THÈSE DE DOCTORAT DE L'UNIVERSITÉ PARIS VI

SPÉCIALITÉ INFORMATIQUE

Présentée par Ludovic JACOMME
Pour obtenir le titre de
DOCTEUR DE L'UNIVERSITÉ PARIS VI

ANALYSE SÉMANTIQUE DE DESCRIPTIONS VHDL SYNCHRONES EN VUE DE LA SYNTHÈSE

Soutenue le 29 octobre 1999, devant le jury composé de

Mr Alain GREINER	Directeur de thèse
Mr Przemyslaw BAKOWSKI	Rapporteur
Mme Dominique BORRIONE	Rapporteur
Mr Alain FONKOUA	Examinateur
Mr Michel MINOUX	Examinateur
Mr Frédéric PÉTROT	Examinateur
Mr Patrice QUINTON	Examinateur

Résumé

Le langage de description de matériel VHDL a été initialement défini pour être utilisé dans le cadre de la simulation. Il s'est imposé depuis plus de dix ans comme un standard incontournable pour la spécification comportementale des circuits numériques.

Il a cependant rapidement été détourné de sa destination première pour être utilisé comme langage d'entrée de la synthèse au niveau transfert entre registres. La synthèse à partir d'une description VHDL est un véritable problème car ce langage possède une très forte sémantique de simulation.

Aussi, afin d'éviter de prendre en compte cette sémantique complexe lors de la phase d'analyse de la compilation, tous les outils de synthèse imposent l'utilisation de motifs syntaxiques particuliers pour identifier facilement les éléments matériels qui sont modélisés dans une description VHDL.

Outre le fait que ces motifs limitent fortement le style de description et restreignent la puissance de VHDL, ils remettent gravement en cause la portabilité du langage car ils diffèrent d'un outil de synthèse à l'autre.

Dans cette thèse nous proposons une méthode d'analyse diamétralement opposée. Elle s'appuie exclusivement sur la sémantique de simulation du langage pour identifier avec précision tous les éléments matériels nécessaire à l'implantation d'une description VHDL.

Un prototype logiciel s'appuyant sur cette méthode d'analyse sémantique a été implanté. Il nous a permis de montrer sur une centaines d'exemples l'efficacité de notre méthode autant d'un point de vue qualitatif que quantitatif.

Mots clés

Langage de description de matériel, VHDL, compilation, analyse sémantique, synthèse

Abstract

The VHDL hardware description language was initially defined to be used for simulation. It has been accepted as a standard for the behavioral specification of digital circuits for more than ten years.

Quite naturally VHDL was quickly used as an input language for synthesis applications. However, due to its strong simulation semantic, the use of VHDL for synthesis appears to be a difficult problem. To avoid taking into account the complex language semantic during the compilation step, all synthesis tools enforce the use of specific templates to identify hardware elements in VHDL descriptions.

Using such templates has clearly two drawbacks : they restrict the style of description and the power of the language, and they affect seriously the portability of the language because they depend on the synthesis environment used.

In this thesis we propose an opposite approach to the identification of the hardware elements. Our method relies exclusively on the simulation semantics and thus ensures post-synthesis results compliant with the VHDL simulation.

A software based on this semantic analysis method has been developed. It has enabled us to show the efficiency of our method as much from a qualitative and quantitative point of view.

Keywords

Hardware description language, VHDL, compilation, semantic analysis, hardware synthesis

À ma femme Daniela et à mon fils Alexandre.

Je tiens à exprimer ma reconnaissance à mon directeur de thèse Monsieur Alain Greiner professeur à l'université Paris VI, pour tous les précieux conseils qu'il m'a donnés et pour le temps qu'il m'a consacré lors de la rédaction de cette thèse.

Je remercie sincèrement Madame Dominique Borrione, professeur à l'université Joseph Fourier, et Monsieur Przemyslaw Bakowski, professeur à l'université de Nantes, d'avoir accepté d'être les rapporteurs de ma thèse de doctorat.

J'associe à ces remerciement Monsieur Michel Minoux professeur à l'université Paris VI, Monsieur Alain Fonkoua enseignant à l'école supérieure d'ingénieurs de Marseille et Monsieur Patrice Quinton professeur à l'institut de recherche en informatique et systèmes aléatoires de Rennes, pour avoir accepté de faire partie de mon jury de thèse.

Je remercie Monsieur Rajesh Bawa et Madame Emmanuelle Encrenaz de l'équipe algorithmique numérique pour m'avoir orienté sur le sujet de ma thèse, et pour m'avoir fait part de leurs remarques après relecture de mon manuscrit.

Je tiens à exprimer ma plus profonde gratitude à Monsieur Frédéric Pétrot pour les conseils, l'encadrement, le soutient et l'amitié qu'il a su me donner depuis mon arrivée au laboratoire, il y a près de dix ans.

Je n'oublie pas l'ensemble des membres du laboratoire LIP6, et plus particulièrement les membres de l'équipe ASIM, pour toute la gentillesse et la bonne humeur qu'ils m'ont témoignées.

Enfin je ne sais comment remercier ma femme Daniela d'avoir su me soutenir et me redonner courage dans les moments difficiles, en particulier durant la longue période de rédaction du présent manuscrit.

Table des matières

Liste des figures

Liste des tableaux

CHAPITRE 1

Introduction

La complexité croissante des circuits intégrés impose l'utilisation de langages de description de matériel de plus en plus haut niveau. Un grand nombre de travaux ont porté sur la recherche d'un langage unique pour décrire différents niveaux d'abstraction.

Deux d'entre eux ont réussi à s'imposer dans l'industrie : le langage Verilog créé par Phil Moore en 1983 et le langage VHDL initié par le DoD (département de la défense des États Unis d'Amérique) en 1981. Ils permettent de représenter le comportement d'un circuit durant presque toutes les étapes de la conception. En 1987 le langage VHDL est devenu le standard IEEE-1076[IEE87]. Le langage Verilog, placé dans le domaine public en 1990, a été accepté comme un standard en décembre 1995 [IEE95].

L'une des évolutions récentes du flot de conception est l'utilisation systématique des outils de synthèse : on écrit une description fonctionnelle de plus ou moins haut niveau dans un de ces langages, on valide cette description par simulation puis on effectue la synthèse. Le processus de synthèse est une transformation automatique d'un programme écrit dans un langage de description de matériel, en une interconnexion de portes numériques prises dans une bibliothèque prédéfinie. L'une des tâches initiales d'un outil de synthèse est de transposer chaque portion d'une description comportementale d'un circuit en un ensemble de fonctions matérielles connues de l'outil et équivalentes vis-à-vis de la simulation.

La synthèse à partir d'une description VHDL est un problème car ce langage a été initialement créé en vue de la simulation et non de la synthèse. La sémantique du langage est strictement liée à la simulation à tel point que la norme [IEE87, IEE93b] spécifie tous les aspects liés à l'implantation d'un simulateur. Le langage VHDL a donc été défini indépendamment des contraintes liées à la synthèse. On peut distinguer deux classes de problèmes.

Le premier problème est lié aux types de VHDL. Il existe des types d'objets, des instructions et des fonctions du langage qui trouvent leur utilité dans le cadre de la simulation mais qui ne peuvent pas être implantés de façon matérielle. Ces fonctionnalités offertes par VHDL doivent être interdites ou ignorées si l'on désire entrer dans un processus de synthèse. Cet aspect du problème à été résolu puisque les normes IEEE-1164 et IEEE-1076.3 [IEE93a, IEE97] définissent un ensemble de types et de fonctions adaptés au processus de synthèse.

Le second problème est lié à la puissance d'expression du langage. En effet VHDL permet de décrire de façon très diverse des éléments matériels aussi simples que des registres ou des portes trois états mais aussi des objets plus complexes tels que les automates d'états finis. L'analyse d'une description VHDL en vue d'identifier des éléments matériels qui doivent être synthétisés est un problème difficile et crucial.

Les outils industriels actuels sont tous capables de prendre une description VHDL respectant la norme [IEE87, IEE93b] et d'en effectuer la synthèse, mais chacun d'eux impose en plus des contraintes différentes sur le style de description à utiliser. Chaque outil de synthèse définit sa propre bijection entre certaines constructions syntaxiques du langage VHDL et certains éléments matériels tels que les registres. Une telle utilisation de VHDL par les outils de synthèse remet en question l'intérêt même de la standardisation du langage.

Ceci a bien sûr des répercutions directes sur la méthode de conception puisqu'il devient nécessaire de connaître *a priori*, l'outil de synthèse qui va être employé avant de pouvoir commencer à décrire son comportement en VHDL. La réutilisation de la totalité ou d'une partie d'un circuit est soumise aux mêmes contraintes, avec des conséquences qui peuvent s'avérer graves lorsqu'on sait qu'aujourd'hui certaines compagnies industrielles décident, du jour au lendemain, de ne plus développer tel ou tel outil de synthèse.

Deux approches peuvent être envisagée pour résoudre ce problème. Une approche possible est la définition non ambiguë d'un sous ensemble du langage comprenant des restrictions grammaticales et syntaxiques. Encore faut-il que le sous ensemble ainsi spécifié, soit accepté comme une norme par les sociétés qui développent des outils de synthèse VHDL. De grands efforts sont menés dans cette direction depuis plusieurs années [IEE99].

Une autre solution est la définition d'une méthode d'analyse formelle des descriptions VHDL, adaptée à la synthèse et fondée sur la sémantique de simulation du langage. Ceci permettrait d'une part d'accepter sans restriction les différents sous ensembles VHDL utilisés par les outils de synthèse existants, et d'autre part de ne plus imposer de fortes contraintes syntaxiques sur les descriptions à synthétiser.

Cette thèse s'attache à résoudre le problème de la reconnaissance d'éléments matériels tels les registres, les barrières trois états et les automates d'états finis dans une description VHDL et ce sans imposer de fortes contraintes sur le style de description. Nous définissons d'une part un modèle permettant de représenter la sémantique du langage VHDL. Nous présentons d'autre part différentes méthodes d'analyses de ce modèle afin de pouvoir effectuer la première étape de la synthèse. L'analyse d'une description VHDL permet la génération d'une description équivalente pouvant être traitée par tous les outils de synthèse actuels. Le résultat pratique de cette étude s'est traduit par le développement d'un prototype logiciel qui démontre l'intérêt et l'efficacité de cette méthode.

1.1 Plan du mémoire

Le chapitre 2 analyse les différentes phases du processus de synthèse à partir d'un langage de description de matériel. Il met en évidence l'ensemble des objectifs à atteindre lors de la synthèse et les problèmes spécifiques liés à l'utilisation du langage VHDL.

Le chapitre 3 expose de façon critique les différentes approches qui ont été proposées pour assurer que le résultat de la synthèse soit en accord avec la sémantique de simulation de VHDL.

Dans le chapitre 4 nous présentons un modèle préservant la sémantique de simulation du langage VHDL et adapté aux besoins de la synthèse.

Le chapitre 5 présente différentes techniques de réduction applicables sur ce modèle. Nous développons ensuite dans les chapitres 6 et 7 notre méthode de reconnaissance des éléments matériels sur ce modèle réduit.

Le chapitre 8 expose l'implantation de notre démonstrateur, et dans le chapitre 9 nous présentons les résultats expérimentaux que nous avons obtenus.

Enfin dans le chapitre 10 nous dressons le bilan des travaux effectués et quelques perspectives de recherches.

CHAPITRE 2

Problématique

Sommaire

La synthèse de matériel a bouleversé la conception des circuits numériques. En effet jusqu'au début des années 1990 les architectes éditaient manuellement le schéma des circuits avant d'entrer dans les phases de placement et de routage.

Aujourd'hui les concepteurs de circuits disposent d'outils permettant de tester rapidement plusieurs architectures pour un même circuit. Les répercussions directes sont à la fois une augmentation de la qualité des circuits et la diminution du temps de conception. Cette automatisation dans le flot de conception est principalement dû à l'apparition des outils de synthèse.

Au cours de ce chapitre, nous présentons les différents niveaux d'abstraction possibles pour entrer dans les outils de synthèse et nous détaillons plus particulièrement la phase de compilation d'un circuit décrit au niveau transfert entre registres. Le VHDL est un des langages d'entrée des outils de synthèse actuels. Certaines de ses propriétés sont soulignées afin de mettre en relief les problèmes liés à son utilisation. Nous définissons ensuite précisément le problème que nous nous proposons d'aborder dans le cadre de cette thèse.

2.1 La synthèse

La synthèse est définie comme une séquence de raffinements successifs visant à transformer un comportement spécifié dans un langage de description de matériel (*Hardware Description Langage*) en une liste de portes ou d'opérateurs numériques interconnectés. Elle correspond donc au passage d'un domaine comportemental à un domaine structurel [GDW92].

2.1.1 Les différents niveaux de synthèse

Les outils de synthèse cherchent tout d'abord à compiler une vue comportementale en un modèle cible adapté à la synthèse et indépendant du langage utilisé. Puis ils optimisent ce modèle avant d'en effecteur une projection structurelle sur un niveau d'abstraction immédiatement inférieur. Il existe donc plusieurs niveaux de synthèse qui correspondent chacun au passage d'un niveau d'abstraction n à un niveau $n-1$.

Deux critères principaux permettent de juger la qualité d'un outil de synthèse : la richesse de son langage d'entrée et la puissance de ses méthodes d'optimisation qui donnent la qualité de l'implémentation. Quel que soit le niveau d'abstraction, la synthèse prend en compte différents objectifs d'optimisation tels que la minimisation du délai, de la surface ou bien plus récemment, de la consommation du circuit.

La synthèse comportementale

La synthèse comportementale, appelée aussi synthèse de haut niveau, est la transformation d'une description de type algorithmique en une description de type transfert entre registres : *Register Transfert Level*.

Une description algorithmique ne comporte aucune information architecturale et temporelle, elle représente simplement l'algorithme à compiler sur le silicium. Les outils de synthèse de haut niveau doivent :

- effectuer une allocation efficace d'un ensemble de ressources matérielles permettant d'implanter les opérations nécessaire à l'exécution de l'algorithme. Cela consiste en général à interconnecter des registres avec des opérateurs complexes afin d'élaborer un ou plusieurs chemins de données ;
- construire un ou plusieurs séquenceurs capables de piloter dans le temps les opérateurs des différents chemins de données.

Le résultat de cette synthèse est donc une interconnexion de mémoires, de chemins de données et d'automates de contrôle dont le comportement est synthétisable par les outils de plus bas niveau.

La synthèse *RTL*

Le point d'entrée de la synthèse *RTL* est une description comportementale synchrone spécifiant le comportement d'un circuit cycle par cycle. Les états du circuit sont complètement définis et l'ensemble des transitions d'un état à un autre est décrit sous la forme d'un ensemble de fonctions de transfert entre des registres. Cependant les registres matériels ne sont pas nécessairement identifiés.

La description comportementale est généralement constituée d'un ensemble d'affectations concurrentes et d'un ensemble de processus séquentiels communicants. Elle défini le comportement d'éléments matériels interconnectés, tels que des automates d'états, des mémoires, des blocs combinatoires, des opérateurs arithmétiques et des registres. Un outil de synthèse *RTL* peut cependant modifier l'allocation de ces ressources matérielles. C'est le niveau d'entrée des outils industriels tels que : *Design Compiler* de *Synopsys*[Syn98], *Synergy* de *Cadence*[Cad97], *Asic Synthesizer* de *Compass*[Com96] ou encore *Autologic* de *Mentor Graphics*[Men95].

La tâche d'un outil de synthèse *RTL* est de transformer une telle description comportementale en une interconnexion d'éléments matériels connus de l'outil, tout en respectant des contraintes d'optimisation. Ces éléments matériels sont par exemple des portes logiques, des bascules, des barrières trois états ou bien encore des mémoires ou des opérateurs arithmétiques. C'est précisément ce niveau de synthèse qui nous intéresse dans le cadre de cette thèse.

Lors de la phase d'optimisation, un outil de synthèse *RTL* peut modifier profondément la structure du circuit sans bien sûr changer son comportement au niveau cycle. Par exemple il peut appliquer un nouveau codage aux états d'un automate. Il peut aussi, comme c'est illustré sur la figure 2.1, ajouter des registres en sortie d'une machine de Moore. La fonction de génération G de l'automate de Moore initial dépend exclusivement de l'état courant. Pour pouvoir ajouter des registres sur les sorties il doit transformer la fonction de génération G en une fonction G' qui dépende de l'état futur.

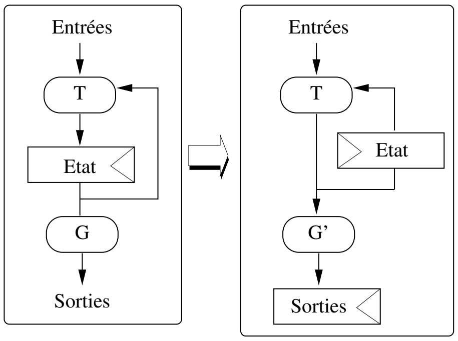

Figure 2.1: L'ajout de registres en sortie d'un automate .

Il peut également modifier la position de certain multiplexeurs pour partager efficacement des ressources matérielles critiques. La figure 2.2 illustre sur un exemple l'application de cette opération qui permet ici de partager un additionneur. Le signal de sélection *Sel* permet de choisir soit le résultat de l'addition des entrées A et B, soit le résultat de l'addition des entrées C et D. Il est donc possible d'utiliser un seul additionneur en effectuant la sélection sur les opérandes de l'addition.

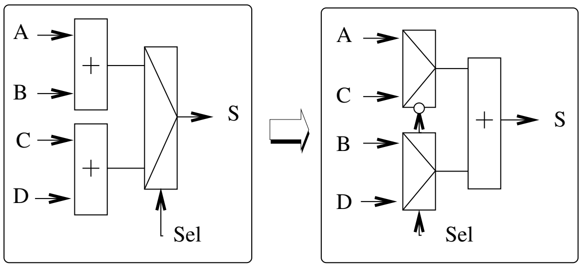

Figure 2.2: Un partage efficace des ressources .

Dans certains cas il est nécessaire d'effectuer des transformations encore plus importantes comme l'ajout de registres pour réduire la taille de certains chemins combinatoires critiques. Cette transformation est développée sur la figure 2.3. Après avoir ajouté un simple registre C' le temps de propagation a été divisé par deux entre le registre A et la sortie S. Le temps de traversée du deuxième additionneur a cependant été reporté en aval.

Dans cette thèse, nous nous situons au niveau transfert entre registres et nous considérons des descriptions comportementale dans lesquelles les éléments matériels tels

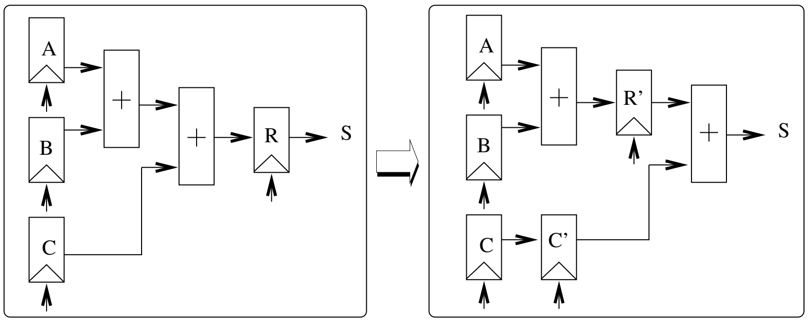

Figure 2.3: La réduction d'un chemin combinatoire inter registres .

que les registres ne sont pas clairement identifiés. Une analyse préliminaire du comportement matériel spécifié par la description est dans ce cas nécessaire. C'est un des problèmes complexes que la phase de compilation de la synthèse doit résoudre.

La synthèse logique

L'entrée de la synthèse logique est une description définissant le comportement d'un circuit sous la forme d'un ensemble d'affectation concurrentes représentant un réseau booléen. Cette description peut être vue comme le résultat intermédiaire d'une synthèse *RTL*: c'est à dire juste avant d'effectuer la projection structurelle, et après avoir ramené à un niveau booléen tous les opérateurs complexes. Ce type de description est donc facilement représentable sous la forme d'une interconnexion de portes logiques simples, de bascules et de barrières trois états.

La synthèse logique doit d'une part optimiser le réseau booléen. Elle doit d'autre part le transformer en une interconnexion de portes caractérisées d'une bibliothèque, en un réseau de logique programmable ou encore en un réseau de portes pré-diffusées. C'est le niveau de sortie de tous les outils industriels actuels.

Contrairement à la synthèse *RTL* les entrées des registres sont considérées comme des sorties du circuit et les sorties des registres comme des entrées primaires par les algorithmes de minimisation et de factorisation des équations booléennes [Bur93]. Cet aspect est illustré sur la figure 2.4 où l'ensemble des entrées des registres est noté R_i et l'ensemble des sorties R_o.

La phase d'optimisation de la synthèse logique ne peut donc pas modifier profondément la structure du circuit mais elle cherche à minimiser les chemins combinatoires.

2.1.2 Les trois principales étapes de la synthèse *RTL*

La synthèse *RTL* est constituée d'une phase de compilation, d'une phase d'optimisation et d'une phase de projection structurelle. L'enchaînement de ces trois phases et

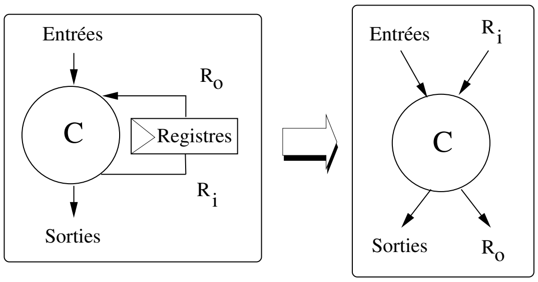

Figure 2.4: Le traitement spécifique appliqué aux registres en synthèse logique. .

le type des informations manipulées lors du passage d'une phase à une autre sont illustrés sur la figure 2.5. Les paragraphes suivants détaillent chacune de ces trois phases en insistant sur la phase de compilation puisque c'est le sujet auquel nous nous intéressons dans le cadre de cette thèse.

La compilation d'une description de matériel

La synthèse *RTL* consiste avant tout en une phase de compilation de la description d'un circuit spécifiant cycle par cycle son comportement. Le résultat de cette compilation est un modèle adapté à la synthèse et indépendant du langage de description utilisé. La phase de compilation de la synthèse doit :

- ramener tous les symboles manipulés à un niveau booléen c'est à dire les exprimer sous la forme de vecteurs de bits. Les symboles ainsi encodés sont éventuellement enrichis par des attributs afin de savoir si par exemple ils représentent des entiers signés ou non signés ;
- identifier précisément les éléments matériels particuliers que sont les registres et les barrières trois états ;
- extraire l'ensemble des fonctions de transfert entre les registres.

La phase de compilation doit donc construire, à partir de la spécification comportementale d'un circuit décrit dans un langage de description de matériel, un système comportant :

- un ensemble d'affectations combinatoires concurrentes et uniques représentant les fonctions de transfert entre les registres.
- un ensemble de registres et de barrières trois états clairement identifiés.

La phase de compilation a pour rôle essentiel de rendre explicite tout ce qui est implicite dans la description initiale. La complexité de cette phase dépend donc complètement du langage de description de matériel qui est utilisé en entrée de la synthèse.

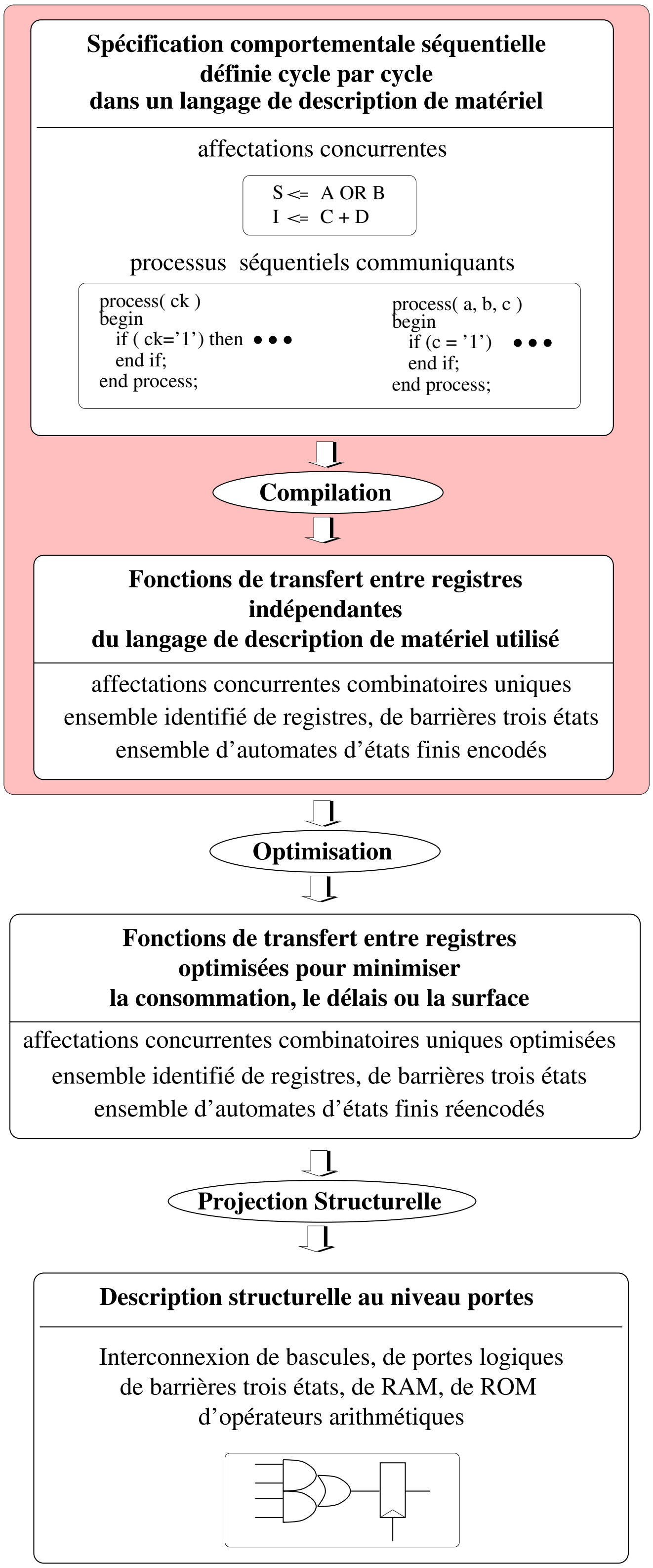

Figure 2.5: Le flot de la synthèse *RTL*.

En effet plus ce langage permet de décrire de façon différente et de façon implicite des éléments matériels, plus les techniques d'analyse à mettre en œuvre sont complexes.

La compilation de la synthèse *RTL* peut être découpée en deux étapes :

- une première étape transforme une spécification comportementale décrite dans un langage de description de matériel en un ensemble d'arbres syntaxiques. Cette étape est complètement spécifique au langage et préserve toute sa sémantique ;
- une seconde étape consiste à analyser cet ensemble d'arbres syntaxiques afin d'en extraire l'ensemble des registres et les fonctions de transfert entre ces registres : c'est la phase d'analyse sémantique de la compilation.

Ces deux étapes de la compilation *RTL* sont illustrées sur la figure 2.6. Nos travaux s'attachent à étudier précisément les problèmes à résoudre lors de la phase d'analyse sémantique de la compilation, lorsque le VHDL est pris comme langage d'entrée pour la synthèse *RTL*.

Durant la compilation il est nécessaire de conserver le maximum d'informations présentes dans la description du circuit et utiles à la synthèse. Il s'agit par exemple de conserver le regroupement des bits d'un bus de données.

Les problèmes posés ultérieurement par les phases d'optimisation et de projection structurelle ne doivent pas être ignorés puisqu'ils permettent d'orienter certains choix effectués pendant la compilation. En effet les outils appliquent éventuellement certaines règles de réécriture liées au langage utilisé. Il s'agit notamment d'appliquer des techniques bien connues des compilateurs telles que :

- l'insertion ou la suppression de variables ;
- le déroulement des boucles ayant des bornes statiques ;
- la substitution d'un appel de fonction par son corps.

L'optimisation

Après avoir achevé la compilation d'une description et extrait un ensemble de registres et de fonctions de transfert entre registres, l'outil doit alors appliquer différentes techniques d'optimisation propre à la synthèse. Ces opérations, qui peuvent être au besoin dirigées par le concepteur, sont par exemple :

- le choix et le partage des ressources matérielles ;
- la minimisation et l'encodage des automates d'états finis ;
- l'optimisation des expressions booléennes.

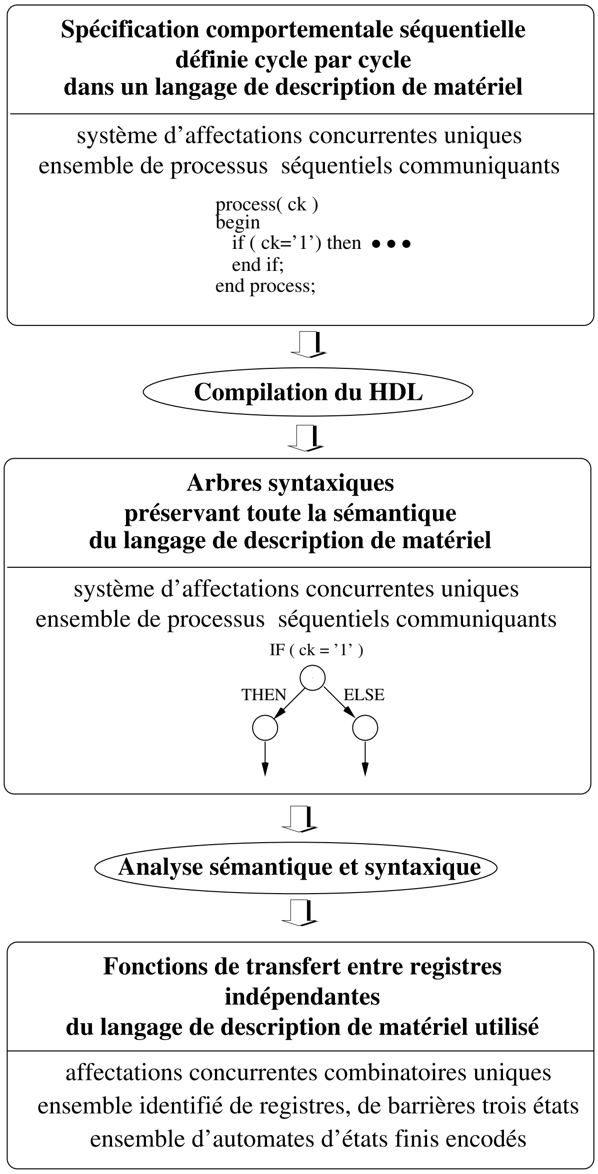

Figure 2.6: Les deux étapes de la phase de compilation .

La projection structurelle

La phase de projection structurelle succède à la phase d'optimisation. Elle consiste à trouver une implantation matérielle aux registres et aux fonctions de transferts précédemment optimisées.

Lors de cette transformation un outil de synthèse peut par exemple :

- ajouter des amplificateurs en sortie d'une porte logique ;
- construire un arbre de distribution pour un signal particulier ;

Le résultat de cette projection structurelle est une interconnexion de portes logiques, de registres, de barrières trois états, de mémoires et d'opérateurs arithmétiques.

2.2 Les particularités du langage VHDL

Le VHDL est très largement inspiré du langage de programmation ADA. Il est basé sur la notion d'exécution parallèle de processus séquentiels qui communiquent à travers des signaux. Il permet de décrire tous les aspects temporels liés aux communications entre processus. Il donne ainsi la possibilité de spécifier précisément le comportement intrinsèquement parallèle des circuits numériques, ainsi que leur caractéristiques temporelles.

2.2.1 Un langage fortement typé

Le langage VHDL offre la possibilité de définir ses propre types de données. Ceci permet notamment de rendre une spécification beaucoup plus lisible et souvent de lever les ambiguïtés. En effet, pour définir par exemple les différents états d'un automate, il est beaucoup plus agréable d'employer un type énuméré plutôt qu'un type entier.

La définition d'un type de données implique la mise en œuvre de fonctions de manipulation et de conversion. En VHDL il est possible de surcharger des fonctions existantes, applicables sur un type de données particulier, pour les adapter à d'autres types.

La vérification de la bonne utilisation des types et de leurs fonctions de manipulation est effectuée lors de la compilation. C'est un des avantages de VHDL, puisque les risques d'utilisation incorrecte des objets du langage sont considérablement diminués. Le prix à payer est la définition d'une multitude de fonctions de conversion et de manipulation pour chacun des types de données.

2.2.2 Un langage de simulation

La norme de VHDL spécifie non seulement la grammaire et la syntaxe du langage mais aussi la sémantique de simulation associée à l'utilisation de chacune des instructions.

L'élaboration d'une description VHDL

Le mécanisme d'élaboration défini par la norme assure en particulier la transformation des instructions concurrentes en processus équivalents. C'est pourquoi nous ne nous intéressons par la suite qu'aux processus VHDL.

