Vérification formelle des résultats de la synthèse de haut niveau

J. Dushina

Thèse

Présentée par

Julia DUSINA

pour obtenir le grade de

Docteur de l'Université Joseph Fourier- Grenoble 1

(arrêté ministériel du 30 Mars 1992)

(Spécialité **Informatique)**

Vérification formelle des résultats de la Synthèse de Haut Niveau

Date de soutenance: 27 Octobre 1999

Composition du Jury:

Mesdames	Dominique Borrione	Directeur de thèse
	Laurence Pierre	
Messieurs	Nicolas Halbwachs	Président
	François Anceau	Rapporteur
	Jean-Luc Paillet	Rapporteur
	Ahmed Amine Jerraya	
	Raimund Ubar	

Thèse préparée au laboratoire **Techniques de l'Informatique et de la Microélectronique pour l'Architecture d'ordinateurs (TIMA)**

2

Remerciements

Je tiens avant tout à remercier Madame Dominique Borrione, qui a assuré la direction de ma thèse, pour m'avoir accueillie au sein de son équipe, pour son suivi régulier et ses conseils judicieux, mais surtout pour son amitié et l'aide qu'elle m'a donnée pendant ces quatre années.

J'ai eu beaucoup de plaisir à travailler également avec Monsieur Ahmed Jerraya, qui guida mon travail concernant la synthèse. Qu'il trouve ici l'expression de mon profond respect.

Un grand merci à mes rapporteurs: Monsieur François Anceau, du Conservatoire National des Arts et Métiers de Paris, et Monsieur Jean-Luc Paillet, de l'Université de Provence de Marseille, pour avoir accepté de rapporter ce travail malgré leurs charges que je sais lourdes et le manque de temps. Leurs remarques m'ont été précieuses.

Je remercie Monsieur Nicolas Halbwachs pour l'honneur qu'il me fait en présidant le Jury.

Je ne saurais oublier Monsieur Raimund Ubar de l'Université Technique de Tallinn, avec qui j'ai fait mes premiers pas dans le domaine de la recherche. Il a toujours su m'apporter aide et soutien.

Je dois beaucoup à Laurence Pierre et Pierre Ostier avec qui j'ai travaillé étroitement. Ils ont toujours été disponibles pour répondre à mes questions interminables et ont supporté avec philosophie ma "tête dure". Merci.

J'exprime toutes mes amitiés aux autres membres de l'équipe VDS, présents et anciens: Philippe Georgelin, Emil Dumitrescu, Claude Le Faou, Gérard Vitry, Adam Morawiec, Hakim Bouamama, Ayman Wahba et Pierre Pomes pour leur bonne humeur, leur humour et l'ambiance agréable de travail qu'ils ont su créer.

Je salue également les membres du laboratoire Tima, avec qui j'ai eu plaisir à travailler, à partager le bureau ou le repas de midi: Elisabeth Berrebi, Wander Cesario, Pascal Coste, Jean-Mark Daveau, Philippe Guillaume, Fabiano Hessel, Gilberto Marchioro, Philippe Le Marrec, Imed Moussa, François Naçabal, Richard Pistorius, Maher Rahmouni, Rodolphe Suescun, Zoltan Sugar, Kholdoun Torki, Carlos Valderrama, Nacer Zergainoh.

Je ne serais pas juste si je ne saluais pas les secrétaires Corinne Durand-Viel, Isabelle Essalhiene, Isabelle Amielh, Patricia Scimone, Chantal Benis et Lucie Torella pour leur disponibilité et leur sourire.

Un très grand merci à mes parents lointains qui m'ont soutenu et me soutiennent toujours.

Je remercie profondément Hichem, à qui je dois, finalement, cette thèse.

Résumé

Pour satisfaire la demande du marché actuel, plusieurs outils commerciaux de vérification formelle sont apparus ces dernières années. Le niveau le plus abstrait de description accepté dans la plupart de ces outils est le niveau appelé transfert de registres, c'est-à-dire une description avec des cycles d'horloge explicitement définis.

Pour rester compétitifs, néanmoins, les concepteurs sont obligés d'élever le niveau d'abstraction et commencent à utiliser des outils de synthèse de haut niveau. Cette thèse a pour objet la vérification formelle des résultats de synthèse de haut niveau par rapport à la spécification initiale décrite en VHDL.

Nous proposons une méthodologie de vérification qui épouse le flot de conception et consiste en la vérification de deux étapes principales: l'ordonnancement et l'allocation. La vérification de chaque étape est fondée sur un modèle de machine abstraite que nous avons défini: contrairement au modèle de machine d'états finis classique, il réduit considérablement l'espace d'états d'où les registres de la partie opérative sont exclus. En outre, la machine abstraite est similaire aux descriptions VHDL utilisées lors de la synthèse et offre, par conséquent, un niveau d'abstraction plus élevé de représentation des circuits. La preuve d'équivalence entre la machine abstraite et la machine d'états finis classique justifie la première et constitue une des contributions théoriques de la thèse.

Un prototype d'outil basé sur la simulation symbolique a été développé et exécuté sur des benchmarks de la synthèse comportementale. La thèse s'achève sur les problèmes ouverts et les axes de recherche à explorer.

Mot Clés: vérification formelle, synthèse de haut niveau, ordonnancement, allocation, machine abstraite, simulation symbolique.

Abstract

To satisfy the present market requirements, several commercial formal verification tools appeared recently. The highest description level accepted by these tools is the so-called Register Transfer Level, i.e. with clearly identified clock periods.

Because of very strong competition the designers must, however, start from more abstract initial descriptions and use high-level synthesis tools. This thesis addresses the formal verification of high-level synthesis results with respect to the initial specification given in VHDL.

We propose a methodology that follows the design flow and includes the verification of two main synthesis steps: scheduling and allocation. The verification of each step is based on an abstract machine model that we developed: contrary to the classic finite state machine, it reduces considerably the state space by keeping the data path registers symbolic. Moreover, the similarity between the abstract machine and the VHDL description provides facilities for the model construction and understanding. The proof of the equivalence between abstract and classical finite state machines justifies the abstract machine and constitutes one of the theoretical contributions of the thesis.

A prototype tool based on symbolic execution was developed and applied to some high-level synthesis benchmarks. Finally, the thesis points out some open problems and suggests some research directions to solve them.

Key words: formal verification, high-level synthesis, scheduling, allocation, abstract machine, symbolic simulation.

Table des matières

Chapitre 1

Introduction

1.1 Problème étudié dans la thèse

Ces dernières années, l'industrie de l'électronique a vu une explosion de la demande en ordinateurs, en téléphone cellulaires, en unités de communication à grande vitesse et en tout genre de produits d'utilité quotidienne. Veillant sur leurs parts du marché, les fournisseurs ont construit des produits ayant un nombre croissant de fonctionnalités, plus performants, moins chers et moins encombrants. Pour ce faire, ils ont crées des systèmes complexes massivement intégrés avec moins de boîtiers et un mélange de parties matérielles et logicielles. L'avènement des technologies sous-microniques, des technologies programmables à grande capacité a aidé à supporter cette intégration sans cesse croissante: c'est l'ère des systèmes sur une puce ou "System on a Chip" en anglais. Le goulot d'étranglement qui s'est manifesté est l'habileté des concepteurs à faire face à cette complexité et à tenir leurs engagements dans les délais impartis.

Cette situation a largement contribué à la prise de conscience du besoin de nouvelles méthodologies de conception, de validation et de test. Ainsi, les langages de description de matériel, les technologies programmables et les outils de synthèse à différents niveaux d'abstraction sont devenus des impératifs et ont gagné une popularité incontestée. Leur large adoption dans l'industrie le prouve.

Outre ces derniers, des outils de vérification formelle, des techniques de simulation conjointe matériel-logiciel, la conception pour la ré-utilisation des blocs appelés méga-fonctions, la conception des composants virtuels s'imposent de plus en plus comme facteurs essentiels pour l'amélioration de la productivité des concepteurs. En effet, les blocs ré-utilisables validés constituent une capitalisation de l'investissement en recherche et développement. De même, la conception et la validation de ces blocs dans l'objectif d'une ré-utilisation ultérieure est un enjeu stratégique pour les constructeurs de systèmes et les fournisseurs de technologies.

C'est dans ce cadre que s'intègre notre travail de recherche. Plus précisément, l'utilisation des descriptions de haut niveau et des outils de synthèse correspondants offre des avantages qu'il est inutile de rappeler tous. Nous ne citons que le grand degré de liberté pour l'exploration de l'espace des solutions et la facilité de correction et d'adaptation des descriptions initiales. D'un autre coté, le défi de validation des résultats de la synthèse de haut niveau reste d'actualité. De récentes études et rapports émanants de l'industrie ([ESV+98], [Yan98]) ont confirmé que cette activité de validation avec des méthodes de simulation prend des proportions très importantes et atteint

dans certains cas jusqu'à 60% du temps total dédié à la conception. Ceci est essentiellement dû à la lenteur de ces outils et à la limitation qui en découle à savoir l'impossibilité de l'exhaustivité. Ceci explique en grande partie, l'émergence de plusieurs outils industriels de vérification formelle. Nous nous intéressons donc aux deux volets : la synthèse de haut niveau et la vérification formelle de ses résultats par rapport à la spécification initiale décrite en VHDL.

1.2 État de l'art

Le sujet de notre thèse se situe à la frontière de deux domaines très vastes: la synthèse de haut niveau et la vérification formelle. Chacun d'eux contient plusieurs directions de recherche qui elles-mêmes constituent un domaine large et possèdent leurs propre états de l'art. Par exemple, l'ordonnancement, l'allocation, le "retiming", la construction de modèles internes tels que les graphes de flot de données de contrôle composent la synthèse de haut niveau.

La vérification formelle, elle, se divise en vérification de logiciels et de matériels. La classification de la vérification de matériels (ou bien de circuits intégrés) dépend à son tour du critère choisi. Si ce critère est la relation entre spécification et implémentation, alors les méthodes suivantes sont à citer: preuve de théorèmes, "model checking", contrôle d'équivalence, inclusion de langages, etc. Un autre critère engendrera une classification différente.

Dans cette thèse nous n'entendons pas faire une étude exhaustive de toute la diversité des approches existantes dans les domaines de la synthèse comportementale et de la vérification formelle. Notons simplement que plusieurs publications et ouvrages ([CP88], [Yoe90], [Gup92], [CW96], [KBES96], [Eve99]) sur la vérification formelle sont recommandés au lecteur désireux d'avoir des notions de base et une idée globale sur ce sujet. L'introduction à la synthèse comportementale peut être trouvée dans [MPC90], [WC91], [GDWL92], [GR94], [Kna96], [Cam96], [JDKR97].

Nous nous concentrons en revanche sur le sujet précis de la vérification des résultats de la synthèse de haut niveau en portant l'accent sur la vérification plutôt que sur la synthèse. Or, le processus de la synthèse qui nous intéresse a pour la spécification une description comportementale VHDL, la formalisation de ce langage influe sur la vérification de ces résultats. Pour cette raison nous présentons un état de l'art de la vérification de la synthèse de haut niveau et de la sémantique formelle proposée dans la littérature pour le langage VHDL.

1.2.1 Vérification formelle de la synthèse de haut niveau

Un petit nombre de travaux sont dédiés à la vérification formelle des résultats de la synthèse de haut niveau.

- La méthode proposée dans [HAFM97], fondée sur la technique des Diagrammes de Décision Binaire (BDD), permet de vérifier des blocs de taille importante: jusqu'à 530 bascules et 30.000 portes. L'efficacité est atteinte par l'exploration de la correspondance entre les registres de la partie opérative des deux descriptions comparées et par la division de l'ensemble de sorties en des sous-ensembles lors de la comparaison. La stratégie de vérification est basée sur la théorie de l'inclusion de machines d'états finis (abrégé en FSM), l'une

représentant la spécification et l'autre l'implémentation. Le niveau le plus abstrait de spécification est une description comportementale ordonnée, c'est-à-dire une description où les opérations sont associées aux périodes d'horloge explicitement définies; elle correspond dans notre cas à la description obtenue après l'étape d'ordonnancement. La description ordonnée est en quelque sorte synthétisée en remplaçant les opérations abstraites, telles que "+", "-", par les blocs logiques prouvés a priori ("golden templates") et ensuite vérifiée par rapport à la description au niveau logique. A notre connaissance, toutes les techniques basées sur les BDD sont limitées par la spécification algorithmique ordonnée et ne peuvent pas, en principe, être utilisées pour la vérification de la synthèse comportementale, dont la spécification se situe à un niveau d'abstraction plus élevé.

- Les méthodes développées à Karlsruhe ([BE97], [EKB97] et [BE98]) représentent une direction de recherche appelée *synthèse formelle* (*formal synthesis*) dans [KBES96]. Contrairement à la pré-vérification (quand le programme lui-même de synthèse est vérifié) et la post-vérification (quand l'exactitude d'un circuit est vérifié indépendamment de l'outil de synthèse utilisé), la synthèse formelle sait *comment* l'outil de synthèse fonctionne. Les petites étapes du flot de conception sont vérifiées séparément l'une de l'autre en arrière plan de l'outil principal de synthèse ([BE97] et [EKB97]). Les exemples des étapes sont l'ordonnancement et le "retiming". Le formalisme employé pour les preuves est la logique d'ordre supérieur et l'outil formel correspondant est le démonstrateur de théorèmes HOL. L'idée de la décomposition du processus de synthèse est proche de l'idée développée dans cette thèse: nous proposons, notamment, de vérifier les deux étapes fondamentales de la synthèse comportementale (l'ordonnancement et l'allocation) séparément. La différence entre notre approche et celle de [BE97] et [EKB97] réside dans le nombre d'étapes considérées: dans [BE97] et [EKB97] ce nombre est grand et les étapes sont, par conséquent, petites. Ce fait est lié à l'utilisation de HOL, qui est un démonstrateur non-automatique et demande beaucoup d'efforts de la part du concepteur pour définir les tactiques de preuves (il est à noter qu'une fois les tactiques trouvées, la preuve de l'étape correspondante devient automatique). L'ordonnancement considéré dans [BE97] se limite par ailleurs à l'ordonnancement du graphe de flot de données seulement. La partie contrôle est exclue. Dans [BE98] une tentative de vérification de la description comportementale par rapport à la description au niveau du transfert de registres est faite.

- Les travaux de [MV98] emploient le démonstrateur de théorèmes PVS pour vérifier la description comportementale par rapport à son implémentation au niveau du transfert de registres. La méthode exposée est similaire à celle développée dans cette thèse par la comparaison de deux machines d'états finis *transition par transition*. Les états et les chemins d'exécution critiques sont supposés être définis par l'outil de synthèse. Les deux circuits sont considérés comme équivalents si les valeurs symboliques des variables sont équivalentes dans les deux circuits à la fin de chaque chemin critique. L'inconvénient majeur de [MV98] est le niveau d'abstraction relativement bas de la spécification, qui est le graphe de flot de contrôle obtenu après "certaines transformations comportementales". L'ordonnancement est considéré dans [MV98] comme "le partage de ressources sous l'échelle temporelle pre-

définie" et correspond à l'étape d'allocation de l'outil que nous vérifions. La formalisation de la méthode se limite, par ailleurs, aux définitions des états, des chemins et des variables critiques. Finalement, l'utilisation de PVS rend la mise en oeuvre de la méthodologie, qui est assez simple, très compliquée et encombrante.
Les autres travaux issus de l'Université de Cincinnati ([NKV96], [NV96] et [NRV96]) proposent de vérifier le contrôleur (la partie opérative n'est pas considérée) généré par la synthèse comportementale à partir d'un processus VHDL. Le contrôleur est décrit comme un ensemble de FSM hiérarchiques en supposant l'existence d'un protocole rigide de communication entre le circuit et l'environnement. Les propriétés des FSM composantes sont ensuite vérifiées à l'aide du "model checker" SMV.

– Une méthode originale est présentée dans [ABRM98]. Elle définit l'équivalence entre une description comportementale ordonnée et une description au niveau transfert de registres si les deux propriétés suivantes sont vérifiées. La première propriété est le partage non-contradictoire de registres par les variables de l'algorithme. La deuxième propriété est les affectations identiques pour la spécification et pour l'implémentation dans n'importe quel moment d'exécution, pourvu que les deux circuits commencent l'exécution dans leurs états initiaux. Pour la vérification du partage de registres, un graphe composé de chemins contradictoires est dérivé à partir de la description initiale. Un chemin est contradictoire s'il contient deux variables attribuées au même registre telles que la première soit utilisée en lecture, tandis que la deuxième est affectée avant la lecture de la première. Cependant, un chemin contradictoire peut n'être jamais exécuté. Dans ce cas l'affectation d'une variable attribuée à un registre n'est jamais suivie par la lecture d'une autre variable attribuée à ce même registre. Cette propriété, exprimée dans la logique CTL, est vérifiée à l'aide du "model checker" VIS. Si aucun chemin contradictoire n'est exécuté, alors le partage de registres est correct. Remarquons, que les opérations non pertinentes sont exclues du graphe fourni à VIS. La deuxième propriété est vérifiée état par état en supposant que la correspondance entre les états de la spécification et de l'implémentation soit connue. Une idée similaire est employée dans notre thèse: on ne considère que les états du contrôleur en faisant abstraction des états de la partie opérative. Cependant, contrairement à notre méthode qui associe à chaque structure une représentation fonctionnelle, dans [ABRM98], une représentation structurelle est déduite des fonctions associées à un état de la spécification. Cette structure est comparée ensuite avec la structure de l'implémentation. La simulation symbolique est utilisée pour prouver que les sorties dans les deux structures ont les mêmes valeurs. La spécification reste toujours limitée par la description comportementale ordonnée. Cependant, la méthode nous apparaît originale, robuste et prometteuse.

– La méthodologie exposée dans [EHR99] vise à vérifier l'ordonnancement en général, indépendamment de l'outil de synthèse. Le circuit avant et après l'ordonnancement est représenté dans un langage interne appelée *Language of Labelled Segments* (LLS). La description LLS consiste en segments acycliques, chaque segment associé à une étiquette d'entrée et éventuellement plusieurs étiquettes de sortie définissant le flot de contrôle. Ensuite les transformations de segments, qui préservent l'équivalence d'une description LLS en terme

de calcul, sont introduites. L'équivalence en terme de calcul ("computational equivalence") de deux descriptions LLS signifie que les deux descriptions produisent les mêmes valeurs finales aux étiquettes finales si les valeurs initiales aux étiquettes initiales sont identiques. La technique de vérification consiste, finalement, à appliquer les transformations à la description initiale en essayant d'obtenir la description finale. Si cette démarche réussit, alors les deux descriptions sont équivalentes et l'étape d'ordonnancement est correcte. Nous allons revenir sur cette méthode et la comparer avec notre approche dans le chapitre 4.

1.2.2 Sémantiques formelles proposées pour le langage VHDL

Le langage VHDL se caractérise par sa sémantique en terme de simulation définie dans un anglais très prosaïque, parfois obscur voire même ambigu ([IEE93]). De plus, dans le but de maintenir le modèle déterministe, le fonctionnement du simulateur est très complexe. Pour ces raisons, les différentes approches pour définir de façon formelle une sémantique de VHDL se limitent à un sous-ensemble du langage et souvent se concentrent sur la formalisation d'un cycle de simulation ([Ska96], [ABOR90]), la simulation δ inclue. Nous citons par la suite quelques approches proposées dans la littérature.

- Ainsi, dans [DB95a], [DB95b] et [Dǿ96] un cycle de simulation représente une transition d'une machine abstraite. Un état de cette machine est défini par les contenus des variables et par les valeurs des signaux entre les transitions. Un "model checker" est conçu pour vérifier les propriétés exprimées par la logique temporelle CTL.

- Une approche similaire est adoptée par [Enc95] et [Baw96] où seuls les états en fin de cycle de simulation sont considérés. Notons encore qu'ils utilisent les réseaux de Petri temporisés comme formalisme intermédiaire.

- Les travaux de [RK95] et [Ree95] proposent une sémantique d'un sous-ensemble de VHDL transformée vers les graphes de flots d'exécution séparant les données de la partie contrôle. Ils utilisent la logique d'ordre supérieur HOL pour la vérification des propriétés de sûreté.

- [Rea94], [RE94], [Rus95] et [Nic99] emploient le démonstrateur de théorèmes de Boyer-Moore. La formalisation de VHDL faite dans [Rus95] se limite à la description d'un circuit au niveau des portes tandis que [Rea94], [RE94] et [Nic99] visent aussi la formalisation des processus. Dans [Rea94] l'idée d'une simulation symbolique à l'aide de Nqthm est proposée. Cependant, la formalisation des constructions "wait" n'est pas accomplie.

- [BFK94] [BFK95] définissent une sémantique dénotationelle d'un sous-ensemble de VHDL où le délai δ est interdit. Cette sémantique est ensuite mise en oeuvre dans la logique de Hoare. Un système de validation écrit en Prolog réduit les preuves sur des programmes VHDL aux preuves sur la validité des conditions initiales.

La plupart des travaux présentés ci-dessus sont dédiés à la *validation* d'un circuit décrit en VHDL. C'est-à-dire, étant donnée une description VHDL, le but est de dire si le circuit correspondant à cette description satisfait certaines propriétés telles que la propriété de sûreté ou de vivacité. Notre tâche est différente. Supposant que la spécification initiale est a priori correcte,

le problème est de prouver l'équivalence entre les descriptions avant et après une étape de raffinement. La preuve d'équivalence est évoquée dans [Rea94] et [Rus95]. Cependant, le niveau d'abstraction est le même pour la spécification et pour l'implémentation. Par exemple, [Rus95] prouve qu'une structure décrite comme une interconnexion des portes correspond à une formule booléenne spécifiée. Dans notre cas, la spécification et l'implémentation se trouvent à des niveaux d'abstraction très différents et très éloignés l'un de l'autre. Cette particularité nécessite le développement de modèles qui à la fois reflètent les différents niveaux d'abstraction et qui en même temps soient suffisamment proches pour la comparaison.

Nous présentons le modèle adopté pour la description initiale dans le chapitre 4.

1.3 Organisation du mémoire

Le chapitre 2 introduit brièvement l'outil de la synthèse de haut niveau Amical qui nous a servi d'étalon et sur lequel nous avons orienté notre approche.

Or, le domaine formel sous-entend l'existence d'un outil mathématique permettant des raisonnements précis sur le problème posé, le chapitre 3 est dédié au modèle de la machine abstraite qui est la base de la méthodologie proposée pour la vérification.

Cette méthodologie consiste à vérifier séparément chacune des deux étapes principales de la synthèse. Ainsi, la vérification de l'étape d'ordonnancement fait l'objet du chapitre 4. Les modèles utilisés, ainsi que les algorithmes nécessaires pour leur comparaison sont présentés.

Le chapitre 5 décrit la méthode pour la vérification de l'étape d'allocation. Il introduit également le prototype d'un outil développé pour la vérification de cette étape.

Le chapitre 6 conclut et présente les perspectives de notre travail.

Chapitre 2

Amical : l'outil de synthèse de haut niveau

Le travail de cette thèse entre dans le cadre du projet de développement du système Amical- l'outil de synthèse de haut niveau développé par le groupe SLS du laboratoire TIMA ([JPO93], [JDKR97]). Amical accepte la description des circuits en VHDL comportemental et produit une architecture au niveau transfert de registres décrite en ce même langage. Le flot de conception d'Amical est constitué principalement de deux phases: l'ordonnancement de la description initiale et l'allocation des ressources avec la génération de l'architecture finale (figure 2.1). Chaque phase est une opération extrêmement complexe et constitue en elle même un sujet de thèse ([Rah97], [Din96]). Nous nous limitons dans ce travail à une description concise de chaque phase dans le but de la vérification formelle.

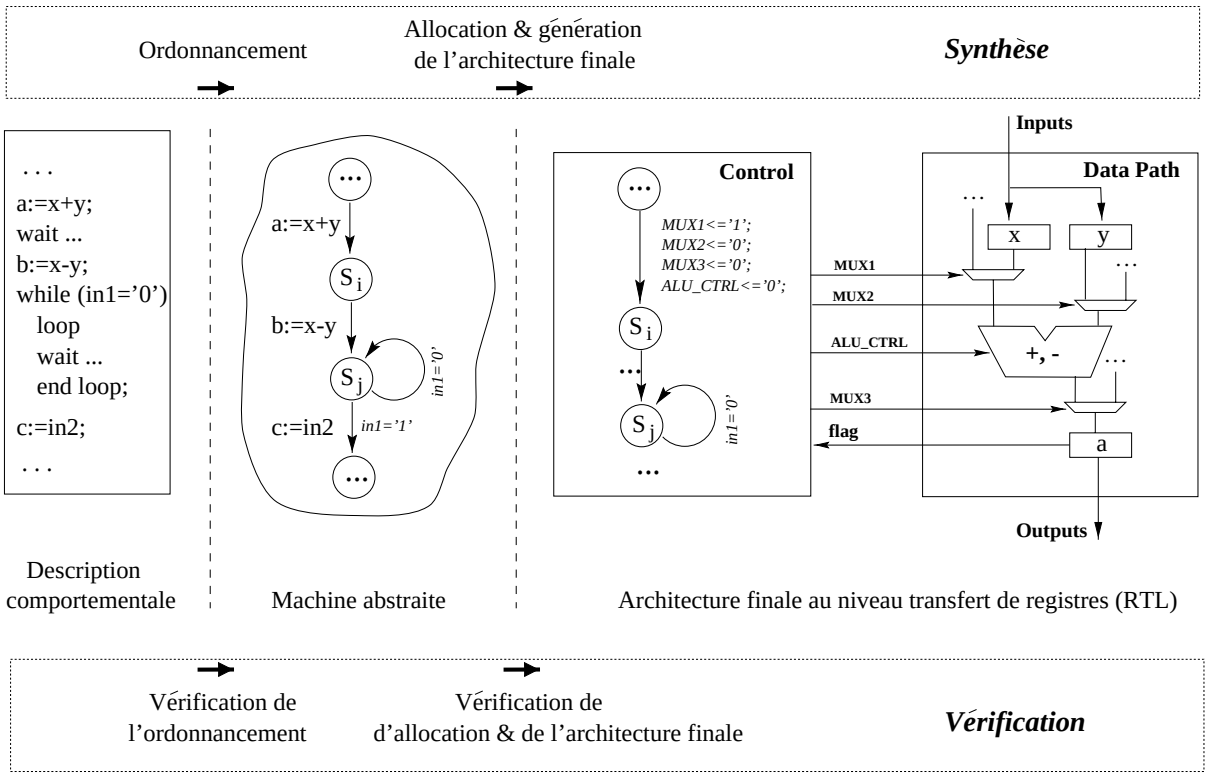


FIG. 2.1 – *Flot de conception d'Amical*

2.1 Ordonnancement de la description initiale

L'ordonnancement transforme une description comportementale VHDL sous la forme d'un processus unique en une *machine d'états finis abstrait*e. Cette machine est constituée d'un ensemble d'états de contrôle et d'opérations *abstraites* (par exemple $a := b + c$) attachées aux transitions entre les états. Les opérations attachées à une transition peuvent en principe s'exécuter en parallèle.

Nous employons le terme *abstrait* car les opérations, et par conséquence la machine d'états finis, ne portent aucune information temporelle ou structurelle. Ni le nombre des cycles d'horloge réels, ni la structure nécessaire pour l'exécution des opérations n'est connu à la fin de cette étape de la synthèse.

La notion d'abstraction est très populaire de nos jours quand on cherche à présenter des systèmes complexes sans détail encombrant et les termes "abstraction", "abstrait", "machine abstraite", etc. sont largement utilisés dans la littérature. Cependant, puisque il n'existe pas encore de notion universelle d'abstraction, comme par exemple, la notion de machine d'états finis, nous utilisons le terme "machine d'états finis abstraite" ou simplement "machine abstraite" tout au long de notre thèse pour désigner le modèle d'un circuit après l'étape d'ordonnancement.

Par la suite nous décrivons brièvement l'ordonnancement employé dans Amical. En premier lieu, un graphe de flot de contrôle (control flow graph) est extrait de la description comportementale VHDL. Les instructions de contrôle, telles que "wait" , "if", "case", "while", constituent les sommets **branchement** de ce graphe et correspondent aux premiers états de contrôle de la machine abstraite. Les affectations des variables/signaux et les appels des procédures constituent les sommets **instruction** de ce graphe et correspondent aux opérations associées aux transitions de la machine abstraite. L'ordonnancement change éventuellement cette machine abstraite initiale:

- de nouveaux états peuvent être insérés;
- certains états peuvent être supprimés;
- l'ordre des opérations peut être changé et
- des opérations peuvent migrer d'une transition de la machine abstraite à une autre.

Certains changements peuvent engendrer une modification des conditions associées aux états de contrôle comme le montre la figure 2.2. L'opération $x := x + 1$ est transférée de la transition $S_1 \rightarrow S_2$ aux transitions $S_2 \rightarrow S_3$ et $S_2 \rightarrow S_4$. Le résultat est le remplacement de la condition $(x == 0)$ par la condition $((x + 1) == 0)$ et la suppression de l'état S_2. Cette modification permet d'économiser un cycle d'horloge pendant l'exécution réelle du circuit.

En outre, Amical permet l'utilisation d'unités complexes telles que des co-processeurs ([KDJ94], [KDJ95], [BKV+96], [Kis96]). Dans ce cas, un ensemble d'opérations est associé à un co-processeur. L'utilisateur est chargé de les employer correctement dans la description initiale, et de fournir l'interface avec le co-processeur pour chaque opération associée. Un exemple de telles opérations associées à un co-processeur est montré sur la figure 2.3. L'opération $mult\,(a, b)$ commande au

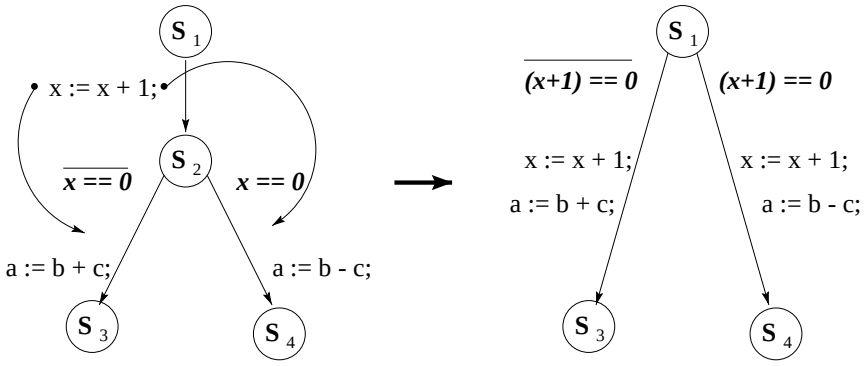


FIG. 2.2 – *Changement des conditions dû à la migration d'une opération*

co-processeur de commencer la multiplication des variables a et b. L'opération $mult\_res\,(c)$ permet de récupérer le résultat de la multiplication. Utilisées ensemble avec la construction *while* dans la description initiale, ces opérations réalisent une opération de multiplication dont la durée d'exécution n'est pas définie a priori (figure 2.3.a). La description initiale se transformera après l'ordonnancement en une machine abstraite (figure 2.3.b). La transition $S_i \rightarrow S_j$ de cette machine lance la multiplication. La machine abstraite restera ensuite dans l'état S_j jusqu'à l'arrivée du signal du co-processeur *done* égal à '1', qui indique que le résultat de la multiplication est disponible. L'opération *mult_res* réalisera le transfert du résultat dans le registre c.

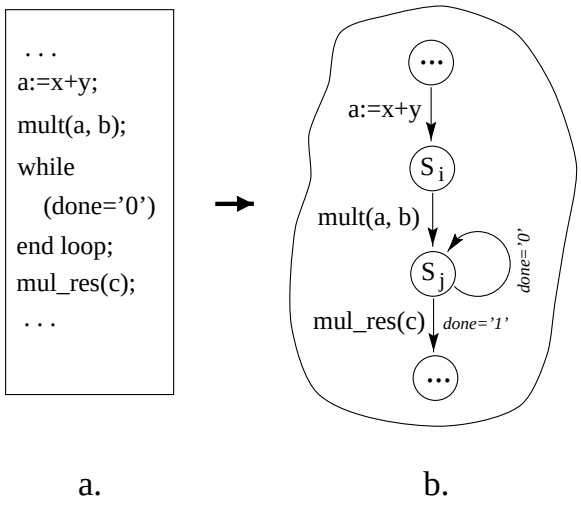


FIG. 2.3 – *L'utilisation du co-processeur*

2.2 Allocation des ressources et génération de l'architecture finale.

La machine abstraite est raffinée pendant cette phase de la synthèse. Un contrôleur et un chemin de données sont dérivés en parallèle. Le contrôleur garde le même nombre d'états que la machine abstraite, sauf que certaines opérations peuvent demander des états supplémentaires si leur temps d'exécution est supérieur au cycle d'horloge fixé. Les opérations attachées aux transitions de la machine abstraite sont remplacées par le positionnement des signaux de contrôle qui

commandent le flot de données nécessaire à l'exécution de ces opérations dans le chemin de données (figure 2.1). Le chemin de données, lui, est construit à partir des unités capables d'exécuter les opérations de la machine abstraite et des unités d'interconnexion telles que multiplexeurs et/ou interrupteurs ("switch").

Les opérations associées à un co-processeur sont raffinées de la même façon que les autres opérations: elles sont remplacées par les signaux de contrôle correspondants. Par exemple, l'opération $mult(a, b)$ (figure 2.3) sera remplacée par les signaux de contrôle qui assurent le transfert des contenus des registres a et b aux entrées du co-processeur et le signal de contrôle du co-processeur pour exécuter une multiplication. L'opération $mult\_res$ (c) sera remplacée par les signaux de contrôle qui réalisent une connexion de la sortie du co-processeur avec l'entrée du registre c et le signal d'écriture pour ce registre.

2.3 Vérification des résultats de la synthèse de haut niveau.

La possibilité d'utiliser des co-processeurs comme des blocs de base rend la vérification des résultats de la synthèse extrêmement difficile. Il faut assurer la cohérence du fonctionnement total d'un circuit où le contrôleur maître et un (des) co-processeur(s) s'exécutent éventuellement en parallèle. Ceci nécessite le développement du modèle de machines d'états finis communicantes, puisque un co-processeur à son tour peut être une machine d'état finis. Une tentative de construction d'un tel modèle a été faite au début de ce travail de recherche ([DBJ96], [Dus97]).

Les travaux accomplis dans [DBJ96] et [Dus97] ont dégagé une série de problèmes à résoudre. Tout d'abord, pour construire le modèle des machine d'états finis communicantes, un modèle clair et rigoureux pour chaque composant, c'est-à-dire pour le contrôleur maître et pour chacun des co-processeurs, doit être développé. Le modèle de la machine d'états finis classique n'est pas suffisant pour représenter l'ensemble des opérations de la description comportementale (affectations de variables, opérations arithmétiques et logiques complexes). Il a donc fallu proposer un autre modèle adapté à la synthèse de haut niveau. Le développement de ce modèle est une étape essentielle pour la vérification des résultats de la synthèse de haut niveau, car tous les raisonnements formels en sont issus. En outre, ce modèle peut servir de base pour la formalisation des circuits avec des co-processeurs et même pour la formalisation des circuits décrits au niveau système.

Après avoir développé le modèle pour chacun des composants et avant la vérification du fonctionnement total d'un circuit il faut s'assurer que chaque élément du système est correctement mis en oeuvre. Dans le cas d'Amical, cela signifie la vérification de la construction correcte du contrôleur maître. Avant de vérifier qu'au moment de la récupération des résultats du co-processeur ces résultats sont réellement prêts, il faut assurer que les données précédemment fournies au co-processeur sont correctes: cela veut dire que les opérations "propres" au contrôleur maître sont bien ordonnées et allouées.

La vérification du contrôleur maître constitue la vérification formelle des résultats de la synthèse de haut niveau avec des unités simples (les co-processeurs ne sont pas utilisés comme blocs de base pour la synthèse) et reste une tâche difficile. Il faut prouver que la spécification sous une forme d'un processus VHDL est bien mise en oeuvre par l'architecture finale au niveau

transfert de registres. La difficulté consiste à démontrer l'équivalence entre deux descriptions dont les niveaux d'abstraction sont très différents. La description initiale se situe au niveau comportemental, c'est à dire sans aucune précision sur le temps d'exécution et sur l'architecture cible. La description finale représente un circuit dont le fonctionnement est raffiné au cycle d'horloge près et la structure du chemin de données est fixée.

La vérification formelle de la synthèse de haut niveau avec des unités simples, comme la tâche primaire de la vérification des circuits décrits aux niveaux d'abstraction plus hauts, constitue le sujet de notre thèse. Compte tenu que la distance entre la description initiale et l'architecture finale est très grande pour une seule étape de vérification, nous proposons une approche de la vérification qui épouse le flot de conception et consiste aussi en deux phases (figure 2.1):

1. vérification de l'ordonnancement
2. vérification de l'allocation des ressources et de la génération d'architecture finale.

La vérification de chaque étape se déroule de la manière suivante: deux modèles formels, correspondant au début et à la fin de l'étape, sont développés. Une étape est correctement mise en oeuvre si deux modèles sont équivalents[1]. Nous définissons donc la notion rigoureuse de l'équivalence entre le modèle de départ et le modèle d'arrivée. Pour valider notre idée, un prototype d'outil fondé sur la définition de l'équivalence de deux modèles est réalisée pour l'étape d'allocation.

Ainsi, dans notre thèse nous avons développé trois modèles: un pour la description comportementale initiale, un pour un état intermédiaire, c'est à dire pour la machine abstraite, et un pour l'architecture finale au niveau de transfert de registres. Évidement, le modèle de la machine abstraite sert pour la vérification de deux phases en même temps: il est le modèle d'arrivée pour la vérification de l'ordonnancement, et le modèle de départ pour la vérification de l'allocation. Ce modèle de la machine abstraite, qui est la base de notre travail, fait l'objet du chapitre suivant. Les modèles pour la description initiale et pour l'architecture finale sont présentés dans les chapitres dédiés à la vérification de l'ordonnancement et l'allocation (respectivement les chapitres 4 et 5). Dans ces mêmes chapitres nous définissons l'équivalence entre le modèle de départ et le modèle final pour chaque phase.

2.4 Résumé

Dans ce chapitre nous avons introduit l'outil de synthèse de haut niveau Amical, les problèmes principaux de la validation de ces résultats et les grandes lignes de la méthodologie que nous proposons pour la vérification formelle de la synthèse comportementale.

1. Il s'agit bien de l'équivalence et non pas de l'implication de deux modèles.

Chapitre 3

Modèle adopté pour la machine abstraite

Le modèle de la machine d'états finis abstraite introduit l'idée essentielle de notre formalisation de la synthèse de haut niveau et constitue une base pour les deux autres modèles.

La machine d'états finis abstraite que nous considérons est inspirée par le modèle FSMD (FSM with a datapath) proposé par Gajski ([GR94]). Ce qui distingue une FSM traditionnelle d'une FSMD est la définition de l'état de la machine. Dans une FSM traditionnelle, tous les éléments de mémorisation , qu'ils soient dans la partie opérative ou dans la partie contrôle, sont des variables d'état. Ainsi, toutes les valeurs possibles d'un registre de 32 bits font partie de l'espace d'états, et ce registre contribue pour 2^{32} valeurs différentes.

Dans une FSMD, seuls les éléments de mémorisation de la partie contrôle sont pris en compte pour énumérer l'état de la machine, tandis que les registres de la partie opérative sont représentés par l'ensemble des variables mémorisées. L'espace d'état de la machine est donc réduit aux états représentés par les éléments de mémorisation de la partie contrôle.

Nous donnons ci-dessous la définition du modèle FSMD et représentons ensuite le modèle de la machine abstraite.

3.1 Définition du modèle FSMD de Gajski

Dans [GR94] les notations suivantes sont adoptées:
S est l'ensemble des états de la partie contrôle,
I est l'ensemble des entrées,
O est l'ensemble des sorties et
VAR est l'ensemble des variables mémorisées.
EXP est l'ensemble des expressions, construites sur les variables mémorisées:

$$EXP = \{g(x, y, z, ...) \mid x, y, z, ... \in VAR\}$$

A est l'ensemble des affectations qui associent une expression EXP à une variable:

$$A = \{X \leftarrow e \mid X \in VAR, e \in EXP\}$$

$STAT$ est l'ensemble des signaux des statuts définis comme des relations logiques:

$$STAT = \{Rel(a,b) \mid a,b \in EXP\}$$

Étant données les définitions précédentes, le modèle FSMD est défini par un quintuplet:

$$< S, I \times STAT, O \times A, f, h >$$

où la fonction de transition f et la fonction de sortie h sont définies comme $S \times (I \times STAT) \rightarrow S$ et $S \times (I \times STAT) \rightarrow O \times A$ respectivement.

Ce modèle représente une future architecture où la partie contrôle et la partie opérative sont séparées. La partie opérative est représentée par les affectations des variables attachées aux transitions de la machine d'état finis qui formera un squelette de la partie contrôle. Les expressions affectées aux variables représentent le résultat de l'exécution des futures unités fonctionnelles employées sans faire aucune hypothèse sur leur nombre et le réseau de communication entre les unités fonctionnelles et les variables mises en oeuvre comme registres.

L'inconvénient est que selon ce modèle, les variables ne peuvent pas prendre les valeurs des entrées puisque les expressions sont définies seulement sur l'ensemble VAR, c'est à dire que les variables ne peuvent pas être affectées de l'extérieur. En outre, mathématiquement, la définition des relations logiques $STAT$ n'est pas claire, ainsi que le produit entre ces relations $STAT$ et l'ensemble des entrées I. La même question se pose pour le produit entre l'ensemble des sorties O et l'ensemble des affectations A. Pour pallier ces défauts et pour rendre le modèle mathématiquement cohérent, nous avons redéfini le modèle FSMD tout en gardant l'idée principale: la réduction du nombre d'états de contrôle par l'introduction des variables mémorisées. D'où la définition de la machine abstraite, comme le précise le prochain paragraphe.

3.2 Définition formelle du modèle de la machine abstraite

Nous adoptons les notations suivantes:

- S est l'ensemble des états de contrôle,

- I est l'ensemble des symboles des entrées, formé par les valeurs des signaux des entrées. Le symbole d'entrée i est présenté par le vecteur des signaux d'entrées $(i_1, i_2, ..., i_l)$:
  $i = (i_1, i_2, ..., i_l) \in IVAL_1 \times IVAL_2 \times ... \times IVAL_l = I$
  où $IVAL_j$ est l'ensemble de toutes les valeurs possibles de l'entrée i_j d'un circuit. Le but d'une telle notation est la distinction claire entre les entrées d'un circuit vues par le concepteur et les symboles d'entrées d'une machine d'états finis classique. Cette notation permettra ensuite d'utiliser les variables $i_1, i_2, ..., i_l$ comme des unités distinctes dans les définitions ultérieures.
  Exemple.
  Supposons un circuit (comme il est vu par le concepteur) ayant deux entrées a et b, telles que
  $a \in IVAL_a = \{Off, On\}$, et

$b \in IVAL_b = \{0, 1, 2, 3, ..., 15\}$.
Alors le symbole d'entrée i représenté par le vecteur (a, b) est un des couples suivants de l'ensemble I:
$i = (a, b) \in \{(Off, 0), (On, 0), (Off, 1), (On, 1), ..., (Off, 15), (On, 15)\} = I$

- O est l'ensemble des symboles des sorties, formé par les valeurs des signaux de sorties.
  $o = (o_1, o_2, ..., o_m) \in OVAL_1 \times OVAL_2 \times ... \times OVAL_m = O$
  où $OVAL_j$ est l'ensemble de toutes les valeurs possibles de la sortie o_j. La notation est la même que pour l'ensemble d'entrées.

- V est l'ensemble des symboles des variables mémorisées.
  $v = (v_1, v_2, ..., v_n) \in VVAL_1 \times VVAL_2 \times ... \times VVAL_n = V$
  où $VVAL_j$ est l'ensemble de toutes les valeurs correspondantes à la variable v_j.

- $STATUS$ est l'ensemble des symboles des statuts formés par les valeurs des signaux des statuts. Chaque signal de statut correspond à un compte rendu réel du chemin de données. Ce signal est examiné par le contrôleur pour déterminer son prochain état.
  $status = (stat_1, stat_2, ..., stat_q) \in STAT_1 \times STAT_2 \times ... \times STAT_q = STATUS$

- $ASSIGN$ est l'ensemble des affectations des variables et des signaux de sortie. Il est défini comme l'ensemble des fonctions de $I \times V$ vers $V^{assign} \times O^{assign}$ où V^{assign} et O^{assign} sont des ensembles tels que $V \subseteq V^{assign}$, $O \subseteq O^{assign}$ [1]. Un élément de l'ensemble $ASSIGN$ est donc une fonction fun_assign de signature $I \times V \rightarrow V^{assign} \times O^{assign}$:
  $fun\_assign \in ASSIGN = \{I \times V \rightarrow V^{assign} \times O^{assign}\}$; $V \subseteq V^{assign} \wedge O \subseteq O^{assign}$.
  En fait, chaque fonction fun_assign est un vecteur [2] de fonctions, une pour chaque variable mémorisée et une pour chaque sortie:
  $fun\_assign = (fun\_ass\_v_1, ..., fun\_ass\_v_n, fun\_ass\_o_1, ..., fun\_ass\_o_m)$ où
  $fun\_ass\_v_i \in ASS\_V_i = \{I \times V \rightarrow VVAL_i^{assign}\}$; $VVAL_i \subseteq VVAL_i^{assign} \quad (1 \leq i \leq n)$ et
  $fun\_ass\_o_j \in ASS\_O_j = \{I \times V \rightarrow OVAL_j^{assign}\}$; $OVAL_j \subseteq OVAL_j^{assign} \quad (1 \leq j \leq m)$.
  Comme le produit $V^{assign} \times O^{assign}$, les ensembles $VVAL_i^{assign}$ et $OVAL_j^{assign}$ sont définis par l'utilisateur et sont plus grand que $VVAL_i$ et $OVAL_j$.

Étant données les définitions précédentes, une machine abstraite est définie par un n-tuplet:

1. Lorsque on a une instruction VHDL conditionnelle ***if*** *cond* ***then*** *v1:=exp1;* ***else*** *v2:=exp2;* ***end if***; expressions $exp1$ et $exp2$ sont formalisées par des fonctions f_1 et f_2 dont les domaines de valeurs $VVAL_1^{assign}$ et $VVAL_2^{assign}$ peuvent être plus grand que les domaines V_1 de $v1$ et V_2 de $v2$. La condition *cond*, formalisée par un statut, peut faire en sorte de ne prendre en compte dans l'affectation qu'un sous ensemble de $VVAL_1^{assign}$ et $VVAL_2^{assign}$. C'est pourquoi nous avons $V \subseteq V^{assign}$, $O \subseteq O^{assign}$. Un exemple complet est donné à la page suivante.

2. En toute rigueur cette définition n'est pas correcte du point de vue des règles de typage ([HV92], [WL93]), où le type d'un vecteur est le produit cartésien des types de ses éléments. Dans notre cas, les éléments du vecteur sont des fonctions, c'est-à-dire du type fonctionnel:
$(f_1, f_2, ..., f_n)$ où $f_1 \in A \times B \rightarrow C_1$, $f_2 \in A \times B \rightarrow C_2$, ..., $f_n \in A \times B \rightarrow C_n$, et le type du produit est
$(A \times B \rightarrow C_1) \times (A \times B \rightarrow C_2) \times ... \times (A \times B \rightarrow C_n)$.
Pour passer de ce type au type $A \times B \rightarrow C_1 \times C_2 \times ... \times C_n$ il faut appliquer un opérateur d'ordre supérieur de transformation de type:
$transform : ((A \times B \rightarrow C_1) \times (A \times B \rightarrow C_2) \times ... \times (A \times B \rightarrow C_n)) \rightarrow (A \times B \rightarrow C_1 \times C_2 \times ... \times C_n)$.
Pour alléger les écritures, et parce qu'il n'y a aucune ambiguïté, l'application de cet opération sera omise dans cette thèse, et on admettra l'abus d'écriture qui consiste à identifier les deux types.

$$< S, I, O, V, STATUS, ASSIGN, fun\_status, f, h >$$

où

- fun_status est une fonction qui définit les valeurs des signaux des statuts à partir des valeurs des entrées et des variables mémorisées. En fait, la fonction fun_status, comme une des fonctions fun_assign, est un vecteur[3] des fonctions, une pour chaque statut:
  $fun\_status = (fun\_stat_1, fun\_stat_2, ..., fun\_stat_q) : I \times V \rightarrow STATUS$ où
  $fun\_stat_k : I \times V \rightarrow STAT_k \ \ (1 \leq k \leq q)$.

- f est une fonction de transition où les entrées primaires sont remplacées par les signaux de statut:
  $f : STATUS \times S \rightarrow S$.

- h est une fonction de sortie où les entrées primaires sont remplacées par les signaux de statut et les sorties primaires sont remplacées par une fonction $fun\_assign \in ASSIGN$:
  $h : STATUS \times S \rightarrow ASSIGN$.

Commentaire

La formalisation de la machine abstraite est fondée sur le fait que la représentation d'une fonction en intention (par formule mathématique) est beaucoup plus avantageuse que la représentation en extension (par énumération explicite). Par exemple, les fonctions f_1 et f_2 peuvent être définies sur l'intervalle entier [1,4] aussi bien par les tableaux 3.1.b et 3.1.c, que par les formules $f_1(i) = i-2$ et $f_2(i) = i+2$. Les formules ont l'avantage d'être concises et de fournir un moyen de calcul du résultat à partir des valeurs des arguments. En outre, les formules permettent d'introduire une notion proche du domaine informatique: la représentation SYMBOLIQUE. Une fonction devient une expression symbolique construite à l'aide des symboles associés aux arguments. Ainsi, $f_1(i)$ est l'expression symbolique $i-2$ ou i est le symbole représentant *toutes* les valeurs de l'argument de la fonction f_1.

Plusieurs formules peuvent être employées si le domaine de la fonction est divisée en blocs. Ainsi, on remarque que $f(i) = f_1(i) = i-2$ si $i < 3$ et $f(i) = f_2(i) = i+2$ sinon (tableau 3.1.a). Cette représentation de la fonction f sous une forme symbolique, appelée $f^{symbolic}$ (figure 3.1), est l'objet de la définition formelle et constitue une base pour la formalisation de la machine abstraite.

```
if (i<3)  then o=i-2
          else o=i+2
```

FIG. 3.1 – *Représentation symbolique $f^{symbolic}$ de la fonction f*

3. La remarque sur le type de la fonction fun_assign s'applique aussi sur le type de la fonction fun_status.

i	$f(i)$
1	-1
2	0
3	5
4	6
a.	

i	$f_1(i)$
1	-1
2	0
3	1
4	2
b.	

i	$f_2(i)$
1	3
2	4
3	5
4	6
c.	

