une fonction de racine carrée (selon la méthode babylonienne [93]). Plusieurs assertions y ont été intégrées. On constate que les assertions de spécification n'ont pas de rapport direct avec son implémentation, mais concernent les conditions d'utilisation de la fonction. En effet, la racine carrée doit pouvoir être calculée pour tout réel positif ou nul, et le résultat est obligatoirement inférieur au réel d'entrée si ce dernier est supérieur à 1. Une assertion d'implémentation, comme celle spécifiée à la ligne 10, est directement liée à la méthode d'implémentation choisie. Ici, i est une variable uniquement utilisée dans cette fonction, et le fait qu'elle doive être positive n'a de justification que par rapport à la méthode choisie.

Listing 2.4– Assertions dans une implémentation en SystemC de la racine carrée

```
1  float sqrt(const float m) {
2      assert(m >= 0); // assertion de spécification
3
4      float i = 0;
5      float x1, x2;
6      int j = 0;
7
8      while(i*i <= m) i+=0.1f;
9
10     assert(i > 0); // assertion d'implémentation
11
12     x1 = i;
13
14     for (j=0; j<10; j++) {
15         x2 = m;
16         x2 /= x1;
17         x2 += x1;
18         x2 /= 2;
19         x1 = x2;
20     }
21
22     float ret = x2;
23
24     assert(ret >= 0); // assertions de spécification
25     assert(m >= 1 ? ret <= m : ret > m);
26
27     return ret;
28 }
```

Les assertions peuvent donc exprimer différents points de vue, à différents niveaux. Les assertions liées à l'implémentation ne permettent pas de détecter les divergences entre la spécification d'une application et son implémentation, mais elles ne sont pas faites pour cela. Au contraire, elles constituent un moyen efficace pour les équipes de conception de s'assurer que l'implémentation d'une fonctionnalité reste toujours valide après une phase de raffinement.

Les assertions de spécification sont utilisables pour conditionner les entrées et sorties

d'un modèle, comme c'était le cas dans l'exemple précédent. Ce type d'utilisation est formellement décrit par la *logique de Hoare*.

2.2 Logique de Hoare / Design-by-Contract

La logique de Hoare [7, 73, 94, 155], méthode de vérification formelle basée sur les assertions, et le paradigme de développement qui en découle, le *Design-by-Contract* [113, 21, 86, 102, 173, 11], constituent une évolution très importante en génie logiciel. Ces méthodes ont un réel intérêt pour la conception de circuits. Le flot de conception d'une architecture s'élevant de plus en plus vers un modèle algorithmique exécutable – écrit dans un langage de programmation comme MATLAB, C++ ou SystemC, les méthodes du génie logiciel sont directement transposables aux descriptions comportementales des systèmes. Elles permettent d'en faciliter la vérification, et constituent donc un complément intéressant aux méthodes développées par les concepteurs de circuits. De plus, elles sont aussi transposables au niveau RTL, notamment pour faciliter le processus d'intégration des IPs dans un système.

Logique de Hoare

La logique de Hoare, ou *logique de Floyd-Hoare*, est une méthode formelle permettant de raisonner sur le fonctionnement correct d'une programme logiciel. Elle a été proposée en 1969 par Charles Hoare [76], qui s'est lui-même basé sur les travaux similaires de Robert Floyd publiés en 1967 [60]. La logique de Hoare est basée sur l'utilisation des assertions sous forme de triplet au sein d'une description logicielle. Elle permet de décrire les évolutions possibles d'un programme. Le triplet de Hoare est défini comme suit :

$$\{P\}\ C\ \{Q\} \tag{2.1}$$

P et Q sont des assertions (ou prédicats), respectivement une *précondition* et une *postcondition*, et C est un *programme*. Lorsque le programme est exécuté, la postcondition est certifiée comme vraie si la précondition est respectée. Le triplet de Hoare est considéré comme partiel lorsque le programme peut ne pas se terminer, empêchant la postcondition d'être vérifiée.

Pour chaque triplet, il existe souvent de très nombreuses paires d'assertions pour P et Q, mais toutes ne sont pas intéressantes. Les plus pertinentes concernent les cas ou P est aussi faible que possible, et Q aussi fort que possible (on dit que A est plus fort que B lorsque A implique B). L'équation 2.2 présente un exemple de triplet de Hoare. Dans

cet exemple, si a et b sont des entiers naturels, et si b est strictement positif, alors nous sommes assurés qu'après l'exécution du programme, la variable s sera égale à $a \times b$.

$$\begin{gathered} \{a, b \in \mathbb{N} \wedge b > 0\} \\ i = s = 0;\ while(i \neq b)\ \{i = i + 1; s = s + a\} \\ \{s = a \times b\} \end{gathered} \tag{2.2}$$

La logique de Hoare permet de définir formellement les conditions de fonctionnement d'un programme (les préconditions) ainsi que les transformations qui vont avoir lieu sur les données du programme (postconditions). Ainsi, les spécifications du programme peuvent être traduites sous forme de préconditions et postconditions afin d'être exploitées en vérification formelle.

Design-by-Contract

Le Design-by-Contract (DbC), ou programmation par contrat, est un paradigme de développement logiciel basé sur le principe de la logique de Hoare. Il est l'oeuvre de Bertrand Meyer, qui le décrit une première fois en 1986 [111, 113]. Bertrand Meyer est l'inventeur du langage de programmation Eiffel [112], qui met la programmation par contrat au centre de l'écriture des programmes. Le DbC est une approche systématique pour la spécification et l'implémentation des composants dans un programme orienté objet. Il peut être néanmoins étendu à l'ensemble des langages de programmation et de modélisation (MATLAB, SystemC, C++, ...).

Un contrat logiciel est basé sur le même principe que la logique de Hoare : des assertions spécifient les préconditions et postconditions entourant l'exécution d'un programme (que l'on parle de programme au sens strict, ou de simple procédure). Par exemple, dans le listing 2.5 écrit en langage SystemC, le contrat est défini par les deux assertions. Ainsi, l'exécution de la fonction *pgcd* est soumise à la condition que les paramètres a et b soient strictement positifs. La postcondition, quant à elle, assure que le résultat est bien un diviseur des deux paramètres. Ce procédé permet au composant qui appelle cette fonction de pouvoir être certain que la postcondition sera respectée, du moment que lui-même respecte la précondition. Un contrat obligation/bénéfice est donc implicitement passé entre l'*appelant* et l'*appelé*. Le DbC définit aussi d'autres types de propositions à inclure dans les contrats, comme les *invariants* (variables qui sont assurées comme restant constantes au cours d'un échange) et les *effets de bord* par exemple.

Le DbC met tant l'accent sur les contrats qu'il considère leur définition comme prioritaire

Listing 2.5– Contrat défini dans un appel de fonction en SystemC comportemental

```
1  int pgcd(int a, int b) {
2      assert(a > 0 && b > 0);
3
4      int oldA = a, oldB = b;
5      while (a != b) {
6          if (a > b) a -= b;
7          else  b -= a;
8      }
9
10     assert((oldA % a == 0) && (oldB % b == 0));
11     return a;
12 }
```

sur l'implémentation des fonctionnalités. Cette idée est aussi à la base du développement mené par les tests, ou *Test-Driven Development* (TDD) [110]. Le TDD implique la création des tests unitaires (en programmation logicielle) et des testbenchs (en programmation matérielle) avant d'implémenter les composants du programme. Les tests doivent guider le développement des composants, et non l'inverse. Ainsi, la notion de DbC est parfaitement adaptée au développement de modèles et au raffinement de composants logiciels et matériels.

2.3 Intérêt des assertions pour l'intégration des composants virtuels

La programmation par contrats, élaborée à l'origine pour le développement logiciel, trouve particulièrement son intérêt avec les composants virtuels. Le fait qu'une IP ait un fonctionnement prouvé correct n'assure pas pour autant que son intégration dans le système ne soit pas source d'erreur. En effet, toute IP définit des conditions d'utilisation : les données d'entrée doivent être envoyée à une cadence précise, et ces données doivent être formatées correctement. Par exemple, envoyer une entrée négative à une IP effectuant le PGCD défini dans le listing 2.5 peut conduire à un comportement indéterminé si l'IP n'est pas prévue pour gérer une telle situation (afin par exemple de limiter son coût matériel). Comme le montre la figure 16, ce genre de mauvaise utilisation ne conduit pas à un arrêt du système comme ce peut être le cas avec un programme exécutable. L'erreur génère un résultat indéfini, mais qui correspond à un nombre quelconque, et qui sera donc potentiellement utilisé par le reste du système sans être considéré comme faux. La détection de cette erreur ne se fera alors que bien plus loin dans la hiérarchie des composants. Par conséquent, remonter à sa source devient rapidement très long et coûteux.

La logique de Hoare et le Design-by-Contract s'imposent naturellement pour vérifier de telles conditions, et ce autant au niveau système qu'au niveau RTL. Le contrat obliga-

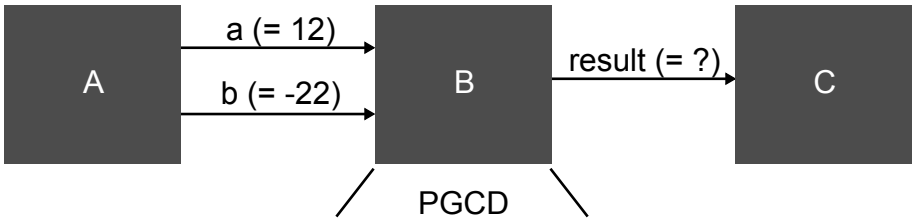

FIGURE 16 – La mauvaise utilisation d'une IP peut conduire à des résultats indéterminés

tion/bénéfice entre le système et le composant virtuel à intégrer permet de rapidement détecter une mauvaise utilisation ou réutilisation du composant. Dans cette optique, les préconditions fournies dans la description d'une IP – quel que soit le niveau d'abstraction – jouent un rôle fondamental pour faciliter la vérification du processus d'intégration.

3 Transformation des assertions de niveau RTL

Du fait des limitations des analyseurs logiques intégrés, plusieurs travaux existent dans la littérature pour transformer les assertions décrites au niveau RTL en moniteurs synthétisables. Ceux-ci réalisent les même vérifications que les assertions correspondantes non plus en simulation mais directement au sein du circuit en fonctionnement. Cela se rapproche du fonctionnement des analyseurs logiques. Toutefois, les assertions peuvent être plus complexes qu'avec ces analyseurs car elles reposent sur le même langage qui est utilisé lors de la vérification en simulation. En effet, les possibilités offertes par les langages de propriété temporelles sont vastes, permettant la synthèse de moniteurs intégrables plus sophistiqués que ceux proposés par les analyseurs logiques. En PSL par exemple, les conditions peuvent être composées à la fois d'opérations arithmétiques, logiques et de relations temporelles. Il est d'ailleurs notable que les évolutions successives des conditions de triggers des analyseurs logiques, notamment concernant *SignalTap II*, font penser que ceux-ci finiront sans doutes par utiliser directement un langage d'assertions normalisé comme PSL.

L'origine de la plupart des travaux effectués sur la transformation des assertions RTL en moniteurs matériels se trouve en partie dans le logiciel *RuleBase* [16, 14]. Cet outil industriel de vérification formelle fut conçu par IBM dans les années 1990. Il prend en entrée un ensemble de propriétés temporelles écrites en langage RCTL (*Region Computation Tree Logic*) [15]. RCTL se base sur les expressions CTL, ou *Computation Tree Logic* [8], en y ajoutant la puissance des expressions régulières [105, 170]. Le CTL est une façon de définir une propriété temporelle utilisée par les approches de vérification formelle, notamment celles axées sur le *model checking*. Le listing 2.6 montre la différence entre une assertion CTL et la même assertion exprimée en RCTL. L'assertion développée dans cet exemple postule qu'un signal de requête (*request*) doit être obligatoirement suivi d'une

confirmation (*ack*) dans les trois cycles d'horloge qui suivent. Le langage CTL impose de préciser chaque possibilité de façon individuelle : il y a donc un cas différent pour tester la confirmation un cycle d'horloge après, deux cycles après, ou trois cycles après. Si l'on imagine la même condition pour trente cycles d'horloge, les limites de ce genre de langage apparaissent rapidement.

Listing 2.6– Une meme propriété temporelle exprimée en CTL et RCTL

```
1 CTL : AG(request -> AX(ack) || AX(AX(ack)) || AX(AX(AX(ack))))
2 RCTL: AG(request -> next_event_f(clk)[1..3](ack))
```

RCTL permet de rendre les conditions plus concises et compréhensibles grâce à l'utilisation d'expressions régulières. Les propriétés temporelles exprimées à l'aide de ce langage permettent la vérification formelle d'une architecture RTL. La version spécifique du langage RCTL utilisé par *RuleBase* s'appelle *Sugar* [13] et a été standardisée par IEEE en 2005 pour devenir le langage PSL (*Property Specification Language*) [R]. Des langages similaires sont apparus à la même époque, comme *ForSpec* [59] d'Intel ou *OVA* [149] de Synopsys.

Plusieurs équipes de recherche ont travaillé sur la transformation des propriétés temporelles en moniteurs matériels.

L'histoire de RuleBase est à la base de l'outil *FoCs*, (*Formal Checkers*) [1, 52], développé par IBM. Cet outil fut le premier, à notre connaissance, à transformer une assertion exprimée en langage dédié en une description de niveau RTL. Ce travail basé sur le langage Sugar de RuleBase constitue donc une extension à cet outil permettant la vérification fonctionnelle des designs en plus de la vérification formelle. Toutefois, FoCs a été pensé pour la simulation. La figure 17, issue de [1], présente le flot de vérification pour lequel FoCs est prévu. Les moniteurs générés vérifient le respect des propriétés temporelles exprimées en RCTL pendant la simulation du système. Toute erreur détectée par un moniteur fait passer un signal de sortie de ce moniteur de 0 à 1. Cette approche facilite le travail des équipes de conceptions. Néanmoins, elle est limitée dans son utilisation à l'utilisation d'un simulateur.

En 2003, Márcio Oliveira publia dans [121] la spécification d'un nouveau langage de propriétés temporelles, PREMiS (*Pipelined Regular Expression Monitor Specification*). Ce langage est dédié aux interfaces, il est imaginé pour décrire facilement les protocoles de communication entre composants. PREMiS est aussi pensé pour être entièrement synthétisable, ce qui n'est pas le cas de PSL. Toutefois, l'introduction d'un nouveau langage ne mène que rarement à son utilisation, qui plus est lorsqu'il n'est pas supporté en amont par une société reconnue. Il n'existe plus à notre connaissance de travaux portant sur ce langage, ni de cas d'utilisation dans un système industriel.

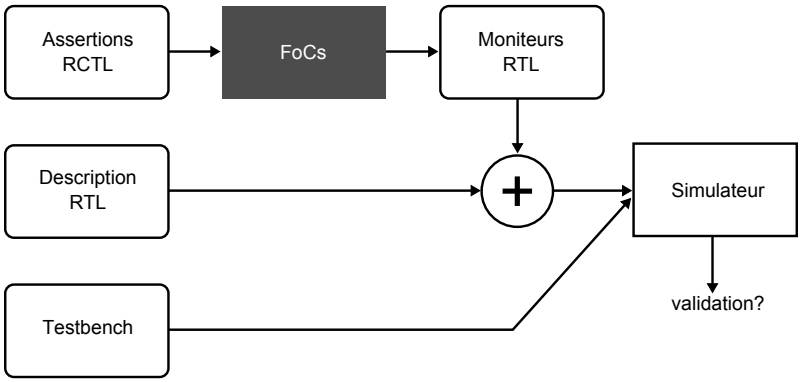

FIGURE 17 – Principe de fonctionnement de l'outil FoCs, tel que donné par ses auteurs

Pellauer et al. proposent une méthode [157] pour transformer des assertions décrites dans le langage SVA (*SystemVerilog Assertions*) en moniteurs synthétisables utilisant le langage *Bluespec SystemVerilog* (BSV). La méthode traditionnelle de synthèse des assertions consiste à appliquer des techniques de logique temporelle directement sur un signal d'horloge. A l'inverse, le langage BSV permet une modélisation de haut-niveau, sans notion de temps (cf. listing 2.7). Le modèle est transformé en description matérielle temporisée par un ordonnanceur lors de la synthèse HLS. Le SystemVerilog quant à lui propose les mêmes fonctionnalités sémantiques que le PSL, la seule différence venant de sa syntaxe inspirée du Verilog. Les assertions sont donc dépendantes d'un signal d'horloge, comme en PSL. Ces travaux adressent donc un problème peu commun : transformer un ensemble d'assertions synchrones (SVA) en «règles» non-temporelles (BSV). Une règle, ou action atomique gardée (*guarded atomic action*) est une opération qui n'a lieu que si une condition donnée est vraie. Chaque assertion est transformée sous la forme d'un ensemble de machines d'état : une FSM principale qui contrôle les étapes de la séquence temporelle donnée par l'assertion, et une FSM secondaire par étape. Michael Pellauer propose plusieurs méthodes d'optimisations pour générer des circuits aussi peu coûteux que possible. Il avance l'idée que les moniteurs matériels générés ne sont pas forcément destinés qu'aux phases de test et debug du circuit, mais qu'ils peuvent être utiles dans le design final afin de permettre la vérification du circuit après commercialisation (*in-field debugging*), ou agir pour améliorer la tolérance aux fautes de l'architecture.

La synthèse des actions atomiques gardées en moniteurs matériels est aussi étudiée par Rosenband [161, 160]. Il propose un nouveau langage dédié à la description des règles, ainsi qu'une méthode de composition de ces règles pour minimiser le nombre de machines d'état générées, et produire des designs moins onéreux que ceux obtenus à l'aide des travaux de Pellauer.

L'équipe de Dominique Borrionne et Katell Morin-Allory du laboratoire TIMA a aussi proposé un outil de génération de moniteurs, Horus [115, 116, 114]. Cet outil dont le point d'entrée est le langage PSL est comparable à FoCs. Il génère des architectures beaucoup

Listing 2.7– Un contrôleur de cache en BSV

```
1  // Envoie toutes les données du cache en mémoire
2  rule sync_cache(state == Synchronize);
3      case (cache[index]) matches
4          tagged Valid {.tag, .data, .isDirty}:
5              if (isDirty) begin
6                  writeToMemory({index, tag}, data);
7                  notDirty(index);
8              end
9          default:
10             noAction;
11     endcase
12     state <= (index == 'MAX_ADDRESS)? Ready : Synchronize;
13     index <= index + 1;
14 endrule
```

plus compactes que FoCs pour les assertions complexes. Cette efficacité diminue toutefois légèrement pour les assertions les plus simples [23]. L'outil a été validé par l'outil de vérification formelle PVS [154], ce qui permet de s'assurer de la validité et de la fiabilité des circuits générés.

Finalement, la plus importante avancée dans le domaine a été proposée par Marc Boulé [27, 28, 23, 25, 24, 26, 30, 29, 31]. Il propose un autre outil de génération de moniteurs à partir d'assertions PSL, nommé MBAC. Son objectif est d'étendre la génération des moniteurs matériels à l'ensemble de la syntaxe PSL, et non un sous-ensemble comme le fait FoCs. De plus, la génération des moniteurs utilise une approche basée sur les automates [23], et non une approche modulaire comme le fait Horus. L'approche modulaire consiste à implémenter chaque opérateur PSL dans un module (composant) dédié. Ces modules sont ensuite reliés les uns aux autres pour former une assertion PSL complète. Les modules ont une interface prédéfinie, comportant généralement un signal d'activation et un signal de résultat. Chaque module active le suivant lorsqu'une partie de l'assertion est vérifiée, et le résultat du dernier module de la chaîne constitue le signal d'erreur. Contrairement à l'approche modulaire, l'approche basée sur les automates essaie de construire un unique module dédié à une assertion complète. Cela permet de réduire le nombre de modules, et de permettre plus d'optimisations. FoCs semble aussi utiliser cette méthode, bien que très peu d'informations soient accessibles, étant un outil industriel. L'approche basée sur les automates permet à l'auteur d'appliquer de très nombreuses optimisations sur les automates construits avant de les synthétiser en architectures RTL. Les états redondants peuvent être supprimés, certaines branches fusionnées, et les conditions de passage d'un état à l'autre peuvent être modifiées tant que la correspondance mathématique à l'assertion initiale est conservée. La figure 18, issue de [23], montre la transformation d'une assertion PSL en automate, et l'architecture matérielle associée. Chaque étape de l'automate donne lieu à la génération d'une bascule synchrone dont l'entrée est conditionnée par la sortie de

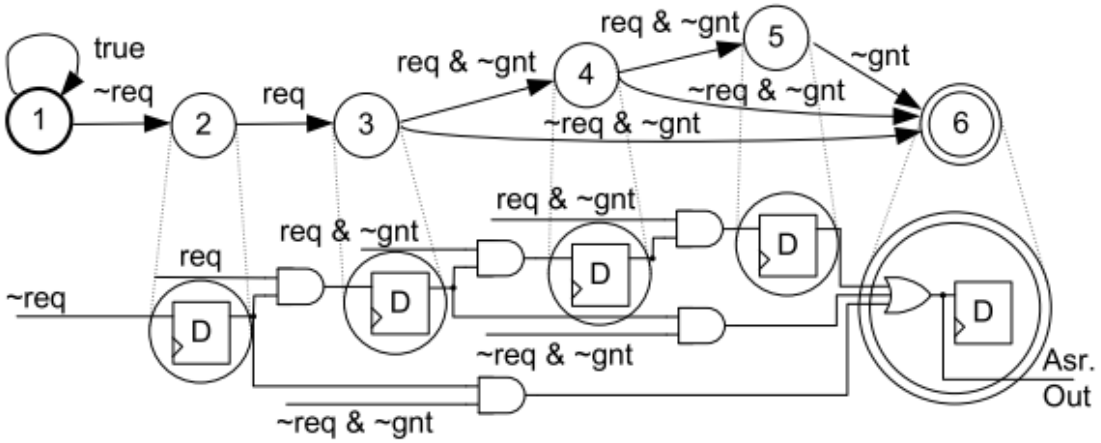

FIGURE 18 – Conversion d'un automate en architecture RTL (Boulé et al.) à partir de l'assertion `assert always {~req ; req} |=> {req[*0:2] ; gnt};`

la bascule précédente et par la condition de franchissement de l'étape. Cette construction permet de gérer toutes les séquences possibles.

4 Transformation des assertions de niveau système

En parallèle de ces travaux sur les méthodes de vérification, des méthodes et des outils ont été proposés afin de faciliter le passage des applications du niveau système au niveau RTL sous contraintes. Les outils de synthèse HLS permettent le raffinement automatique et sous contrainte d'une description de niveau système vers le niveau RTL. Ces derniers ont pour objectif de concevoir une architecture matérielle implantant l'application, respectant les contraintes spécifiées tout en minimisant l'empreinte silicium. Ces outils de HLS n'ont pas été pensés pour la vérification des architectures matérielles générées.

Plusieurs travaux s'attachent à vérifier le processus de synthèse lui-même, afin de prouver formellement que les composants générés sont corrects par construction [118, 119]. La vérification du processus de HLS est pourtant un problème disjoint de la vérification des architectures générées. En effet, une erreur de conception peut subsister dans un modèle comportemental avant sa synthèse. De plus, un concepteur peut avoir fait de mauvais choix de contraintes pour guider la synthèse, utilisant par exemple un nombre de bits insuffisant pour certains registres. Détecter facilement ce type d'erreur lors de la simulation ou de l'émulation de la description matérielle générée peut permettre de remonter rapidement à la section de code comportemental à l'origine de l'erreur, avant que le coût lié à sa correction ne devienne trop important. Or, les outils de HLS actuels, à notre connaissance du moins, ne cherchent pas à produire des architectures faciles à vérifier, mais s'orientent

uniquement vers les performances et le coût des architectures [109, 49, 43]. Cette vision unilatérale de la génération d'architecture commence à être remise en question [95, 67]. Un concept prend petit à petit de l'ampleur : la synthèse orientée pour la vérification (*synthesis-for-verification*, terme employé pour la première fois en 2006 dans [67]).

L'un des plus anciens articles traitant de la vérification au niveau RTL dans un contexte de synthèse HLS a été écrit par Hemmert et al [75]. Ils décrivent un procédé pour inclure des points d'arrêt dans la description comportementale du système. Ces derniers sont alors transformés en moniteurs synthétisables qui mettront en pause le système au moment opportun. Le traitement massivement parallèle propre aux systèmes matériels est alors relié aux instructions séquentielles de la description comportementale. L'utilisateur peut ainsi faire progresser le système cycle par cycle et inspecter le déroulement des opérations. Le débogueur présente à l'utilisateur toutes les opérations en cours au cycle correspondant à la mise en attente du circuit. Les auteurs précisent néanmoins que cette vision parallèle des opérations est contre-intuitive par rapport au traitement séquentiel opéré par les débogueurs logiciels, et peut perturber les utilisateurs. Ils cherchent ainsi à adresser ce problème en divisant un cycle d'horloge en plusieurs paliers, permettant d'observer séquentiellement les opérations effectuées en parallèle sur le circuit.

Toutefois, la transformation HLS des assertions du niveau système en moniteurs matériels est un problème adressé par peu d'équipes de recherche. Gharehbaghi et al. proposent la transformation d'assertions du niveau système en moniteurs à la fois matériels et logiciels. Leurs travaux s'intègrent à l'outil de synthèse ODYSSEY [68]. Cet outil cible la génération d'architectures matérielles basées sur un coeur d'ASIP (*Application Specific Instruction Processor*) [85, 89] à partir d'une description de niveau système et d'un ensemble de contraintes d'intégration. Ainsi, une partie de la description est intégrée à l'aide d'accélérateurs matériels tandis que le reste est exécuté par le coeur de processeur. Les auteurs proposent une syntaxe spécifique pour les assertions. Celle-ci permet de décrire à la fois des assertions de niveau système (comparaison de primitives) et de niveau temporel (simplifié). Le niveau temporel permet de décrire des séquences (simples) d'actions, sans pour autant nécessiter une quelconque horloge, non disponible au niveau comportemental. L'interface des moniteurs est semblable à celle des travaux liés aux assertions RTL : un signal de sortie permet de notifier le système de la détection d'une erreur. Dans le cas présent, le système généré dispose d'une unité de contrôle principal, la MIU (*Method Invocation Unit*). Cette unité déclenche les traitements à réaliser, et analyse les signaux des moniteurs à la fin de chaque traitement (et non en permanence). Lorsqu'une erreur est signalée par un moniteur, une interruption du processeur est déclenchée par la MIU. La figure 19 présente un exemple tiré d'une publication des auteurs où une assertion est transformée en architecture matérielle et en programme logiciel.

Finalement, l'approche la plus proche des travaux que nous proposons est celle de John Curreri, mentionnée dans [51, 50]. Cette méthode s'appuie sur l'outil de synthèse *Impulse-*

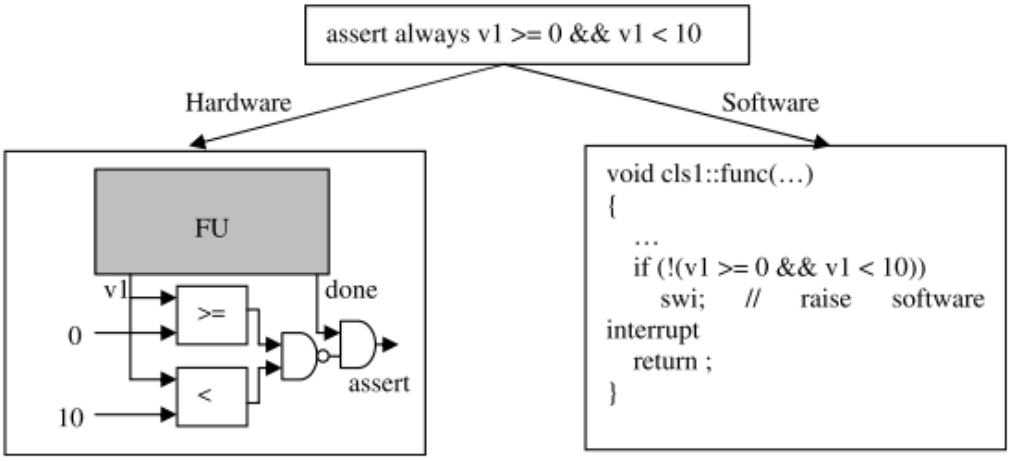

FIGURE 19 – Transformation d'une assertion en moniteurs matériel et logiciel (Gharehbaghi et al.)

C [134], développé par *Impulse Accelerated Technologies*. La description comportementale de l'application génère au travers de l'outil une architecture dédiée de niveau RTL utilisée comme accélérateur matériel pour un processeur généraliste. Ce processeur a pour objectif la gestion des flux de données entre l'accélérateur et le système. Impulse-C, comme la plupart des outils de HLS, ne prend pas en considération les assertions spécifiées dans la description comportementale. Ainsi, l'auteur propose premièrement la traduction des assertions en code compréhensible par l'outil, comme illustré dans la figure 20. Une assertion est alors scindée en deux parties : une partie qui sera implantée matériellement (RTL) et une couche logicielle en charge de la gestion du traitement lors d'une détection d'erreur. La vérification de la condition de l'assertion est effectuée dans un branchement conditionnel $if...then$. Si cette condition est vraie, le processeur est notifié via l'appel de fonction `co_stream_write()` propre à l'API de l'outil Impulse-C. Cela permet de faire passer sur un bus système le fait qu'une erreur ait été détectée. Un numéro d'identification du moniteur est ajouté dans le message.

La figure 21 montre la décomposition des assertions au sein des couches logicielles et matérielles. Les moniteurs matériels notifient la couche logicielle des erreurs détectées en passant par l'interface de communication commune à l'architecture. Le CPU exécute alors la fonction définie : ici l'écriture dans un fichier de l'évènement d'erreur survenu. Il est intéressant de constater que cette fonction peut être aisément adaptée aux besoins du concepteur, et proposer un traitement différent.

John Curreri [51] met en avant le fait que les outils de HLS sont désormais suffisamment avancés pour gérer eux-même l'implémentation des assertions sous forme matérielle. Les techniques de traitement parallèle des opérations utilisées dans les outils de HLS permettent de s'affranchir des latences que peuvent induire les moniteurs lorsqu'ils sont placés au sein des machines d'état. La figure 22, issue de [51], montre le flot d'exécution

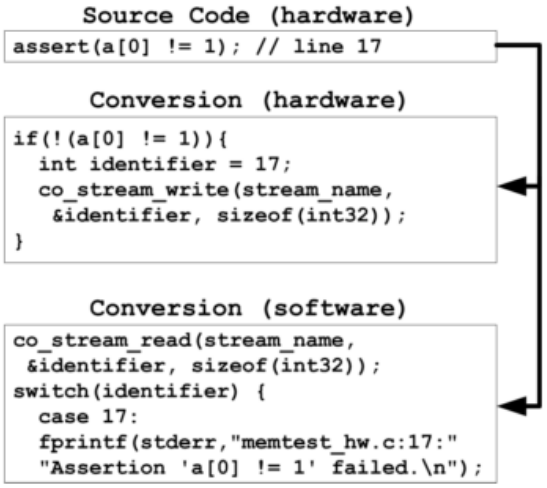

FIGURE 20 – Conversion d'une assertion système en code compréhensible par l'outil de HLS ImpulseC (Curreri et al.)

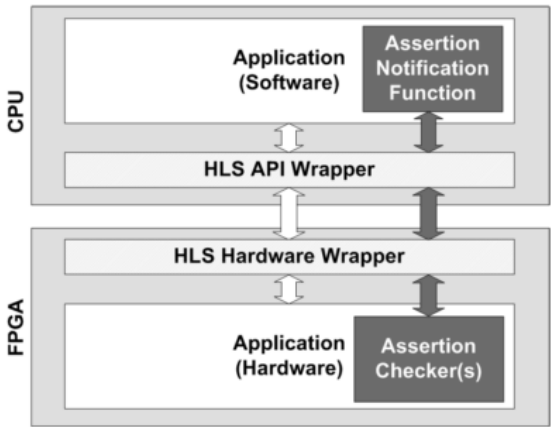

FIGURE 21 – Couches matérielles et logicielles pour la gestion des assertions (Curreri et al.)

d'un algorithme simple, avec le surcout en nombre d'étapes (donc la latence) introduit par la vérification d'une condition d'assertion dans deux cas : en traitement séquentiel de l'assertion (à gauche) puis en traitement parallèle (à droite). Les outils de HLS existants détectent facilement les possibilités de traitement parallèle, et la latence introduite est donc faible. Si ces outils ne supportent pas officiellement les assertions, une transformation *source-to-source* de ces assertions peut suffire.

5 Conclusion

Deux observations sont à l'origine des travaux présentés dans les chapitres suivants :

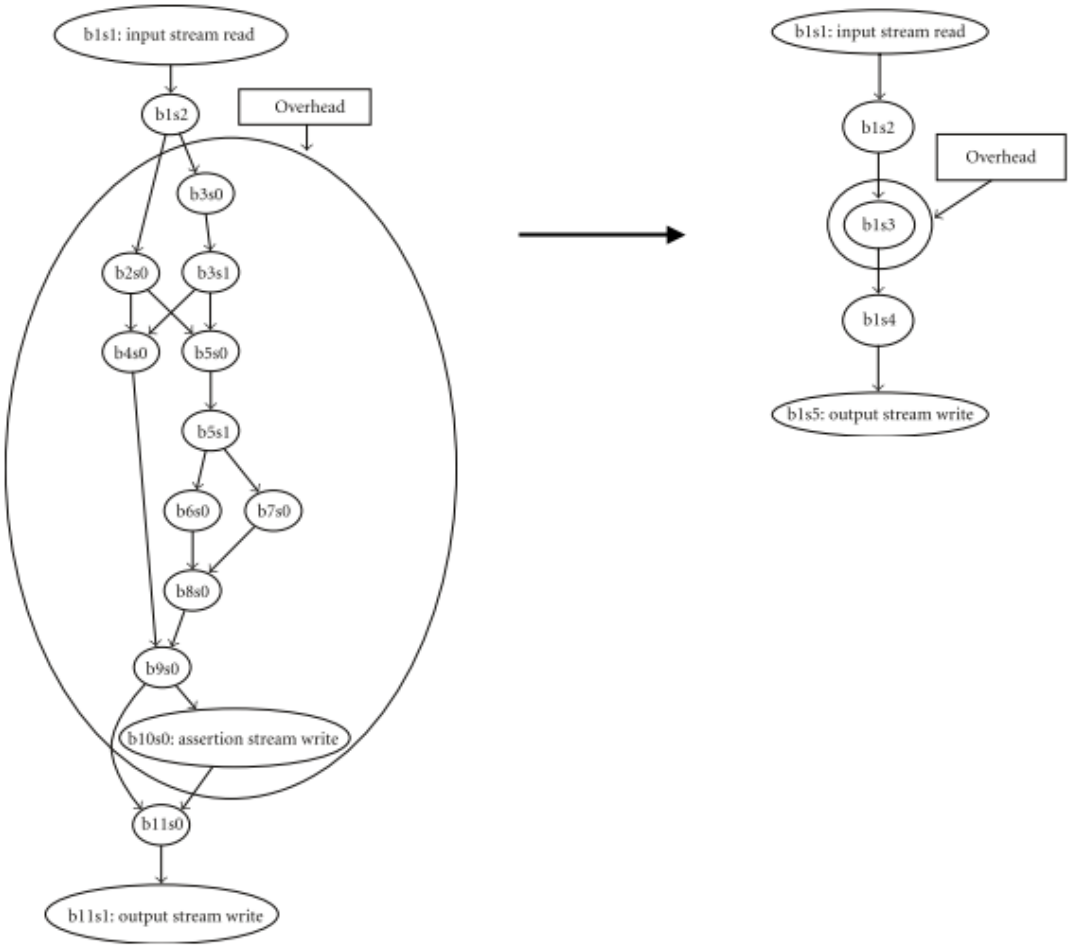

FIGURE 22 – Utilisation du parallélisme matériel pour réduire le coût en latence des moniteurs générés par HLS (Curreri et al.)

– Premièrement, peu de contributions visent la propagation des assertions entre les niveaux de raffinement (cf. figure 23). De plus, ces travaux se concentrent uniquement sur la synthèse de moniteurs intégrés au système matériel, à partir des assertions comportementales. Cette technique autorise l'utilisation de ces assertions en émulation comme en simulation, mais ajoute un surcoût matériel au circuit et supprime le formalisme des assertions. Il n'existe aucun travail à notre connaissance visant la transformation des assertions de niveau système en assertions de niveau RTL (exprimées en langage formel comme le PSL par exemple).

– Deuxièmement, dans l'ensemble des travaux visant à générer des moniteurs matériels à partir d'assertions, qu'elles soient de niveau RTL ou système, l'accent est mis sur la façon de générer ces moniteurs, mais peu souvent sur ce qu'ils peuvent apporter aux concepteurs de circuits. Ainsi, la plupart des travaux ne proposent qu'un signal de sortie binaire pour les moniteurs, parfois relié à un contrôleur, mais dont l'utilisation est laissée libre. John Curreri ajoute toutefois une notion de rapport d'erreur pour les moniteurs. Ces derniers communiquent les erreurs détectées au système par un bus, en y

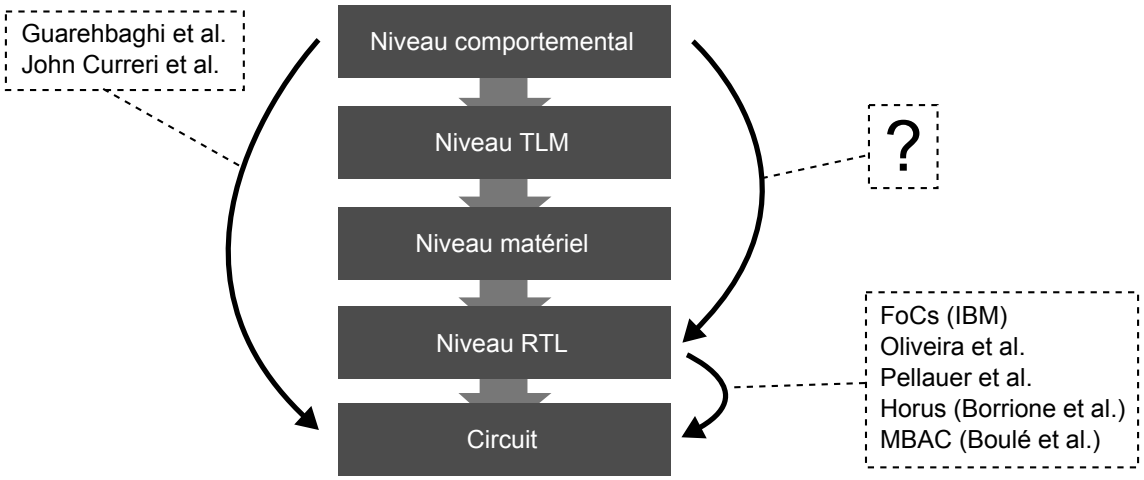

FIGURE 23 – Flot de conception et apports des travaux présentés

ajoutant un numéro d'identification. Cela permet de laisser le système enregistrer toutes les erreurs survenues pendant une période pour y accéder plus tard par exemple.