La figure 2.7 illustre la transformation d'une affectation concurrente modélisant le comportement d'un multiplexeur en un processus VHDL équivalent. La liste de sensibilité du processus correspond à l'ensemble des signaux qui apparaissent dans le membre de droite de l'affectation concurrente.

```
ENTITY EXAMPLE IS
PORT ( A, B, SEL : in  BIT;
       S         : out BIT )
END EXAMPLE;

ARCHITECTURE DF OF EXAMPLE IS
BEGIN
   S <= A WHEN SEL='1' ELSE B;
END DF;
```

$\Rightarrow$

```
ENTITY EXAMPLE IS
PORT ( A, B, SEL : in  BIT;
       S         : out BIT )
END EXAMPLE;

ARCHITECTURE ELAB OF EXAMPLE IS
BEGIN
  PROCESS( A, B, SEL )
  BEGIN
    IF ( SEL='1') THEN S <= A;
                  ELSE S <= B;
    END IF;
  END PROCESS;
END ELAB;
```

Figure 2.7: La construction d'un processus équivalent lors de l'élaboration.

La phase d'élaboration assure également la création de tous les signaux et de toutes les variables de la description et elle leur donne une valeur initiale bien définie.

Le cycle de simulation VHDL

La simulation VHDL est pilotée par l'apparition d'évènements sur les signaux de communication entre processus. Le cycle de simulation appelé delta-cycle est découpé en trois phases distinctes *EXECUTE*, *UPDATE* et *RESUME* définies ainsi :

- la phase *EXECUTE* correspond à une exécution en parallèle des processus jusqu'à leur prochain point de suspension. Au cours de cette phase les variables internes des processus réveillés sont modifiées au moment de leur affectation et les transactions effectuées sur les signaux de communication sont enregistrées dans un échéancier ;

- au cours de la phase *UPDATE*, les valeurs et les différents attributs des signaux sont mis à jour à partir de l'échéancier ;
- la phase *RESUME* établit la liste des processus à réveiller en fonction des évènements intervenus dans l'échéancier.

Avant d'entrer dans la boucle de simulation il est nécessaire d'initialiser la description. Cette étape préliminaire est définie dans le manuel de référence du langage VHDL et elle consiste notamment à exécuter une fois chacun des processus de la description jusqu'à rencontrer un point de suspension. Durant cette exécution les variables et les signaux peuvent changer de valeur.

Ces différentes initialisations n'expriment pas un comportement matériel réel, elles sont simplement le support d'un mécanisme de simulation mais elles rendent la tâche de la compilation de la synthèse d'autant plus délicate.

Les deux classes de processus VHDL

On peut distinguer deux grandes classes de processus VHDL :

- les processus contenant un unique point de suspension. Ils sont réveillés par un unique ensemble d'évènements et reprennent toujours leur exécution au même endroit ;
- les processus contenant plusieurs points de suspension. Il existe ainsi une variable d'état implicite qui représente le prochain point de réveil. Cette variable cachée peut être vue comme l'état courant d'un automate d'états finis.

La figure 2.8 représente à gauche un processus VHDL à plusieurs points de suspension, et à droite un processus à un seul point de suspension. Ces deux descriptions spécifient le comportement d'un même automate d'état avec des registres en sortie.

Dans le cas du processus *PMW* de gauche, une variable d'état implicite lui est associée, alors que dans le cas de la description de droite le registre d'état *STATE* est explicite. Cette variable d'état implicite contient le point de suspension sur lequel le processus *PMW* doit se réveiller au prochain front montant de l'horloge *CK*. Il est du ressort de l'outil de synthèse de rendre explicite ce type de variable d'état.

Du point de vue d'un simulateur VHDL, il n'existe pas de réelle différence entre ces deux classes de processus.

Cependant certains outils de synthèse n'autorisent pas les processus constitués de plusieurs points de suspension car il est souvent difficile de trouver, pour de tels processus une implantation matérielle fidèle à la sémantique de simulation.

<pre>ENTITY FSM IS
PORT (A, B, CK : in BIT;
       O   : out BIT)
END FSM;
 	
ARCHITECTURE MW OF FSM IS
BEGIN
  PMW : PROCESS
    VARIABLE V1, V2 : BIT;
  BEGIN
    WAIT UNTIL CK='1';
    V1 := A;
    WAIT UNTIL CK='1';
    V2 := B;
    WAIT UNTIL CK='1';
    O <= V1 XOR V2;
  END PROCESS PMW;
END MW;</pre>
	<pre>ARCHITECTURE OW OF FSM IS
	BEGIN
	  PSW : PROCESS (CK)
	    TYPE STATE_TYPE IS (S0, S1, S2);
	    VARIABLE V1, V2 : BIT;
	    VARIABLE STATE : STATE_TYPE;
	  BEGIN
	    IF (CK = '1')
	    THEN
	      CASE STATE IS
	        WHEN S0 => V1 := A;
	                   STATE := S1;
	        WHEN S1 => V2 := B;
	                   STATE := S2;
	        WHEN S2 => O <= V1 XOR V2;
	                   STATE := S0;
	      END CASE;
	    END IF;
	  END PROCESS PSW;
	END OW;</pre>

Figure 2.8: Deux descriptions d'un automate avec un registre en sortie .

Les variables et les signaux

Les variables et les signaux gardent leur valeur lorsqu'ils ne sont pas affectés. L'affectation d'une variable est effectuée instantanément et elle peut donc prendre des valeurs différentes au cours d'un même delta-cycle. L'usage des variables dans les processus VHDL est un confort d'écriture lors de la mise en œuvre des spécifications d'un circuit. Cependant, elles n'ont pas forcément de réalité matérielle, cela dépend de leur contexte d'utilisation. Leur analyse lors de la phase de compilation est de ce fait très complexe.

2.3 Le VHDL comme langage d'entrée de la synthèse *RTL*

Comme nous l'avons vu précédemment, le VHDL est avant tout un langage de simulation et il n'a pas été défini pour être directement utilisable en synthèse.

La phase de compilation de la synthèse *RTL* prend en entrée une spécification qui se présente sous la forme :

- d'un ensemble d'affectations concurrentes ;
- d'un ensemble de processus VHDL séquentiels.

La phase d'élaboration de VHDL transforme toutes les affectations concurrentes en processus équivalents. La spécification VHDL comportementale d'entrée peut donc être vue comme un ensemble de processus communicants. La phase de compilation et d'analyse sémantique de la synthèse *RTL* doit analyser chacun de ces processus et construire un système comportant :

- un ensemble de registres et de barrières trois états dont les propriétés ont été clairement identifiées.
- un ensemble d'affectations combinatoires concurrentes et uniques représentant les fonctions de transfert entre les registres. Ces fonctions ne manipulent que des vecteurs de bits et des opérateurs connus de l'outil de synthèse ;

Il existe deux problèmes majeurs liés à l'utilisation de VHDL comme langage d'entrée de la synthèse qui sont développés dans les sections suivantes.

2.3.1 Les contraintes incontournables de la synthèse

Dans le cadre de la synthèse, les types et leurs fonctions de manipulation sont soumis à deux contraintes incontournables : ils doivent pouvoir être implantés de façon matérielle, et ils doivent appartenir à une classe connue de l'outil de synthèse. Ces contraintes impliquent la résolution de trois problèmes distincts :

- d'une part certains types standards de VHDL, tels que les descripteurs de fichier, les pointeurs ou les types utilisés pour représenter le temps, ne trouvent pas d'équivalence matérielle. Il en est de même pour les fonctions de manipulation de ces types. Il est donc indispensable de se restreindre aux types et fonctions utilisables pour la synthèse.
- d'autre part, il est nécessaire de définir un nouvel ensemble de types et de fonctions possédant une sémantique associée à la synthèse. Les types standard *BIT* et *BIT_VECTOR* de VHDL ne permettent pas, par exemple, de modéliser de façon explicite le fait qu'un bus est placé en haute impédance, ou encore qu'un registre n'a pas été initialisé. Ce nouvel ensemble de types et de fonctions doit être défini de façon à ce que les outils puissent établir une correspondance directe entre ces nouveaux types VHDL et une implantation matérielle.
- enfin, le langage VHDL possède des instructions spécifiques qui trouvent leur utilité dans le cadre de la simulation mais qui n'ont aucun sens dans le cadre de la synthèse. Il s'agit par exemple des instructions d'allocation dynamique d'objets, ou des instructions qui permettent de modéliser les aspect temporels d'un circuit. Il est donc indispensable d'interdire ou d'ignorer ces instructions lors de la mise en œuvre d'un sous ensemble VHDL synthétisable.

Ces trois problèmes ont été résolus par les normes IEEE-1164 et IEEE-1076.3 [IEE93a, IEE97]. Elles définissent un ensemble de types et de fonctions ayant une sémantique claire à la fois pour la simulation et pour synthèse. Les outils de synthèse industriels ont tous adopté ces deux normes.

2.3.2 La synthèse des processus VHDL

La compilation de la synthèse *RTL* analyse et transforme un à un chacun des processus d'une description comportementale VHDL élaborée. Cette transformation doit bien sûr se faire en accord avec la sémantique de simulation de VHDL. Elle a pour objectif :

- de vérifier que le comportement d'un processus VHDL est synthétisable ;
- d'identifier les signaux ou les variables qui correspondent à des registres ou à des barrières trois états ;
- d'identifier les fonctions de transfert entre les registres et de les mettre sous la forme d'un ensemble d'affectations combinatoires concurrentes et uniques.

La phase d'initialisation de la simulation VHDL

Une des tâches de la phase d'initialisation de la simulation consiste à donner une valeur initiale à tous les symboles de la description. Ce mécanisme ne peut pas toujours être modélisé en matériel et il est donc illusoire de vouloir obtenir un résultat de synthèse qui préserve strictement le comportement de la description initiale. En effet l'implantation matérielle de la phase d'initialisation de la simulation n'est envisageable que si la technologie cible de la synthèse possède un mécanisme d'initialisation implicite des registres.

Ce problème est illustré sur la figure 2.9. La variable **V** est du type *BIT* et prend donc la valeur *'0'* par défaut dans les deux processus **P1** et **P2**. La phase d'initialisation de la simulation quand à elle, exécute une fois chaque processus jusqu'au premier point de suspension. Lors de son exécution le processus **P1** s'arrête immédiatement sans avoir modifié la variable **V** puisque sa première instruction est un *WAIT*. Le processus **P2** par contre, donne la valeur *'1'* à la variable **V** avant de s'arrêter sur un point de suspension. Ainsi ces deux descriptions VHDL correspondent à deux comportements différents du point de vue de la simulation.

Certaines technologies cibles comme les *FPGA* possèdent un mécanisme d'initialisation de tous les registres. Il est donc possible dans de telles conditions de forcer le registre **V** à prendre la valeur *'0'* ou *'1'* et de trouver une implantation qui préserve totalement le comportement de la description initiale.

<pre>ENTITY INIT_1 IS
PORT (CK : in BIT;
       S : out BIT)
END INIT_1;
 	
ARCHITECTURE BEH OF INIT_1 IS
BEGIN
  P1 : PROCESS
    VARIABLE V : BIT;
  BEGIN
    WAIT UNTIL CK = '1';
    V := NOT V;
    S <= V;
  END PROCESS P1;
END BEH;</pre>
	<pre>ENTITY INIT_2 IS
	PORT (CK : in BIT;
	       S : out BIT)
	END INIT_2;
	
	ARCHITECTURE BEH OF INIT_2 IS
	BEGIN
	  P2 : PROCESS
	    VARIABLE V : BIT;
	  BEGIN
	    V := NOT V;
	    S <= V;
	    WAIT UNTIL CK = '1';
	  END PROCESS P2;
	END BEH;</pre>

Figure 2.9: L'initialisation de la simulation VHDL.

Cependant la plupart des technologies cibles utilisées dans la conception de circuit *VLSI* ne permettent pas de forcer un registre à prendre une valeur initiale de façon implicite à la mise sous tension. Ces deux descriptions seront donc implantées en matériel exactement de la même façon, par un registre ayant une valeur initiale indéterminée. Suivant la bibliothèque de cellules utilisée, le matériel se comportera soit comme le processus **P1**, soit comme le processus **P2**.

L'analyse des processus VHDL

La phase de compilation doit être en mesure d'analyser chacun des processus VHDL pour vérifier si leur comportement est implantable en matériel. Il n'existe malheureusement pas de règles purement syntaxiques qui permettent d'identifier *a priori* et avec certitude les processus synthétisables et ceux qui ne le sont pas.

L'analyse doit prendre en compte la sémantique de simulation du langage VHDL ce qui rend sa tâche complexe. Par exemple la nature des conditions de réveil des processus peut complètement modifier le résultat de l'analyse.

La figure 2.10 illustre l'influence de la liste de sensibilité des processus sur leur comportement et donc sur le résultat de la synthèse. Les corps des processus **P1** et **P2** sont exactement les mêmes et pourtant seul **P2** est synthétisable. En effet la condition de réveil du processus **P1** impose une mise à jour du signal **S** lorsqu'un évènement survient sur le signal **A** ou sur le signal **B** mais pas sur le signal **C**. Ceci ne correspond pas à un comportement matériel légal et le processus **P1** n'est donc pas synthétisable.

L'analyse d'un processus VHDL afin de savoir s'il est ou non synthétisable, est intimement liée à la phase de transformation d'un processus en un ensemble de registres et d'affectations combinatoires concurrentes.

<pre>ENTITY UNSYNTH IS
PORT (A : in BIT;
       B : in BIT;
       C : in BIT;
       S : out BIT)
END UNSYNTH;
 	
ARCHITECTURE BEH OF UNSYNTH IS
BEGIN
  P1 : PROCESS(A, B)
  BEGIN
    S <= A OR B OR C;
  END PROCESS P1;
END BEH;</pre>
	<pre>ENTITY SYNTH IS
	PORT (A : in BIT;
	       B : in BIT;
	       C : in BIT;
	       S : out BIT)
	END SYNTH;
	
	ARCHITECTURE BEH OF SYNTH IS
	BEGIN
	  P2 : PROCESS(A, B, C)
	  BEGIN
	    S <= A OR B OR C;
	  END PROCESS P2;
	END BEH;</pre>

Figure 2.10: L'influence de la liste de sensibilité.

L'identification des registres et la construction des fonctions de transfert

La phase d'identification des registres et de construction des affectations concurrentes traite chaque processus indépendamment des autres. Elle doit pouvoir établir la liste de toutes les modifications qui peuvent intervenir sur les signaux et les variables lors de l'exécution séquentielle du processus quelque soit le nombre de point de suspension. Elle doit aussi savoir dans quelle condition un signal ou une variable garde sa valeur pour pouvoir discerner les parties combinatoires des parties séquentielles.

Un critère simple généralement utilisé dans l'identification des registres consiste à dire qu'un symbole doit être implanté à l'aide d'un registre si et seulement si il n'est pas affecté sur tous les chemins d'exécution.

Par exemple dans le cas du processus de la figure 2.11 la sortie *Q* n'est affectée que lorsque le signal *CK* vaut '1'. Ainsi il existe au moins un chemin d'exécution pour lequel la sortie *Q* n'est pas affectée et conserve sa valeur. La synthèse doit être en mesure de conclure qu'un registre est nécessaire pour implanter en matériel ce comportement.

Cependant le fait qu'un symbole ne soit pas affecté sur tous les chemins d'exécution du graphe de contrôle n'est pas un bon critère pour identifier les registres. En effet les chemins d'exécution sur lesquels le symbole n'est pas affecté peuvent être des chemins qui ne seront jamais empruntés. Par exemple le graphe de contrôle du processus de la figure 2.12 est constitué de trois chemins d'exécution. Le premier chemin est emprunté lorsque l'entrée *SEL* vaut '1', le second chemin lorsque l'entrée *SEL* vaut '0'. La condition de franchissement du dernier chemin peut s'écrire : *(SEL/='1') AND (SEL/='0')*. Cette condition étant toujours fausse, il n'est jamais emprunté. Aussi, bien que le signal *MUX* ne soit pas affecté sur ce chemin, il correspond non pas à un registre, mais à la sortie d'un multiplexeur.

Il existe également des cas où un signal est affecté sur tous les chemins d'exécution et pourtant il correspond à la sortie d'un registre. Une telle configuration est illustrée

```
ENTITY EXAMPLE IS
PORT ( CK : IN  BIT;
       I0 : IN  BIT;
       Q  : OUT BIT );
END EXAMPLE;

ARCHITECTURE BEH OF EXAMPLE IS
BEGIN
  PROCESS( CK )
  BEGIN
    IF ( CK = '1' )
      THEN Q <= I0;
    END IF;
  END PROCESS;
END BEH;
```

Figure 2.11: Un exemple de bascule D.

```
ENTITY EXAMPLE IS
PORT ( SEL : IN  BIT;
       I0  : IN  BIT;
       I1  : IN  BIT;
       MUX : OUT BIT );
END EXAMPLE;

ARCHITECTURE BEH OF EXAMPLE IS
BEGIN
  PROCESS( SEL, I0, I1 )
  BEGIN
    IF ( SEL = '1' )
      THEN MUX <= I0;
    ELSIF ( SEL = '0' )
      THEN MUX <= I1;
    END IF;
  END PROCESS;
END BEH;
```

Figure 2.12: Un exemple de multiplexeur.

sur la figure 2.13. La variable *V* et le signal *S* sont affectés à chaque réveil du processus, et pourtant le signal *S* ne change réellement de valeur que sur front montant de l'horloge *CK*.

```
ENTITY EXAMPLE IS
PORT ( CK : IN  BIT;
       I0 : IN  BIT;
       Q  : OUT BIT );
END EXAMPLE;

ARCHITECTURE BEH OF EXAMPLE IS
   SIGNAL S : BIT;
BEGIN
  PROCESS( CK )
    VARIABLE V : BIT;
  BEGIN
    V := S;
    IF ( CK = '1' )
      THEN V := I0;
    END IF;
    S <= V;
  END PROCESS;
  Q <= S;
END BEH;
```

Figure 2.13: Un exemple de bascule D.

Puisque le simple critère "existence d'une affectation sur tous les chemins d'exécution" ne suffit pas pour identifier les registres, il faut effectuer une analyse plus approfondie. Pour cela il est nécessaire de déterminer pour chaque symbole (variable ou signal) quand et comment sa valeur est modifiée. Il y a donc deux problèmes à résoudre :

- comment extraire ces informations pour chaque symbole de la description?
- comment effectuer une analyse sémantique de ces informations?

La figure 2.14 présente deux descriptions VHDL d'une bascule D. Nous décrivons ci-dessous les informations pertinentes qui doivent être extraites du graphe de contrôle de chacun des deux processus.

Pour le processus **P1** :

- le processus **P1** est réveillé lorsqu'un évènement survient sur le signal **CK** ;
- la variable **V** prend la valeur du signal **A** lorsque la condition **CK = '1'** est vérifiée ;
- la variable **V** garde son ancienne valeur dans les autres cas ;
- la première instruction d'affectation de **S** doit être ignorée ;

- le signal **S** prend systématiquement la valeur de la variable **V**.

Pour le processus **P2** :

- le processus **P2** est réveillé lorsqu'un évènement survient sur le signal **CK** et que la condition **CK = '1'** est vérifiée ;
- le signal **S** prend la valeur du signal **A** lorsque le processus est réveillé.

```
ENTITY DREG IS
PORT ( CK, A : in  BIT;
       S     : out BIT )
END DREG;

ARCHITECTURE BEH1 OF DREG IS
BEGIN
  P1 : PROCESS( CK )
    VARIABLE V : BIT;
  BEGIN
    S <= A;
    IF (CK = '1') THEN
      V := A;
    END IF;
    S <= V;
  END PROCESS P1;
END BEH1;
```

```
ENTITY DREG IS
PORT ( CK, A : in  BIT;
       S     : out BIT )
END DREG;

ARCHITECTURE BEH2 OF DREG IS
BEGIN
  P2 : PROCESS
  BEGIN
    WAIT UNTIL CK='1';
    S <= A;
  END PROCESS P2;
END BEH2;
```

Figure 2.14: Deux descriptions VHDL d'une bascule D.

Les informations concernant ces deux processus VHDL semblent assez différentes. Ils ont pourtant le même comportement et un outil de synthèse doit donc les synthétiser de la même façon. La puissance de description de VHDL offre la possibilité de décrire de façon très diverse un même comportement. Ceci rend d'autant plus difficile l'analyse sémantique des informations extraites à partir du graphe de contrôle d'un processus.

2.4 La restriction du langage VHDL pour la synthèse

La grande majorité des outils de synthèse industriels acceptent des descriptions VHDL en entrée mais comme nous le montrerons plus en détail au chapitre 3, ils restreignent de façon importante la puissance de description du langage VHDL. En effet, afin de simplifier la phase d'analyse de la compilation ils n'acceptent qu'un petit nombre de motifs syntaxiques prédéfinis.

Deux objectifs bien distincts sont atteints en imposant des restrictions syntaxiques.

- cela permet d'ôter du langage tout ce qui ne pourra jamais trouver une implantation matérielle, ce qui est légitime.
- cela permet de simplifier considérablement l'analyse sémantique en la réduisant à une simple analyse syntaxique.

Chaque outil de synthèse a adopté un sous ensemble VHDL qui lui est propre. La disparité des sous ensembles entraîne bien évidemment des problèmes de compatibilité.

Les deux objectifs ci-dessus se traduisent par l'élaboration de deux familles de restrictions.

2.4.1 Les restrictions légitimes

La première famille cherche à retirer du langage les types, les attributs, les opérateurs et les instructions qui ne trouvent aucune équivalence en matériel. Ces restrictions sont simples à mettre en œuvre et laissent au langage toute sa puissance de description. Elles sont généralement les mêmes pour tous les outils de synthèse et sont spécifiées par :

- un ensemble de types auxquels l'outil de synthèse est capable d'associer une sémantique matérielle. La liste de ces types est généralement composée de ceux spécifiés par les normes IEEE-1164 et IEEE-1076.3 [IEE93a, IEE97] et d'une liste de familles de types que l'utilisateur peut lui même définir ;
- une liste des attributs utilisables pour la synthèse. Ces attributs peuvent servir par exemple à spécifier les caractéristiques du front d'horloge d'un registre ;
- un ensemble d'opérateurs et de fonctions qui trouvent une correspondance directe en matériel ;
- la définition d'une grammaire réduite du langage VHDL afin d'interdire ou de limiter l'usage de certaines instructions ou certaines constructions, qui permettent de manipuler le temps ou d'afficher des messages lors d'une simulation.

Il est évidemment illusoire d'espérer relâcher ces restrictions.

2.4.2 Les restrictions en vue d'une reconnaissance syntaxique

La seconde famille de restrictions cherche à simplifier l'identification des registres. Elles varient considérablement d'un outil à l'autre et elles sont généralement spécifiées par :

- un ensemble de règles à respecter lors de l'utilisation de certaines constructions du langage. Ces règles sont directement liées aux motifs que l'outil de synthèse doit pouvoir retrouver dans une description VHDL ;
- un ensemble de conventions d'écriture qui viennent enrichir les contraintes syntaxiques. Il s'agit par exemple de restreindre l'utilisation des variables à l'intérieur d'un processus, car elles peuvent induire des phénomènes comportementaux difficiles à analyser par une méthode de reconnaissance purement syntaxique.

L'effort de normalisation [IEE99] cherche en particulier à définir de façon très précise cette dernière famille de contraintes. Il s'agit entre autre de trouver une partie commune dans l'ensemble des motifs reconnus par les outils de synthèses industriels actuels.

Dans le cadre de nos travaux nous cherchons à relâcher complètement toutes ces restrictions spécifiques à une méthode de reconnaissance syntaxique.

2.5 Conclusion

Au cours de ce chapitre, les différents niveaux de la synthèse ont été présentés en insistant sur celui que nous étudions dans le cadre de cette thèse, c'est-à-dire le niveau transfert entre registre.

Nous avons discerné les trois phases de la synthèse *RTL*, que sont la phase de compilation d'un modèle décrit dans un langage de description de matériel et les phases d'optimisation et de projection structurelle. Nous avons dissociés les deux parties de la compilation de la synthèse que sont l'analyse syntaxique qui fournit un ensemble d'arbres syntaxiques, et l'analyse sémantique qui permet d'identifier les registres et d'extraire un ensemble de fonctions de transfert.

Nous avons analysé les problèmes posés par l'utilisation du langage VHDL comme langage d'entrée de la synthèse *RTL*. Nous avons identifié les différentes tâches de la phase de compilation VHDL en vue de la synthèse. La plus importante consiste à analyser et transformer un ensemble de processus VHDL séquentiels en un ensemble d'affectations concurrentes, de registres et de barrières trois états, tout en respectant la sémantique de simulation du langage.

Nous avons évoqué la nécessité de restreindre VHDL, et nous avons mis en évidence deux familles de restrictions. D'une part les restrictions que nous pouvons qualifier de légitimes et qui visent simplement à rendre ce langage utilisable dans le cadre de la synthèse. Et d'autre part les restrictions extrêmement fortes établies différemment pour chaque outil de synthèse et qui permettent d'utiliser des méthodes purement syntaxiques pour identifier les registres.

La compilation en vue de la synthèse impose de répondre à l'ensemble des questions suivantes :

- comment analyser et vérifier qu'une description VHDL est synthétisable?
- comment extraire d'un processus VHDL séquentiel un ensemble d'affectations concurrentes, tout en restant en accord avec le comportement défini par la sémantique de simulation du langage?
- comment identifier les registres, les barrières trois états sans imposer de contraintes syntaxiques?

Un bon moyen d'évaluer la puissance et la qualité de la méthode que nous proposons est le suivant :

1. nous choisissons un ensemble de descriptions VHDL modélisant un comportement matériel synthétisable, et telles que chacune d'entre elles est rejetée par un ou plusieurs outils commerciaux, ou même refusée par tous les outils existants ;
2. nous compilons et analysons toutes ces descriptions afin de générer pour chacune d'entre elles, une description VHDL équivalente, acceptable et synthétisable par tous les outils commerciaux.

Les nouvelles descriptions VHDL ainsi générées sont constituées :

- d'un ensemble d'affectations combinatoires concurrentes et uniques. Elle ne manipulent que des vecteurs de bits et des opérateurs standards connus des outils commerciaux ;
- d'un ensemble de registres et de barrières trois états dont les propriétés sont clairement identifiées.

CHAPITRE 3

État de l'art

Sommaire

Au cours de ce chapitre nous présentons les deux approches orthogonales qui peuvent être utilisées pour atteindre les objectifs de la phase d'analyse de la compilation en vue de la synthèse de niveau *RTL*, à savoir les méthodes d'analyse syntaxique et les méthodes d'analyse sémantique. Nous développons pour chacune de ces deux approches les limitations et les inconvénients des méthodes existantes.

3.1 Les méthodes d'analyse syntaxique

Afin d'éviter de prendre en compte les mécanismes complexes de la simulation VHDL, tous les outils de synthèse industriels imposent des contraintes syntaxiques. Il en est de même pour les outils de traduction de descriptions VHDL de niveau *RTL* en langage Verilog [X-T97, BVB97].

Chaque outil de synthèse a défini ses propres contraintes et ses propres règles. Les contraintes syntaxiques sont documentées dans les manuels d'utilisation sous la forme d'un ensemble de petits exemples génériques, pour chacun des éléments matériels qu'il est possible de modéliser en VHDL. D'autres règles plus générales visant à simplifier encore d'avantage la phase de compilation y sont ajoutées. Elles portent notamment sur la façon d'utiliser les variables et les signaux à l'intérieur d'un processus.

La mise en œuvre logicielle d'une méthode d'analyse syntaxique est relativement simple puisqu'il ne s'agit pas de prendre en compte la sémantique de simulation du langage mais simplement de reconnaître dans une description les différents motifs autorisés.

3.1.1 La rigidité des restrictions syntaxiques

La figure 3.1 présente les trois styles de description d'une bascule D à mémorisation sur front montant, autorisés par l'outil de synthèse industriel *Design Compiler* de *Synopsys* [Syn98]. Le seule liberté offerte à l'architecte est de permuter les deux membres *CK='1'* et *CK'EVENT*.

```
PROCESS( CK )
BEGIN
 IF (CK='1' AND CK'EVENT)
 THEN Q <= D;
 END IF;
END PROCESS;
```

```
PROCESS
BEGIN
 WAIT UNTIL CK='1';
 Q <= D;
END PROCESS;
```

```
PROCESS
BEGIN
 WAIT UNTIL CK='1' AND CK'EVENT;
 Q <= D;
END PROCESS;
```

Figure 3.1: Les différents styles de descriptions d'une bascule D.

Ces contraintes syntaxiques sont parfois extrêmement fortes et elles imposent de façon absolue l'utilisation d'un style de description prédéfini.

3.1.2 La disparité des sous-ensembles VHDL définis pour la synthèse

Tous les outils n'imposent pas les mêmes contraintes [SEV95], et certains sont plus souples que d'autres. La figure 3.2 présentes différentes descriptions d'une bascule D à mémorisation sur front montant, qui sont acceptées par l'outil de synthèse *Asic Synthesizer*[Com96], mais qui sont rejetées par l'outil *Design Compiler*[Syn98].

Cette disparité entre les différents outils pose un réel problème quant à la migration d'une description VHDL d'une plate-forme de synthèse vers une autre.

Ce problème est aujourd'hui en passe d'être résolu par la norme [IEE99] qui définit les différents motifs qui doivent être acceptés par les différents outils de synthèse industriels.

Néanmoins la synthèse des descriptions VHDL qui ont été écrites avant l'apparition de cette norme reste un véritable problème. De plus, à notre connaissance les outils de synthèse industriels actuels ne sont pas tous en conformité avec cette norme.

```
PROCESS( CK )
BEGIN
 IF (NOT(CK)='0' AND CK'EVENT)
 THEN Q <= D;
 END IF;
END PROCESS;
```

```
PROCESS( CK )
BEGIN
 IF (CK='1' AND CK'EVENT)
 THEN Q <= D;
 ELSE Q <= Q;
 END IF;
END PROCESS;
```

```
PROCESS( CK )
BEGIN
 IF (CK'EVENT)
 THEN
  IF (CK='1') THEN Q <= D;
 END IF;
END PROCESS;
```

Figure 3.2: L'incompatibilité entre les outils de synthèse.

3.1.3 L'identification des registres

Un autre inconvénient majeur de l'analyse syntaxique, est qu'elle ne permet pas toujours d'identifier correctement les registres. Les processus *L1* et *D1* de la figure 3.3, modélisent respectivement le comportement d'un *latch* et d'une bascule D.

```
L1: PROCESS( CK, D )
BEGIN
 IF (CK='1')
 THEN Q <= D;
 ELSE Q <= Q;
 END IF;
END PROCESS L1;
```

```
D1 : PROCESS( CK )
BEGIN
 IF (CK='1')
 THEN Q <= D;
 END IF;
END PROCESS D1;
```

Figure 3.3: L'identification incorrecte des éléments mémorisants.

L'outil de synthèse *Design Compiler*[Syn98] identifie un multiplexeur pour le processus *L1* et un *latch* dans le cas du processus *D1*. C'est un problème d'autant plus grave que d'autres outils de synthèse acceptent et synthétisent correctement ces deux descriptions.

3.1.4 La détection des descriptions non synthétisables

Un dernier inconvénient majeur de l'analyse syntaxique, est qu'elle ne permet pas toujours de s'assurer qu'une description est synthétisable. La figure 3.4 donne deux exemples de processus non synthétisables, et qui pourtant sont acceptés par l'outil de synthèse *Design Compiler*.

```
P1: PROCESS( A )
BEGIN
 Q <= A OR B;
END PROCESS P1;
```

```
P2 : PROCESS( CK, I1 )
  VARIABLE V : BIT;
BEGIN
 O0 <= V OR I1;
 IF ( CK'EVENT AND CK='1')
 THEN V := I0;
 END IF;
END PROCESS P2;
```

Figure 3.4: Les descriptions non synthétisables.

La méthode d'analyse syntaxique utilisée par les outils de synthèse est certes simple à implanter en logiciel mais elle ne permet pas toujours d'atteindre les objectifs de la phase de compilation de la synthèse de niveau *RTL*. Par ailleurs cette méthode est très réductrice par rapport à la puissance de description du langage VHDL.

3.1.5 La migration des descriptions VHDL synthétisables

Une approche a été proposée par Dalila Rahal-Besseghir [RB98] pour résoudre le problème de la migration d'une description VHDL synthétisable, d'une plate-forme de synthèse vers une autre.

Cette méthode consiste à analyser syntaxiquement une description synthétisable qui a été acceptée par un outil de synthèse et de la réécrire pour qu'elle soit acceptée par tous les autres outils, en utilisant le sous-ensemble VHDL commun.

Ainsi on identifie chacun des motifs syntaxiques reconnus par l'outil de synthèse source, et on le transforme en un ou plusieurs motifs équivalents acceptables par tous les outils.

L'application de cette méthode a cependant ses limites car l'incompatibilité entre deux outils de synthèse n'est pas due exclusivement à la disparité des motifs syntaxiques.

La description présentée sur la figure 3.5 est acceptée par l'outil *Synergy* de *Cadence*[Cad97] et elle est rejetée par les deux outils *Design Compiler* et *Asic Synthesizer*. En effet l'utilisation des variables n'est pas soumise aux mêmes règles pour ces différents outils.

Dans ce cas particulier, il ne s'agit pas seulement de retrouver un motif syntaxique dans une description VHDL. Les variables $V1$ et $V2$ sont affectées toutes les deux dans une construction syntaxique de type "front d'horloge descendant". Les deux outils *Design Compiler* et *Asic Synthesizer* interdisent la lecture de la variable $V2$ après une telle

```
P1 : PROCESS( CK )
  VARIABLE V1,V2 : STD_LOGIC;
BEGIN
  IF ( CK = '0' AND CK'EVENT )
  THEN V1 := I0;
       V2 := V1;
  END IF;
  O0 <= V2;
END PROCESS P1;
```

Figure 3.5: Les limites des méthodes de réécritures syntaxiques.

construction, alors que l'outil *Synergy* l'accepte. Cette description demande une analyse syntaxique fine du graphe de contrôle du processus, afin de trouver pour chaque variable où et comment elles sont affectées, et où et comment elles sont lues.