TAB. 3.1 – *Définitions énumératives des fonctions* f ***(a.)***, f_1 ***(b.)***, *et* f_2 ***(c.)***.

Le but de la formalisation de la fonction $f^{symbolic}$est la définition des signatures nécessaires et la preuve d'équivalence entre les fonctions $f^{symbolic}$ et f. Pour faciliter la compréhension, ci-après nous donnons brièvement les définitions essentielles des règles de typage ([WL93], [Gor88], [Har98]).

En mathématiques, une fonction $g^{decur} : A \times B \to C$, par exemple $g^{decur}(a, b) = a + b$, est souvent dite posséder deux arguments a et b. En théorie des types une telle fonction est une fonction à un seul argument qui est une paire (a, b). L'appellation *fonction à deux arguments* est réservée à une fonction $g^{cur} : A \to B \to C$ qui, appliquée à un premier argument $a \in A$ rend la fonction $B \to C$, qui, à son tour appliquée à un deuxième argument $b \in B$, rend l'élément $c \in C$. Par exemple, la fonction g^{cur} associe à l'élément 1 la fonction $b + 1$, à l'élément 2 la fonction $b + 2$, etc. On note $g^{cur}(1) = b + 1$, $g^{cur}(2) = b + 2$, etc. Nous voyons que les deux fonctions g^{cur} et g^{decur} appliquées à des éléments a et b rendent le même élément $c = a + b$. La fonction $g^{cur} : A \to B \to C$ est appelée la version *curryfiée*, et la fonction $g^{decur} : A \times B \to C$ est appelée la version *non curryfiée* de la fonction g. Pareillement, le passage de la fonction g^{decur} vers la fonction g^{cur} s'appelle *curryfication*, et le passage de la fonction g^{cur} vers la fonction g^{decur} s'appelle *décurryfication*. Par la suite, nous appelons fonction à deux arguments aussi bien g^{cur}que g^{decur}.

Reprenons notre exemple du tableau 3.1. La fonction f initiale est définie formellement comme $f : I \to O \,|\, I = \{1,2,3,4\} \wedge O = \{-1,0,5,6\}$. Les fonctions f_1, f_2 sont définies comme $I \to O^{assign} \,|\, I = \{1,2,3,4\} \wedge O^{assign} = \{-1,0,1,2,3,4,5,6\}$ où l'ensemble O^{assign} est l'union des images de $\{1,2,3,4\}$ par les fonctions f_1 et f_2. La formalisation de la fonction $f^{symbolic}$ nécessite la définition de deux fonctions supplémentaires.

La fonction $fun\_status : I \to STATUS$ met en correspondance les éléments de l'ensemble I et les éléments de l'ensemble $STATUS$. Dans notre exemple l'ensemble $STATUS$ est l'ensemble booléen $Bool = \{true, false\}$ et la fonction fun_status est une fonction logique $fun\_status = (i < 3)$, qui rend la valeur $true$ ou $false$ selon l'argument i. La définition de cette fonction par énumération est donnée dans le tableau 3.2.

i	$fun\_status(i)$
1	true
2	true
3	false
4	false

TAB. 3.2 – *Définition de la fonction* fun_status

La deuxième fonction est la fonction $h : STATUS \rightarrow (I \rightarrow O^{assign})$ qui aux éléments de l'ensemble $STATUS$ met en correspondance soit la fonction f_1, soit la fonction f_2. C'est une fonction d'ordre supérieur car elle rend comme résultat une fonction et non pas un simple élément. La définition de la fonction h sous une forme tabulaire est donnée dans la figure 3.2.a. La figure 3.2.b montre la même fonction où les fonctions f_1 et f_2 à leur tour sont représentées sous une forme tabulaire.

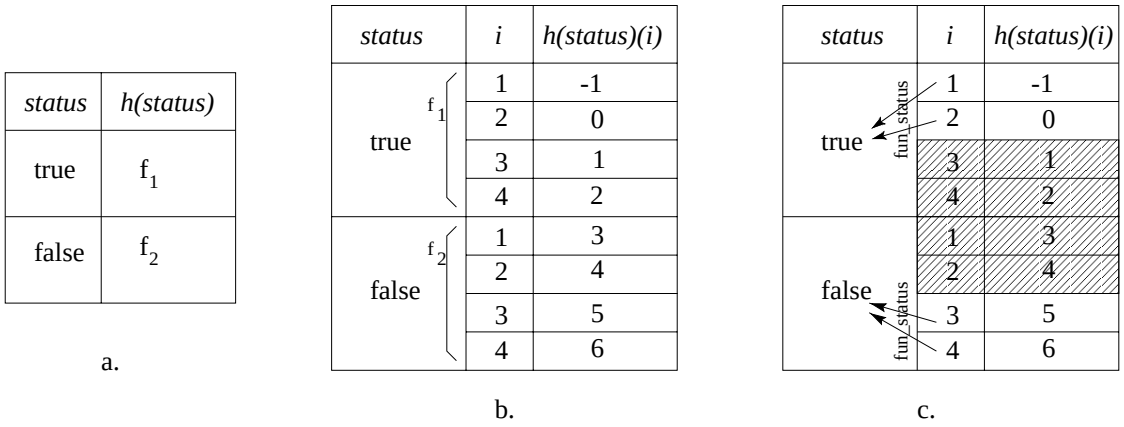


FIG. 3.2 – *Définition de la fonction h*

La fonction $f^{symbolic} : I \rightarrow O$ est définie alors à l'aide de la fonction h dont les éléments $status \in STATUS$ et $i \in I$ sont liés par la fonction fun_status:
$f^{symbolic}(i) = [h(status)](i) \mid_{status=fun\_status(i)} = [h(fun\_status(i))](i)$. Cette fonction $f^{symbolic}$ est équivalente à la fonction f initiale.

Le calcul d'élément $o = f^{symbolic}(i)$ au moyen de la fonction h est montré dans la figure 3.2.c: en appliquant la fonction fun_status à un élément de i on calcule d'abord l'élément de $status$ et ensuite, sachant les deux éléments $status$ et i, on calcule l'élément de o en appliquant la fonction h. Le lien entre les éléments $status$ et i explique, par ailleurs, pourquoi le domaine d'arrivée O de la fonction $f^{symbolic}$ est plus petit que le domaine d'arrivée O^{assign} de la fonction h: $O \subseteq O^{assign}$. Puisque les éléments $status$ et i sont liés par la fonction fun_status, certaines rangées de la fonction h (la figure 3.2.c), et par conséquent, certaines valeurs de o sont exclues. Pour cette raison $[h(status)](i) \mid_{status=fun\_status(i)} \in O$, $O \subseteq O^{assign}$.

En fait, la fonction $f^{symbolic}$ est une composition des fonctions h et fun_status, où les éléments de I sont liés par la relation d'identité. La composition des fonctions $fun\_status : I \rightarrow STATUS$ et $h : STATUS \rightarrow (I \rightarrow O^{assign})$, notée $h \circ fun\_status$, est une fonction $f^{composition} : I \rightarrow (I \rightarrow O^{assign})$ définie par $f^{composition}(i) = h(fun\_status(i))$. La fonction $f^{composition}$, appliquée à l'argument i_1 rend une fonction de signature $I \rightarrow O^{assign}$. Cette fonction, appliquée à son tour à l'argument i_2 rend l'élément o^{ass}. La fonction $f^{symbolic}$ est définie par la fonction $f^{composition}$ restreinte au cas où le premier ($i_1 \in I$) et le deuxième ($i_2 \in I$) argument sont égaux:
$f^{symbolic}(i) = [h(fun\_status(i))](i) = [f^{composition}(i)](i)$.

Nous voyons que la fonction $f^{symbolic}$ est définie à la fois par la fonction f à un argument et par la fonction $f^{composition}$ à deux arguments, lorsque les arguments sont égaux. La relation de bijection entre l'image de la fonction f et l'image de la fonction $f^{composition}$restreinte aux

arguments $i_1 \in I$ et $i_2 \in I$ tels que $i_1 = i_2$ est montrée sur la figure 3.3.

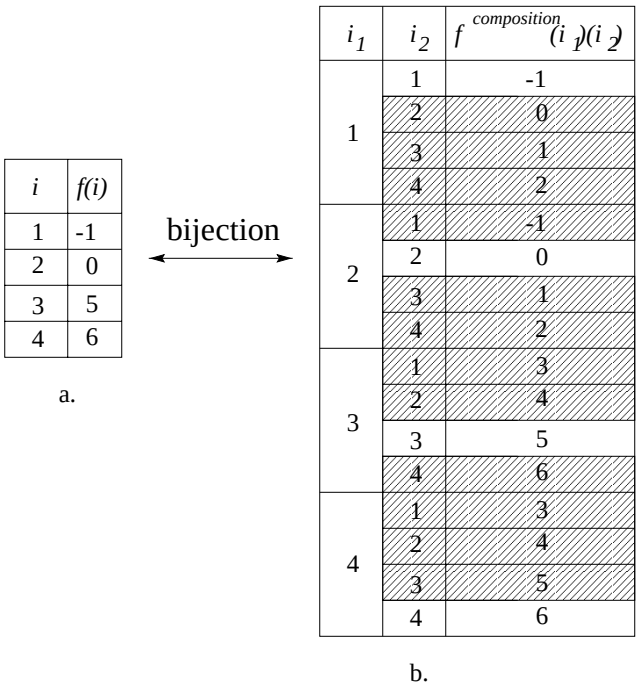


FIG. 3.3 – *Bijection entre les fonctions f et $f^{composition}$ restreinte*

Nous avons montré que la version symbolique $f^{symbolic}$ (la figure 3.1) de la fonction f (le tableau 3.1.a) est la composition de deux fonctions:

1. la fonction $fun\_status : I \rightarrow STATUS$ et
2. la fonction $h : STATUS \rightarrow (I \rightarrow O^{assign})$

Ultérieurement, les fonctions fun_status et h seront présentées exclusivement dans une forme compacte: les définitions par énumération seront remplacées par les formules mathématiques chaque fois que c'est possible. Ainsi, les fonctions fun_status et h de la fonction $f^{symbolic}$ prennent la forme illustrée par la figure 3.4. La valeur $f^{symbolic}(i) = [h(fun\_status(i))](i)$ est utilisée pour calculer une valeur particulière, si nécessaire.

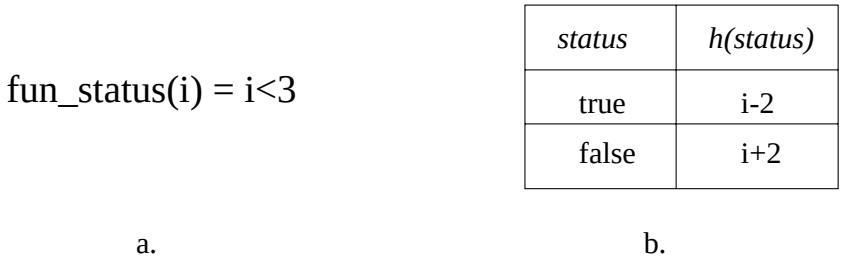


FIG. 3.4 – *Représentation en intention des fonctions fun_status **(a.)** et h **(b.)***

L'intérêt de la fonction $f^{symbolic}$ est d'être en liaison directe avec la description algorithmique de circuits intégrés. Par exemple, la figure 3.1 est extraite d'une description comportementale d'un circuit. La construction de la fonction $f^{symbolic}$ est la base de la formalisation de la machine

abstraite issue d'une telle description . Dans notre exemple les fonctions f_1 et f_2 correspondent aux éléments de l'ensemble $ASSIGN$ dans la formalisation de la machine abstraite. Dans la prochaine section nous allons montrer que la composition des fonctions de la machine abstraite fun_status et h est équivalente à la fonction que nous appellerons $\lambda : I \times V \times S \rightarrow V \times O$. Cette fonction λ (fonction de sortie) nous permettra de transformer formellement le modèle de la machine abstraite vers le modèle de la machine d'états finis classique.
Fin de commentaire

La machine abstraite peut être vue comme un "circuit" (figure 3.5) avec les "fils" qui portent l'information en direction des flèches, les "boites" de logique combinatoire pour calculer les fonctions fun_status, fun_assign, f, h, et les éléments de mémoire pour stocker les variables mémorisées V et les états S de la machine abstraite.

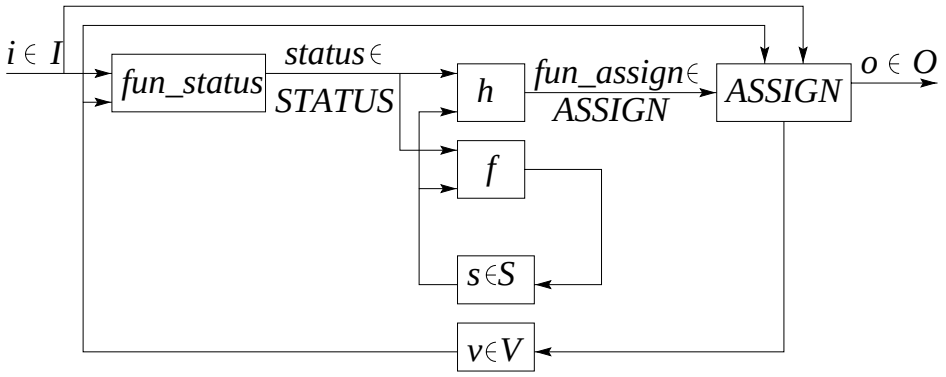


FIG. 3.5 – *Vue schématique de la machine abstraite*

Les fonctions f et h peuvent être considérées comme les fonctions analogues aux fonctions de transition et de sortie de la machine d'états finis classique où les entrées primaires ($i \in I$) sont remplacées par les signaux de statut ($status \in STATUS$) et les sorties primaires ($o \in O$) sont remplacées par les fonctions ($fun\_assign \in ASSIGN$) qui elles, définissent les valeurs des sorties primaires et des variables internes. La raison d'une telle définition complexe de la machine abstraite est la concision de la représentation: la fonction de sortie de la machine abstraite est "divisée" en "morceaux" pour lesquels la représentation sous forme d'une formule est possible. Une formule permet d'exprimer les sorties et les variables mémorisées comme des expressions symboliques construites à l'aide des symboles associés aux entrées et aux valeurs précédentes des variables. Les "morceaux" de la fonction de sortie sont définis par la fonction fun_status.

En fait, le modèle de la machine abstraite n'est rien d'autre qu'une représentation concise de la machine d'états finis classique: le nombre d'états est plus petit, et les fonctions sont présentées en intention et non en extension. La relation d'équivalence entre les deux modèles est explorée dans la prochaine section.

3.3 Comparaison avec la machine d'états finis classique

Cette section est dédiée à la comparaison de la machine abstraite avec la machine d'états finis classique. Cette comparaison a un double intérêt. D'une part, la machine d'états finis classique est largement reconnue comme modèle formel et la comparaison avec elle représente un résultat théorique indépendant. D'autre part, un grand nombre d'outils (de synthèse, de vérification) sont basés sur la machine classique. Pour étudier la possibilité d'adaptation de ces outils pour le modèle de la machine abstraite il est nécessaire de faire une comparaison et trouver éventuellement des règles de transformations entre les deux modèles. Les règles de transformations peuvent servir, entre autres, pour la synthèse de la machine classique à partir de la machine abstraite ou bien pour l'abstraction de la machine classique vers la machine abstraite.

Pour montrer l'équivalence entre la machine abstraite et la machine d'états finis classique, nous avons besoin d'étudier plus à fond le modèle de la machine abstraite. Tout d'abord, précisons le contexte dans lequel nous nous plaçons. Nous supposons que dans le modèle de la machine abstraite les ensembles I, O, V représentent un nombre *fini* de *toutes* les valeurs physiquement réalisables par un circuit réel. Ainsi le modèle de la machine abstraite fait une toute première liaison entre une description comportementale avec des variables dont les types sont en principe infinis (le type des entiers, des naturels, etc.) et une structure d'un circuit réel dans lequel cette infinité est, aussi en principe, impossible. Cette liaison est établie par le concepteur lors des décisions fondamentales sur la largeur des vecteurs de bits nécessaires pour réaliser les entrées, les sorties et les variables mémorisées du futur circuit.

Si ces principales décisions architecturales ne sont pas encore prises, le modèle de la machine abstraite correspond à une description comportementale avec les ensembles I, O, V infinis, identiques aux types de la description-source VHDL. Dans ce cas $ASSIGN$ est défini comme un ensemble de fonctions de signature $I \times V \rightarrow V \times O$: $fun\_assign \in ASSIGN = \{I \times V \rightarrow V \times O\}$. Nous utiliserons cette variante de la machine abstraite pour la construction du modèle correspondant à la description initiale comportementale et pour la vérification de l'étape d'ordonnancement car à ce stade de la conception, les décisions architecturales ne sont pas encore pertinentes.

Reprenons les définitions générales des trois fonctions de la machine abstraite fun_status, f et h :

- $fun\_status : I \times V \rightarrow STATUS$
- $f : STATUS \times S \rightarrow S$
- $h : STATUS \times S \rightarrow ASSIGN$ ou bien, en remplaçant $ASSIGN$ par sa définition
  $h : STATUS \times S \rightarrow (I \times V \rightarrow V^{assign} \times O^{assign}) \mid V \subseteq V^{assign}, O \subseteq O^{assign}$

Calculons ensuite le prochain état (s') et la sortie de la machine abstraite (un couple (v', o)) en substituant $status$ par $fun\_status(i, v)$ (puisque $status = fun\_status(i, v)$) :

- $s' = f(status, s) = f(fun\_status(i, v), s) = \delta(i, v, s)$
  où $\delta : I \times V \times S \rightarrow S$ est la fonction dont la signature est définie par les types de ses arguments (la partie gauche de la signature) et le type du résultat qui est le même que le type du résultat de la fonction f (la partie droite de la signature)

- $(v', o) = [h(status, s)](i, v) = [h(fun\_status(i, v), s)](i, v) = \lambda(i, v, s)$
  où $\lambda : I \times V \times S \to V^{asign} \times O^{assign}$ est la fonction dont la signature est définie pareillement

Cette simple manipulation mathématique nous donne l'intuition du calcul du prochain état et de la sortie à partir de l'entrée primaire, de l'état courant, et des variables mémorisées de la machine abstraite. Cependant, le fait que les fonctions δ et λ sont les compositions des fonctions f et fun_status (δ) et h et fun_statut (λ) n'est pas évident. De plus, la fonction λ telle, que nous l'avons déduite, n'est pas réalisable par un circuit réel. En effet, selon la définition $V \subseteq V^{assign}$. La situation suivante peut donc avoir lieu: $\lambda(i, v, s) = (v', o) \mid \ v' \in V^{assign}$ et $v' \notin V$. Or si l'ensemble V représente toutes les valeurs possibles des variables mémorisées définies par les registres/mémoires d'un circuit réel, v' n'a pas d'analogue physique.

Un exemple d'une telle incohérence est la fonction λ suivante (supposons que le modèle a une entrée i, une sortie o, une variable v et que s dans le cas précis représente un état quelconque du modèle): $\lambda(i, v, s) = (v + 1, i + 2)$. La variable v est toujours incrementée, indépendamment de sa valeur initiale. Or quel que soit le registre attribué à cette variable, il ne peut avoir qu'un nombre fini de valeurs. Ainsi $(v + 1)$ sera "coupé" quand v atteint la valeur maximum de ce registre.

La fonction $\lambda : I \times V \times S \to V^{asign} \times O^{assign}$ sera réalisable si $V^{assign} = V$. Dans ce cas le résultat de la fonction λ (plus précisément la partie v' du couple (v', o)) a toujours un analogue physique représenté par un élément de l'ensemble V. Par la suite, nous calculons le prochain état et la sortie de la machine abstraite en composant les fonctions f et h avec la fonction fun_status et donnons aussi les conditions de la "faisabilité" de la fonction λ. Pour que la composition soit mathématiquement correcte, réécrivons les fonctions f et h par leur forme curryfiée:

- $f^{cur} : STATUS \to S \to S$.

- $h^{cur} : STATUS \to S \to (I \times V \to V^{assign} \times O^{assign})$ $(V \subseteq V^{assign},\ O \subseteq O^{assign})$

Le prochain état (s') et la sortie de la machine abstraite (v', o) sont calculés alors par les formules suivantes:

- $s' = [f^{cur}(status)](s) = [f^{cur}(fun\_status(i, v)](s) = [\delta^{cur}(i, v)](s) = \delta(i, v, s)$
  où $\delta^{cur} = f^{cur} \circ fun\_status : I \times V \to S \to S$ est la composition des fonctions f^{cur} et fun_status, et $\delta : I \times V \times S \to S$ est la version non curryfiée de la composition.

- $(v', o) = [[h^{cur}(status)](s)](i, v) = [[h^{cur}(fun\_status(i, v))](s)](i, v) =$
  $[[\lambda^{composition-cur}(i, v)](s)](i, v) = [\lambda^{composition}(i, v, s)](i, v) =$
  $[\lambda^{composition}(i_1, v_1, s)](i_2, v_2)|_{i_1=i_2=i,\ v_1=v_2=v} = \lambda(i, v, s)$
  où $\lambda^{composition-cur} = h^{cur} \circ fun\_status : I \times V \to S \to I \times V \to V^{assign} \times O^{assign}$ est la composition des fonctions h^{cur} et fun_status, $\lambda^{composition} : I \times V \times S \to I \times V \to V^{assign} \times O^{assign}$ est la version non curryfiée de la composition , et $\lambda : I \times V \times S \to V^{\lambda} \times O$ $(V^{\lambda} \subseteq V^{assign},\ O \subseteq O^{assign})$ est la restriction de $\lambda^{composition}$ aux arguments i_1, v_1, s, i_2, v_2 tels que $i_1 = i_2$ et $v_1 = v_2$.

Nous voyons que la sortie de la machine abstraite est définie par une fonction $\lambda^{composition}$ dont certains arguments sont égaux. Cette fonction est analogue à la fonction $f^{composition}$ du commentaire précédent. Toujours par analogie, il existe alors une fonction $\lambda : I \times V \times S \rightarrow V^{\lambda} \times O$ ($V^{\lambda} \subseteq V^{assign}$, $O \subseteq O^{assign}$) telle que $[\lambda^{composition}(i,v,s)](i,v) = \lambda(i,v,s)$ pour toutes valeurs de i, v et s. Au début de ce paragraphe nous avons défini la fonction λ physiquement réalisable si l'ensemble résultant des variables mémorisées est équivalent à l'ensemble initial de ces même variables: $V^{\lambda} = V$. Cette condition sera satisfaite si nous imposons explicitement les restrictions sur la fonction h^{cur} car la fonction λ en est déduite.

En effet, $(v', o) = [h^{cur}(status)](s)](i,v) = \lambda(i,v,s)$. Pour que $\lambda : I \times V \times S \rightarrow V \times O$, h^{cur} doit être restreinte de telle façon que $[h^{cur}(status)](s)](i,v) \in V \times O$. Or, λ est une composition de h^{cur} et fun_status, la restriction sur h^{cur} doit être imposée seulement pour tous les i, v et $status$ tels que $status = fun\_status(i,v)$. La restriction $[h^{cur}(status)](s)](i,v) \in V \times O$ signifie en d'autres termes que l'application de la fonction fun_assign aux arguments i et v tels que $status = fun\_status(i,v)$ et $h(status, s) = fun\_assign$, reste toujours dans les limites du produit $V \times O$. Ce postulat est formulé par le théorème suivant.

Théorème: Si la fonction h est définie de telle façon que $h(status, s) = fun\_assign$, $fun\_assign(i,v) \in V \times O$ pour toutes les valeurs de i et de v telles que $status = fun\_status(i,v)$, alors $V^{\lambda} = V$ et $\lambda : I \times V \times S \rightarrow V \times O$.

Preuve: La restriction sur la fonction h se transforme pour la fonction h^{cur}dans la restriction suivante: la fonction h^{cur} est définie de telle façon que $[h^{cur}(status)](s) = fun\_assign$, $fun\_assign(i,v) \in V \times O$ pour toutes les valeurs de i et de v telles que $status = fun\_status(i,v)$; ou bien encore dans la restriction: $[[h^{cur}(status)](s)]\,(i,v)|_{status=fun\_status(i,v)} \in V \times O$.

Ensuite la preuve est directe car la restriction $status = fun\_status(i,v)$ est satisfaite par la définition de la fonction fun_status et, en conséquence $[h^{cur}(status)](s)]\,(i,v)|_{status=fun\_status(i,v)} = h^{cur}(status)](s)]\,(i,v) = \lambda(i,v,s)$.

La dernière égalité implique que pour tous les triplets (i,v,s), $\lambda(i,v,s) \in V \times O$, et que $V \times O$ est donc $V^{\lambda} = V$. $\Diamond$

Ce théorème signifie que pas toutes les machines abstraites sont formellement synthétisables. Certaines fonctions h dont la composition avec la fonction fun_status aboutit à une fonction $\lambda : I \times V \times S \rightarrow V^{\lambda} \times O \quad (V^{\lambda} \neq V)$ ne peuvent pas être implémentées en principe. Lors de la synthèse classique, néanmoins, la condition de la faisabilité n'est pas vérifiée. Par exemple, une machine avec la fonction λ dont l'extrait a été mentionné plus tôt: $\lambda(i,v,s) = (v+1, i+2)$, sera synthétisée par les outils de la synthèse conventionnels. Le registre attribué à la variable v recevra toujours une valeur $v+1$ au cycle d'horloge suivant (bien sûr par l'intermédiaire d'un additionneur et d'un réseau de connexion). Cependant, pour la valeur maximum permise par le registre, le fonctionnement du circuit synthétisé ne satisfera pas la spécification initiale ($v+1$ ne peut pas être réalisé).

Commentaire

Nous avons déjà mentionné que les fonctions $\lambda^{composition}$ et λ sont analogues aux fonctions $f^{composition}$ et f du commentaire précédent. Comme le montre la figure 3.3, la fonction $f^{composition}$:

$I \to I \to O^{assign}$ partiellement définie sur i_1, i_2 tels que $i_1 = i_2$ est équivalente à la fonction $f : I \to O$. L'image O de la fonction f est plus petite que l'image O^{assign} de la fonction $f^{composition}$ car certaines rangées de la fonction $f^{composition}$ ne sont pas considérées en raison de la restriction sur les arguments i_1 et i_2. Pour la même raison l'image $V \times O$ de la fonction λ est plus petite que l'image $V^{assign} \times O^{assign}$ de la fonction $\lambda^{composition}$: la restriction sur les arguments i_1, v_1, s, i_2, v_2 de la fonction $\lambda^{composition}$ exclut certaines valeurs $\lambda^{composition}(i_1, v_1, s, i_2, v_2)$.
Fin de commentaire

Compte tenu des nouvelles fonctions δ et λ, la machine abstraite peut être considérée comme un "circuit" avec les entrées I, les sorties O, les deux éléments de mémoire S et V et les deux fonctions δ et λ qui définissent les valeurs des sorties et des éléments mémorisés. Formellement, la machine abstraite est redéfinie par le n-tuplet:

$$< S, I, O, V, \delta, \lambda >$$

où

- S est l'ensemble des états de la partie contrôle
- I est l'ensemble des symboles des entrées
- O est l'ensemble des symboles des sorties
- V est l'ensemble des symboles des variables mémorisées
- δ est une fonction de transition:
  $\delta : I \times V \times S \mapsto S$
- λ est une fonction de sortie:
  $\lambda : I \times V \times S \mapsto V \times O$

La nouvelle définition est en quelque sorte une abstraction de la définition initiale: les "détails" de la définition initiale tels que les fonctions fun_status, f, h et l'ensemble $STATUS$ sont "cachés"et "englobés" par les fonctions δ et λ dont "le niveau d'abstraction" est plus haut que celui des fonctions fun_status, f, et h. L'évolution de la machine abstraite en vue de la nouvelle définition est montrée dans la figure 3.6.b.

Pour démontrer l'équivalence entre la machine abstraite et la machine d'états finis classique, modifions les fonctions δ et λ en plusieurs étapes:

- concaténation des deux fonctions qui consiste à calculer le vecteur [4]des fonctions δ et λ; cette opération aboutit à une nouvelle fonction δ_and_λ (la figure 3.6.c):
  $(\delta, \lambda) = \delta\_and\_\lambda : I \times V \times S \mapsto S \times V \times O$
- projections de la fonction δ_and_λ sur $S \times V$ et sur O, produisant les fonctions δ' et λ' telles que $(\delta', \lambda') = \delta\_and\_\lambda$ (la figure 3.6.d) :
  $\delta' : I \times V \times S \mapsto S \times V$
  $\lambda' : I \times V \times S \mapsto O$

– considération du produit $V \times S$ comme un nouvel ensemble (la figure 3.6.e) et renommage de $V \times S$ par S_{cl} (cl pour "classique"), δ' par δ_{cl}, et λ' par λ_{cl} (la figure 3.6.f):
$\delta_{cl} : I \times S_{cl} \mapsto S_{cl}$ [5]
$\lambda_{cl} : I \times S_{cl} \mapsto O$

Après les modifications énumérées ci-dessus, la machine abstraite devient exactement la machine d'états finis classique où l'ensemble des états inclut l'ensemble des valeurs des variables internes mémorisées: $S_{cl} = V \times S$ (la figure 3.6.f).

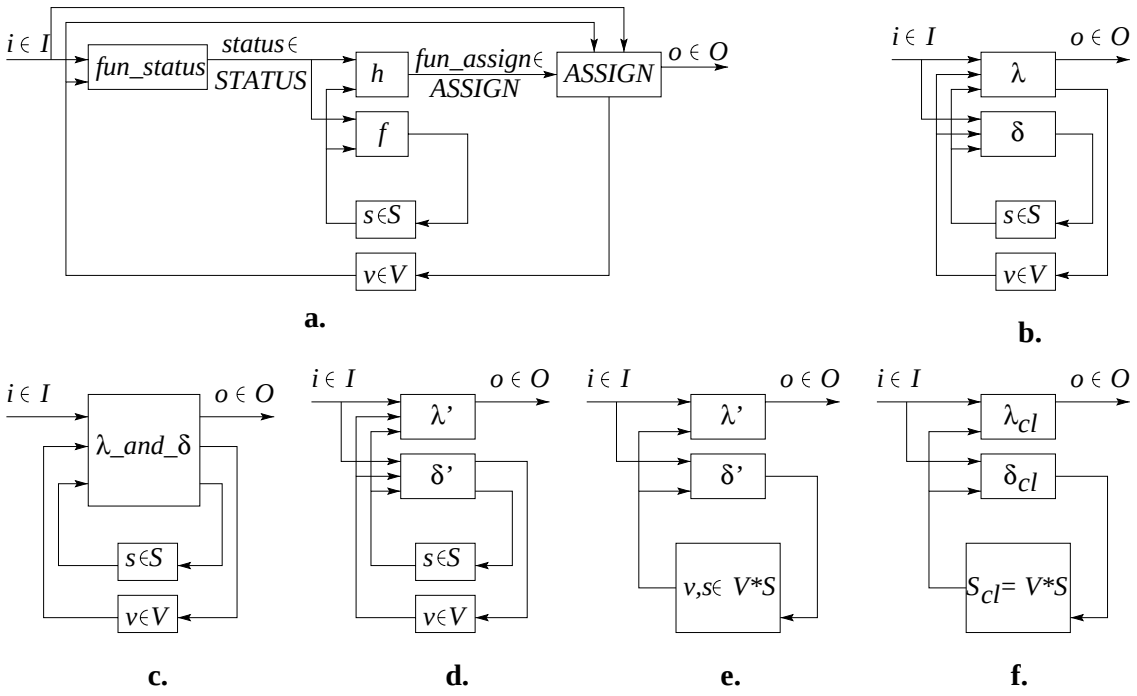


FIG. 3.6 – *Évolution de la machine abstraite*

La machine abstraite n'est donc rien d'autre que la machine d'états finis classique de Mealy modifiée. D'abord on divise l'ensemble des états de contrôle en deux sous-ensembles. Ensuite, on redistribue le calcul des sorties et des nouveaux états entre la fonction de transition et la fonction de sortie de telle façon que la fonction de transition calcule seulement un des sous-ensembles d'état et la fonction de sortie calcule les sorties et le deuxième sous-ensemble d'état. Finalement, les nouvelles fonctions de transition et de sortie sont représentées comme les compositions des fonctions fun_status, f, et h, où la fonction h définit les sorties sous une forme symbolique à l'aide des fonctions $ASSIGN$.

L'évolution de la machine abstraite illustrée par la figure 3.6 peut être considérée comme une synthèse "théorique" au niveau du modèle. En effet, la "synthèse" commence par une description comportementale, où le flot du programme est gouverné par les relations booléennes intrinsèques à la description algorithmique, telles que, par exemple, $input\_a \leq 5$ ou bien $input\_a \neq register\_b$. Les relations booléennes sont représentées par les fonctions fun_stat_1, fun_stat_2, ..., fun_stat_q

4. Ici encore nous identifions les types $(I \times V \times S \mapsto S) \times (I \times V \times S \mapsto V \times O)$ et $I \times V \times S \mapsto S \times V \times O$.
5. La fonction $\delta' : I \times V \times S \mapsto S \times V$ a été réécrite en $\delta' : I \times V \times S \mapsto V \times S$ vu que l'ordre des composants des produits cartésiens $(V \times S)$ et $(S \times V)$ n'est pas important. En effet, les paires $(v, s) \in (V \times S)$ et $(s, v) \in (S \times V)$ représentent le même objet, contenant deux informations en provenance respectivement de V et de S.

et les résultats de ces fonctions remplacent les entrées primaires I dans les définitions des fonctions de transition et de sortie. Au niveau comportemental, les relations entre entrées primaires et non leurs valeurs particulières sont utilisées pour contrôler l'évolution du programme. Les sorties sont exprimées sous une forme symbolique, avec les symboles associés aux entrées et aux variables du programme. La "synthèse" aboutit à une description mise à plat (au niveau des portes) où les éléments mémorisants correspondant aux variables de la description initiale participent également à l'espace d'état du circuit. Ainsi, l'espace total des états devient le produit $V \times S$. Le comportement algorithmique est "dissout" dans le comportement du circuit au niveau des portes.

La synthèse "théorique", à condition que les algorithmes de la synthèse soient développés et la machine d'états finis classique soit convertie dans des équations booléennes, ouvre des perspectives intéressantes. Tout d'abord elle peut apporter une nouvelle approche pour la vérification des résultats de la synthèse logique (puisque la machine abstraite correspond à une description d'un circuit au niveau transfert de registres et la machine classique correspond à une description d'un circuit au niveau des portes). La machine d'états finis classique obtenue lors de la synthèse théorique, peut être considérée comme correcte par la construction formelle. On la compare alors avec celle obtenue lors de la synthèse conventionnelle, en vérifiant ainsi l'exactitude de la synthèse classique. En outre, la synthèse "théorique" peut apporter une nouvelle vision de la synthèse elle-même en utilisant les algorithmes formels au lieu des algorithmes heuristiques (l'allocation des opérations aux blocs fonctionnels, "binding", etc.).

La dernière remarque que nous faisons porte sur les niveaux d'abstraction du modèle et de la description d'un circuit. Pour le modèle, le niveau d'abstraction augmente à partir de la machine abstraite vers la machine d'états finis classique: le modèle devient de plus en plus "propre" et libre de détails "encombrants". En revanche pour la description, le niveau d'abstraction le plus haut correspond au modèle de la machine abstraite (niveau comportemental) et le niveau d'abstraction le plus bas correspond au modèle de la machine d'états finis classique (niveau logique). Ce phénomène est montré dans la figure 3.7.

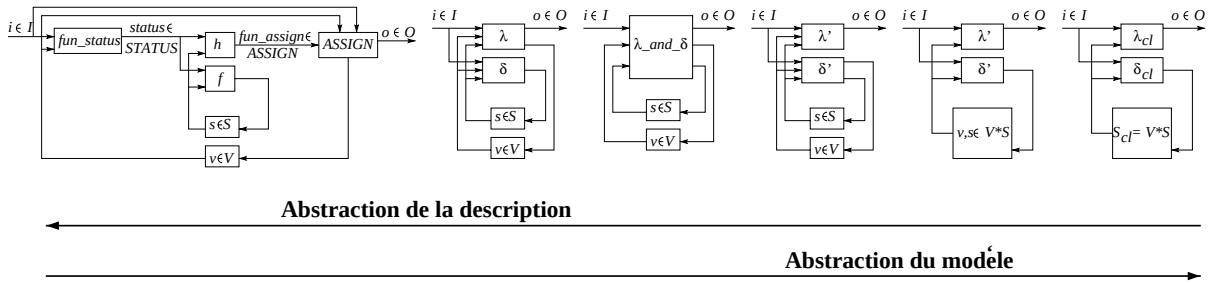


FIG. 3.7 – *Abstraction de la description et du modèle d'un circuit*

Dans la section suivante, un exemple de machine abstraite et son équivalent en machine d'états finis classique de Mealy sont présentés.

3.4 Exemple de machine abstraite

Pour montrer sur un exemple l'équivalence entre le modèle de machine abstraite et le modèle de Mealy de la machine d'états finis classique, partons de la machine classique dont l'espace d'état est divisé en deux sous-ensemble S et V, et voyons comment à partir de ce modèle on obtient la machine abstraite correspondante.

Supposons que les ensembles S, V, I, et O de la machine classique M_{cl} sont définis comme suit:

- $S = \{s_1, s_2\}$
- $V = \{1, 2, 3\}$
- $I = \{1, 2\}$
- $O = \{4, 5, 6\}$

Pour des raisons de simplicité les ensembles V, I, et O, généralement définis dans la machine abstraite comme les produits de sous-ensembles, sont restreints seulement à un sous-ensemble ($V = VVAL_1$, $I = IVAL_1$, $O = OVAL_1$). Par conséquent, leurs éléments, respectivement $v \in V$, $i \in I$, et $o \in O$, sont des valeurs scalaires et non vectorielles. Ce choix, cependant, n'affecte en aucune façon la généralité de l'exemple.

L'espace total des états de la machine classique $M_{cl} =< S \times V, I, O, \delta_{cl}, \lambda_{cl} >$ est le produit cartésien $S \times V = \{(s_1,1),(s_1,2),(s_1,3),(s_2,1),(s_2,2),(s_2,3)\}$. On note par (s, v) un état composé: $(s, v) \in S \times V$. Les fonctions de transition δ_{cl} et de sortie λ_{cl} sont définies à l'aide du tableau 3.8.a.

Dans le tableau 3.8.b le calcul du prochain état et de la sortie est redistribué entre la fonction de transition et la fonction de sortie: la fonction de transition calcule seulement la première projection d'état (s'); la fonction de sortie, elle, calcule la deuxième projection d'état (v') et la sortie (o). En outre, le tableau 3.8.b remplace la nouvelle fonction de sortie par la fonction $\lambda^{composition} : I \times V \times S \to I \times V \to V^{assign} \times O^{assign}$ restreinte aux arguments i_1, v_1, s, i_2, v_2 tels que $i_1 = i_2 = i$ et $v_1 = v_2 = v$. Le but de la fonction $\lambda^{composition}$ est l'expression de la sortie (couple (v', o)) sous une forme symbolique[6]. Par exemple, dans la troisième rangée du tableau 3.8.b, l'élément rendu par $\lambda^{composition}(i = 1, v = 2, s = s_1)$ est la fonction $fun\_assign : I \times V \to V^{assign} \times O^{assign}$ qui définit le couple (v', o) de la façon suivante: $(v', o) = ((2v - i), (3i + v))$.

Le tableau 3.8.c[7] réunit les rangées avec des expressions symboliques identiques. L'unification demande une re-considération des notions d'état et d'entrée du modèle. Après l'unification, seuls les éléments de l'ensemble S forment l'espace d'états, tandis que les éléments de l'ensemble V participent à la construction des nouvelles entrées définies comme les résultats des fonctions sur les ensembles I et V.

Finalement, le tableau 3.8.d représente la machine abstraite, qui introduit explicitement les nouvelles entrées $STATUS$. Formellement, la machine abstraite M_{ab}, obtenue à partir de la

6. Les constantes telles que 4 et 5 sont considérées comme un cas particulier de fonctions et, par conséquent, comme un cas particulier de forme symbolique.

7. Ici et ultérieurement le symbole $\overline{a}$ signifie la négation de variable booléenne a.

State / Input / Transition function / Output function

S	V	I	S	V	O
s_1	1	1	s_2	1	4
		2	s_2	1	4
s_1	2	1	s_2	3	5
		2	s_2	2	6
s_1	3	1	s_1	2	5
		2	s_1	2	5
s_2	1	1	s_1	2	4
		2	s_2	1	4
s_2	2	1	s_2	2	4
		2	s_1	3	6
s_2	3	1	s_2	3	4
		2	s_2	3	4

a.

State / Input / Transition function / Output function

S	V	I	S	V	O
s_1	1	1	s_2	2v- i	3i +v
		2	s_2	v	i +2v
s_1	2	1	s_2	2v- i	3i +v
		2	s_2	v	i +2v
s_1	3	1	s_1	2	5
		2	s_1	2	5
s_2	1	1	s_1	i+1	2v+2
		2	s_2	v	4
s_2	2	1	s_2	v	4
		2	s_1	i+1	2v+2
s_2	3	1	s_2	v	4
		2	s_2	v	4

b.

State / Input / Transition function / Output function

S	*Fun*(V, I)	S	V	O
s_1	v=3	s_1	2	5
	$\overline{\text{v=3}}$ & i =2	s_2	v	i +2v
	$\overline{\text{v=3}}$ & $\overline{\text{i =2}}$	s_2	2v-i	3i +v
s_2	v= i	s_1	i+1	2v+2
	$\overline{\text{v= i}}$	s_2	v	4

c.

State / Input / Transition function / Output function

S	STATUS	S	V	O
s_1	stat1	s_1	2	5
	$\overline{\text{stat1}}$ & stat2	s_2	v	i +2v
	$\overline{\text{stat1}}$ & $\overline{\text{stat2}}$	s_2	2v-i	3i +v
s_2	stat3	s_1	i+1	2v+2
	$\overline{\text{stat3}}$	s_2	v	4

stat1: v=3; stat2: i =2; stat3: v=i;

d.

FIG. 3.8 – *Relations entre la machine abstraite et la machine d'états finis classique de Mealy*

machine classique M_{cl}, est définie comme suit:

- S est l'ensemble des états de contrôle:
  $s \in S = \{s_1, s_2\}$

- I est l'ensemble des symboles de l'entrée:
  $i \in I = \{1, 2\}$

- O est l'ensemble des symboles de la sortie:
  $o \in O = \{4, 5, 6\}$

- V est l'ensemble des symboles de la variable mémorisée:
  $v \in V = \{1, 2, 3\}$

– $STATUS$ est l'ensemble des symboles des statuts:
$status = (stat_1, stat_2, stat_3) \in STATUS = Bool \times Bool \times Bool = Bool^3$

– $ASSIGN$ est l'ensemble des affectations de la variable v et de la sortie o:
$fun\_assign = (fun\_ass\_v,\ fun\_ass\_o) : I \times V \mapsto V^{assign} \times O^{assign}$ où
$fun\_ass\_v \in ASS\_V = \{fun\_ass\_v_1,\ fun\_ass\_v_2,\ fun\_ass\_v_3,\ fun\_ass\_v_4\}$
$fun\_ass\_o \in ASS\_O = \{fun\_ass\_o_1,\ fun\_ass\_o_2,\ fun\_ass\_o_3,\ fun\_ass\_o_4,\ fun\_ass\_o_5\}$
$fun\_assign \in ASSIGN = \{fun\_ass_1,\ fun\_ass_2,\ fun\_ass_3,\ fun\_ass_4,\ fun\_ass_5\}$

$fun\_ass\_v_1(i,v) = 2$	$fun\_ass\_o_1(i,v) = 5$	$fun\_ass_1 = (fun\_ass\_v_1,\ fun\_ass\_o_1)$
$fun\_ass\_v_2(i,v) = v$	$fun\_ass\_o_2(i,v) = i+2v$	$fun\_ass_2 = (fun\_ass\_v_2,\ fun\_ass\_o_2)$
$fun\_ass\_v_3(i,v) = 2v-i$	$fun\_ass\_o_3(i,v) = 3i+v$	$fun\_ass_3 = (fun\_ass\_v_3,\ fun\_ass\_o_3)$
$fun\_ass\_v_4(i,v) = i+1$	$fun\_ass\_o_4(i,v) = 2v+2$	$fun\_ass_4 = (fun\_ass\_v_4,\ fun\_ass\_o_4)$
	$fun\_ass\_o_5(i,v) = 4$	$fun\_ass_5 = (fun\_ass\_v_2,\ fun\_ass\_o_5)$

Étant données les définitions précédentes, la machine M_{ab} est définit par le n-tuplet:

$$M_{ab} =< S, I, O, V, STATUS, ASSIGN, fun\_status, f, h >$$

où

– fun_status est le vecteur de trois fonctions booléennes:
$fun\_status = (fun\_stat_1,\ fun\_stat_2,\ fun\_stat_3)$ où
$fun\_stat_1(i,v) = (v = 3);\ fun\_stat_2(i,v) = (i = 2);\ fun\_stat_3(i,v) = (v = i)$

– f est la fonction de transition:
$f(s_1,\ true, -, -) = s_1;$
$f(s_1,\ false, true, -) = s_2;$
$f(s_1,\ false, false, -) = s_2;$
$f(s_2,\ -, -true) = s_1;$
$f(s_2,\ -, -, false) = s_2$

– h est la fonction de sortie:
$h(s_1,\ true, -, -) = fun\_ass_1;$
$h(s_1,\ false, true, -) = fun\_ass_2;$
$h(s_1,\ false, false, -) = fun\_ass_3;$
$h(s_2,\ -, -, true) = fun\_ass_4;$
$h(s_2,\ -, -, false) = fun\_ass_5;$

Pour la fonction de transition $f\ (s, stat_1, stat_2, stat_3)$ et la fonction de sortie $h\ (s, stat_1, stat_2, stat_3)$ nous avons adopté la notation abrégée usuelle en synthèse logique dans laquelle $'-'$ signifie valeur indifférente. Un vecteur avec le symbole $'-'$ représente en fait plusieurs vecteurs. Par exemple, le vecteur $status = (true, -, -)$ "contient" les quatre vecteurs: $(true, true, true)$, $(true, true, false), (true, false, true), (true, false, false)$. En conséquence, la transition $f(s_1, true, -, -) = s_1$ définie par la fonction de transition, représente quatre transitions différentes, une pour chaque combinaison entièrement fixée des paramètres.

Or dans la pratique les variables $stat_1$, $stat_2$, ..., $stat_q$ et, par conséquent, la variable *status* sont du type *Boolean*, ultérieurement nous allons utiliser des expressions booléennes construites à partir des variables primitives des statuts comme des valeurs de la variable *status*. Cette notation, pratiquée dans la synthèse logique ([Bar94]), permettra de manipuler plus facilement des valeurs de cette variable et d'appliquer pour la synthèse de haut niveau certains raisonnements usuels dans la synthèse logique.

Dans l'exemple ci-dessus chaque valeur de *status* est associée à une conjonction des variables $stat_1$, $stat_2$, et $stat_3$ soit positives (si la valeur d'une variable est $true$ dans le vecteur *status*) soit négatives (si la valeur d'une variable est $false$ dans le vecteur *status*). Ainsi, la valeur $(true, false, false)$ est représentée par l'expression booléenne $stat_1 \& \overline{stat_2} \& \overline{stat_3}$. Les vecteurs "incomplets" de statuts sont associés aux monômes booléens "incomplets" qui ne contiennent que certaines variables primitives. Par exemple, le vecteur $(false, true, -)$ est associé à l'expression $\overline{stat_1} \& stat_2$. Comme pour un vecteur avec des valeurs indifférentes, un monôme booléen "incomplet" représente plusieurs valeurs de la variable *status*.

La fonction de transition et la fonction de sortie de l'exemple ci-dessus peuvent donc être réécrites comme suit:

la fonction de transition f:	la fonction de sortie h:
$f(s_1,\ stat_1) = s_1;$	$h(s_1,\ stat_1) = fun\_ass_1;$
$f(s_1,\ \overline{stat_1} \& stat_2) = s_2;$	$h(s_1,\ \overline{stat_1} \& stat_2) = fun\_ass_2;$
$f(s_1,\ \overline{stat_1} \& \overline{stat_2}) = s_2;$	$h(s_1,\ \overline{stat_1} \& \overline{stat_2}) = fun\_ass_3;$
$f(s_2,\ stat_3) = s_1;$	$h(s_2,\ stat_3) = fun\_ass_4;$
$f(s_2,\ \overline{stat_3}) = s_2$	$h(s_2,\ \overline{stat_3}) = fun\_ass_5;$

Les représentations graphiques des machines M_{cl} et M_{ab} sont données dans la figure 3.9. La machine classique (figure 3.9.a) possède six états (le produit des ensembles $S = \{s_1, s_2\}$ et $V = \{1,2,3\}$). Les transitions sont étiquetées par un couple i/o d'entrée/sortie. La machine abstraite possède seulement deux états de contrôle. Les transitions sont étiquetées par les statuts (les signaux de statut sont remplacés par les fonctions de statut correspondantes: $stat_i = fun\_stat_i(i, v)$) et les affectations de la variable v et de la sortie o. Les affectations correspondent à la fonction $fun\_asssign$ définie par la fonction de sortie h. Par exemple, $h(s_2,\ stat_3) = fun\_ass_4 = (fun\_ass\_v_4,\ fun\_ass\_o_4)$. Par conséquent $v' = i + 1$ et $o = 2v + 2$.

3.5 Discussions

Nous avons montré que la machine classique de Mealy (figure 3.9.a) et la machine abstraite (figure 3.9.b) sont deux représentations différentes du même circuit. Du point de vue de la synthèse de haut niveau, la représentation d'un circuit par la machine abstraite présentes certains avantages.