Les travaux présentés dans les sections qui suivent sont divisés en deux contributions distinctes. La première est une méthodologie pour la détection et la transformation des assertions systèmes d'une description comportementale en assertions équivalentes au niveau RTL, de façon générique et indépendante de toute technologie ou outil. Cette méthode vise la propagation des assertions et de leur formalisme entre les niveaux d'abstraction du flot de conception. La seconde contribution cherche à pousser jusqu'au bout la notion de rapport d'erreur pour les moniteurs intégrés au circuit. Un gestionnaire d'erreurs est proposé. Chaque erreur détectée par un moniteur donne lieu à un rapport complet mentionnant l'identification du moniteur ainsi que les valeurs courantes de plusieurs variables spécifiées par l'utilisateur (que l'on appelle contexte d'exécution).

Chapitre 3

Moniteurs pour la simulation

Sommaire

Dans ce chapitre, nous allons détailler la première contribution réalisée dans le cadre de l'amélioration du processus de vérification des systèmes conçus à l'aide d'outils de synthèse d'architectures. Cette contribution permet l'automatisation de la propagation des assertions décrites au niveau comportemental en assertions équivalentes au niveau RTL (au sein de l'architecture générée à l'aide d'un outil de HLS). Les assertions supportées par ce processus de transformations sont soit explicitement décrites par le concepteur dans la description comportementale, soit issues des contraintes imposées sur le processus de synthèse HLS.

1 Introduction

La synthèse d'architectures [109, 49, 47] est un processus automatisé qui interprète une description algorithmique d'une application et génère une architecture matérielle au niveau transfert de registres (*Register Transfer Level*, RTL), sous contraintes.

Les assertions sont couramment utilisées en amont et en aval de l'étape de synthèse d'architectures (aux niveaux ESL et RTL). Les propager au sein du flot permet de simplifier et d'accélérer les étapes de vérification réalisées en aval. De plus, la propagation des assertions renforce la cohérence des vérifications entre les différents niveaux d'abstraction. Par exemple, une assertion sur les E/S du circuit à haut niveau doit être propagée jusqu'au niveau RTL afin de vérifier que le système dans lequel le composant est placé respecte les hypothèses de fonctionnement.

Le besoin de propager les assertions entre les niveaux d'abstraction est renforcée par la réutilisation interne et le commerce des composants virtuels (IP). En effet, à l'origine ces derniers étaient conçus manuellement et diffusés à bas niveau abstraction (hard, firm, soft). Toutefois, l'émergence des outils de synthèse d'architectures a permis de remonter leur niveau d'abstraction jusqu'au niveau comportemental. Une architecture ainsi conçue peut être réemployée dans de multiples systèmes ou bien la description comportementale peut être employée afin de générer plusieurs architectures dédiées à des systèmes distincts. Afin que la réutilisation d'IP générées soit possible dans de multiples systèmes, il est nécessaire de conserver et vérifier les choix algorithmiques réalisés et les contraintes d'implantation employées. Par exemple, au niveau comportemental, le concepteur a pu spécifier que les entrées du système doivent être toujours positives et non nulles, cela afin de simplifier l'algorithme employé. Cette spécificité de l'algorithme – et donc de l'architecture générée – doit être vérifiée au niveau RTL lors de la simulation du système complet afin de valider la cohérence des données provenant du système.

Cependant, les outils de synthèse d'architectures actuels ignorent les assertions présentes dans la description comportementale. Ils essaient d'améliorer les performances des architectures mais n'ont pas été conçus à l'origine pour la vérification. Cet état de fait implique un travail conséquent pour le concepteur. En effet, s'il souhaite maintenir une équivalence du niveau de vérification entre les différents niveaux d'abstraction, il doit convertir et réécrire ses assertions manuellement.

Propager automatiquement les assertions au niveau système vers la description RTL possède des avantages considérables. Les conditions exprimées au niveau comportemental restent souvent valables pour le niveau RTL. De plus, les assertions (notamment de spécification) peuvent aider l'intégration d'une IP dans un système en vérifiant qu'elle soit correctement utilisée. En définissant des contrats d'utilisation entre les IP générées et le

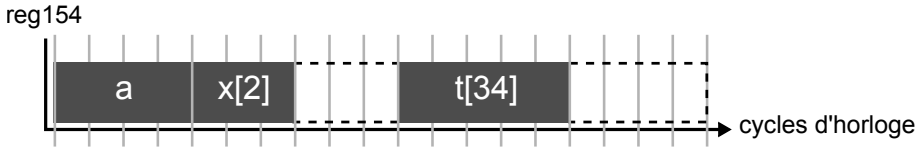

FIGURE 24 – évolution des variables contenues dans un registre au cours du temps

système, les assertions permettent d'assurer l'intégrateur qu'un composant virtuel reçoit des données valides, et par conséquent qu'il fonctionnera correctement. Ce principe, issu du *Design-by-Contract*, trouve ainsi son intérêt au niveau ESL comme au niveau RTL.

Malheureusement, réécrire les assertions au niveau RTL est rendu très complexe à cause des différences de niveau d'abstraction et de langage de description. Par exemple, les listings 3.1 et 3.2 présentent une assertion de niveau comportemental dans un code SystemC et un exemple de ce que pourrait être l'assertion équivalente au niveau RTL (en langage PSL).

Listing 3.1– Une assertion système dans un code SystemC

```
1 void foo(int[] t, int[] x, short a, short b) {
2     t[34] = sqrt(2 * x[2] - b);
3     assert (t[34] < a);
4 }
```

Listing 3.2– L'assertion équivalente au niveau RTL (en PSL)

```
1 assert always
2     (state = s85) -> prev(reg154, 2) < prev(reg154, 8)
```

Cette différence s'explique par le fait que lors de la synthèse d'architecture, une variable de la description comportementale est implémentée au sein d'un registre. Cette donnée en mémoire est disponible uniquement durant le laps de temps où elle est nécessaire aux calculs. Dans la figure 24, le registre *reg154* est utilisé pour mémoriser successivement les variables a, $x[2]$ et $t[34]$. Par conséquent, transformer l'assertion présente dans le listing 3.1 en assertion au niveau RTL nécessite de connaître parfaitement la structure de l'architecture matérielle générée ainsi que son comportement temporel (ordonnancement des calculs et les durées de vie des variables dans les registres).

Parmi les travaux relatés au chapitre précédent qui permettent cette propagation, l'objectif est toujours de générer des moniteurs matériels intégrés au circuit. Le formalisme des assertions se perd donc lors de la transformation du niveau comportemental au niveau RTL. C'est la raison pour laquelle nous avons souhaité décomposer cette synthèse en deux temps. En premier lieu, il s'agit de propager les assertions du niveau comportemental au niveau RTL tout en conservant le formalisme des assertions (elles peuvent alors être exprimées en PSL par exemple au niveau matériel). Dans un second temps, les méthodes

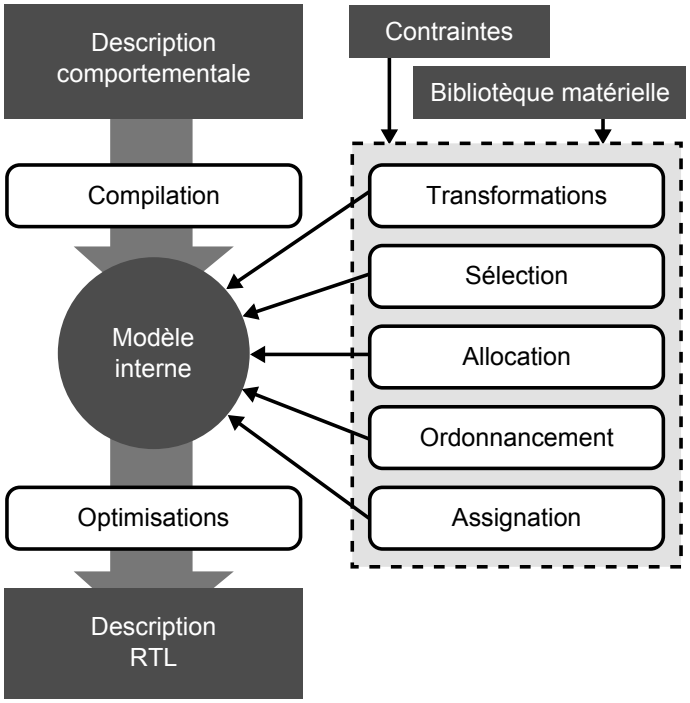

FIGURE 25 – Flot de synthèse d'architectures

existantes de synthèse des assertions du niveau RTL en moniteurs intégrables (cf. IBM [1], Marc Boulé [23] ou Dominique Borrione [22]) sont directement applicables pour générer les moniteurs synthétisables souhaités. Ce chapitre présente une méthode pour réaliser la première partie de la synthèse, qui n'est pas traitée par la littérature.

2 Flot de synthèse d'architectures

Le processus de synthèse d'architectures comprend plusieurs étapes clefs [57, 48]. Le nombre et l'ordre d'exécution de ces étapes varient en fonction des méthodologies. Toutefois, nous allons dans cette partie détailler un flot standard afin de présenter ensuite les modifications apportées pour propager les assertions. L'ensemble des étapes de raffinement automatique est présenté dans la figure 25, décrite par les paragraphes suivants.

L'utilisation d'outils de synthèse d'architecture permet au concepteur de se focaliser sur la fonctionnalité d'un module à concevoir ainsi que sur ses interfaces de communication. Pour ce faire, la définition de l'architecture d'implantation et l'ordonnancement des opérations cycle par cycle sont réalisés sous contrainte lors de la synthèse. Afin de permettre à l'outil de synthèse de construire une architecture matérielle, le concepteur doit fournir en entrée du flot la description algorithmique de l'application, un jeu de contraintes et une bibliothèque de ressources détaillant les caractéristiques de la technologie ciblée. À

partir de ces différentes informations, l'outil de synthèse va enchaîner différentes étapes afin d'aboutir à la génération d'une l'architecture matérielle au niveau RTL.

L'étape de compilation : le point d'entrée du flot de synthèse est une description algorithmique de la fonctionnalité à implanter. Ce comportement est décrit dans un langage de haut niveau (MATLAB, SystemC, C, etc.). La première étape du flot de synthèse réalise la vérification syntaxique et sémantique de la description algorithmique et la traduit en un format intermédiaire propre à l'environnement de synthèse.

L'étape de transformation du modèle interne : les techniques employées durant cette étape sont communes à celles employées pour la compilation logicielle : propagation des expressions constantes, élimination des sous expressions communes, élimination du code mort, suppression des invariants de boucles, etc.

L'étape de sélection et d'allocation des ressources : cette étape a pour objectif de déterminer le type et le nombre des ressources matérielles nécessaires à l'implantation de l'application sous contrainte.

L'étape d'ordonnancement des opérations : cette étape est à l'origine de l'introduction de la notion de temps. L'objectif est d'associer à chaque opération de la description une date d'exécution. Cette étape primordiale se base sur les dépendances existant entre les données ainsi que sur le nombre et le type des ressources de calcul afin d'assigner chaque opération à un cycle de calcul.

L'étape d'assignation : cette étape associe les opérations à exécuter aux unités fonctionnelles allouées. Par ailleurs, les données produites et consommées par les unités fonctionnelles nécessitent la mise en oeuvre d'éléments de mémorisation (mémoires et registres). Une analyse de la durée de vie des données permet de déterminer celles qui peuvent partager un même élément de mémorisation afin de minimiser le nombre de ressources à allouer. Suite à cette analyse, les ressources de mémorisation nécessaires sont instanciées et les variables leur sont assignées. Finalement, les ressources d'interconnexions (multiplexeurs, tri-states) permettant de lier les unités fonctionnelles aux éléments de mémorisation sont allouées. À la fin de cette étape, l'architecture matérielle du chemin de données (partie opérative) est intégralement définie et la machine d'états (FSM) de la partie contrôle peut être construite.

L'étape de génération de l'architecture : le chemin de données de la partie opérative et la machine d'états de la partie contrôle résultant des étapes précédentes sont décrits sous forme de code source de niveau RTL dans un ou plusieurs fichiers.

Le processus décrit ci-dessus est illustré à l'aide de la description comportementale fournie dans le listing 3.3. La représentation sous forme de graphe de cette application est fournie en figure 26.

Listing 3.3– Description de niveau algorithmique de l'architecture à concevoir

```
1 int application(int a, int b, int c, int d) {
2     int t = 3 * a;
3     int s = t + (5 * b) + (b * c) * (13 * d);
4     return s;
5 }
```

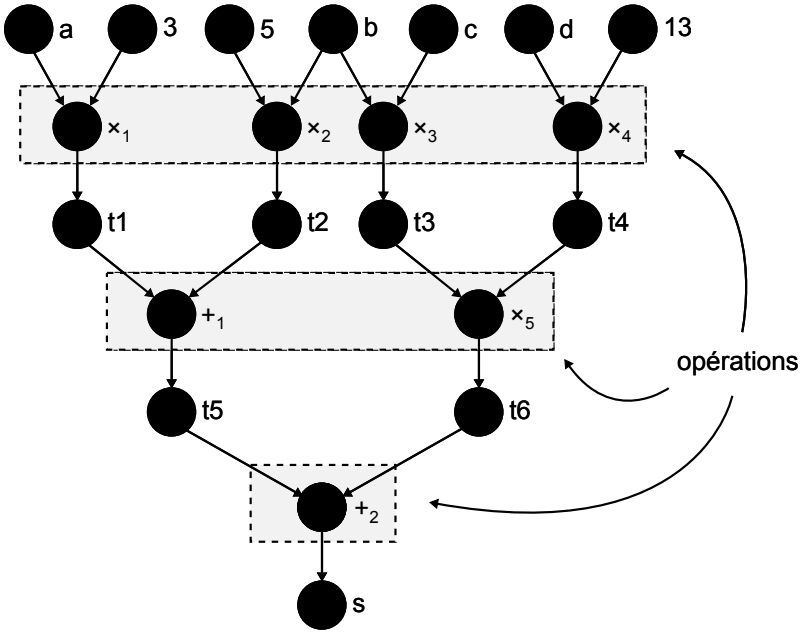

FIGURE 26 – Représentation sous forme de graphe de la description comportementale du listing 3.3

Dans cette application, cinq multiplications et deux additions doivent être exécutées afin de calculer le résultat (s). En fonction des contraintes d'intégration spécifiées par le concepteur, le nombre de ressources matérielles et les performances temporelles seront différentes. La figure 27 détaille l'ordonnancement des opérations et leur assignation lorsque le processus de synthèse est contraint respectivement par une contrainte de latence de valeurs 6, 5 et 4 cycles d'horloge.

En fonction des résultats d'ordonnancement et d'assignation, le flot de synthèse génère l'architecture matérielle correspondante. Cette architecture est composée d'opérateurs

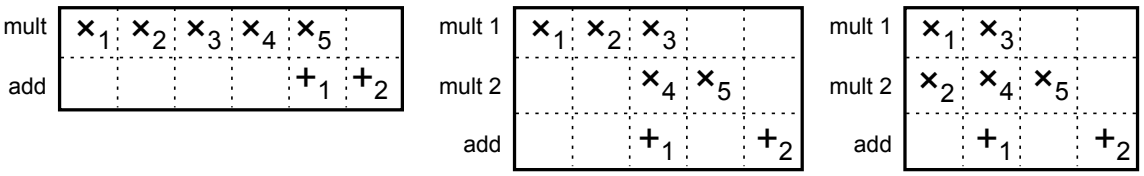

FIGURE 27 – Ordonnancements et assignations des opérations : synthèse contrainte par une latence de 6, 5 et 4 cycles d'horloge

arithmétiques et logiques et de ressources d'interconnexion. Il n'existe plus de relation directe entre la description comportementale et l'architecture matérielle qui l'implante sous contrainte : les données des variables algorithmiques de la description comportementale sont stockées dans des registres de l'architecture matérielle tant qu'elles sont nécessaires, puis le contenu de ces registres est remplacé.

Cette projection spatio-temporelle des variables algorithmiques au sein de l'architecture rend complexe la propagation et l'écriture au niveau RTL d'une assertion comportementale telle que `assert(s != t)`. En effet, dans l'exemple courant, les durées de vie des données s et t au sein de l'architecture semblent non recouvrantes (t devient inutile une fois que s est calculée). Afin de rendre l'évaluation de l'assertion possible, il est nécessaire de mémoriser la donnée t jusqu'à la disponibilité de la donnée s. Toutefois, dans notre cas, cette mémorisation doit être réalisée de telle manière qu'elle ne modifie pas l'architecture matérielle (structure et coût silicium).

Cette conclusion est renforcée par le fait qu'un changement de contrainte lors de la synthèse d'architecture induit une réécriture de l'ensemble des assertions car le nombre d'opérateurs, de registres et les dates d'exécution varient. La propagation manuelle des assertions implique donc alors un fort risque d'erreur et peut s'avérer très chronophage, d'où la nécessité d'automatiser le processus.

3 Flot de synthèse avec propagation des assertions

Afin d'éviter la tache fastidieuse pour les concepteurs de transformer et réécrire les assertions comportementales au niveau RTL, avec toutes les contraintes que cela implique, nous proposons une modification du flot de synthèse d'architectures. Cette modification permettra :

- la transformation des assertions décrites dans la description comportementale,
- l'identification de certains problèmes potentiels induits par les contraintes de synthèse imposées par le concepteur en générant les assertions adéquates.

Les assertions générées automatiquement au niveau RTL ont pour objectif d'améliorer la vérification du circuit en simulation. Le flot de génération proposé pour la transformation des assertions de la description comportementale en assertions dans la description matérielle est divisé en sept étapes. Ces étapes s'interfacent avec le flot de synthèse d'architectures traditionnel. La majorité d'entre elles sont exécutées en parallèle des étapes de synthèse d'architecture, toutefois il est nécessaire de synchroniser ces deux flots comme cela sera expliqué dans la suite de ce chapitre.

Le flot de traitement des assertions se base sur le modèle interne utilisé par la majorité des outils de synthèse d'architectures. Il se compose d'un graphe bi-partite orienté acyclique ou *Directed Acyclic Graph* (DAG) [87]. Voici donc un rapide rappel sur la théorie des graphes.

3.1 Éléments de théorie des graphes

Généralités

Un graphe, de façon générale, est un ensemble de *noeuds* reliés ou non par des *liens*. Ces liens sont appelés *arcs* lorsqu'ils sont orientés, c'est à dire lorsque un lien entre deux noeuds n_1 et n_2 relie soit n_1 à n_2, soit n_2 à n_1. On parle dans ce cas de graphe orienté. La notion de cycle, quant à elle, renvoie au fait qu'il peut être possible en partant d'au moins un noeud de revenir à ce noeud en traversant le graphe. Par opposition, un graphe acyclique ne présente aucun noeud ayant cette propriété.

Un graphe orienté acyclique (DAG) est défini comme une paire $G = (V, A)$ où $V = \{v_0, \ldots, v_N\}$ désigne l'ensemble des noeuds du graphe, et A l'ensemble des arcs, ou liens orientés. Un arc $a = (v_1, v_2)$ est composé d'un noeud source, v_1, et d'un noeud puits, v_2.

Ensuite, pour chaque noeud du graphe, deux ensembles sont définis, $P(v) \subset V$ et $C(v) \subset V$. $P(v)$ représente l'ensemble des noeuds parents du noeud v, c'est à dire l'ensemble des noeuds tels qu'il existe un arc $a \in A$ dont v soit le noeud puits. De la même façon, $C(v)$ est l'ensemble des noeuds enfants de v, tels qu'il existe un arc dont v soit le noeud source. Ces définitions sont résumées dans les équations 3.1.

$$\begin{aligned} P(v) &:= \{v_1 \in V : (v_1, v) \in A\} \\ C(v) &:= \{v_1 \in V : (v, v_1) \in A\} \end{aligned} \tag{3.1}$$

Graphes et synthèse HLS

Dans le cas d'un modèle interne de synthèse HLS (la représentation compilée d'une description comportementale), l'ensemble fini des noeuds V représente les variables et les opérations de l'application. Ainsi, V se décompose en V_{vars} et V_{oprs} tel que $V = V_{vars} \cup V_{oprs}$ et $V_{vars} \cap V_{oprs} = \emptyset$. L'ensemble des arcs A, quant à lui, représente les dépendances de données existant entre les noeuds du graphe.

Un noeud représente soit une opération (arithmétique ou logique), soit une variable. Le graphe est bi-partite, c'est à dire qu'un noeud de type opération sera toujours précédé et suivi par un noeud de type variable. De plus, l'ordre des opérations est imposé par les liens entre les noeuds. Plus formellement, si v_i et v_j sont deux noeuds opérations, et s'il existe deux arcs $a_{i,k}$ et $a_{k,j}$ reliant ces noeuds par l'intermédiaire d'un noeud variable v_k, alors l'opération représentée par v_j ne peut avoir lieu qu'à une date strictement postérieure à celle de l'opération de v_i. Ce principe est à base de la phase d'ordonnancement des opérations d'une synthèse HLS.

L'ensemble V_{vars} se compose traditionnellement de trois ensembles mutuellement exclusifs, tels que $V_{vars} = V_{io} \cup V_{inter} \cup V_{consts}$. V_{io} est l'ensemble des noeuds variables constituant les entrées et sorties de l'algorithme, V_{inter} représente les résultats intermédiaires des opérations et V_{consts} contient les constantes utilisées dans certaines opérations.

Graphes et assertions

La méthodologie proposée prend en compte la spécificité des assertions. Une assertion est représentée dans le graphe de l'application comme un ensemble de noeuds variables et opérations constituant la condition de l'assertion. Le résultat booléen de cette condition est un noeud variable mais sans pour autant appartenir à V_{io} ou V_{inter}. Ainsi, il est nécessaire d'ajouter à l'ensemble V_{vars} l'ensemble $V_{assertOut}$, regroupant les résultats des évaluations des assertions. On a ainsi dans notre cas $V_{vars} = V_{io} \cup V_{inter} \cup V_{consts} \cup V_{assertOut}$.

Au sein de ce modèle, on définit la notion de *branche d'assertion* qui correspond à l'ensemble des noeuds (variables et opérations) nécessaires à l'évaluation d'une condition d'assertion. Tous les noeuds faisant partie d'au moins une branche d'assertion constituent l'ensemble V_{assert}. De façon similaire, tous les noeuds faisant *uniquement* partie d'une ou plusieurs branches d'assertions constituent l'ensemble $V_{assertPure}$. Au moment de la génération du graphe par l'outil de HLS, les branches d'assertions sont toutefois inconnues, et aussi par conséquent les ensembles V_{assert} et $V_{assertPure}$. La méthode décrite dans la suite de cette section permet d'identifier ces ensembles et de les traiter en conséquence.

Toutes ces notions sont illustrées ensembles dans la figure 28. Cette figure modélise la description comportementale décrite dans le listing 3.4. Les noeuds variables ar_i (*assertion result*) sont les résultats booléens des conditions des quatre assertions. Ils constituent donc $V_{assertOut}$. Dans la suite de ce chapitre, les noeuds blancs représenteront toujours les noeuds utilisés *uniquement* dans les assertions, soit $V_{assertPure}$.

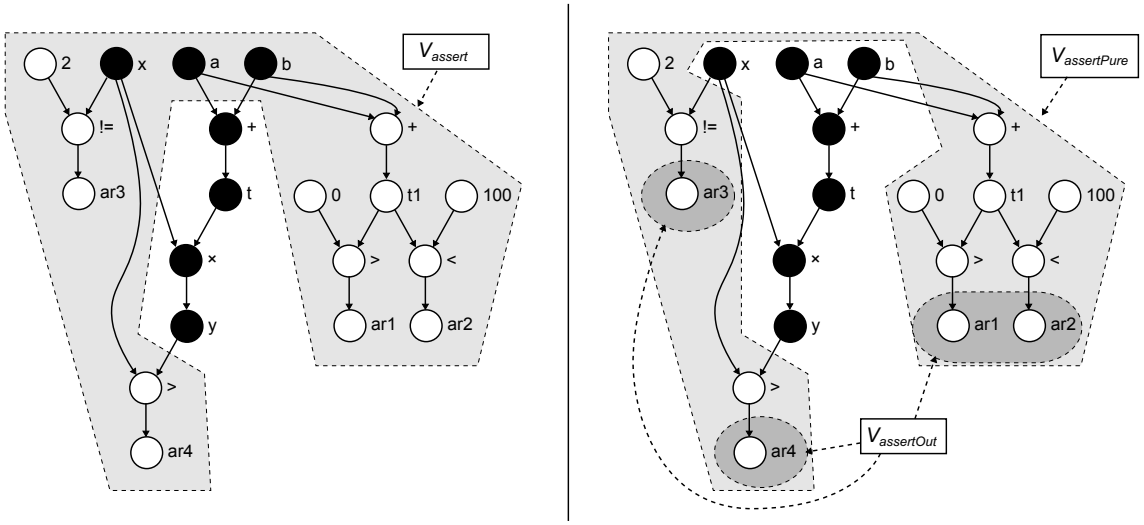

FIGURE 28 – Illustration des ensembles $V_{assertOut}$, V_{assert} et $V_{assertPure}$. Les noeuds noirs participent à l'algorithme de l'application tandis que les noeuds blancs sont *uniquement* utilisés dans les branches d'assertion.

Listing 3.4– Exemple de code comportemental avec trois préconditions et une postcondition

```
1  short foo(int x, int a, short b) {
2      assert(a + b > 0);
3      assert(a + b < 100);
4      assert(x != 2);
5
6      short y = (a + b) * x;
7
8      assert(y > x);
9      return y;
10 }
```

3.2 Synthèse des étapes du flot

La figure 29 présente le flot de conception que nous proposons afin de propager les assertions de niveau système au sein de l'architecture de niveau RTL. Les notions de graphes, noeuds, arcs et branches d'assertions étant connues, les sept étapes du flot de traitement des assertions peuvent être décrites comme suit :

1. **Pré-traitement du modèle interne** - Les assertions *implicites*, non spécifiées par l'utilisateur dans la description comportementale mais liées aux contraintes de synthèse, sont identifiées par une analyse de ces contraintes et du modèle interne. Les branches correspondant à ces assertions sont ensuite insérées dans le modèle interne.

2. **Identification des branches d'assertion** - Le modèle interne est ensuite analysé afin d'identifier les noeuds et les arcs appartenant soit à l'application à intégrer,

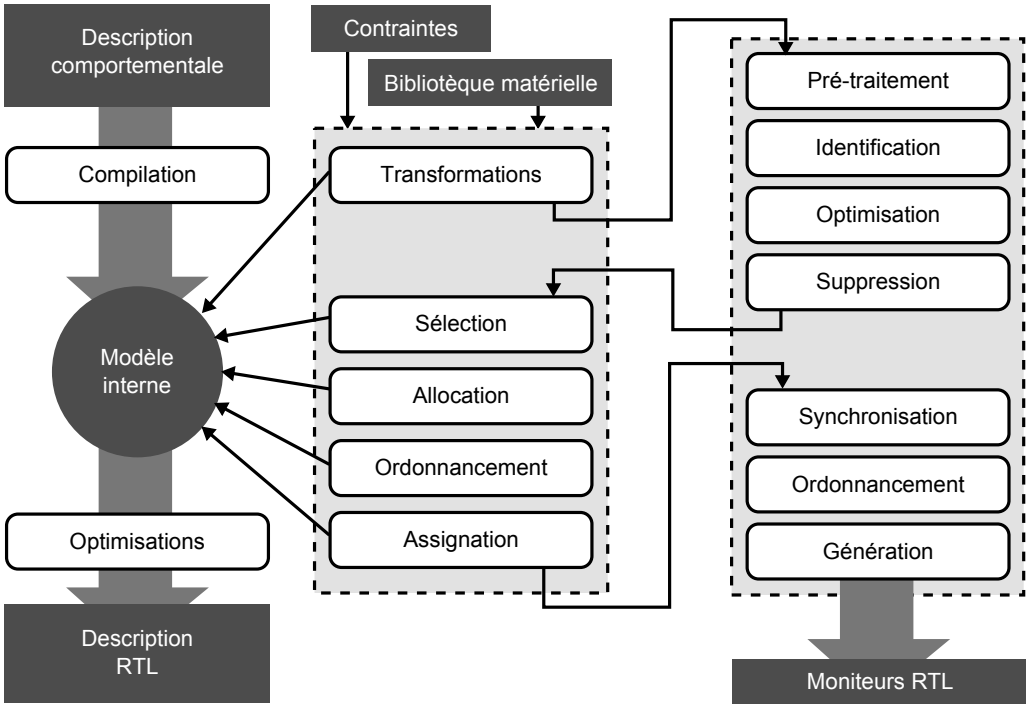

FIGURE 29 – Flot de génération proposée, et inclusion dans le flot de synthèse général

soit aux branches d'assertion, ou bien aux deux. Les noeuds sont alors annotés en fonction de leur appartenance.

3. **Optimisation des branches d'assertion** - Afin de rendre les étapes ultérieures plus robustes aux optimisations effectuées sur le modèle interne, les branches d'assertions partageant des données entre elles sont développées. Les noeuds et les arcs ayant pu être factorisés pour simplifier la synthèse sont alors dupliqués afin de supprimer tout recouvrement.

4. **Suppression des branches d'assertion** - Les branches d'assertion sont retirées du modèle et conservées à part. Un nouveau modèle de type DAG contenant uniquement les branches d'assertion est alors construit. La scission du modèle interne permet de s'assurer que l'outil de synthèse ne traitera pas les opérations liées aux assertions dans le circuit final. Une fois ce traitement réalisé, l'outil de synthèse est invoqué afin de générer l'architecture matérielle sous contrainte (étapes de sélection, allocation, ordonnancement et assignation).

5. **Synchronisation des modèles** - Une fois la synthèse du circuit terminée, tous les noeuds du graphe modélisant l'application possèdent une date de disponibilité ou d'exécution ainsi qu'une ressource matérielle associée. Cette étape de synchronisation des graphes recopie ces informations communes vers le modèle contenant les branches d'assertion. Ainsi, pour chacune des entrées des graphes d'assertion, la ressource matérielle ainsi que la date de disponibilité au sein de l'architecture matérielle est connue.

6. **Ordonnancement des assertions** - A partir des informations collectées dans l'étape précédente, il devient possible de déterminer les dates d'exécution des assertions en fonction de la disponibilité des données dans l'architecture matérielle. Cependant, les assertions ont été spécifiées dans un certain ordre dans la description comportementale. Cet ordre imposé par le concepteur peut être important. C'est la raison pour laquelle les dates d'exécution des assertions sont recalculées si nécessaire afin de respecter cette contrainte.

7. **Génération du code RTL** - Les branches d'assertion sont finalement traduites en langage PSL ou sous forme de machine d'état VHDL en vue de la simulation de l'architecture. En fonction du format choisi par le concepteur, le code est généré au sein du fichier VHDL dans une section non synthetisable ($pragma\ synthesis\ off$) ou bien dans un fichier séparé.

Nous allons dans la suite de ce chapitre détailler les différentes étapes composant le flot de propagation des assertions et exposer les algorithmes utilisés dans les différentes étapes.

4 Étapes de transformation des assertions

Afin d'illustrer la présentation des différentes étapes composant le flot de transformation des assertions, nous allons nous appuyer sur un exemple pédagogique. Pour cela, j'ai choisi la description comportementale spécifiée dans le listing 3.4. Elle est composée d'une expression arithmétique et de quatre assertions (trois préconditions et une post-condition). Le modèle de représentation de type DAG correspondant à ce code comportemental est décrit dans la figure 30.

Dans un premier temps, les branches d'assertions ne sont pas identifiées. Seuls sont connus les noeuds de $V_{assertOut}$, définis lors de la phase de compilation, qui sont donc représentés en blanc sur la figure.

4.1 Pré-traitement du modèle interne

Avant d'identifier les branches d'assertions pour permettre à l'outil de synthèse de générer l'architecture implantant la description comportementale, une première étape d'analyse identifie les sources d'erreurs possibles dans le DAG. L'indetification de ces sources d'erreurs potentielles permettra alors la génération d'assertions spécifiques à la vérification de ces erreurs. Ces assertions n'étant pas directement spécifiées par le concepteur, nous les appelons *assertions implicites*. Les assertions implicites liées aux choix du concepteur

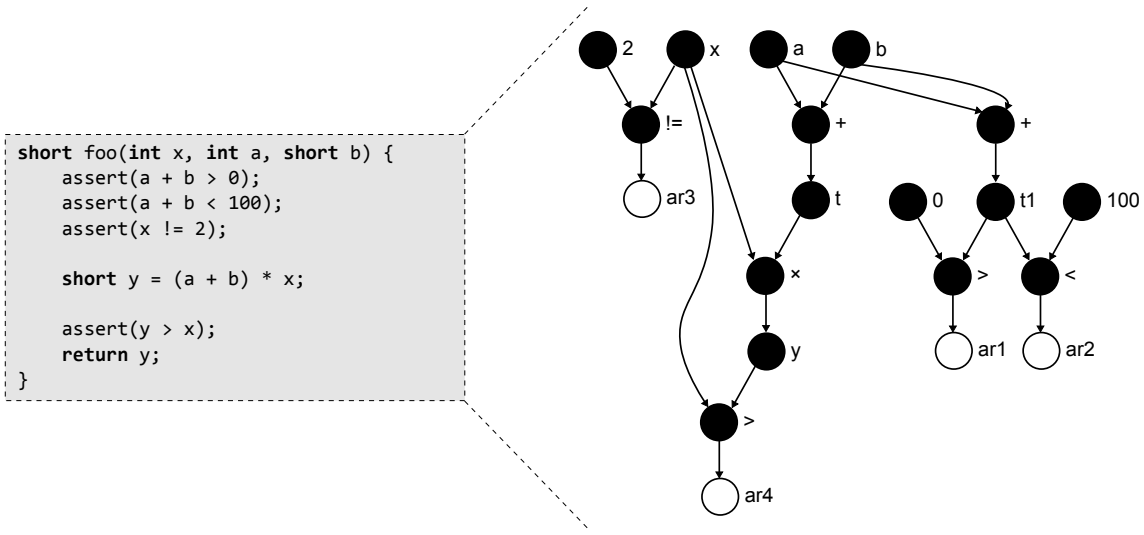

FIGURE 30 – Exemple pédagogique et son modèle DAG associé, avant tout traitement

lors du processus de synthèse de haut-niveau sont ajoutées dans le graphe sous forme de nouvelles branches d'assertion. Ces assertions sont identifiées à partir des contraintes de génération de l'architecture, telles que :

– la dynamique des interfaces d'entrée/sortie du circuit,
– la dynamique des opérations et des ressources matérielles,
– la contrainte de latence d'exécution.

Ces contraintes, qui permettent de paramétrer la génération du circuit, peuvent être source d'erreur par la suite si elles ont été mal identifiées par le concepteur. Ce type d'erreur peut être particulièrement complexe à détecter une fois le circuit simulé. En effet, l'architecture du composant est correcte par construction, mais les opérations internes peuvent produire des résultats non souhaités, par débordement (*overflow*) par exemple. Il est donc nécessaire de transformer les choix liés aux contraintes en assertions. Une simulation du composant avec une bonne couverture de code mettra rapidement en lumière le non-respect de ces assertions par les stimuli.

Nous avons identifié trois types d'erreurs qui peuvent bénéficier de cette identification automatique. Pour des raisons de lisibilité de l'exemple pédagogique, les branches d'assertions implicites rajoutées au graphe ne seront pas conservées par la suite dans cet exemple.

Vérification de la dynamique des interfaces d'E/S

Les architectures matérielles générées automatiquement possèdent généralement moins de ports d'E/S que de variables. Les ports sont multiplexés temporellement afin de recevoir et émettre l'ensemble des informations depuis/vers le système. Toutefois le format des

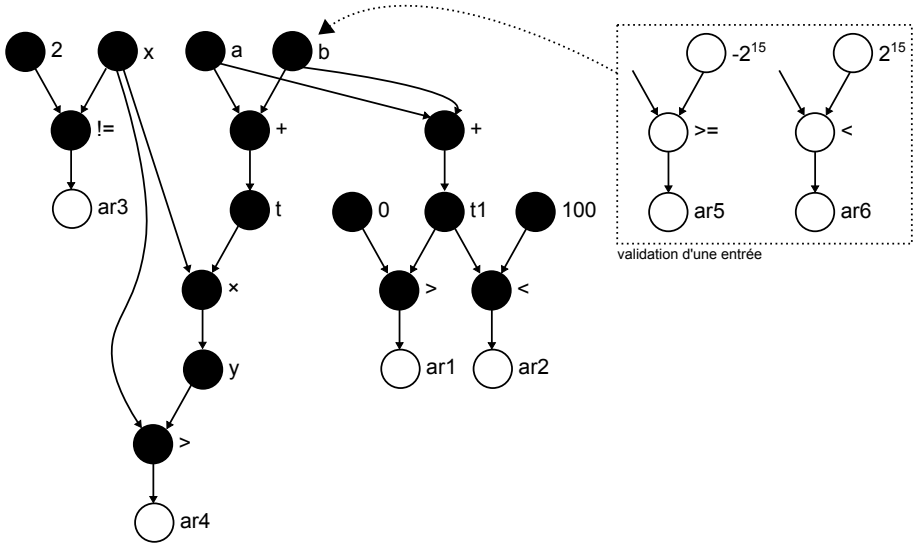

FIGURE 31 – Ajout de deux branches d'assertion pour vérifier la dynamique de l'entrée b

données peut varier au sein de la description comportementale : dans le code de l'exemple pédagogique, la donnée b possède une dynamique inférieure à celles des données a et x. Toutefois ces données peuvent être contraintes à utiliser une unique interface de largeur 32 bits.

Ainsi, si le système dans lequel s'insère le composant transmet une donnée sur 32 bits pour la donnée b, seuls les 16 bits de poids faibles seront mémorisés dans cette dernière. Afin de détecter le mauvais interfaçage du composant avec le système, il semble judicieux de protéger cette donnée b en s'assurant que sa valeur soit bien comprise dans les bornes $[-2^{15}, 2^{15}[$ (dans le cas où b partage le même port d'entrée que a ou x). La figure 31 illustre la création des branches d'assertion permettant de vérifier la dynamique de l'entrée b.