Pour que cette description puisse être acceptée par tous les outils de synthèse il est nécessaire de supprimer la variable $V2$. La figure 3.6 présente un exemple de réécriture de ce processus. On peut remarquer que la structure de la description initiale a été profondément modifiée, puisqu'on a transformé la variable $V2$ en un signal S_V2, et que l'affectation du signal $O0$ a dû être mise à l'extérieur du processus *P1*.

```
P1_REWRITE : PROCESS( CK )
  VARIABLE V1 : STD_LOGIC;
BEGIN
  IF ( CK = '0' AND CK'EVENT )
  THEN V1   := I0;
       S_V2 <= V1;
  END IF;
END PROCESS P1_REWRITE;

O0 <= S_V2;
```

Figure 3.6: Un exemple de réécriture d'un processus VHDL.

La méthode d'analyse syntaxique proposée par Dalila Rahal-Besseghir [RB98] ne permet donc pas de résoudre dans tous les cas le problème de la migration d'une description VHDL synthétisable, d'une plate-forme de synthèse vers une autre.

3.2 Les méthodes d'analyse sémantique

Une approche opposée à l'analyse syntaxique consiste à s'appuyer exclusivement sur la sémantique de simulation du langage. Cette approche nécessite l'utilisation d'un modèle formel de VHDL et d'une méthode d'analyse de ce modèle spécifique à la synthèse.

3.2.1 Les modèles formels de VHDL

Depuis plusieurs années on assiste à l'émergence de nombreux modèles formels de VHDL [KB95]. Ces modèles utilisent différentes représentations : une description

VHDL est représentée sous la forme d'un réseaux de Pétri [Mur89, OC93, Döh94, EBC93, Baw96, MB96], d'automates d'états communiquant [DB93a] ou encore d'un ensemble de graphes de contrôle et de données enrichis de mécanismes de synchronisation [MM95].

La grande majorité de ces modèles ont été définis pour être utilisés dans le domaine de la preuve et de la vérification formelle [DB93b, Döh94, Olc95, DB95, Enc95, Baw96]. Il s'agit souvent de modéliser de façon fine tous les aspects de la simulation VHDL, et de vérifier un ensemble de propriétés sur une ou plusieurs descriptions.

Contrairement aux outils de synthèse, les outils de preuve formelle basés sur ces modèles imposent généralement des restrictions syntaxiques très faibles dans la façon d'écrire les instructions séquentielles à l'intérieur d'un processus VHDL.

Par contre ces applications imposent des restrictions importantes sur les types et les opérateurs VHDL acceptés, et les modèles formels qu'ils utilisent sont eux aussi de ce point de vue assez limités.

Dans le cas de la méthode de vérification formelle de propriétés proposée par Rajesh Bawa [Baw96], seuls les trois types simples *BOOLEAN*, *BIT* et *BIT_VECTOR* sont autorisés, et le modèle formel *VPN* utilisé ne permet de manipuler que des symboles booléens.

Dans le cadre de la synthèse les objectifs sont assez différents. Tout d'abord on ne cherche pas à modéliser de façon fine et explicite tous les mécanismes de la simulation VHDL. Ensuite il est impératif d'accepter tous les opérateurs et tous les types définis pour la synthèse, et en particulier les types et les opérateurs arithmétiques [IEE97].

Afin d'utiliser ces modèles pour la synthèse de niveau *RTL*, on peut imaginer les étendre pour qu'ils permettent de représenter tous les types et tous les opérateurs définis pour la synthèse [IEE93a, IEE97].

On peut également s'appuyer sur les modèles de VHDL [MM95, MB96] qui ont été définis explicitement pour la synthèse de haut niveau.

3.2.2 L'analyse sémantique en vue de la synthèse *RTL*

À notre connaissance très peu de travaux ont été entrepris dans le cadre de l'analyse sémantique d'une description VHDL pour la synthèse *RTL*.

Une méthode d'analyse sémantique et de synthèse a été proposée par Adrian Evans [EE95]. Elle consiste à traduire une description VHDL sous la forme d'un réseau de Pétri puis à effectuer une simulation symbolique de ce modèle afin d'obtenir un graphe des états stables. Une analyse formelle de ce graphe de simulation permet ensuite de le transformer en un automate d'états finis synthétisable. Cependant cette méthode est utilisable uniquement pour des descriptions VHDL modélisant le comportement d'un automate d'états finis synchrone et elle ne permet pas de traiter des descriptions de grande complexité, comme des chemins de données.

Une méthode d'identification des éléments mémorisants dans un processus VHDL a été proposée récemment par Emil Dumitrescu [DO98]. Cette méthode n'est cependant applicable que pour des processus ayant la structure syntaxique présentée sur la figure 3.7. Elle ne permet donc pas de répondre complètement aux besoins de la phase de compilation de la synthèse *RTL*.

```
PROCESS
  <Définitions de variables>
BEGIN
   WAIT UNTIL <Front d'horloge>
   <Instructions séquentielles>
END PROCESS;
```

Figure 3.7: L'unique structure syntaxique autorisée pour décrire les registres.

Une méthode d'analyse beaucoup plus générale a été entreprise par Gerhard Mekenkamp. Cette méthode consiste tout d'abord à traduire une description VHDL comportementale en un modèle *SIL*[MMM$^+$96]. Ce modèle *SIL* est un graphe de contrôle et de données enrichi de mécanismes de synchronisation et il a été défini pour être utilisé dans le cadre de la synthèse de haut niveau. Ce modèle est ensuite expansé à l'aide d'un ensemble de techniques de transformation. Cette expansion permet entre autre de faire apparaître explicitement toutes les interactions entre les différents processus VHDL de la description [Mek98].

D'après Gerhard Mekenkamp, ce modèle expansé peut directement être traité par un outil de synthèse de niveau *RTL*. Cependant cette méthode n'apporte aucune réponse précise aux questions primordiales suivantes :

- comment vérifier sur ce modèle que le comportement d'une description VHDL est synthétisable?
- comment identifier sur ce modèle les symboles qui correspondent à des registres ou à des barrières trois états, et comment identifier précisément leurs propriétés matérielles?
- comment identifier les fonctions de transfert entre les registres et comment les mettre sous la forme d'un ensemble d'affectations combinatoires concurrentes et uniques?

3.3 Conclusion

Au cours de ce chapitre nous avons distingué deux approches pour atteindre les objectifs de la phase de compilation de la synthèse : l'analyse syntaxique et l'analyse sémantique d'une description VHDL.

Nous avons tout d'abord montré les limitations et les inconvénients des méthodes d'analyse syntaxique :

- elles limitent fortement la puissance de description de VHDL ;
- elles diffèrent d'un outil de synthèse à un autre et par conséquent elles entraînent des problèmes de compatibilité et de portabilité ;
- elles ne permettent pas toujours d'identifier correctement les éléments mémorisants ;
- elles ne permettent pas non plus de s'assurer qu'une description est synthétisable ;

Nous avons ensuite introduit une autre approche qui consiste à prendre en compte la sémantique de simulation du langage VHDL. Cette approche s'appuie sur la définition d'un modèle formel de VHDL. La plupart des modèles formels existants ont été définis pour la preuve formelle et nous avons montré la nécessité de les étendre pour les utiliser dans le cadre la synthèse.

Nous avons ensuite insisté sur le fait que très peu de méthodes d'analyse sémantique ont été proposées. Enfin nous avons montré qu'aucune de ces méthodes ne permettaient d'atteindre complètement les objectifs de la phase de compilation de la synthèse de niveau *RTL*.

Dans cette thèse nous cherchons donc à définir un modèle formel de VHDL respectant les contraintes de la synthèse de niveau *RTL*, et à proposer une méthode d'analyse de ce modèle afin d'atteindre complètement les objectifs de la phase de compilation de la synthèse.

CHAPITRE 4

Un modèle formel pour représenter le graphe de contrôle d'un processus

Sommaire

La définition d'un modèle capable de représenter de façon exacte la sémantique de simulation d'un programme VHDL tout en étant bien adapté à la synthèse est au centre de nos travaux. En effet comme nous l'avons vu dans le chapitre 3, les méthodes d'analyse sémantique s'appuient sur l'existence d'un tel modèle.

Dans ce chapitre nous détaillons tout d'abord l'ensemble des informations qu'un modèle formel doit permettre de représenter et de manipuler pour répondre aux besoins de la phase de compilation de la synthèse. Nous définissons ensuite un modèle conforme à ces exigences et nous développons les étapes de sa construction à partir d'une description VHDL. Nous présentons enfin une méthode d'initialisation de notre modèle en vue de la synthèse.

4.1 Un modèle formel pour la synthèse

Un modèle formel de VHDL doit être capable d'exprimer la sémantique de simulation du langage. Afin d'être utilisable dans le cadre de la synthèse, il doit permettre de conserver toutes les informations utiles aux phases d'optimisation et de projection structurelle. Il doit également faciliter la résolution des problèmes posés lors de la compilation c'est-à-dire : la transformation d'un ensemble de processus VHDL en un ensemble d'affectations concurrentes combinatoires ainsi que l'identification des registres et des barrières trois états.

4.1.1 La sémantique de simulation du langage

De façon générale un modèle formel de VHDL doit permettre de représenter :

- l'exécution parallèle d'un ensemble de processus et l'exécution séquentielle du corps de chacun d'entre eux.
- les modifications intervenant sur les pilotes des signaux et les variables lors de l'exécution des processus ;
- les trois phases du cycle de simulation de VHDL qui dirigent à la fois l'exécution des processus et les modifications qui interviennent sur les signaux et les variables de la description.

L'objectif de la synthèse n'est pas d'implanter en matériel un simulateur VHDL. La modélisation du cycle de simulation est donc inutile dans le cadre de la synthèse. Il en va de même pour la modélisation fine des transactions effectuées sur les signaux. En effet les mécanismes d'enregistrement d'une transaction dans l'échéancier d'un simulateur n'a pas de sens en synthèse. Une modélisation de l'exécution parallèle de tous les processus est inutile pour la synthèse puisque chaque processus doit être analysé indépendamment des autres.

Un modèle adapté à la synthèse peut donc se limiter à représenter le graphe de contrôle de chacun des processus d'une description VHDL élaborée et donc en particulier :

- l'exécution séquentielle d'un processus ;
- l'affectation immédiate des variables ;
- l'affectation différée des signaux ;
- la suspension d'un processus.

4.1.2 La transformation des types

Le modèle résultant de la compilation ne doit faire référence qu'à des objets d'assez bas niveau, de taille finie et ayant une sémantique matérielle précise. Le VHDL étant un langage fortement typé il est impératif de conserver toutes les informations concernant les types. Néanmoins seuls les types utilisables en synthèse et notamment ceux définis par les normes IEEE-1164 et IEEE-1076.3 [IEE93a, IEE97] doivent être représentés.

Les types spécifiés par la norme ont aussi été définis en vue de la simulation et certains d'entre eux comportent des valeurs qui n'ont aucune signification en synthèse. Le type *STD_LOGIC* par exemple est constitué de neuf valeurs et seules certaines d'entre elles peuvent être exploitées par un outil de synthèse. Nous définissons le type $\mathcal{S}$ correspondant aux six valeurs utilisées en synthèse : $\mathcal{S} = \{\, '0','1','Z','L','H','-'\}$ et dont les éléments représentent respectivement le *zéro logique*, le *un logique*, l'état *haute impédance*, l'état *zéro faible*, l'état *un faible* et l'état *don't care*.

Ainsi un modèle de VHDL adapté à la synthèse doit permettre de représenter tous les types synthétisables de la description, sachant que seules les valeurs de l'ensemble $\mathcal{S}$ sont prisent en compte lors de la phase d'analyse de la compilation.

On peut distinguer deux catégories de types synthétisables :

- les types simples qui seront directement traduit dans le type $\mathcal{S}$ ou $\mathcal{S}^n$. Il s'agit des types *BIT*, *BOOLEAN* ou *STD_LOGIC* ainsi que les types vectoriels *BIT_VECTOR*, *STD_LOGIC_VECTOR*, *SIGNED* et *UNSIGNED*.
- les types plus complexes qui doivent être préalablement encodés dans un type $\mathcal{S}^n$ pour pouvoir être synthétisés. Il s'agit des types énumérés ou des types permettant de représenter des entiers comme *INTEGER* ou *NATURAL*.

Dans la suite on appellera bit et vecteur de bits une variable à valeurs respectivement dans $\mathcal{S}$ et $\mathcal{S}^n$.

Une des tâches de la compilation est de ramener tous les types, tous les opérateurs et toutes les fonctions au niveau bit ou vecteur de bits.

Cette traduction ne doit pas pour autant perdre les informations importantes que constitue l'ensemble des valeurs autorisées pour une variable ou un signal. Il s'agit donc de conserver précieusement le type d'origine de chaque symbole.

Tous les types énumérés sont encodés afin de les transformer en un vecteur de bits équivalent suivant un code entier croissant de 0 à $n-1$, où n est le nombre de valeurs. Ce codage assure que les attributs qui exploitent l'ordre des valeurs d'un type tels que *SUCC*, *PRED*, *VAL* ou *POS*, restent cohérents.

Les types énumérés sont très souvent utilisés pour spécifier l'ensemble des états d'un automate et le choix d'un codage peut modifier considérablement la qualité de la synthèse. En particulier, plusieurs études expérimentales très systématiques que nous avons menées sur un ensemble de 60 automates d'états [JP96, DJP97, DPJ97b, DPJ97a] montrent que dans le cas d'automates de taille importante, il est préférable d'utiliser un codage linéaire *one hot* afin de minimiser à la fois la taille, le délai et la consommation électrique de l'automate synthétisé. Cette réalité expérimentale est bien connue des concepteurs mais elle est rarement mentionnée dans les publications. Tous les outils de synthèse industriels donnent la possibilité de spécifier à la compilation un autre codage que celui qui est appliqué par défaut. La figure 4.1 donne un exemple de la méthode préconisée par Synopsys pour définir le codage d'un type énuméré.

```
TYPE STATE_TYPE IS ( E0, E1, E2 );
ATTRIBUTE ENUM_ENCODING : STRING;
ATTRIBUTE ENUM_ENCODING OF STATE_TYPE : TYPE IS
       "001 010 100";
```

Figure 4.1: Un exemple de spécification de codage pour un type énuméré .

```
ARCHITECTURE BEH_1 OF EXAMPLE IS

 TYPE   MUX3_CMD_TYPE IS  ( M0, M1, M2 );
 SIGNAL C : MUX3_CMD_TYPE;

 SIGNAL I0, I1, I2, S : BIT;

BEGIN
  P1 : PROCESS( C, I0, I1, I2 )
  BEGIN
    IF ( C = M0 ) THEN  S <= I0;
    END IF;
    IF ( C = M1 ) THEN  S <= I1;
    END IF;
    IF ( C = M2 ) THEN  S <= I2;
    END IF;
  END PROCESS P1;
END BEH_1;
```

```
ARCHITECTURE BEH_2 OF EXAMPLE IS

-- M0 : "00", M1 : "01", M2 : "10"
  SIGNAL C : BIT_VECTOR(1 DOWNTO 0);

  SIGNAL I0, I1, I2, S : BIT;

BEGIN
  P2 : PROCESS( C, I0, I1, I2 )
  BEGIN
    IF ( C = "00" ) THEN  S <= I0;
    END IF;
    IF ( C = "01" ) THEN  S <= I1;
    END IF;
    IF ( C = "10" ) THEN  S <= I2;
    END IF;
  END PROCESS P2;
END BEH_2;
```

Figure 4.2: L'encodage des types énumérés.

La figure 4.2 illustre le problème de la complétude du codage d'un type énuméré. Elle représente deux architectures d'un multiplexeur. L'architecture *BEH_1* utilise un type énuméré *MUX3_CMD_TYPE* pour définir les trois valeurs possibles *M0*, *M1* et *M2* de la commande du multiplexeur. L'architecture *BEH_2* décrit ce même multiplexeur une fois le type énuméré encodé avec les valeurs binaires *"00"*, *"01"* et *"10"*. Le nombre de valeurs possibles du type énuméré n'est pas une puissance de deux et le code *"11"* reste donc inutilisé. Pour la simulation mais pas pour la synthèse, le comportement du processus *P1* est complètement équivalent à celui du processus *P2*.

Le processus *P1* prend en compte de façon explicite les trois valeurs possibles du signal de commande du multiplexeur. Le processus *P2* ne teste que trois valeurs sur quatre et donc contrairement au processus *P1*, il ne prend pas en compte toutes les configurations du signal de commande *C*. L'affectation du signal *S* n'étant pas effectuée pour toutes les valeurs de *C*, la synthèse du processus *P2* cherche à implanter *S* à l'aide d'un *latch*.

Pour résoudre ce problème il convient d'une part d'associer à chaque type énuméré le codage qui lui a été appliqué et de conserver ainsi l'ensemble des valeurs que pourrait prendre un symbole de ce type. Il est nécessaire d'autre part de pouvoir identifier facilement les symboles qui représentent les valeurs d'un type énuméré et qui ont été mis sous la forme d'un vecteur de bits.

Les types entiers *INTEGER* ou *NATURAL* sont eux aussi transformés en vecteurs de bits. Leur codage et leur taille peuvent varier suivant leur signe et l'intervalle des valeurs entières qu'ils permettent de représenter. Le problème de la complétude precédemment évoqué pour les types énumérés se pose de la même façon pour les types entiers et il convient donc de conserver les mêmes informations.

4.1.3 La transposition des opérateurs et des fonctions standards

Les opérateurs et les fonctions standards, comme ceux définis par les normes IEEE-1164 et IEEE-1076.3 [IEE93a, IEE97], sont bien sûr connus de l'outils de synthèse et il existe donc une correspondance entre leur utilisation et une famille d'implantations matérielles possibles.

La transformation d'un type complexe comme le type entier en un type élémentaire impose une transposition de toutes les fonctions et de tous les opérateurs associés à ce type. Les opérateurs complexes tels que les opérateurs arithmétiques peuvent être synthétisés différemment suivant le critère d'optimisation choisi (surface, délai ou consommation électrique). Cependant la tâche de la compilation n'est pas de choisir une implantation matérielle précise pour ces opérateurs et ces fonctions.

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

La figure 4.3 montre les conséquences désastreuses de la transformation des opérateurs arithmétiques pendant la phase de compilation. L'architecture *BEH_1* décrit le

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY ADD_SUB IS
PORT (
 SEL : IN  STD_LOGIC;
 A   : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
 B   : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
 S   : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
);
END ADD_SUB;

ARCHITECTURE BEH OF ADD_SUB IS
BEGIN
 P1 : PROCESS( SEL, A, B )
 BEGIN
   IF ( SEL='1')
   THEN
   S <= STD_LOGIC_VECTOR(
     SIGNED(A)+SIGNED(B));
   ELSE
   S <= STD_LOGIC_VECTOR(
     SIGNED(A)-SIGNED(B));
   END IF;
 END PROCESS P1;
END BEH;
```

⇒

```
ARCHITECTURE RTL_1 OF ADD_SUB IS

  SIGNAL ADD_A_B   : STD_LOGIC_VECTOR(3 DOWNTO 0);
  SIGNAL ADD_CARRY : STD_LOGIC_VECTOR(3 DOWNTO 0);
  SIGNAL ADD_CIN   : STD_LOGIC;
  SIGNAL SUB_A_B   : STD_LOGIC_VECTOR(3 DOWNTO 0);
  SIGNAL SUB_CARRY : STD_LOGIC_VECTOR(3 DOWNTO 0);
  SIGNAL SUB_CIN   : STD_LOGIC;

BEGIN
  ADD_CIN   <= '0';
  ADD_A_B   <= A XOR B
                 XOR (ADD_CARRY(2 DOWNTO 0) & ADD_CIN);
  ADD_CARRY <= (A AND B )
            OR (A AND (ADD_CARRY(2 DOWNTO 0) & ADD_CIN))
            OR (B AND (ADD_CARRY(2 DOWNTO 0) & ADD_CIN));
  SUB_CIN   <= '1';
  SUB_A_B   <= A XOR NOT(B)
                 XOR (SUB_CARRY(2 DOWNTO 0) & SUB_CIN);
  SUB_CARRY <= (A      AND NOT(B))
            OR (A      AND (SUB_CARRY(2 DOWNTO 0) & SUB_CIN))
            OR (NOT(B) AND (SUB_CARRY(2 DOWNTO 0) & SUB_CIN));

  S <= ADD_A_B WHEN SEL='1' ELSE SUB_A_B;
END RTL_1;
```

```
ARCHITECTURE RTL_2 OF ADD_SUB IS
BEGIN
 S <= (A + B) WHEN SEL='1' ELSE (A - B);
END RTL_2;
```

Figure 4.3: La transposition des opérateurs arithmétiques.

comportement d'un additionneur soustracteur commandé par le signal *SEL*, avec deux opérandes *A* et *B* et une sortie *S*.

Un résultat possible d'une réécriture de *BEH_1* au niveau booléen est donné par l'architecture *RTL_1*. La transformation des opérateurs arithmétiques en opérateurs strictement booléens fait que toutes les informations qui auraient permis aux phases d'optimisation et de projection structurelle de trouver facilement la meilleure implantation possible ont été inutilement perdues.

L'architecture *RTL_2* montre quand à elle le résultat d'une compilation sans expansion des opérateurs complexes. Elle conserve toute l'information pertinente et nécessaire aux étapes ultérieures de la synthèse. Un modèle formel de VHDL adapté à la synthèse doit donc préserver tous les opérateurs et toutes les fonctions standards.

4.1.4 La macro-expansion des fonctions utilisateur

Les fonctions et les procédures VHDL définies par l'utilisateur doivent être expansées puisque d'une part elles sont inconnues de l'outil de synthèse et d'autre part leur implantation matérielle dépend de leur contexte d'instanciation et des valeurs qui leur sont passées en argument.

La figure 4.4 illustre ce problème en proposant trois appels d'une même fonction avec des paramètres et un contexte différents. La fonction *MY_FUNC* possède trois arguments : *S*, *A* et *B*. Elle renvoie la valeur de *A* lorsque le paramètre de sélection *S* est égal à *'1'* et la valeur de *B* dans le cas contraire.

L'appel à cette fonction par le processus *P1* correspond exactement au comportement matériel d'un multiplexeur avec *SEL* comme signal de commande et les deux signaux *I1* et *I2* comme entrées.

Cette même fonction appelée par le processus *P2* a un comportement non synthétisable car la liste de sensibilité du processus est incomplète.

L'appel à la fonction *MY_FUNC* dans le cas du processus *P3* engendre le comportement d'une bascule avec *SEL* comme signal d'horloge, *I1* comme donnée à mémoriser et *REG_OUT* comme sortie.

```
ENTITY EXAMPLE IS
PORT ( SEL, I1, I2 : IN  BIT;
       O1, O2, O3  : OUT BIT );
END EXAMPLE;

ARCHITECTURE BEH OF EXAMPLE IS

 FUNCTION MY_FUNC( S, A, B : BIT )
   RETURN BIT IS
 BEGIN
   IF ( S = '1' ) THEN RETURN A;
                  ELSE RETURN B;
   END IF;
 END FUNCTION MY_FUNC;

 SIGNAL MUX_OUT : BIT;
 SIGNAL BAD_OUT : BIT;
 SIGNAL REG_OUT : BIT;

BEGIN
   O1 <= MUX_OUT;
   O2 <= BAD_OUT;
   O3 <= REG_OUT;

   P1 : PROCESS( SEL, I1, I2 )
   BEGIN
     MUX_OUT <= MY_FUNC( SEL, I1, I2 );
   END PROCESS P1;

   P2 : PROCESS( SEL )
   BEGIN
     BAD_OUT <= MY_FUNC( SEL, I1, I2 );
   END PROCESS P2;

   P3 : PROCESS( SEL )
   BEGIN
     REG_OUT <= MY_FUNC( SEL, I1, REG_OUT );
   END PROCESS P3;

END BEH;
```

Figure 4.4: Différents appels à une même fonction utilisateur .

Ainsi la fonction *MY_FUNC* peut trouver différentes implantations matérielles suivant la manière dont elle est appelée. Un outil de compilation en vue de la synthèse doit donc macro-expanser tous les appels de fonctions et de procédures définies par l'utilisateur afin d'identifier correctement les registres et les équations de transfert entre ces registres.

La figure 4.5 présente l'expansion d'une fonction utilisateur. La fonction *MIN* prend en entrée *A* et *B* deux valeurs entières et elle renvoie la plus petite des deux. Cette fonction utilise une variable intermédiaire *V* de type booléen. Le résultat de l'expansion de la fonction *MIN* appelée dans le processus *P1* est un processus *P1_EXPAND* équivalent. Il comporte les nouvelles variables locales suivantes :

- deux variables *MIN_A* et *MIN_B* de type entier. Elles représentent les deux arguments *A* et *B* de la fonction *MIN* ;
- une variable *MIN_V* qui correspond à la variable booléenne *V* de la fonction ;
- une variable *MIN_RET* qui contient la valeur de retour de la fonction.

L'expansion de l'appel à la fonction *MIN* est effectuée ainsi :

1. le passage des deux valeurs *I1* et *I2* en paramètre de la fonction est transformé en deux affectations aux variables *MIN_A* et *MIN_B* ;

2. une boucle inconditionnelle *MIN_LOOP* contenant le corps de la fonction est créée afin de modéliser correctement le retour de fonction. Une instruction *RETURN* est ainsi transformée en une instruction *EXIT* qui permet de sortir du corps de la fonction expansée.

3. le corps de la fonction *MIN* est ensuite recopié en veillant cependant à :

   - renommer les variables locales de la fonction : *V* est remplacée par *MIN_V* ;
   - renommer les arguments de la fonction : les variables *A* et *B* sont respectivement remplacées par *MIN_A* et *MIN_B* ;
   - transformer les instructions *RETURN* par :
     - une affectation à la variable contenant la valeur de retour : *MIN_RET* ;
     - une instruction de sortie de boucle *EXIT* pour effectuer la rupture de séquence correspondant à l'exécution de l'instruction *RETURN*.
   - remplacer à l'endroit où la fonction est appelée, la fonction par sa valeur de retour : *MIN_RET*.

```
ENTITY EXAMPLE IS
PORT ( I1, I2 : IN  INTEGER;
       O1,    : OUT INTEGER );
END EXAMPLE;

ARCHITECTURE BEH OF EXAMPLE IS

 FUNCTION MIN( A, B : INTEGER )
 RETURN INTEGER IS
   VARIABLE V : BOOLEAN;
 BEGIN
   V := A < B;
   IF ( V ) THEN RETURN A;
   END IF;
   RETURN B;
 END GET_MIN;

BEGIN

   P1 : PROCESS( I1, I2 )
   BEGIN
     O1 <= MIN( I1, I2 );
   END PROCESS P1;

END BEH;
```

$\Rightarrow$

```
BEGIN

   P1_EXPAND : PROCESS( I1, I2 )
     VARIABLE MIN_A   : INTEGER;
     VARIABLE MIN_B   : INTEGER;
     VARIABLE MIN_V   : BOOLEAN;
     VARIABLE MIN_RET : INTEGER;
   BEGIN
     MIN_A := I1;
     MIN_B := I2;

     MIN_LOOP : LOOP
       MIN_V := MIN_A < MIN_B;
       IF ( MIN_V )
         THEN MIN_RET := MIN_A;
              EXIT MIN_LOOP;
       END IF;
       MIN_RET := MIN_B;
       EXIT MIN_LOOP;
     END LOOP MIN_LOOP;

     O1 <= MIN_RET;
   END PROCESS P1_EXPAND;

END BEH;
```

Figure 4.5: La macro-expansion d'une fonction utilisateur .

L'expansion d'un appel de procédure est effectuée de façon similaire. Il faut néanmoins appliquer un traitement particulier aux arguments déclarés comme étant des sorties : *OUT* ou *INOUT*. La figure 4.6 illustre l'expansion d'une procédure. La procédure *MIN* prend en entrée *A* et *B* deux valeurs entières et elle renvoie la plus petite des deux grâce au paramètre de sortie *S*. Aucune variable n'est créée pour représenter le paramètre *S* puisqu'il s'agit d'un passage par référence et qu'il correspond donc exactement au signal *O1*.

Un autre problème est posé par les procédures et les fonctions dont les arguments ou les valeurs de retour ont une taille indéfinie. La figure 4.7 illustre l'utilisation d'une

```
ENTITY EXAMPLE IS
PORT ( I1, I2 : IN  INTEGER;
       O1,    : OUT INTEGER );
END EXAMPLE;

ARCHITECTURE BEH OF EXAMPLE IS

 PROCEDURE MIN(
   CONSTANT A, B : IN  INTEGER;
   SIGNAL      S : OUT INTEGER )
 IS
   VARIABLE V : BOOLEAN;
 BEGIN
   V := A < B;
   IF ( V ) THEN S <= A;
                 RETURN;
   END IF;
   S <= B;
 END GET_MIN;

BEGIN

   P1 : PROCESS( I1, I2 )
   BEGIN
     MIN( I1, I2, O1 );
   END PROCESS P1;

END BEH;
```

$\Rightarrow$

```
BEGIN

   P2 : PROCESS( I1, I2 )
     VARIABLE MIN_A   : INTEGER;
     VARIABLE MIN_B   : INTEGER;
     VARIABLE MIN_V   : BOOLEAN;
   BEGIN
     MIN_A := I1;
     MIN_B := I2;
     INST_MIN : LOOP
       MIN_V := MIN_A < MIN_B;
       IF ( MIN_V )
         THEN O1 <= MIN_A;
              EXIT;
       END IF;
       O1 <= MIN_B;
       EXIT;
     END LOOP INST_MIN;

   END PROCESS P2;

END BEH;
```

Figure 4.6: La macro-expansion d'une procédure utilisateur .

fonction *ADD* dont les paramètres et la valeur de retour ont une taille variable. La taille des arguments *A* et *B* ainsi que du vecteur de retour dépend uniquement de la taille des opérandes, elle peut être facilement calculée lors de la macro-expansion de la fonction.

```
ENTITY EXAMPLE IS
PORT ( I1, I2 : IN  BIT_VECTOR(3 DOWNTO 0);
       O1,    : OUT BIT_VECTOR(3 DOWNTO 0) );
END EXAMPLE;
ARCHITECTURE BEH OF EXAMPLE IS
 FUNCTION ADD( A, B : BIT_VECTOR )
 RETURN BIT_VECTOR IS
   VARIABLE SUM   : BIT_VECTOR(A'RANGE);
   VARIABLE CARRY : BIT;
 BEGIN
   CARRY := '0';
   FOR I IN A'RANGE
   LOOP
     SUM(I) := A(I) XOR B(I) XOR CARRY;
     CARRY  := ( A(I) AND B(I)  )
            OR ( A(I) AND CARRY )
    OR ( B(I) AND CARRY );
   END LOOP;
   RETURN SUM;
 END ADD;
BEGIN
   P1 : PROCESS( I1, I2 )
   BEGIN
     O1 <= ADD( I1, I2 );
   END PROCESS P1;
END BEH;
```

Figure 4.7: Une fonction avec une valeur de retour et des paramètres de taille variable .

Il n'est donc pas indispensable qu'un modèle formel adapté à la synthèse permette de représenter les fonctions et les procédures définies par l'utilisateur puisque, comme nous venons de le voir, elles peuvent et doivent être expansées.

4.1.5 La structure de la description initiale

Les informations liées à la structure de la description VHDL doivent être conservées car elles peuvent guider de façon efficace la phase d'optimisation et de projection structurelle. En effet, le concepteur qui décrit le comportement d'un circuit en vue d'entrer dans un flot de synthèse peut par exemple :

- utiliser des variables ou des signaux intermédiaires pour spécifier un partage possible des ressources matérielles ;
- définir un certain partitionnement du circuit en isolant certaines fonctionnalités matérielles dans des processus.