Tout d'abord, le modèle de la machine abstraite est très proche de la description algorithmique d'un circuit. En effet, on voit la correspondance quasi-directe entre la machine abstraite de la figure 3.9.b et la description comportementale à la VHDL (3.10.a) et le graphe de flot de contrôle (3.10.b) correspondants[8]. Les entrées de la machine abstraite correspondent aux condi-

8. La description comportementale et le graphe de flot de contrôle ne correspondent à aucun algorithme si-

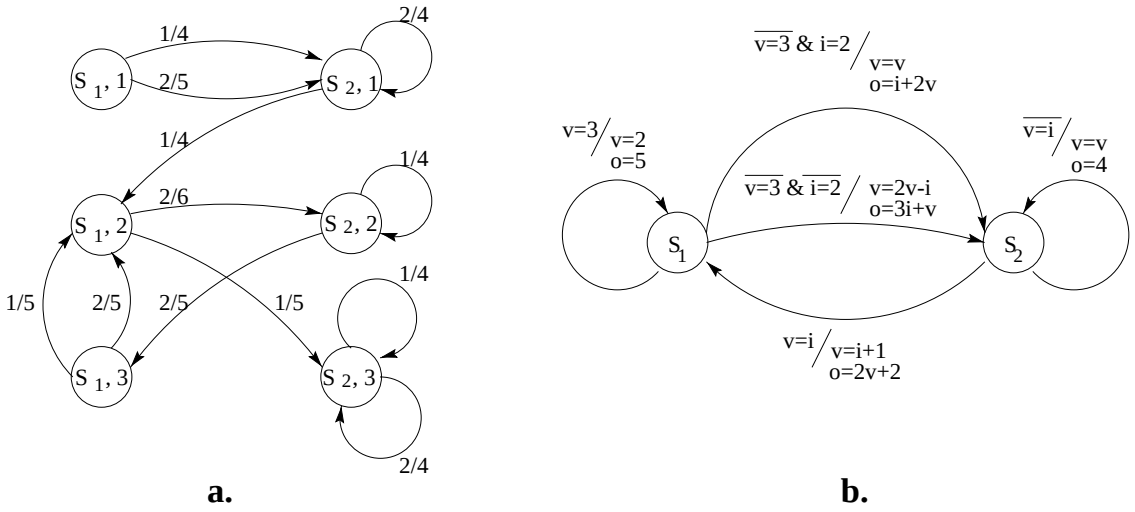


FIG. 3.9 – *Représentation graphique de la machine d'états finis classique de Mealy et la machine abstraite correspondante.*

tions des instructions de contrôle telles que "wait", "if", "case", "while". Ainsi, la condition $(i = 2)$ de la première instruction "if (figure 3.10.a) devient l'entrée $stat_2 = fun\_stat_2(i, v) = (i = 2)$. Les entrées primaires en elles-même ne conviennent plus pour contrôler le comportement du circuit. Les instructions d'affectations, elles, se transforment directement en sorties de la machine abstraite (figure 3.9.b).

Puisque le modèle de la machine abstraite suit très étroitement la description comportementale, il est beaucoup plus concis que le modèle de la machine classique. On le constate en comparant les tableaux 3.8.a et 3.8.d et les représentations graphiques de la figure 3.9. La machine abstraite possède un espace d'état réduit par rapport à la machine classique: seules les "étapes de calcul", issues de l'ordonnancement, sont conservées sous la forme d'états de contrôle. La figure 3.10 montre ces états S_1 et S_2 dans la description comportementale (3.10.a) et dans le graphe de flot de contrôle (3.10.b). Les affectations d'identité $v := v$ sont normalement omises dans la description comportementale: elles apparaissent ici pour rendre plus évidente la correspondance avec la machine abstraite.

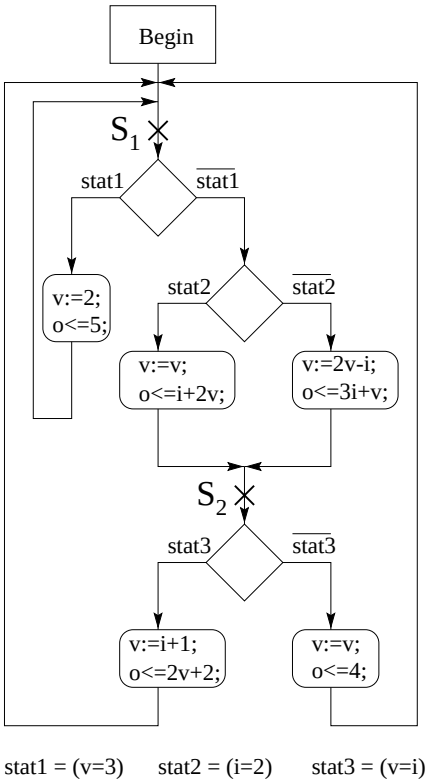
Du point de vue de la future architecture, l'ensemble S constitue le registre d'état du contrôleur et l'ensemble V constitue les registres du chemin de données. On peut donc considérer le modèle de la machine abstraite comme un modèle issu de la répartition de tous les registres d'un circuit en registre d'état et en registres du chemin de données.

Les entrées de la machine abstraite méritent, elle-aussi, quelques remarques. Nous avons déjà vu que l'ensemble des signaux des statuts $STATUS$ remplacent les entrées primaires de la machine d'états finis classique (dans les définitions des fonctions de transition f et de sortie h). En pratique, l'ensemble $STATUS$ est un produit d'ensembles booléens: $STATUS = STAT_1 \times STAT_2 \times \ldots \times STAT_q = Bool^q$, et le vecteur $status \in STATUS$ est un vecteur booléen. On

gnificatif. La raison en est la croissance exponentielle du nombre d'états de la machine classique même pour un algorithme simple. Par exemple, l'algorithme du "plus grand diviseur commun" demande deux variables pour le calcul. Étant donnée un minimum de 3 bits pour chaque variable et 5 états de contrôle, le nombre d'états total est $2^3 * 2^3 * 5 = 8 * 8 * 5 = 320$ états. Ce nombre devient trop grand pour un exemple éducatif d'une machine d'états finis classique pour en montrer l'équivalence avec une machine abstraite.

```
while (true) loop          --- S1
   while (v=3) loop
      v:=2;
      o<=5;
   end loop;
   if (i=2)
      then v:=v; o<=i+2v;
      else v:=2v-i; o<=3i+v;
   end if;
   if (v=i)                --- S2
      then v:=i+1; o<=2v+2;
      else v:=v; o<=4;
   end if;
end loop;
```

a.



b.

FIG. 3.10 – *Description algorithmique et graphe de flot de contrôle correspondant*

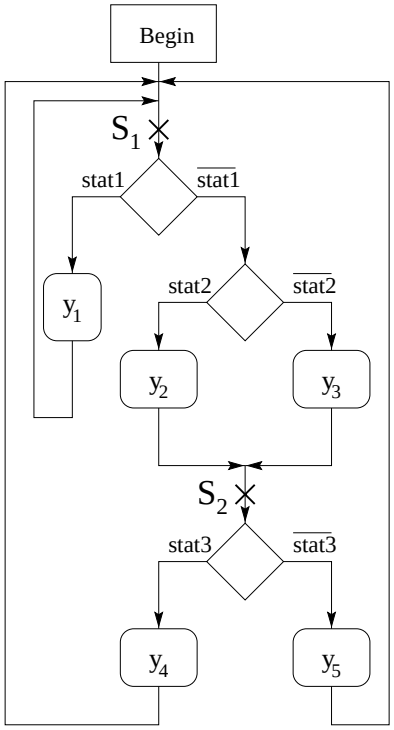
constate alors que pour la synthèse les signaux de statut jouent exactement le même rôle que les entrées primaires booléennes de la machine classique. Par exemple, si l'on considère le flot de contrôle de la figure 3.10.b comme le flot de contrôle de la machine classique avec les entrées booléennes $stat_1$, $stat_2$ et $stat_3$ et les sorties booléennes y_1, y_2, y_3, y_4, y_5 correspondant à chaque rectangle d'affectations (figure 3.11.b), alors la construction de la machine abstraite peut être remplacée par la construction d'un automate de contrôle lors de la synthèse logique. En effet, selon Baranov ([Bar94]), l'automate synthétisé à partir du flot de contrôle de la figure 3.11.b aura deux états: S_1 et S_2, chaque état correspondant à une entrée d'un sommet précédé d'un sommet d'affectations (figure 3.11.a).

De plus, les conditions de déterminisme définies pour la machine d'états finis classique restent valides pour la machine abstraite. La machine d'états finis est dite déterministe si l'on peut, connaissant l'état courant et les entrées primaires, calculer l'unique état suivant ([Koh78], [Hen68]). Mathématiquement, la machine d'états finis est déterministe, si la somme booléenne des prédicats sur les transitions issues de chaque état est égale à 1, et si ces même prédicats sont exclusifs deux à deux. D'une manière plus formelle, si s_i est un état d'une machine d'états finis, et $P_{i,j}$ $(1 \leq j \leq n)$ sont les n prédicats sur les n transitions issues de s_i, alors, la machine d'états finis est déterministe si pour chaque état:

(i) $\bigvee_{j=1}^{n} (P_{i,j}) = 1$

State	Input	Transition function	Output function
S	STATUS	S	Y
s_1	stat1	s_1	y_1
	$\overline{\text{stat1}}$ & stat2	s_2	y_2
	$\overline{\text{stat1}}$ & $\overline{\text{stat2}}$	s_2	y_3
s_2	stat3	s_1	y_4
	$\overline{\text{stat3}}$	s_2	y_5

a.



b.

FIG. 3.11 – *Le flot de contrôle de la machine d'états finis classique et la machine correspondante*

(ii) $(P_{i,j}\&P_{i,k}) = 0$ ou bien $(\overline{P_{i,j}\&P_{i,k}}) = 1$ pour $j,k \in \{1,2,\ldots,n)$ et $j \neq k$

Donnons les conditions de déterminisme pour l'état s_1 de la machine classique illustrée par le tableau 3.11.a :
(i) $[stat_1 \vee (\overline{stat_1}\&stat_2) \vee (\overline{stat_1}\&\overline{stat_2})] = 1$
(ii) $[stat_1\&(\overline{stat_1}\&stat_2)] = 0$, $[stat_1\&(\overline{stat_1}\&\overline{stat_2})] = 0$, $[(\overline{stat_1}\&stat_2)\&(\overline{stat_1}\&\overline{stat_2})] = 0$
Les prédicats $P_{1,1} = stat_1$, $P_{1,2} = \overline{stat_1}\&stat_2$, et $P_{1,3} = \overline{stat_1}\&\overline{stat_2}$ sont définis sur les entrées booléennes de la machine classique et forment les conditions de déterminisme pour cette machine. Cependant, ces même conditions de déterminisme restent valides pour la machine abstraite définie par le tableau 3.8.d. On conclut, qu'il y a une forte correspondance entre les statuts de la machine abstraite et les entrées primaires de la machine d'états finis classique du graphe de contrôle.

3.6 Résumé

Dans ce chapitre nous avons développé le modèle de la machine abstraite. Ce modèle provient de la description comportementale d'un circuit et fournit donc un niveau d'abstraction plus élevé que le modèle de machine d'états finis classique correspondant à la description d'un circuit au niveau des portes. Ce plus haut niveau d'abstraction est assuré par la représentation *symbolique* des éléments du modèle au lieu de l'énumération de leurs valeurs (on peut dire que la machine abstraite assure la *symbolisation* des fonctions de la machine d'états finis classique). En outre, la machine abstraite est plus structurée que la machine d'états finis classique. La structure

implicite de la description initiale est conservée dans la machine abstraite: des points de contrôle sont transformés en des états de contrôle et représentent un futur contrôleur d'un circuit. Des affectations sont supposées mises en oeuvre par des blocs spécifiques et composent un futur chemin de données.

Nous avons montré également que le modèle de la machine abstraite est en fait le modèle de la machine d'états finis classique modifié. Nous avons mis en évidence les points communs entre les deux modèles: les entrées primaires de la machine classique et les signaux de statuts de la machine abstraite sont de même nature, de même que la synthèse d'un automate de contrôle de la machine classique et la synthèse de la machine abstraite.

Finalement, nous avons montré comment effectuer le passage d'un modèle à l'autre, dans les deux directions. La transformation du modèle de la machine abstraite vers la machine classique peut être considérée comme une synthèse formelle, et peut éventuellement servir de base pour les futurs algorithmes de la synthèse de haut niveau formellement définis.

Chapitre 4

Vérification de l'étape d'ordonnancement

La machine abstraite est la base des modèles adoptés pour la description initiale et l'architecture finale. Les méthodes de vérification des étapes de la synthèse de haut niveau (l'ordonnancement et l'allocation) sont également fondées sur ce modèle. La vérification de l'étape d'ordonnancement avec le modèle adopté pour la description initiale font l'objet de ce chapitre.

4.1 Principe de la méthode proposée

Pour faciliter la compréhension de la méthode proposée, nous donnons dans cette section les idées de base qui seront ensuite développées tout au long de ce chapitre. La figure 4.1 montre schématiquement les phases nécessaires pour la vérification de l'ordonnancement.

Tout d'abord, une machine abstraite est construite pour la description initiale comportementale VHDL. Ce processus est illustré par la colonne gauche de la figure 4.1. La machine abstraite est dérivée en deux étapes. Premièrement, on construit un modèle intermédiaire qui est très proche à la description-source VHDL. Deuxièmement, on dérive la machine abstraite à partir du modèle intermédiaire.

Le modèle intermédiaire peut être considéré comme une sorte de machine abstraite qui associe une *séquence* d'affectations à chaque transition. Les états de cette machine correspondent aux instructions "wait" de la description initiale, et les séquences d'affectations imitent les séquences d'affectations placées entre ces instructions dans le processus VHDL. Ainsi, la description initiale et le modèle intermédiaire sont pratiquement semblables.

Les fonctions issues des instructions de contrôle telles que "if", "while", "case" sont également inclues dans les séquences d'affectations associées aux transitions du modèle intermédiaire. C'est le cas de la fonction $(c =' 0')$ qui vient de l'instruction "if" de la description initiale VHDL (figure 4.1, le dessin au milieu de la colonne gauche). Remarquons que les fonctions issues des instructions de contrôle sont distinguées de leurs résultats dans le modèle intermédiaire. Ainsi, les deux transitions du modèle intermédiaire contiennent la même fonction $(c =' 0')$. Cependant, une transition est associée avec la valeur $true$ et l'autre avec la valeur $false$, $true$ et $false$ étant les résultats de la fonction $(c =' 0')$.

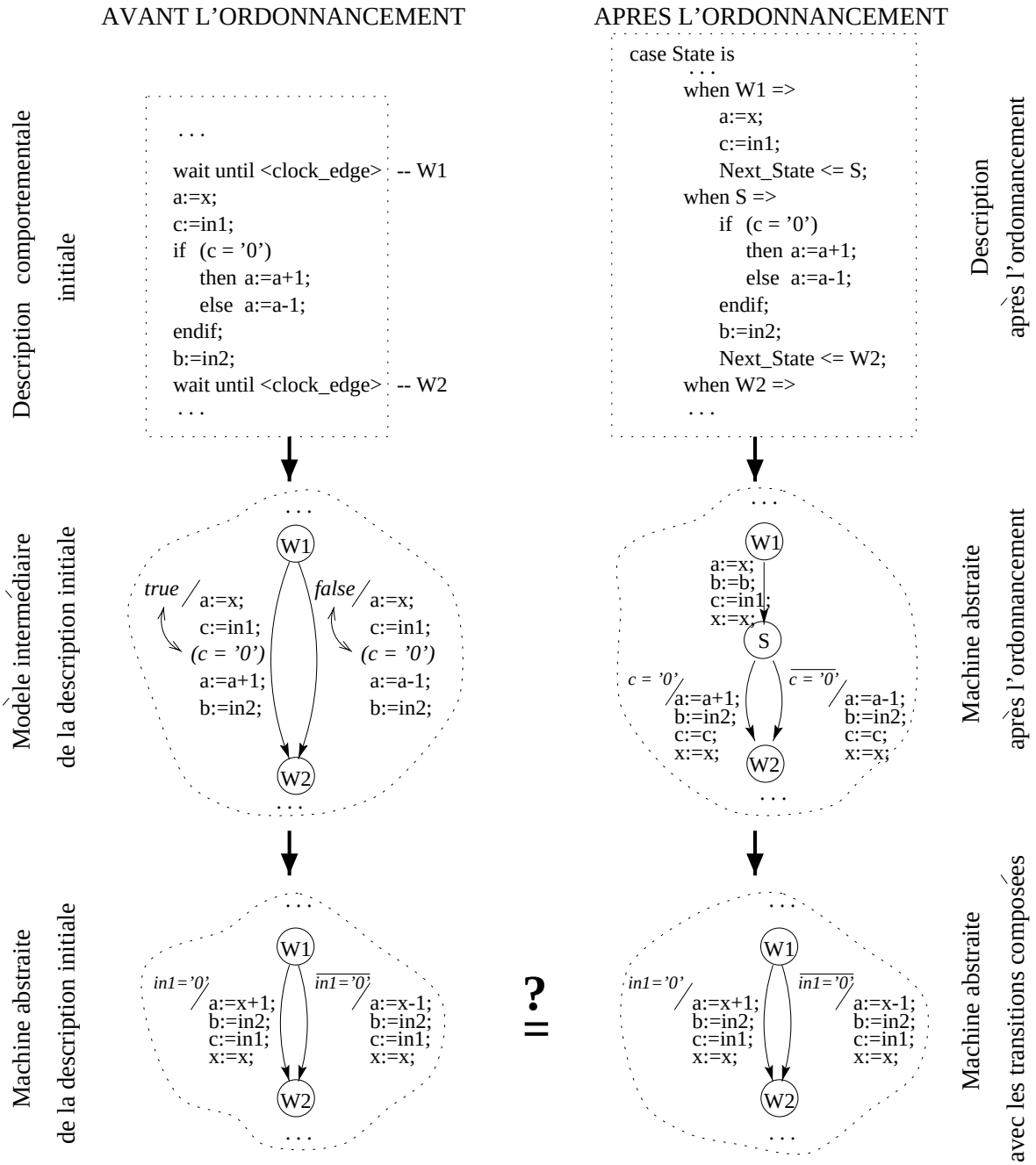


FIG. 4.1 – *Principe de la vérification de l'étape d'ordonnancement*

Cette séparation a été déjà introduite dans la machine abstraite: son modèle contient la fonction de statut fun_status (les fonctions issues des instructions de contrôle) et l'ensemble $STATUS$ (les résultats des fonctions de contrôle). Pour le modèle intermédiaire cette séparation est cruciale car elle permet, pour une transition donnée, de rendre les fonctions de contrôle indépendantes de leurs positions dans une séquence d'affectations. Cette démarche est faite par la composition des affectations associées à chaque transition du modèle intermédiaire. La composition remplace également les affectations *séquentielles* du modèle intermédiaire par les affectations *parallèles* de la machine abstraite (voir la transformation du modèle intermédiaire en une machine abstraite de la figure 4.1). L'algorithme de la composition est donné dans la section 4.4.

Remarquons que du point de vue de la parallélisation, la composition des affectations ressemble aux méthodes employées pour la construction de systèmes parallèles à partir de programmes séquentiels ([CT93]). En général, néanmoins, le passage d'un système mono-processeur à un système parallèle multi-processeurs demande des algorithmes beaucoup plus complexes, qui prennent en compte l'occupation de chaque processeur utilisé.

Après avoir ramené la description initiale VHDL à la forme d'une machine abstraite, nous pouvons comparer les deux machine abstraites: avant et après l'ordonnancement. Leur équivalence impliquera l'exactitude de cette étape. Nous définissons deux machines abstraites équivalentes, si elles sont équivalentes *transition par transition*. Cette définition est similaire à la définition de la bisimulation ([Mil80]) où deux systèmes doivent coïncider à chaque pas d'exécution.

La définition d'équivalence de machines abstraites impose donc que les deux machines comparées aient le même nombre d'états. La machine abstraite obtenue après l'ordonnancement peut, néanmoins, avoir plus d'états que la machine initiale. Afin d'éviter cet inconvénient, la machine obtenue après l'ordonnancement est transformée en une machine ayant le même nombre d'états que la machine abstraite avant l'ordonnancement par composition des transitions insérées lors de l'ordonnancement (section 4.5).

4.2 Sous-ensemble VHDL comportemental reconnu

Le modèle intermédiaire correspond à une spécification initiale décrite comme un *process* (unique) du langage VHDL et représente la sémantique formelle adoptée pour ce langage. De nombreux travaux ont été dédié à la formalisation de VHDL. Nous en avons cité quelques uns dans l'introduction de cette thèse.

Malgré la diversité des méthodes proposées dans la littérature, la plupart d'entre elles vise d'abord à *valider* un circuit décrit en VHDL. Ce but impose la formalisation et la restriction de sous-ensemble VHDL qui ne sont pas toujours appropriées pour la synthèse en général et pour la vérification des résultats de la synthèse de haut niveau en particulier. Par exemple, la formalisation de VHDL porte très souvent sur le cycle de simulation, simulation δ inclue. Lors de la synthèse, les cycles de simulation en eux-même ne sont pas si importants ([Ska96]). De plus, même la correspondance entre le temps dit physique et les valeurs des signaux associées à ce temps, n'est pas respectée, car les états supplémentaires sont presque toujours insérés pendant

la synthèse, ce qui étend l'échelle temporelle initiale.

Un autre exemple de formalisation de VHDL non compatible avec la synthèse sont les travaux [BFK94] et [BFK95] qui restreignent le sous-ensemble de VHDL à l'affectation d'un signal avec un délai non-zéro ($sig <=' 1' after\ 2\ ns;$). Cette affectation dans le meilleur des cas sera ignorée sinon rejetée par des outils de synthèse.

Pour ces raisons, nous nous sommes inspiré de modèles utilisés pour la synthèse de haut niveau plutôt que de modèles issus de la formalisation du langage VHDL lors de la construction du modèle intermédiaire correspondant à la description initiale. Ainsi, les travaux présentés dans [BR96] et [BR97] portent sur la vérification des résultats de la synthèse de haut niveau en utilisant la simulation. Une tentative pour formaliser un process de VHDL est néanmoins faite. Cette formalisation nous a servi de point de départ pour le modèle intermédiaire. La spécification de la synthèse de haut niveau est restreinte dans [BR97] à un processus VHDL avec des instructions "wait" (par opposition à un processus avec une liste de sensibilité). De plus, chaque instruction "wait" est obligatoirement synchronisée par le front d'horloge: dans [BR96] l'instruction "wait" ne peut être utilisée que sous l'une des deux formes suivantes:

- $Wait\ Until\ Not\ clock'Stable\ AND\ (clock =' 1')\ AND\ < expression >$ ou bien
- $Wait\ Until\ Not\ clock'Stable\ AND\ (clock =' 0')\ AND\ < expression >$

Un état est défini comme une séquence d'opérations entre point de suspension et point de reprise, c'est-à-dire entre deux instructions "wait". Une transition, toujours synchronisée par le front d'horloge, se produit quand la simulation reprend l'exécution du processus après une suspension.

Remarquons qu'avec ces restrictions (processus avec les instructions "wait" synchronisées), l'exécution du processus entre les deux "wait" correspond au maximum à deux cycles de simulation: le premier se produit sur le front d'horloge et le deuxième est un cycle de simulation δ, pendant lequel les signaux affectés lors du premier cycle prennent réellement leurs valeurs. Nous rappelons que seules les affectations des signaux avec un délai zéro (δ-delay) sont permises dans [BR96].

Les restrictions imposées associent, par ailleurs, l'avancement du temps physique seulement avec les opérations "wait". En effet, selon l'algorithme de simulation ([IEE93]), l'avancement du temps (physique et δ) se produit soit quand un signal est mis à jour, soit quand un processus reprend son exécution après une suspension. La mise à jour d'un signal est définie par une affectation. La reprise de l'exécution d'un processus est définie par une opération "wait". Puisque seul le délai zéro est accepté ($sig <=' 1' after\ 0\ ns;$ ou simplement $sig <=' 1';$), les affectations des signaux n'avancent pas le temps physique (elles avancent le temps δ). Les opérations "wait", en revanche, n'avancent que le temps physique en raison de la synchronisation ($wait\ until\ not\ clock'Stable\ AND\ clock =' 1'\ AND\ a = b$). Ainsi chaque opération "wait" est associée à un front d'horloge.

Pour le modèle intermédiaire correspondant à la description comportementale VHDL, nous avons utilisé le même concept que [BR96]: la description initiale est restreinte à un processus VHDL avec des instructions "wait" synchronisées, chaque "wait" correspondant à un état du futur modèle.

Pour dériver un système de transitions à partir de la description initiale VHDL dont les instructions "wait" se traduisent en états du système, nous interdisons dans la description initiale les boucles "while" sans instructions "wait" à l'intérieur. Cette décision est tout à fait naturelle du point de vue d'un circuit réel: une boucle introduit soit de nouvelles données, soit une nouvelle étape de calcul. Dans les deux cas, un point de contrôle est nécessaire pour choisir le prochain chemin d'exécution, sinon le circuit sera mal conçu en raison d'une boucle combinatoire. Ce point de contrôle est défini par une instruction "wait" dans la description initiale et se traduira par un vrai état de contrôle dans la machine abstraite. Le *Behavioral Compiler* de Synopsys ([Kna96]) impose d'ailleurs la même restriction sur la description fournie à l'outil.

La dernière limitation sur le sous-ensemble VHDL reconnu concerne l'instruction "for". Cette instruction est un "sucre syntaxique" et peut être toujours soit déroulée, soit remplacée par l'instruction "while". Effectivement, si l'instruction "for" est une boucle avec les index fixés à l'avance, le nombre d'itérations est connu et la boucle peut être déroulée. Si, en revanche, les index ne sont pas connus à l'avance, la boucle "for" peut être remplacée par une boucle "while" où les index sont calculés avant ou à l'intérieur de la boucle.

En résumé, les principales restrictions que nous imposons sur le sous-ensemble VHDL reconnu sont:

1. Description initiale restreinte à un processus VHDL avec des "wait" synchronisés;
2. Délai non-zéro interdit lors de l'affectation des signaux;
3. Instruction "for" interdite;
4. Interdiction des boucles dont le corps ne contient pas d'instructions "wait".

4.3 Définition formelle du modèle intermédiaire

Cette section est dédiée à l'extraction du modèle intermédiaire à partir de la description initiale VHDL. Le modèle intermédiaire constitue une étape ayant pour but de faciliter la construction de la machine abstraite qui modélise la sémantique de la description initiale. Dans le modèle intermédiaire la séquentialité de calcul est préservée. À chaque affectation séquentielle et à chaque calcul de la condition d'une instruction de rupture de séquence ("if", "case", "while") est associée une fonction. Pour rendre le modèle intermédiaire cohérent avec le modèle de la machine abstraite, on considère une affectation globale des sorties et des variables mémorisées dans laquelle seule une composante est modifiée plutôt que considérer séparément chacun des objets affectés comme dans le processus VHDL. Pour obtenir une affectation globale à partir d'une affectation élémentaire de la description initiale, on la complète avec des affectations d'identité pour les variables mémorisées et avec des absences d'affectations pour les sorties. Les affectations d'identité associent chaque variable à elle-même. Les absences d'affectations sont notées $\perp$.

Le modèle intermédiaire se distingue donc de la définition de la machine abstraite par les points suivants:

– Les fonctions fun_assign du modèle intermédiaire contiennent une seule affectation élémentaire différente de la fonctions d'identité et de $\perp$.

- Les fonctions fun_stat_k $(1 \leq k \leq q)$ sont définies comme des objets séparés et non pas comme les projections de la fonction-vecteur fun_status. Pour éviter toute ambiguïté, ces fonctions issues des instructions de contrôle "if", "case", "while" sont appelées fonctions de conditions fun_cond_k $(1 \leq k \leq q)$ dans la définition du modèle intermédiaire. Par conséquent, les résultats des fonctions de conditions sont les variables $cond_k$ $(1 \leq k \leq q)$ (à la place de $stat_k$ dans la machine abstraite). Les variables $cond_k$ forment la variable-vecteur $condition = (cond_1, \ldots, cond_q) \in COND_1 \times \ldots \times COND_q = CONDITION$ (analogue à la variable $status = (stat_1, \ldots, stat_q) \in STAT_1 \times \ldots \times STAT_q = STATUS$ de la machine abstraite).

- Finalement, un nouvel ensemble des transitions $TRANSITION$ est introduit. Les éléments de cet ensemble sont les séquences de fonctions d'affectations (fun_assign_i) et de conditions (fun_cond_k). Chaque séquence correspond à une séquence d'affectations élémentaires et de calculs de conditions de la description initiale entre deux instructions "wait". L'ordre des fonctions est préservé.

Dans le modèle intermédiaire nous distinguons strictement les fonctions qui définissent la future transition (fun_cond_1, ..., fun_cond_q) des résultats de ces fonctions ($cond_1$, ..., $cond_q$).

La séparation entre les fonctions des conditions et leurs résultats est cruciale. La valeur du résultat reste invariable pour une transition donnée, tandis que le corps de la fonction correspondante dépend de sa position dans la transition "d'accueil". Lors de l'ordonnancement, cette position peut changer engendrant ainsi la modification de la fonction de condition. Une telle situation est montrée dans le paragraphe 2.1. Le but de la formalisation est de trouver une forme canonique pour chaque fonction de condition rencontrée dans la description initiale. Cette forme doit être indépendante de la position de la fonction de condition dans une transition "d'accueil". La technique proposée consiste à recalculer chacune des fonctions de condition relativement aux valeurs des variables et des entrées au début de la transition, ce qui aboutit aux définitions des fonctions de statuts de la machine abstraite. Cette technique sera précisée dans la section suivante.

La nature séquentielle des fonctions de condition ne permet pas, en outre, de les réunir dans un vecteur de fonctions comme c'est le cas pour les fonctions de statuts dans le modèle de la machine abstraite. En effet, les fonctions de statuts sont calculées en parallèle relativement aux valeurs des entrées et des variables au début de la transition. Ces valeurs sont les mêmes pour toutes les fonctions de statuts, ce qui autorise la signature $I \times V \rightarrow STATUS$ pour le vecteur de fonctions de statuts. Les fonctions de condition, en revanche, sont calculées relativement aux valeurs des entrées et des variables acquises lors des affectations précédentes. Ces valeurs sont propres à chaque fonction de condition et dépendent de sa position dans la séquence. Ceci ne permet pas d'utiliser un vecteur de fonctions de condition.

Les éléments de l'ensemble $TRANSITION$ sont associés aux transitions du modèle intermédiaire et correspondent aux chemins d'exécution entre deux instructions "wait". S'il existe plusieurs parcours entre deux "wait" conditionnés par la présence d'instructions de "bifurcation" telles que "if" ou "case", alors un élément de l'ensemble $TRANSITION$ correspond à chaque chemin.

Exemple

Ainsi, l'extrait de description comportementale montré dans la figure 4.2, donne lieu à trois chemins d'exécution entre les instructions "wait" W_1 et W_2 appelés $transition_1$, $transition_2$ et $transition_3$ (les fonctions de condition sont marquées en *italique*). Chaque affectation élémentaire est complétée par les fonctions d'identité des variables a, x et y et par l'absence d'affectation de la sortie $out1$. Pour chacun des trois parcours possibles, les fonctions directement issues de la description initiales sont alignées sur une rangée horizontale.

Chaque chemin d'exécution correspond aux valeurs particulières des variables de conditions. Si on note $cond_1$ et $cond_2$ les variables de conditions correspondant aux fonctions $fun\_cond_1\,(in1, a, x, y) = (in1 > 0)$ et $fun\_cond_2\,(in1, a, x, y) = (a = 5)$, alors la $transition_1$ est exécutée si $cond_1 = cond_2 = true$, la $transition_2$ est exécutée si $cond_1 = true$ et $cond_2 = false$, et $transition_3$ est exécutée si $cond_1 = false$.

Comme pour la variable *status* du chapitre précédent, les valeurs de la variable $condition = (cond_1,\ cond_2)$ sont formées par les valeurs particulières des variables élémentaires de conditions ($cond_1$, $cond_2$) et sont définies par des expressions booléennes construites à partir de ces variables élémentaires. Les séquences de fonctions $transition_1$, $transition_2$ et $transition_3$ sont donc associées aux expressions booléennes $cond_1 \& cond_2$, $cond_1 \& \overline{cond_2}$ et $\overline{cond_1}$ respectivement. Remarquons que l'expression booléenne $\overline{cond_1}$ représente deux valeurs de la variable $condition$: ($false$, $true$) et ($false$, $false$). Les expressions booléennes formées par les variables élémentaires de conditions participent aux définitions des fonctions de transition f (la même que pour la machine abstraite) et de sortie h (qui associe à un couple $(condition, i)$ une $transition \in TRANSITION$).

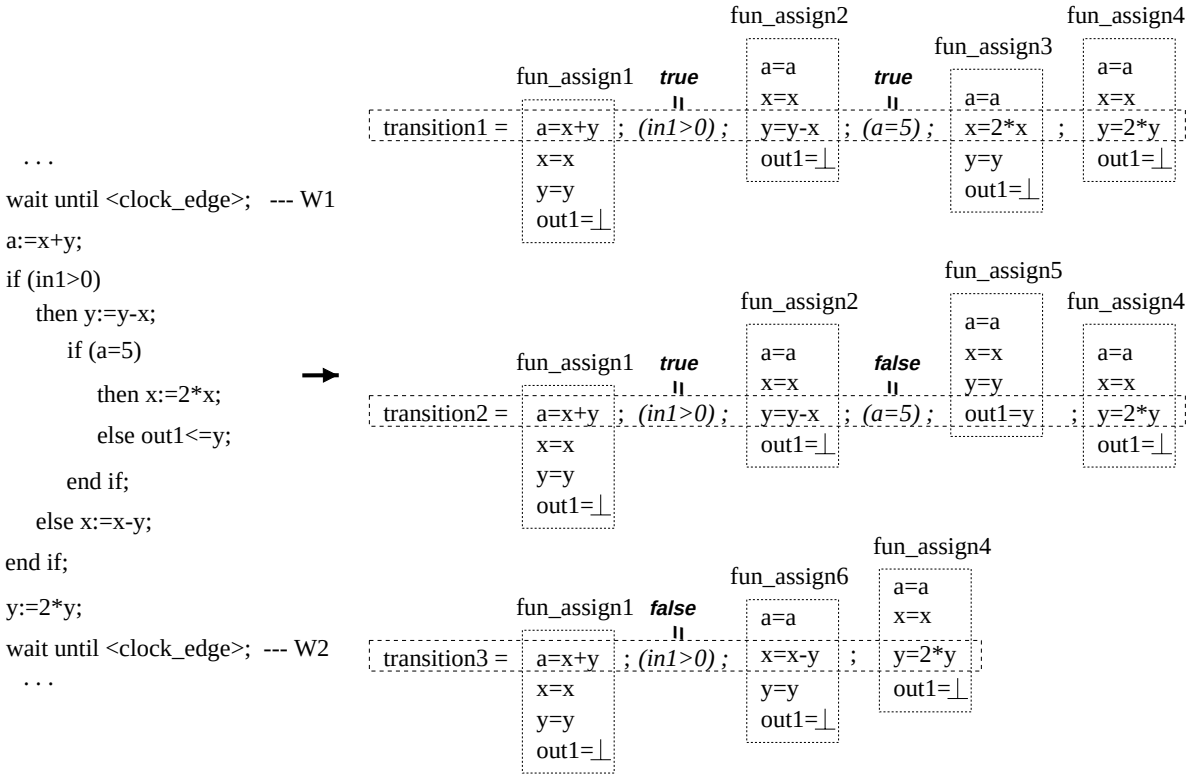


FIG. 4.2 – *Transitions issues de la description comportementale*

Fin exemple

Nous adoptons les notations suivantes:

- S est l'ensemble des états de contrôle, chaque état correspondant à une instruction "wait" sous la forme "$Wait\ Until\ \ < clock\_edge >$" du processus VHDL et à la première instruction du processus si elle différente de l'instruction "wait"; $< clock\_edge >$ étant une expression représentant un front d'horloge (montant ou descendant).

- I est l'ensemble des symboles des entrées, formé par les valeurs des signaux des entrées.
  $i = (i_1, i_2, ..., i_l) \in IVAL_1 \times IVAL_2 \times ... \times IVAL_l = I$
  où $IVAL_j$ est le type de l'entrée i_j dans la description initiale. Nous gardons la même notation que pour la machine abstraite.

- O est l'ensemble des symboles des sorties, formé par les valeurs des signaux des sorties.
  $o = (o_1, o_2, ..., o_m) \in OVAL_1 \times OVAL_2 \times ... \times OVAL_m = O$
  où $OVAL_j$ est le type de de la sortie o_j dans la description initiale. La notation est la même que pour l'ensemble d'entrées.

- V est l'ensemble des symboles des variables mémorisées, chaque variable correspondant à une variable ou un signal utilisé dans le corps du processus VHDL.
  $v = (v_1, v_2, ..., v_n) \in VVAL_1 \times VVAL_2 \times ... \times VVAL_n = V$
  où $VVAL_j$ est le type correspondant à la variable v_j.

- $CONDITION$ est l'ensemble des symboles des conditions formés par les valeurs des variables des conditions. Chaque variable élémentaire de condition correspond à une condition dans les opérations de contrôle telles que "if", "while", "case" de la description initiale.
  $condition = (cond_1, cond_2, ..., cond_q) \in COND_1 \times COND_2 \times ... \times COND_q = CONDITION$
  où $COND_j$ est l'ensemble de toutes les valeurs correspondant à la variable $cond_j$.

- $ASSIGN$ est l'ensemble des affectations des variables et des signaux des sorties. Un élément de cet ensemble est un vecteur fun_assign avec une seule composante différente de fonctions d'identité (pour les variables) et de $\perp$ (pour les sorties[1]):
  $ASSIGN = \{fun\_assign \in I \times V \rightarrow V \times O\}$ où
  $fun\_assign = (fun\_ass\_v_1, \ldots, fun\_ass\_v_n, fun\_ass\_o_1, \ldots, fun\_ass\_o_m)$

- fun_cond_1, fun_cond_2, ..., fun_cond_q sont les fonctions qui calculent les valeurs des conditions apparaissant dans les instructions "if", "case", "while". Ces fonctions sont de signature $I \times V \rightarrow COND_k$ $(1 \leq k \leq q)$:
  $fun\_cond_k : I \times V \rightarrow COND_k$ $(1 \leq k \leq q)$.
  Les fonctions de conditions seront utilisées pour calculer les fonctions des statuts de la machine abstraite.

- $TRANSITION$ est l'ensemble des séquences de fonctions d'affectations et de fonctions de calcul des conditions, dans l ur ordre d'apparition dans le processus-source VHDL entre deux "wait":

$TRANSITION \subseteq \{transition\}$
où une $transition$ est définie de la manière récursive suivante:
$transition ::= fun\_assign_i \mid fun\_cond_k \mid transition\,;\,transition$
où la barre verticale (|) signifie une alternative et le point virgule (;) signifie une séquence.

Étant données les définitions précédentes, un modèle intermédiaire est défini par un n-tuple:

$< S, I, O, V, CONDITION, ASSIGN, fun\_cond_1, ..., fun\_cond_q, TRANSITION, f, h >$
où

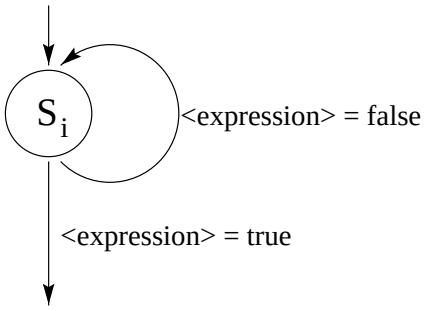
- f est la fonction calculant l'état suivant:
  $f : CONDITION \times S \rightarrow S$.

- h est la fonction qui détermine le chemin parcouru dans la description initiale entre deux états successifs:
  $h : CONDITION \times S \rightarrow TRANSITION$

Nous insistons sur le fait que l'ensemble d'états est "composé" d'instructions "wait" sous une forme "$Wait\ Until\ <clock\_edge>$" car l'instruction "wait" sous une forme "$Wait\ Until\ <clock\_edge>\ AND\ <expression>$" contient une condition ($<expression>$) qui influe sur l'exécution d'un programme. Ainsi, la deuxième forme de "wait" contient à la fois deux informations différentes: l'état courant du programme et une condition qui définit l'état prochain. Dans notre définition du modèle intermédiaire cette condition est représentée par une fonction fun_cond_i. Pour séparer ces deux informations, nous remplaçons les instructions "$WaitUntil\ <clock\_edge>\ AND\ <expression>$" par la boucle de la figure 4.3.a.

```
wait_loop: loop
    wait until <clock_edge>;
    if <expression>
        then exit wait_loop;
    end if;
end loop;
```

a.



b.

FIG. 4.3 – *Boucle équivalente à l'instruction "$WaitUntil\ <clock\_edge>\ AND\ <expression>$" et l'extrait de la machine abstraite correspondant*

1. Le signe spécial $\perp$ (bottom) désigne l'absence d'affectation d'une sortie et apparaît si les sorties ne sont pas mémorisées. Implicitement il appartient aux ensembles O_1, O_2, ..., O_m.
Si les sorties sont mémorisées (comme lors de la synthèse logique des processus synchronisés selon le standard [IEE98]), les variables supplémentaires correspondant aux sorties mémorisées doivent être introduites. Dans ce cas, les non-affectations des sorties seraient remplacées par les affectations similaires aux affectations d'identité des variables. À titre d'exemple, la fonction d'identité pour la sortie o_j serait $fun\_ass\_o_j(\vec{i}, \vec{v}) = v\_o_j$ où v_o_j est une variable mémorisée associée à la sortie o_j.

Sémantiquement les deux constructions sont équivalentes. Les instructions qui se trouvent après l'instruction "$Wait\ Until\ <clock\_edge>\ AND\ <expression>$" dans la description initiale seront placées juste après la boucle correspondante dans la description modifiée, comme le montre l'exemple suivant.

L'intérêt de la transformation de l'instruction "$Wait\ Until\ <clock\_edge>\ AND\ <expression>$" est la construction plus facile et surtout homogène du système de transition à partir de la description initiale. Cette instruction se traduit dans la machine abstraite en un état ayant une transition vers lui-même comme le montre la figure 4.3.b. Cette transition d'attente est dérivée de la même façon que les autres transitions à partir la description de la figure 4.3.a.

Enfin, pour respecter la sémantique de VHDL, un état supplémentaire initial est ajouté si la première instruction d'un processus n'est pas une instruction "wait". Toutefois, pour le calcul des fonctions f et h cet état n'est pris en compte qu'une seul fois, lors du calcul de la transition <état initial - état prochain> correspondant à l'étape d'initialisation du processus. Lors du calcul des transitions liées à la boucle du processus (arrivé à sa fin, un processus s'exécute à nouveau depuis le début), cet état n'est pas pris en compte. Le prochain état sera un état correspondant à la première instruction "wait" dans la séquence des instructions du processus.

Exemple

Pour des raisons de simplicité, nous prenons comme exemple l'algorithme bien connu du plus grand diviseur commun (the greatest common divisor- Gcd). Nous modifions les instructions "wait" selon la règle de la figure 4.3, et construisons le modèle formel correspondant. La description initiale est montrée dans la figure 4.4. Les attributs des entrées *clk* et *reset* servent à identifier l'horloge et la remise à zéro par l'outil de synthèse. Ces signaux, cependant, ne font pas partie de notre formalisation en raison de leur rôle particulier. Les attributs, ainsi que la fonction $rising\_edge$[2], sont définis dans le paquetage *Amical* inclu au début de la description. La figure 4.5 montre le même processus, où les instructions $Wait\ Until\ <clock\_edge>\ AND\ <expression>$ sont remplacées par les boucles correspondantes.

Les états associés aux instructions "wait" de la figure 4.5 sont notés $W1$, $W2$, $W3$ et $W4$. Nous associons également les résultats des conditions $(st =' 1')$, $(din =' 1')$, $(x \neq y)$, $(x < y)$ et $(din =' 0')$ avec les variables de conditions $cond_1$, $cond_2$, $cond_3$, $cond_4$ et $cond_5$ respectivement. Les types VHDL "Bit" et "Boolean" sont tous les deux représentés par l'ensemble mathématique *Bool* dans notre formalisation.

Le modèle formel issu de la description initiale est illustré par la figure 4.6 et contient les définitions suivantes:

- S est l'ensemble des états de contrôle:
  $s \in S = \{W_1, W_2, W_3, W_4\}$

- I est l'ensemble des symboles des entrées (moins l'horloge et la remise à zéro):
  $i = (xi,\ yi,\ st,\ din) \in I = N \times N \times Bool \times Bool$ où xi, yi, st et din sont les entrées du

2. La définition de la fonction $rising\_edge$ est la suivante:
Function Rising_Edge (Signal Horloge : Bit) Return Boolean Is
Begin Return (Horloge'Event And Horloge = '1' And Horloge'Last_Value = '0');
End Rising_Edge;

```
Use Work.Amical.All;

entity gcd is
 port (clk    : in bit;
       reset  : in bit;
       st     : in bit;
       din    : in bit;
       xi, yi : in natural;
       dout   : out bit;
       ou     : out natural);

  Attribute PortType Of Clk   : Signal Is Port_Clock;
  Attribute PortType Of Reset : Signal Is Port_Reset;
end gcd;

architecture behavior of gcd is
begin
  P1 : process
     variable x,y: natural;

  begin
       wait until (st = '1' and rising_edge(clk));
       dout <= '0';

       calculation : loop
            wait until (din = '1' and rising_edge(clk));
            x := xi;
            y := yi;
            while (x /= y ) loop
               if (x < y)
                      then y := y - x;
                      else x := x - y;
               end if;
               wait until rising_edge(clk);
            end loop;
            ou <= x;
            dout <= '1';
            wait until (din = '0' and rising_edge(clk));
            dout <= '0';
       end loop;
  end process;

end behavior;


configuration cfg_gcd of gcd is
for behavior
end for;
end cfg_gcd;
```

FIG. 4.4 – *Description initiale de l'algorithme Gcd (the greatest common divisor)*

```
architecture modified_behavior of gcd is
begin
  P1 : process
     variable x,y: natural;

  begin
       wait_loop: loop
           wait until rising_edge(clk);       --- W1
           if (st = '1')
                  then exit wait_loop;
           end if;
       end loop;
       dout <= '0';

       calculation : loop
            wait_loop: loop
                wait until rising_edge(clk);  --- W2
                if (din = '1')
                       then exit wait_loop;
                end if;
            end loop;
            x := xi;
            y := yi;
            while (x /= y ) loop
               if (x < y)
                      then y := y - x;
                      else x := x - y;
               end if;
               wait until rising_edge(clk);   --- W3
            end loop;
            ou <= x;
            dout <= '1';
            wait_loop: loop
                wait until rising_edge(clk);  --- W4
                if (din = '0')
                       then exit wait_loop;
                end if;
            end loop;
            dout <= '0';
       end loop;
  end process;

end behavior;
```

FIG. 4.5 – *Description modifiée de l'algorithme Gcd*

circuit (déclarées dans l'"entity").

- O est l'ensemble des symboles des sorties:
  $o = (dout, ou) \in O = Bool \times N$ où $dout$ et ou sont les sorties du circuit (déclarées dans l'"entity").

- V est l'ensemble des symboles des variables mémorisées:
  $v = (x, y) \in V = N \times N$ où x et y sont les variables déclarées dans le corps du processus.