Vérification des débordements des opérateurs arithmétiques

Il existe différentes manières de déterminer la dynamique des données et des opérations lors du processus de synthèse d'architectures. L'approche la plus simple consiste à reporter la complexité de cette tâche sur le concepteur (à l'aide des types de données employés dans la description comportementale). Toutefois, certaines spécifications peuvent être ambiguës et/ou spécifiques à une système. Par exemple dans l'exemple pédagogique, la donnée y de type `short` est le résultat d'un calcul non saturé de données de type `integer`. Ce choix délibéré du concepteur peut viser à optimiser la taille des ressources matérielles employées dans l'architecture. Toutefois, cette hypothèse d'optimisation implicite sur la valeur de y est fortement liée au système dans lequel va être intégré l'architecture. Afin de faciliter la détection d'une erreur à ce niveau dans le système courant (ou dans d'autres systèmes dans lesquels l'architecture serait réutilisée), il est judicieux de vérifier l'espace de variation de

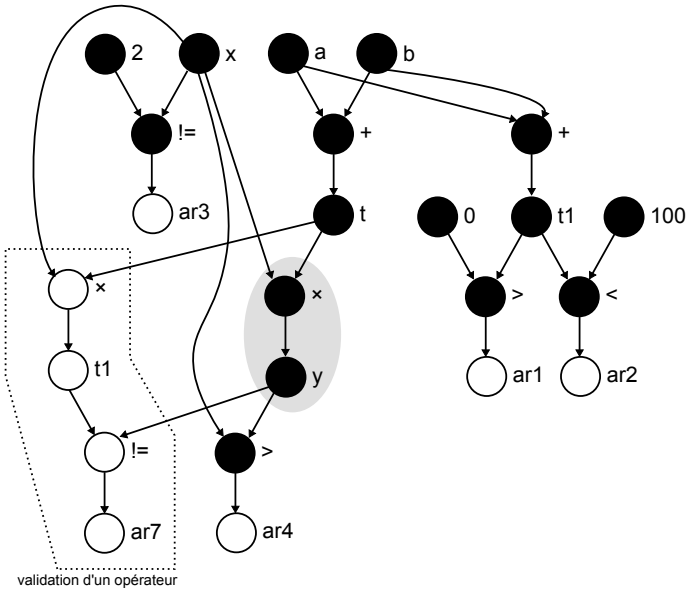

FIGURE 32 – Ajout d'une branche d'assertion pour vérifier le débordement de l'opérateur multiplication dont y est le résultat

y.

Le traitement nécessaire à cette vérification est présenté sur la figure 32. Il n'est pas possible d'appliquer le même genre de traitement que pour vérifier les entrées de l'application. En effet, la valeur de y est stockée dans un registre de taille 16 bits, et l'on souhaite s'assurer que y ne dépasse pas cette plage. Cette vérification est donc impossible à faire directement au niveau du noeud y. La vérification se fait donc sur l'opération dont le résultat est y. Pour ce faire, cette opération est réalisée une seconde fois, dans la branche d'assertion, cette fois sans borne pour le résultat. Il est alors possible de vérifier si le résultat de cette opération est différent de y, ce qui sous-entend que l'opérateur matérielle a débordé.

Des approches automatisées permettant de déterminer automatiquement la dynamique des opérations et des données ont aussi été proposées dans la littérature. Certaines méthodes se basent sur une étude au pire cas des valeurs des données [101]. Cependant ces approches sont généralement sous optimales. Afin de minimiser la complexité matérielle des méthodes de troncature forcée ou guidées (statistiques) ont été proposées [101, 33]. L'inconvénient de ces méthodes provient des risques de débordement des calculs qui peuvent être quantifiés mais pas exclus. En fonction des décisions prises par le concepteur lors de la spécification des contraintes de synthèse, l'architecture matérielle peut dans ce cas ne pas être fonctionnelle. Cette perte de fonctionnalité est alors imputable non pas au processus de synthèse, mais aux choix effectués par le concepteur. Afin d'améliorer le processus de vérification de l'architecture, des assertions peuvent donc être rajoutées automatiquement afin de s'assurer du non débordement des calculs. Ainsi il sera possible d'identifier et de remonter à la cause de l'erreur plus aisément.

Vérification des contraintes temporelles

La latence maximale pour l'exécution d'une itération de l'application fait souvent partie des contraintes imposées par le concepteur. La synthèse d'architecture s'assure du respect de cette contrainte. Toutefois, il est possible que le système avec lequel est interfacé le composant généré introduise des temps d'attente imprévisibles lors de certains échange de données. C'est par exemple le cas lorsqu'un port d'entrée est relié à un canal de type FIFO. Le composant généré peut être prévu pour bloquer son exécution tant qu'une donnée n'est pas présente. Or, si le système rencontre un problème imprévu, il se peut qu'une donnée arrive avec du retard. Cela n'a pas de conséquence sur le bon fonctionnement du composant généré, mais sa latence s'en trouve altérée. Il se peut aussi que le composant rentre dans une boucle infinie, l'empêchant d'effectuer sa tâche.

Afin de pouvoir identifier en simulation ce type de problème, des arcs son introduits dans le modèles afin de représenter les délais maximums entre la réception des données et la production des sortie de l'application. Cette contrainte implantée par la suite sous forme d'assertion identifie tel un *watchdog* toute latence excessive. Cette approche est en partie tiré des travaux de John Curreri [51], qui permet la synthèse d'un watchdog synthétisable. Toutefois, dans son approche, c'est le concepteur qui doit spécifier manuellement le code comportemental (en C++) permettant de calculer le nombre de cycles d'horloges écoulés entre deux lignes de code.

4.2 Identification des branches d'assertion

Afin de pouvoir séparer les opérations présentes dans le modèle (qui doivent être implantées matériellement) de toutes celles liées aux assertions (qui ne doivent pas être implantées), il est nécessaire d'identifier les opérations de l'algorithme et celles liées aux branches d'assertion. Cette identification peut s'avérer complexe lorsque le modèle DAG a été optimisé en amont, par exemple à l'aide d'une factorisation des expressions communes. Dans ce cas, les branches d'assertion peuvent partager des noeuds de type opération avec les branches purement calculatoires.

La méthode proposée pour l'identification des branches d'assertion est divisée en deux étapes : V_{assert} est d'abord identifié globalement, sans notion de branches individuelles, puis divisé en plusieurs branches d'assertions. En effet, l'identification de l'ensemble V_{assert} (et $V_{assertPure}$, car sa définition est liée à V_{assert}) dans le graphe permet d'effectuer en second lieu la séparation des branches. Il est à noter que $V_{assertPure}$ peut être directement obtenu à partir de V_{assert} par la formule donnée en équation 3.2. Cette équation définit les éléments de $V_{assertPure}$ comme étant tous les noeuds dont les enfants font partie d'une branche d'assertion.

$$V_{assertPure} = \{n \in V_{assert} : C(n) \subset V_{assert}\} \tag{3.2}$$

Afin de détecter l'ensemble V_{assert} dans le graphe de flot de données initial, l'algorithme proposé analyse d'abord le graphe pour définir V_{algo}, l'ensemble des noeuds participant à l'application et devant être implantés matériellement. Il peut alors en déduire V_{assert} par l'équation $V_{assert} = V \cap V_{algo}$. L'algorithme de détection de V_{assert}, présenté dans le listing 3.5, parcourt le graphe à partir des noeuds de V_{io} en allant d'un noeud à ses parents, récursivement. Tous les noeuds rencontrés sont ajoutés à V_{algo}.

Listing 3.5– Algorithme de détection de V_{assert}

```
1  def getAllParents(v):
2      branch = {v}
3
4      if P(v) = ∅:
5          return branch
6
7      for p in P(v):
8          branch = branch ∪ getAllParents(p)
9
10     return branch
11
12
13 V_algo = ∅
14
15 for v in V_io:
16     V_algo = V_algo ∪ getAllParents(v)
17
18 V_assert = V ∩ V_algo
```

Enfin, V_{assert} est scindé en différentes branches d'assertions, notées $Branch_i$, et à chacune est assigné un identifiant unique. Pour ce faire, un deuxième algorithme – présenté en figure 3.6 – part d'un noeud de $V_{assertOut}$ et remonte les noeuds du graphe tant qu'ils appartiennent à V_{assert}.

Les différentes branches d'assertion sont alors identifiées, comme l'illustre la figure 33. Sur cette figure, les chiffres indiqués en haut à gauche des noeuds correspondent à ou aux identifiants des branches d'assertion dont ils font partie. On peut observer que certains noeuds de $V_{assertPure}$ (noeuds blancs) appartiennent à deux branches simultanément. La phase suivante traite en particulier ces noeuds.

Listing 3.6– Algorithme d'identification des branches d'assertion $Branch_i$

```
1  def getAssertionBranch(v):
2      assert v in V_assert
3      branch = {v}
4
5      if P(v) = ∅:
6          return branch
7
8      for p in P(v):
9          if p in V_assert:
10             branch = branch ∪ getBranch(p)
11
12     return branch
```

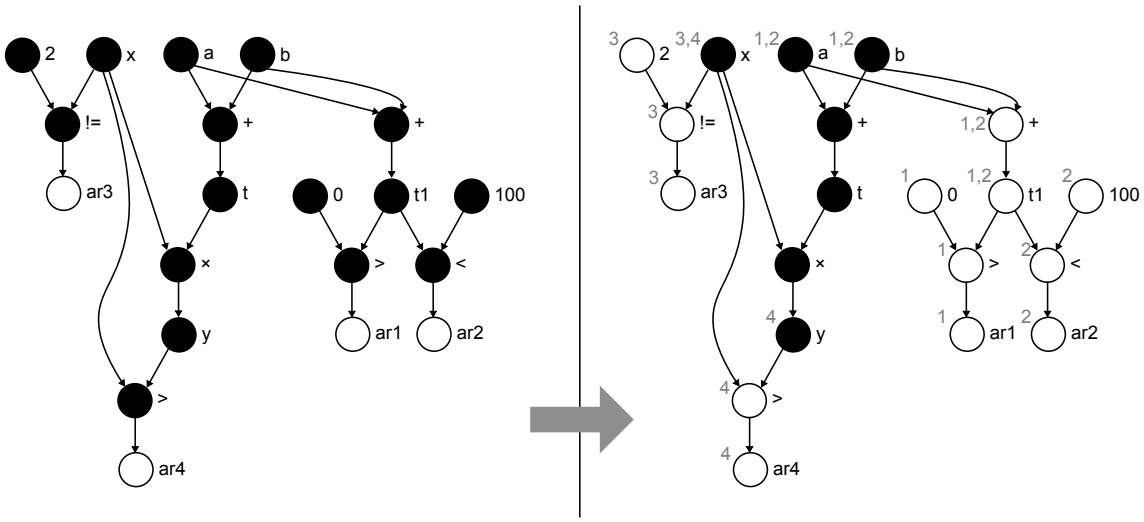

FIGURE 33 – Phase d'identification des branches d'assertion

4.3 Optimisation des branches d'assertion

Les branches d'assertion peuvent partager des noeuds avec d'autres branches d'assertion ou d'autres branches calculatoires. Cela est principalement dû soit aux optimisations du compilateur, soit à la façon dont a été écrite la description comportementale. Ce type d'optimisation peut aussi avoir lieu sur une portion d'un calcul. Par exemple, l'exemple pédagogique montre depuis le début deux conditions d'assertions, $a+b>0$ et $a+b<100$, représentées sous forme de graphe. Le résultat intermédiaire correspondant à l'opération $a+b$ a été optimisé dans le graphe pour n'être calculé qu'une seule et unique fois. De ce fait, les deux opérations, pourtant indépendantes, se recouvrent en partie, comme cela est illustré sur la figure 33.

Ce recouvrement réduit considérablement le nombre de noeuds dans le graphe de flot de données. Néanmoins, tous les traitements futurs effectués par notre méthode sur les

branches d'assertion en sont rendus plus complexes, car ces étapes traitent les branches individuellement les unes des autres. De plus, on perd en flexibilité : il devient impossible par exemple de désactiver telle ou telle assertion lors de la simulation de l'architecture.

L'algorithme présenté dans la figure 3.7 vise à supprimer tout recouvrement entre les branches d'assertion. Pour cela, il s'exécute avec comme paramètres un noeud puits d'une branche d'assertion (donc dans $V_{assertOut}$) et l'identifiant de la branche, et parcourt en remontant récursivement tous les noeuds de la branche, en vérifiant s'ils font partie d'une ou plusieurs branches. Cela se fait en testant l'appartenance d'un noeud aux ensembles B_i, construits dans l'étape précédente. Chaque noeud dans $V_{assertPure}$ présentant un recouvrement est alors dupliqué à l'identique, et inséré dans la seconde branche. Les arcs le reliant à ses enfants et parents sont adaptés en fonction.

Listing 3.7– Algorithme de suppression du recouvrement des branches d'assertion

```
1  def divideBranch(v, id):
2      queue = new list
3      add v to queue
4
5      while queue ≠ ∅:
6          v = queue.poll()
7
8          branches = {Branch_j}, forall j, v ∈ Branch_j
9
10         if v ∈ V_assertPure and branches ≠ ∅:
11             v' = clone v without links
12             add v' to graph
13             add v' to branches-Branch_id
14             remove v from branches-Branch_id
15
16             for p in P(v):
17                 create link (p, v')
18
19             for c in C(v):
20                 create link (v', c)
21                 delete link (v, c)
22
23         if v ∈ V_assertPure:
24             queue = queue ∪ P(v)
```

Un noeud v fait partie de plusieurs branches d'assertion s'il vérifie l'équation $v \in V_{assertPure} \wedge v \in \bigcap_{i=1}^{N} B_i$. En effet, l'intersection $\bigcap_{i=1}^{N} B_i$ doit se limiter à $V_{assert} \setminus V_{assertPure}$ – c'est à dire aux variables d'entrée des assertions – si aucun recouvrement n'a lieu. De cette façon, on peut écrire que deux branches d'assertion i et j se recouvrent au moins en partie si elles vérifie l'équation $Branch_i \cap Branch_j \cap V_{assertPure} \neq \emptyset$.

La figure 34 reprend l'exemple précédent et illustre le résultat après exécution de l'algorithme d'optimisation. Tout recouvrement dans $V_{assertPure}$ a été supprimé : les branches

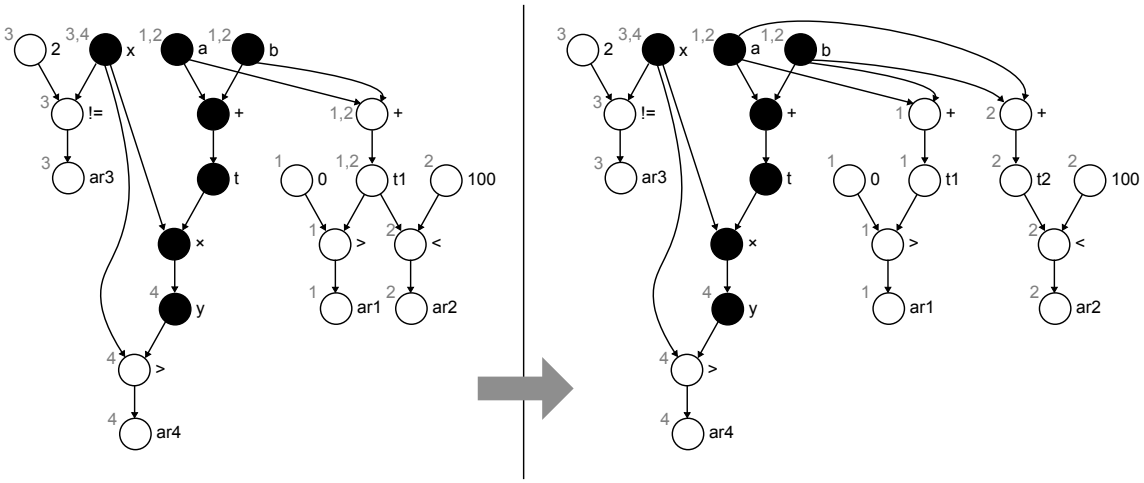

FIGURE 34 – Phase de suppression du recouvrement des branches d'assertion

sont différenciées.

4.4 Suppression des branches d'assertion

Une fois les branches différenciées, elles peuvent être supprimées du graphe principal. Un nouveau graphe de flot de données est construit, $G_A = (V_A, A_A)$. Il regroupe toutes les branches d'assertion $Branch_i$.

Pour chaque branche i, les noeuds appartenant à $Branch_i \cap V_{assertPure}$ sont directement supprimés de V et insérés dans V_A, tandis que les noeuds de $Branch_i \cap (V_{assert} \setminus V_{assertPure})$ c'est à dire les noeuds d'entrée des assertions) sont dupliqués à l'identique pour chaque branche d'assertion, et laissés dans V. La relation entre les noeuds de V et leurs différents clones dans V_A est conservée dans une *table d'association*.

La figure 35 illustre ce traitement. Les noeuds en gris sont les noeuds ayant été clonés, n'appartenant pas à $V_{assertPure}$. Leur relation avec les noeuds sources d'origine est enregistrée dans la table d'association. Ces relations sont représentées par les flèches doubles sur la figure. Ainsi, le graphe de flot de données de départ est laissé tel qu'il aurait été si aucune assertion n'avait été spécifiée dans le code.

A partir d'ici, le processus de synthèse de l'outil de HLS reprend là ou il s'était arrêté. La phase d'assignation lie une ressource matérielle (registre ou opérateur) aux différents noeuds, et la phase d'ordonnancement leur assigne une date. Cette date correspond pour les noeuds opérateurs au moment ou l'opération aura lieu, et pour les noeuds variables au moment où la donnée représentée se trouve physiquement dans la ressource matérielle associée.

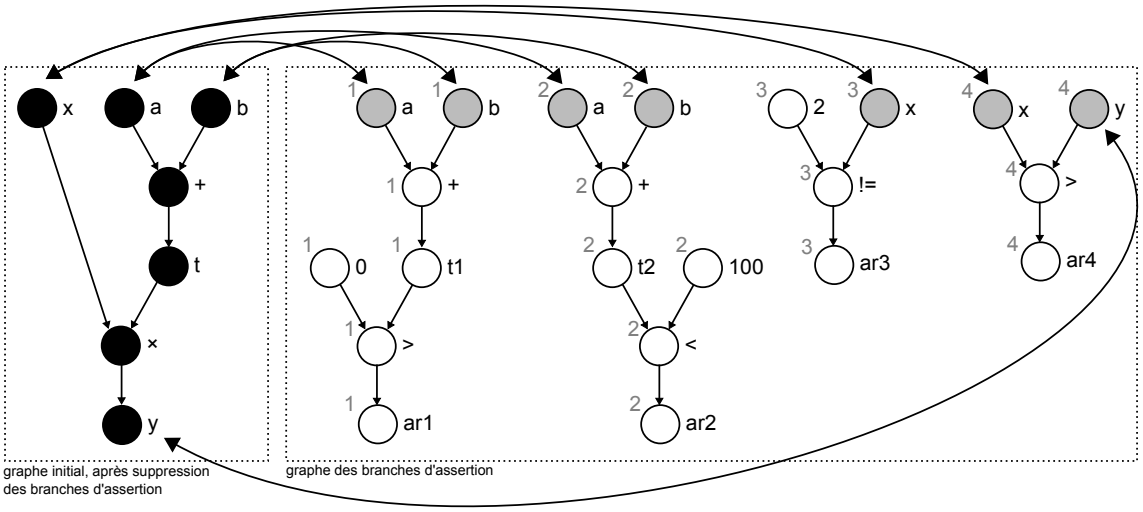

FIGURE 35 – Résultat de la phase de suppression des branches d'assertion. Des liens sont conservés en mémoire entre les noeuds sources et leurs clones dans le nouveau graphe.

4.5 Synchronisation des modèles

Cette étape consiste à retrouver, pour chaque branche d'assertion, les informations des dates et des ressources matérielles assignées durant les phases d'assignation et d'ordonnancement du processus de HLS. Pour chaque branche d'assertion, les informations nécessaires sont celles des noeuds du graphe général qui ont un clone dans V_A. Autrement dit, ce sont les noeuds représentant une variable d'entrée de l'assertion. En effet, ce sont eux qui déterminent le moment où l'assertion peut être vérifiée.

Dans l'architecture matérielle qui sera générée par l'outil de HLS, les variables définies dans le code de haut-niveau (en SystemC par exemple) sont liées à des registres spécifiques. Afin d'économiser les ressources requises par l'application, tous les outils de synthèse réutilisent les espaces mémoires dès lors que les variables qui y sont stockées ne sont plus utilisées par le système. Ainsi, une variable d'entrée d'une assertion n'est présente qu'à un moment précis dans l'architecture, et dans une zone mémoire spécifique. Ce sont ces deux informations dont a besoin une assertion matérielle pour être vérifiée :

– les noms des registres contenant les variables utilisées dans les assertions,

– les dates de disponibilité (en terme de cycles d'horloge) de ces variables au sein des registres correspondant.

Obtenir ces informations se fait par l'utilisation de la table de correspondance précédemment mise en place lors de la suppression des branches d'assertion. La figure 36 illustre l'étape de synchronisation. On peut y voir que le graphe initial a été modifié par la synthèse HLS : l'ordonnancement a placé le noeud b au cycle 2 car il utilise la même ressource de mémorisation (*input2*) que le noeud a. De plus, tous les noeuds, variables comme

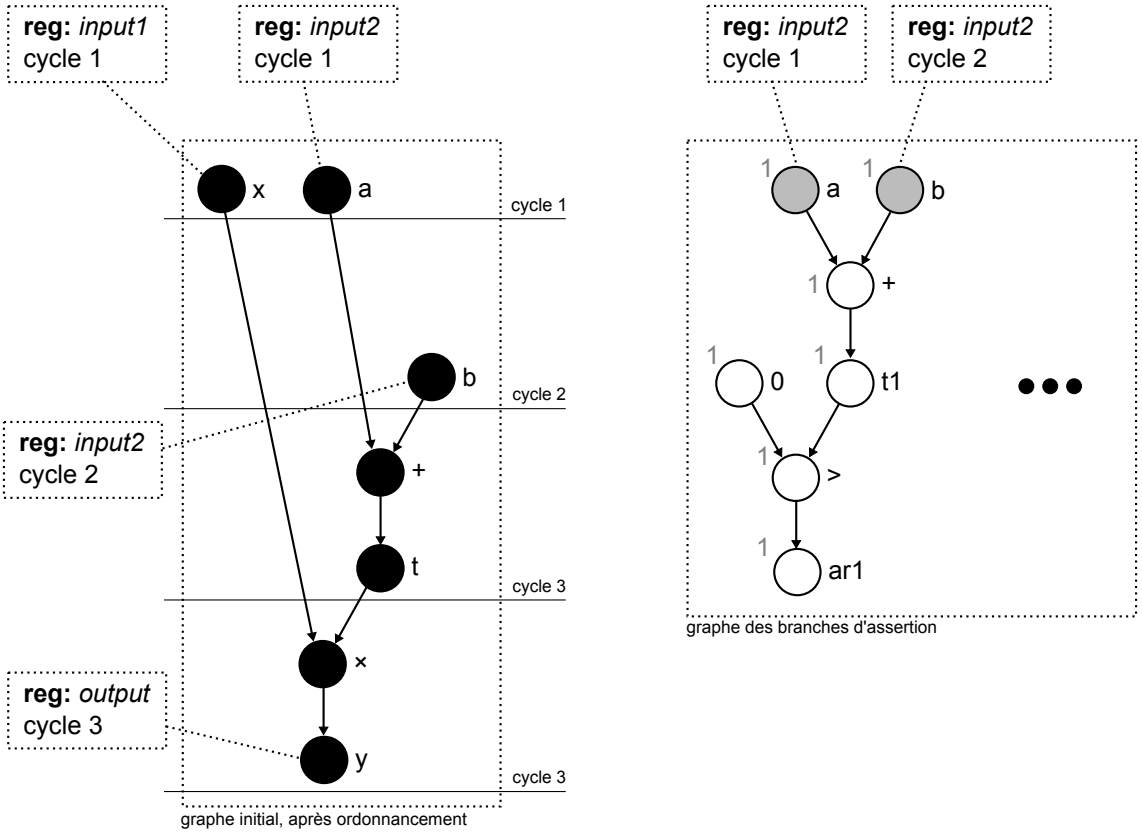

FIGURE 36 – Résultat de la phase de synchronisation du graphe d'assertion. Seules les annotations des noeuds de V_{io} sont montrées pour des raisons de lisibilités, mais tous les noeuds du graphe initial sont en fait annotés.

opérations, se sont vus attribués une ressource matérielle et une date de disponibilité/-d'exécution (bien que seule une partie de ces informations, celles des noeuds variables de V_{io}, soit présente sur la figure pour des raisons de lisibilité).

Une fois ces informations acquises, le graphe d'assertions G_A contient suffisamment d'informations pour que soient générées les assertions matérielles. Toutefois, une étape supplémentaire permet d'en améliorer la fonctionnalité.

4.6 Ordonnancement des assertions

Les branches d'assertions doivent être ordonnancées avant d'être implémentées sous forme de description matérielle. Cela permet de respecter les désirs du concepteur, tels qu'ils sont spécifiés dans la description de haut-niveau. Par exemple, avant de vérifier que le résultat d'une division est compris dans une certaines plage de variation, il est utile de vérifier que le diviseur n'est pas égal à zéro, auquel cas la division n'a pas de sens. L'ordre de vérification des assertions est important pour la compréhension des rapports d'erreurs donnés par le simulateur.

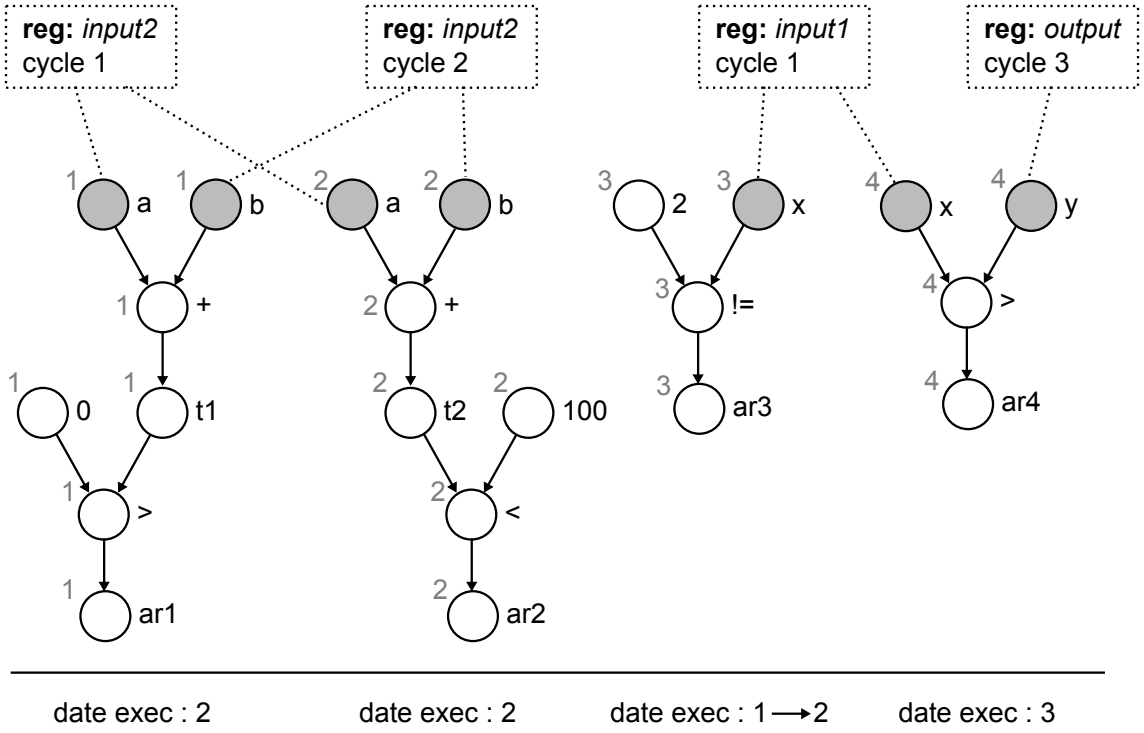

FIGURE 37 – Résultat de la phase d'ordonnancement du graphe d'assertion. L'assertion 3 sera vérifiée au cycle 4 au lieu du 2, pour respecter l'ordre spécifié par le concepteur.

Ainsi, des dates d'exécution des assertions sont ajoutées au graphe, afin de renseigner l'étape suivante de transformation des branches en assertions matérielles. La date d'exécution par défaut d'une branche d'assertion correspond à celle à laquelle toutes les variables d'entrée ont pu être lues. C'est à dire, au maximum des dates de disponibilité de ces variables d'entrée. Ces dates par défaut sont enregistrées dans une table. Une fois la table entièrement construite, elle est triée en suivant l'ordre d'exécution requis pour les assertions, donné par le modèle comportemental. Le processus est le suivant : la table est parcourue linéairement, et si une date d'exécution est inférieure à la précédente, elle est changée pour l'égaler. Ce traitement, illustré en figure 37, assure que l'ordre souhaité sera respecté.

4.7 Génération du code RTL

Le processus de génération des moniteurs pour la simulation à partir du graphes d'assertion cible deux langages de description : le PSL d'une part, et le VHDL d'autre part. L'utilisation du langage PSL pour décrire des assertions et vérifier en simulation le comportement d'une architecture matérielle est naturelle. Ce langage a été conçu pour cela. Toutefois, au moment où ces travaux ont été menés, tous les simulateurs ne supportaient pas son utilisation. C'était par exemple le cas de ModelSim SE [139] (la version professionnelle ModelSim DE proposait ce support) et GHDL [131]. Ainsi, pour ajouter plus de

flexibilité aux moniteurs générés, il a été décidé de proposer également une version VHDL de ces moniteurs.

La génération d'une branche d'assertion en un moniteur en langage PSL ou VHDL nécessite de transformer l'équation définie dans le graphe sous forme d'enchaînement de noeuds en équation écrite. La détermination de cette équation se fait en partant du résultat de l'assertion, le dernier noeud du graphe, et en remontant la branche. Il peut aussi être envisagé d'annoter les noeuds de $V_{assertOut}$ avec l'équation complète au moment de la compilation.

Comme précisé tout au long de ce chapitre, les variables faisant partie d'un calcul d'assertion ne sont disponibles qu'à un moment précis, et au sein d'une ressource matérielle précise. Ces informations de ressources et de dates de disponibilité ont été recueillies lors de la phase de liaison des ressources aux branches d'assertion. La date à laquelle doit se faire la vérification de l'assertion est elle dictée par l'étape de tri des graphes d'assertion. Ainsi, il ne reste qu'à générer l'assertion PSL correspondante.

Moniteurs PSL

L'assertion PSL générée est une séquence temporelle constituée de deux parties : l'attente jusqu'à la date de vérification, puis la vérification elle-même. Le listing 3.8 montre l'implémentation des assertions de l'exemple en langage PSL, ici avec une notation en VHDL des signaux (le PSL ne proposant pas de syntaxe universelle).

Si une assertion doit être exécutée lors d'un cycle d'horloge (ou un état de la machine d'état de l'architecture ici) plus tardif que la date de disponibilité d'une variable, alors la valeur de cette variable est retrouvée par l'utilisation de la fonction intégrée au langage `prev(arg [,N])`. Cette fonction renvoie la valeur du signal qui lui est passé en argument tel qu'il était au cycle d'horloge précédent. Le paramètre optionnel N spécifie le nombre de coups d'horloges à remonter. La valeur de cet argument, pour chaque variable de l'assertion, est la différent entre la date de vérification de l'assertion et la date de disponibilité de la variable considérée.

La suppression de plusieurs assertions en fonction des besoins du concepteur est simple, car chaque ligne décrit une assertion individuelle. Supprimer une ligne revient à supprimer une assertion, sans qu'il n'y ait d'impact sur les autres.

Listing 3.8– Implémentation des trois branches d'assertion en langage PSL

```
1  assertion1 : assert always (state = s2 ->
2      SIGNED(input2) + SIGNED(prev(input2, 1)) > 0);
3
4  assertion2 : assert always (state = s2 ->
5      SIGNED(input2) + SIGNED(prev(input2, 1)) < 100);
6
7  assertion3 : assert always (state = s2 ->
8      SIGNED(prev(input1, 1)) /= 2);
9
10 assertion4 : assert always (state = s3 ->
11     SIGNED(output) > SIGNED(prev(input1, 2)));
```

Moniteurs VHDL

Une assertion VHDL ne peut que représenter une condition purement booléenne. Le langage ne possède aucune sémantique propre (hors inclusion du PSL en commentaires) liée à l'expression d'une condition temporelle dans une assertion. Or, les conditions que l'on souhaite vérifier sont temporelles. Il est donc nécessaire d'entourer les appels d'assertions dans des *process* VHDL dédiés, qui permettront le séquencement temporel des actions.

Un moniteur VHDL fonctionne un peu différent d'un moniteur PSL. Il est lui aussi basé sur deux phases, mais différentes : la récupération des variables, puis la vérification de l'assertion. La première phase consiste à sauvegarder l'état d'un signal au moment de sa date de disponibilité, afin de pouvoir l'exploiter plus tard lors de la vérification de la condition. Pour ce faire, le process entourant l'appel de l'assertion est muni d'autant de variables internes qu'il y a de signaux à enregistrer. Ensuite, les états de la machine d'état générale sont observés et les variables du process enregistrent les signaux correspondants aux moments opportuns. Finalement, l'assertion est vérifiée par un appel au mot clé *ASSERT* du langage (cf. listing 3.9).

Les variables ajoutées au process VHDL ne conduisent à aucun ajout de ressources non souhaité dans la description matérielle, car ces variables ne sont utilisées qu'au sein du process, et uniquement dans les appels aux assertions. De ce fait, comme nous avons pu le vérifier, tous les outils de synthèse logique existant suppriment ces ressources lors de la compilation. En effet, les fonctions d'assertions ne sont pas synthétisables, et sont donc systématiquement supprimésss. Ne restent alors que des variables isolées n'intervenant nulle part. L'optimisation de suppression de code mort [91] empêche ainsi ces variables d'être gardées pour la phase d'assignation des ressources physiques.

L'utilisation d'un seul et unique process pour assurer la temporalité aux assertions pose toutefois un problème : la suppression manuelle d'une ou de plusieurs assertions peut

Listing 3.9– Implémentation des quatre branches d'assertion en langage VHDL

```
1  PROCESS (state)
2      VARIABLE var_x : signed(x'length-1 downto 0);
3      VARIABLE var_a : signed(a'length-1 downto 0);
4      VARIABLE var_b : signed(b'length-1 downto 0);
5      VARIABLE var_y : signed(y'length-1 downto 0);
6  BEGIN
7      CASE state IS
8          WHEN s1 =>
9              var_x := SIGNED(input1);
10             var_a := SIGNED(input2);
11         WHEN s2 =>
12             var_b := SIGNED(input2);
13             ASSERT (var_a + var_b > 0)     -- assertion 1
14                 REPORT "assert(a + b > 0)"
15                 SEVERITY failure;
16             ASSERT (var_a + var_b < 100)  -- assertion 2
17                 REPORT "assert(a + b < 100)"
18                 SEVERITY failure;
19             ASSERT (var_x /= 2)           -- assertion 3
20                 REPORT "assert(x /= 0)"
21                 SEVERITY failure;
22         WHEN s3 =>
23             var_y := SIGNED(output);
24             ASSERT (var_y > var_x)        -- assertion 4
25                 REPORT "assert(y > x)"
26                 SEVERITY failure;
27     END CASE;
28 END PROCESS;
```

s'avérer assez fastidieuse. Pourtant, c'est une possibilité qui doit être gardée, dans le cas ou un concepteur souhaiterait spécialiser les vérifications à faire, ou si la vérification d'une plage de variation n'a plus lieu d'être suite à une modification du cahier des charges par exemple. Ainsi, nous proposons aussi une version des moniteurs utilisant un process dédié par assertion. Ce mode de fonctionnement est illustré dans le listing 3.10, où seules les branches d'assertion 1 et 4 sont décrites. Ce mode augmente fortement la quantité de lignes de code générées, mais offre une plus grande flexibilité.

5 Conclusion

Ce chapitre a présenté une méthode de transformation des assertions d'une description comportementale en moniteurs dans l'architecture matérielle générée par synthèse HLS. Ces moniteurs sont dédiés à la simulation, ils ne sont pas synthétisables. L'objectif principal a été de conserver le formalisme des assertions comportementales au sein de la description

Listing 3.10– Implémentation de deux des quatre d'assertion en langage VHDL (avec un process par assertion)

```
1  assertion1 : PROCESS (state)
2      VARIABLE var_a : signed(a'length-1 downto 0);
3      VARIABLE var_b : signed(b'length-1 downto 0);
4  BEGIN
5      CASE state IS
6          WHEN s1 =>
7              var_a := SIGNED(input2);
8          WHEN s2 =>
9              var_b := SIGNED(input2);
10             ASSERT (var_a + var_b > 0)  -- assertion 1
11                 REPORT "assert(a + b > 0)"
12                 SEVERITY failure;
13     END CASE;
14 END PROCESS;
15
16 assertion4 : PROCESS (state)
17     VARIABLE var_x : signed(x'length-1 downto 0);
18     VARIABLE var_y : signed(y'length-1 downto 0);
19 BEGIN
20     CASE state IS
21         WHEN s1 =>
22             var_x := SIGNED(REG1);
23         WHEN s3 =>
24             var_y := SIGNED(OUT);
25             ASSERT (var_y > var_x)      -- assertion 3
26                 REPORT "assert(y > x)"
27                 SEVERITY failure;
28     END CASE;
29 END PROCESS;
```

matérielle. Les assertions peuvent aider l'intégration d ?une IP dans un système, et vérifier que celle-ci est utilisée correctement. En définissant des contrats d'utilisation entre les IP générées et le système, les assertions permettent d'assurer l'intégrateur qu'un composant virtuel reçoit des données valides, et par conséquent qu'il fonctionnera correctement. Toutefois, intégrer ces moniteurs à la main peut être extrêmement fastidieux, du fait de toutes les optimisations apportées par les outils de HLS lors de la synthèse, et de l'opacité d'une description générée automatiquement par un outil.

Ces travaux permettent aussi la découverte automatique de paramètres qu'il peut être utile de vérifier lors de la simulation d'un composant ou d'un système. La validité des entrées, le bon fonctionnement des opérateurs internes, et le respect de la latence maximale autorisée sont autant de possibilités d'erreurs qui peuvent échapper au contrôle d'un concepteur lors de la vérification d'un système. La méthode proposée autorise la détection de ces paramètres, et les transforment en moniteurs dans la description matérielle, au même titre que les assertions spécifiées par le concepteur.

Chapitre 4

Gestionnaire d'erreurs

Sommaire

Dans ce chapitre, nous allons détailler la seconde contribution réalisée dans le cadre de l'amélioration de la vérification des systèmes synthétisés par HLS. Cette contribution cible la vérification des architectures en fonctionnement réel à l'aide d'un gestionnaire d'erreurs embarqué au sein du circuit.