```
ENTITY EXAMPLE IS PORT (
  A, B, C  : IN  SIGNED(3 DOWNTO 0);
  S1, S2   : OUT SIGNED(3 DOWNTO 0) );
END EXAMPLE;

ARCHITECTURE BEH OF EXAMPLE IS
BEGIN
  P1 : PROCESS( A, B, C )
    VARIABLE V : SIGNED(3 DOWNTO 0)
  BEGIN
    V  := A + B;
    S1 <= V;
    S2 <= C + V;
  END PROCESS P1;
END BEH;
```

```
ENTITY EXAMPLE IS PORT (
  A, B, C  : IN  SIGNED(3 DOWNTO 0);
  S1, S2   : OUT SIGNED(3 DOWNTO 0) );
END EXAMPLE;

ARCHITECTURE RES_RTL OF EXAMPLE IS
BEGIN
    S1 <= A + B;
    S2 <= C + A + B;
END RES_RTL;
```

Figure 4.8: Les informations structurelles.

La figure 4.8 illustre ce besoin. L'architecture *BEH* de gauche décrit le comportement de deux additionneurs en série. Une variable intermédiaire *V* à été introduite afin de partager le calcul de l'addition des deux signaux A et B, puisqu'elle apparaît à la fois dans l'affectation du signal *S1* et dans celle du signal *S2*. Le résultat d'une compilation qui ne préserve pas les variables intermédiaires spécifiées par le concepteur, peut être observé sur l'architecture *RES_RTL*.

4.2 Présentation du modèle formel *XVPN*

Un grand nombre de modèles formels de VHDL ont été développés afin de représenter la sémantique de simulation du langage [KB95]. Le comportement d'un programme VHDL peut en particulier être complètement modélisé sous forme d'un réseau de Pétri [OC93, MB96, Enc95, Baw96, Döh94].

Un modèle *VPN* (VHDL Petri Nets) a été proposé par Emmanuelle Encrenaz [Enc95] et Rajesh Bawa [Baw96]. Il a été conçu pour des outils de preuve formelle et notre objectif est de l'étendre pour qu'il puisse répondre complètement aux besoins de la synthèse.

4.2.1 Un modèle adapté aux techniques de preuve formelle

Le modèle formel *VPN* permet de représenter complètement la sémantique de simulation de VHDL sous la forme d'un réseau de Pétri. Il est constitué de trois parties qui sont illustrées sur la figure 4.9.

La première partie appelée *partie contrôle*, assure la représentation du graphe de contrôle de chacun des processus VHDL. Chaque processus d'une description VHDL élaborée peut être vu comme un automate pour lequel un changement d'état induit des transformations sur les variables internes ainsi que des modifications sur les pilotes des signaux. Un tel automate peut être représenté par un sous réseau de Pétri rebouclé sur lui même, ne comportant qu'un seul jeton correspondant au compteur de programme. Ainsi le réseau de Pétri représentant l'ensemble des processus VHDL est constitué d'un ensemble de composantes connexes disjointes deux à deux, où chaque composante connexe représente un processus VHDL.

La seconde partie appelée *partie données* modélise les mécanismes de mise à jour des pilotes des signaux et des valeurs effectives des variables. Elle permet en particulier de calculer à tout instant la valeur de l'attribut *EVENT* pour tous les signaux de la description.

La troisième partie appelée *séquenceur* est un unique réseau de Pétri circulaire constitué de trois places et de trois transitions. Chaque place de ce réseau correspond à une des phases du cycle de simulation et chaque transition représente le passage d'une phase à une autre. Cette dernière partie modélise ainsi le coeur du simulateur.

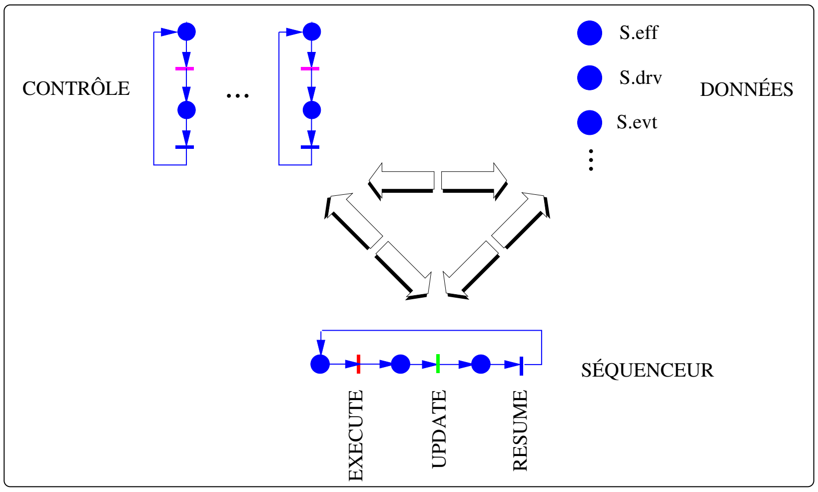

Figure 4.9: Une modélisation formelle de VHDL.

Des techniques de réduction du graphe de contrôle des processus ont été définies [Enc95, Baw96]. Leur application permet d'obtenir une structure minimale pour chacun des processus VHDL.

Cependant ce modèle doit être modifié pour être utilisable en synthèse :

- seule la partie contrôle du réseau de Pétri nous intéresse et il convient donc d'enlever tout ce qui concerne d'une part la modélisation du cycle de simulation VHDL et d'autre part la modélisation des modifications des pilotes des signaux et des valeurs effectives des variables ;
- ce modèle ne permet pas de manipuler des opérateurs autres que les opérateurs booléens. Il convient donc de l'enrichir afin de pouvoir représenter tous les opérateurs prédéfinis pour la synthèse et en particulier les opérateurs arithmétiques ;
- ce modèle se situe au niveau booléen. Il ne supporte que le type bit et ne conserve pas la notion de vecteur de bits. En effet toute affectation à un vecteur de bits dans la description VHDL initiale est expansée en un ensemble d'affectations de niveau bit pour chacune des composantes du vecteur. Il est donc indispensable d'étendre ce modèle afin de conserver la notion de vecteur si précieuse dans le cadre de la synthèse.

L'extension de ce modèle en vue de la synthèse impose de nouvelles contraintes qui doivent être respectées par les algorithmes de réduction. Il est donc nécessaire d'élaborer de nouvelles techniques de réduction spécifiques à la synthèse.

4.2.2 Un modèle adapté à la synthèse

Dans les paragraphes suivants nous définissons un modèle formel de VHDL adapté à la synthèse. Ce modèle appelé *XVPN* (eXtended VHDL Petri Nets) est une extension du modèle *VPN*. Il est constitué de deux parties : une première partie qui correspond aux symboles manipulés et une seconde qui représente le graphe de contrôle de chaque processus de la description à modéliser.

La représentation des symboles VHDL

Comme nous l'avons expliqué précédemment, tous les types VHDL de la description initiale doivent être ramenés au niveau bit ou vecteurs de bits. Cependant afin de ne perdre aucune des informations présentes dans la description initiale, l'ensemble des valeurs que peut prendre chacun des symboles utilisés doit être conservé.

Soit $Data$ l'ensemble des symboles associés au réseau de Pétri. À chaque symbole VHDL, que ce soit un signal, une constante ou bien une variable, correspond un élément dans l'ensemble $Data$. Cet ensemble $Data$ se décompose en trois sous ensembles disjoints $Data = S \cup V \cup C$:

- S est l'ensemble de tous les signaux s définis dans la description. Chaque élément s est défini par un quintuplet $(s_{eff}, s_{evt}, s_{drv}, s_{init}, s_{type})$ où : s_{eff} représente la valeur courante du signal, s_{evt} la valeur de son attribut *EVENT*, s_{drv} la valeur de son pilote, s_{init} sa valeur initiale et s_{type} son type, c'est à dire l'ensemble des valeurs qu'il peut prendre ;

- V est l'ensemble de toutes les variables v définies dans un processus p. Chaque élément v est défini par le quadruplet $(v_{val}, v_{init}, v_{type}, p)$ où : v_{val} représente la valeur courante de la variable v, v_{init} sa valeur initiale, v_{type} son type et p le processus auquel elle appartient ;

- C est l'ensemble de toutes les constantes c définies dans la description. Chaque élément c est défini par un couple (c_{val}, c_{type}) où : c_{val} est la valeur de la constante c et c_{type} son type.

La représentation du graphe de contrôle

Comme les outils de synthèse traitent chaque processus indépendamment, on ne cherche pas à représenter explicitement les interactions entre processus dans le réseau de Pétri. Chacun des processus est décrit par un sous réseau constituant une composante connexe et composé de transitions reliées par des places. Chaque transition correspond à un ensemble d'instructions VHDL séquentielles.

Notre modèle se doit de conserver la sémantique de simulation du langage. Il doit donc en particulier permettre de représenter les deux phases du cycle de simulation VHDL qui interviennent dans l'exécution d'un processus. C'est pourquoi chaque sous réseau de Pétri représentant un processus comporte deux types de transition :

- les transitions T_{res} qui correspondent à la phase *RESUME* du cycle de simulation. Elles représentent les points de suspension d'un processus, c'est à dire les instructions *WAIT* ;

- les transitions de type T_{exe} qui sont associées à la phase *EXECUTE* du cycle de simulation. Elles décrivent l'exécution séquentielles des instructions autres que les points de suspension, à l'intérieur d'un processus.

Les transitions T_{exe} possèdent éventuellement une condition de franchissement appelée aussi condition de garde. La règle suivante est toujours respectée : il n'y a pas plus d'une affectation à un même symbole sur une transition T_{exe}. Cette contrainte permet de simplifier considérablement les techniques de réduction qui seront exposées au chapitre suivant. Une transition T_{exe} ne peut être franchie que lorsque sa condition de garde est vraie. Son franchissement entraîne alors une modification de l'ensemble des données $Data$ avec exactement la même sémantique que l'exécution d'une série d'affectations VHDL séquentielles.

Les transitions T_{res} permettent de modéliser les points de suspension des processus. Elles comportent l'ensemble des évènements qui pourront réveiller un processus dans la phase *RESUME*. Un ensemble de signaux correspondant à la liste de sensibilité d'un point de suspension leur est donc associé. Une transition de ce type ne peut être franchie que lors d'un évènement sur un des signaux appartenant à sa liste de sensibilité.

On notera par la suite $\mathcal{B} = \{0, 1\}$ l'ensemble booléen et $\mathcal{P}(E)$ l'ensemble des parties d'un ensemble E.

Tout symbole d_i de l'ensemble $Data$ prend ses valeurs dans $(\mathcal{S})^{n_i}$. On notera $\mathcal{F}(Data)$ l'ensemble de toutes les fonctions f_x permettant de calculer la valeur d'un symbole d_x à partir des valeurs d'un nombre quelconque m de symboles d_i : $d_1 \times \cdots \times d_m \stackrel{f_x}{\to} dx$, $(\mathcal{S})^{n_1} \times \cdots \times (\mathcal{S})^{n_m} \stackrel{f_x}{\to} (\mathcal{S})^{n_x}$ où n_i représente le nombre de bits du symbole d_i.

On notera $\mathcal{G}(Data)$ l'ensemble de toutes les fonctions g_x à valeur dans $\mathcal{B}$ permettant de calculer la valeur d'une variable booléenne x à partir d'un nombre m quelconque de symboles d_i : $d_1 \times \cdots \times d_m \stackrel{g_x}{\to} x$, $(\mathcal{S})^{n_1} \times \cdots \times (\mathcal{S})^{n_m} \stackrel{g_x}{\to} \mathcal{B}$ où n_i représente le nombre de bits du symbole d_i.

Les fonctions f_x et g_x sont représentées en pratique par des expressions contenant des symboles d_i et des opérateurs booléens ou arithmétiques.

Le réseau de Pétri PN représentant les graphes de contrôle des processus est associé à l'ensemble des données $Data$ et est noté PN_{Data}.

Il est défini par $PN_{Data} =< P, T, ArcPlace, ArcTrans, G, W, Trf, Asg, M0 >$ avec :

- P est l'ensemble de toutes les places ;
- T est l'ensemble de toutes les transitions avec $T = T_{exe} \cup T_{res}$ sachant que T_{exe} et T_{res} sont deux ensembles disjoints. Une transition possède au plus un prédécesseur et un successeur ;
- $ArcPlace \ : P \times T \to \mathcal{B}$ est la fonction définie par : $ArcPlace(p, t) = 1$ si et seulement si il existe un arc de p vers t ;
- $ArcTrans \ : T \times P \to \mathcal{B}$ est la fonction définie par : $ArcTrans(t, p) = 1$ si et seulement si il existe un arc de t vers p ;
- $G \ : T_{exe} \to \mathcal{G}(Data)$ est la fonction qui associe à une transition t de T_{exe} la fonction $g_t \in \mathcal{G}(Data)$. Cette fonction représente la condition de franchissement de la transition t ;
- $W \ : T_{res} \to \mathcal{P}(Data)$ est la fonction qui associe à chaque transition de T_{res} un ensemble de signaux s. Cet ensemble noté $W(t)$ correspond à la liste de sensibilité de t.
- $Trf \ : T_{exe} \to [Data \times \mathcal{F}(Data)]^n$ est la fonction qui associe à une transition de T_{exe} une liste ordonnée de couples (d_i, f_i) avec $d_i \in Data$ et $f_i \in \mathcal{F}(Data)$. La fonction f_i définit la nouvelle valeur affectée au symbole d_i lors de l'exécution de la transition t. Toutes les affectations spécifiées par $Trf(t)$ sont effectuées séquentiellement lors du franchissement de t ;

- $Asg\ : T \rightarrow \mathcal{P}(Data)$ où $Asg(t) = \{d_i, d_i \in Data\}$ est la fonction qui associe à chaque transition l'ensemble de tous les symboles d_i modifiés lors du franchissement de la transition t ;
- M0 est le marquage initial du réseau de Pétri. Chacun des sous-graphes connexes représentant les différents processus VHDL possède un unique jeton. Ainsi, le marquage initial définit l'état initial de chaque processus.

À la définition formelle de notre modèle de représentation d'une description VHDL vient s'ajouter un ensemble de propriétés et de règles de construction.

Une transition possède toujours une unique place en entrée et une unique place en sortie. Il n'existe donc qu'une seule construction possible pour les transitions, et elle est illustrée sur la figure 4.10.

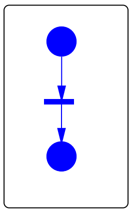

Figure 4.10: Le seul type de construction possible d'une transition.

La construction d'une place est soumise aux règles suivantes :

- une place a toujours au moins une transition en sortie ;
- une place qui possède plus d'une transition en entrée possède une unique transition en sortie ;
- une place qui possède plus d'une transition en sortie possède au plus une transition en entrée.

Les différents types de constructions possibles mettant en jeu des places sont illustrés sur la figure 4.11.

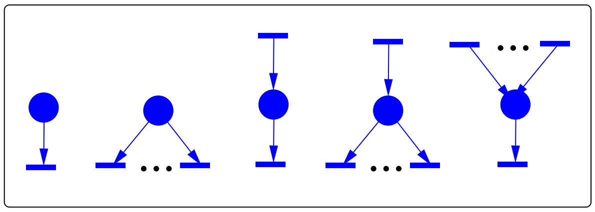

Figure 4.11: Les cinq types de constructions possibles d'une place.

La figure 4.12 présente un sous graphe où une place possède plus d'une transition en sortie. Dans un tel cas, les n transitions t_i successeurs ont les deux propriétés suivantes :

- chaque transition t_i est de type *EXECUTE* et possède une condition de franchissement c_i ;
- toutes les conditions c_i sont mutuellement exclusives et leur somme est égale à 1.

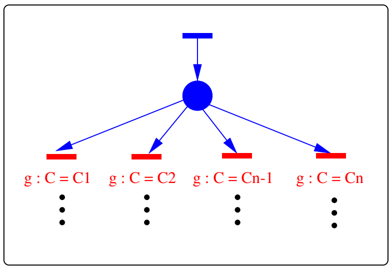

Figure 4.12: L'exclusion mutuelle des conditions de franchissement.

Une dernière règle de construction impose que si une transition t possède une condition de franchissement différente de 1, alors la place qui est entrée de t a obligatoirement plus d'une transition en sortie.

Le modèle *XVPN* ainsi défini et aussi contraint, constitue un sous ensemble extrêmement réduit de la famille des réseaux de Pétri colorés et interprétés. Dans la suite de la thèse nous emploierons donc de préférence le terme de "graphes de contrôle" plutôt que le terme de "réseaux de Pétri".

4.3 La compilation d'une description VHDL vers *XVPN*

La phase de compilation doit tout d'abord construire en mémoire l'ensemble des types et l'ensemble des symboles (les variables, les signaux et les constantes) qui sont manipulés dans la description VHDL. Elle doit ensuite procéder à la modélisation de chacun des processus sous forme d'un sous-graphe.

4.3.1 La traduction des types et la représentation des symboles

Comme nous l'avons souligné précédemment la phase de compilation doit traduire tous les types entiers et tous les types énumérés en des types bit ou vecteur de bits. Tous les types complexes sont donc préalablement encodés.

La figure 4.13 illustre la transformation sur notre modèle d'un type énuméré, d'un type entier et d'un vecteur de bits. La figure 4.14 montre quand à elle, le résultat de la construction de l'ensemble $Data$ à partir des symboles déclarés dans une description VHDL.

Types VHDL	Types encodés

`TYPE STATE_TYPE IS ( S0, S1, S2 );`
`TYPE UNSIGNED_BYTE IS INTEGER RANGE 0 TO 255;`
`TYPE BIT_VEC_3_0 IS BIT_VECTOR(3 DOWNTO 0);`
	`State_type = { "00", "01", "11" }`
	`Unsigned_byte = { "00000000", ..., "11111111" }`
	`Bit_Vec_3_0 = { "0000", ..., "1111" }`

Figure 4.13: La traduction des types sur notre modèle.

Déclarations VHDL	Résultat de la traduction

`SIGNAL A~: STATE_TYPE~:= E1;`
`SIGNAL B~: UNSIGNED_BYTE;`
`SIGNAL C~: BIT_VEC_3_0~:= "1111";`
	$A[1,0] = (a[1,0]_{eff}, a[1,0]_{evt}, a[1,0]_{drv}, "01", State\_type)$
	$B[7,0] = (b[7,0]_{eff}, b[7,0]_{evt}, b[7,0]_{drv}, "00000000", Unsigned\_Byte)$
	$C[3,0] = (c[3,0]_{eff}, c[7,0]_{evt}, c[7,0]_{drv}, "1111", Bit\_Vec\_3\_0)$

Figure 4.14: La traduction des déclarations sur notre modèle.

4.3.2 La compilation des processus VHDL

La compilation d'un processus VHDL consiste à interconnecter un ensemble de places et de transitions reproduisant son comportement.

Une place contenant un jeton est tout d'abord créée afin de représenter l'état initial du processus avant son initialisation. Cette place est ensuite reliée à une transition de type *EXECUTE*. Puis toutes les instructions séquentielles sont traduites en un ensemble de places et de transitions reproduisant leur comportement. Enfin, on crée un arc reliant la dernière instruction à la première place assurant ainsi le rebouclage du processus sur lui même.

La figure 4.15 représente la structure générale d'un processus après avoir été traduit sur notre modèle. La place *P1* et la transition *T1* représentent le début du processus. La transition *Tn* représente quand à elle la dernière transition de la dernière instruction.

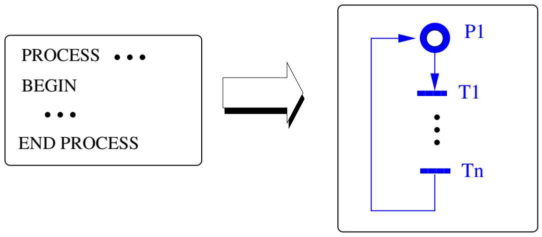

Figure 4.15: La modélisation d'un processus.

Les affectations séquentielles

Chaque affectation d'une variable ou d'un signal $d \in Data$ est modélisée par une unique transition t appartenant à T_{exe}.

Un couple $(d \in Data, f \in \mathcal{F}(Data))$ est placé dans $Trf(t)$ correspondant à la transformation à effectuer sur d lors du franchissement de la transition t. Le symbole d doit ensuite être ajouté à l'ensemble $Asg(t)$.

Après cette première phase de compilation une transition ne possède qu'une seule affectation. Dans le chapitre 5 nous décrivons les mécanismes de réduction du graphe qui permettent d'identifier en particulier les éléments mémorisants. À l'issue de cette phase de réduction, une transition possède en général plusieurs affectations séquentielles et c'est pourquoi la liste des symboles affectés est conservée dans l'ensemble $Asg(t)$.

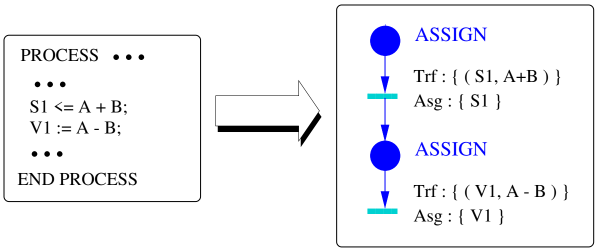

Figure 4.16: Les affectation séquentielles.

La figure 4.16 illustre la transformation sur notre modèle de deux affectations séquentielles qui propagent tout d'abord la valeur de l'expression **A+B** sur le signal **S1**, puis la valeur **A-B** sur la variable **V1**.

Les instructions conditionnelles

Les instructions conditionnelles *IF* et *CASE* permettent de sélectionner différents chemins d'exécution en fonction d'une ou plusieurs conditions booléennes. La compilation de ces instructions est effectuée en créant pour chacune des alternatives, une transition de type T_{exe} munie d'une condition de garde. Chacune de ces transitions représente le point de départ d'un chemin d'exécution. Elles sont toutes connectées à une place initiale qui sert d'aiguillage sur un des chemin d'exécution, en fonction des conditions de garde. Tous ces chemins convergent vers une unique place qui correspond directement à l'instruction *END IF* ou à l'instruction *END CASE*.

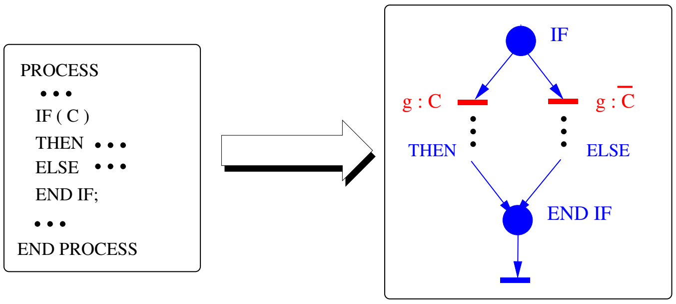

Figure 4.17: L'instruction *IF*.

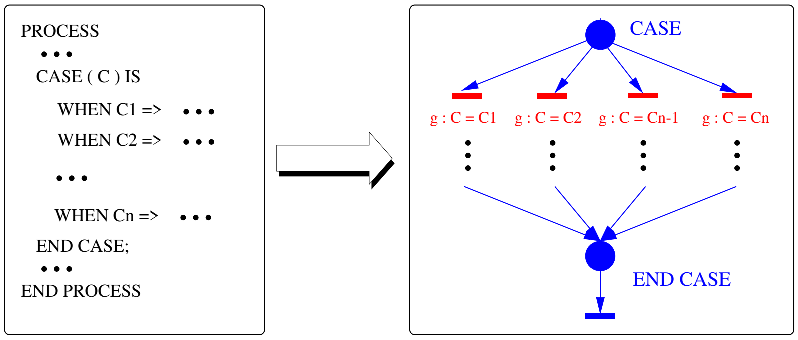

Figure 4.18: L'instruction *CASE*.

La figure 4.17 illustre la transformation de l'instruction *IF*. La figure 4.18 illustre la transformation d'une instruction *CASE*. Une transition est construite à chaque spécification d'une nouvelle branche *WHEN* et donc d'une nouvelle valeur C_i. Lors de l'utilisation du mot clé *OTHERS*, une transition spéciale est créée en accord avec la sémantique du langage : sa condition de franchissement est calculée à partir du complément de toutes les valeurs qui ont été spécifiées sur les différentes branches *WHEN* du *CASE*.

Les instructions de boucle

L'instruction de boucle *LOOP* est compilée en un sous-graphe reproduisant exactement sa structure. Tout d'abord une place p_{begin} et une transition t_{begin} représentant le début de la boucle sont créées. Le corps de la boucle vient ensuite s'accrocher sur cette transition t_{begin}. La transition t_{last} qui correspond à la dernière instruction VHDL du corps de boucle est connectée à la place p_{begin} représentant le début de la boucle. Enfin une place p_{end} et une transition t_{end} sont ajoutées afin de modéliser la sortie de la boucle et l'enchaînement en séquence des instructions suivantes. La figure 4.19 illustre la transformation de l'instruction *LOOP*.

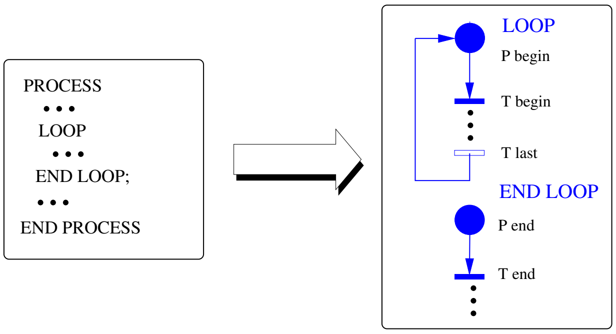

Figure 4.19: L'instruction *LOOP*.

Une boucle de type *FOR* est compilée en un sous réseau de Pétri constitué de quatre parties :

- la première assure l'initialisation de l'indice de boucle ;
- la seconde effectue sur l'indice un test de fin de boucle ;
- la suivante modélise le comportement du corps de la boucle ;
- la dernière permet d'incrémenter ou de décrémenter l'indice de boucle.

La figure 4.20 illustre la transformation d'une boucle de type *FOR* et indique précisément où se situent les différentes parties.

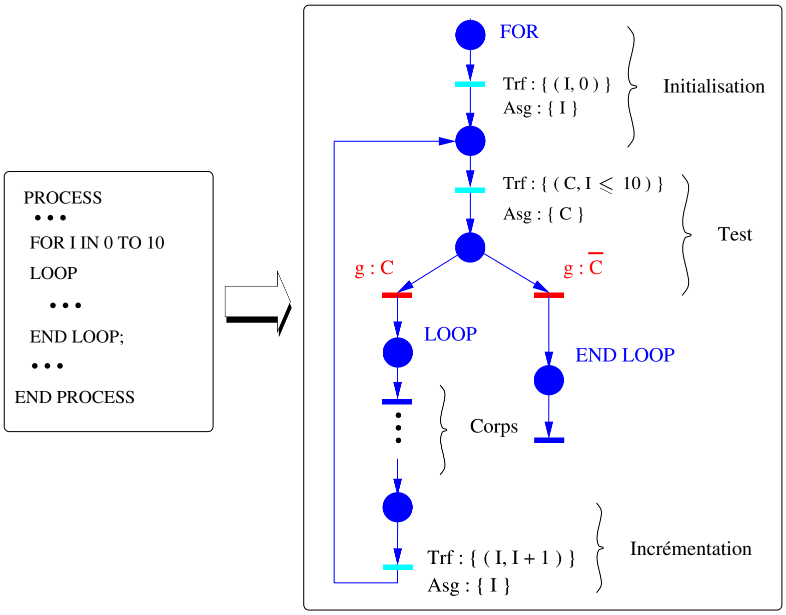

Figure 4.20: L'instruction *FOR*.

La boucle de type *WHILE* est compilée de façon similaire. Elle n'est cependant constituée que de deux parties distinctes : l'une visant à tester une condition de fin de boucle et une autre modélisant le corps de la boucle. La figure 4.21 illustre la compilation d'une boucle de type *WHILE*.

Les instructions de saut de boucle *NEXT* et *EXIT* sont modélisées simplement par une transition connectée soit à la place p_{begin} de début de boucle, dans le cas de l'instruction *NEXT*, soit à la place p_{end} de fin de boucle. Une condition de franchissement peut éventuellement apparaître sur cette transition si une clause *WHEN* spécifie une condition de saut particulière. Les deux figure 4.22 et 4.23 illustrent cette transformation.

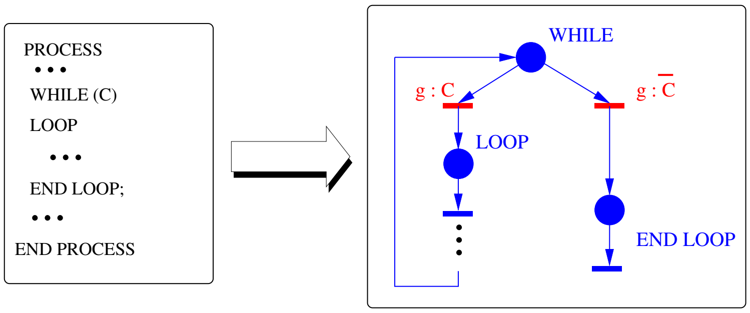

Figure 4.21: L'instruction *WHILE*.

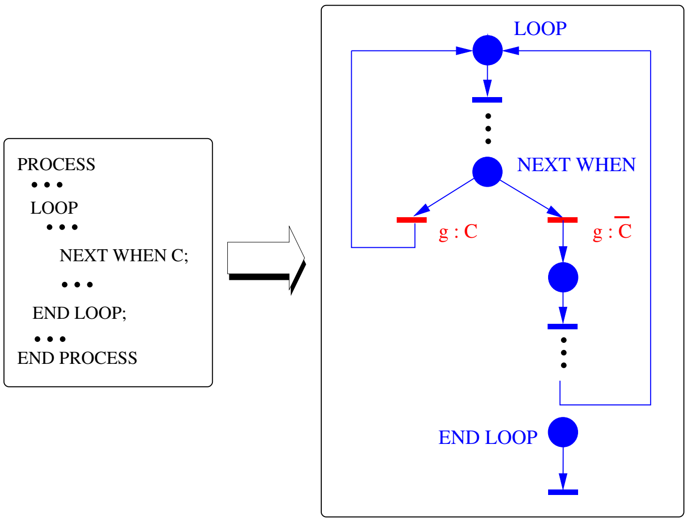

Figure 4.22: L'instruction *NEXT*.

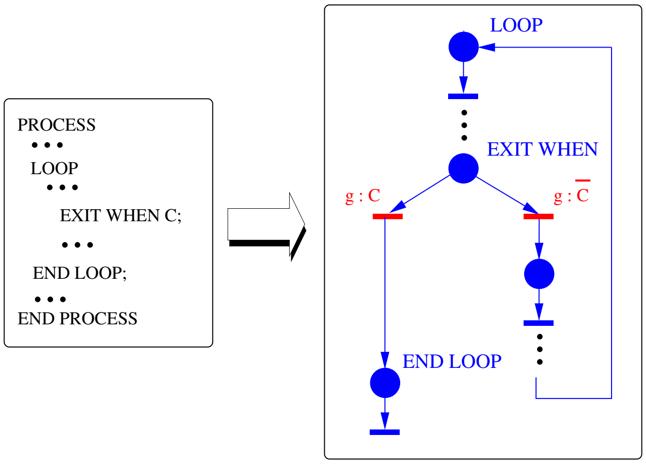

Figure 4.23: L'instruction *EXIT*.

Les instructions de suspension

Une instruction de suspension est transformée en une transition t de type T_{res}. Un ensemble w de signaux est associé à cette transition. Il correspond à la liste de sensibilité du point de suspension. Un processus qui ne possède pas un point de suspension explicite hérite systématiquement d'une instruction *WAIT* correspondant à l'attente d'un évènement sur sa liste de sensibilité. Ce point de suspension est situé en dernière position dans le corps du processus. La figure 4.24 illustre la compilation d'une instruction *WAIT*. La figure 4.25 illustre la compilation d'une instruction *WAIT* enrichie d'une clause *UNTIL*.

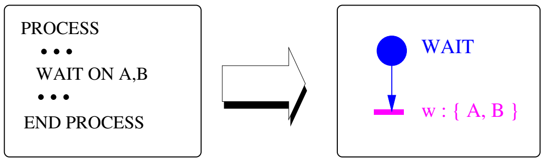

Figure 4.24: L'instruction *WAIT*.

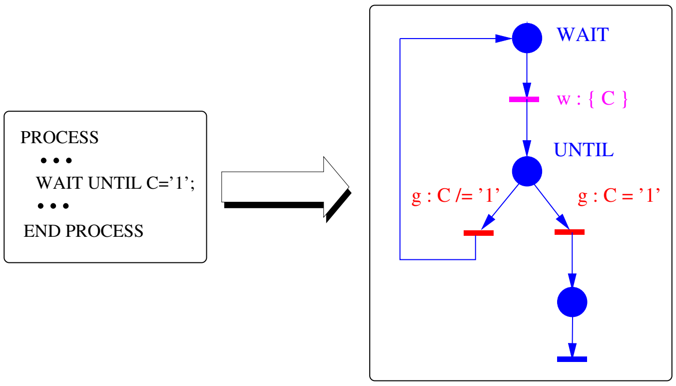

Figure 4.25: L'instruction *WAIT* enrichie d'une clause *UNTIL* .

Les conditions complexes

Certaines instructions conditionnelles telles que le *IF* ou le *CASE* peuvent nécessiter le calcul de conditions de franchissement complexes. Des variables intermédiaires peuvent être introduites pour faciliter le travail des phases ultérieures de la synthèse.

Un critère simple consiste à ajouter une variable lorsque la condition met en jeux plusieurs symboles, ou bien lorsqu'elle utilise des opérateurs arithmétiques ou des opérateurs relationnels. Il existe cependant une exception : lors de l'utilisation d'un opérateur d'égalité entre un symbole de niveau bit et une constante, aucune variable n'est créée.

La figure 4.26 montre tout d'abord la réécriture d'un processus VHDL utilisant l'instruction *IF* avec une condition booléenne utilisant un opérateur arithmétique, puis sa compilation en réseau de Pétri.

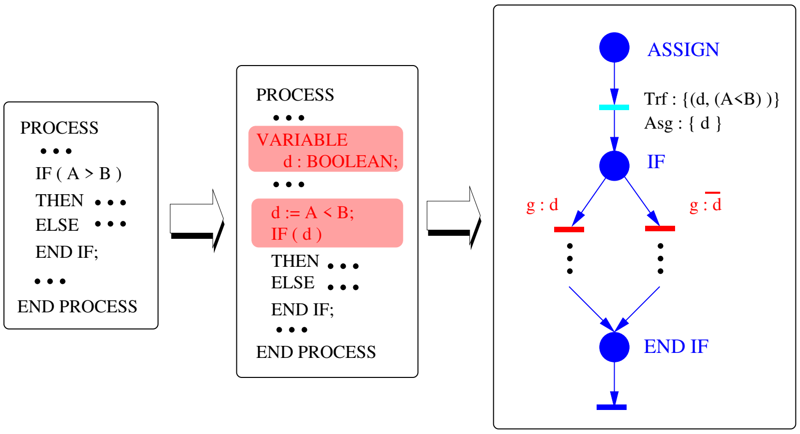

Figure 4.26: L'introduction de variables intermédiaires.

Un exemple complet

Le résultat des différentes transformations effectuées lors de la compilation est illustré sur la figure 4.27. Il s'agit de la description VHDL d'une bascule D qui mémorise sur front montant de l'horloge *CK* la valeur du signal d'entrée A. Le contenu du registre V est recopié sur la sortie T.

Les différentes déclarations de cette description sont traduites sur notre modèle de la façon suivante :

- le type $Bit = \{'0', '1'\}$;
- l'ensemble des symboles $Data = \{CK, A, T\}$, avec
  - $CK = (ck_{eff}, ck_{evt}, ck_{drv}, '0', Bit)$;
  - $A = (a_{eff}, a_{evt}, a_{drv}, '0', Bit)$;
  - $T = (t_{eff}, t_{evt}, t_{drv}, '0', Bit)$;
  - $V = (v_{val}, '0', Bit, P1)$.

Le réseau de Pétri représentant l'unique processus de la description a été construit en assemblant simplement l'ensemble des sous réseaux de chacune des instructions VHDL.

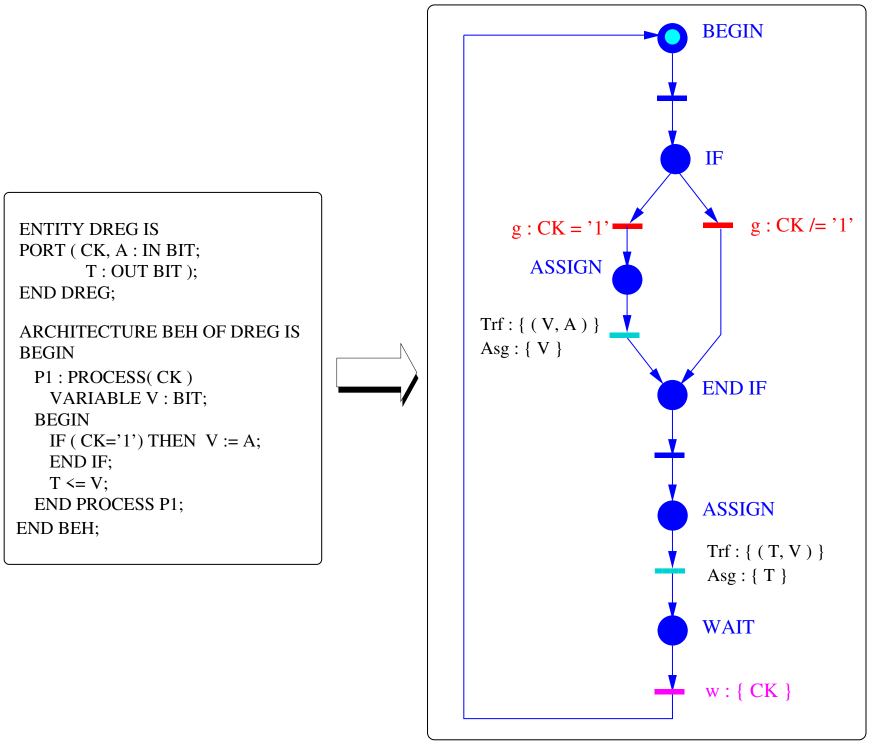

Figure 4.27: Compilation de la description VHDL d'une bascule D .

4.4 La phase d'initialisation de notre modèle

Comme nous l'avons vu au chapitre 2 la simulation d'une description VHDL débute par une phase d'initialisation. Dans les paragraphes suivants nous montrons que certains aspects de cette phase doivent absolument être pris en compte lors de la synthèse. Nous expliquons ensuite comment prendre en compte cette phase sur notre modèle.

4.4.1 L'influence de la phase d'initialisation sur la synthèse

La phase d'initialisation de la simulation définie dans le manuel de référence du langage VHDL fournie des informations indispensables pour obtenir un résultat de synthèse qui soit en accord avec la sémantique du langage. Néanmoins comme nous l'avons évoqué au chapitre 2 tous les mécanismes de cette phase ne peuvent pas être implantés en matériel.

La phase d'initialisation nous conduit à distinguer deux types de comportement pour un processus donné :

- son comportement initial lors de la phase d'initialisation. Il correspond à l'exécution des instructions séquentielles situées entre le mot clé *BEGIN* et le premier point de suspension rencontré à l'intérieur du processus.

- son comportement permanent, c'est à dire le comportement qu'il adopte après la phase d'initialisation. Il correspond à l'exécution des instructions séquentielles situées entre deux points de suspension accessibles.

Par exemple le comportement initial du processus *P1* de la figure 4.28 consiste à donner la valeur $'1'$ au signal S avant de s'arrêter sur un point de suspension. Le comportement permanent de ce même processus, consiste à recopier la valeur de l'entrée $I0$ dans S sur front montant de l'horloge CK.