- $CONDITION$ est l'ensemble des symboles formés par les valeurs des variables élémentaires de condition:
  $condition = (cond_1, cond_2, cond_3, cond_4, cond_5) \in CONDITION = Bool^5$

- $ASSIGN$ est l'ensemble de fonctions qui modélisent la partie droite des affectations des variables x, y, et des sorties $dout$ et ou:
  $fun\_assign = (fun\_ass\_x, fun\_ass\_y, fun\_ass\_dout, fun\_ass\_ou) : I \times V \to V \times O$ où
  $fun\_ass\_x \in \{fun\_ass\_x_0, fun\_ass\_x_1, fun\_ass\_x_2\} \subset \{I \times V \to N\}$
  $fun\_ass\_y \in \{fun\_ass\_y_0, fun\_ass\_y_1, fun\_ass\_y_2\} \subset \{I \times V \to N\}$
  $fun\_ass\_dout \in \{fun\_ass\_dout_0, fun\_ass\_dout_1, fun\_ass\_dout_2\} \subset \{I \times V \to Bool\}$
  $fun\_ass\_ou \in \{fun\_ass\_ou_0, fun\_ass\_ou_1\} \subset \{I \times V \to N\}$
  $fun\_assign \in \{fun\_ass_1, fun\_ass_2, fun\_ass_3, fun\_ass_4, fun\_ass_5, fun\_ass_6, fun\_ass_7\}$

  $fun\_ass\_x_0(xi, yi, st, din, x, y) = x$ $\quad$ $fun\_ass\_dout_0(xi, yi, st, din, x, y) = \perp$
  $fun\_ass\_x_1(xi, yi, st, din, x, y) = xi$ $\quad$ $fun\_ass\_dout_1(xi, yi, st, din, x, y) =' 0'$
  $fun\_ass\_x_2(xi, yi, st, din, x, y) = x - y$ $\quad$ $fun\_ass\_dout_2(xi, yi, st, din, x, y) =' 1'$
  $fun\_ass\_y_0(xi, yi, st, din, x, y) = y$ $\quad$ $fun\_ass\_ou_0(xi, yi, st, din, x, y) = \perp$
  $fun\_ass\_y_1(xi, yi, st, din, x, y) = yi$ $\quad$ $fun\_ass\_ou_1(xi, yi, st, din, x, y) = x$
  $fun\_ass\_y_2(xi, yi, st, din, x, y) = y - x$

  $fun\_ass_1 = (fun\_ass\_x_1, fun\_ass\_y_0, fun\_ass\_dout_0, fun\_ass\_ou_0)$
  $fun\_ass_2 = (fun\_ass\_x_2, fun\_ass\_y_0, fun\_ass\_dout_0, fun\_ass\_ou_0)$
  $fun\_ass_3 = (fun\_ass\_x_0, fun\_ass\_y_1, fun\_ass\_dout_0, fun\_ass\_ou_0)$
  $fun\_ass_4 = (fun\_ass\_x_0, fun\_ass\_y_2, fun\_ass\_dout_0, fun\_ass\_ou_0)$
  $fun\_ass_5 = (fun\_ass\_x_0, fun\_ass\_y_0, fun\_ass\_dout_1, fun\_ass\_ou_0)$
  $fun\_ass_6 = (fun\_ass\_x_0, fun\_ass\_y_0, fun\_ass\_dout_2, fun\_ass\_ou_0)$
  $fun\_ass_7 = (fun\_ass\_x_0, fun\_ass\_y_0, fun\_ass\_dout_0, fun\_ass\_ou_1)$

- $fun\_cond_1, fun\_cond_2, fun\_cond_3, fun\_cond_4, fun\_cond_5$ sont les fonctions qui définissent les valeurs des conditions $cond_1$, $cond_2$, $cond_3$, $cond_4$, $cond_5$:

  $fun\_cond_1(xi, yi, st, din, x, y) = (st =' 1')$ $\quad$ $fun\_cond_4(xi, yi, st, din, x, y) = (x < y)$
  $fun\_cond_2(xi, yi, st, din, x, y) = (din =' 1')$ $\quad$ $fun\_cond_5(xi, yi, st, din, x, y) = (din =' 0')$
  $fun\_cond_3(xi, yi, st, din, x, y) = (x \neq y)$

- $TRANSITION$ est l'ensemble des séquences de fonctions d'affectations et de conditions:
  $TRANSITION = \{transition_1, transition_2, \ldots transition_{11}\}$ où:

$transition_1 = fun\_cond_1;\ fun\_ass_5$
$transition_2 = fun\_cond_1$
$transition_3 = fun\_cond_2;\ fun\_ass_1;\ fun\_ass_3;\ fun\_cond_3;\ fun\_cond_4;\ fun\_ass_4$
$transition_4 = fun\_cond_2;\ fun\_ass_1;\ fun\_ass_3;\ fun\_cond_3;\ fun\_cond_4;\ fun\_ass_2$
$transition_5 = fun\_cond_2;\ fun\_ass_1;\ fun\_ass_3;\ fun\_cond_3;\ fun\_ass_7;\ fun\_ass_6$
$transition_6 = fun\_cond_2$
$transition_7 = fun\_cond_3;\ fun\_cond_4;\ fun\_ass_4$
$transition_8 = fun\_cond_3;\ fun\_cond_4;\ fun\_ass_2$
$transition_9 = fun\_cond_3;\ fun\_ass_7;\ fun\_ass_6$
$transition_{10} = fun\_cond_5;\ fun\_ass_5$
$transition_{11} = fun\_cond_5;$

Étant données les définitions précédentes, le modèle intermédiaire est défini par le n-tuplet

$$< S, I, O, V, CONDITION, ASSIGN,$$
$$fun\_cond_1, fun\_cond_2, fun\_cond_3, fun\_cond_4, fun\_cond_5, TRANSITION, f, h >$$

où

– f est la fonction de transition:

$f(w_1,\ cond_1) = w_2;$
$f(w_1,\ \overline{cond_1}) = w_1;$
$f(w_2,\ cond_2 \& cond_3 \& cond_4) = w_3;$
$f(w_2,\ cond_2 \& cond_3 \& \overline{cond_4}) = w_3;$
$f(w_2,\ cond_2 \& \overline{cond_3}) = w_4;$
$f(w_2,\ \overline{cond_2}) = w_2;$
$f(w_3,\ cond_3 \& cond_4) = w_3;$
$f(w_3,\ cond_3 \& \overline{cond_4}) = w_3;$
$f(w_3,\ \overline{cond_3}) = w_4;$
$f(w_4,\ cond_5) = w_2;$
$f(w_4,\ \overline{cond_5}) = w_4;$

– h est la fonction de sortie:

$h(w_1,\ cond_1) = transition_1;$
$h(w_1,\ \overline{cond_1}) = transition_2;$
$h(w_2,\ cond_2 \& cond_3 \& cond_4) = transition_3;$
$h(w_2,\ cond_2 \& cond_3 \& \overline{cond_4}) = transition_4;$
$h(w_2,\ cond_2 \& \overline{cond_3}) = transition_5;$
$h(w_2,\ \overline{cond_2}) = transition_6;$
$h(w_3,\ cond_3 \& cond_4) = transition_7;$
$h(w_3,\ cond_3 \& \overline{cond_4}) = transition_8;$
$h(w_3,\ \overline{cond_3}) = transition_9;$
$h(w_4,\ cond_5) = transition_{10};$
$h(w_4,\ \overline{cond_5}) = transition_{11};$

Dans la figure 4.6 les transitions sont montrées avec leurs contenus: la police de caractères normale est utilisé pour les foncions d'affectations et la police en *italique* pour les fonctions des conditions. Un état initial supplémentaire n'est pas ajouté car la première instruction du processus VHDL est une instruction "wait".

Fin exemple

L'utilisation des procédures et des fonctions dans la description initiale est permise par le modèle formel. Le traitement est néanmoins différent selon l'interprétation d'une procédure/fonction. Si une procédure ou une fonction est associée avec un composant déjà synthétisé

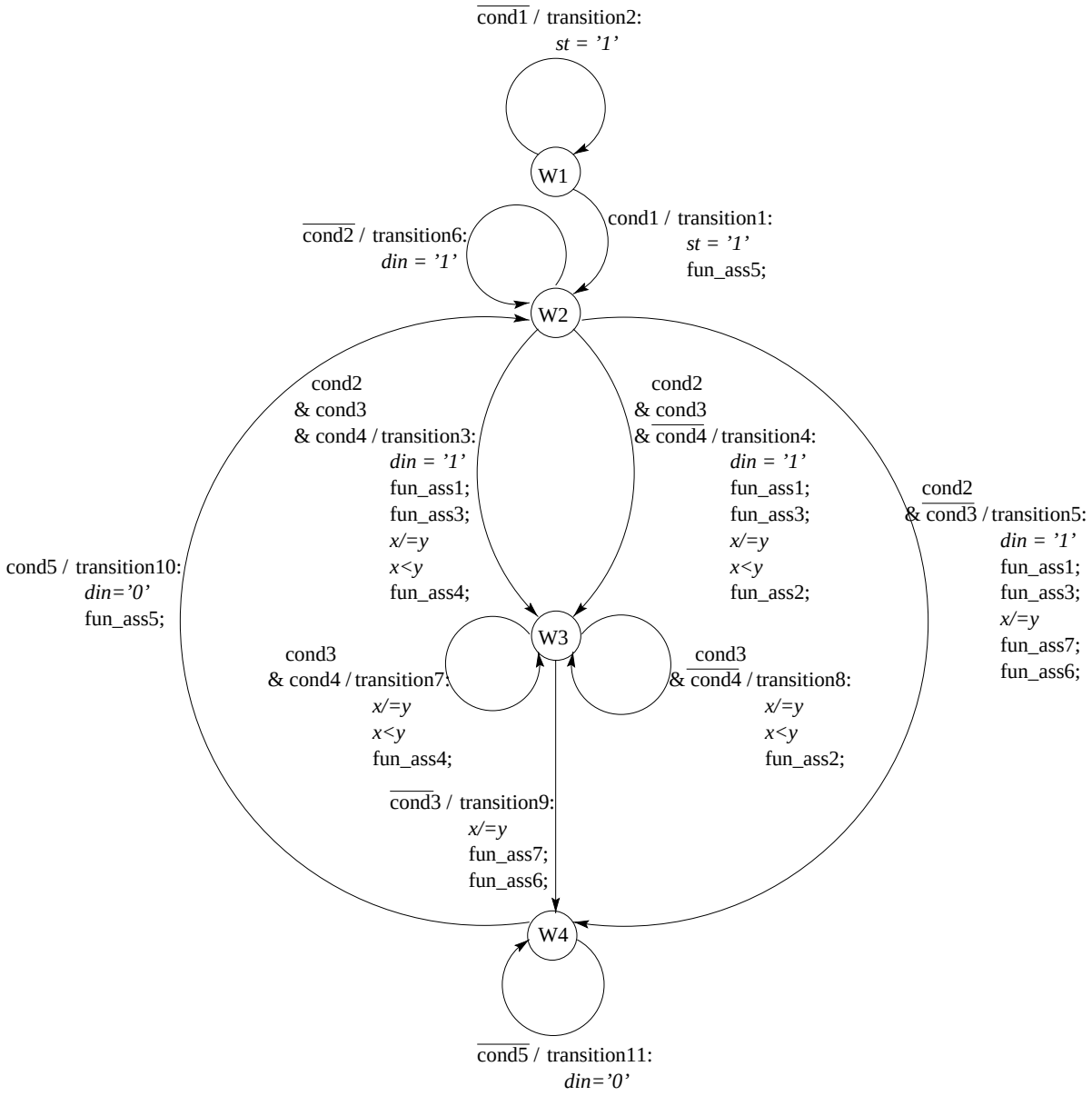


FIG. 4.6 – *Modèle formel intermédiaire correspondant à la description VHDL de Gcd*

et que ce composant est sensé être utilisé comme bloc dans un circuit englobant, alors la procédure/fonction devient une affectation élémentaire dans le modèle. Par exemple, la fonction $my\_function(a,b)$ affectée à la variable x dans le corps d'un processus ($x := my\_function(a,b)$) sera transformée en une affectation [3]: $fun\_ass\_x(\overrightarrow{i},\overrightarrow{v}) = my\_function(a,b)$ de signature $I \times V \rightarrow VVAL\_X$.

Une procédure sera transformées d'abord dans une ou plusieurs affectations élémentaires selon le nombre de paramètres sortants. Par exemple, si la procédure $my\_procedure(a,b,c,d,e)$ a trois paramètres d'entrée (a, b et c) et deux paramètres de sortie (d et e), alors elle se traduira par les deux affectations suivantes: $fun\_ass\_d(\overrightarrow{i},\overrightarrow{v}) = my\_procedure1(a,b,c)$ de signature $I \times V \rightarrow VVAL\_D$ et $fun\_ass\_e(\overrightarrow{i},\overrightarrow{v}) = my\_procedure2(a,b,c)$ de signature $I \times V \rightarrow VVAL\_E$. Ensuite les affectations élémentaires issues d'une procédure/fonction seront traitées comme les autres opérations de la description initiale.

Si une procédure ou une fonction n'est pas associée avec un composant et sensée être synthétisée avec le processus englobant, alors elle est mise à plat et la construction du modèle se poursuit de la même façon qu'avec les affectations simples (sans procédure ni fonction).

4.4 Construction de la machine abstraite du modèle intermédiaire

Cette section est dédiée aux transformations nécessaires pour passer du modèle intermédiaire au modèle de la machine abstraite.

La différence entre les deux modèles se trouve tout d'abord dans la nature des fonctions d'affectation associées aux transitions de ces modèles. Pour la machine abstraite, les affectations des variables et des sorties sont parallèles et constituent une fonction vectorielle $fun\_assign \in ASSIGN$. Le résultat de fun_assign est calculé à partir des valeurs des variables mémorisées au début d'une transition. Dans le modèle intermédiaire les fonctions d'affectation associées aux transitions forment une *séquence* d'affectations. Le résultat de chaque affectation est calculé à partir des valeurs courantes des variables mémorisées (par valeur courante nous entendons une valeur qui peut être acquise par une variable lors des affectations précédentes). La même différence distingue la fonction fun_status de la machine abstraite et les fonctions fun_cond_k ($1 \leq k \leq q$) du modèle intermédiaire: utilisation des valeurs initiales dans le premier cas et des valeurs courantes des variables mémorisées dans le second.

Pour avoir dans le modèle de la description initiale une seule fonction d'affectation (à la place d'une séquence de fonctions d'affectation et de condition) effectuée entre deux états, nous faisons une composition des affectations associées à chaque transition du modèle intermédiaire. Cette composition permettra également d'exprimer les valeurs des variables et des sorties à la fin de la transition à partir des valeurs des entrées et des variables mémorisées au début de la transition. Ainsi notre approche est similaire à celle de la sémantique dénotationnelle d'un programme ([Liv78]): nous construisons une fonction qui définit les résultats d'exécution après

3. Ici et ultérieurement, les notations $\overrightarrow{i}$ et $\overrightarrow{v}$ seront utilisées pour désigner les vecteurs de toutes les entrées et toutes les variables mémorisées. La notation $fun\_ass\_x_i(\overrightarrow{i},\overrightarrow{v})$ remplace alors la notation $fun\_ass\_x_i(i_1, i_2, \ldots, i_l, v_1, v_2, \ldots, v_n)$.

une transition. En revanche, nous n'avons pas jugé utile dans ce mémoire de reprendre toute la formalisation, assez lourde, de la sémantique dénotationnelle, comme cela était fait dans [BS95].

Au départ, la fonction fun_assign constituée des affectations d'identité et des $\perp$ forme le premier résultat de la composition. Par exemple, si un processus VHDL possède trois variables a, x et y et une sortie $out1$, alors les affectations initiales sont les fonctions suivantes: $fun\_ass\_a(\overrightarrow{i}, \overrightarrow{v}) = a$, $fun\_ass\_x(\overrightarrow{i}, \overrightarrow{v}) = x$, $fun\_ass\_y(\overrightarrow{i}, \overrightarrow{v}) = y$, et $fun\_ass\_out1(\overrightarrow{i}, \overrightarrow{v}) = \perp$. Remarquons que la fonction fun_assign est de la même nature que les fonctions d'affectation de la machine abstraite: elle contient les affectations parallèles de toutes les variables et toutes les sorties définies dans le modèle. Ensuite, chaque affectation d'une transition (du modèle intermédiaire) modifie fun_assign en substituant consécutivement les variables mémorisées par les valeurs acquises, exprimées sous la forme d'expressions mathématiques.

Exemple

Après la composition de
$transition_2 = (\ fun\_assign_1;\ fun\_cond_1;\ fun\_assign_2;\ fun\_cond_2;\ fun\_assign_5;\ fun\_assign_4\ )$
de la figure 4.2, les fonctions associées aux variables a, x, y et la sortie $out1$ deviennent:

$$fun\_ass\_a(\overrightarrow{i}, \overrightarrow{v}) = x + y; \qquad fun\_ass\_y(\overrightarrow{i}, \overrightarrow{v}) = 2 * (y - x);$$
$$fun\_ass\_x(\overrightarrow{i}, \overrightarrow{v}) = x; \qquad fun\_ass\_out1(\overrightarrow{i}, \overrightarrow{v}) = y - x;$$

Les fonctions de conditions engendrent les fonctions de statuts suivantes: $fun\_stat_1(\overrightarrow{i}, \overrightarrow{v}) = (in1 > 0)$ et $fun\_stat_2(\overrightarrow{i}, \overrightarrow{v}) = ((x + y) = 5)$ qui calculent les signaux de statuts à partir des valeurs des variables mémorisée au début de la transition. La formalisation de la composition est donnée ci-après.

Fin exemple

Soit une transition du modèle intermédiaire: $transition = (cur^1; cur^2; \ldots, cur^s; \ldots, cur^t)$ avec, pour chaque fonction $cur^s : I \times V \rightarrow FUN$ $(1 \leq s \leq t)$ (cur pour "current"):

$FUN = V \times O$ si cur^s est une fonction d'affectation fun_assign_i,

$FUN = COND_k$ si cur^s est une fonction de condition fun_cond_k.

La composition des fonctions cur^s de la séquence $transition$ construit une suite de fonctions fun_assign^1, fun_assign^2, ..., fun_assign^s, ..., fun_assign^t $(1 \leq s \leq t)$ avec:
$fun\_assign^s = (assign\_v^s, assign\_o^s) : I \times V \rightarrow V \times O$ où
$assign\_v^s = (fun\_ass\_v_1^s, \ldots fun\_ass\_v_n^s) : I \times V \rightarrow V$ sont des affectations des variables et
$assign\_o^s = (fun\_ass\_o_1^s, \ldots fun\_ass\_o_m^s) : I \times V \rightarrow O$ sont des affectations des sorties.

La distinction entre les affectations des variables et les affectations des sorties dans les fonctions fun_assign^s $(1 \leq s \leq t)$ nous est nécessaire pour la composition car seules les affectations des variables y participent. La fonction fun_assign^0 est la fonction initiale de la composition:
$fun\_ass\_v_i^0(i, v) = v_i$ $(1 \leq i \leq n)$: identité pour les variables mémorisées et
$fun\_ass\_o_j^0(i, v) = \perp$ $(1 \leq j \leq m)$: absence d'affectation pour les sorties.

Chaque fonction cur^s de $transition$ est composée avec la fonction fun_assign^{s-1} pour calculer des nouvelles fonctions fun_assign^s et fun_status^s. Si cur^s est une fonction d'affectation, alors chaque affectation élémentaire de cur^s composée avec fun_assign^{s-1} définit l'affectation

élémentaire correspondante de fun_assign^s. Si cur^s est une fonction de condition, alors sa composition avec fun_assign^{s-1} définit une nouvelle fonction de statut fun_stat et fun_assign^s est identique à fun_assign^{s-1}.

La dernière fonction fun_assign^t représente le vecteur des affectations parallèles pour une transition donnée. Le calcul itératif de la fonction fun_assign^t est donné par l'algorithme de composition ci-dessous. Pour mieux distinguer les fonctions de condition des fonctions de statut, les premières sont indexées de 1 jusqu'à q et les deuxièmes de 1 jusqu'à p.

Algorithme

Initialisation:

$fun\_assign^0 : I \times V \to V \times O$

$fun\_status^0 = fun\_status : I \times V \to STATUS^0$

Pour s de 1 jusqu'à t faire:

$fun\_assign^s = cur^s \odot fun\_assign^{s-1} : I \times V \to V \times O$

$fun\_status^s = cur^s \otimes fun\_status^{s-1} : I \times V \to STATUS^s$

Fin pour.

Fin algorithme

Chaque itération contient deux opérations:

I L'opération $\odot$ définit un nouveau résultat intermédiaire $fun\_assign^s = cur^s \odot fun\_assign^{s-1}$ où chaque élément est la composition [4] de l' élément correspondant de cur^s avec les affectations précédentes des variables mémorisées, si cur^s est une fonction d'affectation:

- si $cur^s = (cur\_ass\_v_1^s, \ldots, cur\_ass\_v_n^s, cur\_ass\_o_1^s, \ldots, cur\_ass\_o_m^s)$
  est une fonction d'affectation, alors
  $fun\_ass\_v_i^s = cur\_ass\_v_i^s \circ assign\_v^{s-1}$ $(1 \leq i \leq n)$;
  $fun\_ass\_o_j^s = cur\_ass\_o_j^s \circ assign\_v^{s-1}$ $(1 \leq j \leq m)$
- sinon $fun\_assign^s = fun\_assign^{s-1}$

II L'opération $\otimes$ ajoute une nouvelle fonction de statut $new\_fun^s = cur^s \circ assign\_v^{s-1}$ à la fonction vectorielle fun_status si cur^s est une fonction de condition et si new_fun^s est différente de toutes les fonctions de statuts déjà existantes:

- si cur^s est une fonction de condition fun_cond_k associée à la variable $cond_k$, alors

  1. $new\_fun^s = cur^s \circ assign\_v^{s-1}$
  2. si $new\_fun^s \neq fun\_stat_i^{s-1}$ $(1 \leq i \leq p)$, alors
     $fun\_status^s = (fun\_stat_1^{s-1}, \ldots, fun\_stat_p^{s-1}, new\_fun^s)$,
     sinon $fun\_status^s = fun\_status^{s-1}$

4. Pour alléger les écritures nous omettons ici la démarche formelle de la composition qui consiste à passer aux fonctions curryfiées, les composer, décurryfier le résultat de la composition et introduire une nouvelle fonction qui est une restriction de la composition aux arguments égaux. Nous avons fait cette démarche dans le chapitre précédent pendant la démonstration d'équivalence entre la machine abstraite et la machine d'états finis classique.

– sinon $fun\_status^s = fun\_status^{s-1}$

Après l'opération $\otimes$, la variable $cond_k$ $(1 \leq k \leq q)$ correspond à une variable $stat_l$ $(1 \leq l \leq p)$ et doit être remplacée par cette variable $stat_l$ dans l'expression booléenne (représentant une valeur de la variable *condition*) associée à la transition traitée.

Dans la définition de l'opération $\odot$ chaque affectation élémentaire de cur^s est composée avec le résultat précédent fun_assign^{s-1} pour définir la composante correspondante de la nouvelle fonction fun_assign^s. Cependant, au plus une composante de fun_assign^s est différente de la composante analogue de fun_assign^{s-1}, car une seule affectation élémentaire de cur^s est distincte de fonctions d'identité et de $\perp$. La forme actuelle de l'algorithme nous sera utile, néanmoins, pour la composition des transitions de la machine abstraite (présentée dans le chapitre suivant), où des fonctions d'affectations sont aléatoires et contiennent, en général, plusieurs affectations différentes de fonctions d'identité et de $\perp$.

La fonction $fun\_assign^t = (fun\_ass\_v_1^t, \ldots, fun\_ass\_v_n^t,\ fun\_ass\_o_1^t \ldots, fun\_ass\_o_m^t)$ représente le vecteur des affectations accumulées et effectuées à partir des valeurs des variables au début de la transition. Ainsi, l'algorithme de composition transforme l'ensemble des affectations **consécutives** du modèle intermédiaire en l'ensemble des affectations **parallèles** de la machine abstraite. Les fonctions $fun\_ass\_v_1^t, \ldots, fun\_ass\_v_n^t$, $fun\_ass\_o_1^t \ldots, fun\_ass\_o_m^t$ font partie des ensembles $ASS\_V_1, \ldots ASS\_V_n$, $ASS\_O_1, \ldots ASS\_O_m$ des affectations du modèle de la machine abstraite.

Pour alléger l'algorithme, nous avons omis le problème de la désignation des nouvelles fonctions fun_assign. Comme les fonctions de statuts fun_stat_k, les fonctions d'affectations devraient avoir un nouveau système de désignation pour associer à chaque nouvelle fonction d'affectation un nouveau nom, utilisé ensuite dans la machine abstraite après la composition.

Les fonctions de statuts, elles, sont également définies relativement aux valeurs des variables au début de la transition. Avant le traitement de la toute première transition, $p = 0$. Ensuite l'algorithme crée progressivement la fonction vectorielle fun_status, augmentant p. Le changement des variables de condition $cond_k$ par les variables de statut $stat_l$ est le point important dans l'algorithme.

Les transformations du modèle intermédiaire sont illustrées par l'exemple "du plus grand diviseur commun" (Gcd).

Exemple

Après les transformations, le modèle initial Gcd de la figure 4.6 prend la forme montrée dans la figure 4.7 et contient les définitions suivantes:

– S, I, O, V sont inchangés

– $STATUS$ est l'ensemble des symboles des statuts:
$status = (stat_1, stat_2, stat_3, stat_4, stat_5, stat_6, stat_7) \in STATUS = Bool^7$

– $ASSIGN$ est l'ensemble des affectations[5] de la variable v et de la sortie o:
$fun\_assign = (fun\_ass\_x, fun\_ass\_y, fun\_ass\_dout, fun\_ass\_ou) : I \times V \mapsto V \times O$ où

$fun\_ass\_x \in ASS\_X = \{fun\_ass\_x_0,\ fun\_ass\_x_1,\ fun\_ass\_x_2,\ fun\_ass\_x_3\}$
$fun\_ass\_y \in ASS\_Y = \{fun\_ass\_y_0,\ fun\_ass\_y_1,\ fun\_ass\_y_2,\ fun\_ass\_y_3\}$
$fun\_ass\_dout \in ASS\_DOUT = \{fun\_ass\_dout_0,\ fun\_ass\_dout_1,\ fun\_ass\_dout_2\}$
$fun\_ass\_ou \in ASS\_OU = \{fun\_ass\_ou_0,\ fun\_ass\_ou_1,\ fun\_ass\_ou_2\}$
$fun\_assign \in ASSIGN = \{fun\_ass_0, fun\_ass_1, fun\_ass_2, fun\_ass_3, fun\_ass_4, fun\_ass_5,$
$fun\_ass_6,\ fun\_ass_7\}$

$fun\_ass\_x_0(\overrightarrow{i}, \overrightarrow{v}) = x$
$fun\_ass\_x_1(\overrightarrow{i}, \overrightarrow{v}) = xi$
$fun\_ass\_x_2(\overrightarrow{i}, \overrightarrow{v}) = xi - yi$
$fun\_ass\_x_3(\overrightarrow{i}, \overrightarrow{v}) = x - y$
$fun\_ass\_y_0(\overrightarrow{i}, \overrightarrow{v}) = y$
$fun\_ass\_y_1(\overrightarrow{i}, \overrightarrow{v}) = yi$
$fun\_ass\_y_2(\overrightarrow{i}, \overrightarrow{v}) = yi - xi$
$fun\_ass\_y_3(\overrightarrow{i}, \overrightarrow{v}) = y - x$

$fun\_ass\_dout_0(\overrightarrow{i}, \overrightarrow{v}) = \perp$
$fun\_ass\_dout_1(\overrightarrow{i}, \overrightarrow{v}) =' 0'$
$fun\_ass\_dout_2(\overrightarrow{i}, \overrightarrow{v}) =' 1'$
$fun\_ass\_ou_0(\overrightarrow{i}, \overrightarrow{v}) = \perp$
$fun\_ass\_ou_1(\overrightarrow{i}, \overrightarrow{v}) = xi$
$fun\_ass\_ou_2(\overrightarrow{i}, \overrightarrow{v}) = x$

$fun\_ass_0 = (fun\_ass\_x_0,\ fun\_ass\_y_0,\ fun\_ass\_dout_0,\ fun\_ass\_ou_0)$
$fun\_ass_1 = (fun\_ass\_x_0,\ fun\_ass\_y_0,\ fun\_ass\_dout_1,\ fun\_ass\_ou_0)$
$fun\_ass_2 = (fun\_ass\_x_1,\ fun\_ass\_y_2,\ fun\_ass\_dout_0,\ fun\_ass\_ou_0)$
$fun\_ass_3 = (fun\_ass\_x_2,\ fun\_ass\_y_1,\ fun\_ass\_dout_0,\ fun\_ass\_ou_0)$
$fun\_ass_4 = (fun\_ass\_x_1,\ fun\_ass\_y_1,\ fun\_ass\_dout_2,\ fun\_ass\_ou_1)$
$fun\_ass_5 = (fun\_ass\_x_0,\ fun\_ass\_y_3,\ fun\_ass\_dout_0,\ fun\_ass\_ou_0)$
$fun\_ass_6 = (fun\_ass\_x_3,\ fun\_ass\_y_0,\ fun\_ass\_dout_0,\ fun\_ass\_ou_0)$
$fun\_ass_7 = (fun\_ass\_x_0,\ fun\_ass\_y_0,\ fun\_ass\_dout_2,\ fun\_ass\_ou_2)$

Étant données les définitions précédentes, la machine abstraite $M_{ab-init}$ après les transformations du modèle intermédiaire est définit par le n-tuplet:

$$M_{ab-init} =< S, I, O, V, STATUS, ASSIGN, fun\_status, f, h >$$

où

- $fun\_status : I \times V \mapsto Bool^7$ est le vecteur de sept fonctions booléennes:
  $fun\_status = (fun\_stat_1, fun\_stat_2, fun\_stat_3, fun\_stat_4, fun\_stat_5, fun\_stat_6, fun\_stat_7)$
  où

$fun\_stat_1(\overrightarrow{i}, \overrightarrow{v}) = (st =' 1')$
$fun\_stat_2(\overrightarrow{i}, \overrightarrow{v}) = (din =' 1')$
$fun\_stat_3(\overrightarrow{i}, \overrightarrow{v}) = (x \neq y)$
$fun\_stat_4(\overrightarrow{i}, \overrightarrow{v}) = (x < y)$

$fun\_stat_5(\overrightarrow{i}, \overrightarrow{v}) = (din =' 0')$
$fun\_stat_6(\overrightarrow{i}, \overrightarrow{v}) = (xi \neq yi)$
$fun\_stat_7(\overrightarrow{i}, \overrightarrow{v}) = (xi < yi)$

- f est la fonction de transition:

5. Dans cet exemple la notation $(\overrightarrow{i}, \overrightarrow{v})$ signifie (xi, yi, st, din, x, y). Ainsi $fun\_ass\_x_1(\overrightarrow{i}, \overrightarrow{v}) = x$ doit être compris comme $fun\_ass\_x_1(xi, yi, st, din, x, y) = x$, etc.

$f(w_1, stat_1) = w_2$;
$f(w_1, \overline{stat_1}) = w_1$;
$f(w_2, stat_2 \& stat_6 \& stat_7) = w_3$;
$f(w_2, stat_2 \& stat_6 \& \overline{stat_7}) = w_3$;
$f(w_2, stat_2 \& \overline{stat_6}) = w_4$;
$f(w_2, \overline{stat_2}) = w_2$;
$f(w_3, stat_3 \& stat_4) = w_3$;
$f(w_3, stat_3 \& \overline{stat_4}) = w_3$;
$f(w_3, \overline{stat_3}) = w_4$;
$f(w_4, stat_5) = w_2$;
$f(w_4, \overline{stat_5}) = w_4$;

– h est la fonction de sortie:

$h(w_1, stat_1) = fun\_ass_1$;
$h(w_1, \overline{stat_1}) = fun\_ass_0$;
$h(w_2, stat_2 \& stat_6 \& stat_7) = fun\_ass_2$;
$h(w_2, stat_2 \& stat_6 \& \overline{stat_7}) = fun\_ass_3$;
$h(w_2, stat_2 \& \overline{stat_6}) = fun\_ass_4$;
$h(w_2, \overline{stat_2}) = fun\_ass_0$;
$h(w_3, stat_3 \& stat_4) = fun\_ass_5$;
$h(w_3, stat_3 \& \overline{stat_4}) = fun\_ass_6$;
$h(w_3, \overline{stat_3}) = fun\_ass_7$;
$h(w_4, stat_5) = fun\_ass_1$;
$h(w_4, \overline{stat_5}) = fun\_ass_0$;

Pour cet exemple nous avons choisi le même style de présentation que pour l'exemple de la machine abstraite dans le chapitre précédent. Chaque transition est étiquetée par les fonctions des statuts et les affectations des variables et des sorties. Pour alléger le dessin, pour les transitions $W_1 \rightarrow W_1$, $W_2 \rightarrow W_2$, $W_4 \rightarrow W_4$, on garde sous la forme fun_ass_0 la fonction d'identité sur les variables et $\perp$ sur les sorties (figure 4.7).

En outre, le nombre de fonctions de statuts a augmenté par rapport au nombre de fonctions de conditions du modèle intermédiaire. La raison en est la modification des fonctions des conditions associées aux variables $cond_3$ et $cond_4$. En effet, pendant la composition des fonctions des transitions $W_2 \rightarrow W_3$ (les deux transitions $W_2 \rightarrow W_3$ sont concernées) et $W_2 \rightarrow W_4$, les fonctions de condition $(x \neq y)$ et $(x < y)$ sont transformées en $(xi \neq yi)$ et $(xi < yi)$ en raison des affectations précédentes des variables x et y. Puisque les variables $stat_3$ et $stat_4$ sont déjà associées aux fonctions de statut $fun\_stat_3 = (x \neq y)$ et $fun\_stat_4 = (x < y)$, on choisit alors pour les fonctions $(xi \neq yi)$ et $(xi < yi)$ de nouvelles variables de statut $stat_6$ et $stat_7$ et on leur associe les fonctions fun_stat_6 et fun_stat_7. Les variables $stat_6$, $stat_7$ et les fonctions associées font partie du nouveau vecteur des statuts $status$ et de la fonction correspondante fun_status.

Fin exemple

L'exemple précédent montre que les transformations du modèle intermédiaire attribuent les noms pour les variables des statuts et pour les fonctions correspondantes qui ne sont pas forcément les mêmes que dans la machine abstraite après l'ordonnancement. Le changement du nom de statut ne modifie pas, néanmoins, le principe de la vérification de cette étape. Ce principe consiste à trouver dans la machine abstraite après l'ordonnancement une transition avec les mêmes valeurs des statuts et les mêmes fonctions de statuts associées à ces valeurs, c'est-à-dire une transition avec les mêmes couples valeur/fonction.

Par exemple, la vérification de la transition $W_2 \rightarrow W_4$ de la machine abstraite du Gcd (figure 4.7) consistera à trouver une transition $W_2' \rightarrow W_4'$ dans la machine abstraite après l'ordonnancement avec les valeurs des statuts $true$ et $false$ qui correspondent aux fonctions des statuts suivantes: $(din =' 1')$ (correspond à $true$) et $(xi \neq yi)$ (correspond à $false$). L'identification des

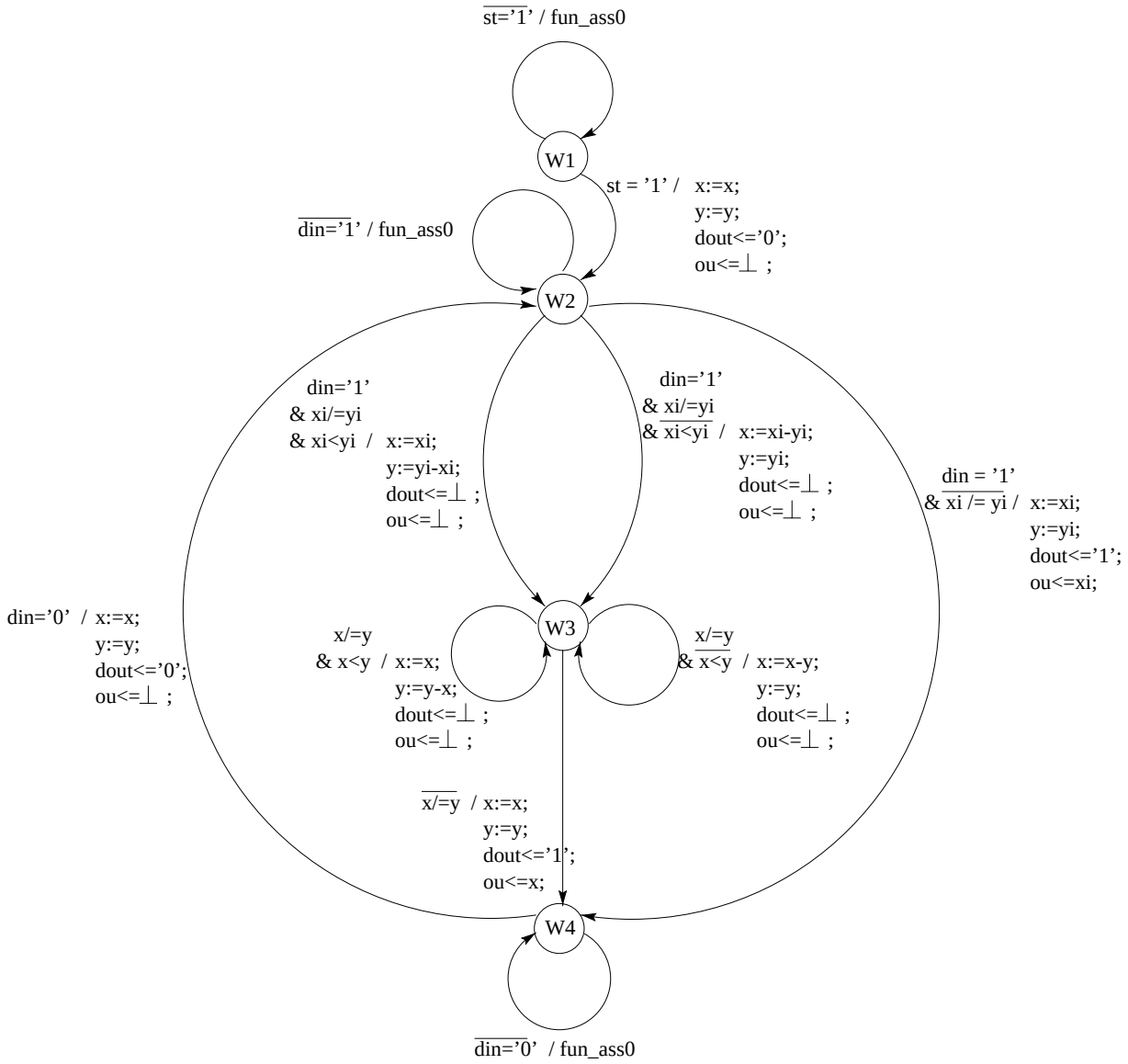


FIG. 4.7 – *Modèle formel transformé (machine abstraite) de la description VHDL de Gcd*

fonctions de statuts correspondantes est de complexité $O(p^2)$ où p est le nombre de statuts[6]. Pour pallier cet inconvénient, des noms identiques seront attribués aux fonctions de statuts sémantiquement équivalentes.

Les noms des statuts sont utilisés, en outre, pour valider la description initiale VHDL. La validation consiste à (ré)vérifier les conditions de déterminisme pour la machine abstraite après les transformations. Dans l'exemple précédent, les prédicats $cond_2\&cond_3\&cond_4$, $cond_2\&cond_3\&\overline{cond_4}$, $cond_2\&\overline{cond_3}$ et $\overline{cond_2}$ sur les transitions issues de l'état W_2 du modèle intermédiaire changent dans la machine abstraite pour $stat_2\&stat_6\&stat_7$, $stat_2\&stat_6\&\overline{stat_7}$, $stat_2\&\overline{stat_6}$ et $\overline{stat_2}$. La machine reste, néanmoins, déterministe concernant l'état W_2:
(i) $(stat_2\&stat_6\&stat_7) \vee (stat_2\&stat_6\&\overline{stat_7}) \vee (stat_2\&\overline{stat_6}) \vee \overline{stat_2} = 1$
(ii) $[(stat_2\&stat_6\&stat_7)\&(stat_2\&stat_6\&\overline{stat_7})] = 0$, $[(stat_2\&stat_6\&stat_7)\&(stat_2\&\overline{stat_6})] = 0$, $[(stat_2\&stat_6\&stat_7)\&\overline{stat_2})] = 0$, $[(stat_2\&stat_6\&\overline{stat_7})\&(stat_2\&\overline{stat_6})] = 0$,

6. Dans le pire des cas, chaque statut d'une machine doit être comparé avec chaque statut de l'autre. D'où la complexité $O(p^2)$.

$[(stat_2 \& stat_6 \& \overline{stat_7}) \& \overline{stat_2}] = 0$, $[(stat_2 \& \overline{stat_6}) \& \overline{stat_2}] = 0$

Pour conserver le déterminisme inhérent à la description initiale dans la machine abstraite correspondante, certaines constructions de VHDL doivent être traitées de manière particulière lors de la construction du modèle intermédiaire. Ainsi, la construction "case" de la figure 4.8.a doit générer les prédicats, montrés par la figure 4.8.b, qui satisfont les conditions de déterminisme pour l'état S_i. L'assymétrie des prédicats attachés aux transitions vient du fait que les branches de l'instruction "case" sont examinées dans l'ordre défini par l'utilisateur. Une branche est exécutée si et seulement si son prédicat est vrai et les prédicats de toutes les branches précédentes sont faux. Cette exécution ajoute au prédicat de la branche courante les négations des prédicats des branches précédentes.

Remarquons que cette même instruction "case" peut être représenté par la machine abstraite montrée sur la figure 4.8.c. L'unique variable de statut *stat* appartient à l'ensemble $STAT = \{0, 1, 2, 3, 4, 5, 6, 7\}$ de valeurs autres que $true$ et $false$. La variable X est sensée appartenir au même ensemble $STAT$. Or, les expressions booléennes ne sont plus convenables pour désigner les valeurs de la variable *status* (comme dans la figure4.8.b, où $\overline{stat_1} \& \overline{stat_2} \& stat_3$ désigne la valeur $(false, false, true)$), les valeurs explicites sont employées pour marquer les transitions de la machine abstraite de la figure 4.8.c. Les raisonnements de la synthèse logique basés sur l'algèbre de Boole ne peuvent plus, cependant, être appliqués à ce modèle.

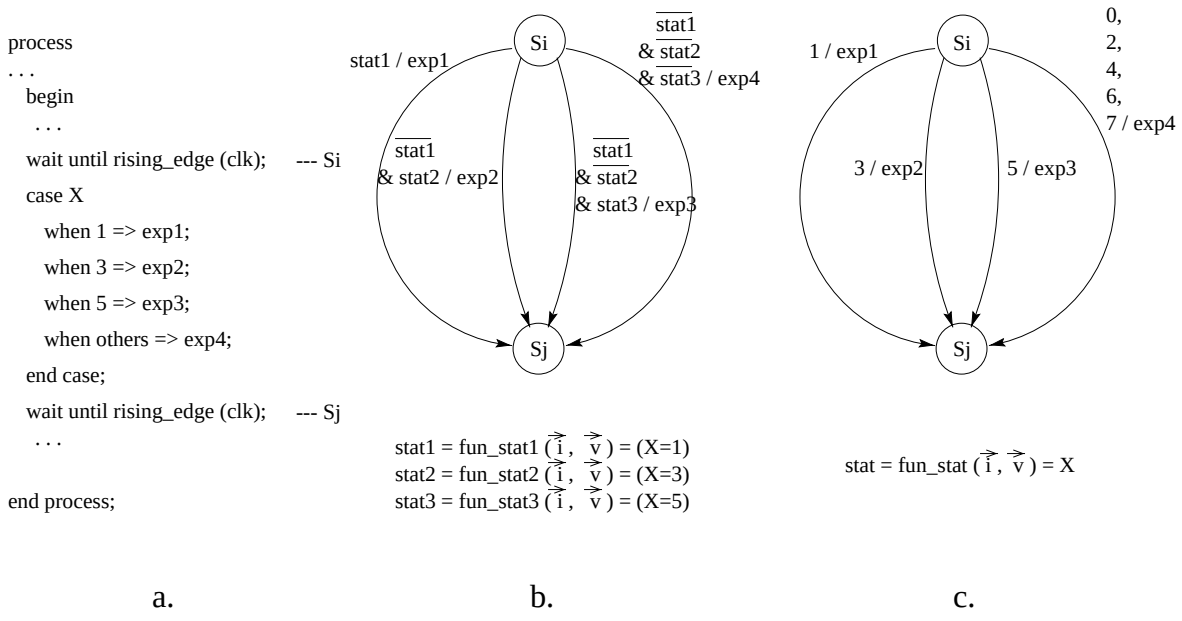


FIG. 4.8 – *Instruction "case"* ***(a.);*** *modèle intermédiaire correspondant* ***(b.)*** *et modèle intermédiaire avec le statut non-booléen* ***(c.)***

4.5 Composition des transitions insérées lors de l'ordonnancement

La machine abstraite construite à partir de la description initiale constitue une **spécification formelle** du circuit avant synthèse. La preuve de l'exactitude de l'ordonnancement consiste

donc à prouver l'équivalence entre la machine abstraite avant l'ordonnancement et la machine abstraite après l'ordonnancement.

Cependant, selon la théorie classique des machine d'états finis ([Hen68], [HAFM97]), la définition d'équivalence demande la même échelle de temps pour les deux machines comparées: deux machine sont équivalentes si elles produisent les mêmes sorties pour toutes entrées possibles. Cette définition suppose que pour les deux machines les sorties soient produites à la même fréquence. Dans notre cas une transition de la spécification peut être raffinée en plusieurs transitions de la machine abstraite obtenue après l'ordonnancement. En conséquence, la notion classique d'équivalence ne peut pas être directement appliquée.

La solution proposée consiste à ramener la machine abstraite après l'ordonnancement à la forme d'une machine abstraite ayant le même nombre d'états que la machine abstraite initiale, en composant les transitions insérées lors de l'ordonnancement. Ainsi, "l'échelle de temps" devient la même pour les deux machines. Ultérieurement, nous appelons *états initiaux* les états de la machine abstraite avant l'ordonnancement et *états insérés* les états supplémentaires de la machine abstraite après l'ordonnancement. Les transitions supplémentaires sont appelées *transitions insérées*.

Nous définissons la composition des transitions insérées de la même façon que la composition des affectations séquentielles du modèle intermédiaire (section précédente): les fonctions fun_assign attachées aux transitions insérées et les fonctions de statuts fun_stat rencontrées lors du passage d'un état initial à l'autre, forment des chemins d'exécution appelés *paths*. Chaque chemin constitue ensuite l'objet de la composition et se transforme en une transition directe entre les états initiaux après la composition.

Le chemin d'exécution *path* de la machine abstraite est similaire au chemin d'exécution *transition* du modèle intermédiaire. En effet, les deux chemins d'exécution sont définis comme une séquence de fonctions d'affectations et de foncions de condition. Toutefois, pour éviter toute ambiguïté nous utilisons les notations différentes.

La définition de chaque fonction de *path* dépend, comme dans le cas de *transition*, de la position dans une séquence de fonctions d'affectations et de statuts. Le but de la composition des transitions insérées est de redéfinir chaque fonction relativement aux valeurs des variables mémorisées au début du chemin d'exécution donné.

La construction d'un chemin d'exécution *path* est effectuée à partir des fonctions d'affectations et des fonctions de statuts associées aux transitions composant le *path*. À chaque transition de *path* est associée exactement une fonction d'affectation. Le nombre de fonctions de statuts dépend néanmoins de l'expression booléenne[7] attachée à une transition: à chaque variable primitive de l'expression est associée une fonction de statut. Si l'expression booléenne est *true*, alors aucune fonction de statut ne correspond à la transition.

Lors de la construction du *path*, les fonctions de statuts associées à une transition composante doivent être placées *avant* la fonction d'affectation associée à cette même transition car, par sa nature, la fonction d'affectation est définie à partir des résultats des fonctions de statuts.

Après la composition des transitions d'un chemin d'exécution *path*, la nouvelle transition

7. Nous rappelons qu'une expression booléenne associée à une transition de la machine abstraite est dérivée à partir des définitions formelles des fonctions de transition et de sortie. Elle est constituée des variables primitives de statuts ($stat_1$, $stat_2$, ...) et définit les valeurs de la variable *status* (paragraphe 3.4).

directe doit être associée à une expression booléenne représentant la valeur de la variable *status*. Cette expression est équivalente à la conjonction des expressions booléennes des transitions composantes.

Exemple

Supposons que l'extrait de la description initiale de la figure 4.2 ait été raffiné en l'extrait de la machine abstraite montrée dans la figure 4.9. Les états S_1 et S_2 ont été insérés entre les états initiaux W_1 et W_2 pendant l'ordonnancement. Les variables de statuts $stat_1$ et $stat_2$ correspondent aux fonctions $fun\_stat_1\,(\overrightarrow{i}, \overrightarrow{v}) = (in1 > 0)$ et $fun\_stat_2\,(\overrightarrow{i}, \overrightarrow{v}) = (a = 5)$.

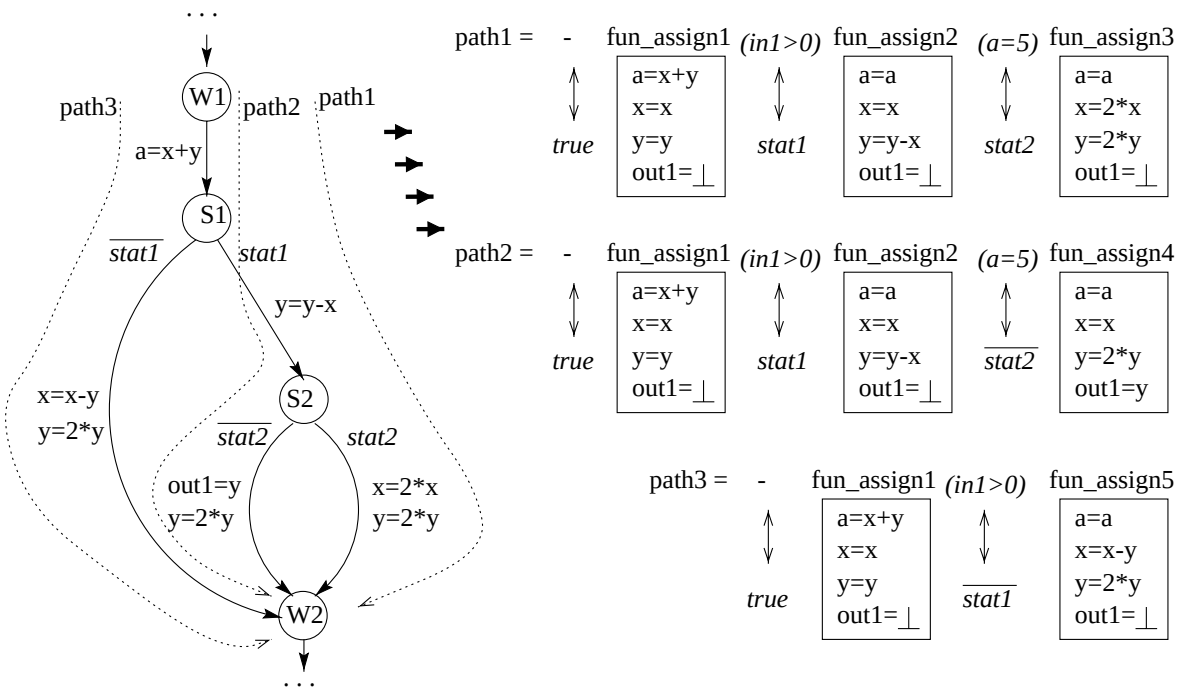


FIG. 4.9 – *Transitions insérées lors de l'ordonnancement de la figure 4.2*

La composition des transitions insérées commence par la recherche de tous les chemins d'exécution entre les états $W1$ et $W2$: il en existe trois marqués dans la figure 4.9 par $path_1$, $path_2$ et $path_3$. Le chemin d'exécution $path_2$ consiste, par exemple, en fonctions d'affectations et de statuts suivantes: $path_2 = ((fun\_assign_1), (fun\_stat_1, fun\_assign_2), (fun\_stat_2, fun\_assign_4))$. Les sous-séquences[8] des fonctions (délimitées par les parenthèses internes) correspondent aux transitions du $path_2$: $W_1 \rightarrow S_1$, $S_1 \rightarrow S_2$ et $S_2 \rightarrow W_2$. Les fonctions d'affectations (fun_assign_1, fun_assign_2 et fun_assign_4) sont complétées par les affectations initiales telles que $a = a$, $x = x$, $y = y$ et $out1 = \perp$. Les fonctions de statuts sont dérivées à partir des expressions booléennes, respectivement $true$, $stat_1$ et $\overline{stat_2}$, associées aux transitions $W_1 \rightarrow S_1$, $S_1 \rightarrow S_2$ et $S_2 \rightarrow W_2$.