Le rôle de ce gestionnaire est d'archiver les erreurs détectées par les moniteurs matériels intégrés au système et synthétisés à partir des assertions de la description comportementale, ainsi que le contexte d'exécution courant. La trace du contexte d'exécution se compose d'un ensemble de données spécifiées par le concepteur. Cet ensemble peut varier en fonction des assertions. Grâce à ce mécanisme, le concepteur peut à tout moment récupérer la liste des erreurs ayant eu lieu, et leur analyse se trouve facilitée par les informations additionnelles mémorisées.

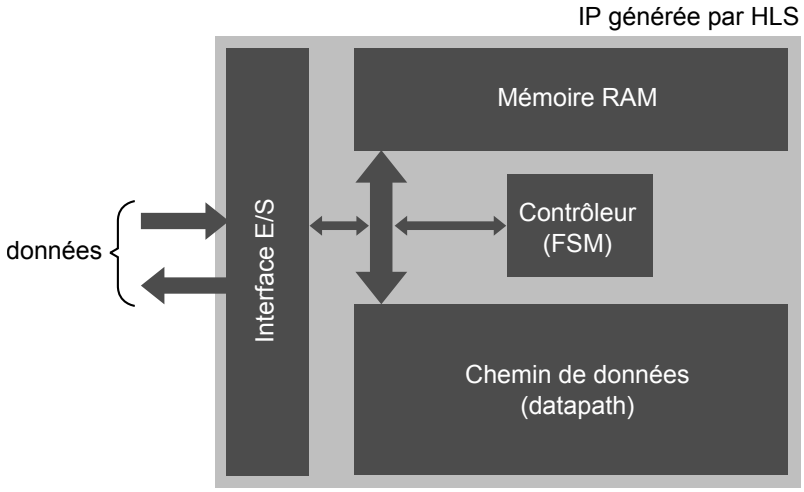

FIGURE 38 – Implémentation d'une architecture standard générée par HLS

1 Introduction

La conception des systèmes a été fortement accélérée par l'apparition de la synthèse de haut-niveau [109, 57, 49]. Les composants générés par HLS sont souvent implémentés comme présenté sur la figure 38. Ils se composent d'un chemin de données associant plusieurs opérateurs logiques et arithmétiques, d'une mémoire de sauvegarde des données, d'une interface de communication et d'un contrôleur gérant l'ordre des opérations à effectuer en fonction des cycles d'horloge.

Toutefois, une mauvaise intégration ou une mauvaise utilisation de ces composants dans un système peut mener à des erreurs parfois très difficiles à détecter. La simulation des systèmes est limitée par la puissance actuelle des processeurs séquentiels, amenant les équipes de vérification à privilégier les tests en fonctionnement réel. La grande facilité de vérification offerte par les moteurs de simulation est alors perdue. Le gain de vitesse d'exécution est en partie contrebalancé par la complexité liée à la récupération des informations de vérification sur un circuit matériel. De plus, dans le cas des architectures générées par HLS, les registres internes ne correspondent pas directement aux variables de l'application implémentée (un même registre peut être réutilisé au cours du temps pour plusieurs variables). Cela complexifie d'autant plus l'analyse des erreurs ainsi que leur identification. Par exemple, si on souhaite vérifier $a \neq 0$, cela peut se traduire par $reg246 \neq 0$, observable seulement pendant 10 cycles d'horloge.

Pour palier aux contraintes de la vérification sur circuit, plusieurs travaux de la littérature proposent la transformation des assertions (de niveau RTL ou ESL) en moniteurs matériels embarqués dans un circuit. Ces moniteurs ont pour rôle d'analyser le système jusqu'à ce qu'une condition spécifique apparaisse, signe qu'une erreur a été détectée. De nombreuses contributions [1, 22, 23, 121, 161] utilisent les assertions temporelles spécifiées au niveau

RTL afin de synthétiser les moniteurs matériels correspondant. D'autres travaux [68, 50, 51] se concentrent sur les outils de synthèse HLS en transformant directement les assertions booléennes spécifiées au niveau comportemental en moniteurs dans l'architecture générée.

Cependant, l'ensemble de ces travaux ne traitent que de la détection des erreurs à l'aide de moniteurs matériels, qu'ils soient générés à partir du niveau RTL ou ESL. Cette détection peut être complexe à interpréter et à analyser sans connaissance du *contexte d'exécution*. Pour illustrer ce point, considérons le listing 4.1. Une fois cette description comportementale implémentée matériellement par synthèse HLS, un moniteur matériel vérifiant la condition de l'assertion peut notifier le concepteur de l'identification d'une erreur. Toutefois, la seule information donnée au concepteur par ce biais est «la variable t vaut zéro», sans pour autant en donner la cause. Or, cette cause n'est pas immédiate : soit a est égal à b, soit c vaut zéro. Remonter à la source de l'erreur demande de multiples tests et de l'instrumentation. La connaissance du contexte d'exécution, c'est à dire les valeurs des variables a, b et c, aurait permis de comprendre immédiatement d'où est issue l'erreur. En fait, dans cet exemple, la seule connaissance de la valeur de c permet de supprimer l'ambiguïté.

Listing 4.1– Une assertion identifie une erreur, mais n'en donne pas la cause.

```
1 int foo(int a, int b, int c, int d) {
2     int t = (a - b) * c;
3     assert(t != 0);
4
5     ...
6 }
```

Dans le domaine de la simulation, les simulateurs gardent en mémoire l'intégralité des chronogrammes associés aux signaux du système tout au long de la simulation. Ils permettent ainsi d'observer l'état de tous ces signaux tels qu'ils étaient au moment d'une erreur. La nature de l'erreur combinée à la valeur des signaux jouent souvent un rôle essentiel dans la résolution des problèmes découverts.

John Curreri [51] propose la génération HLS de moniteurs matériels synthétisables à partir des assertions booléennes d'une description comportementale. L'implémentation de la partie calculatoire d'un moniteur est intégralement laissée à la charge des outils de synthèse HLS. Il propose de notifier un processeur généraliste de chaque détection d'erreur, en lui fournissant le numéro d'identification du moniteur concerné. Le processeur peut alors enregistrer cette information et la valeur du temps courant. Cette approche cible une architecture conjointe, guidée par l'utilisation de l'outil Impulse-C [134] utilisé dans ses travaux. Ce mécanisme est illustré dans la figure 39. Les moniteurs matériels sont implémentés directement dans le chemin de données, et en partagent ainsi les ressources.

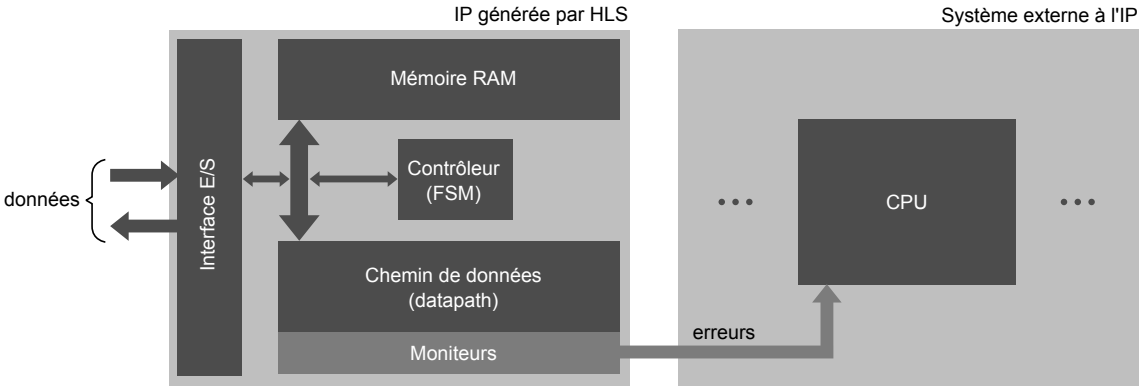

FIGURE 39 – Implémentation et utilisation des moniteurs dans une architecture générée par HLS, telles que proposées par John Curreri [51]

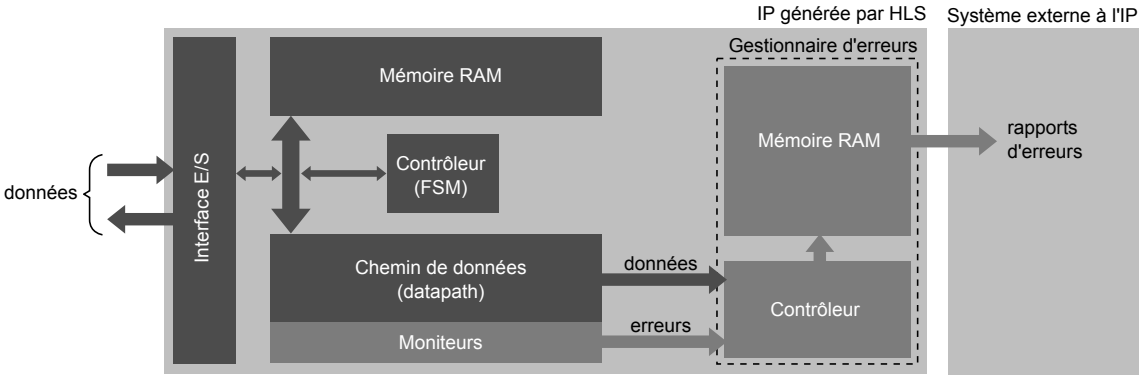

FIGURE 40 – Implantation d'un gestionnaire d'erreurs matériel dans une architecture générée par HLS

Toutefois, les informations liées à l'erreur détectée sont minimes : seul est connu l'identifiant de l'assertion qui a servi à détecter l'erreur. Cette approche présente donc le problème présenté quelques paragraphes plus tôt : remonter à la source de l'erreur n'est pas toujours possible par la seule connaissance qu'une erreur a eu lieu. Connaître les valeurs de certaines variables permet dans la plupart des cas de lever les ambiguïtés.

Pour faciliter la compréhension et la résolution des erreurs détectées par les moniteurs matériels, nous proposons une méthode générique permettant la synthèse automatique d'un gestionnaire d'erreurs matériel et son implantation dans les architectures générées par HLS. Le rôle de ce gestionnaire est de récupérer le contexte d'exécution spécifié par le concepteur pour chaque assertion, et d'archiver ce contexte et l'identifiant de l'assertion, constituant ensemble un *rapport d'erreur*. Pour des raisons de généricité, la conception de cette unité est contrainte par les choix effectués par l'outil de HLS (date d'exécution des calculs, durée de vie des variables au sein de l'architecture). En fonction de ces contraintes, l'objectif est de générer l'architecture la moins onéreuse en surface de silicium. La figure 40 illustre l'implantation de ce gestionnaire au sein d'une architecture.

2 Synthèse d'un gestionnaire d'erreurs

Les variables dont les valeurs sont à mémoriser lorsqu'une assertion n'est pas vérifiée sont spécifiées à l'avance pour chaque assertion. D'un point de vue technique, cela peut se faire par exemple par l'utilisation d'assertions annotées, comme cela est illustré dans le listing 4.2. Cette technique d'annotation est courante. Elle est par exemple employée au niveau RTL pour spécifier des assertions PSL dans un code VHDL.

Listing 4.2– Une assertion identifie une erreur, mais n'en donne pas la cause.

```
1 int foo(int a, int b, int c, int d) {
2     int t = (a - b) * c;
3     assert(t != 0); // pragma report a, b, c
4
5     ...
6 }
```

La mémoire conservant les rapports d'erreurs est conçue pour être accessible par l'environnement extérieur de l'IP. Ainsi, deux cas d'utilisation du gestionnaire d'erreurs sont envisagées :

Analyse offline - La mémoire contenant les rapports d'erreurs est directement accessible par le concepteur (cf. figure 41). Ce type d'utilisation peut être particulièrement adapté à la vérification par émulation sur FPGA. Le concepteur peut récupérer à tout moment l'ensemble des derniers rapports par simple lecture de la mémoire sur un port de sortie du circuit. Leur analyse permet la correction immédiate des erreurs dans le système, qui peut alors être synthétisé et implanté à nouveau pour tester la pertinence des corrections. Un concepteur peut aussi décider de laisser les moniteurs et le gestionnaire dans le circuit final. Cela lui permet de pouvoir continuer à surveiller l'existence d'erreurs potentielles en analysant le circuit sur site, chez le client (*in-field monitoring*). La compréhension des erreurs est rendue plus facile grâce aux derniers rapports sauvegardés. De plus, si les erreurs sont liées à des cas d'utilisation très complexes à reproduire (*corner-cases*), comme par exemple une durée d'utilisation du circuit très longue (plusieurs mois), la sauvegarde d'un contexte d'erreur jusqu'à son analyse peut être très importante.

Analyse online - Les rapports d'erreurs peuvent aussi être analysés en temps réel par le système, qui peut prendre des décisions de correction ou de protection, comme illustré sur la figure 42. Le contexte d'exécution peut permettre au système d'affiner le type de correction à mettre en oeuvre.

Dans ces deux cas de figure, le gestionnaire est contraint par une faible complexité maté-

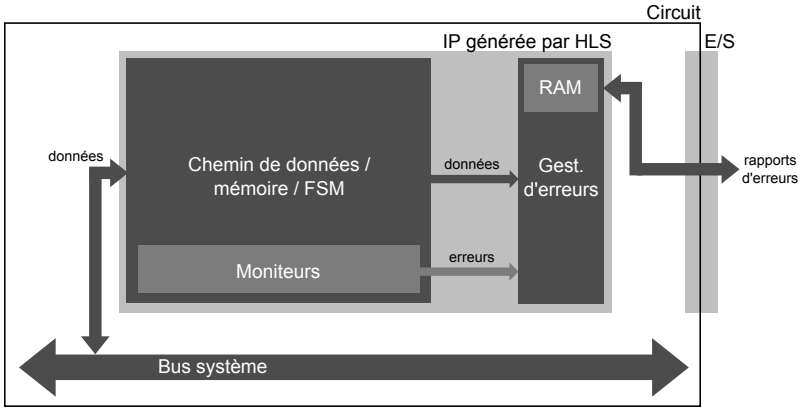

FIGURE 41 – Utilisation du gestionnaire pour analyse offline. La mémoire est directement accessible sur un port de sortie du circuit.

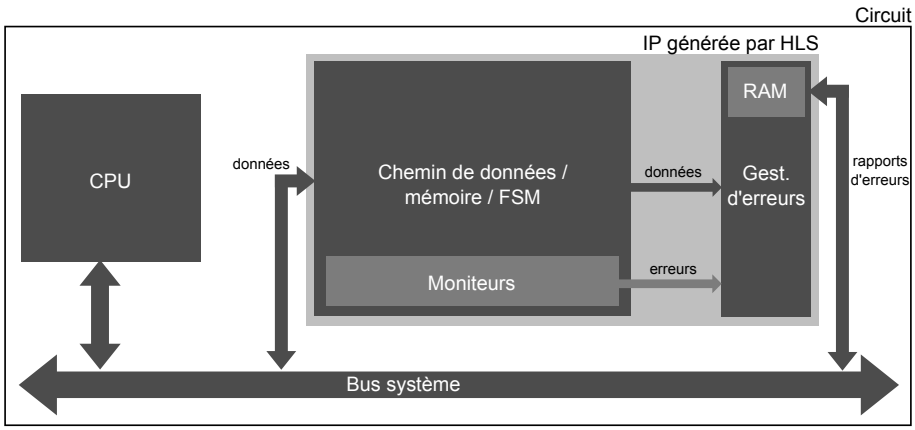

FIGURE 42 – Utilisation du gestionnaire pour analyse online. Le système analyse en temps réel les erreurs et prend des décisions de correction ou de protection automatiquement.

rielle et par le besoin de monitorer en temps réel le fonctionnement du circuit.

Dans notre étude, nous prendrons en compte le pire cas : toutes les assertions peuvent détecter une erreur durant une seule itération du système sous surveillance. La mémorisation des contextes d'exécution associés doit être effectuée en temps réel. Pour chaque assertion, le contexte d'exécution est composé des valeurs des variables applicatives spécifiées dans la description comportementale.

Le listing 4.3 présente un exemple de description comportementale avec trois assertions. Les opérations algorithmiques de la description sont masquées pour des raisons de lisibilité. Pour chaque assertion, les contextes d'exécution sont spécifiés à l'aide d'une syntaxe à base de pragmas. Cet exemple sera réutilisé tout au long de ce chapitre.

La description comportementale est ordonnancée et transformée par l'outil HLS en architecture matérielle. Des moniteurs matériels sont générés dans le chemin de données à partir des assertions spécifiées. Dans l'exemple présenté, nous considérerons que la variable

Listing 4.3– Assertions et spécification des rapports d'erreurs

```
1  int foobarbaz(int a, int b, int c, int d) {
2      ...
3      assert(a + b > 10); // pragma report a, b, c
4      ...
5      int t1, t2;
6      ...
7      assert(t1 != 0); // pragma report c, b, d
8      ...
9      assert(t2 > 0); // pragma report a, d
10     ...
11 }
```

a est codée sur 8 bits, b sur 3 bits, c sur 7 bits et d sur 2 bits.

Définition 1. *On appelle* ***variable applicative*** *toute variable de la description algorithmique de haut-niveau. Cette définition est nécessaire pour éviter l'ambiguïté avec le concept de variable RTL, utilisé dans certains langages comme le VHDL. Les variables applicatives peuvent être spécifiées par le concepteur comme faisant partie du contexte d'exécution d'une assertion. Par exemple, dans le listing 4.3, {a, b, c, d, t1, t2} sont des variables applicatives. a fait partie du contexte d'exécution de la première assertion mais aussi de la troisième. Les variables applicatives sont les principales constituantes des rapports d'erreurs.*

Définition 2. *Un* ***rapport d'erreur*** *représente l'ensemble des données à mémoriser lorsqu'une assertion n'est pas vérifiée. Un rapport est constitué en premier lieu d'une entête : l'identifiant de l'assertion qui est à l'origine de la détection d'erreur, puis des valeurs de toutes les variables applicatives telles qu'elles étaient au moment de l'erreur. Chaque rapport est enregistré dans la mémoire d'historique. Dans l'exemple du listing 4.3, la première assertion donne lieu à un rapport constitué du chiffre 1 (identifiant de l'assertion) codé sur 2 bits (trois identifiants doivent être codés), puis des valeurs des variables a, b et c.*

La taille des rapports d'erreur, en nombre de bits, varie en fonction de deux paramètres :

– le nombre de données à mémoriser,

– la taille des données à mémoriser.

En fonction de ces paramètres, la taille d'un rapport peut varier de quelques bits à plusieurs centaines. Dans ce contexte, la mémorisation des données peut nécessiter plusieurs cycles d'horloge selon la largeur de la mémoire utilisée.

La figure 43 donne la représentation schématique d'un rapport, ainsi que la façon dont les

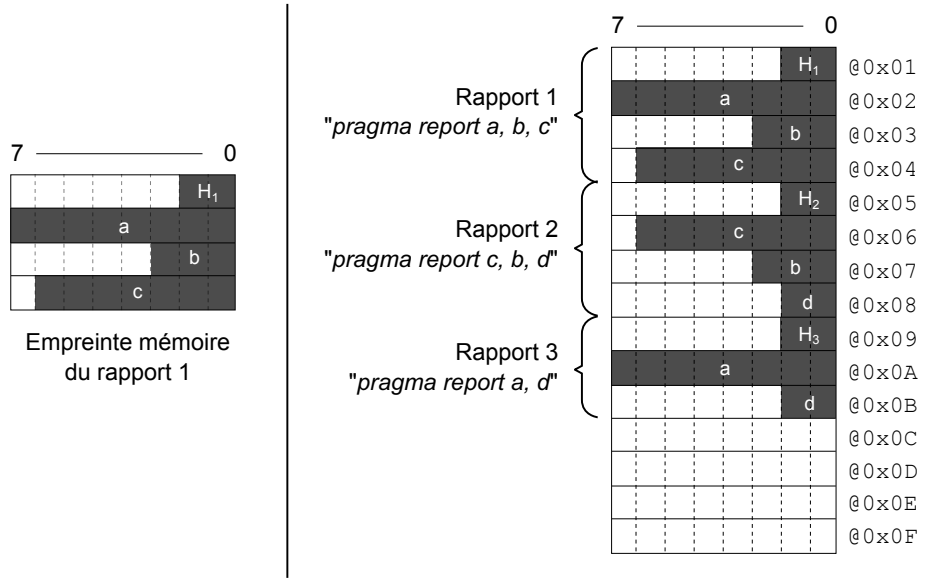

FIGURE 43 – (A gauche) L'empreinte mémoire d'un rapport est constituée de plusieurs données, organisées en cases mémoires. (A droite) Les rapports sont écrits dans la mémoire les uns à la suite des autres, lorsque les erreurs associées sont identifiées.

rapports pourraient être organisés au sein de la mémoire d'historique. Le rapport présenté est celui de la première assertion. Il contient l'identifiant de l'assertion, codé sur 2 bits dans cet exemple, ainsi que les variables a, b, et c, nécessitant respectivement 8, 3 et 7 bits. La mémoire étant d'une certaine largeur spécifiée par la méthode, les rapports sont toujours schématisés avec cette même largeur.

Cette figure présente l'arrangement le plus simple des données au sein d'un rapport : chaque donnée à mémoriser occupe une case mémoire individuelle. Cette configuration constitue la référence par rapport à laquelle les optimisations proposées par la méthode décrite dans ce chapitre sont évaluées.

La génération et la mémorisation d'un rapport d'erreur ne peut avoir lieu qu'après qu'un moniteur matériel ait détecté une erreur. Toutefois, les variables applicatives à mémoriser ne sont pas nécessairement présentes dans le système au moment où l'assertion est évaluée. En effet, parmi les données spécifiées par le concepteur, certaines peuvent avoir cesser d'exister dans le système tandis que d'autres peuvent ne pas avoir été encore calculées ou envoyées sur un port d'entrée.

La figure 44 illustre ce phénomène. Les durées de vie des variables a, b, c et d issues du listing 4.3 sont représentées au sein de l'architecture générée. La variable a est par exemple conservée pendant 9 cycles d'horloge dans le registre 1, puis son contenu devient indéterminé pendant 7 cycles, et ce même registre reçoit finalement la variable d pendant 3 cycles. Dans cet exemple, le rapport de la première assertion doit contenir les trois variables a, b et c. Au moment où le moniteur 1 identifie une erreur, ces trois variables

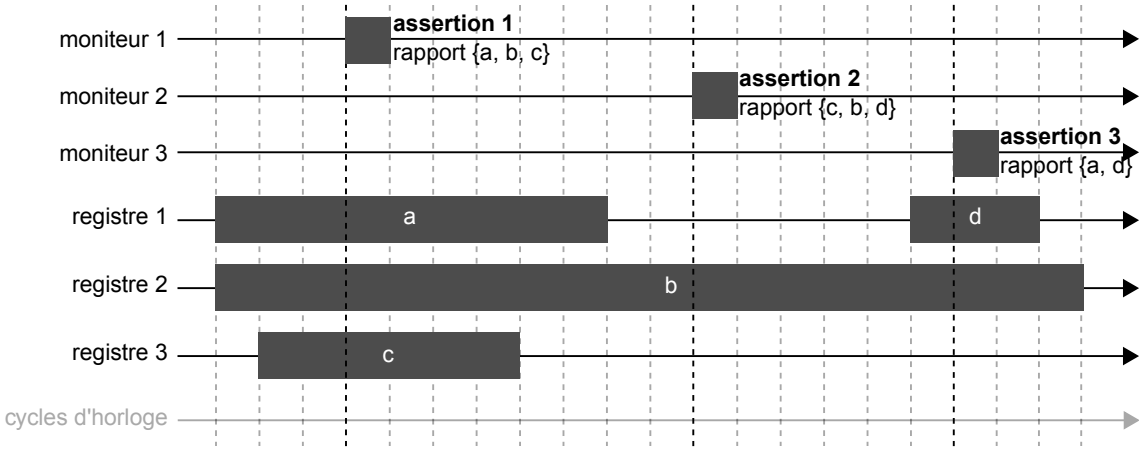

FIGURE 44 – Durées de vie des variables au sein de l'architecture générée par l'outil HLS et dates d'identification des erreurs

sont présentes dans les registres de l'architecture. Toutefois, ce n'est pas le cas pour les deux autres assertions. En effet, au moment où une erreur est détectée par le moniteur 2, ni c ni d ne sont présentes dans l'architecture. En fonction du moment ou la variable d devra être écrite dans le rapport, ce dernier ne pourra peut être pas être écrit dans la mémoire dès l'apparition de l'erreur. De plus, c n'existant plus, il est nécessaire de la mémoriser jusqu'à son utilisation dans le rapport.

En pratique, le fait qu'une variable applicative à sauvegarder ne soit pas encore présente au moment où une erreur est détectée peut tout à fait se produire. En effet, le contexte d'exécution d'une assertion n'est pas nécessairement lié aux variables intervenant dans la condition de l'assertion : le concepteur est libre de préciser ce qu'il souhaite. Il est donc nécessaire de prévoir tous les cas possibles : variables non encore présentes dans l'architecture comme variables déjà disparues de l'architecture.

Pour permettre aux rapports d'être écrits dans la mémoire malgré la disparition d'une variable applicative du système, il faut employer un registre additionnel, ou *buffer*, dont le rôle est de copier la valeur de la variable pendant qu'elle existe dans le système, et de conserver cette copie tant qu'au moins un des rapports en a besoin.

Cette solution est illustrée dans la figure 45, qui montre l'apport matériel du gestionnaire d'erreurs. Les cases mémoires des trois rapports sont envoyées sur le port d'entrée des données de la mémoire lorsque des erreurs sont identifiées par les moniteurs associés, dans l'ordre donné par la configuration des données au sein de ces cases mémoire, comme illustré dans la figure 43. Ainsi, lorsque le premier moniteur détecte une erreur, l'identifiant H_1 est envoyé, puis la valeur de la variable a (donc le contenu du registre 1), puis b (registre 2) et enfin c (registre 3). Pour le rapport de l'assertion 2 par contre, la variable c doit être copiée dans un buffer additionnel pour pouvoir être écrite au bon moment. Tous les rapports nécessitant c (rapports 1 et 2) utilisent alors ce buffer comme source plutôt que

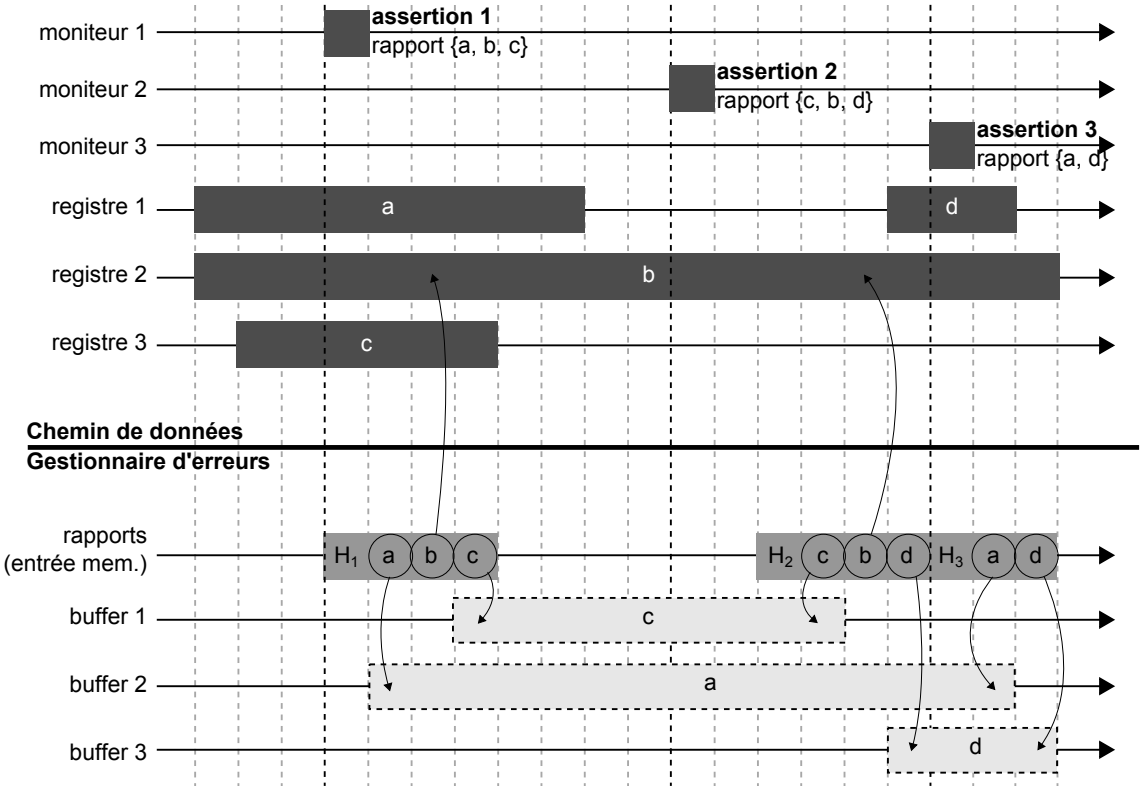

FIGURE 45 – Utilisation de registres additionnels pour copier les variables applicatives qui ne sont plus présentes au moment des rapports en ayant besoin

le registre 3, pour une raison liée au multiplexeur d'entrée de la mémoire, évoqué plus tard dans la section 2.1. De plus, le rapport de l'assertion 2 ne commence que deux cycles après l'identification de l'erreur, et non immédiatement. Cette contrainte est imposée par la variable d, qui n'est présente que cinq cycles après l'identification de l'erreur, alors qu'elle doit être écrite seulement trois cycles après l'identifiant H_2 du rapport. Enfin, le dernier rapport nécessite quant à lui la mémorisation de a, mais aussi de d, ce qui n'était pas forcément le cas à priori, d étant présente au moment où l'erreur est identifiée. Cette mémorisation supplémentaire est néanmoins imposée par la configuration des données au sein du rapport, qui fait que d doit être écrite dans la troisième case mémoire, soit deux cycles après le début du rapport.

Cet exemple fait apparaître des problèmes d'optimisation. En effet, il doit être possible de réduire le nombre de buffers, et donc de minimiser le coût matériel du gestionnaire d'erreurs. Par exemple, en permutant l'ordre d'écriture des données de d et a dans la mémoire lors du troisième rapport (on écrit l'identifiant, puis d, puis a), le buffer nécessaire pour d disparaît car cette variable sera encore présente dans le registre 1 au moment où il faudra l'écrire. De plus, un regroupement des données pourrait permettre de stocker plusieurs données par case mémoire, afin de mieux utiliser l'espace mémoire. Cette approche permet de réduire fortement la quantité de mémoire nécessaire pour mémoriser un même nombre de rapports.

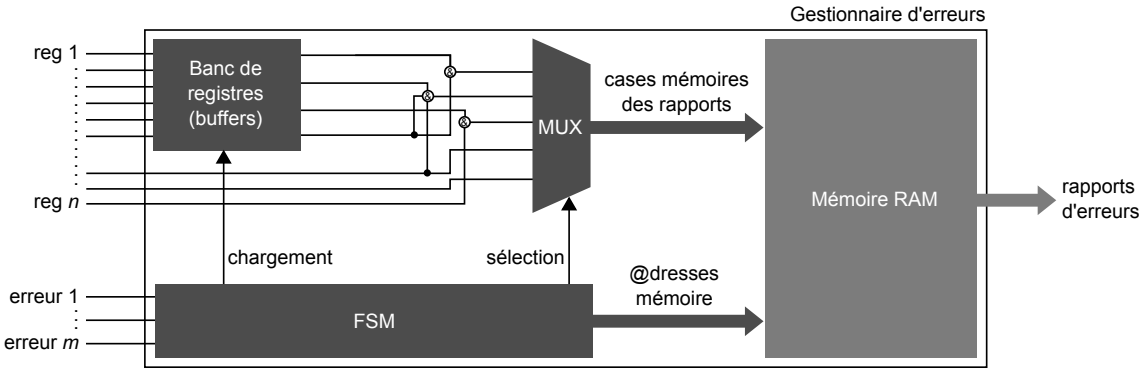

FIGURE 46 – Architecture matérielle générique du gestionnaire d'erreurs proposé

Le rôle du gestionnaire d'erreurs proposé est de mémoriser les variables applicatives spécifiées dans chaque rapport sans pour autant contraindre leurs durées de vie dans le chemin de données. Cela rappelle les problèmes d'adaptation spatio-temporelle, déjà traités dans la littérature [45, 37, 36, 77, 9]. Toutefois, le problème que nous adressons en diffère par ses contraintes. En effet, les adaptateurs spatio-temporels essaient d'arranger les sorties d'un composant A pour satisfaire aux contraintes d'entrée d'un composant B, telles que l'ordre des données ou les dates à laquelles ces données doivent être fournies. Dans le contexte présent, il n'y a aucune contrainte sur l'ordre de mémorisation des données dans les rapports, ni sur l'ordre d'écriture en mémoire des rapports eux-mêmes, ni non plus sur les dates auxquelles les rapports doivent être écrits. Par conséquent, les techniques d'optimisation permettant la réduction du coût du système sont plus nombreuses. Les points d'optimisation que nous avons identifiés sont présentés dans la section 3.

2.1 Projection matérielle

Le gestionnaire proposé repose sur une architecture matérielle générique dont les sous-composants sont adaptés en fonction des besoins issus de l'ordonnancement des rapports. La figure 46 présente cette architecture.

Les entrées $\{reg_1, \ldots, reg_n\}$ du gestionnaire d'erreurs sont liées aux registres du chemin de données qui contiennent les variables applicatives à mémoriser dans au moins un rapport. Les entrées nommées $\{erreur_1, \ldots, erreur_n\}$ sont câblées aux moniteurs matériels présents dans le chemin de données. Ces deux ensembles d'entrées sont les bus *données* et *erreurs* qui étaient présents dans la figure 40. La sortie du gestionnaire est constituée par un port d'accès à la mémoire. Ce port permet la récupération des différents rapports depuis le système externe.

Certaines variables ont besoin d'être mémorisées car leur temps de vie dans le chemin de données est insuffisant. Cette fonctionnalité est assurée par un banc de registres. Ces

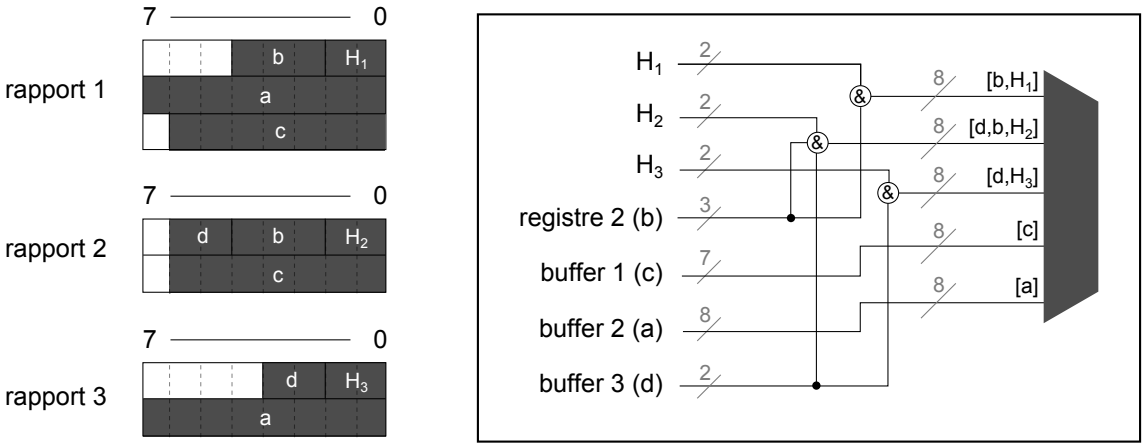

FIGURE 47 – Vue détaillée du réseau de câblage du multiplexeur, en fonction de la configuration des rapports présentés à gauche. Chaque arrangement unique d'une case mémoire crée une entrée sur le multiplexeur.

registres peuvent être réutilisés temporellement pour mémoriser différentes variables applicatives, issues de registres d'entrée différents. Ainsi, le nombre de bus entrant dans le banc de registres n'est pas nécessairement égal au nombre de bus en sortant.

Un réseau de câblage combine les données provenant des registres externes et des buffers vers le multiplexeur d'entrée de la mémoire. En effet, les registres ne sont pas forcément directement reliés individuellement au multiplexeur. Cela dépend de la configuration souhaitée pour les variables applicatives au sein des rapports. La figure 47 illustre cette particularité. Sur cette figure, les rapports sont organisés de façon à ce que plusieurs variables puissent être contenues dans une seule case mémoire. Cette configuration fait diminuer significativement la taille de la mémoire, mais a un impact sur le nombre d'entrées du multiplexeur de la mémoire. Sur la figure, les identifiants H_1, H_2 et H_3 ne sont pas directement reliés à une entrée du multiplexeur chacun, mais combinés avec les variables b et d selon les cas pour satisfaire la configuration demandée. Dans cet exemple simple, cette configuration permet de faire diminuer le nombre d'entrées du multiplexeur de 7 (dans la configuration de référence, si chaque variable était dans une case mémoire individuelle) à 5, tout en rajoutant une couche de logique combinatoire. Dans le cas où il existe plus de cases mémoires uniques que de variables applicatives et d'identifiants à mémoriser, le nombre d'entrées du multiplexeur augmente par rapport à la configuration de référence.

Enfin,un séquenceur génère les signaux de contrôle pour les différents composants en fonction des signaux d'erreur des moniteurs. Lorsqu'une erreur est détectée, le séquenceur pilote les écritures en mémoire ainsi que l'avancement du compteur d'adresse de la mémoire.

2.2 Utilisation efficace de la mémoire

Comme illustré précédemment, les rapports sont découpés sous forme de cases mémoires. La largeur en nombre de bits d'une case mémoire d'un rapport est contraint par la largeur de la mémoire. En effet, nous avons spécifié précédemment qu'une case mémoire d'un rapport doit être mémorisée en un cycle d'horloge. Les variables applicatives du contexte d'exécution sont découpées si nécessaire et organisées sous forme de *paquets*.

Définition 3. *Un paquet est un fragment de case mémoire contenant tout ou partie d'une donnée à enregistrer dans un rapport. Un paquet ne peut être plus grand que la largeur de la mémoire, qui définit aussi la largeur des cases mémoires des rapports. Ainsi, si une donnée nécessite plus de bits que cette largeur maximale, elle est scindée en autant de paquets que nécessaire.*

Par exemple, le rapport 1 est constitué des trois variables a, b et c de dynamique respectivement 8 bits, 3 bits et 7 bits, et l'identifiant h est codé sur 2 bits. Si l'on considère une mémoire de largeur 5 bits, alors :

- la largeur d'une case mémoire du rapport est de 5 bits,
- 6 paquets sont à mémoriser dans le rapport, $\{h, a_1, a_2, b, c_1, c_2\}$, car les variables a et c doivent être scindées en deux paquets chacune (ici a_1 représente les bits $a[4..0]$, et a_2 les bits $a[7..5]$),
- il faudra 4 cases mémoire au minimum pour stocker le rapport,
- le contenu des 4 cases pourrait être $[h, b]$, $[a_1]$, $[a_2, c_2]$ et $[c_1]$.