```
ENTITY EXAMPLE IS
PORT ( CK : IN  BIT;
       I0 : IN  BIT;
       S  : OUT BIT );
END EXAMPLE;

ARCHITECTURE BEH_1 OF EXAMPLE IS
BEGIN
  P1 : PROCESS
  BEGIN
    S <= '1';
    LOOP
      WAIT UNTIL CK='1';
      S <= I0;
    END LOOP;
  END PROCESS P1;
END BEH_1;
```

Figure 4.28: Les deux types de comportement d'un processus VHDL.

L'analyse de ces deux comportements ne donne pas le même type d'information :

- l'analyse du comportement initial permet de déterminer les valeurs initiales des symboles de la description. Elle permet aussi d'isoler les instructions séquentielles qui n'interviennent pas dans le fonctionnement permanent des processus ;
- l'analyse du comportement permanent permet d'identifier les registres et de construire les équations de transfert ;

Sur notre exemple l'analyse du comportement initial du processus *P1* permet de déterminer la valeur initiale du signal *S*. Elle permet également d'isoler la première instruction d'affectation de *S* puisqu'elle n'est exécutée que pendant la phase d'initialisation. L'analyse de son comportement permanent permet d'identifier le registre *S* et d'en trouver toutes les propriétés.

La valeur initiale des symboles VHDL

Tous les symboles possèdent une valeur par défaut qui est définie par la norme comme étant la valeur la plus à gauche de son type. Cette valeur peut être redéfinie avec une valeur par défaut explicite.

Dans le cas du signal *S* de la figure 4.29 la valeur la plus à gauche du type *STD_LOGIC* vaut $'U'$. Sa valeur par défaut est explicitement redéfinie à $'0'$. L'exécution initiale du processus *P1* va modifier la valeur initiale du signal *S* et la positionner à $'1'$.

Du point de vue de la synthèse il est indispensable de considérer que la valeur initiale du signal *S* n'est pas $'U'$ ou $'0'$ mais bien la valeur $'1'$.

```
ARCHITECTURE BEH OF EXAMPLE IS
  SIGNAL S : STD_LOGIC := '0';
BEGIN
  P1 : PROCESS
  BEGIN
    S <= '1';
    WAIT;
  END PROCESS P1;
END BEH;
```

Figure 4.29: L'initialisation d'un signal.

Notons que la phase d'initialisation de la simulation VHDL calcule les valeurs initiales de tous les symboles de la description, mais toutes ces valeurs ne peuvent pas forcément être exploitées en synthèse.

Sur la figure 4.30 le signal *S* est initialisé à $'1'$. Le comportement permanent du processus *P2* consiste à donner au signal *S* la valeur du *OU* logique entre *A* et *B*. Le signal *S* correspond donc au niveau matériel à la sortie d'une porte combinatoire. La valeur initiale du signal *S* ne peut donc pas être prise en compte. En effet forcer la sortie d'une porte combinatoire à prendre une valeur à l'initialisation n'est pas possible au niveau matériel.

```
ARCHITECTURE BEH OF EXAMPLE IS
  SIGNAL A, B : BIT;
  SIGNAL S    : BIT := '1';
BEGIN
  P2 : PROCESS
  BEGIN
    WAIT ON A, B;
    S <= A OR B;
  END PROCESS P2;
END BEH;
```

Figure 4.30: L'initialisation d'un signal combinatoire.

Par contre comme nous l'avons évoqué au chapitre 2, certaines technologies cibles comme les *FPGA* possèdent un mécanisme d'initialisation des registres. La prise en compte de la phase d'initialisation de la simulation permet de calculer la valeur initiale des différents registres. Ainsi dans le cas du processus *P3* de la figure 4.31 la synthèse doit être en mesure de calculer la valeur du registre *S* après initialisation, c'est à dire la valeur $'1'$.

La phase d'initialisation de la simulation permet également de détecter les symboles qui n'ont pas été déclarés comme des constantes mais qui pourtant gardent toujours la même valeur. Si un symbole n'est jamais affecté la synthèse doit alors le considérer comme une constante. Cette règle simple ne permet pas de détecter toutes les constantes : si un symbole est affecté pendant la phase d'initialisation et ne l'est plus jamais après, il doit aussi être considéré comme une constante.

Le processus *P1* de la figure 4.32 donne la valeur $'1'$ au signal *C* lors de la phase d'initialisation de la simulation. Il reste ensuite bloqué sans jamais plus pouvoir se ré-

```
ENTITY EXAMPLE IS
PORT ( CK : IN  BIT;
       I0 : IN  BIT;
       Q  : OUT BIT );
END EXAMPLE;

ARCHITECTURE BEH OF EXAMPLE IS
  SIGNAL S : BIT;
BEGIN
  P3 : PROCESS
  BEGIN
    S <= '1';
    LOOP
      WAIT UNTIL CK='1';
      S <= I0;
    END LOOP;
  END PROCESS P3;
  Q <= S;
END BEH;
```

Figure 4.31: L'initialisation d'un registre.

veiller. Ainsi le signal *C* doit être considéré comme une constante ayant la valeur $'1'$ même si il n'a pas été déclaré comme tel.

La détection des constantes est importante car elle peut avoir une incidence directe sur l'identification des éléments mémorisants. En effet si le signal *C* n'est pas considéré comme un constante pendant la phase d'identification des registres, le processus *P2* sera implanté à tort sous forme d'un *latch* dont la condition de mémorisation est donnée par la valeur de *C*, c'est à dire toujours vraie.

```
ENTITY EXAMPLE IS
PORT ( A  : IN  BIT;
       B  : IN  BIT;
       Q  : OUT BIT );
END EXAMPLE;

ARCHITECTURE BEH OF EXAMPLE IS
  SIGNAL C : BIT;
BEGIN
  P1 : PROCESS
  BEGIN
    C <= '1';
    WAIT;
  END PROCESS P1;

  P2 : PROCESS
  BEGIN
    IF ( C = '1' ) THEN Q <= A OR B;
    END IF;
    WAIT ON A,B,C;
  END PROCESS P2;
END BEH;
```

Figure 4.32: Un signal à valeur constante.

Dans le cas la description de la figure 4.33 le problème est similaire. La variable *V* du processus *P3* est affectée à $'1'$ lors de l'initialisation et elle n'est jamais plus modifiée par la suite. La variable *V* doit être considérée comme une constante même si elle est affectée au moins une fois par le processus. Ainsi la phase d'identification des registres aura à sa disposition toutes les informations nécessaires pour conclure que *Q* n'est pas la sortie d'un *latch* mais simplement la sortie d'une porte *OU*.

```
ENTITY EXAMPLE IS
PORT ( A  : IN  BIT;
       B  : IN  BIT;
       Q  : OUT BIT );
END EXAMPLE;

ARCHITECTURE BEH OF EXAMPLE IS
BEGIN
  P3 : PROCESS
    VARIABLE V : BIT;
  BEGIN
    V := '1';
    LOOP
      IF ( V = '1' ) THEN Q <= A OR B;
      END IF;
      WAIT ON A, B;
    END LOOP;
  END PROCESS P3;
END BEH;
```

Figure 4.33: Une variable à valeur constante.

Les parties accessibles d'un processus

L'initialisation permet de déterminer pour chacun des processus, le premier point de suspension accessible. Cette information est indispensable pour la synthèse des processus à plusieurs points de suspension.

En effet le comportement d'un processus à plusieurs points de suspension dépend de son point de suspension initial. Par exemple, le processus *PMW* de la figure 4.34 possède trois points de suspension notés *W1*, *W2* et *W3*. Son comportement consiste à effectuer une addition de deux opérandes en trois cycles d'horloge *CK*. Durant le premier cycle il lit une donnée provenant de l'entrée A et la sauvegarde dans la variable interne *V1*. Durant le second cycle il lit une nouvelle donnée provenant de l'entrée A et la sauvegarde dans la variable interne *V2*. Dans le troisième cycle, il additionne le contenu des deux variables *V1* et *V2* puis écrit cette valeur sur la sortie Q. Pour obtenir un résultat de synthèse qui respecte à l'initialisation cet ordre d'exécution, il est nécessaire d'identifier le premier point de suspension *W1*.

Certains points de suspension peuvent être inaccessibles pour un point de suspension initial donné. La figure 4.35 présente une telle configuration. Le processus *P2* est constitué de deux boucles infinies *L1* et *L2* et de deux points de suspension *W1* et *W2*. Après la phase d'initialisation, ce processus est bloqué sur la première instruction *WAIT* accessible *W1* à l'intérieur de la boucle *L1*. Ce processus ne peut plus sortir de la boucle *L1* et son comportement permanent consiste à mémoriser la valeur de l'entrée $I0$ dans S sur front montant de l'horloge CK. Il peut donc être implanté en matériel à l'aide d'une bascule D. Ainsi puisque le second point de suspension *W2* de la boucle *L2* n'est pas accessible, il doit être ignoré lors de l'analyse de ce processus.

La connaissance du point de suspension initial est une information cruciale puisqu'elle permet de déterminer l'ensemble des points de suspension accessibles d'un processus. Après l'initialisation, la phase d'analyse peut donc exclure tous les points de suspension et toutes les instructions séquentielles qui n'interviennent pas dans le comportement permanent du processus.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY FSM IS
PORT ( CK : in  BIT;
       A  : in  SIGNED(3 DOWNTO 0);
       Q  : out SIGNED(3 DOWNTO 0) )
END FSM;

ARCHITECTURE MW OF FSM IS
BEGIN
  PMW : PROCESS
    VARIABLE V1, V2 : SIGNED(3 DOWNTO 0);
  BEGIN
    WAIT UNTIL CK='1'; -- W1
    V1 := A;
    WAIT UNTIL CK='1'; -- W2
    V2 := A;
    WAIT UNTIL CK='1'; -- W3
    Q <= V1 + V2;
  END PROCESS PMW;
END MW;
```

Figure 4.34: Le point de suspension initial d'un processus à plusieurs *WAIT* .

```
ENTITY EXAMPLE IS
PORT ( CK : IN  BIT;
       I0 : IN  BIT;
       Q  : OUT BIT );
END EXAMPLE;

ARCHITECTURE BEH_2 OF EXAMPLE IS
BEGIN
  P2 : PROCESS
  BEGIN
    L1 : LOOP
      WAIT UNTIL CK='1';   --- W1
      Q <= I0;
    END LOOP L1;

    L2 : LOOP
      WAIT UNTIL CK='0';   --- W2
      Q <= NOT(I0);
    END LOOP L2;
  END PROCESS P2;
END BEH_2;
```

Figure 4.35: Le premier point de suspension accessible.

4.4.2 Les mécanismes de l'initialisation de notre modèle

L'initialisation de notre modèle consiste à exécuter le graphe de contrôle de chacun des processus jusqu'à rencontrer leur premier point de suspension. Elle s'effectue en trois étapes : une étape d'initialisation des différents symboles (variables et signaux) de la description, une étape d'exécution des différents processus, et une étape de mise à jour de la valeur des signaux.

La valeur par défaut d'un symbole s est définie par s_{init}. Avant d'exécuter le graphe de contrôle des processus il est nécessaire de donner à chaque symbole x de l'ensemble $Data$ sa valeur par défaut x_{init} en procédant de la façon suivante :

- pour initialiser la valeur des signaux : $\forall s \in S, s_{eff} \leftarrow s_{init}$ et $s_{drv} \leftarrow s_{init}$;
- pour initialiser la valeur des variables : $\forall v \in V, v_{val} \leftarrow v_{init}$;

Les constantes c de l'ensemble C n'ont pas besoin d'être initialisées puisque leur valeur est directement donnée par c_{val}.

On procède ensuite à l'exécution séquentielle du corps de chacun des processus. Chaque processus possède un unique jeton, et l'exécution consiste à faire avancer ce jeton jusqu'à rencontrer une transition de type *RESUME*. On note p_i la place courante contenant cet unique jeton. L'exécution d'un processus est effectuée de la façon suivante :

- tant qu'il existe parmi les transitions successeurs de la place courante p_i, une transition t de type *EXECUTE* et telle que sa condition de franchissement $G(t)$ soit vraie alors :

  1. on exécute séquentiellement toutes les affectations de la liste $Trf(t)$.
  2. le jeton passe ensuite de p_i à p_{i+1}, où p_{i+1} est la place successeur de la transition t.

  Ce traitement est illustré sur la figure 4.37 où la transition t_1 est franchie et permet de faire passer le jeton de la place p_i à la place p_{i+1}.

- lorsque la place p_i a pour successeur une transition t de type *RESUME* l'exécution du processus s'arrête.

Après avoir exécuté un à un tous les processus jusqu'à rencontrer leur premier point de suspension, il faut appeler la fonction de résolution *resolved* définie par la norme IEEE-1164 [IEE93a] pour résoudre les éventuels conflits et ce pour tous les signaux qui possèdent plusieurs émetteurs et qui sont d'un type résolu. Il faut ensuite mettre à jour la valeur des signaux en fonction de la valeur de leurs pilotes : $\forall\ s \in S,\ s_{eff} \leftarrow s_{drv}$.

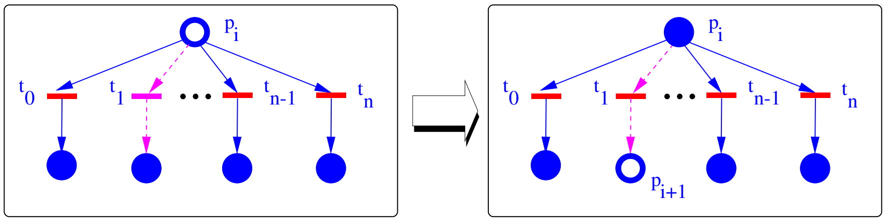

Figure 4.36: Une itération de l'algorithme d'initialisation de notre modèle.

Une fois l'exécution de cet algorithme terminée chaque place marquée d'un jeton précèdent une transition de type T_{res} représentant le premier point de suspension accessible d'un processus. De plus la valeur initiale de toutes les variables v et de tous les signaux s est donnée respectivement par v_{val} et s_{eff}.

Le résultat de l'initialisation de notre modèle est illustré sur la figure 4.37. Il s'agit de la description VHDL d'une bascule D qui mémorise sur front montant de l'horloge *CK* la valeur de l'entrée *A*.

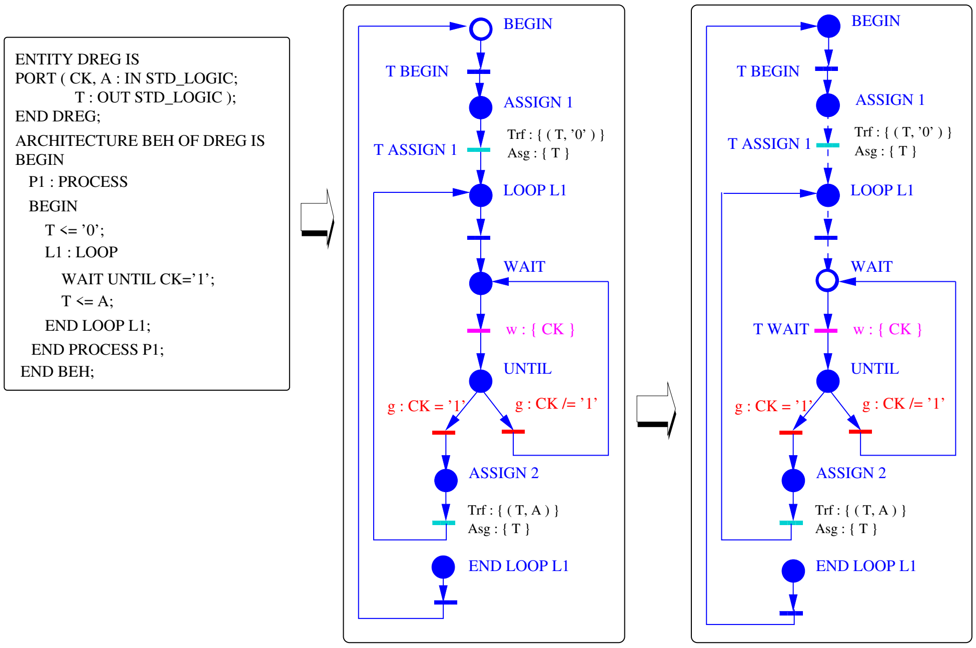

Figure 4.37: L'initialisation de la description VHDL d'une bascule D.

La phase d'initialisation permet au jeton de la place *BEGIN* d'atteindre la place *WAIT*. On peut remarquer qu'une fois la phase d'initialisation terminée, il n'existe plus de chemin permettant d'atteindre les places *BEGIN* et *ASSIGN 1*, ainsi que les transitions *T BEGIN* et *T ASSIGN 1*. La phase d'analyse peut donc enlever ces places et ces transitions du graphe de contrôle puisqu'elles n'entre pas dans le comportement permanent du processus.

La figure 4.38 donne les valeurs des symboles de la description avant et après l'initialisation. La synthèse connaît ainsi précisément la valeur initiale du signal *T*.

Symbole	Valeur par défaut	Valeur après initialisation
CK	'U'	'U'
A	'U'	'U'
T	'U'	'0'

Figure 4.38: Le calcul des valeurs initiales des symboles.

4.5 Conclusion

Au cours de ce chapitre nous avons tout d'abord détaillé les informations qui doivent absolument être préservées pendant la phase de compilation. Il s'agit en particulier :

- de préserver la sémantique de simulation du langage VHDL.
- de conserver l'ensemble des valeurs que peuvent prendre des types complexes comme les types énumérés ou les types entiers malgré leur conversion en vecteurs de bits.
- de pouvoir représenter les opérateurs standards, comme les opérateurs arithmétiques ou relationnels ;
- de garder les variables intermédiaires car elles peuvent servir pour un partage efficace des ressources matérielles ;

La spécification de cet ensemble de directives et de contraintes nous a permis de définir le modèle formel *XVPN* qui est une extension du modèle *VPN* adapté aux besoins de la synthèse. Les deux différences principales sont :

- la représentation des types complexes a été unifiée et ramenée au type "vecteur de bits" sans perte d'information ;
- les affectations attachées aux transitions de type *EXECUTE* ont une sémantique séquentielle.

L'importance de ces deux extensions apparaîtra plus clairement dans les chapitres 5 et 7.

Nous avons ensuite proposé une méthode de construction de ce modèle à partir d'une description VHDL.

Enfin nous avons montré que la phase d'initialisation de la simulation définie par la norme de VHDL devaient absolument être prise en compte lors de la synthèse puisqu'elle permet :

- de calculer les valeurs initiales des signaux, des variables et des constantes ;
- d'identifier pour chacun des processus les instructions qui interviennent dans leur comportement permanent.

Cette phase d'initialisation consiste à simuler la description sur un delta cycle et nous avons donc donné une méthode de simulation de notre modèle.

CHAPITRE 5

Une méthode de réduction du graphe de contrôle des processus possédant un seul point de suspension

Sommaire

Comme nous l'avons vu au chapitre 2, on peut distinguer du point de vue de la synthèse deux catégories de processus : les processus ne contenant qu'un seul point de suspension et ceux possédant plusieurs points de suspension. La méthode d'analyse est différente pour ces deux types de processus.

Dans ce chapitre nous étudions uniquement le cas des processus à un seul point de suspension qui sont les plus utilisés pour la synthèse. Une méthode permettant de traiter les processus à plusieurs points de suspension sera exposée dans le chapitre 7.

Nous présentons tout d'abord les deux classes de problèmes qui doivent être résolus lors de l'analyse d'un processus à un seul point de suspension. La résolution de ces deux classes de problèmes se traduit par deux étapes dans la méthode d'analyse : une étape de réduction du graphe de contrôle du processus et une étape d'analyse du graphe de contrôle réduit. Dans ce chapitre nous détaillons tous les aspects de cette première étape de réduction. La deuxième étape quant à elle fait l'objet du chapitre 6.

5.1 Présentation des différents problèmes à résoudre

Un processus à un seul point de suspension se réveille lorsqu'un évènement intervient sur un des signaux de sa liste de sensibilité. Il va ensuite emprunter un chemin d'exécution particulier suivant la valeur des signaux et de ses variables internes au moment de son réveil jusqu'à rencontrer de nouveau son unique point de suspension. Lors de cette exécution, le processus peut modifier la valeur de ses variables internes et les pilotes de certains signaux. Si un symbole n'est pas affecté lors de cette exécution il conserve sa valeur.

La difficulté principale de l'analyse réside dans l'identification des symboles qui correspondent à des registres et des barrières trois états. Comme nous l'avons évoqué au chapitre 2, un symbole qui n'est pas affecté sur tous les chemins d'exécution du graphe de contrôle ne doit pas nécessairement être considéré comme un registre. De même lorsqu'un symbole est affecté sur tous les chemins d'exécution l'analyse ne doit pas toujours le considérer comme la sortie d'une fonction combinatoire.

Notre méthode d'identification des registres et des barrières trois états s'effectue en deux étapes. Le but de la première étape est de rassembler pour un même symbole, toutes les affectations dont il fait l'objet sur les différents chemins d'exécution. Le but de la seconde étape est d'identifier formellement, pour tous les symboles affectés par le processus, le type d'élément matériel à utiliser pour obtenir un résultat de synthèse en conformité avec la sémantique de simulation : porte combinatoire, registre ou barrière trois états.

5.1.1 L'exploration du graphe de contrôle d'un processus

La première étape est effectuée en explorant les différents chemins d'exécution possibles sur le graphe de contrôle du processus. Il s'agit d'obtenir pour chaque symbole, l'ensemble des affectations qui peuvent lui être appliquées et les conditions associées à chacune de ces affectations.

Par exemple sur la figure 5.1, le processus *P1* se réveille dès qu'un évènement intervient sur les signaux *SEL*, *I0* ou *I1*. Une fois réveillé, il va pouvoir emprunter deux chemins différents suivant la valeur du signal *SEL*. Le premier chemin est pris lorsque le signal *SEL* vaut $'1'$ et le second lorsqu'il vaut $'0'$.

Sur cet exemple la première phase doit obtenir les informations suivantes :

- la variable *V* prend la valeur de *I0* quand le signal *SEL* vaut $'1'$ et sinon elle prend la valeur de *I1* ;
- la sortie *Q* prend toujours la valeur de la variable *V*.

```
ENTITY EXAMPLE IS
PORT ( SEL, I0, I1 : IN  BIT;
       Q           : OUT BIT );
END EXAMPLE;
ARCHITECTURE BEH OF EXAMPLE IS
BEGIN
  P1 : PROCESS
    VARIABLE V : BIT;
  BEGIN
    IF ( SEL = '1' )
     THEN V := I0;
     ELSE V := I1;
    END IF;
    Q <= V;
    WAIT ON SEL, I0, I1;
  END PROCESS P1;
END BEH;
```

Figure 5.1: Un exemple de multiplexeur .

Cette phase d'exploration du graphe de contrôle doit résoudre trois difficultés principales qui sont détaillées dans les paragraphes suivants.

Les affectations multiples des signaux

Une première difficulté est que certaines affectations de signaux effectuées sur les différents chemins d'exécution ne doivent pas être prises en compte. Sur l'exemple de la figure 5.2 la première affectation du signal *MUX* notée *ASG 1* doit être ignorée. En effet les deux affectations *ASG 2* et *ASG 3* écrasent la valeur du pilote du signal *MUX* qui a été précédemment positionnée par l'affectation *ASG 1*.

L'analyse du graphe de contrôle d'un processus ne doit pas considérer les modifications temporaires des pilotes des signaux, mais seulement celles qui sont visibles une fois l'exécution séquentielle du processus terminée. Aussi sur chaque chemin d'exécution l'analyse ne doit conserver que la dernière affectation pour chacun des signaux.

```
ENTITY EXAMPLE IS
PORT ( SEL, I0, I1 : IN  BIT;
       MUX         : OUT BIT );
END EXAMPLE;

ARCHITECTURE BEH OF EXAMPLE IS
BEGIN
  P2 : PROCESS( SEL, I0, I1 )
  BEGIN
    MUX <= '0';         --- ASG 1
    IF ( SEL = '1' )
      THEN MUX <= I0;   --- ASG 2
      ELSE MUX <= I1;   --- ASG 3
    END IF;
  END PROCESS P2;
END BEH;
```

Figure 5.2: Une affectation de signal inutile.

Les affectations multiples des variables

L'existence de plusieurs affectations pour une même variable a une signification très différente du cas des signaux et elle constitue une deuxième difficulté à résoudre. Le fait d'utiliser le même nom de variable à deux endroits différents du graphe de contrôle ne signifie pas qu'il s'agit du même objet matériel à synthétiser. Il peut s'agir de deux expressions combinatoires indépendantes qui doivent être synthétisées séparément, et l'utilisation du même nom de variable est une facilité d'écriture purement parasite du point de vue de la synthèse.

```
ENTITY EXAMPLE IS
PORT ( I0, I1 : IN  BIT;
       O0, O1 : OUT BIT );
END EXAMPLE;

ARCHITECTURE BEH OF EXAMPLE IS
BEGIN
  P1 : PROCESS( I0, I1 )
    VARIABLE V : BIT;
  BEGIN
    V := I0 OR I1;   --- ASG 1
    O0 <= V;         --- ASG 2
    V := I0 AND I1;  --- ASG 3
    O1 <= V;         --- ASG 4
  END PROCESS P1;
END BEH;
```

Figure 5.3: Deux affectations d'une même variable.

Par exemple sur la figure 5.3 la variable V représente successivement la sortie d'une porte *OU* et d'une porte *ET*.

Le nombre de chemins d'exécution à explorer

Une troisième difficulté est que la complexité de cette première étape d'analyse dépend du nombre de chemins d'exécution à explorer sur le graphe de contrôle du processus. Par exemple dans le cas où le graphe de contrôle est constitué d'une séquence de n instructions *IF*, le nombre de chemins est 2^n. Dans le cas où le graphe de contrôle est constitué d'un ensemble d'instructions *IF* imbriquées et de profondeur n, le nombre de chemins est aussi de 2^n. Les figures 5.4 et 5.5 présentent un exemple de ces deux types de graphe de contrôle avec n égal à 2. De telles configurations ne sont pas rares et la valeur de n peut être grande, entraînant une explosion du nombre de chemin.

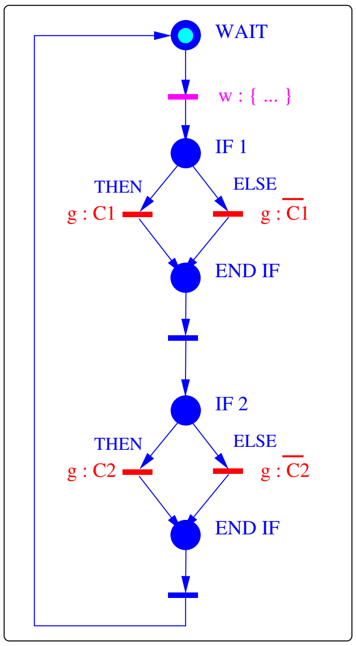

Figure 5.4: Une séquence de deux instructions *IF*.

Notre méthode cherche donc à réduire le nombre de chemins à explorer en appliquant un ensemble de techniques de réduction sur le graphe de contrôle. Les techniques de réduction présentées dans la section 5.2 ont pour objectif de transformer le graphe de contrôle d'un processus élaboré en un graphe de contrôle minimal, canonique, reproduisant le même comportement et constitué de deux places et de deux transitions de type T^{res} et T^{exe}. La figure 5.6 présente la structure d'un graphe de contrôle complètement réduit.

Le principe général de ces techniques de réduction consiste à intégrer dans chaque affectation la condition qui doit être respectée pour que l'affectation soit exécutée. Il s'agit également de rassembler, pour chaque symbole, toutes les affectations présentes sur les différents chemins d'exécution de façon à construire une affectation unique équivalente.

À l'issue de cette phase de réduction, l'ensemble des affectations séquentielles de l'unique transition de type T_{exe} du graphe réduit possède la propriété essentielle suivante : un symbole est affecté au plus une fois.

Les problèmes précédemment exposés concernant les affectations multiples d'un signal ou d'une variable sur un même chemin d'exécution sont donc complètement résolus par l'application de ces techniques de réduction.