Les chemins d'exécution se transformeront après la composition en trois transitions directes entre les états W_1 et W_2. Ainsi, la transition directe issue du $path_2$ sera associée à la nouvelle fonc-

8. Les sous-séquences sont utilisées uniquement pour montrer la correspondance entre les fonctions du *path* et les transitions le composant. Lors du traitement des fonctions par l'algorithme de composition, le *path* est considéré comme une séquence homogène de fonctions, les transitions correspondantes sont sans importance. L'ordre des fonctions dans la séquence homogène doit, néanmoins, être préservé tel qu'il est avec les sous-séquences.

tion fun_assign définie par: $fun\_assign = (fun\_ass\_a, fun\_ass\_x, fun\_ass\_y, fun\_ass\_out1)$ où $fun\_ass\_a(\overrightarrow{i}, \overrightarrow{v}) = x + y$, $fun\_ass\_x(\overrightarrow{i}, \overrightarrow{v}) = x$, $fun\_ass\_y(\overrightarrow{i}, \overrightarrow{v}) = 2 * (y - x)$ et $fun\_ass\_out1(\overrightarrow{i}, \overrightarrow{v}) = y - x$. De plus, après la composition les fonctions fun_stat_1 et fun_stat_2 deviennent $fun\_stat_1(\overrightarrow{i}, \overrightarrow{v}) = (in1 > 0)$ et $fun\_stat_2(\overrightarrow{i}, \overrightarrow{v}) = ((x + y) = 5)$. Enfin, l'expression booléenne associée à la nouvelle transition directe est définie par $true\&stat_1\&\overline{stat_2} = stat_1\&\overline{stat_2}$.

Fin exemple

Une fois un chemin d'exécution *path* construit, l'algorithme de composition de la section précédente lui est appliqué pour obtenir les fonctions d'affectation et les fonctions de statuts associées à la nouvelle transition. Pour ce faire, nous allons appeler les fonctions et les variables de statuts *avant* la composition $fun\_condition = (fun\_cond_1, \ldots, fun\_con_p)$ et $condition = (cond_1, \ldots, cond_p)$ en réservant les noms $fun\_status = (fun\_stat_1, \ldots, fun\_stat_q)$ et $status = (stat_1, \ldots, stat_q)$ pour les fonctions et les variables de statuts *après* la composition. Après ce changement de notations, l'algorithme de la section précédente est directement utilisé pour composer les transitions insérées: le chemin d'exécution de la machine abstraite *path* joue le rôle du chemin d'exécution du modèle intermédiaire *transition*. Comme la *transition*, le *path* consiste en des fonctions composantes appelées cur^1, cur^2, ... , cur^t.

Remarquons, que pour la vérification de l'étape d'ordonnancement nous utilisons la version de la machine abstraite avec les fonctions d'affectation $fun\_assign \in ASSIGN \subset \{I \times V \rightarrow V \times O\}$(section 3.3, page 29). Cela signifie que les décisions architecturales concernant la largeur des entrées, des sorties et des variables mémorisées ne sont pas encore prises pour la machine abstraite obtenue après l'ordonnancement. La version simplifiée de la machine abstraite nous permet de plus facilement composer ses fonctions et unifier les deux algorithmes de composition (du chemin d'exécution *transition* du modèle intermédiaire et du chemin d'exécution *path* de la machine abstraite).

Par la suite nous continuons l'exemple du plus grand diviseur commun (Gcd). Les transitions du modèle obtenu après l'ordonnancement sont composées, après quoi les deux modèles, avant et après l'étape d'ordonnancement, peuvent être comparés.

Exemple

Après l'ordonnancement de la description comportementale du Gcd, Amical rend[9] en sortie la description montrée dans l'annexe A qui est une machine abstraite illustrée par la figure 4.10. Cette machine contient les définitions suivantes:

- S, I, O, V sont inchangés
- $CONDITION$ est l'ensemble des symboles des statuts:
  $condition = (cond_1, cond_2, cond_3, cond_4, cond_5) \in CONDITION = Bool^5$
- $ASSIGN$ est l'ensemble des affectations de la variable v et de la sortie o:
  $fun\_assign = (fun\_ass\_x, fun\_ass\_y, fun\_ass\_dout, fun\_ass\_ou) : I \times V \mapsto V \times O$ où

9. Ici l'ancienne version d'Amical utilisée ne fournissait pas une sortie en VHDL après l'ordonnancement. Par conséquent, la description du Gcd après l'ordonnancement est donnée en langage interne SOLAR.

$fun\_ass\_x \in ASS\_X = \{fun\_ass\_x_0,\ fun\_ass\_x_1,\ fun\_ass\_x_2\}$
$fun\_ass\_y \in ASS\_Y = \{fun\_ass\_y_0,\ fun\_ass\_y_1,\ fun\_ass\_y_2\}$
$fun\_ass\_dout \in ASS\_DOUT = \{fun\_ass\_dout_0,\ fun\_ass\_dout_1,\ fun\_ass\_dout_2\}$
$fun\_ass\_ou \in ASS\_OU = \{fun\_ass\_ou_0,\ fun\_ass\_ou_1\}$
$fun\_assign \in ASSIGN = \{fun\_ass_0, fun\_ass_1, fun\_ass_2, fun\_ass_3, fun\_ass_4, fun\_ass_5\}$

$fun\_ass\_x_0(\overrightarrow{i}, \overrightarrow{v}) = x$ $\quad$ $fun\_ass\_y_0(\overrightarrow{i}, \overrightarrow{v}) = y$ $\quad$ $fun\_ass\_dout_0(\overrightarrow{i}, \overrightarrow{v}) = \perp$
$fun\_ass\_x_1(\overrightarrow{i}, \overrightarrow{v}) = xi$ $\quad$ $fun\_ass\_y_1(\overrightarrow{i}, \overrightarrow{v}) = yi$ $\quad$ $fun\_ass\_dout_1(\overrightarrow{i}, \overrightarrow{v}) =' 0'$
$fun\_ass\_x_2(\overrightarrow{i}, \overrightarrow{v}) = x - y$ $\quad$ $fun\_ass\_y_2(\overrightarrow{i}, \overrightarrow{v}) = y - x$ $\quad$ $fun\_ass\_dout_2(\overrightarrow{i}, \overrightarrow{v}) =' 1'$
$fun\_ass\_ou_0(\overrightarrow{i}, \overrightarrow{v}) = \perp$
$fun\_ass\_ou_1(\overrightarrow{i}, \overrightarrow{v}) = x$

$fun\_ass_0 = (fun\_ass\_x_0,\ fun\_ass\_y_0,\ fun\_ass\_dout_0,\ fun\_ass\_ou_0)$
$fun\_ass_1 = (fun\_ass\_x_0,\ fun\_ass\_y_0,\ fun\_ass\_dout_1,\ fun\_ass\_ou_0)$
$fun\_ass_2 = (fun\_ass\_x_1,\ fun\_ass\_y_1,\ fun\_ass\_dout_0,\ fun\_ass\_ou_0)$
$fun\_ass_3 = (fun\_ass\_x_0,\ fun\_ass\_y_2,\ fun\_ass\_dout_0,\ fun\_ass\_ou_0)$
$fun\_ass_4 = (fun\_ass\_x_2,\ fun\_ass\_y_0,\ fun\_ass\_dout_0,\ fun\_ass\_ou_0)$
$fun\_ass_5 = (fun\_ass\_x_0,\ fun\_ass\_y_0,\ fun\_ass\_dout_2,\ fun\_ass\_ou_1)$

Étant données les définitions précédentes, la machine M_{ab} obtenue après l'ordonnancement de la description initiale du Gcd est définit par le n-tuplet:

$$M_{ab} =< S, I, O, V, CONDITION, ASSIGN, fun\_status, f, h >$$

où

- $fun\_condition$ est le vecteur de cinq fonctions booléennes:
  $fun\_condition = (fun\_cond_1,\ fun\_cond_2,\ fun\_cond_3,\ fun\_cond_4,\ fun\_cond_5)$ où

  $fun\_cond_1(\overrightarrow{i}, \overrightarrow{v}) = (st =' 1')$ $\quad$ $fun\_cond_4(\overrightarrow{i}, \overrightarrow{v}) = (x < y)$
  $fun\_cond_2(\overrightarrow{i}, \overrightarrow{v}) = (din =' 1')$ $\quad$ $fun\_cond_5(\overrightarrow{i}, \overrightarrow{v}) = (din =' 0')$
  $fun\_cond_3(\overrightarrow{i}, \overrightarrow{v}) = (x \neq y)$

- f est la fonction de transition:

  $f(w_1,\ cond_1) = w_2;$ $\quad$ $f(s,\ \overline{cond_3}) = w_4;$
  $f(w_1,\ \overline{cond_1}) = w_1;$ $\quad$ $f(w_3,\ cond_3 \& cond_4) = w_3;$
  $f(w_2,\ cond_2) = s\ ;$ $\quad$ $f(w_3,\ cond_3 \& \overline{cond_4}) = w_3$
  $f(w_2,\ \overline{cond_2}) = w_2;$ $\quad$ $f(w_3,\ \overline{cond_3}) = w_4;$
  $f(s,\ cond_3 \& cond_4) = w_3;$ $\quad$ $f(w_4,\ cond_5) = w_2;$
  $f(s,\ cond_3 \& \overline{cond_4}) = w_3;$ $\quad$ $f(w_4,\ \overline{cond_5}) = w_4;$

- h est la fonction de sortie:

$h(w_1,\ cond_1) = fun\_ass_1;$
$h(w_1,\ \overline{cond_1}) = fun\_ass_0;$
$h(w_2,\ cond_2) = fun\_ass_2;$
$h(w_2,\ \overline{cond_2}) = fun\_ass_0;$
$h(s,\ cond_3\&cond_4) = fun\_ass_3;$
$h(s,\ cond_3\&\overline{cond_4}) = fun\_ass_4;$
$h(s,\ \overline{cond_3}) = fun\_ass_5;$
$h(w_3,\ cond_3\&cond_4) = fun\_ass_3;$
$h(w_3,\ cond_3\&\overline{cond_4}) = fun\_ass_4;$
$h(w_3,\ \overline{cond_3}) = fun\_ass_5;$
$h(w_4,\ cond_5) = fun\_ass_1;$
$h(w_4,\ \overline{cond_5}) = fun\_ass_0;$

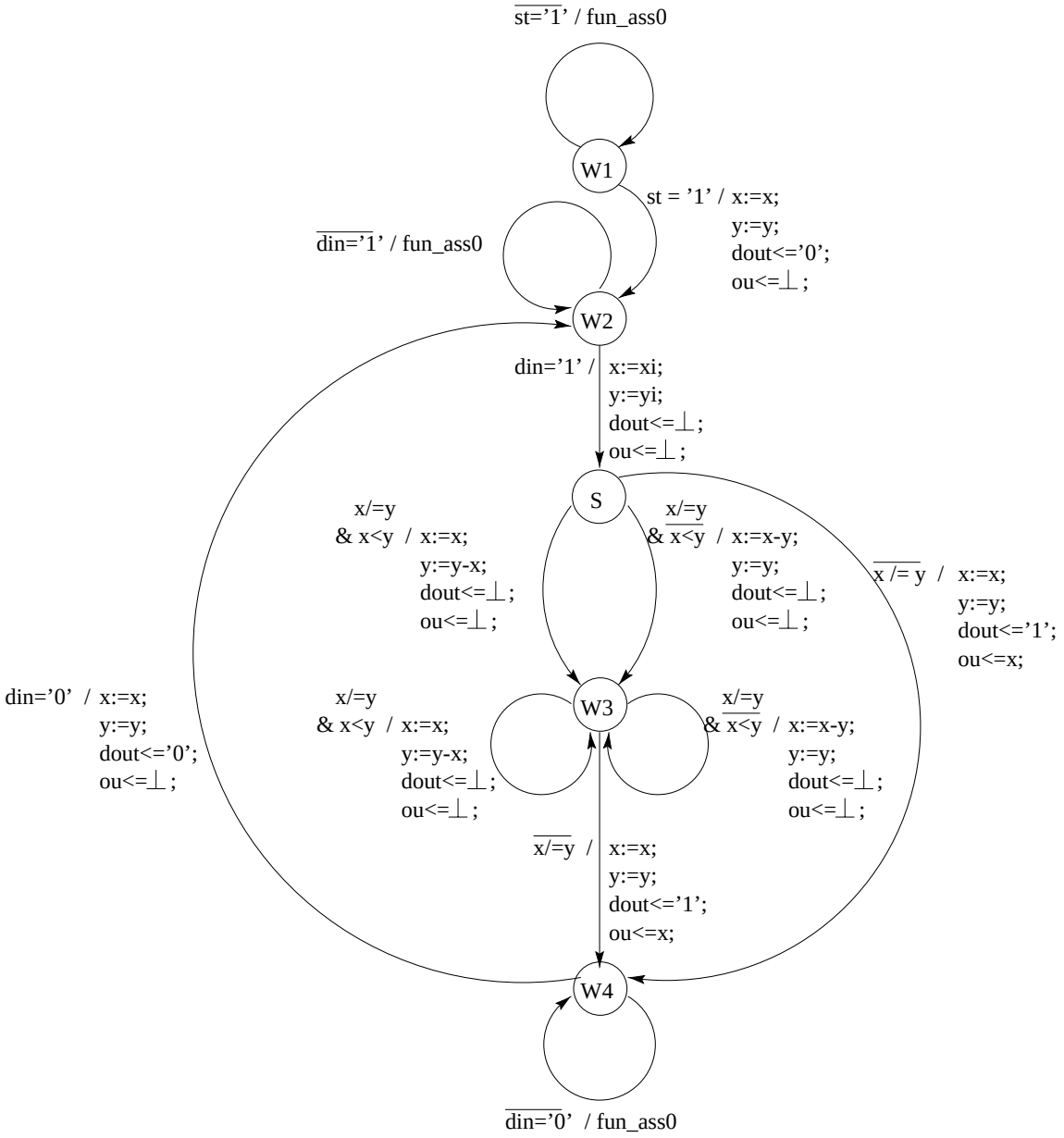


FIG. 4.10 – *Machine abstraite après l'ordonnancement de la description comportementale de Gcd*

Comme pour le modèle correspondant à la description initiale, chaque fonction vectorielle fun_assign a été complétée par les affectations initiales et la fonction fun_ass_0 représente le vecteur de ces affectations initiales.

Nous voyons que l'état S a été inséré lors de l'ordonnancement[10]. Par conséquent, trois

10. Pour faciliter la compréhension, dans la figure 4.10 et dans la définition formelle, les états correspondants aux états de la figure 4.7 portent les même noms (W_1-W_2). Dans l'annexe A (la sortie d'Amical), les états du Gcd

chemins d'exécution $path_{W_2 \to S \to W_3}$ (le chemin gauche), $path_{W_2 \to S \to W_3}$ (le chemin droit) et $path_{W_2 \to S \to W_4}$ qui contiennent cet état doivent être composés (figure 4.10). Formellement, l'algorithme de composition doit être appliqué également aux chemins d'exécution qui ne contiennent pas les états insérés pour dériver les fonctions de statuts fun_stat_j $(1 \leq j \leq p)$ à partir des fonctions de conditions fun_cond_k $(1 \leq k \leq q)$ de la machine abstraite avant la composition. Les fonctions d'affectation pour ces chemins d'exécution ne sont pas modifiées.

Un exemple de la composition du chemin gauche
$path_{w_2 \to s \to w_3} = (fun\_cond_2, fun\_ass_2, fun\_cond_3, fun\_cond_4, fun\_ass_3)$ est donné ci-dessous[11]. Nous supposons que avant le traitement du $path_{w_2 \to s \to w_3}$, fun_status contient déjà la fonction $fun\_stat_1(\overrightarrow{i}, \overrightarrow{v}) = (st =' 1')$. Pour désigner les fonctions d'affectations crées progressivement lors de la composition, nous allons utiliser les notations de la machine abstraite avant l'application de l'algorithme, en introduisant un nouveau nom si nécessaire (par exemple pour la fonction $fun\_ass\_y_3(\overrightarrow{i}, \overrightarrow{v}) = yi - xi$). La composition du $path_{w_2 \to s \to w_3}$ consiste alors en l'initialisation et les cinq itérations suivantes (pour chaque itération s nous donnons le résultat du calcul des fonctions fun_assign^s et fun_status^s):

Initialisation:
$fun\_assign^0 = (fun\_ass\_x_0, fun\_ass\_y_0, fun\_ass\_dout_0, fun\_ass\_ou_0)$
$fun\_status^0 = (fun\_stat_1)$ où $fun\_stat_1(\overrightarrow{i}, \overrightarrow{v}) = (st =' 1')$

1. $cur^1 = fun\_cond_2$:
   $fun\_assign^1 = (fun\_ass\_x_0, fun\_ass\_y_0, fun\_ass\_dout_0, fun\_ass\_ou_0)$
   $fun\_status^1 = (fun\_stat_1, fun\_stat_2)$ où $fun\_stat_2(\overrightarrow{i}, \overrightarrow{v}) = (din =' 1')$

2. $cur^2 = fun\_ass_2$:
   $fun\_assign^2 = (fun\_ass\_x_1, fun\_ass\_y_1, fun\_ass\_dout_0, fun\_ass\_ou_0)$
   $fun\_status^2 = (fun\_stat_1, fun\_stat_2)$

3. $cur^3 = fun\_cond_3$:
   $fun\_assign^3 = (fun\_ass\_x_1, fun\_ass\_y_1, fun\_ass\_dout_0, fun\_ass\_ou_0)$
   $fun\_status^3 = (fun\_stat_1, fun\_stat_2, fun\_stat_3)$ où $fun\_stat_3(\overrightarrow{i}, \overrightarrow{v}) = (xi \neq yi)$

4. $cur^4 = fun\_cond_4$:
   $fun\_assign^4 = (fun\_ass\_x_1, fun\_ass\_y_1, fun\_ass\_dout_0, fun\_ass\_ou_0)$
   $fun\_status^4 = (fun\_stat_1, fun\_stat_2, fun\_stat_3, fun\_stat_4)$
   où $fun\_stat_4(\overrightarrow{i}, \overrightarrow{v}) = (xi < yi)$

5. $cur^5 = fun\_ass_3$:
   $fun\_assign^5 = (fun\_ass\_x_1, fun\_ass\_y_3, fun\_ass\_dout_0, fun\_ass\_ou_0)$
   où $fun\_ass\_y_3(\overrightarrow{i}, \overrightarrow{v}) = yi - xi$
   $fun\_status^5 = (fun\_stat_1, fun\_stat_2, fun\_stat_3, fun\_stat_4)$

La fonction fun_assign^5 sera associée à la nouvelle transition directe entre les états W_2 et W_3. L'expression booléenne correspondant à cette transition se transforme en l'expression suivante: $(stat_2 \& (stat_3 \& stat_4))$.

---

après l'ordonnancement sont notés S_1-S_5, S_3 étant l'état inséré.

11. Dans notre exemple les fonctions $fun\_cond_2, \ldots, fun\_ass_3$ sont les instances des fonctions $cur^1, \ldots, cur^5$.

La composition du $path_{W_2 \to S \to W_3}$ (le chemin droit) et du $path_{W_2 \to S \to W_4}$ donne lieu aux fonctions d'affectations suivantes:

pour $path_{W_2 \to S \to W_3}$: $fun\_assign = (fun\_ass\_x_3,\ fun\_ass\_y_1,\ fun\_ass\_dout_0,\ fun\_ass\_ou_0)$ où $fun\_ass\_x_3(\overrightarrow{i}, \overrightarrow{v}) = xi - yi$ et

pour $path_{W_2 \to S \to W_4}$: $fun\_assign = (fun\_ass\_x_1,\ fun\_ass\_y_1,\ fun\_ass\_dout_2,\ fun\_ass\_ou_1)$.

Le vecteur des fonctions de statuts ne change pas lors de la composition de ces chemins d'exécution car les fonctions de statuts produites par l'algorithme (telle que $fun\_stat = (xi \neq yi)$ et $fun\_stat = (xi < yi)$) existent déjà comme éléments fun_stat_3 et fun_stat_4 de fun_status.

Après la composition des transitions insérées, la machine abstraite obtenue après l'ordonnancement prend la forme montré dans la figure 4.11. Nous ne donnons pas ici la définition formelle de la machine abstraite après la composition, puisque celle-ci se déduit de la figure 4.11 de la même façon que la définition formelle de la machine abstraite correspondant au circuit Gcd après l'ordonnancement se déduit de la figure 4.10.

Fin exemple

4.6 Comparaison de machines abstraites

Les compositions des chemins d'exécution transforment la machine abstraite obtenue après l'ordonnancement en une machine abstraite avec le même ensemble d'états que la machine abstraite correspondant à la description initiale comportementale. Les deux machines peuvent donc être comparées. Deux machines abstraites ayant les mêmes ensembles S, I, O, V et $STATUS$ sont équivalentes si elles ont les mêmes fonctions de statuts, de transition et de sortie. Nous supposons que les fonctions comparées ont le même nombre de composantes. La correspondance entre les états de deux machines est sensée être fournie par l'outil de synthèse.

Définition
Deux fonctions de statuts sont équivalentes s'il existe une permutation des indices telle que leurs composantes de même rang soient deux à deux égales:
$fun\_status_1 = (fun\_stat_1^1, \ldots, fun\_stat_q^1)$ et
$fun\_status_2 = (fun\_stat_1^2, \ldots, fun\_stat_q^2)$
sont équivalentes $fun\_status_1 \approx fun\_status_2$ si
$\forall\ fun\_stat_i^1\ (1 \leq i \leq q)\ \exists\ fun\_stat_j^2, (1 \leq j \leq q)$ telle que $fun\_stat_i^1(\overrightarrow{i}, \overrightarrow{v}) = fun\_stat_j^2(\overrightarrow{i}, \overrightarrow{v})$ pour tous les $\overrightarrow{i} \in I$ et tous les $\overrightarrow{v} \in V$ et
$\forall\ fun\_stat_j^2\ (1 \leq j \leq q)\ \exists\ fun\_stat_i^1, (1 \leq i \leq q)$ telle que $fun\_stat_i^1(\overrightarrow{i}, \overrightarrow{v}) = fun\_stat_j^2(\overrightarrow{i}, \overrightarrow{v})$ pour tous les $\overrightarrow{i} \in I$ et tous les $\overrightarrow{v} \in V$
Fin définition

L'égalité des fonctions d'affectations est vérifiée directement, car les composantes ($fun\_ass\_v_i$ et $fun\_ass\_o_j$) des fonctions comparées sont associées aux variables et aux sorties précises. Il est donc évident de trouver les composantes qui doivent être comparées, ce qui n'est pas le cas pour les fonctions de statuts. En effet, lors de la construction des machines abstraites les mêmes fonctions élémentaires de statuts peuvent avoir des noms différents selon leurs positions dans la

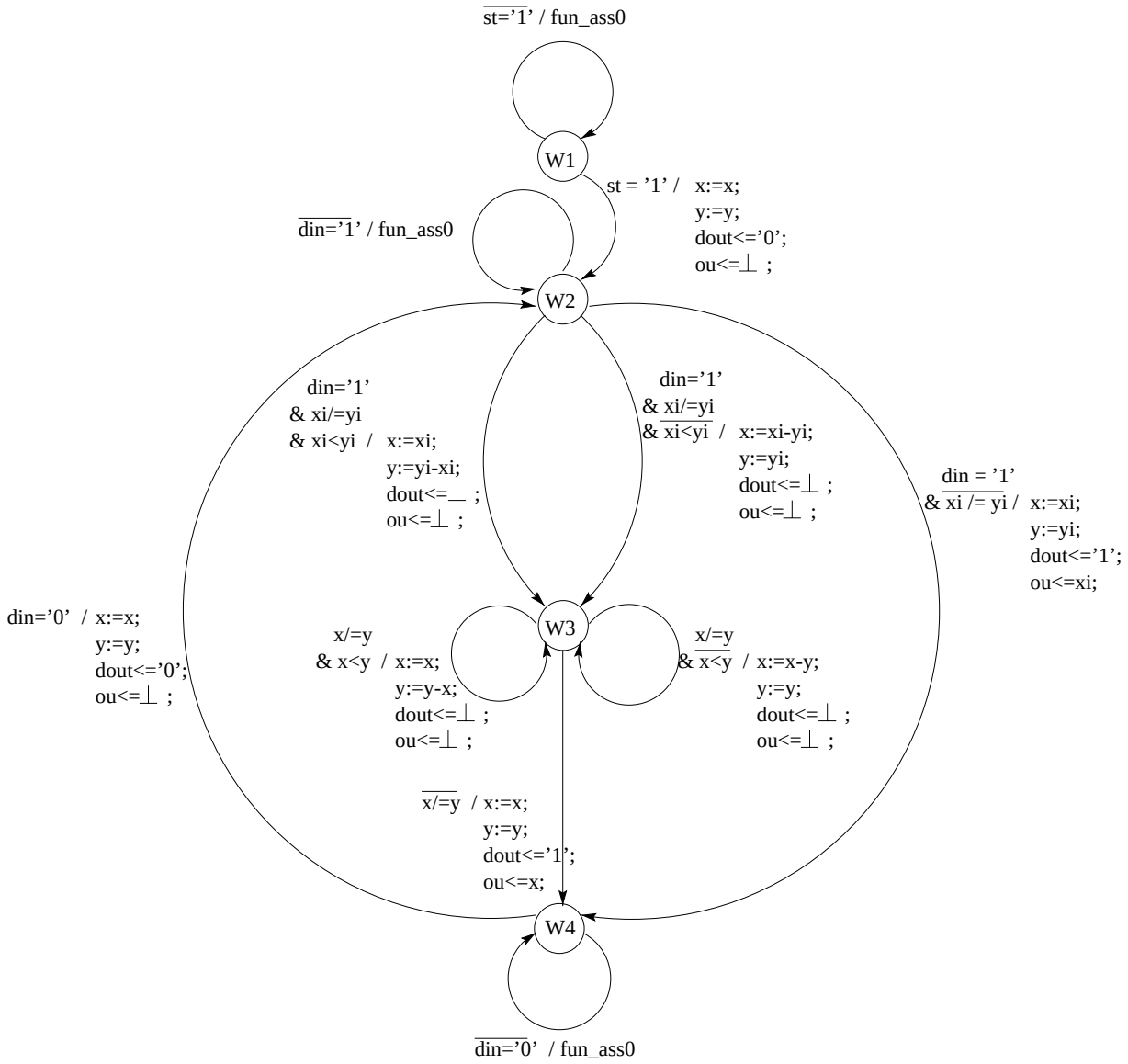


FIG. 4.11 – *Machine abstraite obtenue après l'ordonnancement de la description comportementale du Gcd où les états insérés lors de l'ordonnancement sont éliminés (après la composition des transitions insérées)*

fonction vectorielle fun_status. Il est donc nécessaire de permuter les fonctions de statuts, et de renommer les composantes de sorte que les fonctions élémentaires de statuts sémantiquement équivalentes portent le même nom et soient associées à la même variable de statut dans les deux machines comparées.

Or, chaque fonction de statut fun_stat_k est liée à la variable de statut $stat_k$, la permutation est donc faite aussi sur les noms des variables de statuts. Par conséquent, les fonctions de transition et de sortie doivent être ajustées elles-aussi: on remplace l'ancien nom de chaque variable par le nouveau dans les expressions booléennes participant aux définitions.

Après que les fonctions de statuts équivalentes ont été rendues égales, et les variables renommées, l'équivalence des machines abstraites peut être vérifiée.

Définition

Deux machines abstraites M_1 et M_2 sont équivalentes si, étant donné les ensembles S, I, O, V et $STATUS$ identiques pour les deux machines:
I. $fun\_status_1 = fun\_status_2$ et
II. pour tous les $s \in S$ et tous les $status \in STATUS$,
$f_1(s, status) = f_2(s, status)$ et
$h_1(s, status) = h_2(s, status)$ ce qui signifie que les fonctions h_1 et h_2 produisent les fonctions fun_assign_1 et fun_assign_2 telles que $fun\_assign_1 = fun\_assign_2$
Fin définition

En pratique, nous proposons d'effectuer la comparaison transition par transition à condition que la correspondance entre les états de deux machines soit connue. Dans ce cas, on vérifie que les deux machines ont des transitions équivalentes entre les états correspondants.

Définition
La transition $transition_1$ de la machine abstraite M_1 est équivalente à la $transition_2$ de la machine abstraite M_2 si $fun\_status_1 = fun\_status_2$ et les valeurs des variables de statuts et les fonctions d'affectations associées aux transitions comparées sont égales: $status_1 = status_2$ et $fun\_assign_1 = fun\_assign_2$.
Fin définition

Selon cette définition les deux machines abstraites correspondant au circuit Gcd avant (figure 4.11) et après (figure 4.7) l'ordonnancement sont équivalentes. La comparaison est effectuée transition par transition, en supposant la correspondance entre les états de deux machines portant le même nom. L'algorithme de la comparaison est omis en raison de son évidence. L'équivalence de ces deux machines prouve l'exactitude de l'étape d'ordonnancement pour le circuit Gcd.

4.7 Améliorations de la vérification de l'ordonnancement

Après la description de la méthode de base pour la vérification de l'étape d'ordonnancement, nous introduisions dans cette section deux améliorations de l'algorithme. La première concerne un ordonnancement plus sophistiqué. La deuxième considère le cas où le format intermédiaire (après l'ordonnancement) n'est pas exactement une machine abstraite.

4.7.1 Ordonnancement avancé

Hans Eveking dans [EHR99] et [Eve99] a présenté des transformations de l'ordonnancement [12] qui ne peuvent pas être directement vérifiées par la méthode proposée dans la section précédente. Dans [EHR99] ces transformationssont décrites en langage LLS (Language of Labelled Segments) et sont montrées dans la figure 4.12. Le contenu des fonctions de statuts fun_stat_1 et fun_stat_2 et des fonctions d'affectations fun_ass_1 et fun_ass_2 n'est pas pertinent.

Ces même transformations sont illustrées par la figure 4.13 sous la forme de machines abstraites. Les machines sont construites en prenant la description de la figure 4.12 comme descrip-

12. En fait, ces transformations sont plus fondamentales et sont utilisées dans plusieurs domaines informatiques, par exemple lors de la preuve des théorèmes ([BM97]).

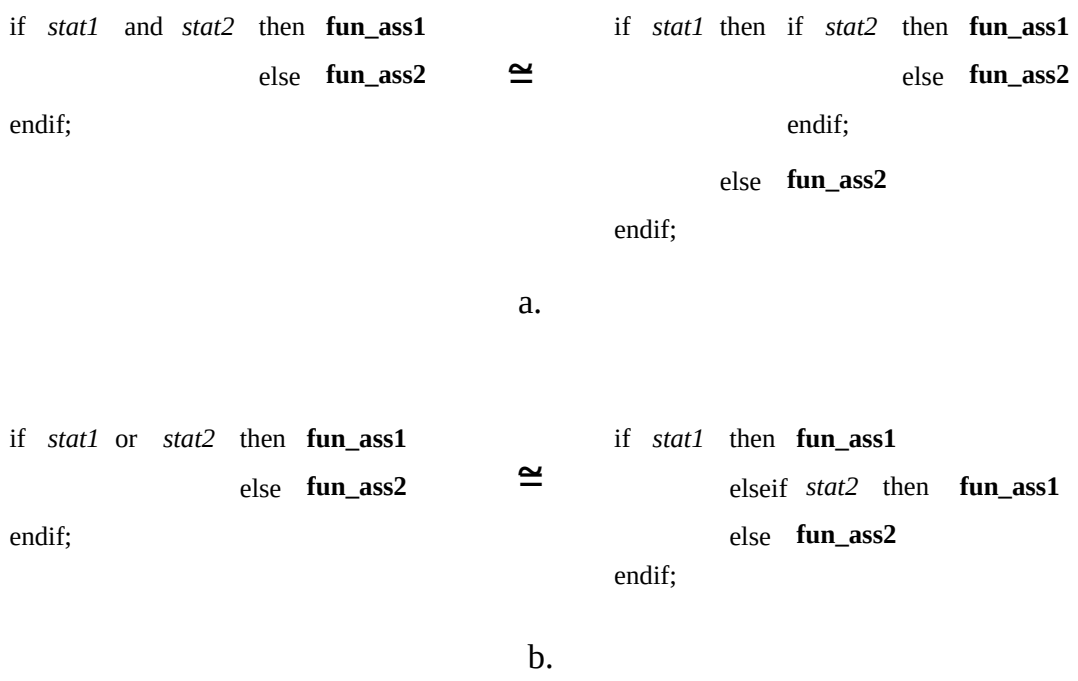


FIG. 4.12 – *Transformations avancées de la synthèse de haut niveau*

tion comportementale et en supposant que deux instructions "wait" sont placées avant et après les segments de code. Les "wait" implicites se transforment en états S_i (le "wait" initial) et S_j (le "wait" final) dans la figure 4.13. Les transitions entre S_i et S_j sont obtenues de la même façon que les transitions du modèle intermédiaire (section 4.3).

Nous voyons que le nombre de transitions entre les états S_i et S_j est différent dans les modèles avant et après transformation. Par conséquent, la comparaison "transition par transition" ne peut pas être appliquée. Cependant, on remarque que certaines transitions après transformation sont associées à la même fonction d'affectation fun_ass et peuvent être réunies: les transitions associées à la fonction fun_ass_2 de la première transformation (figure 4.13*a.)* et les transitions associées à la fonction fun_ass_1 de la deuxième transformation (figure 4.13.*b)*. L'expression booléenne associée à la nouvelle transition est le "ou" logique des expressions booléennes des transitions "fusionnées" (figure 4.14).

Nous insistons sur le fait que l'unification des transitions doit être faite dans les machines abstraites à comparer: dans la machine abstraite correspondant à la description initiale (issue du modèle intermédiaire) et dans la machine abstraite obtenue après l'étape d'ordonnancement (après la composition des transitions insérées). Après l'unification, la comparaison "transition par transition" peut être effectuée. Sur la figure 4.14 vérification d'équivalence est immédiate. Dans le cas général, l'équivalence des expressions booléennes attachées aux transitions peut être prouvée à l'aide, par exemple, de Diagrammes de Décision Binaire (BDD). Si les diagrammes correspondants sont identiques, alors les expressions booléennes sont égales.

En résumé, l'amélioration de l'algorithme proposée dans ce paragraphe consiste en une modification supplémentaire des machines abstraites avant leur comparaison. Cette modification est l'unification des transitions entre deux états associées à la même fonction d'affectation. L'expression booléenne de la nouvelle transition est le "ou" logique des expressions booléennes des transitions réunies. Nous rappelons, que lors de la composition de transitions consécutives, l'ex-

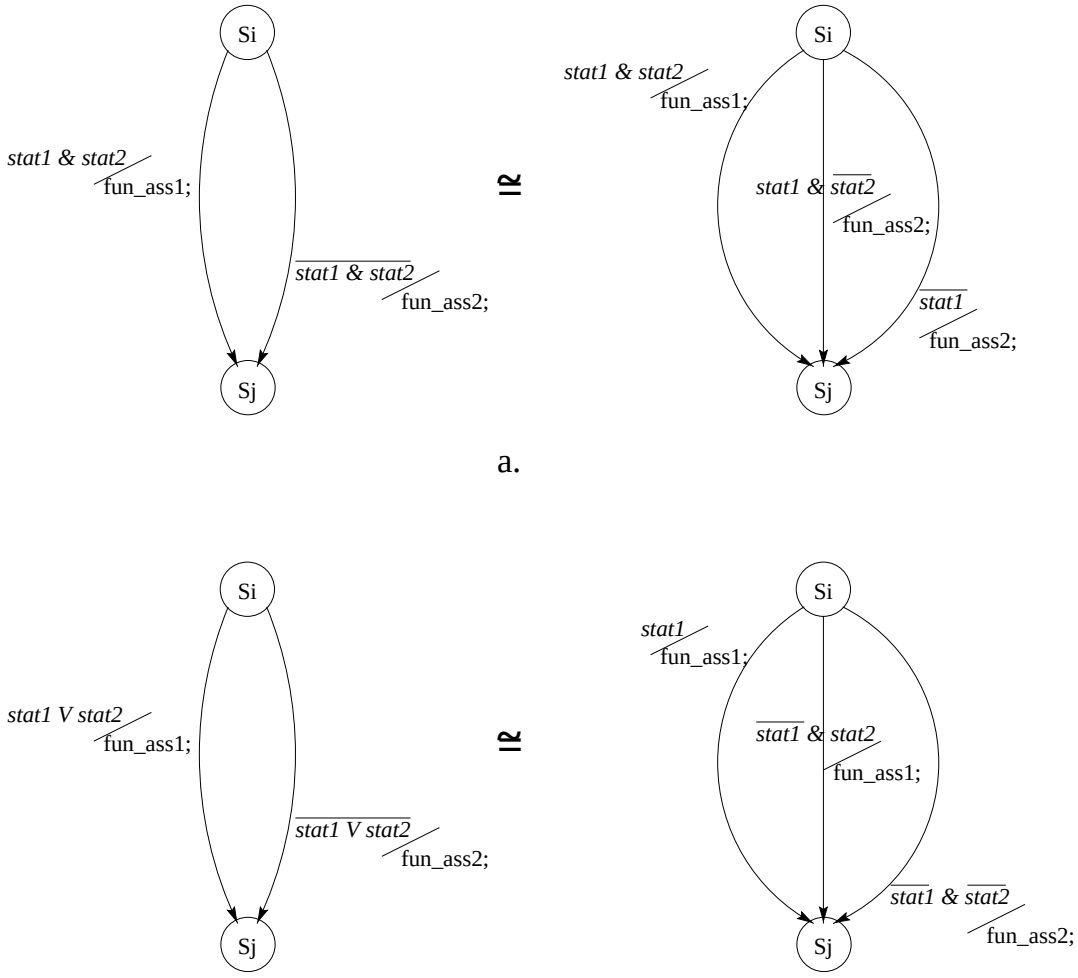


FIG. 4.13 – *Transformations avancées sous la forme des machines abstraites*

pression booléenne de la nouvelle transition est le "et" logique des expressions booléennes des transitions composées.

4.7.2 Forme intermédiaire différente de la machine abstraite

La nouvelle version d'Amical (appelée actuellement Music) rend après l'ordonnancement une description légèrement différente de la machine abstraite. La figure 4.15 montre le résultat de la nouvelle version après l'ordonnancement du circuit Gcd. La machine d'état finis est représentée par deux processus, dont l'un (le processus $p1$) est combinatoire et l'autre (le processus $SYNCH$) est séquentiel. La définition des processus combinatoires et séquentiels est définie dans [IEE98]. Principalement, un processus est réalisé par un circuit séquentiel s'il est sensible au signal d'horloge et par un circuit combinatoire sinon.

Le processus $SYNCH$ sert uniquement pour introduire les éléments séquentiels (registres) correspondant aux variables et aux signaux du processus initial de Gcd (figure 4.4). En revanche, le processus $p1$ représente le résultat de l'ordonnancement. Ce processus introduit les états du futur contrôleur et ressemble à une machine abstraite. Cependant, dans le processus $p1$ les affectations entre les états sont séquentielles et non pas parallèles comme dans le cas de la

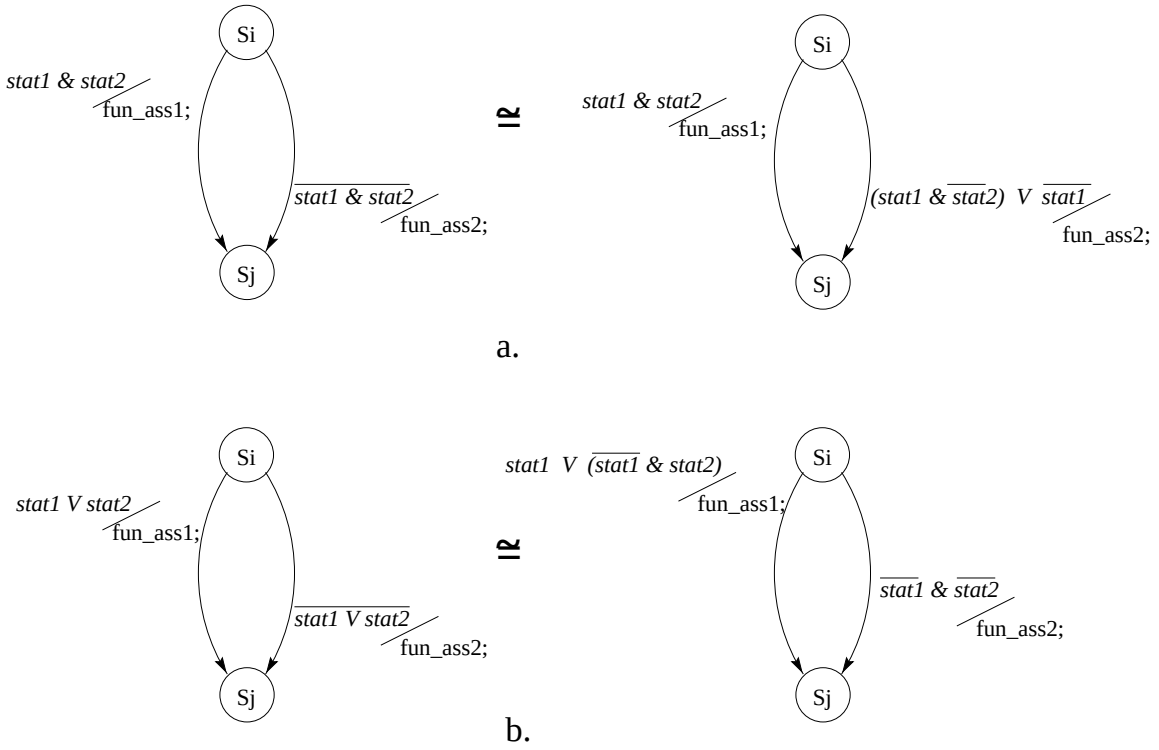


FIG. 4.14 – *Transformations avancées sous la forme des machines abstraites avec les transitions unies*

machine abstraite.

La deuxième différence significative entre le modèle de la machine abstraite et la description fournie par la nouvelle version d'Amical est l'introduction de variables supplémentaires pour le chaînage. Le chaînage sert à diminuer le nombre d'états de contrôle (associés avec les cycles d'horloge) nécessaires pour le calcul. Par exemple, la suite des affectations $\{c := a+b; c := c+y;\}$ peut être remplacée par l'affectation unique $\{c := a+b+y;\}$. Par conséquent, deux états de contrôle, normalement générés par l'ancienne version d'Amical et nécessaires pour ce calcul, seront remplacés par un seul.

Si la suite des affectations est $\{c := a+b; x := c; c := c+y;\}$, alors le résultat intermédiaire de la variable c $(a+b)$ sera sauvegardé dans une variable supplémentaire, par exemple $chained\_0$. Ensuite, la variable x prendra la valeur de la variable $chained\_0$ dans la description obtenue après l'ordonnancement: $\{chained\_0 := a+b; x := chained\_0; c := chained\_0+y;\}$. Les outils de synthèse logique, tels que "Design Compiler" de Synopsys, transformeront la dernière suite d'affectations en deux affectations parallèles: $\{x := a+b; c := a+b+y\}$ car la variable $chained\_0$ est mise à jour avant avoir d'été lue ([IEE98]).

La solution que nous proposons pour vérifier l'étape d'ordonnancement consiste à construire d'abord un modèle analogue au modèle intermédiaire, c'est-à-dire avec des *séquences* d'affectations entre les états explicitement définis. Ensuite, le modèle de la machine abstraite peut être déduit de la même façon que précédemment, par composition des affectations situées entre les états (section 4.4). Une fois la machine abstraite correspondant au circuit après l'ordonnancement construite, on la compare avec la machine abstraite correspondant à la description initiale. L'équivalence des deux machines implique, comme avant, l'exactitude de l'étape d'ordonnance-

```
entity gcd is
  port ( clk : in Bit;
         reset : in Bit;
         st : in Bit;
         din : in Bit;
         xi : in Integer;
         yi : in Integer;
         dout : out Bit;
         ou : out Integer ); end gcd;

architecture RTL of gcd is
  signal x, next_x : Integer;
  signal y, next_y : Integer;
  type S_TYPEp1 is (ST_p1_p1,ST_p1_1_p1,ST_p1_2_p1,ST_p1_3_p1);

signal C_S_p1, N_S_p1 : S_TYPEp1;

  -- net declarations
  -- Memd ports declaration
    signal current_dout, next_dout : Bit ;
    signal current_ou, next_ou : Integer ;

begin

    p1 : process (st, din, xi, yi, x, y,C_S_p1)

    variable C_H_A_I_N_E_D_0 : Integer;
    variable C_H_A_I_N_E_D_1 : Integer;
    variable C_H_A_I_N_E_D_2 : Integer;

    begin -- of process
      next_x <= x ;
      next_y <= y ;
      N_S_p1 <= ST_p1_p1;

      case C_S_p1 is

        when ST_p1_p1 =>
          if ( ( st = '1' ) and True ) then
              next_dout <= '0';
              N_S_p1 <= ST_p1_1_p1 ;
          else
            N_S_p1 <= ST_p1_p1 ;
          end if;

        when ST_p1_1_p1 =>
          if ( ( din = '1' ) and True ) then
            C_H_A_I_N_E_D_0 := xi;
            next_x <= C_H_A_I_N_E_D_2;
            C_H_A_I_N_E_D_1 := yi;
            next_y <= C_H_A_I_N_E_D_1;
            if ( C_H_A_I_N_E_D_0 /= C_H_A_I_N_E_D_1 ) then
              if ( C_H_A_I_N_E_D_0 < C_H_A_I_N_E_D_1 ) then
                C_H_A_I_N_E_D_1 := ( C_H_A_I_N_E_D_1 - C_H_A_I_N_E_D_0 );
                next_y <= C_H_A_I_N_E_D_1;
                N_S_p1 <= ST_p1_2_p1 ;
              else
                C_H_A_I_N_E_D_2 := ( C_H_A_I_N_E_D_0 - C_H_A_I_N_E_D_1 );
                C_H_A_I_N_E_D_0 := C_H_A_I_N_E_D_2;
                next_x <= C_H_A_I_N_E_D_2;
                N_S_p1 <= ST_p1_2_p1 ;
              end if;
            else
              next_ou <= C_H_A_I_N_E_D_0;
              next_dout <= '1';
              N_S_p1 <= ST_p1_3_p1 ;
            end if;
          else
            N_S_p1 <= ST_p1_1_p1 ;
          end if;

        when ST_p1_2_p1 =>
          if ( x /= y ) then
            if ( x < y ) then
              next_y <= ( y - x );
            else
              next_x <= ( x - y );
              N_S_p1 <= ST_p1_2_p1 ;
            end if;
          else
            next_ou <= x;
            next_dout <= '1';
            N_S_p1 <= ST_p1_3_p1 ;
          end if;

        when ST_p1_3_p1 =>
          if ( ( din = '0' ) and True ) then
              next_dout <= '0';
              N_S_p1 <= ST_p1_1_p1 ;
          else
            N_S_p1 <= ST_p1_3_p1 ;
          end if;
          when others => null;
        end case;
      end process p1;

      SYNCH : process ( clk, reset )
      begin  -- of SYNCH process
        if (reset = '0') then
            C_S_p1 <= ST_p1_p1;

      -- Resetting variables

      -- Resetting memd ports
        elsif (clk'event and clk = '1') then
            C_S_p1 <= N_S_p1;

      -- Updating variables
        x <= next_x ;
        y <= next_y ;

      -- Updating memd ports
        current_dout <= next_dout ;
        current_ou <= next_ou ;

        end if;
      end process SYNCH;

      -- net connections
      -- Memd ports assignment

      dout <= current_dout ;
      ou <= current_ou ;

      end RTL ; -- of architecture

      -- synopsys_synthesis_off
      configuration cfg_gcd of gcd is
            for RTL end for;
      end cfg_gcd;
      -- synopsys_synthesis_on
```

FIG. 4.15 – *Sortie du "scheduleur" de la nouvelle version d'Amical après l'ordonnancement du Gcd*

ment.

La figure 4.16 illustre le modèle avec les séquences d'affectation élémentaires[13] correspondant au processus $p1$ de la figure 4.15. Après la composition des affectations séquentielles, ce modèle se transforme en la machine abstraite montrée dans la figure 4.17. Les transitions de cette machine abstraite sont associées avec plus de fonctions d'affectations que la machine abstraite correspondant à la description initiale en raison de l'introduction des variables supplémentaires de chaînage *chained_0*, *chained_1* et *chained_2*. Ces variables, néanmoins ne seront pas considérées lors de la comparaison de la machine abstraite de la figure 4.17 avec la machine abstraite de la figure 4.7. Les variables supplémentaires représentent un raffinement de l'algorithme et ne font pas partie de la spécification initiale. De plus, elles ne seront implémentées que comme des fils de connexion. Seules les variables mémorisées et les sorties du modèle correspondant à la description initiale seront comparées (figure 4.7).

Lors de l'ordonnancement, les variables x et y de la description initiale se sont transformées en signaux x et y mis à jour à l'aide des signaux $next\_x$ et $next\_y$. Nous les considérons néanmoins comme identiques aux variables mémorisées x et y de la description initiale. Du fait que les signaux x et y sont mis à jour dans le processus $SYNCH$ synchronisé par le signal d'horloge clk, ils seront obligatoirement mis en oeuvre comme les registres selon le standard [IEE98] établi pour la synthèse logique. Or le standard pour la synthèse de haut niveau est pas encore apparu, Amical considère les variables affectées dans un processus synchronisé également comme des éléments mémorisés.

La comparaison[14] montre que les transitions $1\_p1 \rightarrow 2\_p1$ (gauche, avec la condition $((din =' 1')\&(xi \neq yi)\&(xi < yi))$ et $1\_p1 \rightarrow 3\_p1$ de la machine abstraite obtenue après l'ordonnancement (figure 4.17) ne sont pas équivalentes à leurs analogues de la machine abstraite avant l'ordonnancement (figure 4.7). L'affectation de la variable x dans les transitions mentionnées après l'ordonnancement devient $x = chained\_2$ à la place de l'affectation $x = xi$ de la machine abstraite avant l'ordonnancement. Une bogue du programme d'ordonnancement a été ainsi révélée. Ce fait prouve l'efficacité de la méthode proposée même pour des exemples simples.

La bogue n'avait pas été trouvée auparavant parce que la valeur de l'entrée xi était choisie intuitivement plus grande que la valeur de l'entrée yi par les personnes qui vérifiaient le circuit. Par conséquent, lors de l'exécution, la transition $1\_p1 \rightarrow 2\_p1$ (droite, avec la condition $((din =' 1')\&(xi \neq yi)\&(\overline{xi < yi}))$ de la machine abstraite était prise, et ensuite le circuit marchait correctement pendant la simulation. L'erreur s'est manifestée en simulation lorsque la valeur de xi a été choisie inférieure à celle de yi.

4.8 Résumé

Dans ce chapitre nous avons introduit une méthode pour la vérification de l'étape d'ordonnancement. La méthode proposée consiste à ramener le modèle correspondant à la description initiale et le modèle correspondant au circuit après l'ordonnancement à deux machines abstraites

13. Les fonctions d'identité pour les variables mémorisées et le $\perp$ pour les sorties qui doivent normalement compléter chaque affectation élémentaire dans le modèle intermédiaire, ne sont pas montrées.