Dans cet exemple, la fusion et les découpage des données permet d'exploiter au mieux les caractéristiques de la mémoire, tout en réduisant le nombre d'adresses mémoire nécessaires au stockage du rapport. Sans fusion des données, il faudrait 6 cycles d'horloge pour mémoriser les 6 paquets (un paquet par case mémoire, donc par cycle). Le taux d'utilisation réel de la mémoire serait de 66,6%. Avec fusion des données, seuls 4 cycles sont nécessaires pour mémoriser les 6 paquets (car 4 cases mémoires). De plus, l'utilisation de la mémoire serait de 100%.

Cet exemple démontre que l'organisation des paquets au sein des cases mémoires impacte le temps nécessaire au stockage en mémoire des rapports, ainsi que l'efficacité de l'utilisation de la mémoire. La figure 48 présente le premier rapport contraint par trois largeurs de mémoire différentes, sans fusion des données au sein des cases mémoires. Pour une largeur de 8 bits, chaque variable peut être contenue dans un seul paquet, mais le rapport ne présente que 62,5% de données utiles. Lorsque la largeur de la mémoire descend, on constate que les variables a et c doivent être scindées en deux, puis trois paquets. Cela a

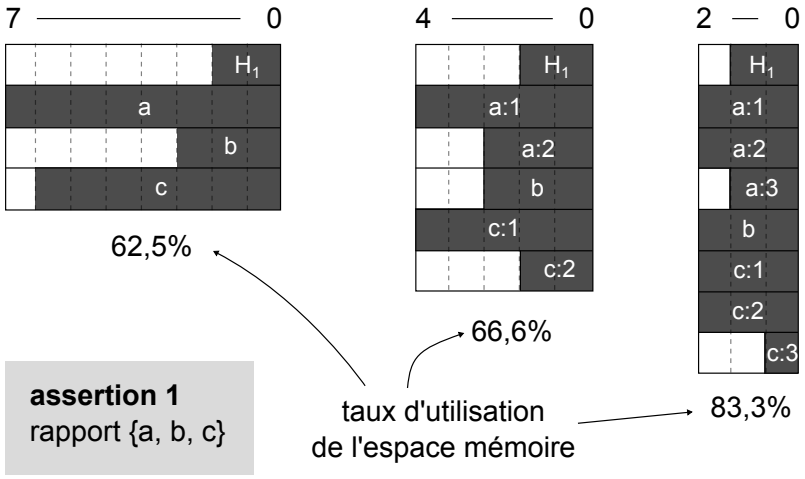

FIGURE 48 – Exemple du rapport (en configuration de référence) de l'assertion 1 pour trois largeurs de mémoire différentes (respectivement 8, 5 et 3 bits)

pour conséquence l'améliorer le taux d'utilisation à 83,3% avec une largeur de mémoire de 3 bits. Toutefois, cet avantage est contrebalancé par la latence qui augmente de 4 à 6 puis à 8 cycles d'horloge.

Étant donné que nous avons contraint notre système à ne posséder qu'une seule mémoire, un seul rapport peut être écrit à la fois. De ce fait, l'augmentation de la latence des rapports en fonction de la diminution de la largeur de la mémoire peut rendre impossible l'ordonnancement de tous les rapports durant une itération de l'application. Par exemple, la figure 49 montre l'ordonnancement des rapports en fonction de trois largeurs de mémoire différentes. Nous avions vu dans la figure 48 qu'une largeur de 3 bits était plus intéressante en terme de pourcentage d'utilisation de la mémoire. Toutefois, la figure 49 démontre que contraint par une telle largeur, il est impossible d'ordonnancer tous les rapports sans dépasser la contrainte de latence issues de l'application.

Afin de permettre l'analyse par le système ou par le concepteur des rapports enregistrés dans la mémoire, nous avons imposé le fait que les paquets d'identification des assertions doivent figurer au début des rapports. De plus, l'empreinte mémoire d'un rapport doit être contiguë. Cette contrainte empêche l'entrelacement des cases de différentes rapports en mémoire. L'entrelacement de plusieurs rapports pourrait être réalisé en ajoutant un champs supplémentaire en début de chaque case mémoire pour indiquer leur appartenance à un rapport spécifique, mais cela augmente d'autant plus l'espace mémoire nécessaire, alors que l'on souhaite le minimiser.

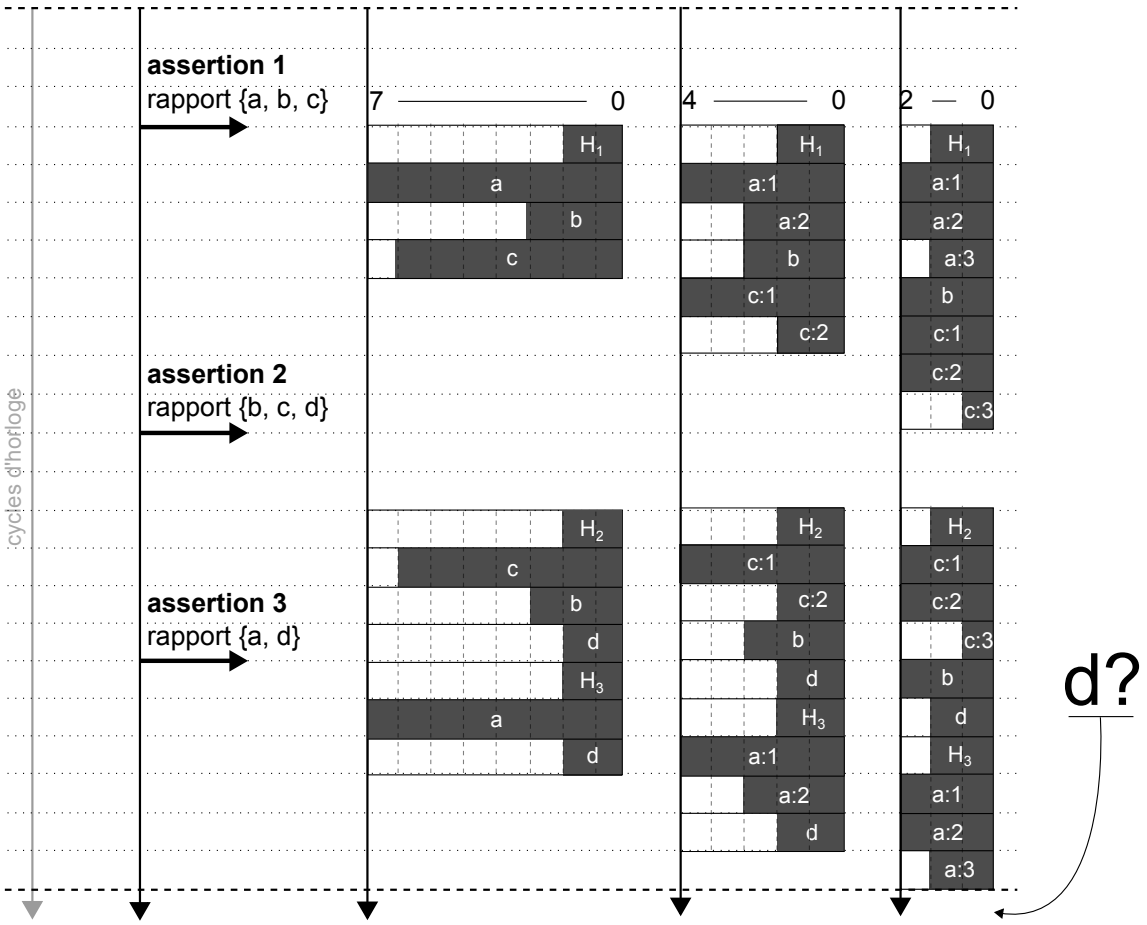

FIGURE 49 – Ordonnancement des rapports en fonction de trois largeurs de mémoire différentes : 8, 5 et 3 bits. Pour la largeur de 8 bits (une variable complète par case mémoire), cette figure présente la même vue que la figure 45, organisée différemment.

2.3 Coûts matériels

L'objectif lors de la conception de l'architecture matérielle est de permettre la mémorisation des rapports de toutes les assertions (pire cas), tout en minimisant la complexité matérielle du système. Le coût matériel de l'architecture n'est pas seulement composé du coût de la mémoire. En effet, afin d'implanter le comportement, d'autres composants matériels sont nécessaires :

Le banc de registres - Toute variable applicative devant être mémorisée dans un buffer entraîne la création d'un registre additionnel dans le système. La tailles des registres est égale à la dynamique des variables à mémoriser. Ce coût peut être réduit en partageant temporellement les registres lorsque les données ont des durées de vie disjointes. La dynamique des registres est alors égale à la dynamique maximale des variables qu'ils contiennent.

Le multiplexeur d'entrée de la mémoire - La figure 47 présentait l'architecture du

multiplexeur en fonction de la configuration des paquets à mémoriser dans un rapport. Les paquets sont répartis sur plusieurs cases mémoires. Ainsi, certaines cases mémoires peuvent être identiques dans plusieurs rapports différents (elles contiennent les mêmes paquets, dans le même ordre). C'est par exemple le cas entre la troisième case du rapport 1 (qui contient le paquet c) et la deuxième case du rapport 2. Ces deux cases sont donc routées sur la même entrée du multiplexeur. Ainsi, chaque case mémoire unique requiert une entrée dédiée sur le multiplexeur. De ce fait, il est possible de réduire le coût du multiplexeur en optimisant le placement des données dans les cases mémoires afin de faire apparaître le plus de similarités possibles entre les cases.

La logique de contrôle - Le coût du séquenceur de l'architecture est principalement lié à la génération des signaux de contrôle du banc de registres et du multiplexeur. La réutilisation des buffers pour plusieurs variables réduit le coût du banc de registres mais peut augmenter la quantité de logique combinatoire requise pour permettre cette réutilisation. De même, la réutilisation d'une entrée du multiplexeur pour les cases identiques de plusieurs rapports entraîne une réorganisation de la logique combinatoire associée. La complexité de cette logique de contrôle est difficile à quantifier à priori.

Le coût global de l'architecture peut être calculé à l'aide de l'équation 4.1. L'objectif de la méthodologie présentée dans la suite de ce chapitre est de minimiser cette fonction de coût. La latence de chaque rapport n'apparaît pas dans la fonction car elle est contraint par la latence de l'architecture, et donc n'apporte aucune pénalité au système (elle n'agit que sur la capacité du processus à trouver une solution d'ordonnancement des rapports).

$$cost = cost_{mem} + cost_{bufs} + cost_{mux} + cost_{control} \tag{4.1}$$

2.4 Définition des étapes de la synthèse

La méthodologie proposée dans ce chapitre vise à permettre la conception d'un gestionnaire d'erreurs efficace. Le point de départ de notre flot se compose d'une description des assertions dont nous devons mémoriser le contexte d'exécution, et d'un chronogramme détaillant les durées de vie des variables au sein du chemin de données.

L'approche proposée est une approche itérative basée sur deux étapes principales, chacune divisée en sous-tâches, comme cela est schématisé sur la figure 50. La première étape consiste à déterminer les bornes de l'espace des solutions. Cette étape est rapide car elle ne nécessite pas de synthèse logique ou physique, seule l'ordonnançabilité des rapports im-

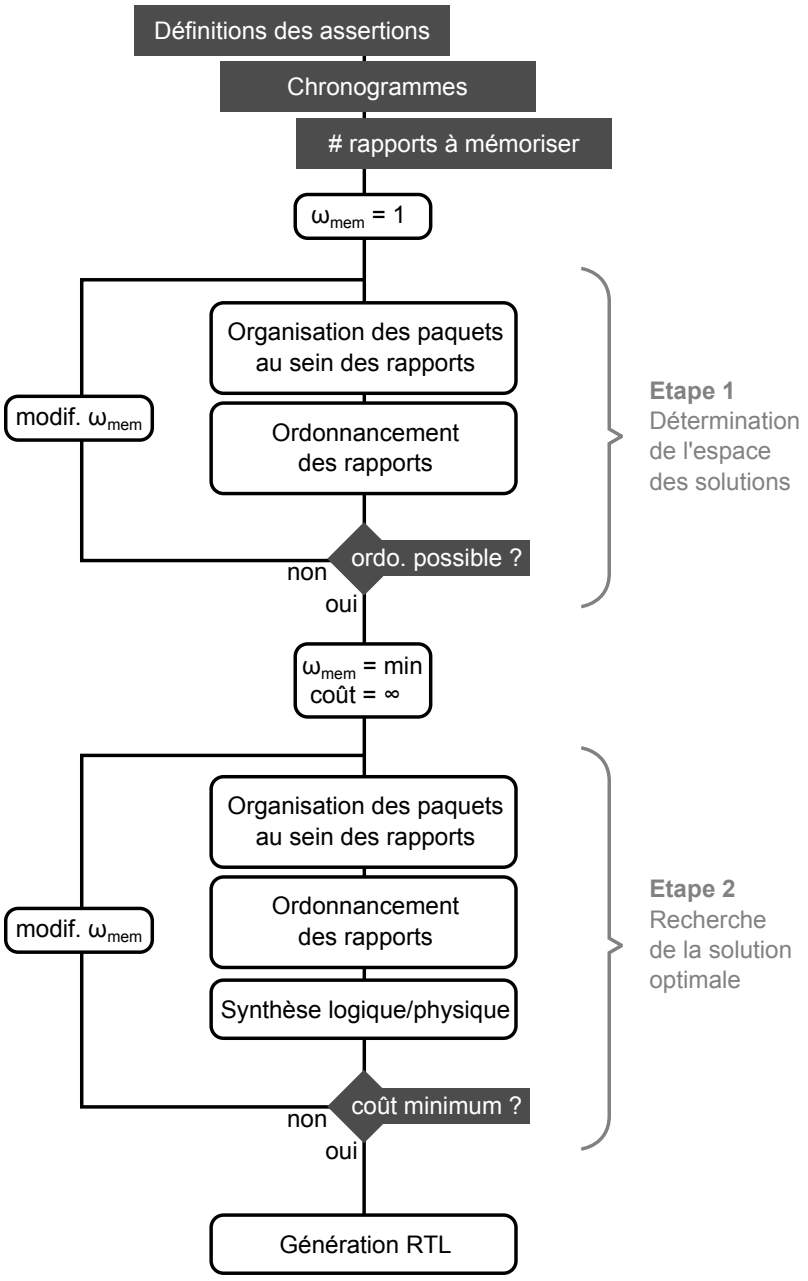

FIGURE 50 – Flot de synthèse en deux étapes itératives. ω_{mem} représente la largeur de la mémoire.

porte. La borne minimale de l'espace des solutions est recherchée par dichotomie (la borne maximale est calculable sans itération). La seconde étape consiste à parcourir l'espace de solutions ainsi borné à la recherche de la solution optimale. L'évaluation du coût matériel peut se faire par synthèse physique de l'architecture générée, ou par analyse approximative des caractéristiques de l'architecture si cette synthèse est trop contraignante.

Ces deux étapes sont divisées en tâches intermédiaires. Deux tâches sont communes aux deux étapes principales : l'organisation des paquets et l'ordonnancement des rapports. Ces

étapes seront présentées en détail dans les sections suivantes.

1. **Organisation des paquets au sein de chaque rapport -** La façon dont les paquets sont ordonnés au sein d'un rapport influence considérablement le coût matériel de l'ensemble de l'architecture (en quantité mémoire et quantité de logique). Cette étape s'occupe de regrouper les paquets dans les cases mémoires des rapports, en utilisant plusieurs techniques d'optimisation, comme la fusion des données au sein des cases mémoires et la minimisation des cases mémoires identiques.

2. **Ordonnancement des rapports -** Un seul rapport peut être mémorisé à la fois, car nous disposons dans l'architecture d'une seule mémoire et nous avons spécifié que les rapports ne peuvent être entrelacés. Afin de s'assurer que tous les rapports peuvent être mémorisés dans le pire cas, ils sont ordonnancés. Leurs dates d'écriture dépendent des dates d'écriture des autres rapports, de la disponibilité des données, et du nombre de paquets à écrire. De plus, on souhaite conserver l'ordre spécifié par le concepteur dans la description comportementale. Cette étape fait apparaître les buffers nécessaires à la mémorisation temporaire de certaines variables. Une technique d'optimisation de partage de ressource est utilisée pour réduire le nombre de registres nécessaires.

Finalement, une fois une solution obtenue, une description RTL du gestionnaire d'erreurs est générée à partir des informations compilées pour chaque rapport.

3 Étapes de la synthèse

Les variables applicatives du contexte d'exécution sont mémorisées à l'aide de registres dans le chemin de données. Toute variable applicative v possède une durée de vie finie pendant laquelle elle est disponible dans le système au sein d'un registre $reg(v)$ du chemin de données. Cette durée de vie débute au cycle $t_{min}(v)$ et se termine à $t_{max}(v)$. La dynamique de la variable est notée $\omega(v)$. Enfin, V désigne l'ensemble de toutes les variables explicitement nommées d'une description algorithmique. Toutes ces notions sont regroupées dans l'équation 4.2.

$$\forall v \in V, \quad v = \{reg(v), t_{min}(v), t_{max}(v), \omega(v)\} \tag{4.2}$$

La date d'exécution de la condition booléenne d'une assertion par un moniteur m, date à laquelle le résultat de l'assertion est connue, est notée $t_{exec}(m)$. A partir de cette date, la

mémorisation du rapport d'erreur associé au moniteur peut commencer si l'assertion n'est pas vérifiée (renvoie un booléen *false*). Toutefois, si toutes les variables à mémoriser ne sont pas disponibles à ce moment, ou si un autre rapport est déjà en cours d'écriture, la date de début d'écriture du rapport doit être décalée. La date de début du rapport est notée $t_r(m)$, avec $t_r(m) \geq t_{exec}(m)$. Le rapport nécessite au minimum autant de cycles d'horloge pour être écrit dans la mémoire qu'il possède de cases mémoires, car l'écriture d'un rapport dans la mémoire se fait case par case. Cette contrainte est due au fait que nous avons spécifié que les cases mémoires des rapports soient de même largeur que la mémoire, et donc que son port d'entrée. Cette durée de rapport est notée $\Delta_r(m)$. Le dernier paramètre d'un rapport est l'ensemble des variables applicatives qu'il doit contenir, $V_r(m)$. Enfin, M désigne l'ensemble de tous les moniteurs spécifiés pour l'application considérée. L'équation 4.3 résume tous ces paramètres. La notation V_r, avec $V_r \subset V$, sera utilisée comme raccourci pour désigner le sous-ensemble de V regroupant toutes les variables applicatives faisant partie d'au moins un rapport.

$$\forall m \in M, \quad m = \{t_{exec}(m), t_r(m), \Delta_r(m), V_r(m)\} \tag{4.3}$$

On appelle $SLOTS := \bigcup_{i=1}^{|M|} Slots(m_i)$ l'ensemble de toutes les cases mémoires des différents rapports de l'application. Pour un moniteur $m \in M$, $Slots(m)$ désigne l'ensemble des cases mémoires du rapport de ce moniteur. Aussi, pour toute case mémoire $slot \in SLOTS$, $Packets(slot)$ représente l'ensemble ordonné des paquets contenus dans cette case, et $PACKETS$ est l'union de tous les paquets de tous les rapports. Chaque paquet p est défini formellement par trois propriétés : la variable de laquelle il est issu ($v(p)$) ainsi que les bits de poids fort ($msb_v(p)$) et faible ($lsb_v(p)$) délimitant le paquet au sein du registre $reg(v)$, avec ($msb_v(p)$ et $lsb_v(p)$) dans l'intervalle $[0, \omega(v)]^2$, et $msb_v(p) \geq lsb_v(p)$.

$$\forall p \in Packets(slot), \quad p = \{v(p), msb_v(p), lsb_v(p)\} \tag{4.4}$$

Enfin, pour tout moniteur $m \in M$ et toute case mémoire de son rapport associé, $slot \in Slots(m)$, on définit le paramètre $t_{min}(slot)$ qui correspond au cycle d'horloge à partir duquel la case en question peut être écrite dans la mémoire. Cette date est au minimum $t_{exec}(m)$, date avant laquelle aucune case de m ne peut être écrite, et dépend ensuite des variables à enregistrer. Dans l'équation 4.5, $V(slot)$ est utilisé comme raccourci pour parler de l'ensemble des variables attachées aux paquets de la case mémoire.

$$\forall slot \in Slots(m), t_{min}(slot) = max(\{t_{exec}(m)\} \cup \{t_{min}(v), \forall v \in V(slot)\}) \quad (4.5)$$

L'ensemble $SLOTS$ permet de calculer le nombre d'entrées et la taille théorique du multiplexeur en entrée de la mémoire. En effet, si l'on appelle ω_{mem} la largeur de la mémoire, alors la taille théorique du multiplexeur est $size_{mux} = |SLOTS| \times \omega_{mem}$. Le cardinal de $SLOTS$ désigne le nombre de cases mémoires uniques (si deux cases de deux rapports différents sont identiques dans deux rapports différents, c'est à dire si elles contiennent les mêmes paquets, alors elles n'ont qu'une seule occurence dans $SLOTS$).

La méthode proposée est organisée en deux étapes principales composées de plusieurs tâches intermédiaires. Les parties suivantes décrivent d'abord ces tâches, puis détaillent le processus itératif de ces tâches mises en jeu dans les deux étapes.

3.1 Organisation des paquets au sein de chaque rapport

La composition des rapports, c'est à dire la façon dont sont répartis les paquets contenant les informations d'identification et des variables applicatives, impacte la profondeur de la mémoire et la structure du multiplexeur, mais aussi dans une moindre mesure le nombre de buffers additionnels requis ainsi la quantité de ressources liées à la logique de contrôle.

La configuration de référence illustrée en figure 51a, *en pile*, consiste à placer un paquet par case mémoire. Ainsi, elle nécessite autant d'entrées pour le multiplexeur qu'il n'y a de paquets formés à partir des éléments de V_r, auxquelles il faut rajouter une entrée par assertion pour tenir compte des identifiants. Cette configuration présente le gros inconvénient de faire perdre énormément de place dans la mémoire lorsque la dispersion des largeurs des paquets est importante. Tout paquet dont la largeur est inférieure à celle de la mémoire induit nécessairement une perte de place. Dans cette configuration, la profondeur de la mémoire nécessaire pour pouvoir enregistrer une occurrence du plus gros rapport, δ_{mem}, se calcule avec l'équation 4.6. Il convient de multiplier cette valeur par le nombre de rapports que l'on souhaite pouvoir mémoriser au minimum pour obtenir la profondeur totale de la mémoire.

$$\delta_{mem} = \sum_{j=1}^{|V_r(m_i)|} \lceil \omega(v_j)/\omega_{mem} \rceil \quad (4.6)$$

Une configuration alternative, appelée *continue*, consiste à placer toutes les variables les

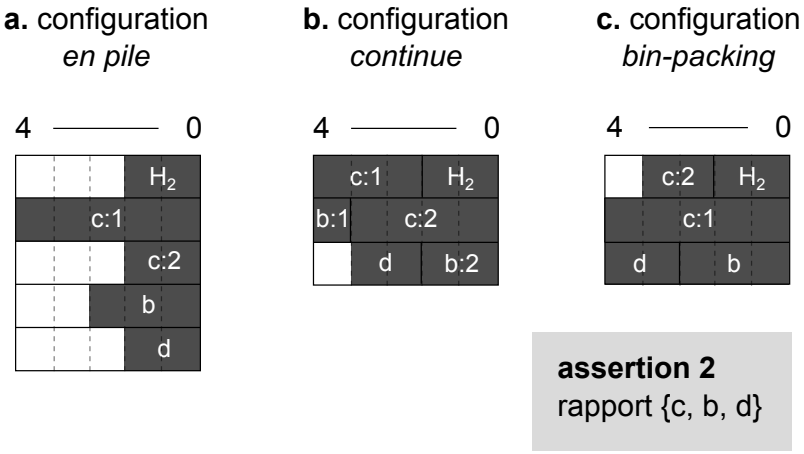

FIGURE 51 – Les différents types de configurations appliqués au rapport de l'assertion 2

unes à la suite des autres. Les paquets ne sont pas fixés à l'avance mais sont créés en fonction de la place des variables dans le rapport. La figure 51b illustre le résultat de cette configuration. Cette méthode permet de réduire au maximum la place perdue dans la mémoire. La profondeur de la mémoire est cette fois calculable à partir de l'équation 4.7. Le passage de la fonction de partie entière en dehors de la somme est significative du gain apporté par rapport à la configuration en pile. Toutefois, à l'inverse de la configuration en pile, la configuration continue est susceptible de générer un nombre très important d'entrées dans le multiplexeur. En effet, même si plusieurs rapports doivent enregistrer des variables en commun, celles-ci peuvent être scindées en paquets de façon très différentes entre les rapports. Cela réduit donc fortement la probabilité de trouver des cases similaires entre les rapports, et augmente donc d'autant la complexité du multiplexeur.

$$\delta_{mem} = \left\lceil \sum_{j=1}^{|V_r(m_i)|} \omega(v_j)/\omega_{mem} \right\rceil \tag{4.7}$$

Nous proposons d'utiliser une approche dédiée pour ce problème. La configuration proposée, illustrée dans la figure 51c, est un compromis entre les deux approches précédentes. Elle utilise la définition standard des paquets, c'est à dire qu'une variable ne peut être divisée en paquets que d'une seule et même façon au sein de tous les rapports (ce qui n'est pas le cas en configuration continue). De plus, une case de rapport peut contenir plusieurs paquets. Nous avons choisi de répartir ces paquets au travers des cases en suivant l'approche du *first-fit decreasing algorithm* (FFD) de résolution des problèmes de *bin-packing* [41]. Plusieurs heuristiques existent pour trouver une solution (souvent sous-optimale) à ce problème NP-complet. La plus simple est l'algorithme dit *first-fit* (FF) : c'est un algorithme glouton qui traite les données dans l'ordre où elles arrivent. L'algorithme dit *best-fit* (BF) teste chaque donnée pour la placer dans la case mémoire qui après insertion de la

donnée présentera le plus fort taux d'occupation. Il est démontré que ces deux algorithmes utilisent au plus $17/10OPT+2$ cases mémoires [41], où OPT représente le nombre de cases de la solution optimale. Une optimisation simple consiste à trier les données à placer par ordre de taille décroissante avant d'exécuter l'algorithme FF ou BF. On obtient ainsi les algorithmes *first-fit decreasing* (FFD) et *best-fit decreasing* (BFD). Il a été démontré que ces algorithmes permettent l'utilisation d'au plus $11/9OPT + 1$ cases mémoires [179] (la démonstration initiale nécessitait plus de 100 pages, nombre ramené 20 ans plus tard à 9). Les performances de ces deux algorithmes ont été montrées comme étant équivalentes [12]. Le listing 4.4 détaille l'algorithme FFD que nous utilisons. Il prend comme paramètre la liste des paquets à organiser et renvoie la liste des cases mémoires (nommées *bins* dans l'algorithme) utilisées pour contenir ces paquets. Sa complexité algorithmique est O(n log(n)).

Listing 4.4– Algorithme *first-fit decreasing* de résolution du problème de *bin-packing*

```
1  def binPackingFFD(packets):
2      bins = []
3      add new bin to bins
4
5      sort(packets)
6
7      for p in packets:
8          for b in bins:
9              if b.usedsize + p.size <= b.size:
10                 add p to b
11                 break
12
13     return bins
```

Plusieurs heuristiques plus complexes existent dans la littérature, traitant notamment le problème pour les situations *online* (pas de possibilité de trier les données en amont) [66] ou multi-dimensionnelles [38]. Nous avons délibérément choisi de nous concentrer sur une heuristique présentant un bon compromis entre performance et complexité calculatoire. Ce choix est nécessaire car nous souhaitons à la fois accélérer la synthèse du gestionnaire d'erreurs dans un contexte d'utilisation ponctuelle, pour détecter une erreur en émulation sur FPGA par exemple, et aussi permettre de générer une architecture aussi peu coûteuse que possible pour permettre l'utilisation à long-terme du gestionnaire, en le laissant dans le circuit final.

Cette répartition donne des résultats meilleurs que la configuration en pile dès lors qu'il existe au moins deux paquets p_1 et p_2 tels que $\omega(p_1) + \omega(p_2) \leq \omega_{mem}$. En effet, il est impossible que des cases mémoires aient un taux d'occupation inférieur à 50% en même temps. La raison pour cela est que si l'algorithme peut placer un paquet dans une case mémoire, il le fera et n'en ouvrira pas une nouvelle. Cela contribue à augmenter le taux d'occupation par rapport à la configuration en pile, et à baisser la latence du rapport

d'une unité pour chaque paquet venant compléter une case mémoire où une donnée est déjà présente.

Les dates de disponibilité des variables des paquets ne sont pas prises en compte lors de la répartition de ces paquets dans les cases mémoires. En effet, le coût de la mémoire est considéré comme prépondérant dans l'architecture (considération appuyée par les résultats présentés dans le chapitre suivant). Contraindre l'algorithme de bin-packing par les dates des paquets peut faire potentiellement baisser le taux d'utilisation de la mémoire, en empêchant certains paquets d'être fusionnés au sein d'une même case mémoire. Cela peut avoir un fort impact sur la taille de la mémoire lorsque l'on souhaite mémoriser quelques centaines ou milliers de rapports. Toutefois, nous proposons une optimisation de même but mais sans impact sur le taux d'utilisation de la mémoire : les différentes cases mémoires d'un rapport sont réordonnées les unes par rapport aux autres par $t_{min}(slot)$ croissant. Cela permet de diminuer potentiellement le nombre de buffers nécessaires sans avoir d'effet négatif sur la taille de la mémoire.

Une dernière optimisation est aussi utilisée. On dit que deux cases mémoires $slot_1$ et $slot_2$ sont équivalentes si leurs paquets sont tous équivalents deux-à-deux. C'est à dire si pour tout paquet p_1 de la première case, il existe un paquet p_2 dans la deuxième case tels que la paire (p_1, p_2) vérifie l'équation 4.8, et inversement pour tous les paquets de la deuxième case.

$$\begin{aligned} \forall (p_1, p_2) \in PACKETS^2, \quad p_1 \equiv p_2 \quad \Leftrightarrow \quad & reg(v(p_1)) = reg(v(p_2)) \\ \wedge \quad & msb(p_1) = msb(p_2) \\ \wedge \quad & lsb(p_1) = lsb(p_2) \end{aligned} \tag{4.8}$$

Cette équation signifie que deux paquets sont équivalents s'ils contiennent une même zone d'un même registre. Les variables dont les paquets sont issus n'ont pas d'importance du moment qu'elles sont disponibles dans le même registre de l'architecture matérielle. Par exemple si deux paquets doivent mémoriser respectivement x et y, toutes deux mémorisées dans le registre reg_1 du chemin de données, alors les deux paquets sont équivalents. Ainsi, si deux cases ont des paquets équivalents deux-à-deux, l'accès à la mémoire par ces deux cases se fait par la même entrée du multiplexeur. L'optimisation proposée est donc de tester, pour chaque case $slot \in SLOTS$, toutes les autres cases afin de trouver toutes les cases équivalentes à $slot$. Chaque case équivalente à $slot$ est alors réorganisée pour que ses paquets apparaissent dans le même ordre que ceux de la case $slot$. Cela permet de faire baisser le nombre de cases mémoires uniques de l'ensemble $SLOTS$, et donc de réduire la complexité du multiplexeur sans ajouter de surcout matériel.

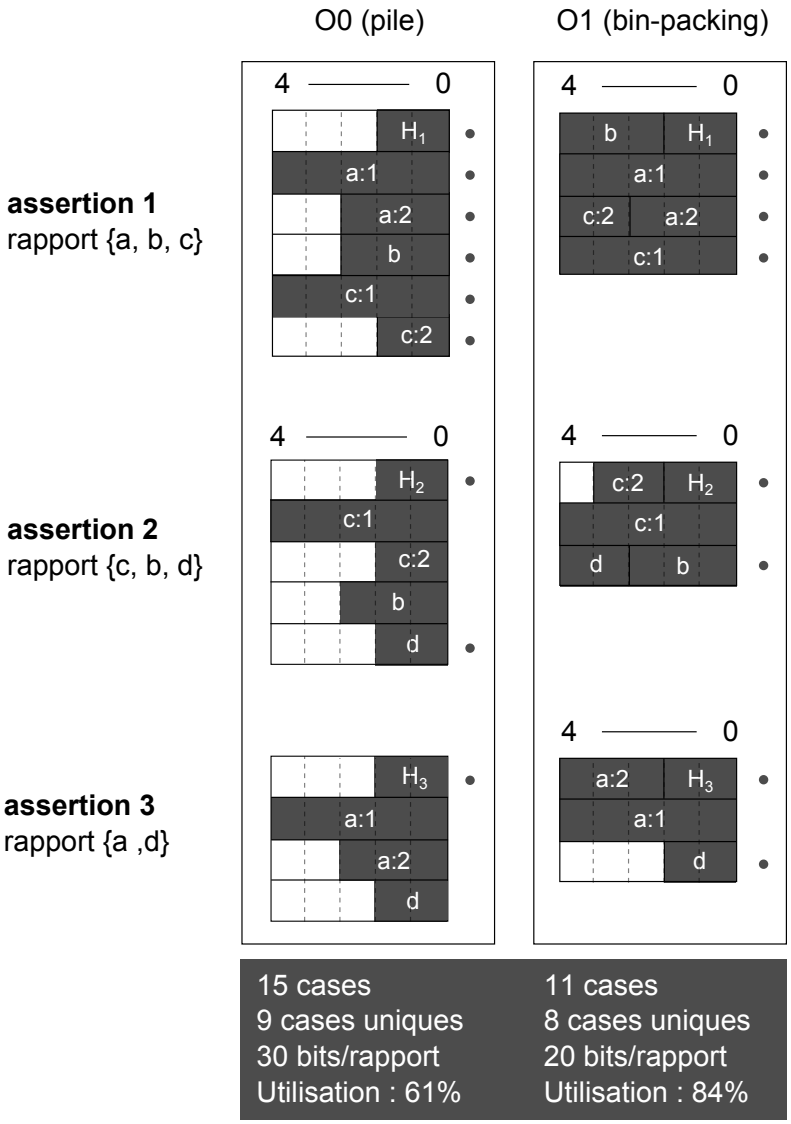

FIGURE 52 – Évolution des rapports en fonction des optimisations. Les points représentent les cases uniques dans chaque configuration.

Cette approche (répartition des paquets, ordonnancement puis optimisation des cases) constitue l'optimisation globale *O1* de la méthode, par comparaison avec la configuration de référence, en pile, qui ne présente aucune optimisation et est notée *O0*. La figure 52 montre l'impact de *O1* comparé à *O0* sur les rapports de l'exemple pédagogique, avec une largeur de mémoire de 5 bits. Le nombre de bits par rapport exprimé correspond à la taille de mémoire nécessaire pour contenir le plus gros des rapports, et par conséquent n'importe quel autre. On constate que la mémoire présente un meilleur taux d'utilisation, et que le nombre de cases mémoires uniques a baissé.

3.2 Ordonnancement des rapports

Une fois la configuration des cases connues pour chaque rapport, ces derniers ont besoin d'être ordonnancés pour éviter les conflits d'accès à la mémoire (un seul rapport peut être écrit à la fois). Pour chaque moniteur $m \in M$, le rapport associé ne peut pas commencer avant la date $t_{exec}(m)$. De plus, pour chaque case $slot \in Slots(m)$, l'écriture de la case en question dans la mémoire ne peut se faire avant la date $t_{min}(slot)$. Chaque case nécessite un cycle d'horloge pour être écrite dans la mémoire. Enfin, si une case doit être écrite à une date pendant laquelle l'une des variables applicatives représentées par les paquets de $P(slot)$ n'est plus disponible dans le système, alors un buffer additionnel doit être instancié dans l'architecture pour mémoriser temporairement cette variable. Ce buffer est dimensionné au minimum, c'est à dire que sa largeur ne fait pas la largeur du registre contenant la variable à dupliquer, mais la largeur de la variable elle-même $\omega(v)$.

Ainsi, il est possible d'identifier au préalable l'ensemble minimum des variables qui auront besoin d'être mémorisées temporairement, quel que soit l'ordonnancement appliqué à la configuration. Ces variables sont définies par l'ensemble $V_{bufs} \subset V_r$, décrit dans l'équation 4.9.

$$V_{bufs} := \{v \in V_r,\ \exists slot \in Slots,\ t_{min}(slot) > t_{max}(v)\} \tag{4.9}$$

L'algorithme d'ordonnancement est le suivant. D'abord, tous les rapports sont considérés continus dans leur écriture. C'est à dire que chaque case d'un rapport est écrite au cycle d'horloge suivant l'écriture de la case précédente. Ensuite, la mobilité de chaque moniteur, $mob(m)$ est définie comme l'intervalle entre une date minimum $mob_{min}(m)$ et une date maximum, $mob_{max}(m)$. $mob_{min}(m)$ est le cycle d'horloge c tel que pour chaque case mémoire $slot \in Slots(m)$, les variables de $V(slot)$ sont toutes disponibles si le rapport commence au cycle c, mais au moins l'une de ces variables n'est pas encore disponible si ce rapport commence au cycle $c-1$. La date $mob_{max}(m)$, quant à elle, correspond à la date de fin de l'application.

Un algorithme efficace pour calculer $mob_{min}(m)$ consiste à la définir comme égale à $t_{min}(slot_i)$, avec $i = |Slots(m)|$ (la dernière case du rapport), et de tester sa validité par itérations, en soustrayant un cycle à chaque itération jusqu'à ce que le rapport ne soit plus valide.

Une fois les mobilités calculées, l'ordonnancement est fait en commençant par le premier rapport, par ordre de date d'écriture. Cet ordonnancement, s'il réussi, peut générer des buffers additionnels dans l'architecture par rapport à ceux identifiés comme obligatoires.