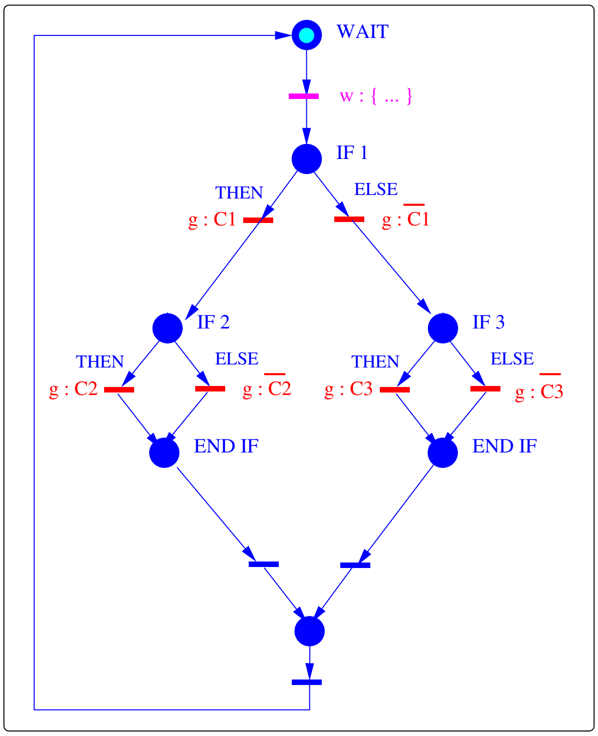

Figure 5.5: L'imbrication de plusieurs instructions *IF*.

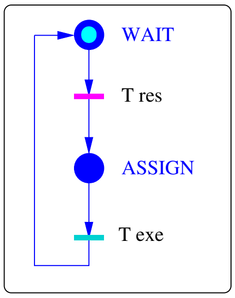

Figure 5.6: Le graphe de contrôle réduit d'un processus à un point de suspension.

5.1.2 L'identification du type d'élément matériel

La seconde étape consiste à analyser les affectations effectuées sur l'unique transition de type T_{exe} du graphe de contrôle réduit afin de déterminer pour chaque symbole le type de matériel nécessaire à son implantation. Il s'agit en particulier de classer chaque symbole suivant les trois types d'éléments matériels possibles que sont les registres, les barrières trois états et les portes combinatoires.

L'identification des registres

L'identification des registres est effectuée en recherchant les symboles s_i ayant la propriété suivante : l'affectation de s_i fait intervenir le symbole s_i lui même, directement ou indirectement. La figure 5.7 présente deux affectations de signaux. Dans le cas de l'affectation *ASG 1*, le signal *S* intervient directement dans l'expression qui lui est affectée. Dans le cas de l'affectation *ASG 2* le signal *T* intervient dans l'expression qui lui est affectée de façon indirecte, à cause de la variable intermédiaire *V*.

```
S <= (CK AND S) OR (NOT(CK) AND I0); -- ASG 1

V := T;
T <= (CK AND V) OR (NOT(CK) AND I1); -- ASG 2
```

Figure 5.7: Un exemple de dépendance cyclique .

Lorsqu'un symbole intervient directement ou indirectement dans l'expression qui lui est affectée, l'analyse doit permettre d'identifier, lorsqu'elle existe, la condition qui doit être vérifiée pour que le symbole conserve sa valeur. Une fois que cette condition est trouvée, il est possible qu'elle soit toujours fausse, et l'analyse peut conclure qu'il s'agit d'un symbole purement combinatoire. Dans le cas contraire, elle représente la condition de mémorisation du registre. L'analyse doit ensuite permettre d'identifier très précisément les caractéristiques de ce registres. Il s'agit par exemple de trouver le type de front d'horloge d'une bascule D, la condition de remise à zéro, ou encore la condition d'autorisation d'écriture. Une fois ces informations obtenues l'analyse doit vérifier leur cohérence avec la liste de sensibilité du processus.

Sur le processus de la figure 5.8 l'expression affectée au signal *S* dépend de *S*, et l'analyse doit obtenir les résultats suivants :

- l'affectation de *S* fait intervenir *S* lui même ;
- le signal *S* conserve sa valeur lorsque *CK* vaut $'0'$;
- la donnée à mémoriser est tout simplement l'entrée *I0* ;

La liste de sensibilité du processus n'est composée que du signal *CK*, et l'analyse peut donc conclure que le signal *S* représente bien la sortie d'une bascule D.

```
ENTITY EXAMPLE IS
PORT ( CK : IN  BIT;
       I0 : IN  BIT;
       Q  : OUT BIT );
END EXAMPLE;

ARCHITECTURE BEH OF EXAMPLE IS
   SIGNAL S : BIT;
BEGIN
  PROCESS( CK )
  BEGIN
    S <= (CK AND I0) OR (NOT(CK) AND S);
  END PROCESS;
  Q <= S;
END BEH;
```

Figure 5.8: Un exemple de bascule D .

L'identification des barrières trois états

L'identification des barrières trois états est également effectuée en analysant les expressions affectées à chaque symbole. Lorsque la constante $'Z'$ intervient directement ou indirectement dans l'expression qui lui est affectée, l'analyse peut supposer qu'il existe une condition pour laquelle le symbole est en haute impédance. Une fois cette condition obtenue l'analyse doit déterminer la valeur de la donnée à placer en entrée de la barrière trois états et vérifier la cohérence de ces informations avec la liste de sensibilité du processus.

Les symboles qui ne sont ni des registres, ni des barrières trois états doivent être implantés à l'aide de portes combinatoires. Les expressions affectées à ces symboles doivent cependant être en accord avec la liste de sensibilité du processus comme nous l'avons exposé au chapitre 2.

5.2 La réduction du graphe de contrôle

La phase de réduction du graphe de contrôle d'un processus se déroule en trois étapes. Tout d'abord les affectations de signaux et de variables présentes sur le graphe de contrôle sont modifiées de façon à faciliter la réduction. Le graphe de contrôle est ensuite simplifié en appliquant deux techniques de réduction locale exploitant la structure des instructions VHDL. La dernière étape consiste à appliquer si nécessaire une technique de réduction globale permettant d'obtenir un graphe de contrôle minimal.

5.2.1 Le traitement des signaux

La première étape de la réduction vise à remplacer les signaux affectés sur le graphe de contrôle du processus par des variables tout en respectant la sémantique de VHDL, pour obtenir un modèle homogène. Il s'agit donc des signaux présents dans le membre de gauche d'une affectation.

Comme nous l'avons vu dans la section précédente, un signal peut être affecté plusieurs fois sur un même chemin d'exécution et la phase d'analyse ne doit cependant considérer que la dernière valeur donnée au pilote d'un signal.

Afin de rendre explicite ce mécanisme d'affectation tardive des signaux, on introduit pour chaque signal s affecté par le processus une variable v_{Sdrv} représentant à tout moment la valeur de son pilote. On introduit ensuite deux nouvelles affectations avant et après l'unique point de suspension du processus. Si n est le nombre de signaux affectés par le processus, la phase initiale de la réduction crée n variables intermédiaires v_{Sdrv}. On note V^{drv} l'ensemble de toutes les variables intermédiaires v_{Sdrv} ainsi créées.

Une première affectation $s \Leftarrow v_{Sdrv}$ placée avant le point de suspension permet de donner à s la dernière valeur de son pilote. Une seconde affectation $v_{Sdrv} := s$ placée après le point de suspension permet d'initialiser correctement la variable représentant le pilote de s. Toutes les affectations du signal s présentes sur les différents chemins d'exécution du processus sont ensuite transformées en des affectations de la variable v_{Sdrv}. Cette variable v_{Sdrv} possède la propriété d'être toujours affectée avant d'être lue.

Une fois cette transformation effectuée le graphe de contrôle du processus ne comporte plus que des affectations de variables : tous les signaux sont affectés une seule fois avant le point de suspension.

La figure 5.9 présente le résultat de cette étape préliminaire de la phase de réduction.

```
ENTITY EXAMPLE IS
PORT ( A  : IN  BIT;
       B  : IN  BIT;
       Q  : OUT BIT );
END EXAMPLE;

ARCHITECTURE BEH_1 OF EXAMPLE IS
  SIGNAL S : BIT;
BEGIN
  P1 : PROCESS
  BEGIN
    S <= A OR B;  -- ASG 1
    S <= A AND B; -- ASG 2
    WAIT ON A,B;
  END PROCESS P1;
  Q <= S;
END BEH_1;
```

$\Rightarrow$

```
ENTITY EXAMPLE IS
PORT ( A  : IN  BIT;
       B  : IN  BIT;
       Q  : OUT BIT );
END EXAMPLE;

ARCHITECTURE BEH_2 OF EXAMPLE IS
  SIGNAL S : BIT;
BEGIN
  P2 : PROCESS
    VARIABLE V_Sdrv : BIT;
  BEGIN
    V_Sdrv := A OR B;   -- ASG 3
    V_Sdrv := A AND B;  -- ASG 4
    S      <= V_Sdrv;   -- ASG 5
    WAIT ON A,B;
    V_Sdrv := S;        -- ASG 6
  END PROCESS P2;
  Q <= S;
END BEH_2;
```

Figure 5.9: L'introduction d'une variable représentant le pilote d'un signal.

5.2.2 L'introduction de nouvelles variables

Il s'agit maintenant de traiter toutes les affectations de variables, c'est à dire aussi bien les affectations de variables initialement présentes dans la description que les affectations de variables introduites lors de la transformation des affectations de signaux. Une variable peut être affectée plusieurs fois sur un même chemin d'exécution.

On cherche donc à rendre explicite la "durée de vie" des variables du processus analysé, de façon à identifier toutes les variables "indépendantes". Pour cela, on remplace toutes les affectations de variables par une paire d'affectations :

$$v := expression; \quad \Longrightarrow \quad \begin{array}{l} red_i := expression; \\ v := red_i; \end{array}$$

Si il y a m affectations de variables dans le processus, on introduit alors m nouvelles variables $red_i, (1 \leq i \leq m)$. La réduction du graphe de contrôle commence donc par une phase d'expansion du nombre de variables.

On note V^{red} l'ensemble de toutes les variables intermédiaires red_i ainsi créées et V^{ini} l'ensemble de toutes les autres variables. On peut remarquer que les variables créées lors de la transformation des affectations de signaux ne font pas partie de l'ensemble V^{red}.

Cette transformation préserve le comportement du processus, et chaque variable intermédiaire red_i est toujours affectée avant d'être lue. De plus chaque variable red_i est affectée une seule fois et sur une seule transition du graphe de contrôle. Après cette transformation, la partie droite d'une affectation de variable $v \in V^{ini}$ est toujours une variable red_i. Cette propriété facilitera en outre le traitement des affectations partielles de vecteurs de bits lors de la réduction, comme nous le verrons dans les sections 5.2.3 et 5.2.4.

Les étapes suivantes de la réduction devront remplacer dans toutes les affectations, la lecture d'une variable v par la lecture d'une ou plusieurs variables intermédiaires red_i. L'objectif majeur est de supprimer les variables de l'ensemble V^{ini} de tous les membres de droite des affectations et de ne conserver que les variables de l'ensemble V^{red}.

```
ENTITY EXAMPLE IS
PORT ( A,B  : IN  BIT;
       S,T  : OUT BIT );
END EXAMPLE;
ARCHITECTURE BEH_1 OF EXAMPLE IS
BEGIN
  P1 : PROCESS( A, B )
    VARIABLE V : BIT;
  BEGIN
    V := A OR B;  -- ASG 1
    ...
    V := A AND B; -- ASG 2
    ...
  END PROCESS P1;
END BEH_1;
```

$\Longrightarrow$

```
ENTITY EXAMPLE IS
PORT ( A,B  : IN  BIT;
       S,T  : OUT BIT );
END EXAMPLE;
ARCHITECTURE BEH_2 OF EXAMPLE IS
BEGIN
  P2 : PROCESS( A, B )
    VARIABLE V : BIT;
    VARIABLE red_1, red_2 : BIT;
  BEGIN
    red_1 := A OR B;  -- ASG 3
    V     := red_1;   -- ASG 4
    ...
    red_2 := A AND B; -- ASG 5
    V     := red_2;   -- ASG 6
    ...
  END PROCESS P2;
END BEH_2;
```

Figure 5.10: L'introduction de variables intermédiaires.

La figure 5.10 présente le résultat de cette étape de transformation. Sur cet exemple les phases ultérieures de la réduction devront remplacer dans les membres de droite des affectations situées entre *ASG 4* et *ASG 5*, toutes les occurences de la variable v par

red_1, et entre *ASG 6* et la fin du processus elles devront remplacer les occurences de v par red_2.

5.2.3 La réduction locale latérale

La réduction locale latérale minimise le nombre de chemins d'exécution du graphe de contrôle d'un processus en exploitant la structure des différentes instructions conditionnelles. Elle préserve le comportement permanent du processus et respecte la sémantique de simulation de VHDL.

Les conditions d'application de la réduction latérale

La réduction locale latérale cherche à transformer en une unique transition un ensemble $T_{lat} = \{t_1, \cdots, t_n\}$ de transitions de type T^{exe} représentant chacune une alternative d'exécution.

Elle peut être appliquée sur chaque sous graphe associé à une instruction conditionnelle comme les constructions de type *CASE* ou de type *IF* pour peu que toutes les transitions t_i vérifient les conditions suivantes :

- elles appartiennent toutes à l'ensemble T_{exe} ;
- elles possèdent une unique et même place p_{begin} en entrée et cette place n'a pas d'autres successeurs que les transitions de T_{lat} ;
- elles possèdent une unique et même place p_{end} en sortie et cette place n'a pas d'autres prédécesseurs que les transitions de T_{lat}.
- toutes les transitions t_i possèdent une condition de franchissement g_{t_i}. Les conditions booléennes g_{t_i} sont mutuellement exclusives et leur somme vaut 1.

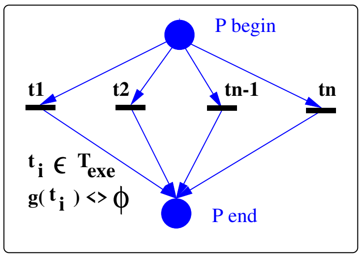

Figure 5.11: Les conditions d'application de la réduction latérale.

Les conditions que doivent vérifier les transitions de l'ensemble T_{lat} sont illustrées sur la figure 5.11.

La modification de la structure du sous graphe de contrôle

La réduction latérale de l'ensemble T_{lat} remplace toutes les transitions t_i par une unique transition t_{lat} équivalente. Cette transition t_{lat} est construite ainsi :

1. une nouvelle transition t_{lat} est ajoutée à l'ensemble T des transitions du réseau de Pétri, puis deux arcs la reliant aux deux places p_{begin} et p_{end} sont créées ;
2. la condition booléenne de franchissement associée à t_{lat} est toujours vraie.
3. l'ensemble $Asg(t_{lat})$ des symboles modifiés par l'exécution de la transition t_{lat} est l'ensemble de tous les symboles modifiés lors du franchissement des différentes transitions t_i, on a donc : $Asg(t_{lat}) \;= \bigcup_{i=1}^{n} Asg(t_i)$;
4. il s'agit ensuite de construire la liste $\mathit{Trf}(t_{lat})$ des affectations séquentielles à exécuter lors du franchissement de t_{lat}. La définition de l'ordre des affectations est le problème principal à résoudre et l'algorithme est détaillé dans les paragraphes suivants ;
5. enfin toutes les transitions de l'ensemble T_{lat} sont enlevées du graphe de contrôle ainsi que tous les arcs associés.

Le résultat des modifications structurelles apportées au sous graphe de contrôle lors de la réduction locale latérale est illustré sur la figure 5.12.

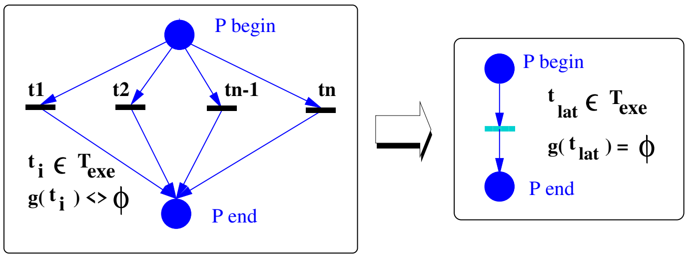

Figure 5.12: La structure du sous graphe résultant de la réduction latérale.

La construction des affectations du sous graphe de contrôle réduit par réduction latérale

La création de la liste des affectations $\mathit{Trf}(t_{lat})$ doit :

- conserver l'ordre partiel d'exécution de chacune des listes d'affectations $\mathit{Trf}(t_i)$ présentes sur les différentes transitions t_i pour préserver le comportement du processus ;
- garantir qui'il existe au plus une affectation à un même symbole.

La construction de la liste d'affectations $Trf(t_{lat})$ est effectuée en deux étapes qui sont décrites dans les paragraphes suivants.

Les deux opérateurs IFT et $REDUCE$: La construction de la liste d'affectations nécessite l'introduction de deux opérateurs IFT et $REDUCE$ dont nous donnons une définition formelle en VHDL sur la figure 5.13 pour le type *STD_LOGIC*.

```
FUNCTION IFT(COND: BOOLEAN; VAL: STD_LOGIC) RETURN STD_LOGIC IS
  VARIABLE RESULT: STD_LOGIC;
BEGIN
  IF ( COND ) THEN RESULT := VAL;
            ELSE RESULT := '0';
  END IF;
  RETURN RESULT;
END IFT;

FUNCTION REDUCE(VAL1, VAL2: STD_LOGIC) RETURN STD_LOGIC IS
  VARIABLE RESULT: STD_LOGIC;
BEGIN
  IF (VAL1 /= '0') THEN RESULT := VAL1;
                   ELSE RESULT := VAL2;
  END IF;
  RETURN RESULT;
END REDUCE;
```

Figure 5.13: Les deux fonctions utilisées lors de la réduction latérale.

La fonction IFT possède deux arguments *COND* et *VAL* représentant respectivement une condition booléenne et une valeur sur n bits dans $(\mathcal{S})^n$. Lorsque la condition *COND* est vérifiée, la fonction IFT retourne la valeur de l'argument *VAL*, et lorsqu'elle n'est pas vérifiée elle renvoie un vecteur de taille n constitué uniquement de $'0'$.

La fonction $REDUCE$ permet de regrouper toutes les affectations d'un même symbole qui sont effectuées sur les différentes transitions t_i. Elle possède deux arguments $VAL1$ et $VAL2$ représentant deux valeurs possibles pour le symbole affecté. Il y a au plus une de ces deux valeurs différente de $'0'$, puisqu'un seul de tous les appels à la fonction IFT renvoie la valeur à affecter. La fonction $REDUCE$ analyse donc la valeur de *VAL1* et de *VAL2*. Il existe trois possibilités :

- *VAL1* est différent de $'0'$, *VAL2* est donc forcément égal à $'0'$, et par conséquent la fonction $REDUCE$ doit renvoyer la valeur de *VAL1* ;
- *VAL2* est différent de $'0'$, *VAL1* est donc obligatoirement égal à $'0'$, et la fonction $REDUCE$ doit renvoyer la valeur de *VAL2* ;
- *VAL1* et *VAL2* sont tous deux égaux à $'0'$, et la fonction $REDUCE$ doit renvoyer $'0'$.

La première étape : Elle traite les affectations de variables red_j de l'ensemble V^{red}. Ces variables ont la propriété d'être affectées une seule fois à un unique endroit du graphe de contrôle du processus. De plus elles sont toujours affectées avant d'être lues.

Aussi, les éventuelles affectations de ces variables red_j présentes sur les différentes transitions t_i de T_{lat} peuvent être directement recopiées sur la transition t_{lat}. Il faut cependant respecter l'ordre partiel des affectations de chaque transition t_i et donc les recopier dans l'ordre défini par les différentes listes d'affectations $Trf(t_i)$.

La seconde étape : Elle traite les affectations des autres variables v présentes sur les différentes transitions t_i de l'ensemble T_{lat}. Contrairement aux variables red_j, les variables v peuvent être affectées sur plusieurs transitions t_i.

La réduction locale latérale doit regrouper toutes les affectations des variables v présentes sur les différentes transitions t_i. Ce regroupement est effectué à l'aide des deux fonctions IFT et $REDUCE$ précédemment définies.

On modifie tout d'abord toutes les affectations des différentes variables v sur toutes les transitions t_i de la façon suivante :

- si v est affectée sur la transition t_i alors $v\ :=\ f^{t_i}$ devient $v\ :=\ IFT(g_{t_i}, f^{t_i})$;
- lorsqu'une variable v n'est pas affectée sur toutes les transitions il est nécessaire de créer une nouvelle affectation sur chacune des transitions où elle n'est pas affectée. Cette nouvelle affectation doit représenter le fait que lorsque la condition de franchissement de la transition est vraie, alors la variable n'est pas modifiée et elle conserve sa valeur. La nouvelle affectation à ajouter à la fin de la liste $Trf(t_i)$ est donc $v := IFT(g_{t_i}, v)$.

On regroupe ensuite pour chaque variable v, toutes les expressions $IFT(g_{t_i}, f^{t_i})$ ou $IFT(g_{t_i}, v_j)$ en procédant de la façon suivante :

$$red_j := REDUCE(f^{t_1}, REDUCE(f^{t_2}, \ldots REDUCE(f^{t_{n-1}}, f^{t_n}) \ldots));$$
$$v := red_j;$$

Pour une variable v, l'ordre dans lequel on effectue la réduction est sans importance puisqu'une seule condition de garde est vraie. Il est nécessaire d'introduire la variable intermédiaire red_j pour respecter la règle selon laquelle toute affectation d'une variable $v \in\ V^{ini}$ contient une variable de type V^{red} dans son membre de droite.

L'ordre dans lequel sont traitées les variables v est également sans importance puisqu'il ne peut y avoir de dépendance entre deux variables v_x et v_y grâce à l'introduction des variables $red_j \in\ V^{red}$.

D'un coté la réduction locale latérale propage les conditions de franchissement g_{t_i} de chacune des transitions t_i de T_{lat} sur chacune des listes d'affectation $Trf(t_i)$. Et d'un autre coté elle rassemble sur la transition t_{lat} toutes les affectations qui sont effectuées, et ce pour chacune des variables modifiées lors du franchissement d'au moins une des transitions t_i.

Chaque variable est affectée une seule fois et toutes les dépendances séquentielles sont respectées. L'application de la réduction locale latérale ne modifie donc en rien le

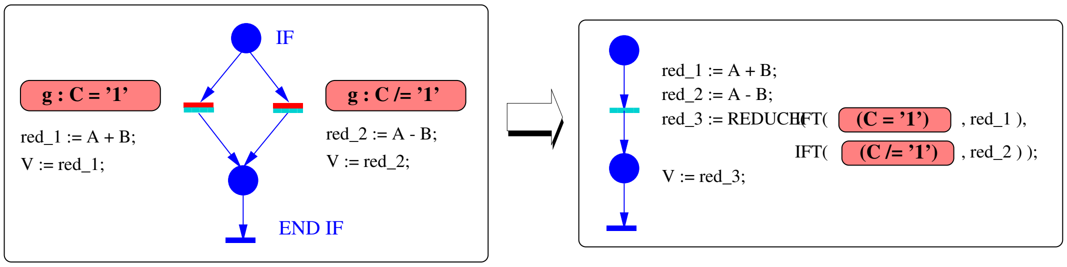

Figure 5.14: La réduction locale latérale.

comportement du processus. Le résultat de cette technique de réduction est illustré sur la figure 5.14.

Les variables de type vecteur de bits : Lorsque des variables v de type vecteur de bits sont affectées sur les différentes transitions t_i, il est parfois nécessaire de modifier les affectations de ces variables avant d'effectuer la première étape de construction de $Trf(t_{lat})$.

En effet si v est une variable de type vecteur de bits, on rencontre le problème des affectations partielles, puisqu'en VHDL l'affectation d'un vecteur peut être effectuée sur un intervalle réduit de ses composantes.

On note $\mathcal{I}_{(v,t_i)} = \{\mathcal{I}^1_{(v,t_i)}, \dots, \mathcal{I}^m_{(v,t_i)}\}$ l'ensemble des m intervalles d'affectation de la variable v sur la transition t_i.

Par exemple, sur la figure 5.15, les deux ensembles d'intervalles des affectations de la variable v sont : $\mathcal{I}_{(v,t_1)} = \{[0;4],[5]\}$ et $\mathcal{I}_{(v,t_2)} = \{[1;3],[6;7]\}$.

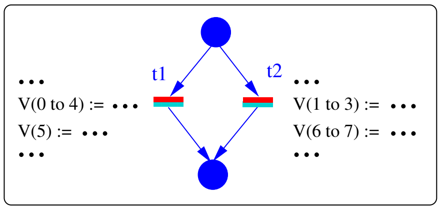

Figure 5.15: Les intervalles d'affectations d'une variable de type vecteur.

Les intervalles d'affectation d'une variable v peuvent être différents suivant les transitions t_i, comme c'est le cas sur l'exemple de la figure 5.15. Pour rendre homogène les intervalles d'affectation d'une variable v de type vecteur sur les différentes transitions t_i, il est nécessaire de trouver pour chacune de ces transitions un nouvel ensemble d'intervalles. Ce nouvel ensemble $\mathcal{I}'_{(v,t)}$ est construit de tel façon que $\forall k, p, q$ on ait $\mathcal{I}'^p_{(v,t)} \bigcap \mathcal{I}'^q_{(v,t_k)} = \varnothing$ ou $\mathcal{I}'^p_{(v,t)} = \mathcal{I}'^q_{(v,t_k)}$.

Ainsi sur l'exemple de la figure 5.15 on obtient les deux ensembles d'intervalles suivants : $\mathcal{I}'_{(v,t_1)} = \{[0],[1;3],[4],[5]\}$ et $\mathcal{I}'_{(v,t_2)} = \{[1;3],[6;7]\}$.

Il s'agit ensuite de découper les affectations de la variable v en accord avec ces nouveaux ensembles d'intervalles sur toutes les transitions t_i. La partie droite de l'affectation d'une variable v étant toujours une variable de l'ensemble V^{red} le découpage des affectations ne présente aucune difficulté. L'exemple d'un tel découpage est donné sur la figure 5.16.

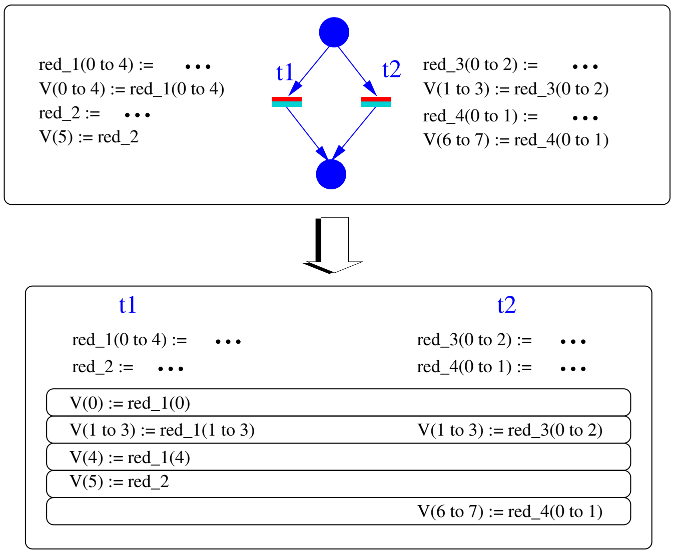

Figure 5.16: Le découpage des affectations de variable de type vecteur.

5.2.4 La réduction locale séquentielle

La réduction locale séquentielle cherche à transformer en une unique transition deux transitions t_1 et t_2 consécutives de type T^{exe} et représentant deux séquences d'affectations.

Les conditions d'application de la réduction séquentielle

On note p_1 la place située en entrée de t_1, p_2 la place située en sortie de t_1 et p_3 la place située en sortie de t_2. Cette réduction peut être appliquée sur chaque sous graphe associé à plusieurs instructions d'affectation consécutives, pour peu que les deux transitions t_1 et t_2 vérifient les conditions suivantes :

- t_1 et t_2 appartiennent toutes les deux à l'ensemble T_{exe} ;
- la transition t_2 est franchissable de façon inconditionnelle ;
- la place p_2 n'a pas d'autre prédécesseur que la transition t_1 ;

- la place p_2 n'a pas d'autre successeur que la transition t_2.

Les conditions que doivent vérifier les deux transitions t_1 et t_2, ainsi que les trois places p_1, p_2 et p_3 sont illustrées sur la figure 5.17.

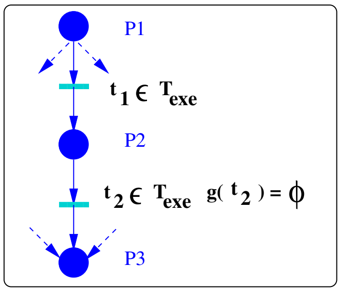

Figure 5.17: Les conditions d'application de la réduction séquentielle.

La modification de la structure du sous graphe de contrôle

La réduction séquentielle remplace les deux transitions t_1 et t_2 par une unique transition t_{seq} équivalente :

1. une nouvelle transition t_{seq} est ajoutée à l'ensemble T des transitions du réseau de Pétri, puis deux arcs la reliant aux deux places p_1 et p_3 sont créés ;

2. la condition booléenne de franchissement associée à t_{seq} est égale à la condition de franchissement g_{t_1}.

3. l'ensemble $Asg(t_{seq})$ des symboles modifiés par l'exécution de la transition t_{seq} est l'ensemble de tous les symboles modifiés lors du franchissement des deux transitions t_1 et t_2, on a donc : $Asg(t_{seq}) = Asg(t_1) \bigcup Asg(t_2)$;

4. il s'agit ensuite de construire la liste $Trf(t_{seq})$ des affectations séquentielles à exécuter lors du franchissement de t_{seq}. La difficulté est ici de garantir que chaque variable n'est affectée qu'une fois. L'algorithme de création de cette liste est détaillé dans la section suivante ;

5. enfin les deux transitions t_1 et t_2 sont enlevées du graphe de contrôle ainsi que tous les arcs associés.

Le résultat des modifications structurelles apportées au sous graphe de contrôle lors de la réduction locale séquentielle, est illustré sur la figure 5.18.

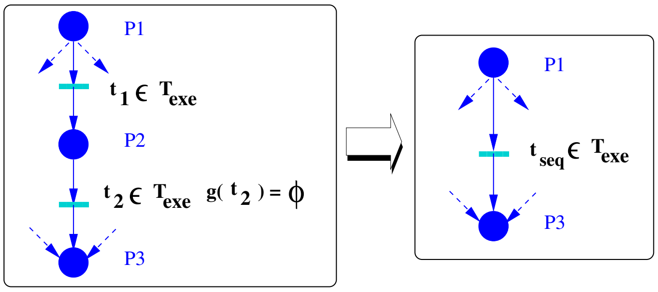

Figure 5.18: La structure du sous graphe résultant de la réduction séquentielle.

La construction des affectations du sous graphe de contrôle réduit par réduction séquentielle

La création de la liste des affectations $Trf(t_{seq})$ doit :

- conserver l'ordre d'exécution des listes d'affectations $Trf(t_1)$ et $Trf(t_2)$;
- garantir qu'il existe au plus une affectation à un même symbole. Rappelons que dans les deux listes $Trf(t_1)$ et $Trf(t_2)$ chaque variable n'est affectée qu'une fois. Par contre des variables $v \in V^{ini}$ peuvent être affectées à la fois sur t_1 et t_2.

La construction de la liste d'affectations $Trf(t_{seq})$ est effectuée en quatre étapes.

La première étape : Elle a pour objectif de propager la valeur de toutes les variables v de l'ensemble V^{ini} qui sont affectées sur la transition t_1 et qui sont lues par les affectations de la transition t_2.

Après la modification initiale des affectations que nous avons présenté à la section 5.2.1, la partie droite d'une affectation de variable $v \in V^{ini}$ est toujours une variable intermédiaire $red_i \in V^{red}$. La substitution d'une variable v par sa valeur est donc effectuée simplement en substituant la variable red_i à la variable v.

On note $Supp(f)$ le support d'une expression f, c'est à dire l'ensemble des variables et des signaux qui apparaissent dans f. On parcourt séquentiellement toutes les affectations de variables (v, f^{t_2}) de la liste $Trf(t_2)$ et pour chaque variable $v_j \in V^{ini}$ qui apparaît dans $Supp(f^{t_2})$ on substitue dans f^{t_2} toutes les occurences de v_j par red_i s'il existe une affectation (v_j, red_i) dans la liste des affectations de la transition t_1.

Un exemple de propagation de la valeur des variables affectées par t_1 et lues par t_2 est présenté sur la figure 5.19.

La seconde étape : Elle traite les affectations de variables red_i de l'ensemble V^{red} effectuées sur l'une des deux transition t_1 et t_2. Ces variables étant affectées une seule fois à un unique endroit du graphe de contrôle du processus, les affectations de ces

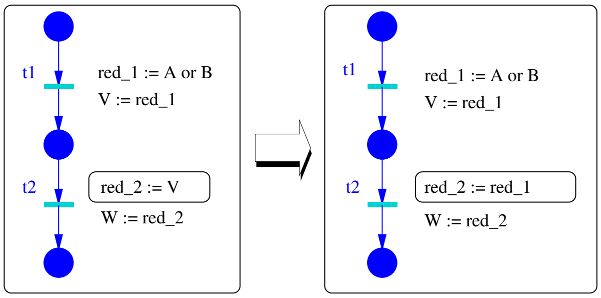

Figure 5.19: La substitution des variables.

variables peuvent donc être directement recopiées sur la transition t_{seq} dans l'ordre où elles apparaissent.