14. Nous supposons que la correspondance entre les états W_1, W_2, W_3, et W_4 de la figure 4.7 et $p1$, 1_p1, 2_p1, et 3_p1 de la figure 4.17 est connue.

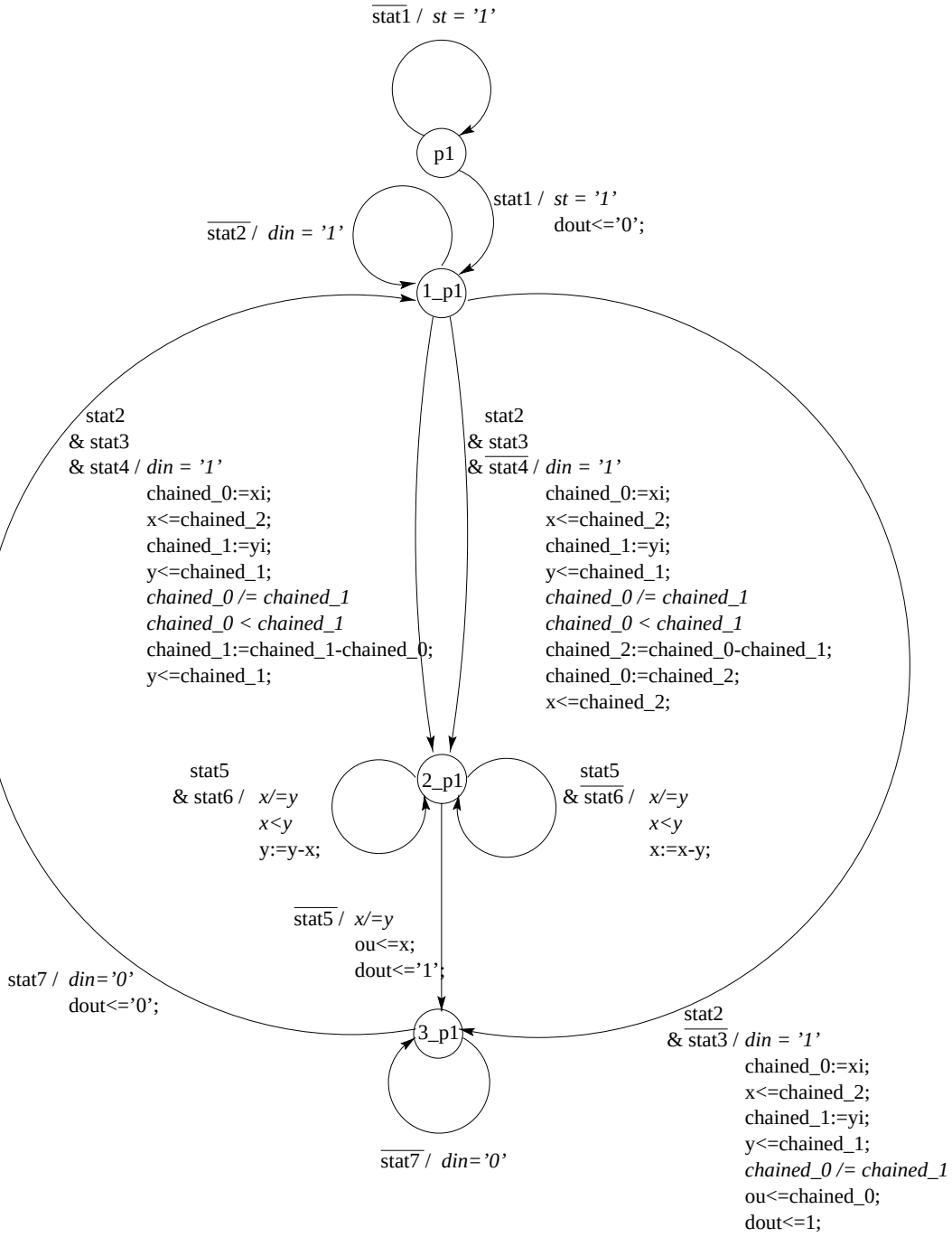


FIG. 4.16 – *Modèle correspondant à la description du Gcd après l'ordonnancement avec la nouvelle version d'Amical*

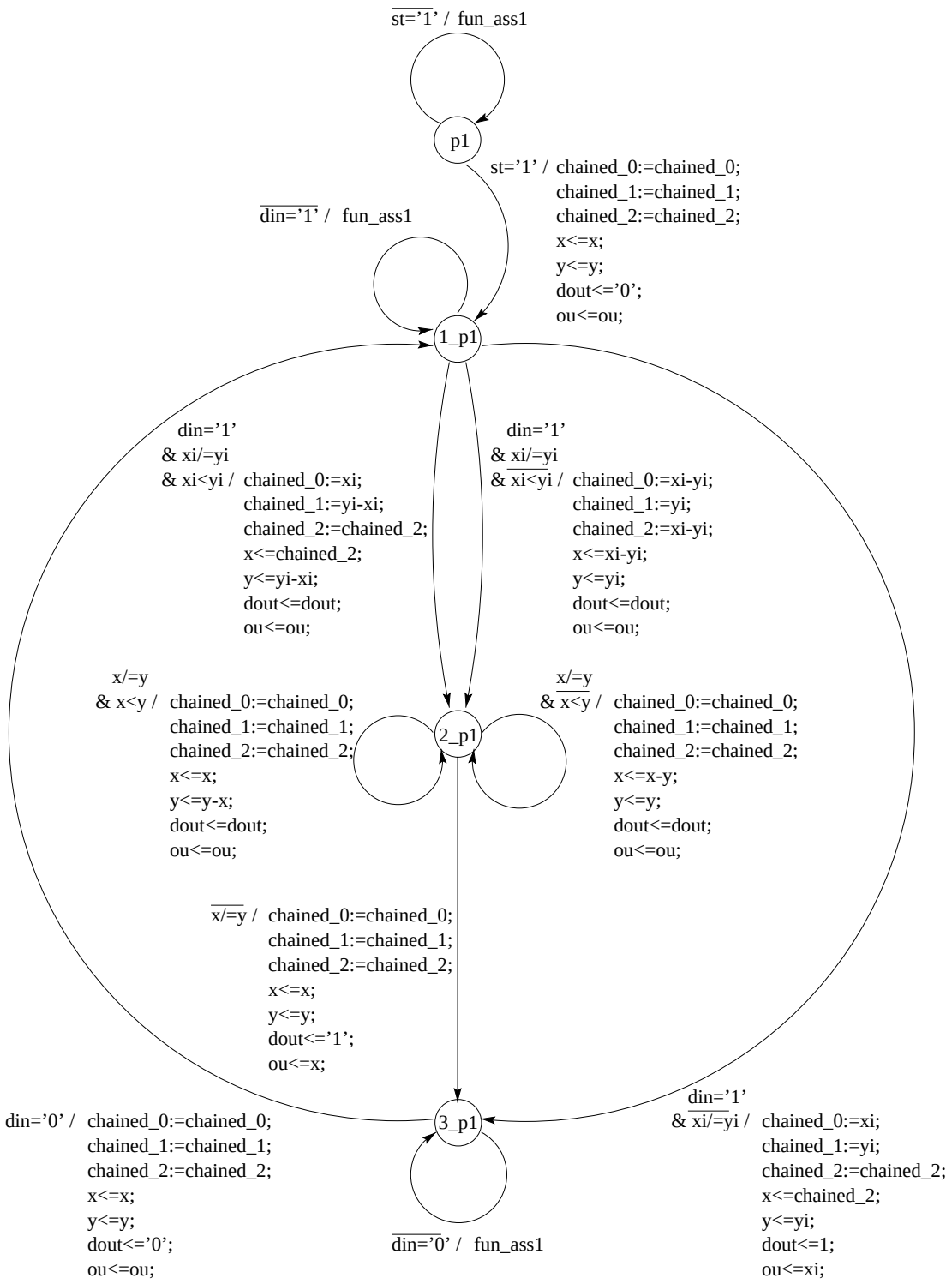


FIG. 4.17 – *Machine abstraite correspondante à la description du Gcd après l'ordonnancement avec la nouvelle version d'Amical*

ayant le même nombre d'états. L'équivalence des deux machines, qui est définie comme l'équivalence des transitions correspondantes, implique l'exactitude de l'étape d'ordonnancement.

Un algorithme, utilisé à la fois pour les transformations du modèle initial et pour les transformations de la machine abstraite obtenue après l'ordonnancement, a été développé. Deux améliorations de cet algorithme visent à vérifier des ordonnancements plus complexes.

La méthode a été illustrée par l'exemple du circuit qui calcule le PGCD de deux entiers. L'exemple a révélé un bug du programme d'ordonnancement, montrant ainsi l'efficacité de la méthodologie proposée.

Chapitre 5

Vérification de l'étape d'allocation

Ce chapitre est dédié à la vérification de la deuxième étape de la synthèse de haut niveau qui est l'allocation des ressources et la génération de l'architecture finale (figure 2.1).

L'allocation associe à chaque opération et/ou procédure de la machine abstraite une unité fonctionnelle pouvant exécuter cette opération. La génération de l'architecture finale aboutit à une architecture qui consiste en un contrôleur et un chemin de données (ou partie opérative[1]). Le chemin de données contient des registres, des unités fonctionnelles et le réseau de communication. Les signaux de contrôle émis par le contrôleur définissent le transfert des données dans la partie opérative pour l'exécution d'une opération ou d'une procédure spécifiée dans la machine abstraite.

L'objectif de la vérification de cette étape est de vérifier que:

1. une "bonne" unité fonctionnelle a été choisie pour l'exécution d'une opération spécifiée dans la machine abstraite,
2. la fonctionnalité exacte de cette unité a été choisie pour l'opération spécifiée dans la machine abstraite (dans le cas où une unité fonctionnelle peut exécuter plusieurs opérations),
3. le réseau de communication a été correctement généré (le réseau de communication contient des multiplexeurs, des bus et des "switch"),
4. les signaux de contrôle venant du contrôleur ont été correctement affectés pour réaliser le bon transfert des bonnes données pour l'exécution d'une opération donnée.

Dans la figure 2.1, par exemple, pour l'opération "a:=x+y", il faut vérifier que

1. le contenu des registres "x" et "y" est transféré en entrées de l'UAL (le réseau de communication et les signaux de sélection des multiplexeurs correspondants doivent être correctement générés),
2. le signal de contrôle de l'UAL est affecté pour que l'UAL réalise une addition,
3. le résultat de l'UAL est transféré en entrée du registre "a" et, à la fin,
4. le signal d'écriture pour ce registre est activé.

1. Ultérieurement, nous allons utiliser les deux termes (*chemin de données* et *partie opérative*) pour désigner cette partie de l'architecture finale.

5.1 Principe de la méthode proposée

Comme pour l'étape d'ordonnancement, nous proposons de vérifier que l'architecture finale décrite au niveau de transfert de registres satisfait la machine abstraite *transition par transition*. La méthodologie de vérification consiste à faire une *exécution symbolique* pour *chaque transition* du chemin de données et à comparer ensuite les résultats de l'exécution avec les affectations spécifiées dans la machine abstraite. La figure 5.1 décrit ce processus. Lors de l'exécution, de nouvelles valeurs symboliques sont attribuées aux registres et aux entrées du circuit pour chaque transition. Ainsi, le problème de la propagation des résultats à travers des points de "bifurcation" est évité.

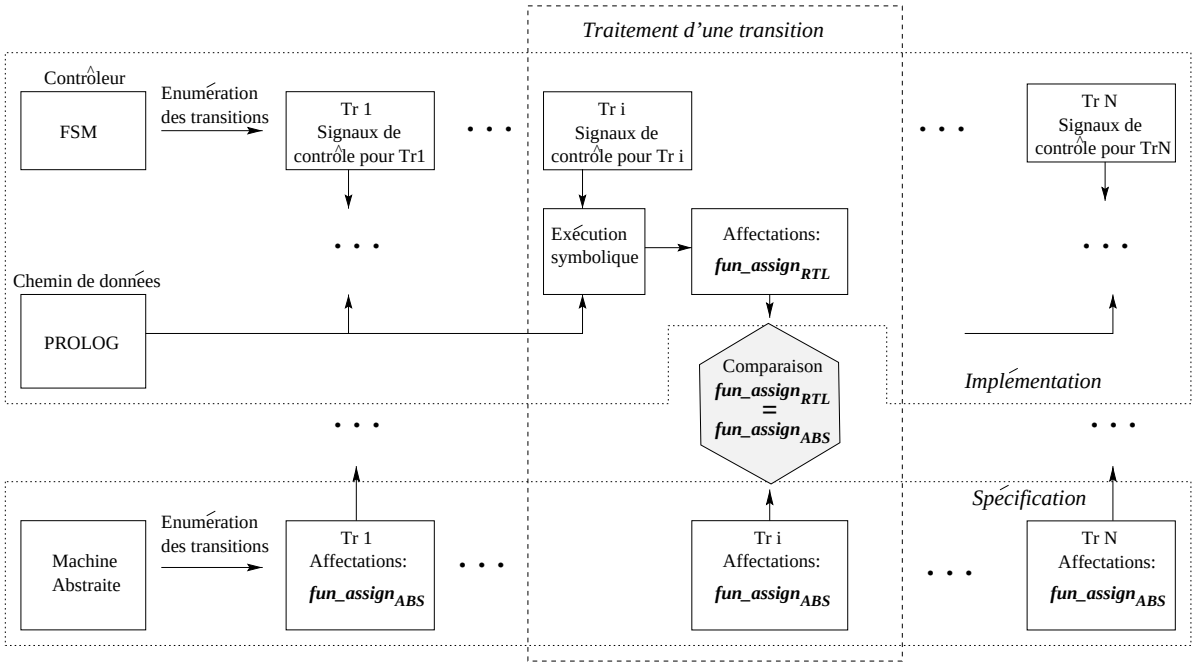


FIG. 5.1 – *Principe de la vérification de l'étape d'allocation*

L'exécution symbolique permet d'obtenir les affectations des registres et des sorties à partir des affectations associées aux composants. Plus précisément, au début chaque composant du chemin de données est représenté comme l'ensemble des fonctions d'affectations qui spécifient le comportement de ses sorties en fonction de ses entrées. A titre d'exemple, la fonction $f\_out\ (in1, in2) = in1 + in2$ sera associée à la sortie *out* de l'additionneur si $in1$ et $in2$ représentent ses entrées.

Ensuite, le chemin de données est décrit comme une interconnexion de composants. L'exécution symbolique permet de composer les fonctions des modules de base selon leur connexion et selon les signaux de contrôle associés avec une transition donnée. Le résultat de la composition est l'ensemble d'affectations des registres et des sorties fun_assign_{RTL} qui sera comparé avec un ensemble des affectations fun_assign_{ABS} de la machine abstraite. Par exemple, si les entrées de l'additionneur mentionné plus haut sont liées avec les registres x et y, sa sortie liée avec l'entrée du registre a et le signal d'écriture de ce registre activé pour une transition donnée, alors l'exécution symbolique donne lieu à la fonction $fun\_a\ (all\_inputs, all\_registers) = cur\_x + cur\_y$ associée avec le registre a. Cur_x et cur_y sont les valeurs *symboliques* associées, pour une

transition donnée, avec les registres x et y.

Nous admettons les hypothèses suivantes:

1. Nous utilisons des composants simples, déjà vérifiés et qui se comportent conformément à leurs spécifications. Cette pré-vérification peut être faite par d'autres outils et en appliquant d'autres méthodologies, comme, par exemple, la démonstration de théorèmes ou les diagrammes de moment binaire - *BMD ([Pie91], [Pie95], [BHY92], [DB97], [BC95], [Bry95], [CKZ96]).

2. Pendant la synthèse, un registre est alloué à chaque variable de la description comportementale initiale. Une affectation d'une variable dans la machine abstraite est considérée alors comme une affectation d'un registre correspondant dans l'architecture finale pendant la comparaison des ensembles fun_assign_{RTL} et fun_assign_{ABS}. La connaissance du flot de conception d'Amical nous permet de faire cette supposition.

3. Chaque transition de la machine abstraite correspond à une transition du contrôleur final (un cycle d'horloge réel) et tous les éléments de base ne prennent qu'un seul cycle d'horloge pour leur exécution. Cette supposition nous permet de réaliser la comparaison la machine abstraite avec l'architecture finale, transition par transition.

4. Nous supposons que la machine abstraite raffinée pendant la deuxième étape de la synthèse de haut niveau est synthétisable; c'est-à-dire que les fonctions d'affectations attachées à chaque transition sont de signature $I \times V \rightarrow V \times O$ à la place $I \times V \rightarrow V^{assign} \times O^{assign}$ dans le cas général où $V \subseteq V^{assign}$ et $O \subseteq O^{assign}$.

La dernière hypothèse est très importante car elle permet de simplifier les fonctions d'affectations associées aux composants de la partie opérative de l'architecture finale. En effet, selon le théorème du paragraphe 3.3 cette restriction signifie que, quelle que soit la fonction attachée à une transition, les valeurs des variables et des signaux d'entrée au moment de l'exécution de cette transition sont telles que le résultat de la fonction est restreint à l'ensemble $V \times O$ et non plus à l'ensemble $V^{assign} \times O^{assign}$. Par conséquent, dans les définitions des fonctions d'affectation associées au chemin de données, nous n'avons pas besoin de considérer les valeurs tronquées due à la largeur des bus et des registres.

Ainsi, lors de la définition des composants du chemin de données, nous nous limitons à l'aspect fonctionnel et nous ne prenons pas en compte la largeur des bus et des registres. Comme nous l'avons indiqué, la sortie d'additionneur sera associée à la fonction $f\_out\ (in1, in2) = in1 + in2$ et non à la fonction $f\_out\ (in1, in2) = (in1 + in2)\ mod\ 2^{out\_width}$. La définition formelle de la partie opérative entière est présentée dans la prochaine section.

Grâce à la quatrième hypothèse, dans le modèle formel de la machine abstraite les fonctions d'affectations seront notées comme étant de signature $I \times V \rightarrow V \times O$ (à la place de la signature $I \times V \rightarrow V^{assign} \times O^{assign}$) car elles portent cette signature à chaque transition.

5.2 Modèle adopté pour l'architecture finale

Le modèle de l'architecture finale ([DBJ98], [BDP98]) est une composition des modèles du contrôleur CP et du chemin de données DP liés par les signaux booléens $COMMAND$ (générés par le contrôleur) et $STATUS$ (générés par le chemin de données). La composition aboutit exactement à la même forme que le modèle de la machine abstraite. La comparaison de la composition avec la machine abstraite avant l'allocation est donc possible. L'équivalence des deux machines abstraites (avant et après l'allocation) implique l'exactitude de cette étape de la synthèse de haut niveau.

Nous adoptons les notations suivantes pour le modèle du contrôleur CP:

- S est l'ensemble des états de contrôle,
- $STATUS$ est l'ensemble des symboles des entrées du contrôleur, formé par les valeurs des signaux des statuts:
  $status = (stat_1, stat_2, ..., stat_q) \in STAT_1 \times STAT_2 \times ... \times STAT_q = STATUS$
  où $STAT_j$ est l'ensemble de toutes les valeurs possibles du statut $stat_j$. Dans un circuit réel $STAT_j$ est l'ensemble booléen et $status$ est un vecteur de bits
- $COMMAND$ est l'ensemble des symboles des sorties du contrôleur, formé par les valeurs des signaux des commandes:
  $command = (com_1, com_2, ..., com_u) \in COM_1 \times COM_2 \times ... \times COM_u = COMMAND$
  où COM_j est l'ensemble de toutes les valeurs possibles du signal de commande com_j. Comme dans le cas des signaux des statuts, dans un circuit réel COM_j est l'ensemble booléen et $command$ est un vecteur de bits

Étant données les définitions précédentes, le contrôleur CP est défini par un 5-tuplet:

$$CP = < S, STATUS, COMMAND, f, h_{cp} >$$

où

- $f : S \times STATUS \rightarrow S$ est une fonction de transition et
- $h_{cp} : S \times STATUS \rightarrow COMMAND$ est une fonction de sortie.

Ainsi, le modèle du contrôleur est une machine d'états finis classique qui prend en entrée les signaux de statuts fournis par le chemin de données et produit les signaux de sortie qui vont vers le chemin de données. La fonction de transition f est exactement la même que la fonction de transition pour le modèle de la machine abstraite. La fonction de sortie h_{cp} est propre au contrôleur: elle génère les commandes pour exécuter les affectations dans la partie opérative.

Le modèle pour le chemin de données est défini comme une sorte de machine d'états finis classique dont les entrées et les sorties sont divisées en deux sous-ensembles. De plus, le modèle

ne contient pas d'ensemble des états de contrôle. En revanche, il possède l'ensemble des registres V et l'ensemble des affectations $ASSIGN_{RTL}$ similaires à celles de la machine abstraite.

Nous adoptons les notations suivantes pour le modèle du chemin de données DP:

- I est l'ensemble des symboles des entrées, formé par les valeurs des signaux des entrées primaires fournis par l'environnement:
  $i = (i_1, i_2, ..., i_l) \in IVAL_1 \times IVAL_2 \times ... \times IVAL_l = I$
  où $IVAL_j$ est l'ensemble de toutes les valeurs possibles de l'entrée i_j.

- $COMMAND$ est l'ensemble des symboles des entrées, formé par les valeurs des signaux des entrées fournis par le contrôleur:
  $command = (com_1, com_2, ..., com_u) \in COM_1 \times COM_2 \times ... \times COM_u = COMMAND$
  où COM_j est l'ensemble de toutes les valeurs possibles de l'entrée com_j.

- O est l'ensemble des symboles des sorties, formé par les valeurs des signaux des sorties primaires qui vont vers l'environnement:
  $o = (o_1, o_2, ..., o_m) \in OVAL_1 \times OVAL_2 \times ... \times OVAL_m = O$
  où $OVAL_j$ est l'ensemble de toutes les valeurs possibles de la sortie o_j.

- $STATUS$ est l'ensemble des symboles des sorties, formé par les valeurs des signaux des sorties qui vont vers le contrôleur:
  $status = (stat_1, stat_2, ..., stat_q) \in STAT_1 \times STAT_2 \times ... \times STAT_q = STATUS$
  où $STAT_j$ est l'ensemble de toutes les valeurs possibles de la sortie $stat_j$.

- V est l'ensemble des symboles des registres du chemin de données:
  $v = (v_1, v_2, ..., v_n) \in VVAL_1 \times VVAL_2 \times ... \times VVAL_n = V$
  où $VVAL_j$ est l'ensemble de toutes les valeurs correspondantes au registre v_j.

- $ASSIGN_{RTL}$ est l'ensemble des affectations des registres et des signaux des sorties primaires. Il est défini comme l'ensemble de toutes les fonctions de $I \times V$ vers $V \times O$. Un élément de cet ensemble est une fonction fun_assign_{RTL} de signature $I \times V \rightarrow V \times O$:
  $ASSIGN_{RTL} = \{fun\_assign_{RTL} \in I \times V \rightarrow V \times O\}$.
  En fait, chaque fonction fun_assign_{RTL} est un vecteur[2] de fonctions, une pour chaque registre et une pour chaque sortie:
  $fun\_assign_{RTL} = (fun\_ass\_v_1, ..., fun\_ass\_v_n, fun\_ass\_o_1, ..., fun\_ass\_o_m)$ où
  $fun\_ass\_v_i \in ASS\_V_i = \{I \times V \rightarrow VVAL_i\}; \quad (1 \leq i \leq n)$ et
  $fun\_ass\_o_j \in ASS\_O_j = \{I \times V \rightarrow OVAL_j\}; \quad (1 \leq j \leq m)$.

Étant données les définitions précédentes, le chemin de données est définie par un n-tuplet:

$$DP = < I, COMMAND, O, STATUS, V, ASSIGN_{RTL}, h_{dp}, fun\_status >$$

2. Comme dans la définition de la machine abstraite (paragraphe 3.2), nous identifions les deux types:
$(A \times B \rightarrow C_1) \times (A \times B \rightarrow C_2) \times ... \times (A \times B \rightarrow C_n)$ et
$A \times B \rightarrow C_1 \times C_2 \times ... \times C_n$.

où

- h_{dp} est la première fonction de sortie qui définit les affectations à exécuter à partir des valeurs des signaux des commandes (les affectations, elles, définissent à leur tour les valeurs des sorties primaires et des registres):
  $h_{dp} : COMMAND \rightarrow ASSIGN_{RTL}$

- fun_status est la deuxième fonction de sortie qui définit les valeurs des sorties *status* à partir des valeurs des entrées primaires et des registres; cette fonction est exactement la même que dans le modèle de la machine abstraite: c'est un vecteur [3]de fonctions, une pour chaque statut:
  $fun\_status = (fun\_stat_1, fun\_stat_2, ..., fun\_stat_q) : I \times V \rightarrow STATUS$ où
  $fun\_stat_k : I \times V \rightarrow STAT_k \ \ (1 \leq k \leq q)$.

Or le modèle du chemin de données ne contient pas les états de contrôle, la fonction de transition n'existe pas dans ce modèle.

La composition des modèles pour le contrôleur et pour le chemin de données est montrée dans la figure 5.2.b et consiste en une composition des fonctions h_{dp} et h_{cp}. Mathématiquement la composition des fonctions h_{dp} et h_{cp} est possible car l'ensemble d'arrivée ($COMMAND$) de la fonctions h_{cp} est en même temps l'ensemble de départ de la fonction h_{dp}. Après la composition des fonctions h_{dp} et h_{cp} l'ensemble $COMMAND$ n'est plus pertinent. Ainsi, la composition M_{RTL} des modèles du contrôleur et du chemin de données se résume par la définition suivante:

$$M_{RTL} = DP \circ CP =< S, I, O, V, STATUS, ASSIGN_{RTL}, fun\_status, f, h_{RTL} >$$

où

- fun_status est la fonction des statuts:
  $fun\_status = (fun\_stat_1, fun\_stat_2, ..., fun\_stat_q) : I \times V \rightarrow STATUS$ où
  $fun\_stat_k : I \times V \rightarrow STAT_k \ \ (1 \leq k \leq q)$.

- f est la fonction de transition:
  $f : STATUS \times S \rightarrow S$ et

- h_{RTL} est la fonction de sortie:
  $h_{RTL} = h_{dp} \circ h_{cp} : STATUS \times S \rightarrow ASSIGN_{RTL}$

Pour simplifier les modèles, nous avons choisi que le "circuit combinatoire" fun_status appartienne au chemin de données, ce qui n'est pas obligatoire. La place exacte du "circuit" fun_status est définie lors de la synthèse logique et est souvent le contrôleur. Ce fait, néanmoins n'affecte en aucune façon notre formalisation car la composition des modèles du contrôleur et du chemin de données reste la même.

La dernière remarque concerne la stabilité du modèle. Il est bien connu que la composition de deux machines classiques n'est pas toujours stable en raison de l'introduction des boucles combinatoires ([Boo67], [HS66]). La composition des modèles du contrôleur et du chemin de données est exempte de ces boucles grâce à la présence des éléments mémorisants V.

3. La remarque sur le type de la fonction fun_assign s'applique aussi sur le type de la fonction fun_status.

5.3 Comparaison de la machine abstraite avec l'architecture finale

Après la composition du modèle du contrôleur avec le modèle du chemin de données, le modèle de l'architecture finale (figure 5.2.b) se ramène à la forme de la machine abstraite (figure 5.2.a).

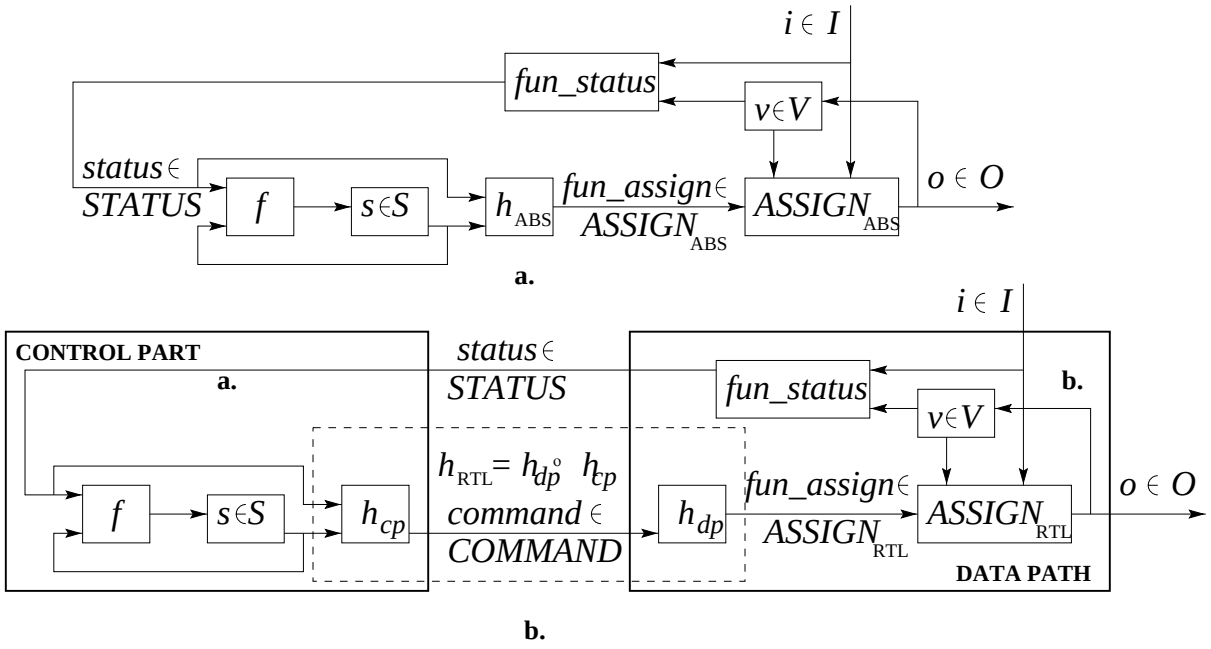


FIG. 5.2 – *Représentation schématique de la machine abstraite (a.) et de l'architecture finale (b.)*

A condition que les ensembles I, V, S soient les mêmes[4] pour les deux modèles et que la fonction fun_status ne change pas lors de l'allocation, la preuve d'équivalence de deux modèles revient à la preuve d'équivalence des fonctions h_{ABS} et h_{RTL}.

Les fonctions h_{ABS} et h_{RTL}, cependant, ne peuvent pas être directement comparées car la fonction $h_{dp} : COMMAND \rightarrow ASSIGN$ (et par conséquent la fonction h_{RTL}) est implicite. En effet, le chemin de données issu de l'étape d'allocation est une interconnexion des éléments standards de bibliothèque. *Les affectations sont donc "cachées" dans la structure de la partie opérative.* Pour avoir la fonction h_{dp} définie explicitement, nous faisons abstraction de la structure vers une description comportementale pour chaque symbole pertinent de *command*.

L'abstraction est une notion fondamentale dans la vérification formelle des circuits et des programmes. Elle consiste à supprimer des détails ou des informations dans le but de plus facilement manipuler, et donc vérifier, des systèmes. Il existe différents mécanismes d'abstraction qui ont été surtout décrits dans [Mel87].

Notre approche est similaire à celle employée dans [Ard96b] et [Ard96a] pour la vérification des microprocesseurs. Chaque élément de la partie opérative est associé avec l'ensemble des fonc-

4. L'ensemble I est le même par sa nature. L'ensemble S reste le même car, d'après la troisième hypothèse du paragraphe 5.1, Amical ne fait que raffiner la machine abstraite obtenue après l'ordonnancement, sans rajouter d'états supplémentaires. Par conséquent, la fonction f reste aussi sans modification. L'ensemble V est le même selon la deuxième hypothèse du paragraphe 5.1.

tions qui caractérisent son comportement. Ensuite, le comportement (les fonctions d'affectations) du chemin de données entier est dérivé par la composition des fonctions associées aux éléments. On réalise ainsi une abstraction structurelle car la structure interne du chemin de données est "dissoute" dans les fonctions d'affectations. Or la fonction $h_{dp} : COMMAND \rightarrow ASSIGN_{RTL}$ n'est intéressante que composée avec la fonction h_{cp}, l'abstraction est faite seulement pour les symboles de *command* qui sont les résultats de la fonction h_{cp}. Autrement dit, l'abstraction est faite seulement pour les symboles de commande suivants: $\{command \mid command = h_{cp}(s, status);\ s \in S\ \&\ status \in STATUS\}$.

La composition des fonctions des éléments de la partie opérative est effectuée par exécution symbolique du chemin de données. Les détails de la mise en oeuvre sont donnés dans la prochaine section. Notons simplement que le langage de programmation PROLOG a été choisi pour parvenir à ce but . Ce choix est justifié par le fait que les valeurs symboliques sont naturellement manipulées dans ce langage. Le principe d'unification de PROLOG nous permet, par exemple, à partir de la description d'un additionneur d'obtenir la valeur de sa sortie, qui est $in1 + in2$, si ses entrées sont les valeurs symboliques $in1$ et $in2$. Ainsi, la valeur symbolique de la sortie représente la fonction d'affectation associée avec l'additionneur, de même que les valeurs symboliques des entrées représentent les arguments de cette fonction. La valeur symbolique de la sortie sera fournie à l'entrée d'un composant lié à l'additionneur. En d'autres termes, la valeur sortante sera fournie comme argument d'une fonction d'affectation associée à un élément suivant, réalisant ainsi la composition des affectations des éléments de base.

La méthode proposée ressemble à celle de Barrow ([Bar84]) sauf que dans [Bar84] un circuit est décrit en PROLOG dans sa totalité et les équations sont dérivées pour une machine d'états finis correspondante.

Des langages de programmation plus répandus tels que C ou C++ sont mal adaptés à nos besoins en raison d'un système d'exécution d'où l'unification est absente: nous devrions nous-mêmes mettre en oeuvre ce principe pour pouvoir obtenir, par exemple, $in1+in2$ comme résultat d'un additionneur et faire ensuite la composition de ce résultat avec les fonctions d'affectations d'autres éléments.

Le système PROLOG que nous utilisons est le système SISCtus2.1, version 0.7 \#9.

5.4 Mise en oeuvre en PROLOG du modèle de l'architecture finale

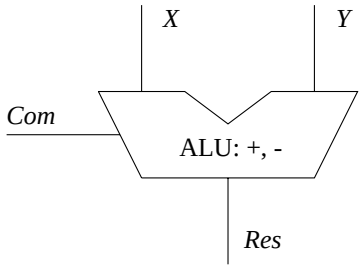
Dans cette section nous montrons comment, à partir de la description structurelle de l'architecture finale, obtenir le vecteur des fonctions d'affectations $fun\_assign_{RTL} \in ASSIGN_{RTL}$ qui est comparé au vecteur analogue des fonctions d'affectation $fun\_assign_{ABS} \in ASSIGN_{ABS}$ de la machine abstraite. D'abord, nous présentons la définition des éléments de base. Ensuite nous passons à la description entière d'un chemin de données. Un exemple simple facilitera la compréhension de la méthodologie proposée.

5.4.1 Description des éléments de base

Nous allons décrire en PROLOG le chemin de donnée sous une forme analogue à celle d'une architecture structurelle en VHDL. Le chemin de données est présenté comme l'interconnexion de composants qui sont des exemplaires d'éléments de bibliothèque. Les éléments de bibliothèque sont divisés en deux groupes:

1. éléments combinatoires (l'unité arithmétique et logique, l'unité de décalage, etc.)
2. éléments séquentiels qui mémorisent les valeurs de leurs sorties calculées au moment du front d'horloge. Ce sont les registres ou bascules ("flip-flop").

Éléments combinatoires Considérons un exemple d'unité arithmétique et logique (UAL), illustré dans la figure 5.3.




```
%%% alu (X, Y, Res, Com) :-
alu (X, Y, Res, 0) :-
    Res = X + Y.
alu (X, Y, Res, 1) :-
    Res = X - Y.
```

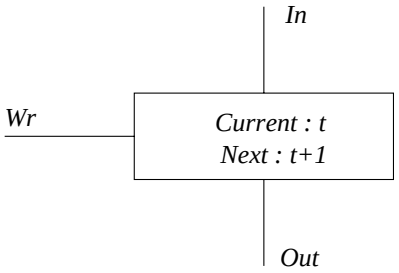
FIG. 5.3 – *Exemple d'UAL et de sa description en PROLOG*

L'UAL additionne les entrées X et Y si le signal de contrôle Com est égal à 0 et les soustrait si Com est égal à 1. La description de cette unité en PROLOG est montrée également dans la figure 5.3. La première ligne est en commentaire (les commentaires commencent par le symbole "%"): elle désigne les noms des paramètres et leur ordre pour les appels à cette unité fonctionnelle. Par exemple, dans la deuxième ligne de la description: $alu\,(X, Y, Res, 0)$, 0 est fourni pour le signal Com. Cette deuxième ligne désigne le comportement de l'UAL si le signal Com est égal à 0. Si ce n'est pas le cas, le système PROLOG cherchera la définition suivante de l'UAL, définie pour Com égal à 1. La sortie de l'UAL sera $X - Y$.

De manière générale, le comportement d'une unité fonctionnelle sera décrit pour chaque combinaison possible des valeurs de ses signaux de contrôle. Pendant l'appel à cette unité, le système PROLOG cherchera une définition convenable, à l'issue de laquelle il affectera les sorties de l'unité fonctionnelle. A titre d'exemple, si l'appel à l'UAL est $alu\,(arg3, arg4, Out, 1)$, alors, la variable Out sera égale à l'expression $arg3 - arg4$.

Registres Un exemple de registre et sa description en PROLOG sont donnés dans la figure 5.4.

Or en PROLOG il est impossible d'utiliser la même variable à la fois comme argument et comme résultat d'une fonction, deux variables représentent le contenu du registre. La variable $Current$ apparaît dans la partie droite des affectations, elle est fournie pour le contenu du registre au cycle d'horloge courant. La variable $Next$ est utilisée dans la partie gauche des affectations (les résultats des fonctions), elle représente le contenu du registre au cycle d'horloge suivant.




```
%%% reg (In, Out, Wr, Current, Next) :-
    reg (In, Out, 1, Current, Next) :-
        Out = Current,
        Next = In.
    reg (In, Out, 0, Current, Next) :-
        Out = Current,
        Next = Current.
```

FIG. 5.4 – *Exemple d'un registre et de sa description en PROLOG*

La variable $Next$ change sa valeur seulement si le signal Wr (le signal d'écriture) est égal à 1. La sortie Out est toujours égale au contenu du registre pendant le cycle d'horloge courant (la variable $Current$).

5.4.2 Chemin de données comme interconnexion d'éléments de base

Après la description des éléments de base, nous pouvons décrire le chemin de données entièrement. Illustrons cette démarche sur l'exemple d'une partie opérative très simple (figure 5.5). $Mux1$, $Mux2$, Alu et Wr sont les signaux de contrôle fournis par le contrôleur. Le but de l'exécution symbolique est l'extraction des fonctions d'affectations pour la sortie Out et le contenu du registre Reg au cycle d'horloge suivant à partir des entrées primaires A, B, C et le contenu du registre au cycle d'horloge courant.

D'abord on nomme toutes les connexions internes: ce sont les connexions $Mux1\_Alu$, $Mux2\_Alu$, Alu_Reg et Out[5]. Ensuite le chemin de données est décrit de la même façon qu'en VHDL: après la description des éléments de bibliothèque utilisés, on remplace chaque composant du chemin de données par une instance de bibliothèque.

La différence est que les instances $inst\_mux1$, $inst\_mux2$, $inst\_alu$ et $inst\_reg$ sont uniquement les références sur les éléments de la bibliothèque. Au moment de leur définition, elles ne sont pas encore connectées comme en VHDL par le moyen des signaux. La liaison entre les éléments se fait dans le prédicat du chemin de données appelé *datapath* à l'aide des variables d'interconnexions ($Mux1\_Alu$, $Mux2\_Alu$, etc). La variable $Mux1\_Alu$, par exemple, correspond au signal VHDL qui relie la sortie du multiplexeur $Mux1$ à l'entrée gauche de l'additionneur.

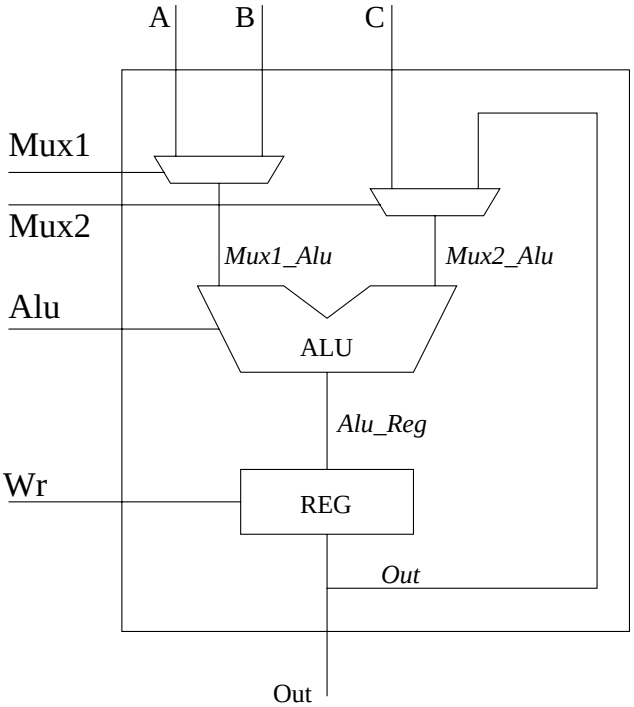
A partir de la description du chemin de données, l'ensemble des affectations pour chaque transition du contrôleur peut être dérivé. Par exemple si, pour une transition donnée, les signaux de contrôle $Mux1$, $Mux2$, Alu, Alu, Wr sont égaux à 1, 1, 0 et 1 respectivement, alors les fonctions pour la sortie Out et pour le contenu du registre au cycle d'horloge suivant $Next\_reg$[6] seront:

$$Out = Cur\_reg \text{ et}$$
$$Next\_reg = B + Cur\_reg.$$

5. Nous utilisons le même nom *Out* pour designer la sortie du circuit et l'interconnexion registre - multiplexeur $Mux2$.

6. Nous signalons que les noms du contenu du registre aux cycles d'horloge courant et suivant sont Cur_reg et $Next\_reg$, i.e. ce sont les noms fournis comme arguments du prédicat *datapath*. Les noms $Current$ et $Next$ désignent les paramètres de la description de l'élément de base reg.

```
%%%%%%%%%%%%
%%% Library elements
%%%%%%%%%%%%

%%% mux (In1, In2, Out, Sel) :-
      mux (In1, _, Out, 0) :-
            Out = In1.
      mux (_, In2, Out, 1) :-
            Out = In2.

%%% alu (X, Y, Res, Com) :-
      alu (X, Y, Res, 0) :-
            Res = X + Y.
      alu (X, Y, Res, 1) :-
            Res = X - Y.

%%% reg (In, Out, Wr, Current, Next) :-
      reg (In, Out, 1, Current, Next) :-
            Out = Current,
            Next = In.
      reg (_, Out, 0, Current, _) :-
            Out = Current.

%%%%%%%%%%%
%%% Instantiation
%%%%%%%%%%%

inst_mux1 (A, B, C, D) :-
      mux (A, B, C, D).

inst_mux2 (A, B, C, D) :-
      mux (A, B, C, D).

inst_alu (A, B, C, D) :-
      alu (A, B, C, D).

inst_reg (A, B, C, D, E) :-
      reg (A, B, C, D, E).
```




```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Datapath description (instances binding)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%

datapath (A, B, C, Out, Mux1, Mux2, Alu, Wr, Cur_reg, Next_reg) :-
      inst_mux1 (A, B, Mux1_Alu, Mux1),
      inst_mux2 (C, Out, Mux2_Alu, Mux2),
      inst_alu (Mux1_Alu, Mux2_Alu, Alu_Reg, Alu),
      inst_reg (Alu_Reg, Out, Wr, Cur_reg, Next_reg).
```

FIG. 5.5 – *Exemple d'un chemin de données et sa description en PROLOG*

5.4.3 Exécution symbolique du chemin de données et comparaison avec la machine abstraite

L'exécution symbolique du chemin de données consiste à appeler le prédicat du chemin de données *datapath* pour chaque transition du contrôleur de l'architecture finale. Les variables des signaux de contrôle sont remplacées par les valeurs concrètes affectées par chaque transition. Les variables des entrées et des registres sont remplacées par des valeurs symboliques. Ainsi, l'exécution symbolique pour la transition $S_i \rightarrow S_j$ de la figure 5.6 est réalisée par l'appel suivant du prédicat "datapath":

$$datapath\ (a, b, c, Out, 1, 1, 0, 1, reg, Next\_reg)$$

Notons que les variables d'entrées A, B, C sont remplacées par les valeurs symboliques a, b, c et la variable du contenu du registre au cycle d'horloge courant Cur_reg par la valeur symbolique reg.

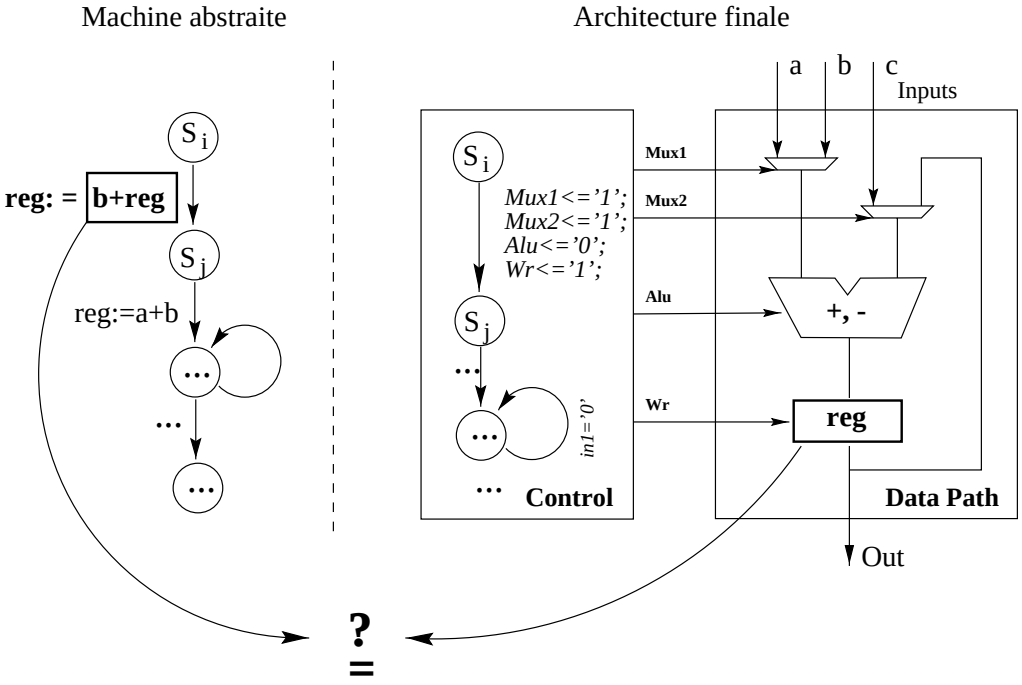


FIG. 5.6 – *Comparaison des résultats de l'exécution avec la machine abstraite*

Le système PROLOG calcule la sortie Out et le contenu du registre au cycle d'horloge suivant[7]:

$$Out = reg \text{ et}$$
$$Next\_reg = b + reg$$

Les fonctions d'affectations ainsi obtenues sont comparées à celles attachées à la même transition de la machine abstraite (figure 5.6). L'outil de vérification sait que la variable $Next\_reg$ contient l'expression du registre reg au prochain cycle d'horloge.

Dans notre exemple, les fonctions associées au registre reg dans la machine abstraite et obtenues après l'exécution symbolique sont équivalentes. Mais si, par exemple à cause d'une faute de la synthèse, le signal de contrôle $Mux1$ était affecté à la valeur '0', alors le prédicat

7. C'est le principe d'unification qui est employé pour calculer ces valeurs.

datapath serait appelé avec les valeurs d'arguments suivantes:

$$datapath\ (a, b, c, Out, 0, 1, 0, 1, reg, Next\_reg)$$

Ceci conduirait pour le registre reg à une fonction $a + reg$. Pendant la comparaison des deux fonctions la faute serait révélée.

La dernière remarque de cette section concerne la comparaison elle-même des fonctions d'affectation. Actuellement, la comparaison de chaque paire de fonctions d'affectations est basée sur la comparaison syntaxique des expressions symboliques associées aux fonctions d'affectations. De plus, la propriété de commutativitée des fonctions pertinentes (+", "*", etc.) est prise en compte. Ainsi, l'outil de la vérification prouve l'équivalence des deux expressions $(a+b)$ et $(b+a)$. Puisque les outils actuels de la synthèse de haut niveau ne font pas de transformations avancées, cette technique de la comparaison suffit, dans la plupart des cas, pour établir une équivalence de deux expressions sémantiquement égales.

Elle est cependant incapable de prouver l'équivalence des expressions $(a+(b+c))$ et $((a+b)+c)$. Une solution éventuelle consiste à ramener les deux expressions à une forme canonique par des règles de réécriture. Une approche similaire est employée dans [ZB95]. Néanmoins la preuve de la confluence (l'existence d'une forme canonique et la terminaison d'un processus de réécriture) entre dans le domaine très compliqué des preuves par réécriture ([HKLR92], [BKK+96]). De plus, même en utilisant les algorithmes les plus complexes, cette méthode ne peut pas prouver a priori l'équivalence des expressions $(a + a)$ et $(2 * a)$ à moins d'introduire de nouvelles règles. Pour pallier cette difficulté un démonstrateur de théorèmes général ([BM97]) doit être appliqué pour les futures transformations sophistiquées de la synthèse de haut niveau.

5.5 Prototype d'outil développé

Pour bien situer l'outil développé, considérons d'abord le flot[8] d'Amical (figure 5.7). La première transformation d'Amical est une transformation de VHDL vers le format interne Solar. Toutes les transformations ultérieures seront réalisées sur ce format. La première étape est l'ordonnancement, pendant laquelle la description comportementale initiale se transforme en une machine d'états finis abstraite. La deuxième étape est l'allocation des ressources et la génération d'architecture finale. La phase finale est une traduction de l'architecture finale en VHDL au niveau transfert de registres, acceptable par les outils de synthèse logique tels que par exemple, "Design Compiler" de Synopsys.

L'outil de vérification accepte en entrée les descriptions en Solar de la machine abstraite et de l'architecture finale composée d'un contrôleur et d'un chemin de données (figure 5.8). Actuellement, l'outil contient trois parties qui doivent être lancées l'une après l'autre. Cette décomposition est due à la version de PROLOG utilisée et en principe peut être éliminée en employant une version plus récente. Le langage de développement est C++, car il représente le langage de développement commun pour tout le système Amical. L'utilisation de PROLOG, à part la description des blocs de base montrée dans le paragraphe 5.4.1, est transparente à l'utilisateur.