L'ensemble des variables nécessitant un buffer, V'_{bufs} est donné dans l'équation 4.10.

$$V'_{bufs} := \{v \in V_r,\ \exists slot \in Slots,\ t_{write}(slot) > t_{max}(v)\} \tag{4.10}$$

Cependant, le nombre de ces buffers peut être réduit une fois l'ordonnancement effectué. Les techniques décrites ici définissent l'optimisation globale *O2* de la méthode. Ces techniques sont les suivantes :

– Une première possibilité consiste à supprimer la contrainte de continuité des rapports (celle-ci n'étant utilisée que pour faciliter l'ordonnancement). De ce fait, certains rapports peuvent voir leur date de début d'écriture reculer de quelques cycles, supprimant potentiellement la nécessité d'un buffer. La figure 53 illustre cette optimisation. Dans cette figure, la système est composé d'une unique assertion x à des fins d'illustration. La disponibilité de la variable d contraint la date de la troisième case du rapport. La contrainte de continuité impose aux deux autres cases d'être accolées à la troisième. Cet ordonnancement requiert donc un buffer pour la variable c, qui n'est plus disponible au moment où elle doit être écrite dans la mémoire. Sans cette contrainte de continuité, les deux premières cases mémoires du rapport peuvent être déplacées dans le temps, ce qui dans cet exemple permet de supprimer le besoin de mémorisation de c.

– Les registres matériels assignés aux buffers sont réutilisés en fonction des durées de vie de ces buffers, au lieu d'assigner un registre par buffer. Cette approche est liée au problème de *channel routing* [178], qui consiste dans le domaine des circuits matériels à router un ensemble de liens entre deux composants dans un canal de transmission à deux couches (une couche permet un routage horizontal, l'autre vertical). Dans notre cas, un canal horizontal représente un registre matériel, et les liens à router sont les buffers. Ainsi, ce qui nous intéresse ici se limite au routage horizontal, sans contrainte verticale : on cherche à minimiser le nombre de canaux horizontaux, donc de registres. L'algorithme le plus connu dans ce domaine, que nous utilisons dans la méthode, est le *unconstrained left-edge algorithm* [178]. Cette optimisation permet de réduire le coût matériel du banc de registres dès lors qu'il existe au moins deux variables à mémoriser temporairement, dont les durées de vie ne se chevauchent pas.

Toutefois, la définition de $mob_{max}(m)$ donnée ici – égale à la durée d'exécution de l'application – pose problème. En effet, tous les rapports ne peuvent pas nécessairement être écrits dans la mémoire avant la fin de l'itération courante de l'application. Notamment, le résultat d'une assertion de postcondition est par définition connu au dernier cycle de la durée de vie de l'application, ce qui empêche son rapport de pouvoir se faire avant la fin (un rapport nécessite au minimum un cycle pour être écrit, s'il ne contient qu'une seule case mémoire).

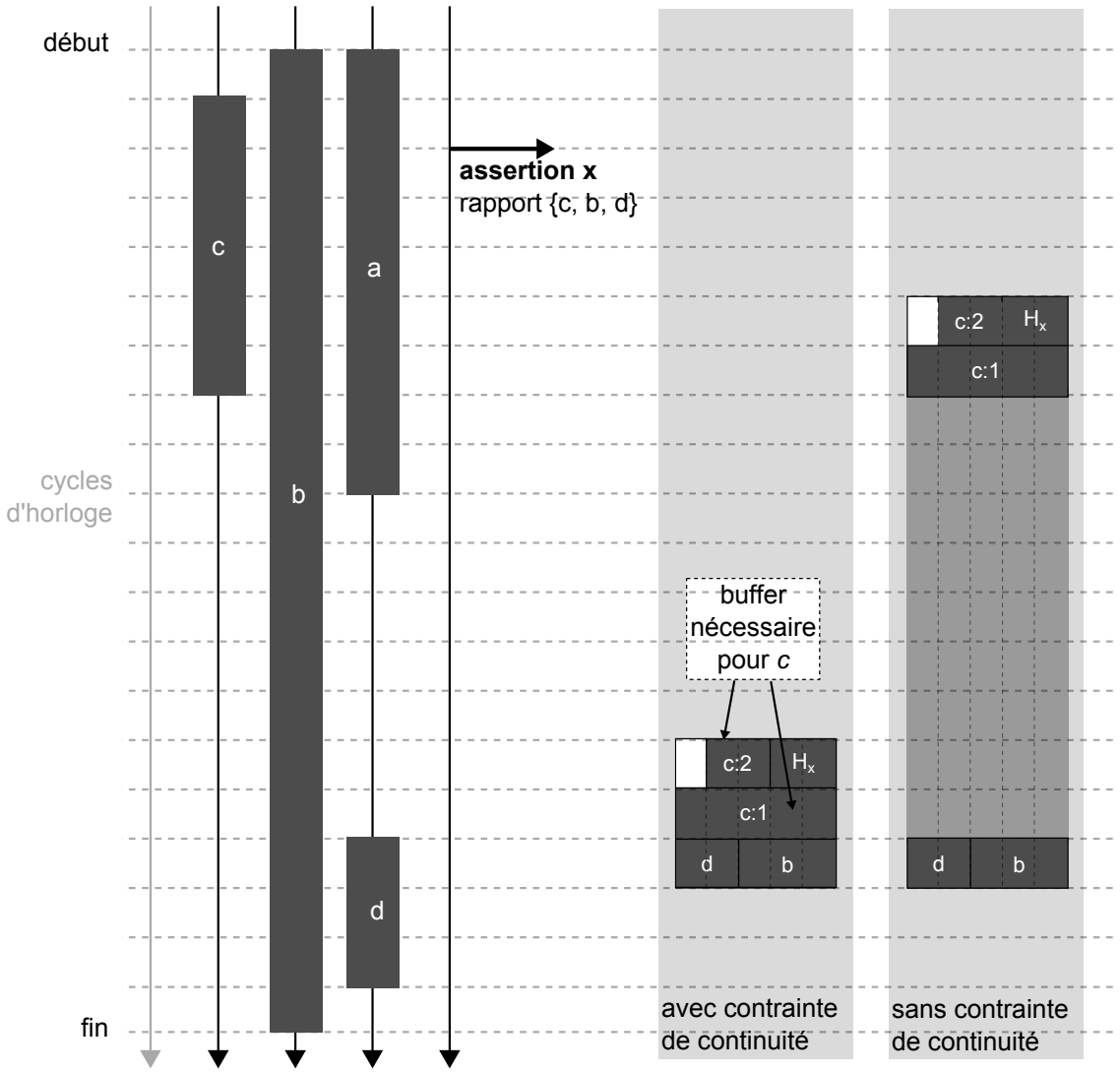

FIGURE 53 – Sans contrainte de continuité, un rapport peut voir sa date de début reculer (si applicable) afin de supprimer un besoin de mémorisation d'une ou plusieurs variables à enregistrer.

Pour résoudre ce problème, nous avons décidé de laisser les rapports s'effectuer lors du début de l'itération suivante si besoin. La date $mob_{max}(m)$ correspond ainsi à deux fois la latence de l'architecture. Afin de limiter la création de buffers, ces rapports sur itération suivante sont contraints à être effectués avant les rapports propres à cette deuxième itération. La figure 54 présente le problème et sa solution. Dans cette figure, une assertion de postcondition (assertion 4) a été ajoutée à l'exemple pédagogique pour mettre en avant ce problème. Le passage du quatrième rapport sur l'itération suivante entraîne le décalage du premier rapport d'un cycle vers l'avant. Ce quatrième rapport doit enregistrer les variables a, b et c telles qu'elles étaient dans l'itération 1 et non telles qu'elles sont dans l'itération 2. De ce fait, chacune de ces variables est nécessairement mémorisée dans un buffer individuel. Ainsi, les assertions de postcondition entraînent nécessairement un surcout matériel incompressible lié à la mémorisation temporaires des données du contexte d'exécution (si

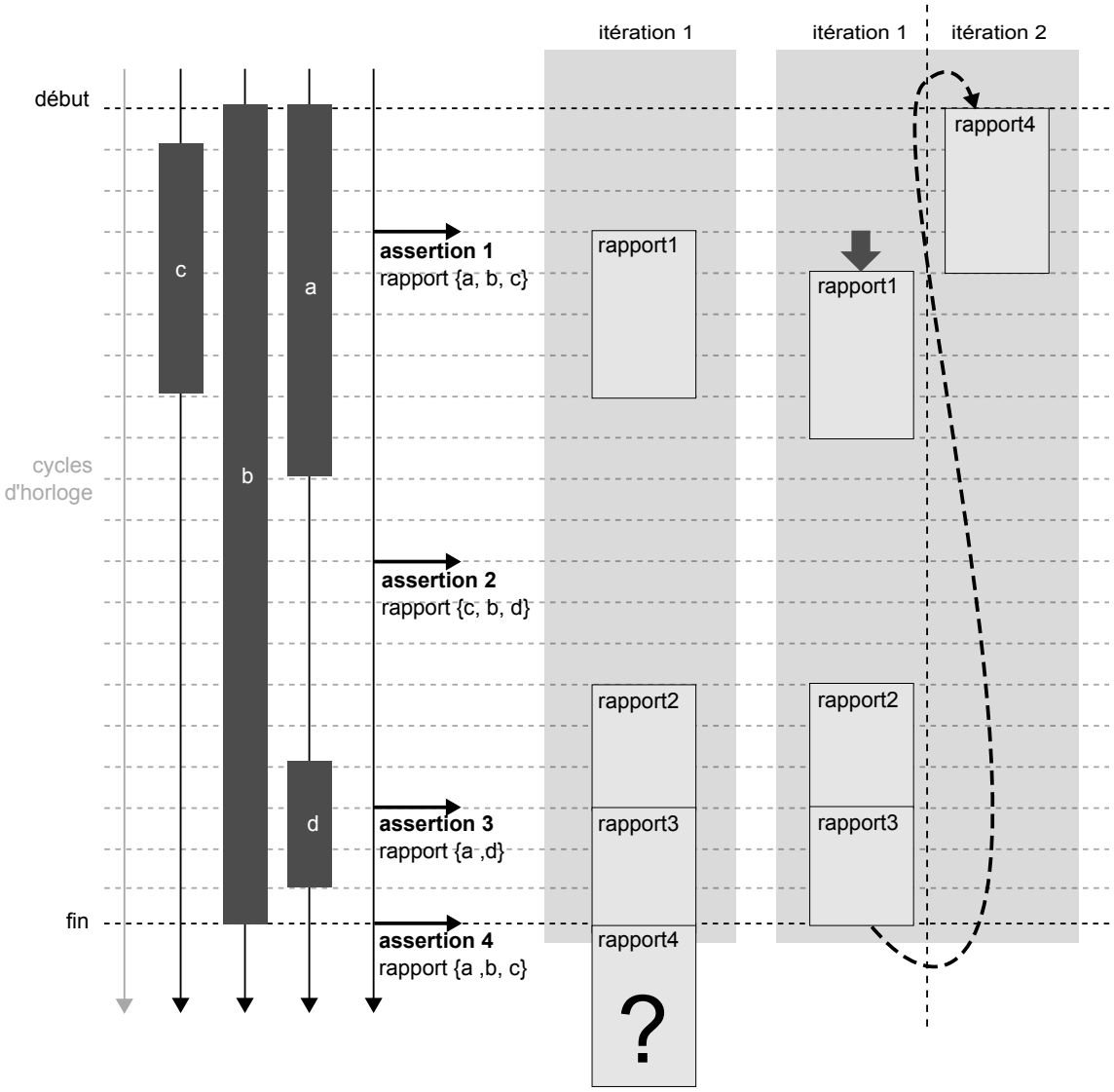

FIGURE 54 – Un rapport de postcondition ne peut pas, par définition, se faire avant la fin de la durée de vie de l'application. Il est donc enregistré lors de l'itération suivante, décalant les autres rapports dans le temps.

un contexte a été défini pour ces assertions).

3.3 Étapes d'itération du flot

Les deux tâches intermédiaires présentées sont utilisées au sein de deux étapes principales. L'objectif de la première étape est de définir les limites minimum et maximum pour la largeur de la mémoire, et ainsi de borner l'espace des solutions d'ordonnancement. La seconde étape a pour objectif l'identification de la solution optimale.

Le processus d'itération mis en place pendant la première étape vise uniquement la dé-

couverte de la borne minimale pour la largeur de la mémoire. En effet, la borne maximale peut être calculée directement à partir des données du système. Cette borne, $\omega_{mem_{max}}$, est donnée dans l'équation 4.11. Dans cette équation, ω_{id} représente la dynamique nécessaire pour coder un identifiant de moniteur dans un rapport. Cette borne maximum est donc la largeur nécessaire pour que le rapport le plus volumineux tienne sur une seule case mémoire.

$$\omega_{mem_{max}} = \omega_{id} + max\left(\bigcup_{m \in M}\left\{\sum_{i=1}^{|V(m)|}\omega(v_i)\right\}\right) \tag{4.11}$$

Afin de déterminer la borne minimum de l'espace des solutions, deux possibilités sont envisagées :

– soit choisir initialement une largeur de 1 bit et tester la possibilité d'ordonnancement des rapports via la méthodologie définie par les deux sous-étapes, puis réitérer en augmentant la largeur de 1 à chaque fois tant que l'ordonnancement est impossible,

– soit commencer avec une largeur de 1 bit, puis itérer par dichotomie entre 1 et $\omega_{mem_{max}}$.

Une fois l'espace des solutions borné, il est dans certains cas intéressant de rechercher la solution optimale relative au coût matériel de l'architecture. En effet, si la synthèse d'un gestionnaire d'erreurs est uniquement effectuée pour la vérification d'un système en émulation sur FPGA, et que les moniteurs ne sont pas gardés dans le circuit final, n'importe quelle solution d'ordonnancement peut être utilisée (la borne minimum et la borne maximum de l'espace des solutions sont directement utilisables). Toutefois, si les moniteurs et le gestionnaire doivent être gardés dans le circuit, la recherche d'une solution optimale est nécessaire.

La recherche de la solution optimale nécessite d'itérer l'ordonnancement et la synthèse physique de l'architecture générée pour chaque largeur de mémoire de l'espace des solutions. Toutefois, si la synthèse de l'architecture est trop contraignante (système très complexe, synthèse très longue), il peut être préférable d'évaluer approximativement les caractéristiques de l'architecture pour en déduire une solution convenable. Dans cette optique, le chapitre suivant présente l'évolution des caractéristiques de plusieurs architectures ainsi que les résultats de synthèse associés, afin de faire apparaître une corrélation entre les deux.

4 Conclusion

Ce chapitre a présenté une méthode pour la synthèse automatique d'un gestionnaire d'erreurs dédié aux architectures générées par HLS. Le rôle de ce gestionnaire est d'analyser en temps réel les événements des moniteurs matériels intégrés à l'architecture, issus par exemple des travaux de la littérature. Chaque signal d'erreur rapporté par un moniteur déclenche l'écriture dans une mémoire d'un rapport décrivant l'identification du moniteur source, mais aussi et surtout la valeur d'une ou plusieurs variables de la description de haut-niveau, telles qu'elles étaient au moment de l'erreur.

Ainsi, cette proposition, tout comme la précédente, contribue à l'amélioration du processus de vérification des systèmes. Elle vise en particulier un plus grand confort des concepteurs lors des tests en émulation sur FPGA. En effet, au contraire de l'émulation sur FPGA, la simulation d'un système permet d'obtenir une grande quantité d'informations lorsqu'une erreur est détectée, car toutes les variations de l'ensemble des signaux sont conservées par le simulateur. Le gestionnaire proposé permet donc de fournir une partie de ces signaux au concepteur, offrant ainsi un contexte d'exécution à chaque erreur et facilitant d'autant l'analyse des erreurs. Le gestionnaire proposé peut aussi être laissé dans le circuit final. Il est alors possible de récupérer les dernières erreurs ayant eu lieu directement sur site, ce qui permet l'identification et la mémorisation d'erreurs rares et difficiles à reproduire sur un banc de test standard.

Chapitre 5

Expériences et résultats

Sommaire

Ce chapitre présente les expériences qui ont été menées pour valider les contributions détaillées dans les chapitres précédents. Plusieurs expérimentations sont décrites et leurs résultats analysés.

1 Introduction

Deux méthodologies ont été présentées dans ce manuscrit. La première cible l'amélioration du processus de vérification en simulation des architectures générées par synthèse HLS, et la seconde contribue à la vérification en fonctionnement réel de ces architectures. Dans ce chapitre, nous souhaitons évaluer ces deux apports avec plusieurs expérimentations.

Les expérimentations réalisées dans le cadre de la première contribution ont visé d'une part à valider le fonctionnement des moniteurs générés. D'autre part, nous avons cherché à mesurer l'impact des moniteurs sur le temps de simulation des systèmes, afin de mesurer si le confort induit par le surplus d'informations fournies au concepteur était plus important que la gêne occasionnée par un temps de simulation plus grand.

Enfin, le gestionnaire d'erreurs proposé dans la seconde contribution cible à la fois l'émulation sur FPGA, mais aussi la vérification continue sur site. Nous avons évalué dans ce chapitre les performances de la méthodologie proposée en considérant deux critères : la complexité matérielle de l'architecture générée et le temps nécessaire à la recherche de la solution optimale.

Les sections suivantes présentent tour à tour les expériences effectuées pour les deux méthodes présentées, et en détaillent les résultats. Nous avons suivi deux approches différentes pour ces expériences. Nous avons évalué la première méthodologie sur des applications usuelles de traitement du signal (DCT 1D, DCT 2D, JPEG). La seconde méthodologie – volontairement généraliste pour être applicable à tout type d'application – a été évaluée à partir de *modèles d'applications aléatoires* (les paramètres requis par la méthode, comme les durées de vie des variables, sont générés aléatoirement). Cette approche est volontairement différente de la première afin de mettre ici en avant des tendances générales qui peuvent être retrouvées dans des cas d'études réels.

2 Moniteurs pour la simulation

La première méthode proposée cible la génération de moniteurs RTL à partir des assertions spécifiées dans la description comportementale d'une application. Le flot de génération détaillé est composé de plusieurs étapes qui s'intègrent dans un outil de synthèse de haut-niveau. Il est donc nécessaire de pouvoir modifier le flot de synthèse d'un outil.

Afin de valider la méthode proposée, cette dernière a été implémentée dans l'outil de synthèse HLS GraphLab [132, 101, 35, 65]. Cet outil permet la transformation d'algorithmes

Application	# assertions	# lignes MATLAB	# lignes PSL	# lignes VHDL (sp)	# lignes VHDL (mp)
DCT 1D 8x8	16	8	16	109	543
DCT 2D 8x8 (faible lat.)	320	17	320	1089	5743
DCT 2D 8x8 (grande lat.)	320	17	320	1125	5835
JPEG (faible lat.)	896	26	896	2777	16613
JPEG (grande lat.)	896	26	896	2883	16647

TABLE 2 – Comparaison du nombre de lignes de code générées dans les descriptions RTL des applications, en fonction du nombre d'assertions

de traitement du signal, spécifiés en langage MATLAB, en architectures RTL décrites en VHDL. Les descriptions algorithmiques sont analysées et transformer en premier lieu en une représentation abstraite sous forme de graphe dirigé acyclique (DAG) et bi-partite, puis les étapes classiques de la synthèse HLS ont lieu.

La validation de la méthode a reposé sur l'insertion de plusieurs assertions dans différentes descriptions d'applications usuelles de traitement du signal : une DCT 8X8 à une dimension, une DCT 8x8 à deux dimensions et un algorithme de compression JPEG. Les synthèse des applications DCD 2D et JPEG ont été contraintes de deux façons différentes : à l'aide d'une contrainte de faible latence puis avec une contrainte de latence plus lâche. Une faible latence force la phase d'ordonnancement à ordonnancer plus d'opérations à chaque cycle d'horloge, résultant en l'utilisation de plus de ressources matérielles. L'intérêt d'utiliser ces deux contraintes est de permettre la génération de circuits de complexité différentes à partir d'une même description algorithmique.

Les architectures générées ont été simulées plusieurs fois avec des stimuli aléatoires contraints pour pouvoir vérifier le comportement d'une assertion spécifique à chaque simulation. Cette phase a permis de s'assurer de l'équivalence correcte entre les moniteurs générés et les assertions de la description algorithmique.

L'impact de l'intégration des moniteurs sur les descriptions RTL a aussi été évalué. Le tableau 2 montre le nombre de lignes de code générées pour implémenter les moniteurs à partir des assertions comportementales. Dans ce tableau, la notation *VHDL (sp)* indique que l'implémentation VHDL des moniteurs se base sur un seul et unique process, tandis que la dénomination *VHDL (mp)* spécifie que l'implémentation utilise un process VHDL par moniteur. Ces résultats sont justifiés et analysés dans les paragraphes suivants.

Le tableau fournit le nombre de lignes MATLAB nécessaires pour spécifier l'ensemble des assertions comportementales de chaque application. Le nombre d'assertions par application (seconde colonne du tableau) dépend d'un choix que nous avons fait afin de mettre plus facilement en avant l'impact des moniteurs. Par exemple, dans l'application JPEG, nous

Application	# assertions	Simulation	Temps de simulation	Surcout
DCT 1D 8x8	16	Référence	0,281s	
		VHDL (sp)	0,285s	+1,49%
		VHDL (mp)	0,289s	+2,85%
DCT 2D 8x8 (faible lat.)	320	Référence	45.828s	
		VHDL (sp)	46.039s	+0,46%
		VHDL (mp)	46.906s	+2,30%
DCT 2D 8x8 (grande lat.)	320	Référence	25.516s	
		VHDL (sp)	25.664s	+0,58%
		VHDL (mp)	26.633s	+4,20%
JPEG (faible lat.)	896	Référence	183.891s	
		VHDL (sp)	184.625s	+0,40%
		VHDL (mp)	186.406s	+1,35%
JPEG (grande lat.)	896	Référence	131.125s	
		VHDL (sp)	132.109s	+0,74%
		VHDL (mp)	134.844s	+2,76%

TABLE 3 – Temps de simulation pour 300 itérations de chaque application

avons défini manuellement 2 assertions pour vérifier les bornes des 192 entrées, 2 autres pour 64 résultats intermédiaires, et enfin 2 assertions pour vérifier les 192 sorties, d'où 896 assertions au total. Le nombre de lignes de code MATLAB est parfois bien plus faible que le nombre d'assertions, car certaines assertions ont été spécifiées à l'aide de structures itératives (boucle *for*). Ces résultats mettent en avant l'intérêt d'une génération automatique des moniteurs par rapport à une écriture manuelle : dans l'application JPEG, 26 lignes de code MATLAB décrivant les assertions sont transformées en 16647 lignes VHDL.

A partir des descriptions RTL générées, nous avons mesuré l'augmentation des temps de simulation induits par les moniteurs. Nous avons effectué les simulations à l'aide de l'outil ModelSim SE 6.6 [139]. Chaque description a été simulée sur 300 itérations avec des stimuli fixes, contraints pour ne déclencher aucun moniteur. Cette contrainte est nécessaire dans ce cas d'étude car un moniteur détectant une erreur stoppe la simulation pour en informer le concepteur. Or, nous cherchons à mesurer l'impact par rapport à une simulation de référence, sans moniteurs. Le tableau 3 détaille les temps de simulation mesurés. Comme nous pouvions nous y attendre, l'utilisation d'un process VHDL par moniteur induit des pénalités plus importantes que lorsqu'un seul process est utilisé pour implémenter l'ensemble des moniteurs. Toutefois, même avec un très grand nombre d'assertions, le surcoût temporel ne dépasse pas 5% par rapport aux simulations de référence.

3 Gestionnaire d'erreurs

La seconde contribution vise la génération d'un gestionnaire d'erreurs matériel dont le rôle est d'archiver les erreurs survenant dans un circuit en fonctionnement réel, ainsi que leurs contextes d'exécutions. La détection et l'identification des erreurs repose sur des moniteurs matériels intégrés au système, générés par exemple en utilisant une approche de la littérature [51, 50].

L'approche utilisée pour analyser les implémentations matérielles du gestionnaire d'erreurs en fonction de la configuration des applications est délibérément généraliste. La méthode de synthèse du gestionnaire a été testée à partir de modèles aléatoires. Cette approche permet de ne pas se restreindre à un choix limité de cas d'études, et ainsi de pouvoir proposer une analyse statistique basée sur plusieurs milliers de modèles différents. Cette technique permet de faire apparaître des tendances générales.

Chaque modèle aléatoire est défini par :

- le nombre de variables présentes dans la description algorithmique,
- la dynamique maximale de ces variables,
- le nombre d'assertions présentes dans la description algorithmique,
- la contrainte de latence pour la génération de l'architecture RTL.

Ces paramètres sont suffisants pour permettre la synthèse d'un gestionnaire d'erreurs. En effet, il est possible de générer aléatoirement toutes les informations dont a besoin la méthode de synthèse proposée : durée de vie et dynamique des variables, registres matériels utilisés pour la mémorisation de ces variables, et enfin date d'exécution des moniteurs et contexte applicatif associé. La génération de ces informations a été faite en trois étapes :

1. **Définition des variables** - Chaque variable possède une durée de vie dans l'architecture. Cette durée de vie est générée aléatoirement tout en s'assurant qu'elle satisfasse la contrainte de latence du système. La dynamique de la variable est prise aléatoirement entre 1 et la dynamique maximale spécifiée.

2. **Définition des registres** - Des registres matériels sont créés pour mémoriser ces variables. Ils sont générés de façon à être réutilisés pour contenir différentes variables au cours du temps. Le nombre de registres obtenu est au plus égal au nombre de variables.

3. **Définition des moniteurs** - Chaque assertion définit un moniteur matériel. La date d'exécution de chaque moniteur au sein du chemin de données est générée aléa-

toirement entre 0 et la latence de l'architecture. Nous avons arbitrairement défini le contexte d'exécution de chaque moniteur comme suit : pour chaque moniteur, chaque variable de l'application a une probabilité de 0,1 de faire partie du contexte d'exécution de ce moniteur. Cela permet d'obtenir des moniteurs très différents. Certains n'ont aucune variable applicative à mémoriser, d'autres une seule et certains en mémorisent un grand nombre. La forte disparité obtenue permet de mettre plus facilement en avant les apports et inconvénients des techniques d'optimisation proposées dans la méthodologie.

Le modèle pseudo-aléatoire ainsi défini couvre entièrement les besoins de la méthode proposée pour la génération du gestionnaire d'erreurs.

Trois paramètres distincts ont été étudiés :

– l'influence de la largeur de la mémoire sur ces architectures,

– l'impact des optimisations proposées sur les architectures générées,

– l'analyse du temps nécessaire à la détermination de l'espace des solutions d'ordonnancement.

3.1 Influence de la largeur de la mémoire

La recherche d'une solution optimale passe par un parcours itératif des différentes largeurs de mémoire possibles afin de déterminer la plus intéressante quant au coût matériel. Afin d'analyser l'influence de cette largeur de mémoire sur l'architecture générée, nous avons utilisé la configuration de test suivante : contrainte de latence de 85 cycles, 20 variables de 24 bits maximum et 15 moniteurs. Après identification de son espace de solutions (toutes les largeurs de mémoire permettant l'ordonnancement des rapports), nous avons fait varier la largeur de la mémoire tout en observant les caractéristiques des architectures générées. La méthode proposée permet d'obtenir une largeur mémoire contenue dans l'intervalle $[9, 52]$ bits.

La figure 55 montre l'évolution de la taille de la mémoire, en nombre de bits, nécessaire à la mémorisation d'au moins un rapport (le plus gros). On observe une évolution par paliers. En effet, augmenter la largeur d'une unité ne change pas nécessairement les possibilités d'organisation des paquets au sein des cases. Puis, lorsque la largeur augmente suffisamment, le rapport peut utiliser une case mémoire de moins qu'à l'itération précédente, ce qui a pour effet de faire chuter d'un coup la quantité de mémoire nécessaire pour le mémoriser. On remarque qu'à partir d'une largeur de 27 bits, la taille de la mémoire ne fait qu'augmenter jusqu'à la dernière valeur de la largeur, ou elle redescend brusquement.

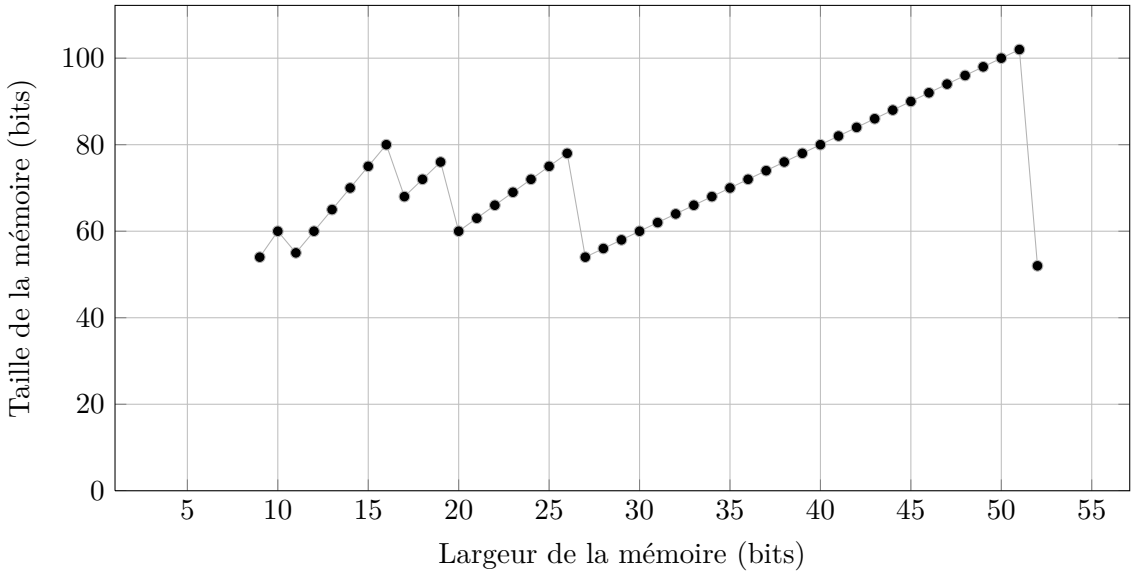

FIGURE 55 – Évolution de la taille de la mémoire, en nombre de bits, nécessaire pour permettre la mémorisation du plus gros rapport de l'application

Cela est du au fait qu'à partir de 27 bits, le plus gros rapport utilisait 2 cases mémoires. La seule façon de passer à une seule case mémoire était par définition d'utiliser la largeur maximale de la mémoire, d'où cette augmentation croissante jusqu'à l'avant dernière valeur. Ainsi, la largeur maximale de la mémoire permet toujours d'avoir une taille de mémoire minimale si l'on ne considère que le plus gros rapport. Cette largeur de mémoire produira toutefois le taux d'occupation le plus faible, comme étudié ci-après. Cette évolution par paliers se retrouve dans toutes les applications, mais le nombre de paliers et leurs intervalles d'évolution varient en fonction du nombre d'assertions et de leurs contextes d'exécution.

Ce résultat est nuancé lorsque l'on s'intéresse à l'occupation de la mémoire. La taille de la mémoire présentée était calculée *au pire cas,* c'est à dire pour le plus gros rapport. Ainsi, pour être certain de mémoriser 10 rapports, il suffit de multiplier la taille mémoire nécessaire pour le plus gros rapport obtenue par 10. Toutefois, une quantité de mémoire calculée pour contenir 10 fois le plus gros rapport peut en contenir beaucoup plus, en fonction de ceux qui sont réellement écrits au cours du temps. La figure 56 montre l'évolution de l'utilisation moyenne de la mémoire (chaque rapport est considéré comme ayant la même probabilité que les autres d'être écrit). Plus la largeur de la mémoire augmente, moins l'occupation est efficace (avec une évolution par paliers ici encore). Cela est dû au fait que moins la mémoire est large, et plus les variables sont scindées en de multiples paquets. Cela permet à l'heuristique de résolution du problème de *bin-packing* d'être plus performante. Par conséquent, il est alors plus intéressant de privilégier une largeur faible plutôt que la largeur maximale, ainsi un plus grand nombre de rapports pourront être mémorisés à coût mémoire identique. Par exemple, pour une largeur de 9 bits, l'ensemble des

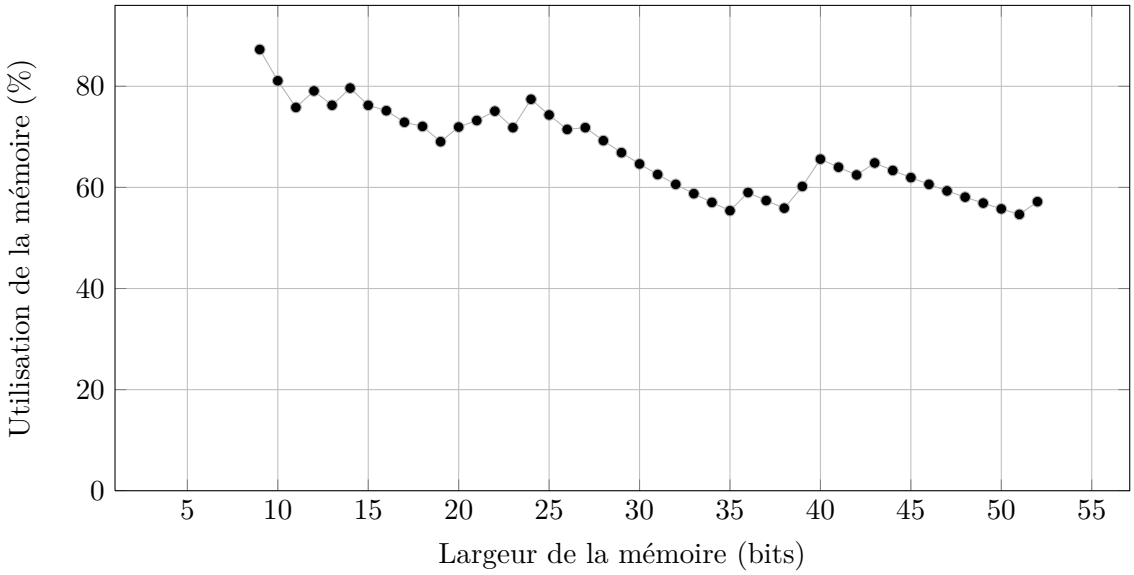

FIGURE 56 – Évolution de l'utilisation de la mémoire en fonction de la largeur. Une bonne utilisation signifie que plus de rapports peuvent être potentiellement écrits dans un même nombre de bits.

rapports ont besoin de 55 cases mémoires pour être mémorisés, soit une taille de mémoire de $9 \times 55 = 495$ bits. En utilisant la largeur maximale (52 bits), les rapports ont besoin de 15 cases mémoires (une case par rapport), soit $52 \times 15 = 780$ bits. Si le concepteur décide d'utiliser une mémoire de taille totale égale à 2048 bits, il pourra donc mémoriser l'ensemble des rapports de 4 itérations de l'application (si aucune assertion n'est vérifiée à chaque fois) avec la largeur de 9 bits, contre seulement 2,5 itérations avec la largeur maximale.

La figure 57 montre l'évolution du nombre total de cases mémoires (c'est à dire de la somme des cases de tous les rapports), ainsi que du nombre de cases mémoires uniques. Le nombre total de cases mémoires décroit lorsque la largeur de la mémoire augmente. Cela est du au fait que plus la largeur de mémoire est importante, plus il est possible de placer de données dans chaque case mémoire et donc moins il faut de cases mémoires. Toutefois, ce qui nous intéresse réellement est le nombre de cases mémoires uniques. En effet, la complexité du multiplexeur de l'architecture est proportionnelle à ce nombre, car chaque case mémoire unique correspond à une entrée distincte du multiplexeur. On peut remarquer que l'optimisation des paquets au sein de chaque case mémoire, notée O1, permet de maintenir un nombre de cases uniques presque constant, indépendamment de la largeur de mémoire. Cela permet de limiter la complexité matérielle du multiplexeur même pour de faibles largeurs, car ce sont ces largeurs qui nous intéressent du fait de leur haut taux d'utilisation de la mémoire.

En ce qui concerne le coût matériel théorique des buffers, celui-ci décroît avec la largeur de la mémoire, comme illustré dans la figure 58. En effet, plus la largeur de la mémoire

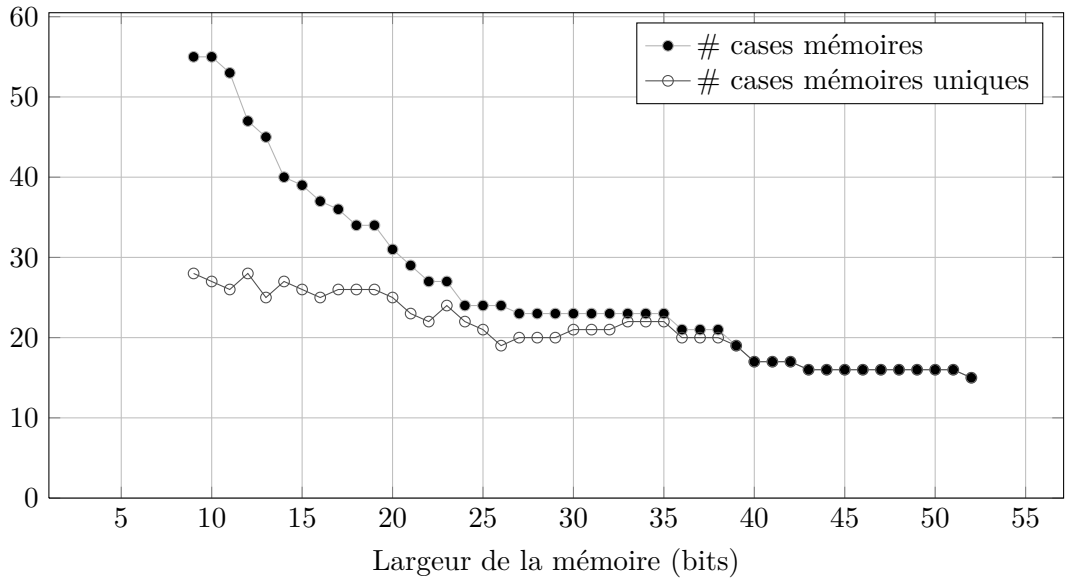

FIGURE 57 – Évolution du nombre de cases mémoires total (addition du nombre de cases de chaque rapport) et de cases uniques en fonction de la largeur

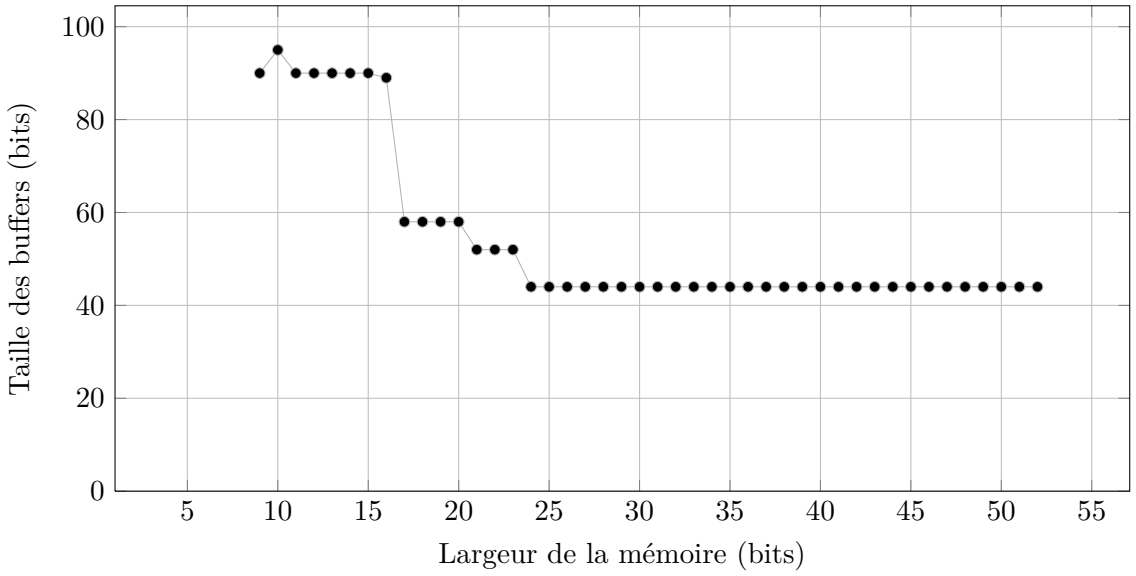

FIGURE 58 – Évolution de la taille des registres additionnels, en nombre de bits

est importante et plus le nombre de cases mémoires nécessaires pour stocker un rapport diminue. Cette diminution contribue à améliorer les performances de la phase d'optimisation des buffers, nommée O2, et donc à décroître le besoin en mémorisation temporaire des données avant leur stockage dans la mémoire.