La troisième étape : Elle traite les variables $v \in V^{ini}$ affectées sur la transition t_1 ou t_2. On recopie séquentiellement toutes les affectations de variables $v \in V^{ini}$ et si v est affectée à la fois sur t_1 et t_2 alors on ne copie que l'affectation présente sur t_2.

La dernière étape : Elle traite les affectations de signaux. Tous les signaux sont affectés une seule fois et à un unique endroit dans tous le graphe de contrôle du processus. On recopie donc simplement toutes les affectations de signaux s_j de $Trf(t_1)$ et de $Trf(t_2)$ à la fin de la liste $Trf(t_{seq})$.

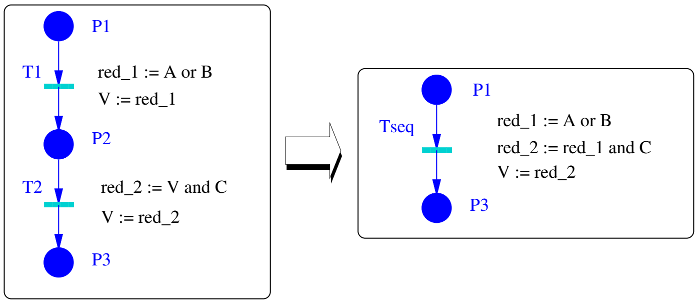

Figure 5.20: La réduction locale séquentielle.

Le résultat de cette technique de réduction est illustré sur la figure 5.20. L'exécution de la transition t_{seq} produit exactement le même comportement qu'une exécution séquentielle des deux transition t_1 et t_2.

La réduction séquentielle nous a permis tout d'abord d'enlever localement les affectations multiples d'une même variable de V^{ini}. Sur la figure 5.20 par exemple, on n'a conservé que la dernière affectation de la variable v. La réduction séquentielle nous a permis également de remplacer dans les membres de droite des affectations de la transition t_2, toutes les occurences de variables $v \in V^{ini}$ affectées sur t_1 par une variable de V^{red}. Par conséquent lors de la réduction du graphe de contrôle toutes les variables

v_k de V^{ini} qui sont affectées avant d'être lues sont remplacées par une variable red_j de V^{red}. Ainsi lorsque le graphe de contrôle sera complètement réduit, ces variables v_k n'interviendront plus dans aucun membre de droite d'une affectation et l'analyse pourra donc les éliminer du graphe de contrôle.

Les variables de type vecteur de bits : Dans le cas des affectations de variables de type vecteur de bits, la propagation de la valeur des variables peut être effectuée de façon partielle lors de la première étape. Un exemple d'une telle configuration est donné sur la figure 5.21. La variable V est affectée partiellement sur la transition t_1, et elle est lue dans sa totalité lors de l'affectation de la variable *Wred_1*. On utilise donc l'opérateur de concaténation pour propager partiellement la valeur de la variable V.

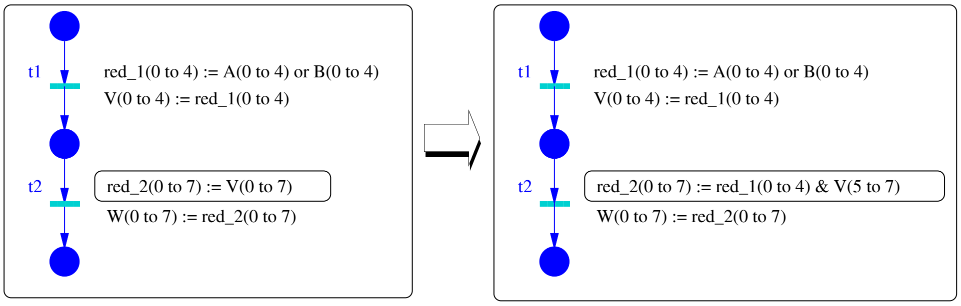

Figure 5.21: Le renommage partiel des variables de type vecteur.

Lors de la troisième étape, si une variable v est de type vecteur de bits, alors seules certaines de ses composantes peuvent être affectées, et ce différemment sur les deux transitions t_1 et t_2. Il convient donc de rendre homogène les intervalles d'affectations d'une même variable v sur les deux transitions t_1 et t_2 et on utilise pour cela la même technique que celle employée lors de la réduction locale latérale. On construit pour chaque variable v affectée à la fois sur t_1 et sur t_2 un nouvel ensemble d'intervalles d'affectation homogènes et on découpe ensuite les affectations de $Trf(t_1)$ et $Trf(t_2)$ en conséquence. L'exemple d'un tel découpage est donné sur la figure 5.22.

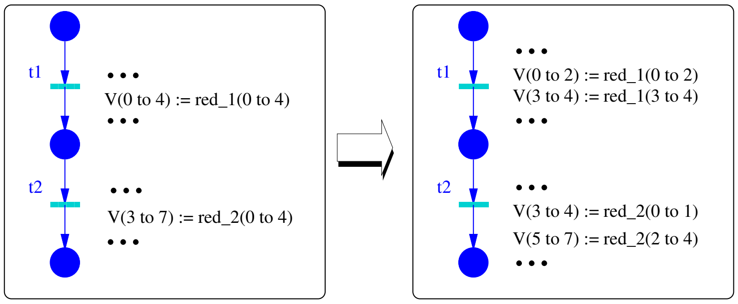

Figure 5.22: Le découpage des affectations de variable de type vecteur.

5.2.5 La stratégie de réduction du graphe de contrôle

Comme nous l'avons vu au chapitre 4, lors de la compilation d'une description VHDL, chaque instruction séquentielle d'un processus est traduite sur notre modèle sous la forme d'un sous réseau de Pétri. Le sous réseau de Pétri correspondant à une instruction VHDL possède deux propriétés :

- il est toujours constitué d'une première place $p^1_{Inst_i}$ qui représente le début de l'instruction $Inst_i$;
- il est toujours terminé par une place $p^n_{Inst_i}$ et une transition $t^n_{Inst_i}$ qui modélisent la fin de l'instruction $Inst_i$.

On peut donc utiliser efficacement les algorithmes de réduction locale décrits aux paragraphes 5.2.3 et 5.2.4.

Cette réduction locale exploite directement la structure de l'instruction et il existe pour chaque type d'instruction une technique particulière de réduction locale. Ces techniques sont décrites dans l'annexe C et leur application est illustrée sur la figure 5.23.

La figure présente la description d'un processus décrivant le comportement d'une bascule s qui mémorise sur front montant de l'horloge ck la donnée d. Ce processus est traité de la façon suivante :

1. on construit le modèle *XVPN* du processus ;
2. l'étape initiale de la réduction introduit ensuite des variables intermédiaires dans les affectations de signaux et de variables. Une variable v_{Sdrv} a été ajoutée pour représenter le pilote du signal s et deux variables intermédiaires red_0 et red_1 ont été introduites pour représenter les deux valeurs que peut prendre v_{Sdrv}.
3. les instructions du corps du processus sont localement réduites, en commençant par l'instruction *IF* située juste après la place *BEGIN*. L'instruction *IF* est réduite en une affectation équivalente *ASSIGN red* ;
4. la nouvelle instruction d'affectation est ensuite localement réduite ;
5. l'instruction d'affectation du signal s est elle aussi localement réduite ;
6. l'instruction *WAIT* et l'instruction d'affectation située juste après ne peuvent être réduites localement. On procède donc à la réduction de la place *BEGIN* en la considérant comme une instruction d'affectation, et on obtient ainsi une unique transition de type T^{exe} équivalente ;

L'application de la réduction locale sur cet exemple nous a donc permis d'obtenir un graphe de contrôle minimal et équivalent, composé d'une transition t_1 de type T^{exe} et d'une transition t_2 de type T^{res}.

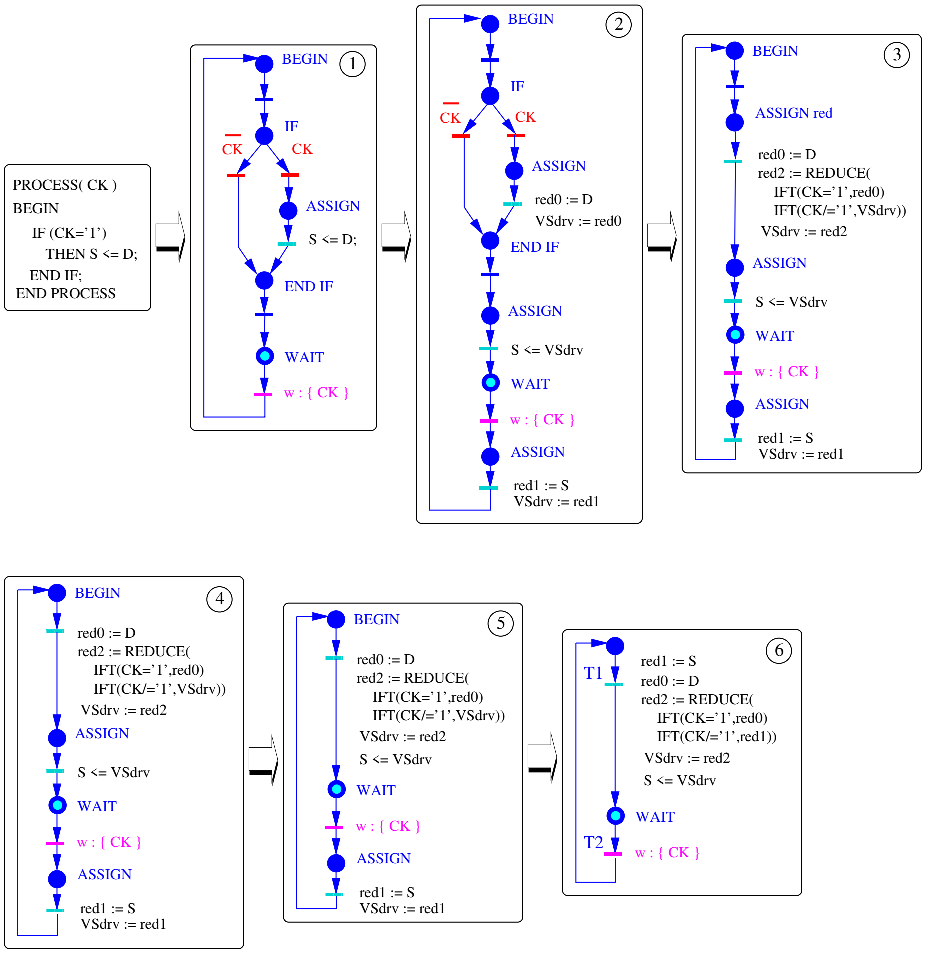

Figure 5.23: La réduction locale d'un processus.

En pratique, la plupart des processus à un seul point de suspension sont complètement réduits par l'application des deux techniques de réduction locale. Cependant il existe des graphes de contrôle comme celui de la figure 5.24 pour lesquels cette réduction locale échoue, et il est donc nécessaire d'élaborer une autre technique de réduction, permettant d'obtenir une forme minimale quelque soit la structure initiale du graphe de contrôle.

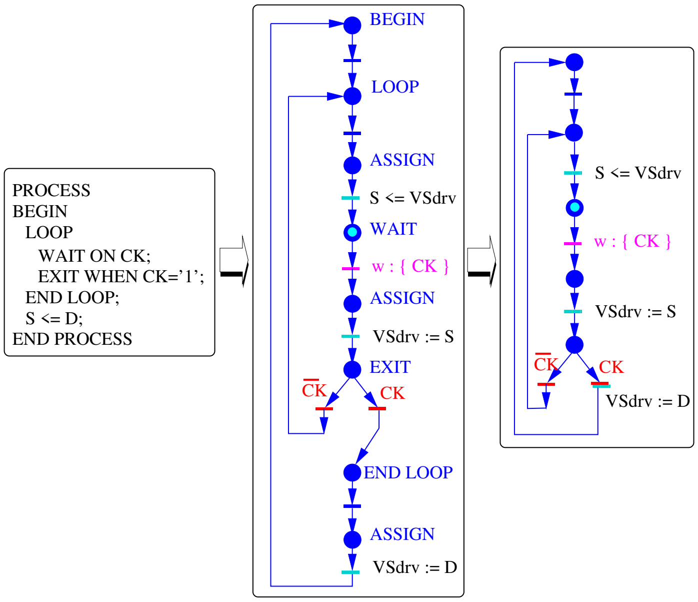

Figure 5.24: L'échec de la réduction locale.

5.2.6 La réduction globale du graphe de contrôle

On exécute l'algorithme de réduction globale décrit ci-après lorsque la réduction locale n'a pas permis d'obtenir la forme canonique. Notons cependant que cet algorithme de réduction globale est très général : il peut être utilisé sur un graphe de contrôle n'ayant subit aucune réduction locale. La réduction locale peut donc être vu comme une optimisation visant à réduire de façon significative à la fois le nombre total de chemins d'exécution du graphe de contrôle et le nombre de transitions présentes sur l'ensemble des chemins d'exécution.

La définition d'un chemin d'exécution

Un chemin d'exécution $Path(t_{from}, t_{to})$ est une liste ordonnée de n transitions t_i consécutives de type T_{exe} reliant le point de suspension t_{from} au point de suspension t_{to}. Dans le cas particulier des processus à un seul point de suspension, les deux transitions t_{from} et t_{to} sont identiques.

Un chemin d'exécution $Path(t_{from}, t_{to}) = \{t_1, \dots, t_n\}$ a les propriétés suivantes :

- les deux transitions t_{from} et t_{to} sont de type T_{res} ;
- les n transitions t_i du chemin d'exécution sont de type T_{exe} ;
- toutes les transitions du chemin d'exécution sont séquentiellement connectées : $\forall i \in \{0, \dots, n\}, \exists p_i \in P$ tel que $ArcTrans(t_i, p_i) = 1$ et $ArcPlace(p_i, t_{i+1}) = 1$ avec $t_0 = t_{from}$ et $t_{n+1} = t_{to}$;

L'énumération de tous les chemins d'exécution

On note t_{wait} la transition de type T_{res} qui correspond au point de suspension du processus. Le graphe de contrôle est constitué d'un ensemble de m chemins d'exécution $Path^j$ différents. La méthode de réduction globale repose sur une énumération explicite de tous ces chemins d'exécution. On construit donc la liste des transitions présentes sur chacun des chemins $Path^j$. La figure 5.25 illustre sur un exemple cette étape d'énumération.

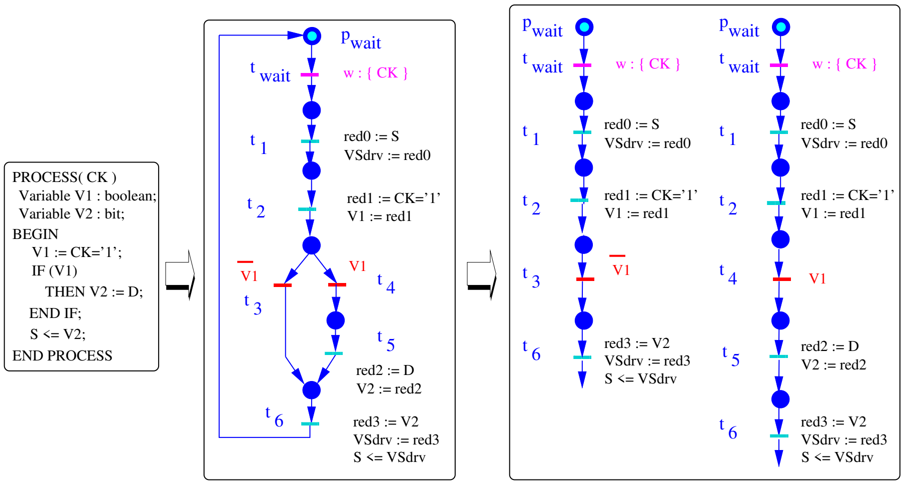

Figure 5.25: L'énumération des chemins d'exécution.

La construction du graphe de contrôle semi-réduit

On construit à partir des listes de transitions de chacun des chemins d'exécution, un graphe de contrôle semi-réduit dont la structure est illustrée sur la figure 5.26.

Cette construction est effectuée de la façon suivante :

1. on conserve la place p_{wait} et la transition t_{wait} de type T_{res} correspondant au point de suspension ;

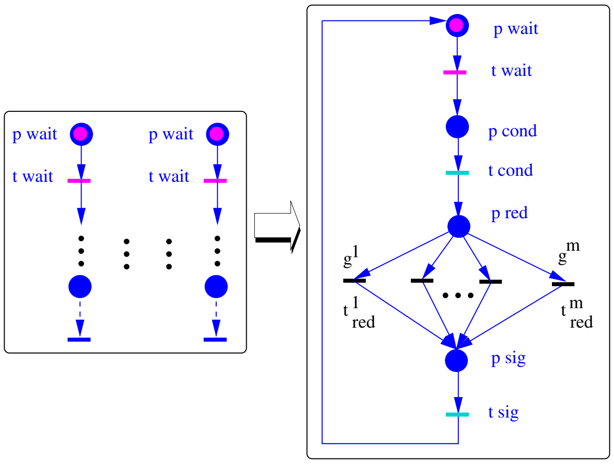

Figure 5.26: La structure du graphe de contrôle semi-réduit.

2. une place p_{cond} et une transition t_{cond} de type T_{exe} sont créées, ainsi qu'un arc reliant p_{cond} à t_{cond}. La liste $Trf(t_{cond})$ regroupe toutes les affectations de variables $red_k \in V^{red}$ qui sont effectuées sur les différents chemins d'exécution ;

3. pour chaque chemin $Path^j$ on crée une transition t^j_{red} de type T_{exe} qui produit le même comportement que l'exécution séquentielle de toutes les transitions de $Path^j$. Chaque transition t^j_{red} possède une condition booléenne de franchissement $g(t^j_{red})$ qui est le produit de toutes les conditions g^j_k présentes sur le chemin $Path_j$. Les conditions g^j_k peuvent dépendre de la valeur des variables affectées sur le chemin $Path^j$ et les affectations de variables red_k présentes sur la transition t_{cond} permettent d'en calculer la valeur. L'algorithme de création des listes $Trf(t^j_{red})$ et des conditions de franchissement $g(t^j_{red})$ est détaillé dans les paragraphes suivants ;

4. une place p_{sig} et une transition t_{sig} de type T_{exe} sont créées, ainsi que les arcs reliant p_{sig} à t_{sig} et t_{sig} à p_{wait}. Les m transitions t^j_{red} sont ensuite reliées par des arcs à la place p_{sig}. La liste $Trf(t_{sig})$ regroupe toutes les affectations de signaux $s \Leftarrow v_{Sdrv}$ qui sont effectuées par le processus. L'algorithme de création de $Trf(t_{sig})$ est détaillé dans les paragraphes suivants.

La construction des affectations de signaux du graphe de contrôle semi-réduit

La phase initiale de la réduction a transformé toutes les affectations de signaux s en affectations de variables v_{Sdrv}. Elle a ensuite ajouté une unique affectation $s \Leftarrow v_{Sdrv}$ pour chacun des signaux s, juste avant le point de suspension t_{wait} du processus. Aussi toutes les affectations de signaux sont identiques sur tous les chemins d'exécution. Pour construire la liste des affectations $Trf(t_{sig})$ de la transition t_{sig} il suffit donc simplement de parcourir un des chemins $Path^j$ précédemment énumérés et de recopier toutes les affectations de signaux sur la transition t_{sig}.

La construction des conditions de franchissement et des affectations de variables du graphe de contrôle semi-réduit

L'expansion du graphe de contrôle semi-réduit : On recopie séquentiellement immédiatement après la transition t^j_{red} toutes les transitions de type T^{exe} qui sont présentes sur le chemin $Path^j$. Le résultat de cette opération pour l'exemple de la figure 5.25 est illustré sur la figure 5.27.

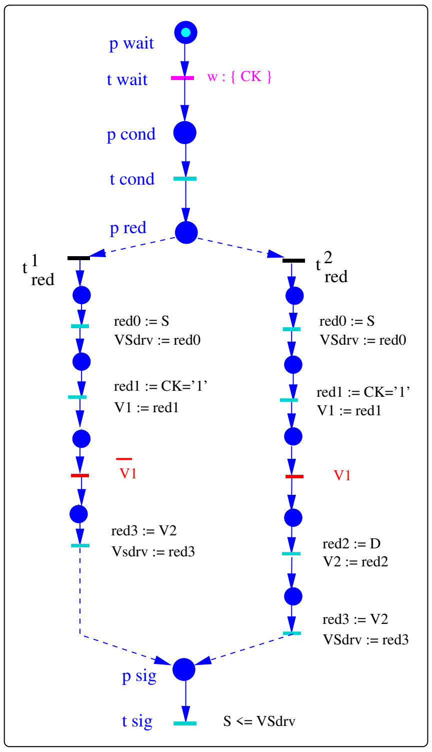

Figure 5.27: Le résultat final de l'expansion du graphe de contrôle semi-réduit.

Comme nous l'avons vu dans la section 5.2.3 les variables $red_k \in V^{red}$ sont affectées au plus une fois sur une seule transition du graphe de contrôle et elles sont toujours affectées avant d'être lues. Cependant une même affectation (red_k, f_k) peut apparaître sur plusieurs chemins d'exécution. Sur la figure 5.27 par exemple, on peut remarquer que les variables red_0, red_1 et red_3 sont affectées sur deux transitions différentes.

L'expression f_k est identique sur toutes les transitions où la variable red_k est affectée, mais cela n'implique pas pour autant que la variable red_k prenne la même valeur quelque soit le chemin emprunté. Sur l'exemple de la figure 5.27 l'expression v_2 est affectée à la variable red_3 et la variable v_2 est affectée différemment sur les deux chemins d'exécution. Sur le premier chemin v_2 conserve sa valeur alors que sur le second elle prend la valeur de l'entrée d. Les deux variables red_0 et red_1 au contraire, prennent la même valeur quelque soit le chemin emprunté puisqu'elles ne dépendent que de la valeur de signaux.

La réduction des chemins du graphe de contrôle semi-réduit : On applique ensuite de façon itérative, une technique de réduction séquentielle entre la transition t^j_{red} et la transition t_i qui la suit. Cette technique vise trois objectifs principaux :

- toutes les affectations de variables $red \in V^{red}$ présentes sur l'une des transitions du chemin $Path^j$ doivent être déplacées sur la transition t_{cond} ;
- la transition t^j_{red} doit contenir les affectations de variables $v \in V^{ini}$ qui sont effectuées sur le chemin $Path^j$. Lorsqu'une variable $v \in V^{ini}$ est affectée plusieurs fois, alors seule la dernière affectation doit être conservée sur t^j_{red} ;
- la condition de franchissement $g(t^j_{red})$ doit être le produit de toutes les conditions de franchissement présentes sur les transitions du chemin $Path^j$.

La réduction séquentielle des transitions t^j_{red} et t_i est effectuée de la façon suivante :

1. comme nous l'avons vu précédemment les variables $red \in V^{red}$ peuvent être affectées sur plusieurs transitions et surtout prendre des valeurs différentes suivant le chemin d'exécution qui est emprunté, aussi on cherche à identifier et à renommer ces variables. On parcourt l'ensemble des affectations $(red_k, f_k) \in Trf(t_i)$ et si $red_k \in V^{red}$ et si red_k est affectée sur plusieurs chemins d'exécution et enfin si l'expression qui lui est affectée fait intervenir des variables alors on doit la renommer en procédant de la façon suivante :
   - on crée une nouvelle variable $red_{j\,k}$ et on ajoute $red_{j\,k}$ dans l'ensemble V^{red} ;
   - on substitue $red_{j\,k}$ à red_k dans toutes les affectations et toutes les conditions de garde présentes sur les transitions du chemin $Path^j$.
2. on propage la valeur des variables affectées par $Trf(t^{red}_j)$ dans la condition de franchissement $g(t_i)$ en faisant :
   pour chaque variable $v_k \in V^{ini}$ qui apparaît dans $Supp(g(t_i))$ on substitue dans $g(t_i)$ toutes les occurences de v_k par red_k s'il existe une affectation (v_k, red_k) dans la liste des affectations de la transition t^{cond}.
3. la condition booléenne $g(t_i)$ est ensuite déplacée sur la transition t^{red}_j :
   $g(t^{red}_j) \leftarrow g(t^{red}_j) \wedge g(t_i)$, où $\wedge$ est l'opérateur *ET* booléen.
4. on applique une réduction locale séquentielle entre les deux listes d'affectations $Trf(t^{red}_j)$ et $Trf(t_i)$ et on obtient ainsi une nouvelle liste d'affectations $Trf'(t^{red}_j)$ équivalente ;

5. on déplace ensuite séquentiellement les affectations de variables $red_k \in V^{red}$ de la liste $Trf'(t_j^{red})$ à la fin de la liste $Trf(t_{cond})$. Après cette opération la liste $Trf'(t_j^{red})$ ne contient plus que des affectations de variables $v_k \in V^{ini}$ et elle devient la nouvelle liste d'affectations de la transition t_j^{red}.

Cette réduction séquentielle est répétée jusqu'à ce qu'il ne reste plus aucune transition t_i sur le chemin $Path^j$.

L'application de cette technique de réduction d'un chemin du graphe de contrôle semi-réduit est illustrée sur la figure 5.28.

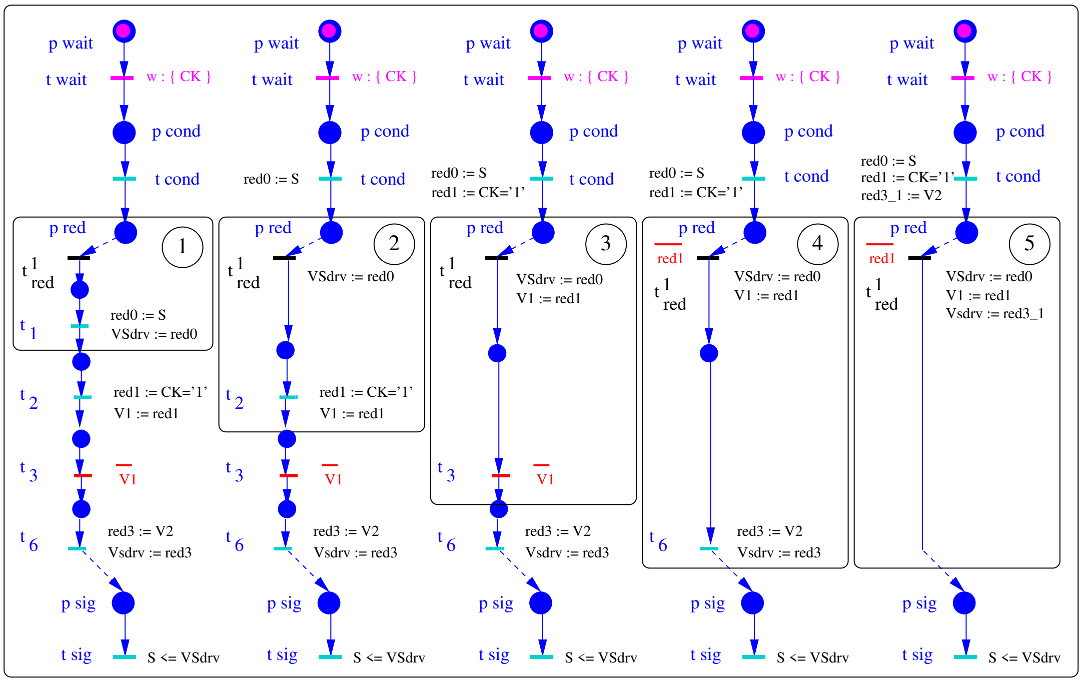

Figure 5.28: La réduction d'un chemin sur le graphe de contrôle semi-réduit.

Lors du passage du graphe 5.28(1) au graphe 5.28(2) l'affectation de la variable red_0 est simplement déplacée sur la transition t_{cond}. Il est inutile en effet de la renommer puisqu'elle ne dépend que du signal s et elle prend donc la même valeur sur tous les chemins d'exécution. L'affectation de la variable v_{Sdrv} quant à elle, est simplement déplacée sur la transition t^1_{red} lors de l'application de la réduction locale séquentielle entre t^1_{red} et t_1.

Lors du passage du graphe 5.28(3) au graphe 5.28(4) la variable v_1 est substituée par red_1 dans la condition de franchissement. Cette condition est ensuite déplacée sur la transition t^1_{red}.

Lors du passage du graphe 5.28(4) au graphe 5.28(5), la variable red_3 est renommée en red_{31} avant d'être déplacée sur la transition t_{cond} puisqu'elle est affectée sur les deux chemins d'exécution et qu'elle dépend d'une variable.

Une fois que les listes d'affectations et les conditions de franchissement de toutes les transitions t^j_{red} ont été construites, toutes les affectations de variables $red_k \in V^{red}$ sont présentes sur la transition t_{cond}.

La réduction finale du graphe de contrôle semi-réduit

La réduction finale du graphe de contrôle semi-réduit est effectuée ainsi :

- on applique une réduction latérale sur toutes les transitions t^j_{red} présentes entre les deux places p_{red} et p_{sig} ;
- on applique ensuite successivement deux réductions séquentielles entre les deux places p_{cond} et p_{sig}.

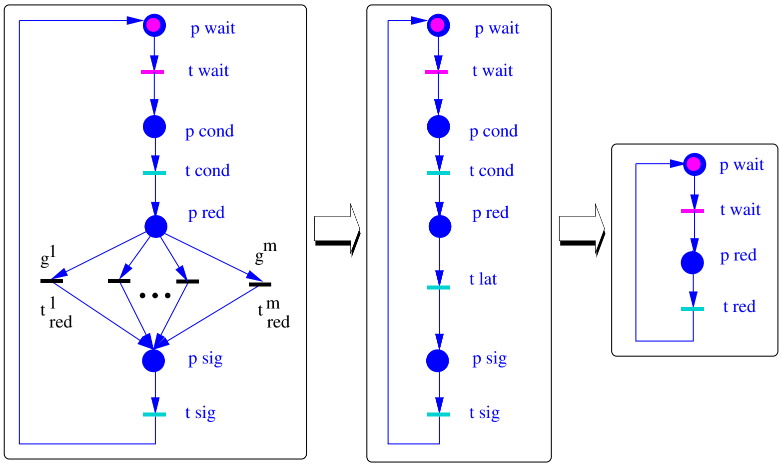

Figure 5.29: La réduction finale du graphe de contrôle semi-réduit.

Le résultat des modifications structurelles apportées au sous graphe de contrôle semi-réduit est illustré sur la figure 5.29.

Le graphe de contrôle réduit ne comporte plus qu'une seule liste d'affectations $Trf(t_{red})$. Tout les variables v_{Sdrv} de V^{Sdrv} qui ont été ajoutées lors de la phase d'initialisation de la réduction sont affectées une seule fois et ce toujours avant d'être lues. Après la phase de réduction globale elles sont donc ajoutées à l'ensemble V^{red}.

5.3 Les propriétés du graphe de contrôle réduit

Le graphe de contrôle réduit possède des propriétés qui seront exploitées lors de la phase d'analyse, c'est à dire lors de l'identification formelle des éléments mémorisants et des barrières trois états.

5.3.1 Les propriétés structurelles

Le graphe de contrôle complètement réduit d'un processus élaboré est toujours constitué de deux transitions t_{red} et t_{wait} qui sont respectivement de type T^{exe} et T^{res}, reliées par des arcs à deux places p_{red} et p_{wait}. Cette structure possède la propriété d'être minimale et canonique. Par conséquent elle permet à la phase d'analyse de s'affranchir complètement du style initial de la description VHDL.

La figure 5.6 présente cette structure minimale et canonique d'un graphe de contrôle complètement réduit. La transition t_{wait} représente l'unique point de suspension du processus et elle contient sa liste de sensibilité. La transition t_{red} représente, sous la forme d'une liste d'affectations séquentielles, toutes les modifications qui sont apportées aux variables et aux signaux lors du réveil du processus.

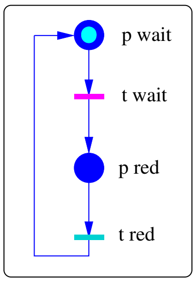

Figure 5.30: Les propriétés structurelles du graphe de contrôle réduit.

Tous les chemins d'exécution ont été ramenés à une seule transition t_{exe}. Par conséquent, comme nous le verrons dans le chapitre 6, l'identification formelle des registres et des barrières trois états s'effectue en étudiant uniquement la liste d'affectations de la transition t_{red} et la liste de sensibilité de la transition t_{wait}.

5.3.2 Les propriétés de l'unique liste d'affectations

Contrairement à la structure du graphe de contrôle réduit, l'unique liste d'affectations $Trf(t_{red})$ dépend de la structure initiale du processus et par conséquent cette liste n'est ni canonique, ni minimale. Elle possède cependant les propriétés suivantes :

- cette liste est séquentielle et tous les symboles sont affectés une seule fois. Ainsi seule la dernière valeur donnée à un signal ou à une variable est conservée. Les problèmes liés aux affectations multiples de signaux et de variables sont donc complètement résolus.

- toutes les variables $v \in V^{ini}$ qui sont affectées avant d'être lues ont été remplacées par une variable *red* dans tous les membres de droite de toutes les affectations ;

- toutes les variables $red \in V^{red}$ sont affectées avant d'être lues. Une conséquence directe de cette propriété est que ces variables ne peuvent en aucun cas se comporter comme des registres. Elles servent uniquement à propager des valeurs sur les pilotes des signaux et sur les variables de l'ensemble V^{ini}. Lors de l'analyse formelle elles seront soit détruites lorsqu'elles ne sont pas utilisées, soit implantées à l'aide de portes combinatoires.