La première partie (I) traduit le chemin de données de Solar en PROLOG (dans le prédicat

8. L'ancienne version d'Amical est considérée.

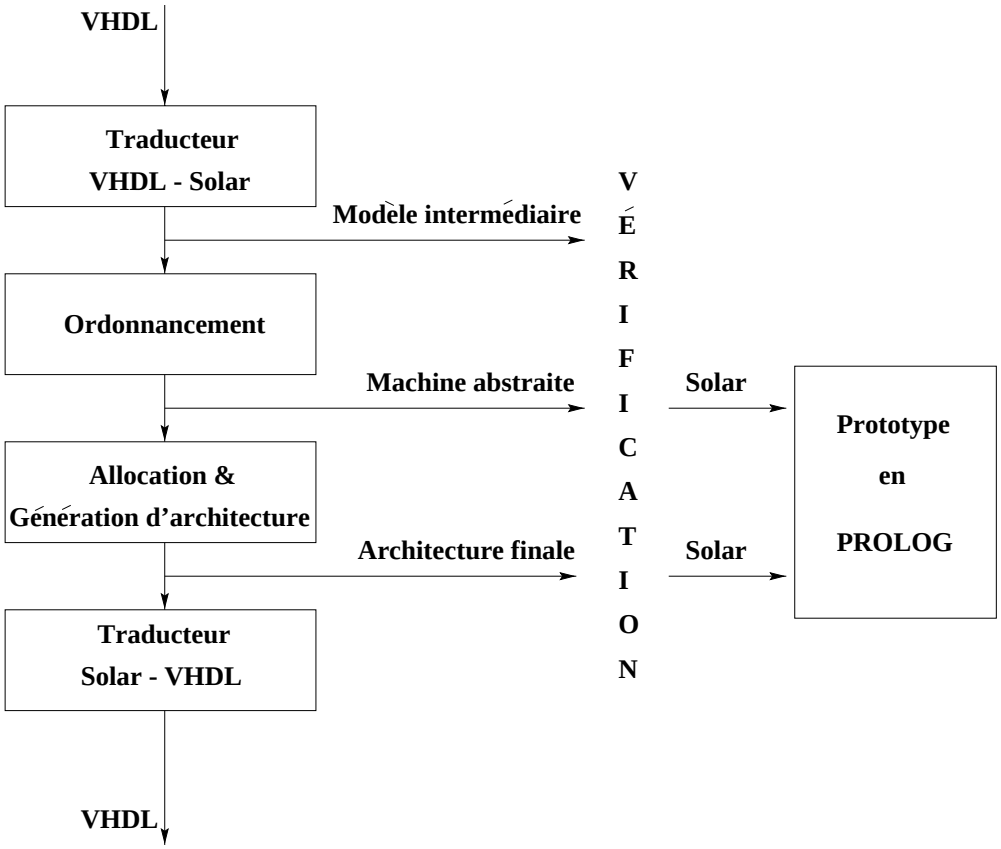


FIG. 5.7 – *Flot d'Amical*

"datapath"). La deuxième partie (II) traduit chaque transition du contrôleur en un appel de ce prédicat en fournissant des valeurs précises aux signaux de contrôle. La troisième partie (III) fait une exécution symbolique du chemin de données pour chaque transition et les compare ensuite avec les affectations de la machine abstraite.

Les résultats de la vérification sont enregistrés dans un fichier spécial. Pour chaque transition, les résultats de la comparaison des registres et des sorties sont reportés séparément. Si l'implémentation ne satisfait pas la spécification, alors les deux expressions sont imprimées et une faute d'implémentation est signalée. Si dans l'architecture finale un registre ou une sortie est affecté tandis qu'il ne l'est pas dans la machine abstraite, cette situation est indiquée par un message particulier. La situation inverse (la spécification spécifie une affectation et l'implémentation ne la fait pas) est aussi signalée.

5.6 Exemples d'application

Cette section décrit les exemples avec lesquels nous avons testé l'outil développé. Tout d'abord, pour finir l'exemple du Gcd, nous présentons les résultats de la vérification de l'étape d'allocation pour ce circuit. L'annexe B montre les descriptions VHDL du Gcd après une session de la synthèse avec Amical[9]. Le premier fichier est le contrôleur contenant cinq états et le deuxième fichier est le chemin de données. Le chemin de données traduit en Prolog lors de la

9. Pour obtenir ces descriptions, l'ancienne version d'Amical a été utilisée. Ces résultats correspondent à la description ordonnée de la figure 4.10.

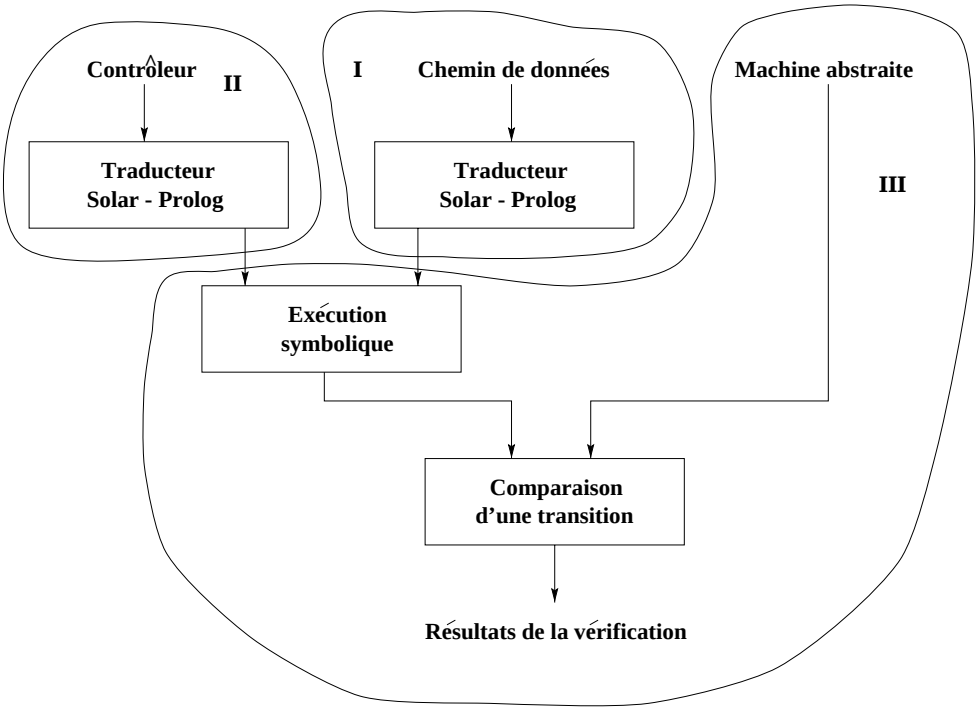


FIG. 5.8 – *Flot d'outil développé*

première étape de la vérification est montrée dans l'annexe C. Chaque transition du contrôleur est également transformée en Prolog sous la forme d'un appel du prédicat *data_path*. Un exemple de cet appel pour la première transition $S_1 \rightarrow S_2$ est illustré dans la figure 5.9.

Dans l'appel (figure 5.9) les signaux de contrôle sont remplacés par leurs valeurs effectives ($'0'$ ou $'1'$) de la transition $S_1 \rightarrow S_2$, les entrées primaires xi et yi sont représentées par les symboles i_xi et i_yi et les sorties du chemin de données sont remplacées par les variables du prédicat *data_path*. Les noms de ces variables commencent par la lettre majuscule "O" [10]: O_ou, O_dout, O_Flag_x et O_Flag_y. Lors de la comparaison des résultats de l'exécution symbolique seules les sorties O_ou et O_dout seront prises en compte comme sorties primaires du circuit Gcd. Les registres sont représentés par les symboles de la valeur courante (cur_i_x et cur_i_y) et par les variables de la valeur suivante ($Next\_i\_x$ et $Next\_i\_y$) calculées pendant l'exécution symbolique.

Lors de la deuxième étape de la vérification, les fichiers Prolog sont compilés à l'intérieur du système (Prolog) par la commande *compile_files* invariante pour tous les circuits vérifiés. Le chemin de données et les transitions du contrôleur sont alors prêts pour l'exécution symbolique. La troisième étape de la vérification charge d'abord le prédicat *data_path* (la traduction en Prolog du chemin de données) et ensuite les "queries" (les appels au prédicat présentés par les traductions des transitions du contrôleur). Les résultats de la comparaison sont sauvegardés dans le fichier "REPORT_ERROR" montré dans l'annexe D.

Le fichier contient une erreur de l'implémentation pour la première transition du contrôleur. La sortie *dout* prend la valeur '0' à la place de la valeur '1'. L'erreur a consisté à changer la valeur du signal $MX1\_dout\_Sel$ (le signal de contrôle du multiplexeur $MX1$) issu du contrôleur

10. Dans le système de Prolog utilisé un nom commençant par une lettre majuscule signifie une variable dont la valeur sera calculée lors de l'appel d'un prédicat.

```
:- data_path(
       O_ou,
       i_yi,
       i_xi,
       O_dout,
       1,
       1,
       0,
       0,
       O_FLAG_x,
       0,
       0,
       O_FLAG_y,
       0,
       0,
       0,
       0,
       0,

       cur_i_y,  Next_i_y,
       cur_i_x,  Next_i_x),

%%
%% Writing the outputs and registers contents during next clock cycle into FILE
%%

tell('evalS1_S2_1.aux'),

%%% The output values %%%
write(O_ou),nl,
write(O_dout),nl,
write(O_FLAG_x),nl,
write(O_FLAG_y),nl,

%%% The registers contents during next clock cycle %%%
write(Next_i_y),nl,
write(Next_i_x),nl,

told.
```

FIG. 5.9 – *Appel du prédicat data_path correspondant à la transition* $S_1 \rightarrow S_2$ *du contrôleur du Gcd*

de '0' en '1' dans le fichier du contrôleur (annexe B, le deuxième fichier[11]). Ce changement s'est manifesté lors de la vérification de l'allocation.

La liste des autres circuits avec lesquels nous avons testé notre outil est donnée ci-dessous:

- Le circuit qui réalise l'opération de multiplication à partir des opérations simples telles que addition, soustraction, décalage, etc. La description initiale en VHDL est donnée dans l'annexe E. Les fonctions *shftright*, *shiftleft*, *zerobit*, *resize* et *negation* sont mises en oeuvre par les unités fonctionnelles correspondantes. Après l'ordonnancement le circuit contient huit états et quatorze transitions. Le chemin de données généré consiste en trente-et-un éléments de onze type différents (les éléments d'entrée/sortie et de connexion inclus).

- Le bloc "tbunit" du circuit ATM ("Asynchronous Transfer Mode") conçu par le groupe SLS. Le bloc fournit la fonctionnalité de chronomètre ("tbunit" est l'abréviation de "time based unit"); sa description initiale est montrée dans l'annexe F. Comme pour le circuit de multiplication, les fonctions définies au début de l'architecture sont implémentées par des unités fonctionnelles. Après la synthèse, le "tbuint" contient sept états, onze transitions et vingt deux éléments de la partie opérative de dix types différents.

11. Dans l'annexe B, la version correcte du fichier est présentée.

Les deux circuits suivants sont les benchmarks de la synthèse de haut niveau utilisés pour tester l'étape d'allocation d'Amical. Ils nous ont été fournis par la personne qui met en oeuvre cette étape.

- Le filtre elliptique de cinquième degré ("The Fifth Order Elliptical Wave Filter") développé à l'Université de Californie par Sreenivase Rao. La description initiale VHDL est donnée dans l'annexe G et sa fonctionnalité peut être trouvé dans [PB]. Ce circuit est intéressant en raison d'une longue séquence d'affectations des variables et des signaux situées dans le processus principal. Même si les opérations n'ont pas une grande diversité (seules l'addition et la multiplication sont utilisées), leur nombre important implique un chemin de données assez large. Ainsi, après une session de la synthèse avec Amical le circuit contient quatorze états et quatorze transitions. La partie opérative, elle, consiste en soixante-seize éléments de treize types différents.

- Le dernier circuit que nous avons vérifié avec notre outil est une application médicale capable de détecter les caractéristiques des points appelés Q-R-S dans la séquence d'un ECG (électrocardiogramme). Le circuit peut servir, par exemple, pour l'interception (monitoring) des battements du coeur. La sortie RRpeak de cette application est surtout intéressante car elle passe à zéro si le coeur s'arrête. Le circuit a été ultérieurement développé par un groupe de chercheurs et d'ingénieurs de pays différents (Allemagne, États Unis). La description de la fonctionnalité peut être trouvée dans [Roy90], [RNMK90] et [PBB92]. Le VHDL initial comportemental avant synthèse est montré dans l'annexe H. Après la synthèse le circuit contient soixante-quinze états et cent quarante six transitions. La partie opérative générée après la synthèse consiste en soixante-quinze éléments de dix-huit types différents.

Sur la station Sparc-20 (130Mb, sous la SunOs 4.1.3_U1), notre outil vérifie l'étape d'allocation des circuits énumérés ci-dessus en moins de trois minutes, toutes étapes de la vérification inclues. L'outil n'a pas trouvé d'erreurs d'implémentation sauf pour le dernier circuit où il n'a pas su trouver l'équivalence entre l'expression $((ftm1+tmp)-v)$ et l'expression $(ftm1+(tmp-v))$. Une solution à ce type de problèmes est l'utilisation conjointe de notre outil avec un démonstrateur de théorèmes général comme nous l'avons indiqué auparavant. Toutes les erreurs volontairement introduites ont été détectées.

5.7 Résumé

Dans ce chapitre nous avons décrit la méthode pour la vérification de la deuxième étape de la synthèse de haut niveau - l'allocation. Tout d'abord nous avons développé le modèle du circuit obtenu, composé d'un modèle pour le contrôleur et d'un modèle pour le chemin de données. La composition de ces deux modèles revient au modèle de la machine abstraite présenté dans le chapitre 3. La forme unique (la machine abstraite) des modèles avant et après l'allocation permet de les comparer et de prouver l'exactitude de cette étape.

Nous avons décrit ensuite le prototype d'un outil développé. Cet outil compare les deux machines abstraites transition par transition. Les affectations du modèle après l'allocation sont

obtenues par l'exécution symbolique mise en oeuvre à l'aide du système Prolog.

Finalement, nous avons présenté des exemples d'application avec lesquels nous avons testé notre outil. Le temps de la vérification de l'étape d'allocation est très prometteur malgré l'utilisation de Prolog qui est un système relativement lent. L'amélioration immédiate de l'outil consiste à le mettre en interaction avec un démonstrateur général de théorèmes.

Chapitre 6

Conclusion

6.1 Travaux accomplis

Dans cette thèse nous avons développé une méthodologie pour la vérification des résultats de la synthèse de haut niveau. La méthode proposée est fondée sur le modèle de la machine abstraite correspondant à un circuit après l'étape d'ordonnancement. Le modèle, inspiré par le FSMD (Finite State Machine with a Datapath) de Gajski, représente un premier raffinement d'une description comportementale d'un circuit vers une architecture réelle. Les notions fondamentales d'une description comportementale, telles que les opérations d'un algorithme, sont conservées dans le modèle de la machine abstraite sous la forme des affectations des variables mémorisées. Les objets manipulés par les opérations (variables, signaux) sont restreints, cependant, à un nombre de valeurs fini et limité par les largeurs des vecteurs de bits attribués à ces objets. On passe ainsi d'une représentation abstraite avec des ensembles de valeurs *infinis* à une représentation concrète avec des ensembles de valeurs *finis* et réalisables par un vrai circuit matériel.

Nous avons démontré que le modèle de la machine abstraite est équivalent au modèle de la machine d'états finis classique et que les transformations dans les deux directions sont possibles. Ce résultat théorique ouvre des perspectives intéressantes pour le développement des algorithmes formels de la synthèse logique à partir de la machine abstraite.

Fondés sur le modèle de base, deux autres modèles, pour la description initiale comportementale et pour l'architecture finale, ont été développés. L'avantage de ces modèles est leur ressemblance avec les descriptions VHDL correspondantes. Ainsi, dans le modèle initial une séquence d'affectations primitives représente une séquence d'opérations de la description comportementale. Le modèle de l'architecture finale est composé, comme la description VHDL correspondante, de deux sous-modèles: l'un pour le contrôleur et l'autre pour le chemin de données.

La vérification des résultats de la synthèse de haut niveau est effectuée pour chaque étape principale de la synthèse: l'ordonnancement et l'allocation. L'exactitude de chaque étape est prouvée si les deux modèles, au début et à la fin de l'étape, sont équivalents.

Or les trois modèles étant légèrement différents, nous avons développé deux algorithmes qui transforment le modèle de la description initiale et le modèle de l'architecture finale en modèle de base - la machine abstraite. Les étapes de la synthèse sont ensuite prouvées exactes si les deux machines abstraites, au début et à la fin de chaque étape, sont équivalentes. L'équivalence de

machines abstraites est définie transition par transition, à condition que les machines comparées possèdent le même ensemble d'états. Lors du développement des algorithmes de vérification, un bug dans le programme d'ordonnancement d'Amical a été relevé.

Finalement, basé sur la méthodologie proposée, un prototype de vérification de l'étape d'allocation a été réalisé. Cet outil utilise le système Prolog pour extraire les opérations effectuées par l'architecture finale et les compare ensuite avec les opérations spécifiées dans la machine abstraite correspondant au circuit avant l'allocation. L'outil a été testé avec quelque exemples, parmi lesquels deux benchmarks de la synthèse comportementale. Les expériences montrent l'efficacité de la méthode pour la vérification fonctionnelle car le temps de vérification ne dépasse pas trois minutes même en utilisant Prolog, système relativement lent.

6.2 Perspectives

Les perspectives de notre travail portent sur l'aspect pratique aussi bien que sur l'aspect théorique. Sur le plan pratique, des améliorations du prototype développé sont possibles. Le "mariage" de notre outil avec un démonstrateur général de théorèmes permettra de prouver l'équivalence des opérations sémantiquement (et non seulement syntaxiquement) égales. Par conséquent, une étape d'allocation plus sophistiquée pourra être vérifiée par l'outil.

La mise en oeuvre d'un outil pour la vérification de l'étape d'ordonnancement, non implémenté en raison du manque de temps, constitue une deuxième perspective immédiate. Tous les algorithmes nécessaires sont développés dans la thèse et la réalisation ne doit pas poser de problèmes surtout si la comparaison syntaxique des opérations est implémentée dans un premier temps. Pour un ordonnancement plus complexe, un démonstrateur de théorèmes doit être appliqué comme pour la vérification de l'étape d'allocation.

Sur le plan théorique, plusieurs axes de travail sont envisageables. Tout d'abord, l'équivalence entre la machine abstraite et la machine d'états finis classique (chapitre 3) permet d'espérer que des algorithmes formels de synthèse logique peuvent être développés à partir de la machine abstraite. Ces algorithmes doivent complètement se différencier des algorithmes heuristique actuels et se fonder sur les notions formelles telles que la définition de fonctions, de types, λ-calcul, etc. Les résultats de la synthèse formelle peuvent être utilisés pour la vérification des résultats de la synthèse ordinaire, ou bien, si une efficacité suffisante est atteinte, pour remplacer la synthèse logique actuelle.

Le deuxième axe de la recherche porte sur la définition d'équivalence de machines abstraites. Dans cette thèse, nous avons défini deux machines comme équivalentes si elles ont un ensemble identique d'états. On ramène ainsi les machine comparées à la même échelle temporelle. Lors de la synthèse, cependant, une transition de la description initiale peut être raffinée en plusieurs transitions de l'architecture finale. Pour mieux refléter cette particularité de la synthèse comportementale, deux machines abstraites peuvent être redéfinies équivalentes, si chaque sortie d'une machine (la spécification) est équivalente à une composition de sorties de l'autre (l'implémentation). Une telle définition d'équivalence peut servir de base pour la vérification des résultats de toutes les synthèses à partir de la synthèse comportementale car la notion de raffinement et d'abstraction est largement utilisée: une étape de la spécification est raffinée en plusieurs étapes

de l'implémentation.

La définition, néanmoins, est loin d'être facile. Premièrement, une sortie de la spécification est la composition de combien de sorties de l'implémentation? Comment savoir à quel moment on peut arrêter la composition et conclure qu'une sortie de la spécification n'a pas d'analogue dans l'implémentation? Une solution éventuelle peut être inspirée par la théorie des automates infinis où les automates sont équivalents si ils produisent des traces infinies équivalentes. Deuxièmement, comment définir l'équivalence si la composition de sorties de l'implémentation est égale non pas à une sortie, mais à une composition de sorties de la spécification? La considération de ce cas compliquera encore la définition et nécessitera le développement de nouveaux algorithmes pour la vérification de l'équivalence de machines abstraites.

Finalement, le troisième axe qui offre, à notre avis, le plus de perspective de recherche, concerne la vérification de la faisabilité de la machine abstraite. Rappelons (chapitre 3) que les machines abstraites ne sont pas toutes faisables. Par exemple, quel que soit le registre attribué à la variable v, l'opération $v := v + 1$ ne peut pas être réalisée si v atteint la valeur maximum permise par ce registre. Cependant, la possibilité d'atteindre la valeur maximum dépend de l'algorithme représenté par la machine abstraite.

La preuve de la faisabilité consiste donc à examiner la machine abstraite pour chaque variable mémorisée et démontrer que la valeur acquise par la variable ne dépasse jamais les valeurs extrêmes permises par le registre correspondant. La technique employée peut ressembler à celle du "symbolic model checking" où un système de transitions est examiné état par état jusqu'à ce que tous les états soient explorés. Si la condition de la faisabilité est satisfaite, alors n'importe quelle opération $c = a \otimes b$ de la machine abstraite peut être réécrite de la façon suivante: $c = (a \otimes b) \, mod \, 2^{length\_of\_result}$ où $length\_of\_result$ est la largeur du registre choisi pour c. Cette nouvelle opération ressemble aux opérations réalisées par des composants de circuits intégrés: le résultat est toujours limité par la largeur du vecteur de bits de la sortie. Nous espérons donc que cette direction de recherche permettra de franchir la barrière entre la vérification purement fonctionnelle et la vérification des circuits réels où la fonctionnalité est toujours limitée par des paramètres matériels.

Annexe A

Sortie du "scheduleur" d'Amical après l'ordonnancement du Gcd

```
(SOLAR  gcd_v2_example
   (DESIGNUNIT  gcd
      (VIEW  behavior
         (INTERFACE
            (PORT (ARRAY clk 1)  (DIRECTION  in )  (BIT ))
            (PORT (ARRAY reset 1)  (DIRECTION  in )  (BIT ))
            (PORT (ARRAY st 1)  (DIRECTION  in )  (BIT ))
            (PORT (ARRAY din 1)  (DIRECTION  in )  (BIT ))
            (PORT (ARRAY xi 16)  (DIRECTION  in )  (INTEGER ))
            (PORT (ARRAY yi 16)  (DIRECTION  in )  (INTEGER ))
            (PORT (ARRAY dout 1)  (DIRECTION  out )  (BIT ))
            (PORT (ARRAY ou 16)  (DIRECTION  out )  (INTEGER )) )
         (CONTENTS
            (STATETABLE  p1
            (STATELIST  S1 S2 S3 S4 S5)
            (ENTRYSTATE S1  )
            (VARIABLE (ARRAY x 16))
            (VARIABLE (ARRAY y 16))
               (STATE S1
                  (CASE
                     (Alt (AND ( =  st 1) (True) )
                           (ASSIGN  dout  0)
                        (NEXTSTATE S2))
                     (Alt  (| (/=  st 1) (False) )
                        (NEXTSTATE S1)) ) )
               (STATE S2
                  (CASE
                     (Alt (AND ( =  din 1) (True) )
                           (ASSIGN  x  xi)
                           (ASSIGN  y  yi)
                        (NEXTSTATE S3))
                     (Alt  (| (/=  din 1) (False) )
                        (NEXTSTATE S2)) ) )
               (STATE S3
                  (CASE
                     (Alt (AND ( /=  x y)( <  x y))
                           (ASSIGN  y (-  y  x ))
                        (NEXTSTATE S4))
                     (Alt (AND ( /=  x y)(>=  x y))
                           (ASSIGN  x (-  x  y ))
                        (NEXTSTATE S4))
```

```
                  (Alt  (=  x y)
                        (ASSIGN  ou  x)
                        (ASSIGN  dout  1)
                     (NEXTSTATE S5)) ) )
               (STATE S4
                  (CASE
                     (Alt (AND ( /=  x y)( <  x y))
                           (ASSIGN  y (-  y  x ))
                        (NEXTSTATE S4))
                     (Alt (AND ( /=  x y)(>=  x y))
                           (ASSIGN  x (-  x  y ))
                        (NEXTSTATE S4))
                     (Alt  (=  x y)
                           (ASSIGN  ou  x)
                           (ASSIGN  dout  1)
                        (NEXTSTATE S5)) ) )
               (STATE S5
                  (CASE
                     (Alt (AND ( =  din 0) (True) )
                           (ASSIGN  dout  0)
                        (NEXTSTATE S2))
                     (Alt  (| (/=  din 0) (False) )
                        (NEXTSTATE S5)) ) )
            )
         )
      )
   )
)
```

Annexe B

Circuit Gcd après synthèse de haut niveau

B.1 Contrôleur du Gcd

```
--------------------------------------------------------------------------
-- Generated by AMICAL transVHDL (PAT) -- (C) TIMA/INPG
-- Version      : V 3.0 Beta of   Thu Jan 25 09:46:35 MET 1996
-- Revision     : 96/03 (C)
--
-- Design Unit  : gcd_v2_control
-- Solar  File  : gcd_v2_control.solar  (** MUX based architecture **)
-- Global File  : gcd_v2.global
--
-- This   File  : gcd_v2_control.vhd
-- Generated on : Wednesday, May 12, 1999 at 16:30:24
--------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity gcd_v2_control  is
------
port    (cont_clk : IN std_ulogic;
         cont_reset : IN std_ulogic;
         st  :  IN  STD_ULOGIC;
         din  :  IN  STD_ULOGIC;
         MX1_dout_Sel  :  OUT  STD_ULOGIC;
         C_Sel_dout  :  OUT  STD_ULOGIC;
         MX2_x_Sel  :  OUT  STD_ULOGIC;
         C_W_x  :  OUT  STD_ULOGIC;
         FLAG_x:  IN  STD_Logic_Vector(15 downto 0);
         MX3_y_Sel  :  OUT  STD_ULOGIC;
         C_W_y  :  OUT  STD_ULOGIC;
         FLAG_y:  IN  STD_Logic_Vector(15 downto 0);
         MX4_in1_FU_1_Sel  :  OUT  STD_ULOGIC;
         MX5_in2_FU_1_Sel  :  OUT  STD_ULOGIC;
         C_Sel_ou  :  OUT  STD_ULOGIC;
         C_Sel_yi  :  OUT  STD_ULOGIC;
         C_Sel_xi  :  OUT  STD_ULOGIC);
end gcd_v2_control ;    -- of Entity
```

```
architecture  RTL  of gcd_v2_control  is
type STATE_TYPE is ( S1, S2, S3, S4, S5 );
signal CURRENT_STATE,NEXT_STATE:STATE_TYPE;

begin      -- of architecture
-----
   Controller  : process ( st,
                           din,
                           FLAG_x,
                           FLAG_y,
                           CURRENT_STATE)
   begin     -- of process
                         MX1_dout_Sel  <=  '0';
                         C_Sel_dout  <=  '0';
                         MX2_x_Sel  <=  '0';
                         C_W_x  <=  '0';
                         MX3_y_Sel  <=  '0';
                         C_W_y  <=  '0';
                         MX4_in1_FU_1_Sel  <=  '0';
                         MX5_in2_FU_1_Sel  <=  '0';
                         C_Sel_ou  <=  '0';
                         C_Sel_yi  <=  '0';
                         C_Sel_xi  <=  '0';
                         NEXT_STATE <= S1;
case CURRENT_STATE is

when ( S1 ) =>
----
  if ( (  st  =  '1') and  True ) then
                         C_Sel_dout  <=  '1';
                         MX1_dout_Sel  <=  '0';
                         NEXT_STATE <= S2;
  end if;
  if ( (  st  /=  '1') or  False ) then
                         NEXT_STATE <= S1;
  end if;

when ( S2 ) =>
----
  if ( (  din  =  '1') and  True ) then
                         C_W_x  <=  '1';
                         MX2_x_Sel  <=  '0';
                         C_Sel_xi  <=  '1';
                         C_W_y  <=  '1';
                         MX3_y_Sel  <=  '0';
                         C_Sel_yi  <=  '1';
                         NEXT_STATE <= S3;
  end if;
  if ( (  din  /=  '1') or  False ) then
                         NEXT_STATE <= S2;
  end if;

when ( S3 ) =>
----
  if ( ( ( FLAG_x ) /= ( FLAG_y )) and ( ( FLAG_x ) < ( FLAG_y ))) then
                         MX4_in1_FU_1_Sel  <=  '0';
                         C_W_y  <=  '1';
                         MX5_in2_FU_1_Sel  <=  '0';
```

```
                        C_W_x  <=  '0';
                        MX3_y_Sel  <=  '1';
                        NEXT_STATE <= S4;
  end if;
  if ( ( ( FLAG_x ) /= ( FLAG_y )) and ( ( FLAG_x ) >= ( FLAG_y ))) then
                        MX4_in1_FU_1_Sel  <=  '1';
                        C_W_x  <=  '1';
                        MX5_in2_FU_1_Sel  <=  '1';
                        C_W_y  <=  '0';
                        MX2_x_Sel  <=  '1';
                        NEXT_STATE <= S4;
  end if;
  if ( ( FLAG_x ) = ( FLAG_y )) then
                        C_Sel_ou  <=  '1';
                        C_W_x  <=  '0';
                        C_Sel_dout  <=  '1';
                        MX1_dout_Sel  <=  '1';
                        NEXT_STATE <= S5;
  end if;

when ( S4 ) =>
----
  if ( ( ( FLAG_x ) /= ( FLAG_y )) and ( ( FLAG_x ) < ( FLAG_y ))) then
                        MX4_in1_FU_1_Sel  <=  '0';
                        C_W_y  <=  '1';
                        MX5_in2_FU_1_Sel  <=  '0';
                        C_W_x  <=  '0';
                        MX3_y_Sel  <=  '1';
                        NEXT_STATE <= S4;
  end if;
  if ( ( ( FLAG_x ) /= ( FLAG_y )) and ( ( FLAG_x ) >= ( FLAG_y ))) then
                        MX4_in1_FU_1_Sel  <=  '1';
                        C_W_x  <=  '1';
                        MX5_in2_FU_1_Sel  <=  '1';
                        C_W_y  <=  '0';
                        MX2_x_Sel  <=  '1';
                        NEXT_STATE <= S4;
  end if;
  if ( ( FLAG_x ) = ( FLAG_y )) then
                        C_Sel_ou  <=  '1';
                        C_W_x  <=  '0';
                        C_Sel_dout  <=  '1';
                        MX1_dout_Sel  <=  '1';
                        NEXT_STATE <= S5;
  end if;

when ( S5 ) =>
----
  if ( (  din  =  '0') and  True ) then
                                C_Sel_dout  <=  '1';
                        MX1_dout_Sel  <=  '0';
                        NEXT_STATE <= S2;
  end if;
  if ( (  din  /=  '0') or  False ) then
                        NEXT_STATE <= S5;
  end if;
end case ;
  end process  Controller ;
```

```
  SYNCH:process ( cont_clk, cont_reset )
    begin     -- of SYNCH: process
    if (cont_reset = '0') then
       CURRENT_STATE <= S1;
    elsif (cont_clk'event and cont_clk = '1') then
       CURRENT_STATE <= NEXT_STATE;
    end if;
  end process SYNCH;

end RTL;    -- of architecture

-- ============================================================+
-- IMPORTANT : If modified, please SAVE the contents to another file. |
--                                                             |
-- We support:   E-mail: amical@imag.fr  <**>  Fax: (+33) 76-47-38-14 |
-- ============================================================+
-- Wednesday, May 12, 1999 at 16:30:24
```

B.2 Chemin de données du Gcd

```
-------------------------------------------------------------------
-- Generated by AMICAL transVHDL (PAT) -- (C) TIMA/INPG
-- Version      : V 3.0 Beta of   Thu Jan 25 09:46:35 MET 1996
-- Revision     : 96/03 (C)
--
-- Design Unit  : gcd_v2_datapath
-- Solar  File  : gcd_v2_datapath.solar  (** MUX based architecture **)
-- Global File  : gcd_v2.global
--
-- This   File  : gcd_v2_datapath.vhd  (* Generic *)
-- Generated on : Wednesday, May 12, 1999 at 16:31:04
-------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity  gcd_v2_datapath is
------
port    (data_clk : IN std_ulogic;
         data_reset : IN std_ulogic;
         EBUS_1 :  OUT STD_LOGIC_VECTOR (15 downto 0);
         EBUS_2 :  IN STD_LOGIC_VECTOR (15 downto 0);
         EBUS_3 :  IN STD_LOGIC_VECTOR (15 downto 0);
         EBUS_4 :  OUT STD_ULOGIC;
         MX1_dout_Sel :  IN STD_ULOGIC;
         C_Sel_dout :  IN STD_ULOGIC;
         MX2_x_Sel :  IN STD_ULOGIC;
         C_W_x :  IN STD_ULOGIC;
         FLAG_x :  OUT STD_LOGIC_VECTOR (15 downto 0);
         MX3_y_Sel :  IN STD_ULOGIC;
         C_W_y :  IN STD_ULOGIC;
         FLAG_y :  OUT STD_LOGIC_VECTOR (15 downto 0);
         MX4_in1_FU_1_Sel :  IN STD_ULOGIC;
         MX5_in2_FU_1_Sel :  IN STD_ULOGIC;
```

```
        C_Sel_ou :  IN STD_ULOGIC;
        C_Sel_yi :  IN STD_ULOGIC;
        C_Sel_xi :  IN STD_ULOGIC);
end gcd_v2_datapath;

architecture  STRUCTURE   of gcd_v2_datapath is
------------
-- Signals from the Global File                      --- G L O B A L --+
-- End of Signals from the Global File               --- G L O B A L --+
-------------------------------------------------------------------------+
-- Signal Prefixing Conventions:                                        |
-------------------------------------------------------------------------|
--                                                                      |
-- 'A_'   : Alias Signal (Replacable by 'alias' if synthesis permits.) |
-- 'UR_'  : UnResolved signal (Some of them, probably, are unused.)     |
-- 'B_'   : Signal required to map a vector data type onto BIT.         |
-- 'V_'   : Signal required to map a bit data type onto Vector.         |
-------------------------------------------------------------------------+
signal  in1_MX1_Data_OUT_0   : STD_Logic_Vector (15 downto 0);
signal  UR_in1_MX1_Data_OUT_0: STD_Ulogic_Vector(15 downto 0);
signal  MX1_in1_A_in1_MX1_Data_OUT_0_0downto0   : STD_Logic_Vector (0 downto 0);
signal  UR_in1_MX1_A_0downto0in1_MX1_Data_OUT_0: STD_Ulogic_Vector(0 downto 0);

signal  in2_MX1_Data_OUT_1   : STD_Logic_Vector (15 downto 0);
signal  UR_in2_MX1_Data_OUT_1: STD_Ulogic_Vector(15 downto 0);
signal  MX1_in2_A_in2_MX1_Data_OUT_1_0downto0   : STD_Logic_Vector (0 downto 0);
signal  UR_in2_MX1_A_0downto0in2_MX1_Data_OUT_1: STD_Ulogic_Vector(0 downto 0);

signal  out1_MX1_E_IN_dout   : STD_Logic_Vector (0 downto 0);
signal  B_out1_MX1_E_IN_dout : STD_Logic;
signal  UR_out1_MX1_E_IN_dout: STD_Ulogic_Vector(0 downto 0);

signal  in1_MX2_E_OUT_xi   : STD_Logic_Vector (15 downto 0);
signal  UR_in1_MX2_E_OUT_xi: STD_Ulogic_Vector(15 downto 0);

signal  in2_MX2_out1_FU_1   : STD_Logic_Vector (15 downto 0);
signal  UR_in2_MX2_out1_FU_1: STD_Ulogic_Vector(15 downto 0);
signal  FU_1_out1_A_in2_MX2_out1_FU_1_7downto0   : STD_Logic_Vector (7 downto 0);
signal  UR_out1_FU_1_A_7downto0in2_MX2_out1_FU_1: STD_Ulogic_Vector(7 downto 0);

signal  out1_MX2_Data_IN_x   : STD_Logic_Vector (15 downto 0);
signal  UR_out1_MX2_Data_IN_x: STD_Ulogic_Vector(15 downto 0);

signal  in1_MX3_E_OUT_yi   : STD_Logic_Vector (15 downto 0);
signal  UR_in1_MX3_E_OUT_yi: STD_Ulogic_Vector(15 downto 0);

signal  out1_MX3_Data_IN_y   : STD_Logic_Vector (15 downto 0);
signal  UR_out1_MX3_Data_IN_y: STD_Ulogic_Vector(15 downto 0);

signal  in1_MX4_Data_OUT_y   : STD_Logic_Vector (15 downto 0);
signal  UR_in1_MX4_Data_OUT_y: STD_Ulogic_Vector(15 downto 0);
signal  MX4_in1_A_in1_MX4_Data_OUT_y_7downto0   : STD_Logic_Vector (7 downto 0);
signal  UR_in1_MX4_A_7downto0in1_MX4_Data_OUT_y: STD_Ulogic_Vector(7 downto 0);
signal  MX5_in2_A_in1_MX4_Data_OUT_y_7downto0   : STD_Logic_Vector (7 downto 0);
signal  UR_in2_MX5_A_7downto0in1_MX4_Data_OUT_y: STD_Ulogic_Vector(7 downto 0);

signal  in2_MX4_Data_OUT_x   : STD_Logic_Vector (15 downto 0);
signal  UR_in2_MX4_Data_OUT_x: STD_Ulogic_Vector(15 downto 0);
```

```
signal  MX4_in2_A_in2_MX4_Data_OUT_x_7downto0   : STD_Logic_Vector (7 downto 0);
signal  UR_in2_MX4_A_7downto0in2_MX4_Data_OUT_x: STD_Ulogic_Vector(7 downto 0);
signal  MX5_in1_A_in2_MX4_Data_OUT_x_7downto0   : STD_Logic_Vector (7 downto 0);
signal  UR_in1_MX5_A_7downto0in2_MX4_Data_OUT_x: STD_Ulogic_Vector(7 downto 0);

signal  out1_MX4_in1_FU_1   : STD_Logic_Vector (7 downto 0);
signal  UR_out1_MX4_in1_FU_1: STD_Ulogic_Vector(7 downto 0);

signal  out1_MX5_in2_FU_1   : STD_Logic_Vector (7 downto 0);
signal  UR_out1_MX5_in2_FU_1: STD_Ulogic_Vector(7 downto 0);
--
-- Following signals may be used to connect input ports if left open:
--
signal  open_0           : STD_Logic;
signal  UR_open_0        : STD_Ulogic;

-------------------------------------------------------------------------
-- Following signals (if present) are necessary to map:
-- a) STD_Logic  <=> STD_LOGIC_Vector  (0 downto 0)
-- b) STD_Ulogic <=> STD_Ulogic_Vector (0 downto 0)
-- c) STD_Ulogic_Vector (N downto 0) <=> STD_Logic_Vector (N downto 0)
-------------------------------------------------------------------------
signal  UR_EBUS_1: STD_Ulogic_Vector(15 downto 0);
signal  V_EBUS_4   : STD_Logic_Vector (0 downto 0);
signal  UR_V_EBUS_4: STD_Ulogic_Vector(0 downto 0);
signal  UR_FLAG_x: STD_Ulogic_Vector(15 downto 0);
signal  UR_FLAG_y: STD_Ulogic_Vector(15 downto 0);

-- END of signal declarations. --

component  ExtCon_out
---------
generic (Width_E_IN,
         Width_E_OUT : positive := 1);
port    (E_IN :  IN STD_LOGIC_VECTOR (Width_E_IN-1 downto 0);
         E_OUT :  OUT STD_LOGIC_VECTOR (Width_E_OUT-1 downto 0));
end component;

component  ExtCon_in
---------
generic (Width_E_IN,
         Width_E_OUT : positive := 1);
port    (E_IN :  IN STD_LOGIC_VECTOR (Width_E_IN-1 downto 0);
         E_OUT :  OUT STD_LOGIC_VECTOR (Width_E_OUT-1 downto 0));
end component;

component  FU_SUB
---------
generic (Width_in1,
         Width_in2,
         Width_out1 : positive := 1);
port    (Sel :  IN STD_ULOGIC;
         in1 :  IN STD_LOGIC_VECTOR (Width_in1-1 downto 0);
         in2 :  IN STD_LOGIC_VECTOR (Width_in2-1 downto 0);
         out1 :  OUT STD_LOGIC_VECTOR (Width_out1-1 downto 0));
end component;

component  ConstReg
```

```
---------
generic (VALUE: natural; Width_Data_OUT : Positive := 1);
port    (Data_OUT :  OUT STD_LOGIC_VECTOR (Width_Data_OUT-1 downto 0));
end component;

component  FlagReg
---------
generic (Width_Data_IN,
         Width_Data_OUT,
         Width_CR : positive := 1);
port    (reset :  IN STD_ULOGIC;
         clk :  IN STD_ULOGIC;
         W :  IN STD_ULOGIC;
         Data_IN :  IN STD_LOGIC_VECTOR (Width_Data_IN-1 downto 0);
         Data_OUT :  OUT STD_LOGIC_VECTOR (Width_Data_OUT-1 downto 0);
         CR :  OUT STD_LOGIC_VECTOR (Width_Data_OUT-1 downto 0));
end component;

component  Mux_2
---------
generic (Width_in1,
         Width_in2,
         Width_out1 : positive := 1);
port    (in1 :  IN STD_LOGIC_VECTOR (Width_in1-1 downto 0);
         in2 :  IN STD_LOGIC_VECTOR (Width_in2-1 downto 0);
         out1 :  OUT STD_LOGIC_VECTOR (Width_out1-1 downto 0);
         Sel :  IN STD_ULOGIC);
end component;

-------------------------------------------------------------+
-- Abbreviations followed in the 'port map' documentation    |
-------------------------------------------------------------|
--                                                           |
-- 'R'    : Port is of type Resolved      (Ex: STD_Logic).   |
-- 'UR'   : Port is of type UnResolved.   (Ex: STD_Ulogic).  |
-- 'B'    : Port is Single Bit.                              |
-- 'V'    : Port is a Vector (array).                        |
-- 'IN'   : Port direction is IN (similarly OUT, INOUT).     |
-- '(G)'  : Port map is established via the Global File.     |
-------------------------------------------------------------+

begin      -- of architecture
-----
I_ou :  ExtCon_out
           generic map(
            Width_E_IN => 16,
            Width_E_OUT => 16 )
           port map(
            E_IN => in2_MX4_Data_OUT_x,  -- IN (R,V)
            E_OUT => EBUS_1);  -- OUT (R,V)
I_yi :  ExtCon_in
           generic map(
            Width_E_IN => 16,
            Width_E_OUT => 16 )
           port map(
            E_IN => EBUS_2,  -- IN (R,V)
            E_OUT => in1_MX3_E_OUT_yi);  -- OUT (R,V)
I_xi :  ExtCon_in
```

```
           generic map(
            Width_E_IN => 16,
            Width_E_OUT => 16 )
           port map(
            E_IN => EBUS_3,  -- IN (R,V)
            E_OUT => in1_MX2_E_OUT_xi);  -- OUT (R,V)
I_dout :  ExtCon_out
           generic map(
            Width_E_IN => 1,
            Width_E_OUT => 1 )
           port map(
            E_IN => out1_MX1_E_IN_dout,  -- IN (R,V)
            E_OUT => V_EBUS_4);  -- OUT (R,V)
--
EBUS_4 <= V_EBUS_4(0); -- Updating Bit from Vector
--
I_FU_1 :  FU_SUB
           generic map(
            Width_in1 => 8,
            Width_in2 => 8,
            Width_out1 => 8 )
           port map(
            Sel => UR_open_0,  -- IN (UR,B)   <<< WARNING ***: o p e n IN port >>>
            in1 => out1_MX4_in1_FU_1,  -- IN (R,V)
            in2 => out1_MX5_in2_FU_1,  -- IN (R,V)
            out1 => FU_1_out1_A_in2_MX2_out1_FU_1_7downto0);  -- OUT (R,V)
--
in2_MX2_out1_FU_1 (7 downto 0) <= FU_1_out1_A_in2_MX2_out1_FU_1_7downto0;    -- Update OUT port
in2_MX2_out1_FU_1 (15 downto 8) <=  "00000000";    -- Unused OUT bit(s)
--
I_C1 :  ConstReg
           generic map(
            VALUE => 1,
            Width_Data_OUT => 16 )
           port map(
            Data_OUT => in2_MX1_Data_OUT_1);  -- OUT (R,V)
I_C0 :  ConstReg
           generic map(
            VALUE => 0,
            Width_Data_OUT => 16 )
           port map(
            Data_OUT => in1_MX1_Data_OUT_0);  -- OUT (R,V)
I_y :  FlagReg
           generic map(
            Width_Data_IN => 16,
            Width_Data_OUT => 16,
            Width_CR => 1 )
           port map(
            reset => data_reset,  -- IN (UR,B)   -- (Global Port)
            clk => data_clk,  -- IN (UR,B)   -- (G)
            W => C_W_y,  -- IN (UR,B)
            Data_IN => out1_MX3_Data_IN_y,  -- IN (R,V)
            Data_OUT => in1_MX4_Data_OUT_y,  -- OUT (R,V)
            CR => FLAG_y);  -- OUT (R,V)
I_x :  FlagReg
           generic map(
            Width_Data_IN => 16,
            Width_Data_OUT => 16,
```

```
            Width_CR => 1 )
           port map(
            reset => data_reset,  -- IN (UR,B)   -- (G)
            clk => data_clk,  -- IN (UR,B)   -- (G)
            W => C_W_x,  -- IN (UR,B)
            Data_IN => out1_MX2_Data_IN_x,  -- IN (R,V)
            Data_OUT => in2_MX4_Data_OUT_x,  -- OUT (R,V)
            CR => FLAG_x);  -- OUT (R,V)
I_MX1 :  Mux_2
           generic map(
            Width_in1 => 1,
            Width_in2 => 1,
            Width_out1 => 1 )
           port map(
            in1 => MX1_in1_A_in1_MX1_Data_OUT_0_0downto0,  -- IN (R,V)
            in2 => MX1_in2_A_in2_MX1_Data_OUT_1_0downto0,  -- IN (R,V)
            out1 => out1_MX1_E_IN_dout,  -- OUT (R,V)
            Sel => MX1_dout_Sel);  -- IN (R,B)
--
MX1_in1_A_in1_MX1_Data_OUT_0_0downto0 <= in1_MX1_Data_OUT_0 (0 downto 0);    -- Load IN port
MX1_in2_A_in2_MX1_Data_OUT_1_0downto0 <= in2_MX1_Data_OUT_1 (0 downto 0);    -- Load IN port
--
I_MX2 :  Mux_2
           generic map(
            Width_in1 => 16,
            Width_in2 => 16,
            Width_out1 => 16 )
           port map(
            in1 => in1_MX2_E_OUT_xi,  -- IN (R,V)
            in2 => in2_MX2_out1_FU_1,  -- IN (R,V)
            out1 => out1_MX2_Data_IN_x,  -- OUT (R,V)
            Sel => MX2_x_Sel);  -- IN (R,B)
I_MX3 :  Mux_2
           generic map(
            Width_in1 => 16,
            Width_in2 => 16,
            Width_out1 => 16 )
           port map(
            in1 => in1_MX3_E_OUT_yi,  -- IN (R,V)
            in2 => in2_MX2_out1_FU_1,  -- IN (R,V)
            out1 => out1_MX3_Data_IN_y,  -- OUT (R,V)
            Sel => MX3_y_Sel);  -- IN (R,B)
I_MX4 :  Mux_2
           generic map(
            Width_in1 => 8,
            Width_in2 => 8,
            Width_out1 => 8 )
           port map(
            in1 => MX4_in1_A_in1_MX4_Data_OUT_y_7downto0,  -- IN (R,V)
            in2 => MX4_in2_A_in2_MX4_Data_OUT_x_7downto0,  -- IN (R,V)
            out1 => out1_MX4_in1_FU_1,  -- OUT (R,V)
            Sel => MX4_in1_FU_1_Sel);  -- IN (R,B)
--
MX4_in1_A_in1_MX4_Data_OUT_y_7downto0 <= in1_MX4_Data_OUT_y (7 downto 0);    -- Load IN port
MX4_in2_A_in2_MX4_Data_OUT_x_7downto0 <= in2_MX4_Data_OUT_x (7 downto 0);    -- Load IN port
--
I_MX5 :  Mux_2
           generic map(
```

```
            Width_in1 => 8,
            Width_in2 => 8,
            Width_out1 => 8 )
           port map(
            in1 => MX5_in1_A_in2_MX4_Data_OUT_x_7downto0,  -- IN (R,V)
            in2 => MX5_in2_A_in1_MX4_Data_OUT_y_7downto0,  -- IN (R,V)
            out1 => out1_MX5_in2_FU_1,  -- OUT (R,V)
            Sel => MX5_in2_FU_1_Sel);  -- IN (R,B)
--
MX5_in1_A_in2_MX4_Data_OUT_x_7downto0 <= in2_MX4_Data_OUT_x (7 downto 0);    -- Load IN port
MX5_in2_A_in1_MX4_Data_OUT_y_7downto0 <= in1_MX4_Data_OUT_y (7 downto 0);    -- Load IN port
--
end STRUCTURE;    -- of architecture

-- synopsys_synthesis_off
-- configuration cfg_gcd_v2_datapath of gcd_v2_datapath is
--  for STRUCTURE
--  end for;
-- end cfg_gcd_v2_datapath;   -- of Configuration
-- synopsys_synthesis_on

-- ============================================================+
-- IMPORTANT : If modified, please SAVE the contents to another file. |
--                                                                    |
-- We support:   E-mail: amical@imag.fr  <**>  Fax: (+33) 76-47-38-14 |
-- ============================================================+
-- Wednesday, May 12, 1999 at 16:31:05
```

Annexe C

Chemin de données du Gcd traduit en Prolog

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% Library element ExtCon_out(In,Out)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

pu_ExtCon_out(In,Out,1) :-
   Out = In.