Enfin, il est intéressant d'observer l'évolution des ressources matérielles mises en oeuvre pour implanter l'architecture matérielle du gestionnaire d'erreurs après synthèse logique. Ces ressources ont été estimées en premier lieu pour un FPGA Virtex6 de Xilinx (synthèse effectuée avec l'outil XST 12.4 M.81d). L'ensemble de l'architecture du gestionnaire

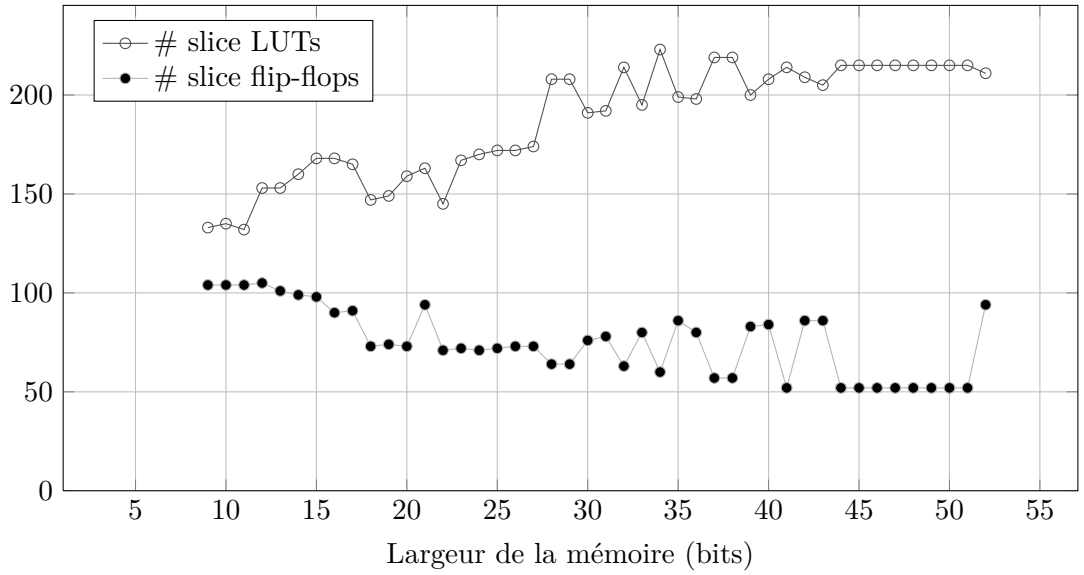

FIGURE 59 – Évolution du coût matériel du gestionnaire proposé, pour une technologie FPGA Virtex 6

(contrôleur compris), à l'exception de la mémoire, est synthétisé dans chaque cas. La figure 59 montre l'évolution du nombre de bascules et de LUTs nécessaires à l'implantation de cette architecture en fonction de la largeur de la mémoire. La quantité de bascules mise en oeuvre suit l'évolution du coût théorique des buffers. Les brusques variations ponctuelles observées sont liées aux choix d'implantation de l'outil de synthèse. En effet, en analysant les différents rapports de synthèse de chaque implantation, le nombre de bascules détectées par l'outil avant implantation ne correspond par toujours au résultat final de la synthèse. Par exemple, on remarque sur la courbe un pic d'utilisation de bascules pour une largeur de mémoire de 21 bits. Or, les rapports de synthèses indiquent la détection de 152 bascules pour la largeur de 20 bits, 151 bascules à 21 bits, et 146 bascules à 22 bits. Cela correspond à la baisse attendue. La façon dont ces ressources ont été implantées par l'outil sur le FPGA dépendent donc de choix arbitraires qu'il est difficile de prévoir. Enfin, la quantité de LUTs, quant à elle, est fortement liée aux structures de contrôle, et augmente donc avec la largeur de la mémoire. On remarque que les pics d'utilisation ponctuels les plus importants correspondent aux basses valeurs de la courbe d'utilisation des bascules, ce qui fait ressortir les choix d'implantation de l'outil.

En second lieu, la complexité de l'architecture a été évaluée sur une cible ASIC 65nm. Les figure 60 et 61 présentent le coût silicium (en μm^2) nécessaire pour intégrer l'architecture proposée, selon deux stratégies différentes pour la mémoire. Les résultats détaillent le coût engendré avec et sans la mémoire. L'évolution de la surface (sans mémoire) suit la même tendance que celle du coût des bascules pour la cible FPGA. La surface en μm^2 baisse jusqu'à son niveau optimum (largeur 21 bits), puis augmente progressivement. Lorsque l'on prend en compte le coût de la mémoire, l'évolution est différente : plusieurs paliers

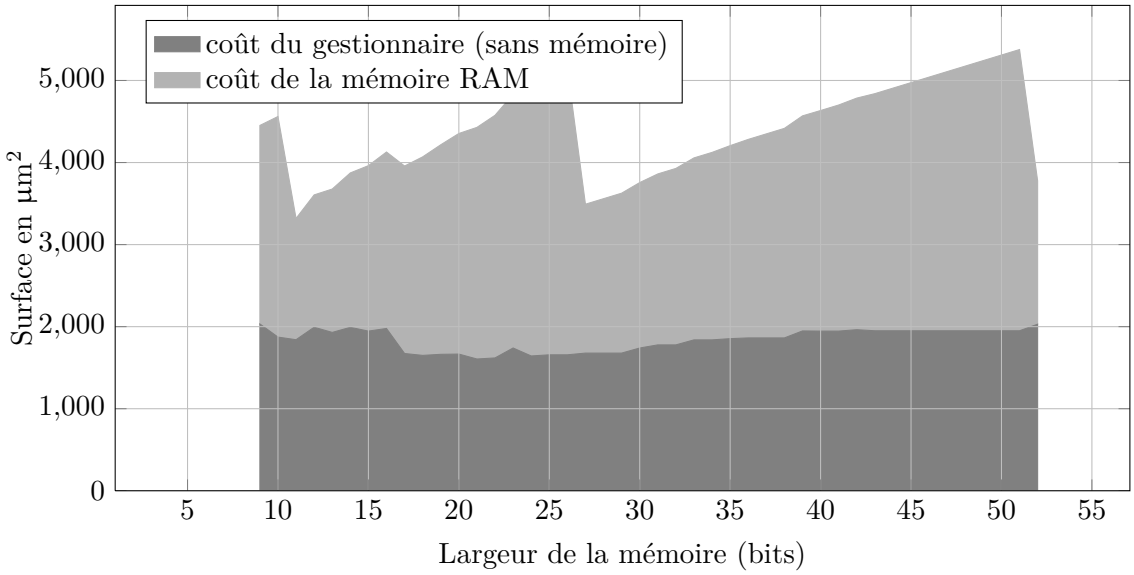

FIGURE 60 – Évolution du coût matériel du gestionnaire avec prise en considération de la mémoire, pour une technologie ASIC 65nm^2. La mémoire est calculée pour contenir 50 fois le plus gros rapport.

apparaissent. Dans notre expérience, nous avons en premier lieu fixé la taille de la mémoire de façon à ce qu'elle puisse contenir 50 fois le plus gros rapport (figure 60). En second lieu, la mémoire a été contrainte pour contenir tous les rapports de 5 itérations différentes de l'application (figure 61). Le coût en surface d'un bit de mémoire RAM simple-port en technologie ASIC 65nm est de 0,525µm^2. La profondeur minimum de la mémoire a été majorée à la puissance de 2 supérieure. Ce choix lié à des contraintes technologiques explique l'apparition des paliers dans les deux figures. Les variations observées sur la figure 60 ne sont pas les mêmes que celles de la figure 55, car cette dernière présentait aussi la taille de mémoire minimale pour contenir (une fois) le plus gros rapport, mais sans contrainte technologique.

En conclusion, la solution optimale dépend de la stratégie envisagée pour le dimensionnement de la mémoire. Dans notre exemple, si le concepteur souhaite explicitement pouvoir mémoriser 50 rapports, la largeur de mémoire de 11 bits constitue la meilleure solution pour réduire le coût matériel du circuit. Si par contre, c'est le nombre d'itérations de l'application qui importe, alors la solution optimale est à 24 bits. Un bon compromis peut être trouvé par recherche d'un minimum local dans l'espace des solutions, en itérant à partir de la largeur minimale. Dans le premier cas, le premier minimum local est à 11 bits : c'est aussi la solution optimale. Il n'aura fallu que 3 itérations pour trouver cette valeur (l'espace des solutions contient 44 valeurs possibles). Dans le second cas, le premier minimum est à 12 bits, 4 itérations sont nécessaires pour le trouver. Cette solution propose un coût matériel de 3606µm^2, contre 3256µm^2 pour la solution optimale, soit une différence de 10%. Le second minimum local donne la solution optimale et nécessite 16 itérations.

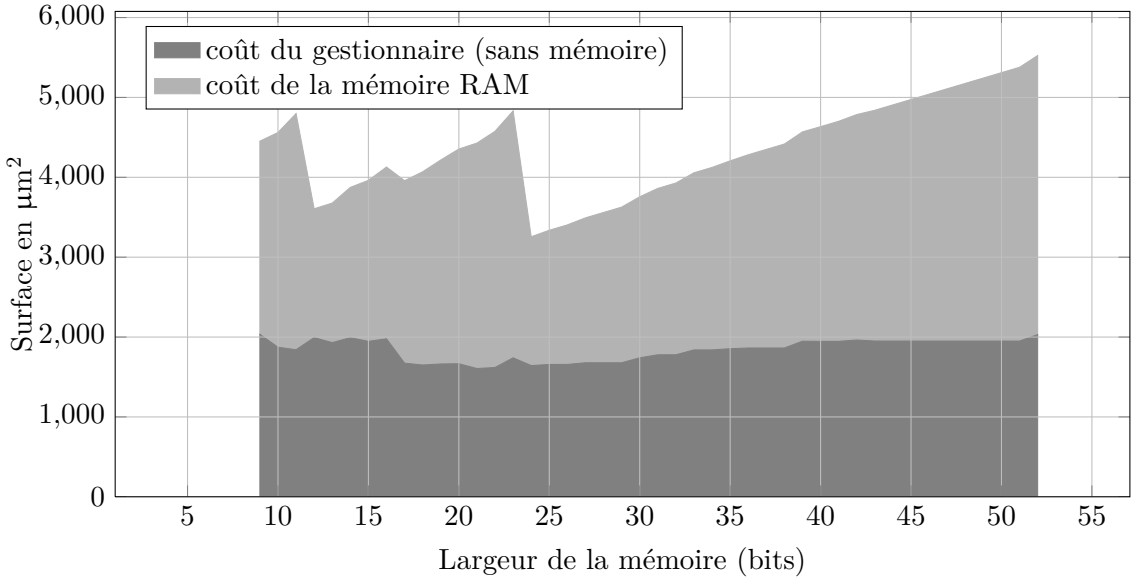

FIGURE 61 – Évolution du coût matériel du gestionnaire avec prise en considération de la mémoire, pour une technologie ASIC 65nm^2. La mémoire est calculée pour contenir tous les rapports de 5 itérations de l'application différentes (soit 75 rapports au maximum).

3.2 Détermination de l'espace des solutions d'ordonnancement

La borne maximum de l'espace des solutions, taille à partir de laquelle chaque rapport est contenu dans une unique case mémoire, est calculable directement. La borne minimum est quand à elle recherchée par itérations successives. A chaque itération, on évalue la faisabilité d'un ordonnancement par rapport à la largeur de mémoire considérée. Deux techniques ont été évaluées pour la recherche de cette borne : un parcours linéaire de la largeur de mémoire, à partir de 1, et une recherche par dichotomie entre 1 et la largeur maximale. Afin de comparer ces deux approches en terme de complexité calculatoire et de performances, deux jeux d'expériences ont été menés.

Nous avons choisi pour le premier jeu d'expériences les contraintes suivantes pour le système à concevoir : la latence de l'architecture est de 85 cycles, et 20 variables (de dynamique maximale 24 bits) sont à mémoriser dans un, plusieurs ou aucun des 15 moniteurs différents de l'architecture. Ce jeu de contraintes a été utilisé pour générer aléatoirement 1000 modèles d'architectures. Les résultats obtenus sont donnés dans la figure 62. La courbe en gris clair représente le nombre d'itérations nécessaires à chaque expérience pour trouver la borne minimum à l'aide du parcours linéaire, et la courbe en gris foncé présente la même information mais avec la recherche dichotomique. Avec les contraintes imposées, on constate que l'approche par recherche dichotomique est toujours la plus performante, avec un rapport de nombre d'itérations supérieur à 6 dans le cas d'une largeur de mémoire supérieur à 50 bits.

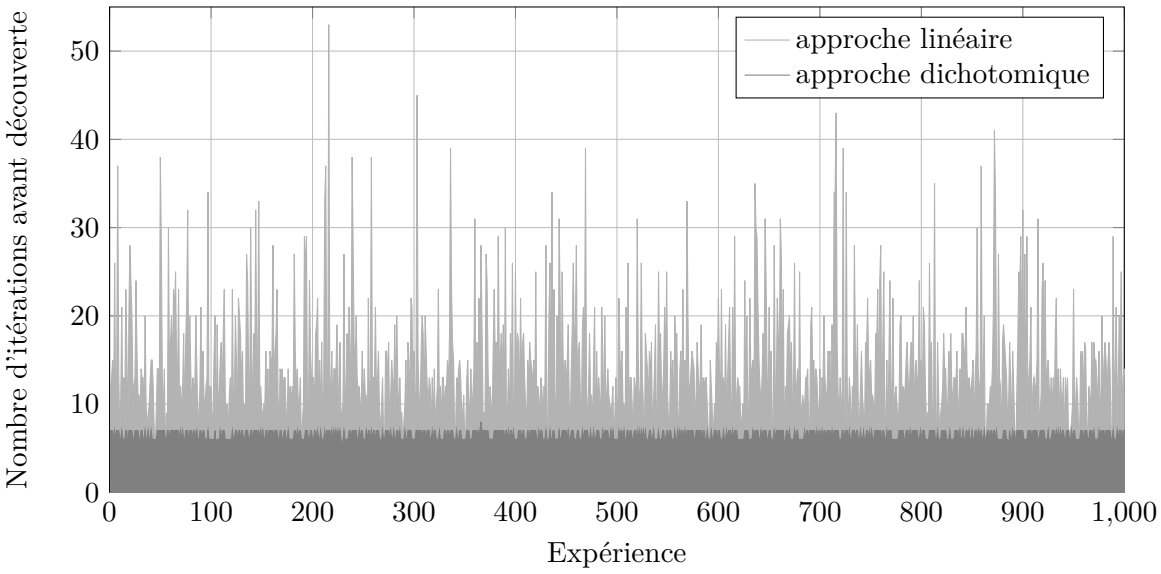

FIGURE 62 – Nombre d'itérations nécessaires avant de trouver la largeur minimum permettant un ordonnancement, selon deux approches différentes, avec une architecture contraint par une faible latence

Toutefois, la latence imposée par rapport au nombre de moniteurs nécessite dans la plupart des expériences une largeur de mémoire supérieure à 10 bits pour permettre l'ordonnancement. Ainsi, un deuxième jeu d'expériences a été mené dans les mêmes conditions mais avec cette fois une latence imposée de 200 cycles. La figure 63 présente les résultats obtenus. Cette fois-ci, la largeur minimale de la mémoire est dans la majorité des cas bien plus faible que lors des expériences précédentes. Dans ce cas de figure, la réduction de la complexité calculatoire obtenue grâce à l'approche dichotomique est moindre. Dans certains cas, l'approche dichotomique est même légèrement plus complexe que le parcours séquentiel (avec un delta maximal de 2 itérations). Cependant, les gains substantiels obtenus dans le premier jeu d'expérimentations justifient la généralisation de son utilisation.

3.3 Évaluation des optimisations architecturales sur la complexité matérielle

Deux techniques d'optimisation de l'architecture ont été proposées. La première ciblait l'amélioration de la taille et de l'utilisation de la mémoire, en fusionnant les données au sein des cases mémoires. Elle contribuait aussi à améliorer la faisabilité de l'ordonnancement des rapports, et donc la détermination d'une solution possible pour la synthèse du gestionnaire. La seconde optimisation, quant à elle, visait en particulier la réduction de la complexité matérielle des buffers de l'architecture, nécessaires pour la mémorisation temporaires de certaines données.

Nous avons évalué les performances de ces techniques sur différents paramètres :

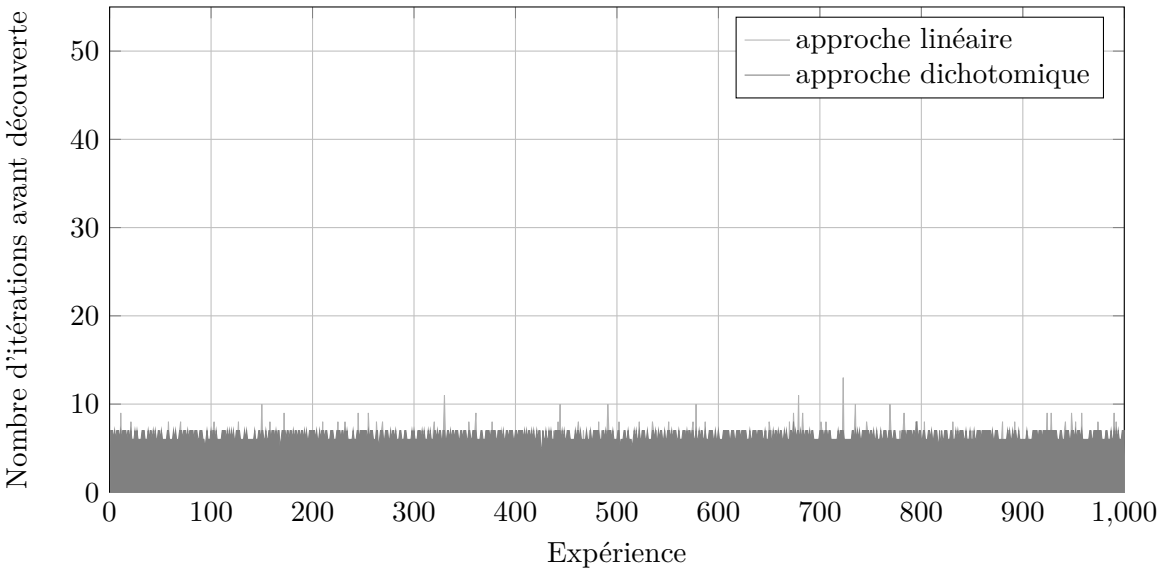

FIGURE 63 – Nombre d'itérations nécessaires avant de trouver la largeur minimum permettant un ordonnancement, selon deux approches différentes, avec une architecture contraint par une forte latence

– l'ordonnançabilité des rapports c'est à dire l'existence d'une solution),

– le coût de la mémoire nécessaire pour mémoriser au minimum 50 rapports,

– le coût de la structure matérielle (FSM, buffers, multiplexeur) nécessaire à l'implantation de l'architecture.

Pour évaluer l'impact de ces optimisations, 500 jeux de contraintes ont été générés aléatoirement. Tous les jeux de contraintes ont été générés aléatoirement à partir des mêmes paramètres, à savoir une latence d'architecture de 85 cycles, 20 variables applicatives (24 bits de dynamique maximum), et 15 moniteurs intégrés au système. Chaque jeu de contraintes a été utilisé dans le but de concevoir une architecture à l'aide des trois approches évaluées (aucune optimisation, O1, puis O2).

Parmi les 1500 expériences menées, 287 n'ont pas pu être ordonnancées à l'aide de l'approche en pile, sans optimisation (cf. tableau 4). Seules 42,6 % des expérimentations ont obtenu une solution d'ordonnancement à l'aide de cette approche. L'absence de solution est due à une violation de la latence maximale imposée à cause de la mémorisation d'un seul paquet par cycle d'horloge. Le taux de réussite des deux autres approches est quant à lui de 100%.

Les 1500 expérimentations ont permis d'identifier des tendances. Les résultats sont présentés sous forme de distributions, afin de simplifier la visualisation de l'impact des techniques choisies. Chaque expérimentation ayant une solution d'ordonnancement utilise la largeur de mémoire minimale dans l'espace des solutions. Les résultats sont divisés en deux par-

Technique d'optimisation	Nombre d'expérimentations avec solution
Approche en pile (O0)	213 (42,6%)
Optimisation O1	500 (100%)
Optimisation O2	500 (100%)

TABLE 4 – Nombre d'expérimentations (sur 500) pour lesquelles il existe au moins une solution d'ordonnancement, en fonction du niveau d'optimisation

ties :

– Une comparaison de l'approche en pile, sans optimisation, avec l'approche O1. Cette comparaison se focalise sur des critères liés à la mémoire, comme sa taille, son taux d'utilisation, et la complexité matérielle du multiplexeur.

– Une comparaison de l'approche O2, qui optimise le coût matériel des buffers, avec l'approche O1, afin de déterminer l'apport réel de O2 sur la complexité de l'architecture (taille théorique des buffers, coût en surface de silicium du contrôleur complet).

Comparaison des approches O1 et O0

La figure 64 présente l'impact de la technique d'optimisation O1 sur la taille de mémoire nécessaire pour mémoriser le plus gros rapport de chaque expérimentation. On constate que 69% des expérimentations utilisant l'approche O0 ont conduit à une taille de mémoire minimale pour contenir le plus gros rapport supérieure à 80 bits, contre seulement 36% des expérimentations optimisées avec l'approche O1. Le fort décalage vers la gauche de la distribution des résultats de O1 par rapport à ceux de O0 est caractéristique de l'influence de cette approche. Nous étions néanmoins en droit de nous y attendre. En effet, l'optimisation O1 autorise la fusion des données au sein des cases mémoires, ce qui a pour effet premier la réduction de la profondeur de la mémoire. De plus, cette optimisation facilite l'ordonnancement des rapports, et donc permet généralement l'utilisation d'une largeur de mémoire plus faible qu'avec l'approche O0. Une profondeur et une largeur moins importante ne peuvent que mener à une baisse de la taille totale.

L'évolution du taux d'utilisation de la mémoire est présenté dans la figure 65. Ce taux d'utilisation est calculé en prenant en considération l'ensemble des rapports à mémoriser lors d'une itération de l'application. Chaque rapport est considéré comme ayant la même probabilité que les autres d'être mémorisé. Le constat est net : 96% des expérimentations utilisant l'optimisation O1 présentent un taux d'utilisation de la mémoire supérieur à 70%, contre seulement 17% des expérimentations sans optimisation. De plus, on remarque qu'un tiers des expérimentations sans optimisation utilise moins de la moitié de la mémoire allouée. Par conséquent, l'optimisation O1 permet de réduire fortement la quantité de mémoire nécessaire lorsque l'on souhaite mémoriser l'ensemble des rapports d'un certain

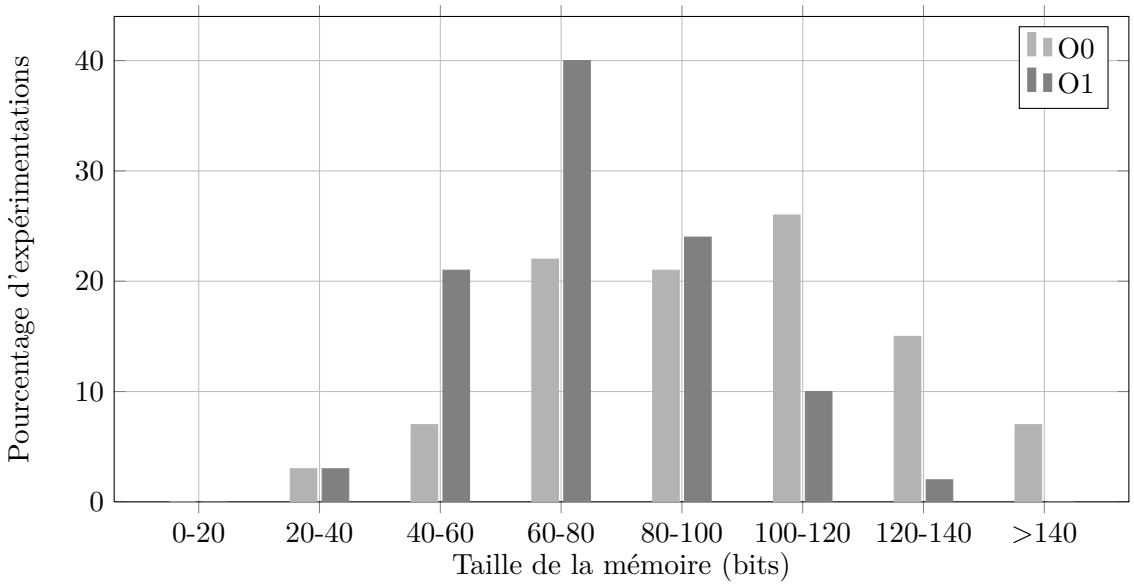

FIGURE 64 – Impact de O1 sur la taille de la mémoire nécessaire pour mémoriser le plus gros rapport

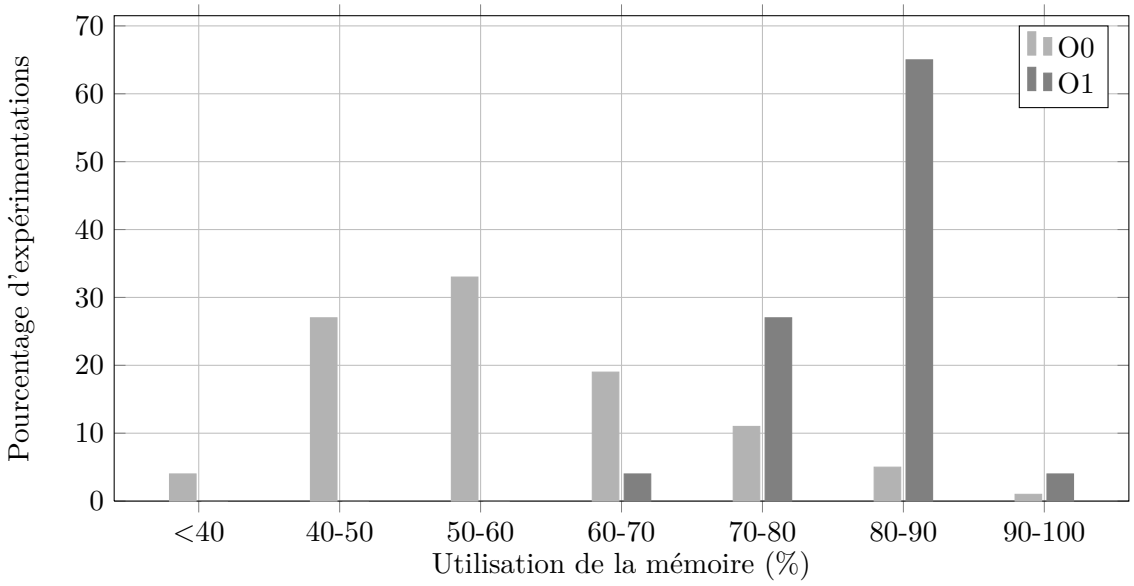

FIGURE 65 – Impact de O1 sur le pourcentage de bits utiles dans la mémoire

nombre d'itérations de l'application.

Enfin, la figure 66 montre le nombre de cases mémoires uniques de chaque expérimentation. Ce nombre prend en compte l'ensemble des rapports d'une itération de l'application. Il définit la complexité matérielle théorique du multiplexeur de l'architecture. Cette fois encore, l'avantage est à l'approche par optimisation O1, qui présente 30% d'expérimentations avec moins de 25 cases uniques, contre 10% pour l'approche O0. La distribution des résultats de O1 est sensiblement décalée sur la gauche par rapport aux résultats de O0. Cela caractérise une tendance à la réduction du nombre de cases mémoires uniques lorsque

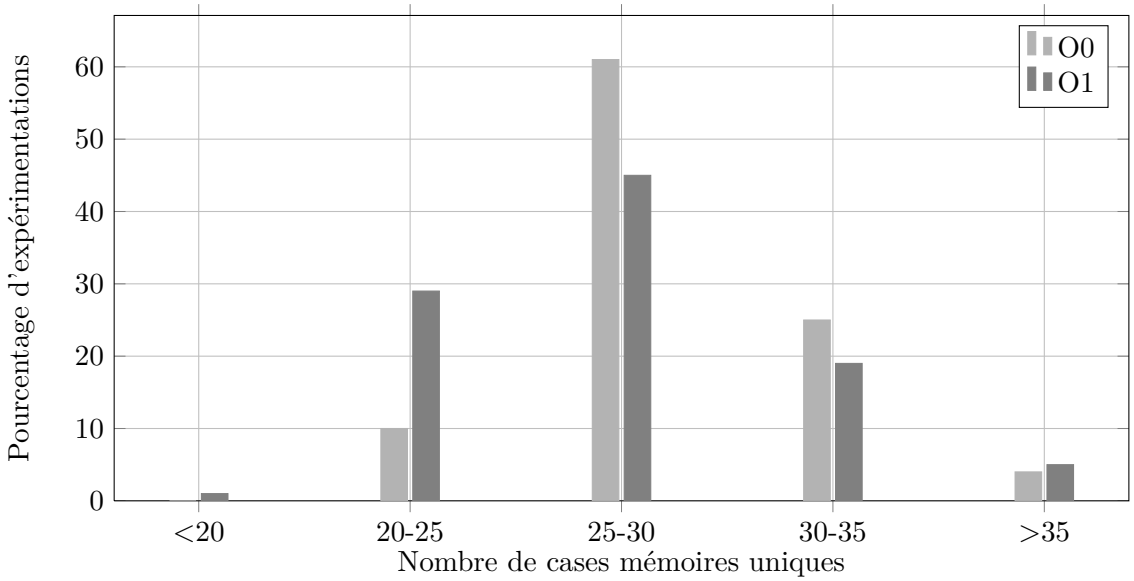

FIGURE 66 – Impact de O1 sur le nombre de cases mémoires uniques

O1 est employée. Ce résultat dépend toutefois du rapport entre le nombre de variables à mémoriser et le nombre de moniteurs. En effet, dans le cas de O0, le nombre de cases mémoires uniques ne dépend que du nombre de variables : une variable (ou du moins un paquet la composant) est mémorisée par case mémoire. Avec O1, plusieurs variables (plusieurs paquets) peuvent être fusionnées au sein de la même case mémoire, ce qui augmente le nombre de combinaisons possibles. La méthode O1 tient toutefois compte de cela et essaie de minimiser ces combinaisons, en modifiant l'ordre des paquets dans chaque case mémoire à posteriori notamment.

Comparaison des approches O2 et O1

Concernant l'apport de l'approche O2 par rapport à l'approche O1, l'étude s'est focalisée en premier lieu sur le nombre de bits nécessaires pour les registres additionnels (buffers) de l'architecture, car l'optimisation O2 est dédiée à la diminution de ce nombre. L'impact de cette optimisation est présenté sur la figure 67. On constate que 56% des expérimentations avec O2 ont nécessité moins de 100 bits pour l'ensemble des buffers, contre 42% des expérimentations O1. Une analyse supplémentaire a montré que dans la totalité des cas, le nombre de bits a baissé ou est resté stable entre deux expérimentations O1 et O2 utilisant la même configuration initiale, ce qui est le résultat attendu. Le décalage vers la gauche de la distribution des résultats des expérimentations avec O2 montre l'apport de cette approche. Cet apport varie toutefois avec le nombre de variables à mémoriser temporairement et la répartition des durées de vie de ces variables. Moins les durées de vie des données se recouvrent, et plus le nombre de buffers peut être réduit. L'efficacité de cette optimisation est donc fortement lié à l'application considérée, mais cette approche

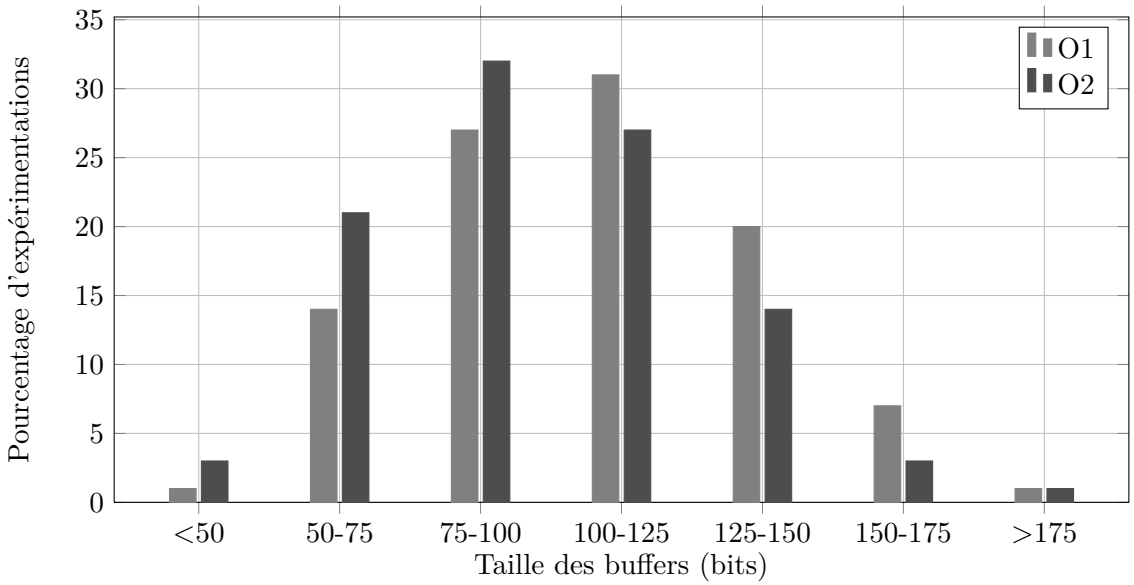

FIGURE 67 – Impact de O2 sur le nombre de bits nécessaires pour les registres additionnels

n'entraîne toutefois jamais de pénalité au niveau de la taille des buffers.

Enfin, l'impact de l'optimisation O2 par rapport à O1 a aussi été mesuré en terme de coût matériel, en synthétisant les architectures obtenues pour une technologie ASIC 65 nm^2. La figure 68 présente la distribution des résultats en fonction de la surface de silicium nécessaire. On constate que la distribution des résultats de O2 est légèrement décalée vers les plus faibles surfaces par rapport à celle des résultats de O1. Les architectures ont toutes été générées deux à deux à partir des mêmes configurations initiales, et pour une même largeur de mémoire (O2 n'influe pas sur cette largeur). De ce fait, il est clair que l'optimisation O2 a permis une réduction du coût matériel par rapport à O1 dans les cas où cela était applicable. Le gain moyen est de 64μm^2, avec un écart-type de 80,6μm^2. Ici encore, l'efficacité de la réduction de surface dépend de l'application considérée et de la répartition des durées de vie des variables à mémoriser, mais cette efficacité n'est jamais négative.

4 Conclusion

La première méthodologie proposée cible la génération de moniteurs RTL dédiés à la simulation. L'introduction de ces moniteurs dans les architectures générées par synthèse HLS n'influe que faiblement (<5%) sur le temps de simulation des architectures. La quantité de code VHDL générée est toutefois suffisamment importante pour ne pas laisser de doute quant à l'intérêt de privilégier une génération automatique des moniteurs par rapport à une description manuelle. L'inclusion des moniteurs dans les architectures générées ne mo-

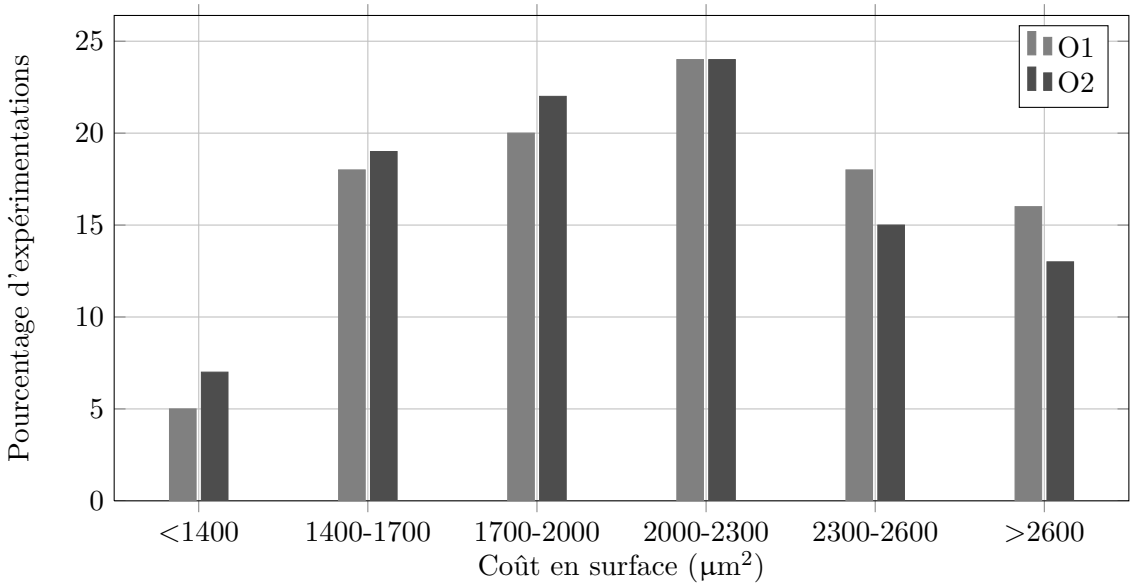

FIGURE 68 – Impact de O2 sur le coût en surface du gestionnaire d'erreurs (hors mémoire)

difie pas les résultats des synthèses logiques de ces architectures. Ceci est toutefois très différent pour la seconde méthodologie.