pu_ExtCon_out(_,Out,_) :-
   Out = Out.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
i_ou(P_E_IN, P_E_OUT, P_Sel) :-
      pu_ExtCon_out(P_E_IN, P_E_OUT, P_Sel).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% Library element ExtCon_in(In,Out)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

pu_ExtCon_in(In,Out,_) :-
  Out = In.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

i_yi(P_E_IN, P_E_OUT, P_Sel) :-
      pu_ExtCon_in(P_E_IN, P_E_OUT, P_Sel).

i_xi(P_E_IN, P_E_OUT, P_Sel) :-
      pu_ExtCon_in(P_E_IN, P_E_OUT, P_Sel).

i_dout(P_E_IN, P_E_OUT, P_Sel) :-
      pu_ExtCon_out(P_E_IN, P_E_OUT, P_Sel).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% Library element FU_alu(X,Y,Op_code,Out)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

pu_FU_SUB(X,Y,Out) :-
   Out = X - Y.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
i_FU_1(P_in1, P_in2, P_out1) :-
      pu_FU_SUB(P_in1, P_in2, P_out1).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% Library element varreg ConstReg(Out, Value)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

pu_ConstReg(Out, Value) :-
   Out = Value.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

i_C1(P_Data_OUT) :-
      pu_ConstReg(P_Data_OUT, 1).

i_C0(P_Data_OUT) :-
      pu_ConstReg(P_Data_OUT, 0).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% Library element FlagReg(Reset,Write,In,Out,Flag,Current,Next)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

pu_FlagReg(In,Out,Flag,1,Current,Next) :-
   Out = Current,
   Flag = Current,
   Next = In.
pu_FlagReg(_,Out,Flag,_,Current,_) :-
   Out = Current,
   Flag = Current.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

i_y(P_Data_IN, P_Data_OUT, P_CR, P_W, Current, Next) :-
      pu_FlagReg(P_Data_IN, P_Data_OUT, P_CR, P_W, Current, Next).

i_x(P_Data_IN, P_Data_OUT, P_CR, P_W, Current, Next) :-
      pu_FlagReg(P_Data_IN, P_Data_OUT, P_CR, P_W, Current, Next).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% Library element Mux2(In1,In2,Out,Sel)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

pu_Mux_2(_,In2,Out,1) :-
   Out = In2.
pu_Mux_2(In1,_,Out,_) :-
   Out = In1.
%%%pu_Mux_2(_,_,_,_) :-
%%%   nl, write('!!!!! The control signal of the MUX2 block is not valid !!!!!'), nl.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

i_MX1(P_in1, P_in2, P_out1, P_Sel) :-
      pu_Mux_2(P_in1, P_in2, P_out1, P_Sel).

i_MX2(P_in1, P_in2, P_out1, P_Sel) :-
      pu_Mux_2(P_in1, P_in2, P_out1, P_Sel).

i_MX3(P_in1, P_in2, P_out1, P_Sel) :-
```

```
      pu_Mux_2(P_in1, P_in2, P_out1, P_Sel).

i_MX4(P_in1, P_in2, P_out1, P_Sel) :-
      pu_Mux_2(P_in1, P_in2, P_out1, P_Sel).

i_MX5(P_in1, P_in2, P_out1, P_Sel) :-
      pu_Mux_2(P_in1, P_in2, P_out1, P_Sel).


data_path(
         C_EBUS_1,
         C_EBUS_2,
         C_EBUS_3,
         C_EBUS_4,
         C_MX1_dout_Sel,
         C_C_Sel_dout,
         C_MX2_x_Sel,
         C_C_W_x,
         C_FLAG_x,
         C_MX3_y_Sel,
         C_C_W_y,
         C_FLAG_y,
         C_MX4_in1_FU_1_Sel,
         C_MX5_in2_FU_1_Sel,
         C_C_Sel_ou,
         C_C_Sel_yi,
         C_C_Sel_xi,

         Cur_i_y, Next_i_y,
         Cur_i_x, Next_i_x) :-

i_ou(C_in2_MX4_Data_OUT_x, C_EBUS_1, C_C_Sel_ou),

i_yi(C_EBUS_2, C_in1_MX3_E_OUT_yi, C_C_Sel_yi),

i_xi(C_EBUS_3, C_in1_MX2_E_OUT_xi, C_C_Sel_xi),

i_dout(C_out1_MX1_E_IN_dout, C_EBUS_4, C_C_Sel_dout),

i_FU_1(C_out1_MX4_in1_FU_1, C_out1_MX5_in2_FU_1, C_in2_MX2_out1_FU_1),

i_C1(C_in2_MX1_Data_OUT_1),

i_C0(C_in1_MX1_Data_OUT_0),

i_y(C_out1_MX3_Data_IN_y, C_in1_MX4_Data_OUT_y, C_FLAG_y, C_C_W_y, Cur_i_y, Next_i_y),

i_x(C_out1_MX2_Data_IN_x, C_in2_MX4_Data_OUT_x, C_FLAG_x, C_C_W_x, Cur_i_x, Next_i_x),

i_MX1(C_in1_MX1_Data_OUT_0, C_in2_MX1_Data_OUT_1, C_out1_MX1_E_IN_dout, C_MX1_dout_Sel),

i_MX2(C_in1_MX2_E_OUT_xi, C_in2_MX2_out1_FU_1, C_out1_MX2_Data_IN_x, C_MX2_x_Sel),

i_MX3(C_in1_MX3_E_OUT_yi, C_in2_MX2_out1_FU_1, C_out1_MX3_Data_IN_y, C_MX3_y_Sel),

i_MX4(C_in1_MX4_Data_OUT_y, C_in2_MX4_Data_OUT_x, C_out1_MX4_in1_FU_1, C_MX4_in1_FU_1_Sel),

i_MX5(C_in2_MX4_Data_OUT_x, C_in1_MX4_Data_OUT_y, C_out1_MX5_in2_FU_1, C_MX5_in2_FU_1_Sel), !.
```

Annexe D

Résultats de la vérification du circuit GCD (l'étape d'allocation)

```
               ATTENTION!
 Verification is possible if and only if
Variable and Output names are NOT changed
                  and
 State names and NUMBER are NOT changed

TRANSITION evalS1_S2_1:
----------
OUTPUTS:

ERROR: the assignment of ‘dout’ is not consistent with specification
       SPECIFICATION:
 0
       IMPLEMENTATION:
 1
REGISTERS: Ok

TRANSITION evalS1_S1_1:
----------
OUTPUTS: Ok
REGISTERS: Ok

TRANSITION evalS2_S3_1:
----------
OUTPUTS: Ok
REGISTERS: Ok

TRANSITION evalS2_S2_1:
----------
OUTPUTS: Ok
REGISTERS: Ok

TRANSITION evalS3_S4_1:
----------
OUTPUTS: Ok
REGISTERS: Ok

TRANSITION evalS3_S4_2:
----------
OUTPUTS: Ok
```

```
REGISTERS: Ok

TRANSITION evalS3_S5_1:
----------
OUTPUTS: Ok
REGISTERS: Ok

TRANSITION evalS4_S4_1:
----------
OUTPUTS: Ok
REGISTERS: Ok

TRANSITION evalS4_S4_2:
----------
OUTPUTS: Ok
REGISTERS: Ok

TRANSITION evalS4_S5_1:
----------
OUTPUTS: Ok
REGISTERS: Ok

TRANSITION evalS5_S2_1:
----------
OUTPUTS: Ok
REGISTERS: Ok

TRANSITION evalS5_S5_1:
----------
OUTPUTS: Ok
REGISTERS: Ok
```

Annexe E

Description initiale VHDL du circuit Mult

```
package op_pkg is
   subtype int16bit is integer range -1024 to 1023;
end op_pkg;

package synchro is
   function rising_edge(signal sig_clock:bit) return boolean;
   function falling_edge(signal sig_clock:bit) return boolean;
end synchro;

package body synchro is
   function rising_edge(signal sig_clock:bit) return boolean is
   begin
     return (sig_clock'event and sig_clock='1' and sig_clock'last_value='0');
   end rising_edge;

   function falling_edge(signal sig_clock:bit) return boolean is
   begin
     return (sig_clock'event and sig_clock='0' and sig_clock'last_value='1');
   end falling_edge;
end synchro;

use work.op_pkg.all;
use work.synchro.All;

entity mult is
        port (reset        : in bit;               -- reset signal
              clock        : in bit;               -- clock signal
              sel        : in bit;              -- chip enable signal
              com        : in bit;               -- operation select signal
              input1        : in int16bit;               -- data input 1
              input2        : in int16bit;               -- data input 2
              outval        : out integer;               -- data output
              outdone        : out bit);               -- data output flag
end mult;


architecture behavior of mult is

  component FU_shiftrt
        port (in1        :in integer;
              out1        :out integer);
```

```
  end component;
  signal shiftrt_in1: integer:=0; signal shiftrt_out: integer;

  component FU_shiftlt
        port (in1        :in integer;
              out1        :out integer);
  end component;
  signal shiftlt_in1: integer:=0; signal shiftlt_out: integer;

  component FU_zerobit
        port (in1        :in integer;
              out1        :out bit);
  end component;
  signal zerobit_in1: integer:=0; signal zerobit_out1: bit;

  component FU_resize
        port (in1        :in int16bit;
              out1        :out integer);
  end component;
  signal resize_in1: int16bit:=5; signal resize_out1: integer;

  component FU_negation
        port (in1        :in integer;
              out1        :out integer);
  end component;
  signal negation_in1: integer:=2; signal negation_out1: integer;


begin
    process
        variable ADVAL, B_32bit, RES, X: integer;
        variable A, B: int16bit;
        variable bit0: bit;

        procedure shiftright(a: in integer; x: out integer) is
        begin
             shiftrt_in1 <= a;
             wait until rising_edge (clock);
             x := shiftrt_out;
        end shiftright;

        procedure shiftleft(a: in integer; x: out integer) is
        begin
             shiftlt_in1 <= a;
             wait until rising_edge (clock);
             x := shiftlt_out;
        end shiftleft;

        procedure zerobit(a: in integer; x: out bit) is
        begin
             zerobit_in1 <= a;
             wait until rising_edge (clock);
             x := zerobit_out1;
        end zerobit;

        procedure resize(a: in int16bit; x: out integer) is
        begin
             resize_in1 <= a;
```

```
          wait until rising_edge (clock);
          x := resize_out1;
     end resize;

     procedure negation(a: in integer; x: out integer) is
     begin
          negation_in1 <= a;
          wait until rising_edge (clock);
          x := negation_out1;
     end negation;

begin
     wait until (sel='1' and rising_edge (clock));
     if (com='1')
        then        -- mul 32bits
             -- shift and add algorithm
             -- ADVAL := A;
             -- RES := 0;
             -- For i:=0 to 30 do
             --                 If (B[0] = 1)
             --                    Then RES := RES + ADDVAL
             --                 End if;
             --                 ADVAL := shiftleft (ADVAL);
             --                 B := shiftright (B);
             -- End do;
             --

             X := 0;
             RES := 0;
             outdone <= '0';
             outval <= 0;
             A := input1;
             B := input2;

             wait until rising_edge (clock);
             if (B < 0)
             then
                  negation (A, A);
                  negation (B, B);
             end if;

             resize(A, ADVAL);  -- RESIZE
             resize(B, B_32bit);  -- RESIZE

             while (X /= 16) loop
                 zerobit(B_32bit, bit0);
                 if (bit0 = '1')
                    then RES := RES + ADVAL;
                 end if;
                 shiftleft(ADVAL, ADVAL);
                 shiftright(B_32bit, B_32bit);
                 X := X + 1;
             end loop;
             outdone <= '1';

        else -- result 32bits
             outval <= RES;
     end if;
```

```
    end process;


  Inst_FU_shiftrt : FU_shiftrt
  port map (in1     => shiftrt_in1,
            out1    => shiftrt_out);

  Inst_FU_shiftlt : FU_shiftlt
  port map (in1     => shiftlt_in1,
            out1    => shiftlt_out);

  Inst_FU_zerobit : FU_zerobit
  port map (in1     => zerobit_in1,
            out1    => zerobit_out1);

  Inst_FU_resize : FU_resize
  port map (in1     => resize_in1,
            out1    => resize_out1);

  Inst_FU_negation : FU_negation
  port map (in1     => negation_in1,
            out1    => negation_out1);

end behavior;

configuration mult_cfg of mult is
   for behavior
   end for;
end mult_cfg;
```

Annexe F

Description initiale VHDL du circuit Tbunit

```
library IEEE;

use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
use IEEE.std_logic_arith.all;
use work.types.all;

-----------------------------------------------------------
entity Timeunit  is

port ( RST,CLK      :  in    std_logic;
       SOC          :  in    std_logic;
       TB_cout      :  out   integer_13b;
       ADTF         :  out   integer_10b;
       Trm          :  out   integer_8b);

end Timeunit;
-------------------------------------------------------------

architecture behavior of Timeunit  is

begin

   TIME_BASE : process

        Variable CMT1    : integer_13b;
        Variable val_mask: integer_13b;
        Variable CMT2    : integer_8b;
        Variable fori    : integer;
        Variable CMT3    : integer_10b;

-------------------------------------------------------------

   Function add13(I : integer_13b) Return integer_13b is

    begin
        return (I + 1);
   end add13;
-------------------------------------------------------------
   Function add8(I : integer_8b) Return integer_8b is
```

```
    begin
        return (I + 1);
   end add8;

-------------------------------------------------------
   Function add10(I : integer_10b) Return integer_10b is

    begin
        return (I + 1);
   end add10;

-------------------------------------------------------

function mask (value, mask : in integer_13b) return integer_13b is

        variable i, m, iv, v : integer_13b;
        begin
                iv := value;
                m := mask;
                v := 0;
                for i in 0 to 12 loop
                    if m /= (m / 2)*2
                       then
                            if iv /= (iv /2)*2
                               then
                                   v := v + 2**i;
                               end if;
                       end if;
                    iv := iv / 2;
                    m := m / 2;
end loop;
return v;
        end mask;
-------------------------------------------------------

--procedure affect(in1 : in integer_13b; in2  : in integer_10b; in3 : in --integer_8b) is
--    begin
--        TB_cout <= in1;
--        ADTF    <= in2;
--        Trm     <= in3;
--        wait until rising_edge(clk);
--      end affect;

 -------------------------------------------------------

 begin
        CMT1 := 0;
        CMT2 := 0;
        CMT3 := 0 ;
        WAIT UNTIL SOC = '1';

    loop
        fori:=0;
        loop_count : loop
          wait until rising_edge(clk);
            if fori > 105 then
               exit loop_count;
            else fori := fori + 1;
```

```
            end if;
         end loop loop_count;

         CMT1 := add13(CMT1);
         val_mask := mask (CMT1, 16#00FF#);

         if val_mask = 16#00FF#  then
            CMT2 := add8(CMT2);
         end if;

         val_mask := mask (CMT1, 16#0FFF#);
         if val_mask = 16#0FFF# then
            CMT3 := add10(CMT3);
         end if;

         TB_cout <= CMT1;
         ADTF    <= CMT2;
         Trm     <= CMT3;
--       wait until rising_edge(clk);

     end loop;
   end process TIME_BASE;

end behavior;

configuration cfg_Timeunit of Timeunit is
   for behavior
       end for;
end cfg_Timeunit;
```

Annexe G

Description initiale VHDL du filtre elliptique (Ellip)

```
--------------------------------------------------------------------------
--
--
--                   Elliptical Wave Filter Benchmark
--
--
-- VHDL Benchmark author: D. Sreenivasa Rao
--                        University Of California, Irvine, CA 92717
--                        dsr@balboa.eng.uci.edu, (714)856-5106
--
-- Developed on 12 September, 1992
--
-- Verification Information:
--
--                  Verified     By whom?           Date         Simulator
--                  --------   ------------        --------     ------------
--  Syntax            yes         DSR              09/12/92       ZYCAD
--  Functionality     yes         DSR              09/12/92       ZYCAD
--------------------------------------------------------------------------

--use std.std_logic.all;
--use work.bit_functions.all;
use work.amical.all;

library IEEE;

use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
use IEEE.std_logic_arith.all;

entity ellipf is
    port ( reset  : IN     bit;                 -- Global reset
           clk    : IN     bit;                 -- Global clock
           inp : in std_logic_vector(15 downto 0);
           outp : out std_logic_vector(15 downto 0);
           sv2, sv13, sv18, sv26, sv33, sv38, sv39 :
                 in std_logic_vector(15 downto 0);
           sv2_o, sv13_o, sv18_o, sv26_o, sv33_o, sv38_o, sv39_o :
                 out std_logic_vector(15 downto 0));
 Attribute PortType Of clk   : Signal Is Port_Clock;
 Attribute PortType Of reset : Signal Is Port_Reset;
```

```
end ellipf;

architecture ellipf of ellipf is

begin

--process (inp, sv2, sv13, sv18, sv26, sv33, sv38, sv39)
elf : process
    constant m1 : std_logic_vector(15 downto 0) := "0100100101010101";
    constant m2 : std_logic_vector(15 downto 0) := "1001010000100101";
    constant m3 : std_logic_vector(15 downto 0) := "0010010001000001";
    constant m4 : std_logic_vector(15 downto 0) := "1011010001000101";
    constant m5 : std_logic_vector(15 downto 0) := "0010001000000011";
    constant m6 : std_logic_vector(15 downto 0) := "1001000011000001";
    constant m7 : std_logic_vector(15 downto 0) := "0001001001000101";
    constant m8 : std_logic_vector(15 downto 0) := "1001001001010001";
    variable n1, n2, n3, n4, n5, n6, n7 : std_logic_vector(15 downto 0);
    variable n8, n9, n10, n11, n12, n13 : std_logic_vector(15 downto 0);
    variable n14, n15, n16, n17, n18, n19 : std_logic_vector(15 downto 0);
    variable n20, n21, n22, n23, n24, n25 : std_logic_vector(15 downto 0);
    variable n26, n27, n28, n29 : std_logic_vector(15 downto 0);
--    constant i : integer := (1);

 begin
--  while (i = 1) LOOP
    n1 := inp + sv2;
    n2 := sv33 + sv39;
    n3 := n1 + sv13;
    n4 := n3 + sv26;
    n5 := n4 + n2;
    n6 := n5 * m1;
    n7 := n5 * m2;
    n8 := n3 + n6;
    n9 := n7 + n2;
    n10 := n3 + n8;
    n11 := n8 + n5;
    n12 := n2 + n9;
    n13 := n10 * m3;
    n14 := n12 * m4;
    n15 := n1 + n13;
    n16 := n14 + sv39;
    n17 := n1 + n15;
    n18 := n15 + n8;
    n19 := n9 + n16;
    n20 := n16 + sv39;
    n21 := n17 * m5;
    n22 := n18 + sv18;
    n23 := sv38 + n19;
    n24 := n20 * m6;
    n25 := inp + n21;
    n26 := n22 * m7;
    n27 := n23 * m8;
    n28 := n26 + sv18;
    n29 := n27 + sv38;
    sv2_o <= n25 + n15;
    sv13_o <= n17 + n28;
    sv18_o <= n28;
    sv26_o <= n9 + n11;
```

```
    sv38_o <= n29;
    sv33_o <= n19 + n29;
    sv39_o <= n16 + n24;
    outp <= n24;
    wait until (not clock'stable) and (clock='1');

--   end LOOP;
 end process elf;

end ellipf;

--configuration ellipcon of ellipf is
--  for ellip_beh
--  end for;
--end ellipcon;
```

Annexe H

Description initiale VHDL du circuit QRS

H.1 Entity

```
--**VHDL*************************************************************************
-- SRC-MODULE : QRS
-- NAME       : qrs.vhdl
-- VERSION    : 1.1
--
-- PURPOSE    : Entity of QRS Chip
--
-- LAST UPDATE: Thu Feb 11 09:44:45 1993
--
-- Verification Information:
--
--                  Verified     By whom?           Date         Simulator
--                  --------   ------------        --------     ------------
--  Syntax            yes     Preeti R. Panda      17 Jan 95    Synopsys
--  Functionality     yes     Manu Gulati          01 Dec 93    Synopsys
--**********************************************************************
--
-- Entity of QRS
--
USE work.qrs_types.all;
use work.amical.all;
library IEEE;
use IEEE.STD_LOGIC_1164.all;

ENTITY qrs IS
PORT (reset  : IN     bit;                              -- Global reset
      clk    : IN     bit;                              -- Global clock
      data   : IN     std_logic_vector(15 DOWNTO 0);    -- Data bus (input only, 16 pins)
      we     : IN     bit;              -- Write-Enable, indicating valid data on Data 15-0
      rc     : IN     bit;                              -- Restart Command
      rdy    : OUT    bit;                              -- Ready to read data
      fl3o   : OUT    std_logic_vector(15 DOWNTO 0);
      RRpeak : OUT    bit;                              -- Peak signal
      RRo    : OUT    std_logic_vector(15 DOWNTO 0));   -- Number of cycles between peaks
Attribute PortType Of clk   : Signal Is Port_Clock;
Attribute PortType Of reset : Signal Is Port_Reset;

END qrs;
```

H.2 Architecture

```
--**VHDL*************************************************************************
--
-- SRC-MODULE : QRS
-- NAME       : qrs_sys.vhdl
-- VERSION    : 1.1
--
-- PURPOSE    : Architecture of QRS Chip (system description)
--
-- LAST UPDATE: Thu Feb 11 19:51:45 1993
--
-- Verification Information:
--
--                  Verified     By whom?           Date         Simulator
--                  --------   ------------       --------     ------------
--  Syntax            yes     Preeti R. Panda     17 Jan 95    Synopsys
--  Functionality     yes     Manu Gulati         01 Dec 93    Synopsys
--**********************************************************************************
--
-- Architecture of QRS
--

USE work.qrs_types.all;
use work.amical.all;

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
--use IEEE.STD_LOGIC_signed.all;
use IEEE.std_logic_arith.all;

ARCHITECTURE system OF qrs IS
subtype int16 is std_logic_vector (15 downto 0);
BEGIN
  qrs: PROCESS

  CONSTANT    ACTIVE           : bit := '0';
  CONSTANT    INACTIVE         : bit := '1';
  CONSTANT    zero        : std_logic_vector(15 DOWNTO 0) := "0000000000000000";
  CONSTANT    one        : std_logic_vector(15 DOWNTO 0) := "0000000000000001";

    VARIABLE   ft, ftm1, ftm2, ecgm1, ecg1, ysi    : std_logic_vector(15 DOWNTO 0);
    VARIABLE   ymax,xmax,y0,ath,ys,y0m2,zmax,y0m1  : std_logic_vector(15 DOWNTO 0);
    VARIABLE   sth1,sth2,lxmax,lymax,lzmax         : std_logic_vector(15 DOWNTO 0);
    VARIABLE   low,high                            : std_logic_vector(15 DOWNTO 0);
    VARIABLE   i, cnt, indx, RR, v, tmp   : std_logic_vector(15 DOWNTO 0);
    VARIABLE   fl3                                 : std_logic_vector(15 DOWNTO 0);
    VARIABLE   fl1, fl2                            : bit;

  -- make division by shifting rigth 'in2' positions
  procedure shfdiv (a : in int16; b : in std_logic_vector(3 DOWNTO 0); v: out int16) is
  -- function shfdiv (in1 : in int16; in2 : in integer) return int16 is
   BEGIN
   if (b = std_logic_vector'("0001")) then v := "0" & a(15 downto 1);
   elsif (b = std_logic_vector'("0010")) then v := "00" & a(15 downto 2);
   elsif (b = std_logic_vector'("0011")) then v := "000" & a(15 downto 3);
   elsif (b = std_logic_vector'("0100")) then v := "0000" & a(15 downto 4);
```

```
   elsif (b = std_logic_vector'("1000")) then v := "00000000" & a(15 downto 8);
   else v := "0000000000000000";
   end if;
   end shfdiv;
--  Attribute FunctionType of shfdiv : Function Is Ordinary;
   Attribute ProcedureType of shfdiv : Procedure Is Ordinary;

  begin
    rdy    <= Inactive;
    RRpeak <= Inactive;
    fl3o   <= zero;
    RRo    <= zero;
    y0m1   := zero;
    y0m2   := zero;
    ymax   := zero;
    xmax   := zero;
    zmax   := zero;
    RR     := zero;
    lymax  := zero;
    lzmax  := zero;
    lxmax  := zero;
    fl3    := zero;
    fl1    := '0';
    fl2    := '0';
    cnt  := zero;

    WAIT until we = INACTIVE;               --   wait until sender is free
    rdy <= Active;                          --   announce ready to read
    WAIT until we = ACTIVE;                 --   wait until sender has sent
    low := data;                            --   read data
    rdy <= Inactive;                        --   disable ready to read

    WAIT until we = INACTIVE;               --   wait until sender is free
    rdy <= Active;                          --   announce ready to read
    WAIT until we = ACTIVE;                 --   wait until sender has sent
    high := data;                           --   read data
    rdy  <= Inactive;                       --   disable ready to read

    WAIT until we = INACTIVE;               --   wait until sender is free
    rdy <= Active;                          --   announce ready to read
    WAIT until we = ACTIVE;                 --   wait until sender has sent
    indx := data;                           --   read data
    rdy  <= Inactive;                       --   disable ready to read

    WAIT until we = INACTIVE;               --   wait until sender is free
    rdy <= Active;                          --   announce ready to read
    WAIT until we = ACTIVE;                 --   wait until sender has sent
    ftm2 := data;                           --   read data
    rdy  <= Inactive;                       --   disable ready to read

    WAIT until we = INACTIVE;               --   wait until sender is free
    rdy <= Active;                          --   announce ready to read
    WAIT until we = ACTIVE;                 --   wait until sender has sent
    ftm1  := data;                          --   read data
    ecgm1 := ftm1;

--    init: FOR i IN 1 TO indx LOOP           --   initialization loop
    i := one;
```

```
init :    while (i <= indx) loop

        rdy <= Inactive;                    --   disable ready to read
        WAIT until we = INACTIVE;           --   wait until sender is free
        rdy <= Active;                      --   announce ready to read
        WAIT until we = ACTIVE;             --   wait until sender has sent
        ecg1 := data;                       --   read data

        --   ft := ftm1 + 0.9965*(ecg1 - ecgm1)
tmp := (ecg1 - ecgm1);
shfdiv(tmp,"1000",v);
        ft  := ftm1 + tmp - v;
        ysi := ft - ftm2;
IF (ysi > ymax) THEN ymax := ysi; END IF;
IF ( ft > xmax) THEN xmax := ft;  END IF;
       IF (ft > 0) THEN y0 := ft;  END IF;
IF (ft < 0) THEN y0 := zero-ft; END IF;
shfdiv(xmax,"0010",v);
ath := v;
IF ( ath > y0) THEN y0 := ath; END IF;
ys := y0 - y0m2;
IF (ys > zmax) THEN zmax := ys; END IF;
ftm2  := ftm1;
ftm1  := ft;
ecgm1 := ecg1;
y0m2  := y0m1;
y0m1  := y0;
        --   sth1 = ymax * 0.6875;
shfdiv(ymax,"0001",v);
        sth1  :=  v;
        shfdiv(ymax,"0011",v);
sth1 := sth1 + v;
        shfdiv(ymax,"0100",v);
sth1 := sth1 + v;
        --   sth2 = zmax * 0.6875; */
shfdiv(zmax,"0001",v);
        sth2  := v;
        shfdiv(zmax,"0011",v);
        sth2  := sth2 + v;
   shfdiv(zmax,"0100",v);
        sth2  := sth2 + v;
    END LOOP init;

    regular : LOOP

        IF (ysi > sth1) THEN
    fl1   := '1';
    cnt := zero;
        END IF;
        IF (ys > sth2) THEN
            fl2   := '1';
            cnt := zero;
        END IF;
IF ((fl1 = '1') AND (fl2 = '1') AND (RR > low)) THEN
            RRpeak <= Active;
--            xmax  := shfdiv(xmax,1) + shfdiv(xmax,2) + shfdiv(xmax,3) + shfdiv(lxmax,3);
    shfdiv(xmax,"0001",v);
    xmax := v;
```

```
    shfdiv(xmax,"0010",v);
    xmax := xmax + v;
            shfdiv(xmax,"0011",v);
    xmax := xmax + v;
            shfdiv(lxmax,"0011",v);
    xmax := xmax + v;

--              ymax  := shfdiv(ymax,1) + shfdiv(ymax,2) + shfdiv(ymax,3) + shfdiv(lymax,3);
    shfdiv(ymax,"0001",v);
    ymax := v;
    shfdiv(ymax,"0010",v);
    ymax := ymax + v;
            shfdiv(ymax,"0011",v);
    ymax := ymax + v;
            shfdiv(lymax,"0011",v);
    ymax := ymax + v;

--              zmax  := shfdiv(zmax,1) + shfdiv(zmax,2) + shfdiv(zmax,3) + shfdiv(lzmax,3);
    shfdiv(zmax,"0001",v);
    zmax := v;
    shfdiv(zmax,"0010",v);
    zmax := zmax + v;
            shfdiv(zmax,"0011",v);
    zmax := zmax + v;
            shfdiv(lzmax,"0011",v);
    zmax := zmax + v;

            RR    := zero;
            cnt := zero;
            fl1   := '0';
            fl2   := '0';
            fl3   := zero;
            lxmax := zero;
            lymax := zero;
            lzmax := zero;
        ELSE
            RRpeak <= Inactive;
        END IF;

        IF ((fl1 = '1') OR  (fl2 = '1')) THEN
            cnt := cnt + one;
        END IF;

        fl3o  <= fl3;
        RRo   <= RR;

        rdy <= Inactive;                    --   disable ready to read
        WAIT until we = INACTIVE;           --   wait until sender is free
        rdy <= Active;                      --   announce ready to read
        WAIT until we = ACTIVE;             --   wait until sender has sent
        ecg1 := data;                       --   read data

        --ft  := ftm1 + 0.9965*(ecg1 - ecgm1)
        tmp := (ecg1 - ecgm1);
        shfdiv(tmp,"1000",v);
        ft  := ftm1 + tmp - v;
        ysi := ft - ftm2;
        IF (ysi > lymax) THEN lymax := ysi; END IF;
```

```
        IF ( ft > lxmax) THEN lxmax := ft;  END IF;
        IF (ft > 0) THEN y0 := ft;  END IF;
        IF (ft < 0) THEN y0 := zero-ft; END IF;
        shfdiv(xmax,"0010",v);
        ath := v;
        IF (y0 < ath) THEN y0 := ath; END IF;
        ys := y0 - y0m2;
        IF (ys > lzmax) THEN lzmax := ys; END IF;
        IF (cnt = 8) THEN
            fl1   := '0';
            fl2   := '0';
            cnt := zero;
        END IF;
        IF (RR > high) THEN
            fl3  := fl3 + one;
            RR   := zero;
            shfdiv(ymax,"0001",v);
            ymax := v;
            shfdiv(zmax,"0001",v);
            zmax := v;
        END IF;
--      sth1 := ymax *0.6875
--        sth1 := shfdiv(ymax,1) + shfdiv(ymax,3) + shfdiv(ymax,4);
    shfdiv(ymax,"0001",v);
    sth1 := v;
        shfdiv(ymax,"0011",v);
    sth1 := sth1 + v;
        shfdiv(ymax,"0100",v);
    sth1 := sth1 + v;

--      sth2 := zmax * 0.6875
--        sth2 := shfdiv(zmax,2) + shfdiv(zmax,3) + shfdiv(zmax,4);
    shfdiv(zmax,"0010",v);
    sth2 := v;
        shfdiv(zmax,"0011",v);
    sth2 := sth2 + v;
        shfdiv(zmax,"0100",v);
    sth2 := sth2 + v;

        RR   := RR + one;
        ecgm1 := ecg1;
        y0m2  := y0m1;
        y0m1  := y0;
        ftm2  := ftm1;
        ftm1  := ft;

    END LOOP regular;

  END PROCESS qrs;

END system;
--
-- End of Architecture
--
```

Bibliographie

[ABOR90] R. Airiau, J.-M. Bergé, V. Olive, and J. Rouillard. *VHDL du langage à la modélisation*. Presses polytechniques et universitaires romandes et CNET-ENST, 1990.

[ABRM98] Pranav Ashar, Subhrajit Bhattacharya, Anand Raghunathan, and Akira Mukaiyama. Verification of RTL Generated from Scheduled Behavior in a High-Level Synthesis Flow. In *IEEE/ACM International Conference on Computer Aided Design (ICCAD'98)*, pages 517–524, San Jose, Ca, 1998.

[Ard96a] Laurent Arditi. *BMD Can Delay the Use of Theorem Proving for Verifying Arithmetic Assembly Instructions. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD'96)*, volume 1166 of *LNCS*, pages 34–48, Pala Alto, CA, USA, November 1996. Springer.

[Ard96b] Laurent Arditi. *Spécification et Preuves des Microprocesseurs*. PhD thesis, L'Université de Nice – Sophia Antipolis, 1996.

[Bar84] Harry G. Barrow. VERIFY: A Program for Proving Correctness of Digital Hardware Designs. In *Artificial Intelligence*, volume 24, pages 437–491, 1984.

[Bar94] Samary Baranov. *Logic Synthesis for Control Automata*. Kluwer Academic Publishers, Dordrecht/Boston/London, 1994.

[Baw96] Rajesh K. Bawa. *Un environnement intégré pour la vérification formelle et l'analyse des systèmes décrits en VHDL*. PhD thesis, L'Université Pierre et Marie Curie, Paris VI, 1996.

[BC95] Randal E. Bryant and Yirng-An Chen. Verification of Arithmetic Circuits with Binary Moment Diagrams. In *Proceedings of 32nd Design Automation Conference (DAC'95)*, pages 535–541, San Francisco, CA, USA, 1995.

[BDP98] Dominique Borrione, Julia Dushina, and Laurence Pierre. Formalization ot Finite State Machines with Data Path for the Verification of High-Level Synthesis. In *XI Brazilian Symposium on Integrated Circuit Design (SBCCI'98)*, Buzios, Rio de Janeiro, Brazil, September 30 to October 3 1998. IEEE Computer Society.

[BE97] Christian Blumenrohr and Dirk Eisenbiegler. An Efficient Representation for Formal Synthesis. In *10th International Symposium on System Level Synthesis*, pages 9–15, Antwerp, Belgium, September 1997 1997. IEEE.

[BE98] Christian Blumenrohr and Dirk Eisenbiegler. Deriving Structural RT-Implementation from Algorithmic Descriptions by means of Logical Transformations. In *GI/ITG/GMM Verification Workshop*, pages 38–49, Paderborn, Germany, March, 9–11 1998.

[BFK94] Peter T. Breuer, Luis Sanchez Fernandez, and Carlos Delgado Kloos. Proof Theory and a Validation Condition Generator for VHDL. In *European Design Automation Conference with EURO-VHDL'94*, volume 7, pages 512–517, Grenoble, France, September, 19-23 1994.

[BFK95] Peter T. Breuer, Luis Sanchez Fernandez, and Carlos Delgado Kloos. A Simple Denotational Semantics, Proof Theory and a Validation Condition Generator for Unit-Delay VHDL. In *Formal Methods in System Design*, volume 7, Number 1/2, pages 27–51, August 1995.

[BHY92] B.C. Brock, W.A. Hunt, and W.D. Young. Introduction to a formally defined hardware description language. In *Proc. of the International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 3–35, Nijmegen, June 1992. IFIP TC10/WG 10.2, North-Holland.

[BKK+96] Peter Borovansky, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Marian Vettek. ELAN: A logical framework based on computational systems. In *Electronic Notes in Theoretical Computer Science*, volume 5, New York, 1996. Elsevier Science Publishers B.V.

[BKV+96] E. Berrebi, P. Kission, S. Vernalde, J.C. Herluison, S. de Troch, J. Frehel, A.A. Jerraya, and I. Bolsens. Combined Control Flow Dominated and Data Flow Dominated High-Level Synthesis. In *33rd ACM/IEEE Design Automation Conference*. ACM/IEEE, 1996.

[BM97] R.S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press Inc, second edition, 1997.

[Boo67] Taylor L. Booth. *Sequential Machines and Automata Theory*. John Wiley and Sons Inc., NewYork-London-Sydney, 1967.

[BR96] Reinaldo A. Bergamaschi and Salil Raje. Observable Time Windows: Verifying the Results of High-Level Synthesis. In *European Design & Test Conference (ED&TC'96)*, pages 350–356. IEEE, 1996.

[BR97] Reinaldo A. Bergamaschi and Salil Raje. Observable Time Windows: Verifying High-Level Synthesis Results. In *IEEE Design & Test on Computers*, volume April-June, pages 40–50. IEEE, 1997.

[Bry95] Randal E. Bryant. Binary Decision Diagrams and Beyond: Enabling Technologies for Formal Verification. In *Proceedings of ICCAD'95*, pages 236–243, 1995.

[BS95] Dominique Borrione and Ashraf Salem. Denotational semantics of a synchronous vhdl subset. In *Formal Methods in System Design*, volume 7, Number 1/2, pages 53–71, August 1995.

[Cam96] R. Camposano. Behavioral Synthesis. In *Proceedings of 33rd Design Automation Conference (DAC'96)*, Las Vegas, NV, USA, June 1996.

[CKZ96] E.M. Clarke, M. Khaira, and X. Zhao. Word Level Model Checking- Avoiding the Pentium FDIV Error. In *Proceedings of 33rd Design Automation Conference (DAC'96)*, pages 645–648, Las Vegas, NV, USA, 1996.

[CP88] P. Camurati and P. Prinetto. Formal verification of hardware correctness: Introduction and survey of current reseach. In *Computer (Journal)*, volume 21(7), pages 9–19, July 1988.

[CT93] Michel Cosnard and Denis Trystram. *Algorithmes et architectures paralleles*. Informatique intelligence artificielle. InterEditions, Paris, 1993.

[CW96] Edmund M. Clarke and Jeannette M. Wing. Formal Methods: State of the Art and Future. Technical report, Carnegi Mellon University, 1996. Report CMU-CS-96-178.

[D96] David Déharbe. *Vérification Formelle de Proprietés Temporelles: Études et Application au Langage VHDL*. PhD thesis, L'Universite' Joseph Fourier, 1996.

[DB95a] David Déharbe and Dominique Borrione. Semantics of a Verification-Oriented Subset of VHDL. In Paolo E. Camurati and Hans Eveking, editors, *Lecture Notes in Computer Scienece*, volume 987, pages 293–310. IFIP WG10.5, Springer, 1995.

[DB95b] David Déharbe and Dominique Borrione. Symbolic model checking of VHDL design entities. In *Current Issues in Electronic Modeling*, volume 1. Kluwer Academic Publisher, 1995.

[DB97] Julia Dushina and Dominique Borrione. Formalisation and Validation of the Std_Logic_1164 and Numeric_Std VHDL Packages using the Nqthm Theorem Prover. In *2nd Workshop on Libraries, Component Modeling, and Quality Assurance*, Toledo, Spain, April, 23-25 1997. SIG-VHDL, In coop. with IFIP WG 10.5.

[DBJ96] Julia Dushina, Dominique Borrione, and Ahmed Jerraya. Correct Reuse of Complex Design Units During High Level Synthesis: Verification Issues. In *1-st IEEE International High Level Design Validation and Test Workshop (HLDVT)*, Spa, Oakland, California, USA, November, 15-16 1996. IEEE.

[DBJ98] Julia Dushina, Dominique Borrione, and Ahmed Jerraya. Formal Verifcation of the Allocation Step in High Level Synthesis. In *Forum on Design Languages*, Lausanne, Switzerland, September, 6-11 1998.

[Din96] Hong Ding. *La Synthèse Architecturale Interactive et Flexible*. PhD thesis, l'Institut National Politechnique de Grenoble, 1996.

[Dus97] Julia Dushina. Un modèle formel pour la synthèse de haut niveau utilisant des composants complexes. In *Colloque CAO de circuits intégrés et systèmes*, pages 98–101, Grenoble (Villard de Lans), January, 15-17 1997.

[EHR99] Hans Eveking, Holger Hinrichsen, and Gerd Ritter. Automatic Verification of Scheduling Results in High-Level Synthesis. In *Proceedings of Design, Automation and Test in Europe Conference (DATE'99)*, pages 59–64, 1999.

[EKB97] Dirk Eisenbiegler, Ramayya Kumar, and Christian Blumenrohr. A Constructive Approache towards Correctness of Synthesis-Application within Retiming. In *The European Design & Test Conference*, pages 427–432, Paris, France, March 1997. IEEE Computer Society and ACM/SIGDA, IEEE Computer Society Press.

[Enc95] Emmanuelle Encrenaz. A Symbolic Relation for a Subset of VHDL'87 Descriptions and its Application to Symbolic Model Checking. In Paolo E. Camurati and Hnas Eveking, editors, *Lecture Notes in Computer Scienece*, volume 987, pages 328–342. IFIP WG10.5, Springer, 1995.

[ESV+98] A. Evans, A. Silburt, G. Vrckovnik, T. Brown, M. Dufresne, G. Hall, T. Ho, and Y. Liu. Functional Verification of Large ASICs. In *35rd ACM/IEEE Design Automation Conference*, pages 650–655, San Francisco, CA, June 15–19 1998. ACM/IEEE.

[Eve99] Hans Eveking. Machine Assisted Verification (Draft version). 1999.

[GDWL92] D. Gajski, N. Dutt, A. Wu, and Y. Lin. *High Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, Boston, Massachusetts, 1992.

[Gor88] Michael J. C. Gordon. *Programming language theory and its implementation: applicative and imperative paradigms*. Prentice-Hall International series in computer science. Prentice-Hall, New-York/London/Toronto, 1988.

[GR94] Daniel D. Gajski and Loganath Ramachandran. Introduction to High-Level Synthesis. In *IEEE Design & Test of Computers*, volume Winter, pages 44–54, 1994.

[Gup92] Aarti Gupta. Formal Hardware Verification Methods: A Survey. In *Formal methods in System Design*, volume 1, 2/3, October 1992.

[HAFM97] Yatin V. Hoskote, Jacob A. Abraham, Donald S. Fussell, and John Moondanos. Automatic Verification of Implementation of Large Circuits Against HDL Specification. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 16, No.3, pages 217–227. IEEE, March 1997.

[Har98] John Harrison. *Introduction to Functional Programming*. Copy of slides, http://www.cl.cam.ac.uk/Teaching/Lectures/funprog-jrh/, 1997-98.

[Hen68] Frederic C. Hennie. *Finite-State Models for Logical Machines*. John Wiley and Sons Inc., NewYork-London-Sydney, 1968.

[HKLR92] Jieh Hsiang, Hélène Kirchner, Pierre Lescanne, and Michaël Rusinowitch. The Term Rewriting Approach to Automated Theorem Proving. In *THE JOURNAL OF LOGIC PROGRAMMING*, pages 71–99, New York, 1992. Elsevier Science Publishers Co.

[HS66] J. Hartmanis and R.E. Stearns. *Algebraic Structure Theory of Sequential Machines.* Prentice-Hall, Inc., 1966.

[HV92] Thérèse Accart Hardin and Véronique Donzeau-Couge Viguié. *CONCEPTS ET OUTILS DE PROGRAMMATION. Le style fonctionnel, le style impératif avec Caml et Ada.* InterEditions, Paris, 1992.

[IEE93] IEEE Standards Board. *IEEE Std 1076-1993 VHDL Language Reference Manual.* The Institute of Electrical and Electronics Engineers, Inc, New York, USA, September 15 1993.

[IEE98] IEEE VHDL Synthesis Interoperability Working Group. *IEEE P1076.6/D2.0 Draft Standard For VHDL Register Transfer Level Synthesis.* WWW access://vhdl.org/vi/vhdlsynth/vhdlsynth.html, 1998.

[JDKR97] A. A. Jerraya, H. Ding, P. Kission, and M. Rahmouni. *Behavioral Synthesis And Component Reuse With VHDL.* Kluwer Academic Publishers, Borton/London/Dordrecht, 1997.

[JPO93] A. A. Jerraya, I. Park, and K. O'Brien. AMICAL : An Interactive High Level Synthesis Environement. In *Proceedings on The European Conference on Design Automation (EDAC)*, February 1993.

[KBES96] Ramayya Kumar, Christian Blumenrohr, Dirk Eisenbiegler, and Detlef Schmid. Formal Synthesis in Circuit Design- A Survey. In M.Srivas and A.Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD'96)*, volume 1166 of *LNCS*, pages 294–309, Paolo Alto, CA, USA, November 1996. IEEE.

[KDJ94] P. Kission, H. Ding, and A.A. Jerraya. Structured Design Methodology for High-Level Design. In *31st ACM/IEEE Design Automation Conference.* ACM/IEEE, 1994.

[KDJ95] P. Kission, H. Ding, and A.A. Jerraya. VHDL Based Design Methodology for Hierarchy and Component Re-Use. In *EuroDAC-EuroVHDL*, 1995.

[Kis96] Polen Kission. *Exploitation de la Hiérarchie et de la Réutilisation de Blocs Existants par la Synthèse de Haut Niveau.* PhD thesis, l'Institut National Politechnique de Grenoble, 1996.

[Kna96] David W. Knapp. *Behavioral Synthesis. Digital System Design Using the Synopsys Behavioral Compiler.* Prentice Hall PTR, New Jersey, 1996.

[Koh78] Zvi Kohavi. *Switching and Finite Automata Theory.* Tata McGraw-Hill: Computer Science Series, New Delhi, 1978.

[Liv78] C. Livercy. *Théorie des programmes (schémas, preuves, sémantique)*. BORDAS, Paris, 1978.

[Mel87] T. Melham. Abstraction Mechanisms for Hardware Verification. In Birtwhistle and Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 267–291, Calgary, Canada, January 1987. Kluwer Academic Publisheres.

[Mil80] R. Milner. A Calculus of Communicating Systems. In *Lecture Notes in Computer Science (LNCS)*, volume 92, New York, 1980. Springer-Verlag.

[MPC90] M.C. McFarland, A.C. Parker, and R. Camposano. The High-Level Synthesis of Digital Systems. In *Proceedings of the IEEE*, volume 79, No 2, pages 301–317, February 1990.

[MV98] Nazanin Mansouri and Ranga Vemuri. A methodology for Automated Verification of Synthesized RTL designs and Its Integration with a High-Level Synthesis Tool. In *Proceedings on Second International Conference, FMCAD'98*, pages 204–221, Palo Alto, CA, USA, November 4–6 1998.

[Nic99] Felix Nicoli. *Vérification formelle de Descriptions VHDL Comportementales*. PhD thesis, Université de Provence, July 1999.

[NKV96] Naren Narasimhan, Ravi Kalyanaraman, and Ranga Vemuri. Validation of Synthesizid Register Transfer Level Designs Using Simulation and Formal Verification. In *High Level Design Validation and Test Workshop (HLDVT)*, November 1996.

[NRV96] Naren Narasimhan, Joy Roy, and Ranga Vemuri. Synchronous Controller Models for Synthesis from Communication VHDL Processes. In *International Conference on VLSI*, January 1996.

[NV96] Naren Narasimhan and Ranga Vemuri. Specification of Control Flow Properties for Verification of Synthesized VHDL Designs. In *Formal Methods in CAD (FMCAD)*, November 1996.

[PB] C. W. Parks and J. Burrus. *Digital Filter Theory and Design*.

[PBB92] M. Pilsl, S. Bhattacharya, and F. Brglez. QRS/BIST: A Reliable Cardiac Arrhythmia Monitor ASIC. In *Synthesizing Behavioral Benchmarks in VHDL into Standard Cell Layout*, California, USA, November 1992.

[Pie91] Laurence Pierre. One Aspect of Mechanizing Formal Proof of Hardware: the Generalization of Partial Specifications. In *Proc. ACM International Workshop on Formal Methods in VLSI Design*, Miami, January 1991.

[Pie95] Laurence Pierre. Describing and verifying synchronous circuits with the Boyer-Moore theorem prover. In Paolo E. Camurati and Hans Eveking, editors, *Correct Hardware Design and Verification Methods (CHARME'95)*, volume 987 of *LNCS*, pages 35–55, Frankfurt/Main, Germany, October 1995. Springer.

[Rah97] Maher Rahmouni. *Ordonnancement et Optimisations pour la Synthèse de Haut Niveau des Circuits de Contrôle.* PhD thesis, l'Institut National Politechnique de Grenoble, 1997.

[RE94] Simon Read and Martyn Edwards. A Formal Semantics of VHDL in Boyer-Moore Logic. In *Conference on Concurrent Engineering and EDA (CEEDA)*, Poole, Great Britain, 1994.

[Rea94] Simon Read. *Formal Methods for VLSI Design.* PhD thesis, The University of Manchester, 1994.

[Ree95] Ralf Reetz. Deep Embedding VHDL. In *International Workshop on Higher Order Logic Theorem Proving and its Applications*, Aspen Grove, Utah, USA, September 1995.

[RK95] Ralf Reetz and Thomas Kropf. A Flowgraph Semantics of VHDL: Toward a VHDL Verification Workbench in HOL. In *Formal Methods in System Design*, volume 7, pages 73–99. Kluwer Academic Publishers, August 1995.

[RNMK90] S.C. Roy, H.T. Nagle, M.G. McNamer, and W.T. Krakow. QRS/BIST: A Reliable Cardiac Arrhythmia Monitor ASIC. In *In Proceedings 3rd Annual IEEE ASIC Seminar and Exhibit*, September 1990.

[Roy90] S.C. Roy. A Digital QRS Detector and Arrhythmia Monitor. Master's thesis, University of North Carolina at Chapel Hill, 1990.

[Rus95] David M. Russinoff. A Formalization of a Subset of VHDL in the Boyer-Moore Logic. In *Formal Methods in System Design*, volume 7, pages 7–25, August 1995.

[Ska96] Kevin Skahill. *VHDL FOR PROGRAMMABLE LOGIC.* Addison Wesley Publishing, Inc (http://www.awl.com), 1996.

[WC91] R.A. Walker and R. Camposano. *A Survey of High-Level Synthesis Systems.* Kluwer Academic Publishers, Boston, Ma, 1991.

[WL93] Pierre Weis and Xavier Leroy. *Le langage Caml.* InterEditions, Paris, 1993.

[Yan98] C. Han Yang. Hardware Design Verifcation: New Frontiers. In *3rd International Conference on ASIC PROCEEDINGS*, Beijing, China, October 21–23 1998.

[Yoe90] Michael Yoeli. *Formal Verification of Hardware Design.* IEEE Computer Society Press, Los Alamitos, CA, 1990.

[ZB95] Zheng Zhou and Wayne Burleson. Equivalence Checking of Datapaths Based on Canonical Arithmetic Expressions. In *Design Automation Conference.* IEEE, 1995.