Le gestionnaire d'erreurs synthétisé a un coût matériel non négligeable. Ce coût est divisé en deux parties : d'une part, il est dû aux registres additionnels, au multiplexeur d'entrée de la mémoire et à la logique de contrôle, et d'autre part, il est constitué du coût de la mémoire. Cette dernière partie du coût varie fortement en fonction du nombre de rapports que l'on souhaite mémoriser, ou du nombre d'itérations différentes que l'on souhaite monitorer. En fonction de la configuration de l'application, l'impact de la mémoire peut être tel que les optimisations proposées pour réduire le coût des composants du gestionnaire ne permettent qu'un gain relativement faible par rapport au coût de la mémoire. Toutefois, les deux optimisations proposées ont dans chaque cas un impact positif sur le système. Ainsi, la recherche de la solution optimale peut se faire en conservant ces optimisations, et en agissant seulement sur la largeur de la mémoire.

Chapitre 6

Conclusion et perspectives

Nous avons présenté dans ce mémoire deux contributions ciblant l'amélioration du processus de vérification par assertions dans les systèmes matériels.

La première contribution partait d'un constat simple : les assertions sont utilisées à tous les niveaux du flot de conception des circuits (modèle comportemental, modèles TLM, description RTL, circuit matériel). Il existe dans la littérature des méthodes pour générer automatiquement des assertions au niveau RTL à partir d'autres niveaux plus abstraits. Toutefois, rien n'avait été proposé, à notre connaissance, pour passer du niveau comportemental au niveau RTL tout en conservant le formalisme des assertions. Or, le raffinement automatique du niveau comportemental au niveau RTL est un processus bien connu sous le nom de synthèse de haut-niveau. L'utilisation de ce processus dans un flot de conception de circuit matériel prend de plus en plus d'importance à mesure que les outils existants gagnent en maturité.

Ainsi, nous avons d'abord proposé une méthode pour ajouter le support des assertions à ces outils, et faciliter ainsi la vérification des architectures générées et leur intégration dans un système. Pour ce faire, la méthode proposée identifie les assertions dans la représentation abstraite de la description algorithmique après compilation par un outil de HLS, et les détache de cette représentation afin de laisser l'outil travailler sur un modèle sans assertions. Les relations entre les données algorithmiques et les ressources matérielles de l'architecture générée sont analysées et reportées sur les modèles des assertions. Enfin, les assertions temporelles sont générées en langage PSL ou VHDL. Ces assertions RTL proposent ainsi les mêmes fonctionnalités que les assertions booléennes de la description algorithmique, tout en s'accommodant de la localisation et de la durée de vie des variables dans la description RTL. Cette transformation automatique évite au concepteur d'avoir à réécrire les assertions RTL à chaque changement de la description algorithmique.

En second lieu, nous avons proposé une approche pour la synthèse automatique d'un gestionnaire d'erreurs matériel et son inclusion au sein d'un système. La simulation des systèmes matériels permet l'identification et l'analyse des erreurs présentes dans ces systèmes. Les concepteurs disposent de l'ensemble des chronogrammes des signaux du système pendant sa simulation, et peuvent ainsi remonter à la source d'une erreur en analysant les données qui sont à l'origine de toute erreur identifiée. Toutefois, la complexité actuelle des systèmes matériels (SoC) et les performances limitées des processeurs généralistes employés par les outils de simulation poussent les concepteurs à privilégier la vérification directement en fonctionnement ou en émulation sur FPGA. Malheureusement, le gain en rapidité d'exécution des applications est contrebalancé par la difficulté de sonder le circuit et de détecter les erreurs au moment où elles se produisent.

Plusieurs travaux sont décrits dans la littérature pour permettre la transformation d'assertions d'une description comportementale en moniteurs matériels détectant les erreurs pendant le fonctionnement réel du circuit. Toutefois, ces moniteurs ne permettent que d'identifier les erreurs, et la seule connaissance de leur existence ne suffit pas toujours à en comprendre la cause. Ainsi, le rôle du gestionnaire d'erreurs automatiquement généré à l'aide de l'approche que nous proposons est complémentaire à ces moniteurs. Il sauvegarde le contexte d'exécution courant de l'application au moment où une erreur est détectée. Ce contexte d'exécution est constitué de l'identifiant de l'assertion, mais aussi des valeurs de toute variable spécifiée par le concepteur dans la description algorithmique de l'application. L'approche proposée s'inscrit dans la démarche d'offrir plus de confort aux concepteurs en facilitant la compréhension des erreurs. La définition des contextes d'exécution étant spécifiée dans les modèles comportementaux, cela permet la création de rapports d'erreurs directement en lien avec les données de ces modèles, et évite au concepteur de devoir inspecter la description RTL générée par la synthèse HLS.

Perspectives

Le gestionnaire d'erreurs présenté sauvegarde les rapports d'erreurs dans une mémoire. En utilisation *online*, le système peut lire en temps réel les rapports en accédant directement à cette mémoire. Ainsi, nous pouvons envisager d'étendre le traitement effectué par le système à une réelle prise de décision en fonction des erreurs détectées.

Par exemple, dans le domaine de la sécurité et de la fiabilité des composants, une technique comme la *triple redondance modulaire* permet de détecter certaines erreurs au niveau des opérateurs en effectuant trois fois les calculs avec trois opérateurs matériels différents. Dans le cas où une erreur est identifiée (au moins deux opérateurs donnent un résultat différent), le système peut alors décider de s'éteindre pour faute majeure, de redémarrer ou bien de suspendre la tache en cours et continuer l'exécution. La décision se fait en

fonction de la gravité de l'erreur.

Les moniteurs matériels générés à partir d'assertions agissent dans ce sens, et fournissent le même type d'information au système. Avec le gestionnaire d'erreurs proposé, nous pouvons imaginer une gestion plus fine des erreurs rencontrées. En effet, puisqu'il est possible d'inclure les valeurs de n'importe quelle variable du système, un même moniteur matériel peut déclencher différentes actions selon la valeur des variables en cause dans l'erreur, ou même en fonction de variables n'intervenant pas dans l'erreur mais précisant la configuration de l'application par exemple.

Cette gestion plus fine des erreurs peut déjà être effectuée à partir des moniteurs matériels issus des travaux existants de la littérature. Toutefois, il est alors nécessaire de spécifier une assertion pour chaque traitement particulier d'une même erreur, là où une seule et unique assertion suffit avec notre méthode. Cela a la mauvaise conséquence de lier sémantiquement la description algorithmique contenant les assertions, et la description du composant effectuant les traitements en fonction des erreurs. En effet, l'indépendance entre les deux composants est perdue car toute modification du traitement des erreurs doit être reportée dans la façon dont sont spécifiées les assertions de l'application. Or, l'indépendance entre les composants est le point le plus important pour permettre une bonne maintenabilité du système et ainsi accroître la productivité des concepteurs. Le gestionnaire d'erreurs proposé pourrait donc être utilisé dans ce sens, en permettant un transfert simple et efficace de n'importe quelle donnée de l'application vers le reste du système pour y être analysée et déclencher une action en fonction de cette analyse.

Le gestionnaire d'erreurs présenté peut aussi être utilisé pour émuler la fonction `printf` du langage C que l'on retrouve dans tous les langages de programmation logicielle sous une syntaxe similaire. Cette fonction permet d'écrire un message dans le terminal de l'utilisateur pendant l'exécution d'un programme. Elle est très utilisée par les concepteurs pour connaître la valeur d'une variable à un moment précis de l'exécution du programme, en complément des systèmes de debug permettant l'inspection pas à pas de l'exécution d'un programme.

Ainsi, cette facilité d'inspection des données du programme peut être émulée par notre gestionnaire d'erreurs. En effet, la définition du contexte d'exécution de chaque assertion permet de spécifier toute variable dont on souhaite enregistrer l'état. Ainsi, en permettant la spécification de ce contexte d'exécution d'une façon différente (`printf(a, b, c)` par exemple), nous obtenons en l'état actuel des choses la mémorisation des valeurs de ces variables dans la mémoire au moment souhaité. En utilisant un canal de communication de type FIFO englobant cette mémoire, un logiciel exécuté dans un ordinateur externe et relié au système en cours de vérification peut observer cette FIFO en temps réel. De ce fait, toute écriture d'un rapport dans la mémoire déclenchera la lecture de ce rapport par le logiciel externe et l'affichage des données au concepteur, sous une forme simple et claire.

Chapitre 7

Bibliographie

[1] Y. Abarbanel, I. Beer, L. Gluhovsky, and S. Keidar. FoCs : automatic generation of simulation checkers from formal specifications. In *Proc. 12th International Conference on computer aided verification, CAV 2000*, pages 538–542, 2000.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers : Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2 edition, 2006.

[3] Altera. SignalTap II Embedded Logic Analyzer. In *AV TOOL 2006*, 2006.

[4] R. J. Anderson and M. G. Kuhn. Low Cost Attacks on Tamper Resistant Devices. In *Proceedings of the 5th International Workshop on Security Protocols*, pages 125–136, London, UK, UK, 1998. Springer-Verlag.

[5] W. M. Arden. The International Technology Roadmap for Semiconductors - Perspectives and challenges for the next 15 years. *Current Opinion in Solid State and Materials Science*, 6(5) :371–377, 2002.

[6] S. Attaway. *Matlab, Second Edition : A Practical Introduction to Programming and Problem Solving*. Butterworth-Heinemann, 2011.

[7] P. Audebaud and C. Paulin-Mohring, editors. *Mathematics of Program Construction*, volume 5133 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[8] R. Axelsson, M. Hague, S. Kreutzer, M. Lange, and M. Latte. Extended Computation Tree Logic. In C. Fermüller and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 6397 of *Lecture Notes in Computer Science*, pages 67–81. Springer Berlin / Heidelberg, 2010.

[9] A. Baganne, J.-L. Philippe, and E. Martin. A formal technique for hardware interface design. *IEEE Transactions on Circuits and Systems II : Analog and Digital Signal Processing*, 45(5) :584–591, May 1998.

[10] M. Bartley and V. Solutions. Lies , Damned Lies and Hardware Verification. In *SNUG 2008*, 2008.

[11] B. Baudry, Y. Le Traon, and J.-M. Jezequel. Robustness and diagnosability of OO systems designed by contracts. In *Proceedings Seventh International Software Metrics Symposium*, pages 272–284. IEEE Comput. Soc, 2001.

[12] C. Bays. A comparison of next-fit, first-fit, and best-fit. *Communications of the ACM*, 20(3) :191–192, 1977.

[13] I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh. The Temporal Logic Sugar. In G. Berry, H. Comon, and A. Finkel, editors, *Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 363–367. Springer Berlin / Heidelberg, 2001.

[14] I. Beer, S. Ben-David, C. Eisner, and A. Landver. RuleBase : an industry-oriented formal verification tool. In *33rd Design Automation Conference Proceedings, 1996*, pages 655–660. ACM, 1996.

[15] I. Beer, S. Ben-David, and A. Landver. On-the-fly model checking of RCTL formulas. In A. Hu and M. Vardi, editors, *Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 184–194. Springer Berlin / Heidelberg, 1998.

[16] S. Ben-David, C. Eisner, D. Geist, and Y. Wolfsthal. Model Checking at IBM. *Formal Methods in System Design*, 22(2) :101–108, Mar. 2003.

[17] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen. Systems and Software Verification : Model-Checking Techniques and Tools. Dec. 2010.

[18] J. Bergeron. *Writing Testbenches : Functional Verification of HDL Models, Second Edition*. Springer, 2003.

[19] J. Bhadra, M. S. Abadir, L.-C. Wang, and S. Ray. A Survey of Hybrid Techniques for Functional Verification. *IEEE Design and Test of Computers*, 24 :112–122, 2007.

[20] R. Bhakthavatchalu, G. R. Deepthy, S. Vidhya, and V. Nisha. Design and analysis of low power open core protocol compliant interface using VHDL. In *2011 International Conference on Emerging Trends in Electrical and Computer Technology*, pages 621–625. IEEE, Mar. 2011.

[21] M. Bolstad. Design by Contract : A Simple Technique for Improving the Quality of Software. In *2004 Users Group Conference (DOD_UGC'04)*, pages 319–323. IEEE, 2004.

[22] D. Borrione, K. Morin-Allory, P. Ostier, and L. Fesquet. On-Line Assertion-Based Verification with Proven Correct Monitors. In *2005 International Conference on Information and Communication Technology*, pages 123–143. IEEE, 2005.

[23] M. Boulé. *Assertion-Checker Synthesis for Hardware Verification, In-Circuit Debugging and On-Line Monitoring*. PhD thesis, McGill University, Montréal, 2008.

[24] M. Boulé, J.-S. Chenard, and Z. Zilic. Adding Debug Enhancements to Assertion Checkers for Hardware Emulation and Silicon Debug. In *2006 International Conference on Computer Design*, pages 294–299. IEEE, Oct. 2006.

[25] M. Boule, J.-S. Chenard, and Z. Zilic. Assertion Checkers in Verification, Silicon Debug and In-Field Diagnosis. In *8th International Symposium on Quality Electronic Design (ISQED'07)*, pages 613–620. IEEE, Mar. 2007.

[26] M. Boulé, J.-S. Chenard, and Z. Zilic. Assertion Checkers in Verification, Silicon Debug and In-Field Diagnosis. In *8th International Symposium on Quality Electronic Design (ISQED'07)*, pages 613–620. IEEE, Mar. 2007.

[27] M. Boulé, J.-S. Chenard, and Z. Zilic. Debug enhancements in assertion-checker generation. *IET Computers & Digital Techniques*, 1(6) :669, 2007.

[28] M. Boulé and Z. Zilic. Incorporating efficient assertion checkers into hardware emulation. In *2005 International Conference on Computer Design*, pages 221–228. IEEE Comput. Soc, 2005.

[29] M. Boulé and Z. Zilic. Efficient Automata-Based Assertion-Checker Synthesis of PSL Properties. In *2006 IEEE International High Level Design Validation and Test Workshop*, pages 69–76. IEEE, Nov. 2006.

[30] M. Boulé and Z. Zilic. Efficient Automata-Based Assertion-Checker Synthesis of SEREs for Hardware Emulation. In *2007 Asia and South Pacific Design Automation Conference*, pages 324–329. IEEE, Jan. 2007.

[31] M. Boulé and Z. Zilic. Automata-based assertion-checker synthesis of PSL properties. *ACM Transactions on Design Automation of Electronic Systems*, 13(1) :1–21, Jan. 2008.

[32] D. Bustan, D. Korchemny, E. Seligman, and J. Yang. SystemVerilog Assertions : Past, Present, and Future SVA Standardization Experience. *IEEE Design & Test of Computers*, 29(2) :23–31, Apr. 2012.

[33] C. Carreras, J. A. López, and O. Nieto-Taladriz. Bit-Width Selection for Data-Path Implementations. In *Proceedings of the 12th international symposium on System synthesis*, ISSS '99, pages 114—-119, Washington, DC, USA, 1999. IEEE Computer Society.

[34] E. Casseau and B. Le Gal. High-level synthesis for the design of FPGA-based signal processing systems. In *2009 International Symposium on Systems, Architectures, Modeling, and Simulation*, pages 25–32. IEEE, July 2009.

[35] E. Casseau and B. Le Gal. Design of multi-mode application-specific cores based on high-level synthesis. *Integration*, 45(1) :9–21, 2012.

[36] C. Chavet, P. Coussy, P. Urard, and E. Martin. A design methodology for space-time adapter. In *Proceedings of the 17th great lakes symposium on Great lakes symposium on VLSI - GLSVLSI '07*, page 347, New York, New York, USA, Mar. 2007. ACM Press.

[37] C. Chavet, P. Coussy, P. Urard, and E. Martin. Design space exploration tool for Space-Time AdapteRs. In *Workshop The new wave of High Level Synthesis, in Design, Automation and Test in Europe (DATE)*, Munich, Allemagne, 2008.

[38] C. Chekuri and S. Khanna. On multi-dimensional packing problems. In *Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '99, pages 185–194, Philadelphia, PA, USA, 1999. Society for Industrial and Applied Mathematics.

[39] W. Chih-Wea, L. Chi-Shao, W. Chi-Feng, H. Shih-Arn, and L. Ying-Hsi. On-chip interconnection design and SoC integration with OCP. In *2008 IEEE International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, pages 25–28. IEEE, Apr. 2008.

[40] E. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.

[41] E. G. Coffman Jr., M. R. Garey, and D. S. Johnson. Approximation algorithms for bin packing : a survey. In D. S. Hochbaum, editor, *Approximation algorithms for NP-hard problems*, chapter Approximat, pages 46–93. PWS Publishing Co., Boston, MA, USA, 1997.

[42] O. Cohen, M. Gordon, M. Lifshits, A. Nadel, and V. Ryvchin. Designers work less with quality formal equivalence checking. In *Proceedings of the Design and Verification Conference and Exhibition (DVCon '10)*, 2010.

[43] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-Level Synthesis for FPGAs : From Prototyping to Deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4) :473–491, Apr. 2011.

[44] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-Level Synthesis for FPGAs : From Prototyping to Deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4) :473–491, Apr. 2011.

[45] P. Coussy, E. Casseau, P. Bomel, A. Baganne, and E. Martin. A formal method for hardware IP design and integration under I/O and timing constraints. *ACM Transactions on Embedded Computing Systems*, 5(1) :29–53, Feb. 2006.

[46] P. Coussy, C. Chavet, P. Bomel, D. Heller, E. Senn, and E. Martin. \partial GAUT : A High-Level Synthesis Tool for DSP applications. In P. C. . A. Morawiec, editor, *High-Level Synthesis : From Algorithm to Digital Circuits*, pages 147–170. Springer, 2008.

[47] P. Coussy, D. Gajski, M. Meredith, and A. Takach. An Introduction to High-Level Synthesis. *IEEE Design & Test of Computers*, 26(4) :8–17, July 2009.

[48] P. Coussy and A. Morawiec. *High-Level Synthesis : from Algorithm to Digital Circuit*. Springer Publishing Company, Incorporated, 1st edition, 2008.

[49] P. Coussy and A. Takach. Special Issue on High-Level Synthesis. *IEEE Design and Test of Computers*, 26(4), 2009.

[50] J. Curreri, G. Stitt, and A. D. George. High-level synthesis techniques for in-circuit assertion-based verification. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8, 2010.

[51] J. Curreri, G. Stitt, and A. D. George. High-Level Synthesis of In-Circuit Assertions for Verification, Debugging, and Timing Analysis. *International Journal of Reconfigurable Computing*, 2011 :1–17, 2011.

[52] A. Dahan, D. Geist, L. Gluhovsky, D. Pidan, G. Shapir, Y. Wolfsthal, L. Benalycherif, R. Kamdem, and Y. Lahbib. Combining System Level Modeling with Assertion Based Verification. *Quality Electronic Design, International Symposium on*, 0 :310–315, 2005.

[53] S. Das, R. Mohanty, P. Dasgupta, and P. Chakrabarti. Synthesis of System Verilog Assertions. In *Proceedings of the Design Automation &amp ; Test in Europe Conference*, pages 1–6. IEEE, 2006.

[54] G. De Michell and R. Gupta. Hardware / software co-design. *Proceedings of the IEEE*, 85(3) :349–365, Mar. 1997.

[55] D. Densmore and R. Passerone. A Platform-Based Taxonomy for ESL Design. *IEEE Design & Test of Computers*, 23(5) :359–374, May 2006.

[56] R. Drechsler and G. Fey. Formal verification meets robustness checking - Techniques and challenges. In *13th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems*, pages 4–4. IEEE, 2010.

[57] J. P. Elliott. *Understanding Behavioral Synthesis : A Practical Guide to High-Level Design.* Kluwer Academic Publishers, Norwell, MA, USA, 1999.

[58] R. Ernst. Codesign of embedded systems : status and trends. *IEEE Design & Test of Computers*, 15(2) :45–54, 1998.

[59] L. Fix. Fifteen Years of Formal Property Verification in Intel. In O. Grumberg and H. Veith, editors, *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 139–144. Springer Berlin / Heidelberg, 2008.

[60] R. W. Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19(19-32) :1, 1967.

[61] H. Foster. *Applied Assertion-Based Verification : An Industry Perspective.* now publishers Inc.

[62] H. Foster. Assertion-Based Verification : Industry Myths to Realities (Invited Tutorial). In A. Gupta and S. Malik, editors, *Computer Aided Verification*, volume 5123 of *Lecture Notes in Computer Science*, pages 5–10. Springer Berlin / Heidelberg, 2008.

[63] H. Foster, D. Lacey, and A. Krolnik. *Assertion-Based Design.* Kluwer Academic Publishers, Norwell, MA, USA, 2 edition, 2003.

[64] D. Gajski, A.-H. Wu, V. Chaiyakul, S. Mori, T. Nukiyama, and P. Bricaud. Essential issues for IP reuse. In *Proceedings 2000. Design Automation Conference. (IEEE Cat. No.00CH37106)*, pages 37–42. IEEE, 2000.

[65] B. L. Gal and E. Casseau. Latency-Sensitive High-Level Synthesis for Multiple Word-Length DSP Design. *EURASIP J. Adv. Sig. Proc.*, 2011, 2011.

[66] G. Galambos and G. J. Woeginger. On-line bin packing - A restricted survey. *Mathematical Methods of Operations Research*, 42(1) :25–45, 1995.

[67] M. K. Ganai, A. Gupta, A. Mukaiyama, and K. Wakabayashi. Another Dimension to High Level Synthesis : Verification. In *Workshop on Designing Correct Circuits*, 2006.

[68] A. Gharehbaghi, B. Yaran, S. Hessabi, and M. Goudarzi. An assertion-based verification methodology for system-level design. *Computers & Electrical Engineering*, 33(4) :269–284, July 2007.

[69] F. Ghenassia. *Transaction-Level Modeling with SystemC*. Springer, 2005.

[70] J. Gierak. Focused ion beam technology and ultimate applications. *Semiconductor Science and Technology*, 24(4), 2009.

[71] E. Girczyc and S. Carlson. Increasing design quality and engineering productivity through design reuse. In *Proceedings of the 30th international on Design automation conference - DAC '93*, pages 48–53, New York, New York, USA, July 1993. ACM Press.

[72] M. Glasser. *Open Verification Methodology Cookbook*. Springer, 2009.

[73] K. Goshi and P. Wray. An intelligent tutoring system for teaching and learning Hoare logic. In *Proceedings Fourth International Conference on Computational Intelligence and Multimedia Applications. ICCIMA 2001*, pages 293–297. IEEE, 2001.

[74] A. Gupta, A. Casavant, P. Ashar, X. Liu, A. Mukaiyama, and K. Wakabayashi. Property-specific testbench generation for guided simulation. In *Proceedings of ASP-DAC/VLSI Design 2002. 7th Asia and South Pacific Design Automation Conference and 15h International Conference on VLSI Design*, pages 524–531. IEEE Comput. Soc, 2002.

[75] K. Hemmert, J. Tripp, B. Hutchings, and P. Jackson. Source level debugger for the Sea Cucumber synthesizing compiler. In *11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2003. FCCM 2003.*, pages 228–237. IEEE Comput. Soc, 2003.

[76] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10) :576–580, 1969.

[77] D. Hommais, F. Pétrot, and I. Augé. A practical tool box for system level communication synthesis. In *Proceedings of the ninth international symposium on Hardware/software codesign - CODES '01*, pages 48–53, New York, New York, USA, Apr. 2001. ACM Press.

[78] IEEE. 1364-2005 - IEEE Approved Draft Standard for Verilog - Hardware Description Language. 2007.

[79] IEEE. 1800-2007 - IEC Standard for Systemverilog - Unified Hardware Design, Specification, and Verification Language. 2007.

[80] IEEE. 1076-2008 - IEEE Standard VHDL Language Reference Manual. 2009.

[81] IEEE. 1666-2011 - IEEE Standard for Standard SystemC Language Reference Manual. 2012.

[82] IEEE. 1850-2010 - Standard for Property Specification Language (PSL). 2012.

[83] H. Iwai. Roadmap for 22nm and beyond. *Microelectronic Engineering*, 86(7–9) :1520–1528, 2009.

[84] M. Jacome and H. Peixoto. A survey of digital design reuse. *IEEE Design & Test of Computers*, 18(3) :98–107, 2001.

[85] M. Jain, M. Balakrishnan, and A. Kumar. ASIP design methodologies : survey and issues. In *VLSI Design 2001. Fourteenth International Conference on VLSI Design*, pages 76–81. IEEE Comput. Soc, 2001.

[86] J.-M. Jazequel and B. Meyer. Design by contract : the lessons of Ariane. *Computer*, 30(1) :129–130, 1997.

[87] K. Kavi, B. Buckles, and U. Bhat. A formal definition of data flow graph models. *Computers, IEEE Transactions on*, 100(11) :940–948, 2006.

[88] B. W. Kernighan and D. M. Ritchie. *C Programming Language (2nd Edition)*. Prentice Hall, 1988.

[89] K. Keutzer, S. Malik, and A. Newton. From ASIC to ASIP : the next design discontinuity. In *Proceedings. IEEE International Conference on Computer Design : VLSI in Computers and Processors*, pages 84–90. IEEE Comput. Soc, 2002.

[90] M. Khalil-Hani, V. P. Nambiar, and M. N. Marsono. Hardware Acceleration of OpenSSL Cryptographic Functions for High-Performance Internet Security. In *2010 International Conference on Intelligent Systems, Modelling and Simulation*, pages 374–379. IEEE, Jan. 2010.

[91] J. Knoop, O. Rüthing, and B. Steffen. Partial dead code elimination. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, PLDI '94, pages 147–158, New York, NY, USA, 1994. ACM.

[92] A. Koelbl and R. Jacoby. Solver technology for system-level to RTL equivalence checking. In *Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE '09*, pages 196 – 201, 2009.

[93] O. Kosheleva. Babylonian method of computing the square root : Justifications based on fuzzy techniques and on computational complexity. In *NAFIPS 2009 - 2009 Annual Meeting of the North American Fuzzy Information Processing Society*, pages 1–6. IEEE, June 2009.

[94] D. Kozen. On Hoare logic and Kleene algebra with tests. In *Proceedings. 14th Symposium on Logic in Computer Science*, pages 167–172. IEEE Comput. Soc, 1999.

[95] S. Kundu, S. Lerner, and R. Gupta. High-Level Verification. *IPSJ Transactions on System LSI Design Methodology*, 2 :131–144, 2009.

[96] K. Larsen. Practical Assertion-based Formal Verification for SoC Designs. In *2005 International Symposium on System-on-Chip*, pages 58–61. IEEE, 2005.

[97] K. Larson. Translation of an existing VMM-based SystemVerilog testbench to OVM.

[98] C. Lattner. Introduction to the llvm compiler system. *Proceedings of International Workshop on Advanced ...*, 2008.

[99] C. Lattner. LLVM and Clang : Next generation compiler technology. *The BSD Conference, Ottawa, Canada*, 2008.

[100] C. Lattner and V. Adve. LLVM : A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE.

[101] B. Le Gal and E. Casseau. Word-Length Aware DSP Hardware Design Flow Based on High-Level Synthesis. *Journal of Signal Processing Systems, Springer*, Online :1–17, 2010.

[102] Y. Le Traon, B. Baudry, and J.-M. Jezequel. Design by Contract to Improve Software Vigilance. *IEEE Transactions on Software Engineering*, 32(8) :571–586, Aug. 2006.

[103] C. Lee, S. Kim, and S. Ha. A Systematic Design Space Exploration of MPSoC Based on Synchronous Data Flow Specification. *Journal of Signal Processing Systems*, 58(2) :193–213, Mar. 2009.

[104] S. Lee, S. Min, and R. Eigenmann. OpenMP to GPGPU : a compiler framework for automatic translation and optimization. In *PPoPP '09 Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 101–110, 2009.

[105] T.-H. Lee. Hardware Architecture for High-Performance Regular Expression Matching. *IEEE Transactions on Computers*, 58(7) :984–993, July 2009.

[106] J. Levitt. Automatic assume guarantee analysis for assertion-based formal verification. In *Proceedings of the ASP-DAC 2005. Asia and South Pacific Design Automation Conference, 2005.*, volume 1, pages 561–566. IEEE.

[107] Y. Li, W. Wu, L. Hou, and H. Cheng. A Study on the Assertion-Based Verification of Digital IC. In *2009 Second International Conference on Information and Computing Science*, pages 25–28. IEEE, 2009.

[108] G. Martin and H. Chang. *Winning the SoC Revolution.* Springer, 2003.

[109] G. Martin and G. Smith. High-Level Synthesis : Past, Present, and Future. *IEEE Design and Test of Computers*, 26 :18–25, 2009.

[110] R. C. Martin. *Clean Code : A Handbook of Agile Software Craftsmanship.* Prentice Hall, 2008.

[111] B. Meyer. Design by Contract. In D. Mandrioli and B. Meyer, editors, *Advances in Object-Oriented Software Engineering*, pages 1—-50. Prentice Hall, 1991.

[112] B. Meyer. *Eiffel : The Language (Prentice Hall Object-Oriented Series).* Prentice Hall, 1991.

[113] B. Meyer. Applying "Design by Contract". *Computer*, 25(10) :40—-51, 1992.

[114] K. Morin-Allory and D. Borrione. Proven correct monitors from PSL specifications. In *Proceedings of the conference on Design, automation and test in Europe : Proceedings*, DATE '06, pages 1246–1251, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[115] K. Morin-Allory, M. Boule, D. Borrione, and Z. Zilic. Proving and disproving assertion rewrite rules with automated theorem provers. In *2008 IEEE International High Level Design Validation and Test Workshop*, pages 56–63. IEEE, Nov. 2008.

[116] K. Morin-Allory, M. Boule, D. Borrione, and Z. Zilic. Validating Assertion Language Rewrite Rules and Semantics With Automated Theorem Provers. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(9) :1436–1448, Sept. 2010.

[117] J. Myrcha and P. Rokita. Multimedia Objects Conversion for a Digital Repository - A Case Study. *Computer Vision and Graphics*, 7594/2012 :196–203, 2012.

[118] N. Narasimhan, E. Teica, R. Radhakrishnan, S. Govindarajan, and R. Vemuri. Theorem Proving Guided Development of Formal Assertions in a Resource-Constrained Scheduler for High-Level Synthesis. *Formal Methods in System Design*, 19(3), 2001.

[119] N. Narasimhan and R. Vemuri. On the Effectiveness of Theorem Proving Guided Discovery of Formal Assertions for a Register Allocator in a High-Level Synthesis System. In *Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics*, number 513, pages 367–386. Springer-Verlag, 1998.

[120] Y. Naveh and R. Emek. Random Stimuli Generation for Functional Hardware Verification as a CP Application. In P. Beek, editor, *Principles and Practice of Constraint Programming - CP 2005*, volume 3709 of *Lecture Notes in Computer Science*, page 882. Springer Berlin Heidelberg, 2005.

[121] M. Oliveira. *High-level specification and automatic generation of IP interface monitors*. Masterthesis, The University of British Columbia, 2003.

[122] Online. Accelera/OSCI - http ://www.accellera.org.

[123] Online. Altera - http ://www.altera.com.

[124] Online. Arm - http ://www.arm.com/.

[125] Online. C-to-Silicon - http ://www.cadence.com.

[126] Online. C-to-Verilog - http ://www.c-to-verilog.com.

[127] Online. Cadence - http ://www.cadence.com.

[128] Online. Catapult-C - http ://www.calypto.com.

[129] Online. ChipScope Pro - http ://www.xilinx.com/tools/cspro.htm.

[130] Online. Constrained Random Testing Causes Paradigm Shift - http ://chipdesignmag.com/display.php ?articleId=301&issueId=13.

[131] Online. GHDL - http ://ghdl.free.fr/.

[132] Online. GraphLab - http ://uuu.enseirb.fr/~legal/wp_graphlab/.

[133] Online. IEEE - http ://www.ieee.org.

[134] Online. Impulse-C - http ://www.impulseaccelerated.com/.

[135] Online. LLVM - http ://www.llvm.org.

[136] Online. math.h - http ://www.cplusplus.com/reference/clibrary/cmath/.

[137] Online. Mentor Graphics - http ://www.mentor.com/.

[138] Online. Mips - http ://www.mips.com/.

[139] Online. ModelSim SE - http ://www.model.com.

[140] Online. OCP-IP - http ://www.ocpip.org/.

[141] Online. Open Verification Library - http ://www.accellera.org/activities/committees/ovl.

[142] Online. Questa - http ://www.mentor.com/questa.

[143] Online. Reveal - http ://www.latticesemi.com.

[144] Online. SignalTap II Logic Analyzer State-Based Triggering Flow Design Examples - http ://www.altera.com/support/examples.

[145] Online. Symphony C Compiler - http ://www.synopsys.com.

[146] Online. SystemC TLM-2.0 - http ://www.doulos.com/knowhow/systemc/tlm2/.

[147] Online. Verification Academy - https ://verificationacademy.com/.

[148] Online. Xilinx - http ://www.xilinx.com.

[149] Online. Authoring assertion IP using OpenVera assertion language - http ://www.design-reuse.com/, 2002.

[150] Online. The Myth of EDA - the 70% rule - http ://www.chipdesignmag.com, 2008.

[151] Online. Best Square Root Method - Algorithm - Function (Precision VS Speed) - http ://www.codeproject.com/, 2010.

[152] Online. ITRS 2011 - http ://www.itrs.net/Links/2011ITRS/Home2011.htm, 2011.

[153] Online. Intelligent Testbench Automation - Catching on Fast - http ://-blogs.mentor.com/verificationhorizons, 2012.

[154] S. Owre and N. Shankar. A Brief Overview of PVS. In O. Mohamed, C. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science*, pages 22–27. Springer Berlin / Heidelberg, 2008.

[155] M. Parkinson, R. Bornat, and C. Calcagno. Variables as Resource in Hoare Logics. In *21st Annual IEEE Symposium on Logic in Computer Science (LICS'06)*, pages 137–146. IEEE, 2006.

[156] T. Parr. *The Definitive Antlr Reference : Building Domain-Specific Languages (Pragmatic Programmers)*. Pragmatic Bookshelf, 2007.

[157] M. Pellauer, M. Liz, D. Baltus, and R. Nikhi. Synthesis of synchronous assertions with guarded atomic actions. *Proceedings. Third ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2005. MEMOCODE '05.*, pages 15–24, 2005.

[158] D. K. Pradhan and I. G. Harris. *Practical Design Verification.* Cambridge University Press, 2009.

[159] J. M. Rabaey, A. Chandrakasan, and B. Nikolic. *Digital Integrated Circuits (2nd Edition).* Prentice Hall, 2003.

[160] D. Rosenband. *A Performance Driven Approach for Hardware Synthesis of Guarded Atomic Actions.* PhD thesis, MASSACHUSETTS INSTITUTE OF TECHNOLOGY, 2005.

[161] D. Rosenband. Hardware synthesis from guarded atomic actions with performance specifications. In *ICCAD-2005. IEEE/ACM International Conference on Computer-Aided Design, 2005.*, pages 784–791. Ieee, 2005.

[162] S. Rosenberg and K. Meade. *A Practical Guide to Adopting the Universal Verification Methodology (UVM).* Cadence Design Systems, 2010.

[163] D. Saha and S. Sur-Kolay. SoC : a real platform for IP reuse, IP infringement, and IP protection. *VLSI Design - Special issue on CAD for Gigascale SoC Design and Verification Solutions*, 2011 :1—-10, 2011.

[164] W. Savage, J. Chilton, and R. Camposano. IP reuse in the system on a chip era. In *Proceedings 13th International Symposium on System Synthesis*, pages 2–7. IEEE Comput. Soc, 2000.

[165] S. Sawant, U. Desai, G. Shamanna, L. Sharma, M. Ranade, A. Agarwal, S. Dakshinamurthy, and R. Narayanan. A 32nm Westmere-EX Xeon enterprise processor. In *2011 IEEE International Solid-State Circuits Conference*, pages 74–75. IEEE, Feb. 2011.

[166] B. C. Schafer and K. Wakabayashi. Design Space Exploration Acceleration Through Operation Clustering. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(1) :153–157, Jan. 2010.

[167] R. Schaller. Moore's law : past, present and future. *IEEE Spectrum*, 34(6) :52–59, June 1997.

[168] W. D. Schwaderer. *Introduction to Open Core Protocol : Fastpath to System-on-Chip Design.* Springer, 2012.

[169] A. Sengupta, R. Sedaghat, and Z. Zeng. Rapid design space exploration for multi parametric optimization of VLSI designs. In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pages 3164–3167. IEEE, May 2010.

[170] I. Sourdis, J. a. Bispo, J. a. M. P. Cardoso, and S. Vassiliadis. Regular Expression Matching in Reconfigurable Hardware. *Journal of Signal Processing Systems*, 51(1) :99–121, Oct. 2007.

[171] B. Stroustrup. *The C++ Programming Language, 4th Edition.* Addison-Wesley Professional, 2013.

[172] P. Sule and N. Mansouri. PROVERIFIC : experiments in employing (PSL) standard assertions in theorem-proving-based verification. In *48th Midwest Symposium on Circuits and Systems, 2005.*, pages 112–115 Vol. 1. IEEE, 2005.

[173] J. Tantivongsathaporn and D. Stearns. An Experience With Design by Contract. In *2006 13th Asia Pacific Software Engineering Conference (APSEC'06)*, pages 335–341. IEEE, 2006.

[174] Y. Tao. An introduction to assertion-based verification. In *2009 IEEE 8th International Conference on ASIC*, pages 1318–1323. IEEE, Oct. 2009.

[175] W. Wolf. Hardware-software co-design of embedded systems. *Proceedings of the IEEE*, 82(7) :967–989, July 1994.

[176] W. Wolf. A decade of hardware / software codesign. *Computer*, 36(4) :38–43, Apr. 2003.

[177] Y. Wolfsthal and R. Gott. Formal verification - is it real enough ? In *Proceedings. 42nd Design Automation Conference, 2005.*, pages 670–671. IEEE, 2005.

[178] T. Yoshimura and E. S. Kuh. Efficient Algorithms for Channel Routing. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 1(1) :25–35, 2006.

[179] M. Yue. A simple proof of the inequality "FFD (L) < 11/9 OPT (L) + 1, for all L" for the FFD bin-packing algorithm. *Acta Mathematicae Applicatae Sinica*, 7(4) :321–331, 1991.

[180] M. Zaki, S. Tahar, and G. Bois. Formal verification of analog and mixed signal designs : A survey. *Microelectronics Journal*, 39(12) :1395–1404, 2008.

[181] S. Zhang, A. I. Ahmed, and O. A. Mohamed. A re-usable verification framework of Open Core Protocol (OCP). In *2009 Joint IEEE North-East Workshop on Circuits and Systems and TAISA Conference*, pages 1–4. IEEE, June 2009.