Ces trois propriétés sont essentielles pour l'analyse formelle, comme nous le verrons dans le chapitre 6.

5.4 Conclusion

Dans ce chapitre nous avons tout d'abord identifié les deux classes de problèmes à résoudre lors de l'analyse d'un processus à un seul point de suspension à savoir :

1. comment extraire, quelle que soit la structure du graphe de contrôle, les affectations qui peuvent être appliquées à un symbole et les conditions associées à chacune de ces affectations?
2. comment analyser pour chaque symbole, ces conditions et ces affectations afin d'identifier formellement les registres, les barrières trois états et les sorties de portes combinatoires?

Ceci nous a conduit à découper en deux étapes l'analyse d'un processus à un seul point de suspension : une étape de réduction de son graphe de contrôle et une étape d'analyse de son graphe de contrôle réduit.

Afin de résoudre la première étape de l'analyse, nous avons proposé une méthode de réduction locale du graphe de contrôle d'un processus élaboré s'appuyant sur la structure des instructions du langage VHDL. Nous avons détaillé les deux techniques de réduction locale (séquentielle et latérale) qui servent de base à cette méthode de réduction.

Nous avons ensuite montré qu'il existait des cas pour lesquels la méthode de réduction locale ne permettait pas d'obtenir un graphe de contrôle minimal et qu'il était nécessaire d'introduire une seconde méthode de réduction.

Nous avons donc introduit la méthode de réduction globale, basée sur l'énumération de tous les chemins d'exécution possibles. Nous avons montré comment cette méthode permettait d'obtenir un graphe de contrôle minimal quelque soit la structure du graphe initial.

Enfin nous avons énuméré les propriétés structurelles du graphe de contrôle réduit ainsi que les propriétés de son unique liste d'affectations.

Toutes ces propriétés seront exploitées lors de l'identification formelle des éléments mémorisants et des barrières trois états décrite dans le chapitre 6.

CHAPITRE 6

Une méthode d'analyse du graphe de contrôle réduit des processus possédant un seul point de suspension

Sommaire

Dans ce chapitre nous développons une méthode d'analyse permettant d'identifier formellement, pour tous les symboles affectés par un processus, le type d'élément matériel à utiliser pour obtenir un résultat de synthèse en conformité avec la sémantique de simulation : porte combinatoire, registre ou barrière trois états. Cette méthode d'analyse est appliquée une fois que le graphe de contrôle du processus à un seul point de suspension a été complètement réduit par les techniques précédemment exposées au chapitre 5.

Notre méthode d'analyse comporte trois étapes. La première étape permet d'identifier *a priori*, pour chaque symbole affecté le type de matériel nécessaire à son implantation. La seconde étape détermine de façon formelle toutes les propriétés des registres et des barrières trois états précédemment identifiés, en utilisant des techniques basées sur les graphes de décision binaire réduits et ordonnés (Reduced Ordered Binary Decision Diagram)[Bry86]. La dernière étape vérifie la cohérence de ces résultats avec la liste de sensibilité du processus.

6.1 L'analyse préliminaire des affectations

Comme nous l'avons vu au chapitre 5, le graphe de contrôle réduit d'un processus à un seul point de suspension est composé d'une transition t_{red} de type T_{exe} et d'une transition t_{wait} de type T_{res}.

La phase d'analyse préliminaire des affectations substitue tout d'abord les occurences des constantes par leur valeur, puis elle construit le graphe de dépendance des affectations de $Trf(t_{seq})$. Elle identifie ensuite parmi tous les symboles affectés ceux qui doivent être pris en compte par la synthèse et enfin elle détermine *a priori* le type de matériel nécessaire à leur l'implantation.

6.1.1 La substitution des constantes

Comme nous l'avons vu au chapitre 4 dans la section 4.4, certains signaux et certaines variables doivent être considérés comme des constantes lorsqu'ils ne sont plus affectés après la phase d'initialisation de notre modèle.

On parcourt toutes toutes les affectations (d, f) de $Trf(t_{red})$ et on remplace dans l'expression f ces signaux et ces variables par leur valeur calculée lors de l'initialisation de notre modèle.

6.1.2 Le graphe de dépendance des affectations

L'analyse de la liste $Trf(t_{red})$ s'appuies sur le graphe qui exprime les dépendances entre les symboles présents dans les affectations. À partir de ce graphe, elle construit pour chaque affectation (d, f) deux ensembles $SuppExt(f)$ et $CsteExt(f)$ qui représentent respectivement le support étendu de l'expression f et l'ensemble des constantes qui interviennent dans f.

La définition du graphe de dépendance des affectations

On note $G_{dep} = \{Node, Arc\}$ le graphe de dépendance des affectations de la liste $Trf(t_{red})$. L'ensemble des nœuds $Node$ est composé de trois sous ensembles disjoints $Node^{in}$, $Node^{red}$ et $Node^{out}$ qui sont définis de la façon suivante :

- les nœuds de $Node^{in}$ n'ont pas de prédécesseurs. Ils représentent toutes les constantes $c \in \mathcal{S} = \{'0','1','Z','L','H','-'\}$, tous les signaux $s \in S$ et toutes les variables $v \in V^{ini}$ qui apparaissent dans le membre de droite d'une affectation de $Trf(t_{red})$. Chacun des nœuds de cet ensemble représente la valeur effective d'un symbole au moment où le processus se réveille ;
- les nœuds de $Node^{red}$ représentent toutes les affectations de variables $red \in V^{red}$ de la liste $Trf(t^{red})$. Tout ces nœuds ont au moins un prédécesseur et ils servent à modéliser la propagation de la valeur des variables $red \in V^{red}$ sur les signaux et les variables de V^{ini} ;
- les nœuds de $Node^{out}$ représentent les affectations de signaux et de variables $v \in V^{ini}$ de la liste $Trf(t^{red})$.

On associe à tous les nœuds de $Node^{red}$ et de $Node^{out}$ l'expression du membre de droite de l'affectation qu'ils représentent.

L'ensemble des arcs (n_{from}, n_{to}) est construit en parcourant les affectations (d_{to}, f) de la liste $Trf(t^{red})$. Pour tous les signaux, toutes les constantes et toutes les variables d_{from} qui apparaissent dans l'expression de f on crée un arc reliant le nœud n_{from} au nœud n_{to} où n_{from} et n_{to} sont respectivement les nœuds correspondant à d_{from} et d_{to}. L'ensemble des prédécesseurs d'un nœud n du graphe G_{dep} constitue donc le support de l'expression associée à n.

La figure 6.1 présente la description d'un processus, son graphe de contrôle complètement réduit et le graphe de dépendance des affectations de $Trf(t_{red})$.

Une propriété essentielle de ce graphe est qu'il n'existe pas de cycle :

- toutes les variables $red \in V^{red}$ sont toujours affectées avant d'être lues, il n'existe donc pas de chemin permettant d'aller d'un nœud $node \in Node^{red}$ vers lui même ;

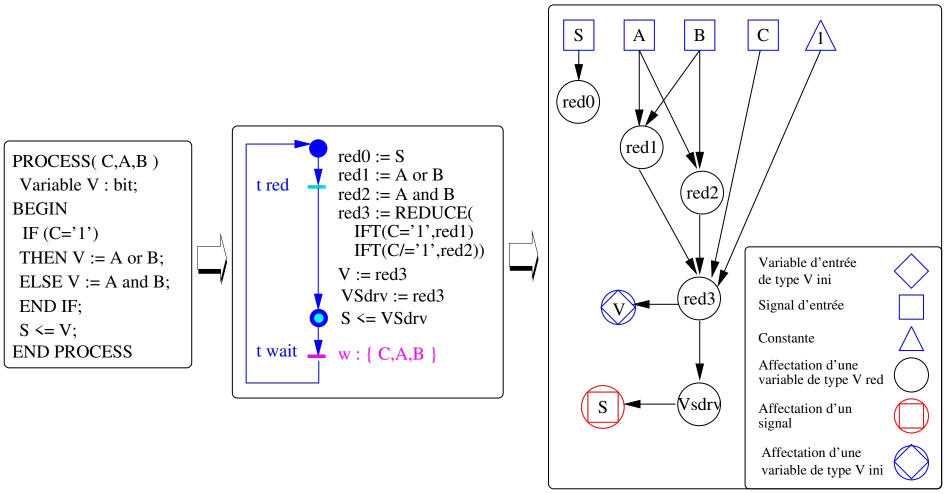

Figure 6.1: Le graphe de dépendance des affectations.

- pour tous les symboles qui ne sont pas des variables $red \in V^{red}$ nous avons créé un nœud différent pour modéliser la valeur qu'ils ont au moment du réveil du processus, et la valeur qu'ils prennent lorsque le processus s'endort. Par conséquent il ne peut y avoir de cycle entre ces différents nœuds.

La définition du support étendu d'une expression

Soit une affectation (d, f_d) de $Trf(t_{red})$. On note $SuppExt(f_d)$ le support étendu de l'expression f_d, c'est à dire l'ensemble des bits des signaux et des variables qui interviennent directement où indirectement dans f_d.

La définition est récursive et on a $SuppExt(f_d) = Supp(f_d) \bigcup_{red_k} [SuppExt(f_{red_k})]$ avec $red_k \in Supp(f_d) \cap V^{red}$, où f_{red_k} est l'expression affectée à la variable $red_k \in V^{red}$.

Ainsi sur le graphe de dépendance des affectations, le support étendu d'une expression affectée f_d représente l'ensemble des prédécesseurs transitifs du nœud d en excluant tous les nœuds qui représentent une constante. La figure 6.2 illustre le calcul du support étendu.

Dans le cas où d et f sont des vecteur de bits, on note $SuppExt^b(f_d)$ le support étendu du bit de rang b de l'expression f_d. La figure 6.2 présente une liste d'affectations et le support étendu $SuppExt^b(f_d)$ de chacune des expressions f_d affectées.

La notion de support étendu est primordiale dans l'identification des registres. Un registre est un élément matériel qui conserve sa valeur lorsque sa condition d'écriture est fausse et qui ne peut changer de valeur que lorsqu'elle est vraie. Par conséquent, du point de vue de la synthèse une condition nécessaire pour qu'un symbole r soit un registre est que l'expression qu'il lui est affectée fasse intervenir, directement ou indirectement, le symbole r lui même, c'est à dire la valeur qu'il avait au moment où le

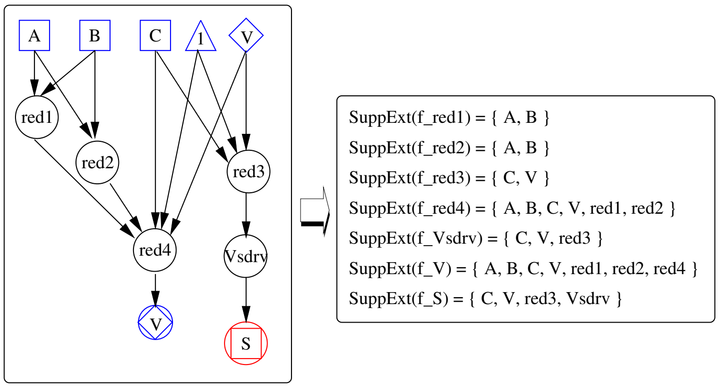

Figure 6.2: Le support étendu d'une expression affectée.

```
red0  := A(0 to 1);
red1  := B(0 to 1);
VSdrv := red0 AND
         red1;
S <= VSdrv;
```

$$SuppExt^0(f_{red_0}) = \{A(0)\}$$
$$SuppExt^1(f_{red_0}) = \{A(1)\}$$
$$SuppExt^0(f_{red_1}) = \{B(0)\}$$
$$SuppExt^1(f_{red_1}) = \{B(1)\}$$
$$SuppExt^0(f_{v_{Sdrv}}) = \{A(0), B(0), red_0(0), red_1(0)\}$$
$$SuppExt^1(f_{v_{Sdrv}}) = \{A(1), B(1), red_0(1), red_1(1)\}$$
$$SuppExt^0(f_s) = \{A(0), B(0), red_0(0), red_1(0), v_{Sdrv}(0)\}$$
$$SuppExt^1(f_s) = \{A(1), B(1), red_0(1), red_1(1), v_{Sdrv}(1)\}$$

Figure 6.3: Le support étendu d'une expression au niveau bit.

processus s'est réveillé.

La figure 6.4 présente une telle configuration sur le graphe de dépendance des affectations. Sur cet exemple il existe un chemin reliant le nœud représentant la valeur effective du signal s au moment du réveil du processus et le nœud représentant l'affectation de s. On a donc $s \in \ SuppExt(f_s)$ et on peut par conséquent supposer que le signal s se comporte comme un élément mémorisant.

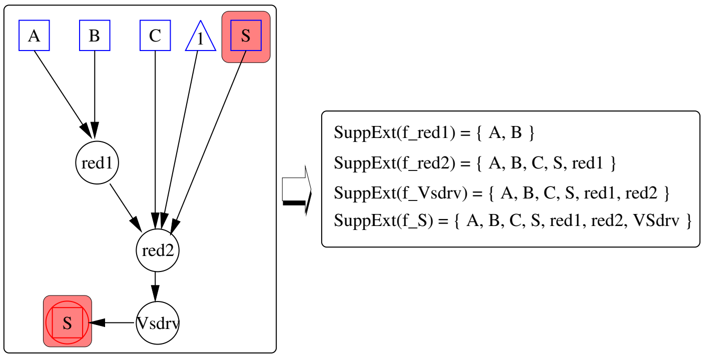

Figure 6.4: L'identification d'un élément mémorisant.

Comme nous l'avons évoqué dans le chapitre 2, la phase d'analyse de la compilation considère chacun des processus indépendamment des autres. Une autre hypothèse commune à tous les outils de synthèse est faite ici : les registres sont identifiables en analysant le comportement permanent d'un processus sur un unique delta-cycle.

Dans le cas du processus de la figure 6.5, les deux signaux s et t sont tels que $s \notin SuppExt(f_s)$ et $t \notin SuppExt(f_t)$.

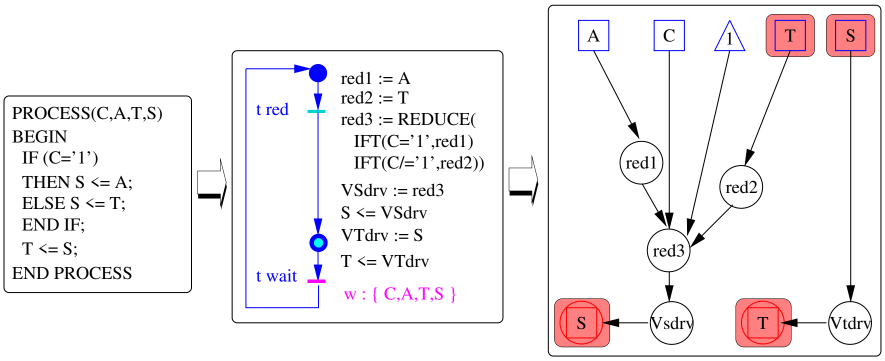

Figure 6.5: L'analyse du comportement d'un processus sur un seul delta-cycle.

Cependant si on considère le comportement de ce processus sur deux delta-cycles, c'est à dire si on considère deux réveils consécutifs du processus on observe alors que le signal s peut conserver sa valeur. En effet si le signal s est modifié lors de l'exécution du premier delta-cycle alors le processus est de nouveau réveillé et la nouvelle valeur de s est propagée sur t lors du deuxième delta-cycle. On peut conclure que s et t ont toujours la même valeur et par conséquent que s dépend de t, c'est à dire de lui même.

On considère que la description de la figure 6.5 n'est pas synthétisable et elle sort donc du cadre de notre méthode d'analyse.

Les constantes qui interviennent dans une affectation

Soit une affectation (d, f_d) de $Trf(t_{red})$. On note $Cste(f_d)$ l'ensemble des éléments de $\mathcal{S}$={$'0','1','Z','L','H','-'$} qui interviennent directement dans l'expression f_d. On note également $CsteExt(f_d)$ l'ensemble des éléments de $\mathcal{S}$ qui interviennent directement ou indirectement dans f_d.

On a $CsteExt(f_d) = Cste(f_d) \bigcup_{red_k} [CsteExt(f_{red_k})]$ avec $red_k \in Supp(f_d) \cap V^{red}$, où f_{red_k} est la fonction affectée à la variable $red_k \in V^{red}$.

Ainsi sur le graphe de dépendance des affectations, l'ensemble $CsteExt(f_d)$ d'une expression affectée f_d est l'ensemble des prédécesseurs transitifs du nœud d qui représentent une constante. La figure 6.6 illustre le calcul de l'ensemble $CsteExt(f_d)$.

Dans le cas où d et f_d sont des vecteur de bits, on note $CsteExt^b(f_d)$ l'ensemble des constantes apparaissant sur le bit de rang b de l'expression f_d. La figure 6.7 présente une liste d'affectations (d, f_d) et pour chacune des expressions f_d la valeur de $CsteExt^b(f_d)$.

Les ensembles $CsteExt(f_d)$ de chacune des affectations (d, f_d) sont utilisés pour identifier les barrières trois états. Une barrières trois états est un élément matériel dont

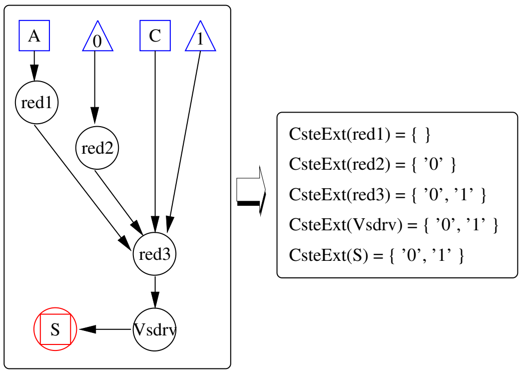

Figure 6.6: Les constantes apparaissant dans une expression affectée.

```
red0  := "10";
red1  := A OR B;
VSdrv  := red0 AND
          red1;
S <= VSdrv;
```

$$CsteExt^0(f_{red_0}) = \{'1'\}$$
$$CsteExt^1(f_{red_0}) = \{'0'\}$$
$$CsteExt^0(f_{red_1}) = \varnothing$$
$$CsteExt^1(f_{red_1}) = \varnothing$$
$$CsteExt^0(f_{v_{Sdrv}}) = \{'1'\}$$
$$CsteExt^1(f_{v_{Sdrv}}) = \{'0'\}$$
$$CsteExt^0(f_s) = \{'1'\}$$
$$CsteExt^1(f_s) = \{'0'\}$$

Figure 6.7: Les constantes apparaissant sur un des bits d'une expression.

la sortie est en haute impédance lorsque sa condition d'émission est fausse et qui laisse passer une donnée lorsque cette condition est vraie. En synthèse, l'état haute impédance est modélisé par la constante $'Z'$. Par conséquent une condition nécessaire pour qu'un symbole s soit une barrières trois états est que l'expression qu'il lui est affectée fasse intervenir directement ou indirectement cette constante.

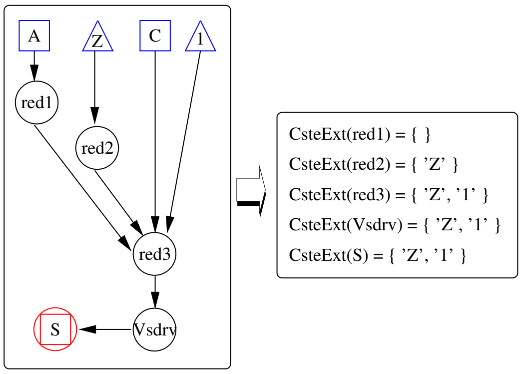

Figure 6.8: L'identification d'une barrière trois états.

La figure 6.8 présente une telle configuration sur le graphe de dépendance des affectations. Sur cet exemple il existe un chemin reliant le nœud représentant la constante $'Z'$ et le nœud représentant l'affectation de s. On a $'Z' \in \; CsteExt(f_s)$ et on peut par conséquent supposer que s se comporte comme une barrière trois états.

6.1.3 Les symboles utiles à la synthèse

Comme nous l'avons vu dans la section 5.1, la synthèse ne doit prendre en compte que les variables qui interviennent dans le calcul du pilote d'un signal.

Sur l'exemple de la figure 6.9 seules les variables red_4 et v_{Sdrv} doivent être conservées puisqu'elles interviennent directement ou indirectement dans le calcul de la valeur du pilote du signal s.

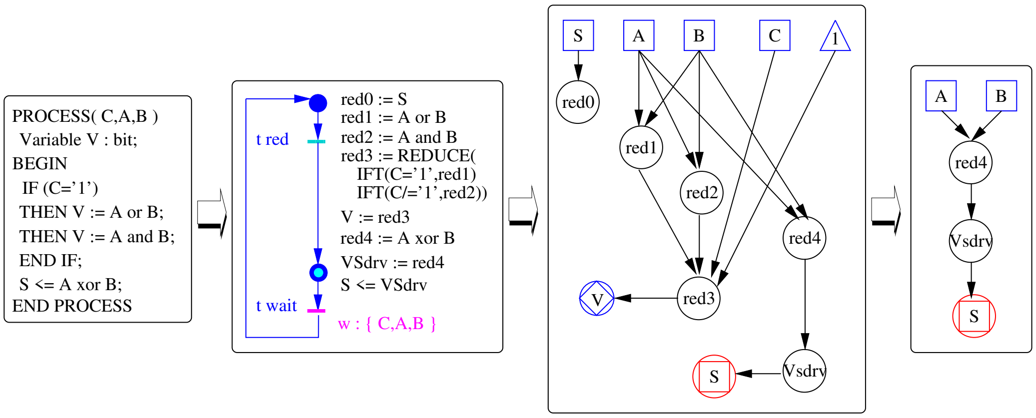

Figure 6.9: Les symboles utiles à la synthèse.

Sur l'exemple de la figure 6.10 on doit conserver les variables de V^{red} qui interviennent dans le calcul du pilote du signal s, c'est à dire v, red_0, red_1 red_3 et v_{Sdrv}. Cependant s dépend de la variable $v \in V^{ini}$ et on doit aussi conserver l'ensemble des symboles qui permettent d'en calculer la valeur, c'est à dire red_2 et red_4.

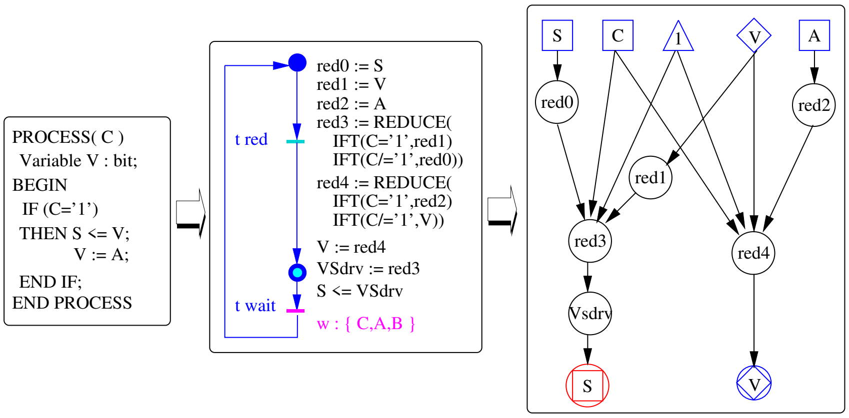

Figure 6.10: Les variables utiles à la synthèse.

L'ensemble V^{used} des variables que la synthèse doit prendre en compte est donc défini de la façon suivante :

$V^{used} = \bigcup_s [SuppExt(f_s) \bigcup_{v_k} SuppExt(f_{v_k})]$ avec $v_k \in SuppExt(f_s) \cap V^{ini}$, où (s, f_s)

est une affectation de signal de la liste $Trf(t_{red})$ et où f_{v_k} est la fonction affectée à la variable $v_k \in V^{ini}$.

6.1.4 Les variables lues avant d'être affectées

Comme nous l'avons vu dans la 5.3 du chapitre 5, toutes les occurences des variables de V^{ini} qui sont affectées avant d'être lues ont été remplacées par une variable $red \in V^{red}$ lors de la phase de réduction. Cependant les occurences d'une variable $v \in V^{ini}$ n'ont pu être substituées que lorsque la variable v a été préalablement affectée. Ainsi toutes les variables $v \in V^{ini}$ présentes après la phase de réduction dans l'ensemble V^{used} sont nécessairement des variables lues avant d'être affectées.

Sur l'exemple de la figure 6.11, la variable v fait partie de l'ensemble V^{used} puisqu'elle intervient dans le calcul du pilote de s. Elle est lue lors de l'affectation de red_2 avant d'avoir été affectée.

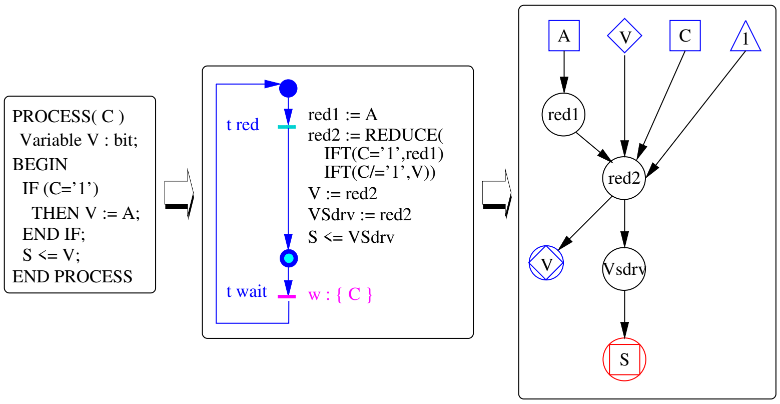

Figure 6.11: Les variables lues avant d'être affectées.

Lorsqu'une variable $v \in V^{ini}$ apparaît dans l'expression droite d'une affectation, c'est son ancienne valeur qui est prise comme valeur effective, c'est à dire celle qui lui a été donnée la dernière fois que le processus s'est réveillé.

Du point de vue de la synthèse il faut mettre en place un dispositif matériel qui permette de conserver l'ancienne valeur de v. Ainsi pour que le processus soit synthétisable, il est nécessaire que la variable v se comporte comme un registre. L'analyse doit donc vérifier que $v \in SuppExt(f_v)$. Si ce n'est pas le cas le processus n'est pas synthétisable.

Comme on peut le constater sur le graphe de dépendance des affectations de la figure 6.11, la variable v vérifie bien cette règle et elle doit être synthétisée comme un registre.

On note V^{ini}_{reg} l'ensemble des variables du processus qui sont lues avant d'être affectées et qui par conséquent doivent vérifier la règle $v \in SuppExt(f_v)$.

6.1.5 Les variables considérées comme des points d'arrêts

On distingue deux classes de symboles candidats pour être implantés en matériel à l'aide d'éléments mémorisants : les signaux s tels que $s \in \mathit{SuppExt}(f_s)$ et les variables v telles que $v \in V_{reg}^{ini}$.

Les variables $v \in V_{reg}^{ini}$ ne sont pas nécessairement synthétisables. Les conditions qui doivent être respectées pour qu'un symbole soit implantable en matériel à l'aide d'un registre sont décrites dans la section 6.2. Cependant toutes ces variables doivent être considérées comme des points d'arrêts dans l'exploration du graphe de dépendance des affectations.

Par exemple dans le cas du processus de la figure 6.12, la variable v est lue avant d'être affectée et elle se comporte comme une bascule D. Le signal s prend quant à lui systématiquement la valeur de v. Pendant la phase de réduction du graphe de contrôle la variable v a été remplacée par la variable red_2 dans l'affectation de v_{Sdrv}.

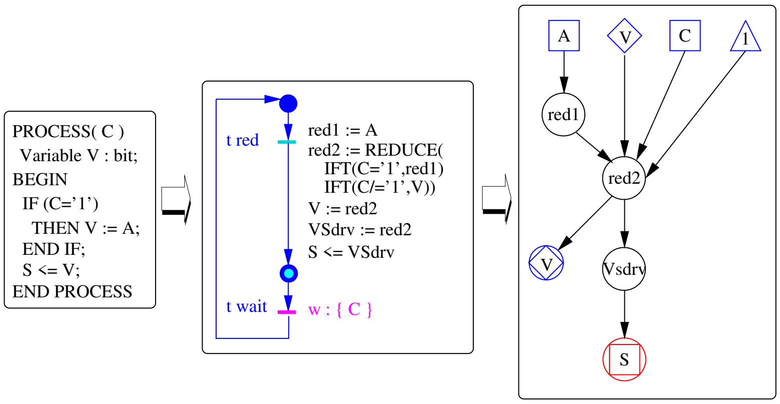

Figure 6.12: La lecture d'une variable de $v \in V_{reg}^{ini}$.

Sur le graphe de dépendance on peut remarquer que s dépend indirectement des signaux A, C et des variables v, red_1 et red_2. Ces symboles servent à calculer la nouvelle valeur du registre v. Du point de vue de la synthèse, lors de l'analyse du signal s on ne s'intéresse pas à la façon de calculer la valeur du registre v, mais seulement au fait que v intervient dans le calcul du signal s. Autrement dit v doit être considérée comme un point d'arrêt pour la synthèse.

La solution à ce problème consiste à remplacer red_2 par v comme c'est illustré sur la figure 6.13. La variable v est un point d'arrêt et lors de l'analyse du signal s on ne tient compte que de sa dépendance avec les variables v_{Sdrv} et v.

Ainsi on fait apparaître explicitement chaque variable $v \in V_{reg}^{ini}$ dans toutes les expressions situées après leur affectation. Pour cela on parcourt séquentiellement les affectations (v, f_v) de la liste $\mathit{Trf}(t_{red})$ telles que $v \in V_{reg}^{ini}$. Après la phase de réduction on a par construction $f_v = red_k$ avec $red_k \in V^{red}$. Ainsi on parcourt toutes les affectations (d_i, f_i) situées après l'affectation de v et on renomme dans f_i les occurences de

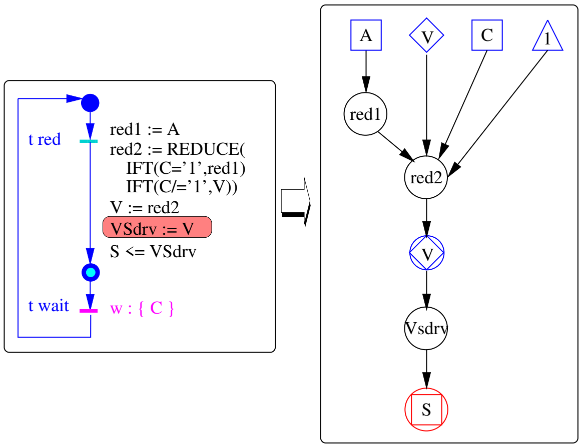

Figure 6.13: Le renommage des variables qui se comportent comme des registres.

red_k par v.

6.1.6 Les contraintes liées à la lecture d'une variable de V_{reg}^{ini}

Le fait qu'une variable v soit lue avant d'être affectée impose des contraintes de synthèse sur la variable elle même car elle doit nécessairement être implantée à l'aide d'un registre. Dans certains cas, cela impose aussi des contraintes sur les symboles qui lisent cette variable v.

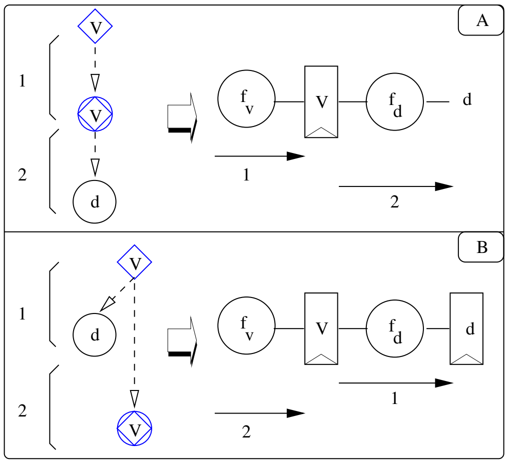

Figure 6.14: Les contraintes liées à lecture d'une variable de V_{reg}^{ini}.

Soit une affectation (d, f_d) et une variable $v \neq d$ lue avant d'être affectée et faisant partie du support étendu de f_d. On distingue les deux cas suivants qui sont illustrés sur la figure 6.14 :

- si v est affectée avant d, lorsque la variable v est modifiée, sa nouvelle valeur est immédiatement propagée sur d. La figure 6.14(A), présente l'implantation matérielle correspondante à ce comportement. Si le processus est synthétisable, la variable v est un registre dont l'entrée est la sortie f_v d'un ensemble de portes combinatoires. Le symbole d n'est soumis à aucune contrainte particulière ;

- si v est affectée après d, lorsque la variable v est modifiée, sa nouvelle valeur n'est pas répercutée sur d. Il faudra donc attendre que le processus se réveille de nouveau pour que d soit mis à jour. Ainsi pour que d puisse être synthétisé il est nécessaire qu'il se comporte comme un registre. La figure 6.14(B), présente l'implantation matérielle correspondante à ce comportement. Si le processus est synthétisable, la variable v est un registre dont l'entrée est la sortie f_v d'un ensemble de portes combinatoires, et le symbole d est un registre dont l'entrée est la sortie du bloc combinatoire f_d.

6.1.7 L'analyse du support des expressions

On note $S^{used} = Asg(t_{red}) \cap S$ l'ensemble des signaux affectés par le processus après la phase d'initialisation. L'analyse préliminaire des affectations doit permettre d'identifier pour tous les bits de tous les symboles $d_k \in V^{used} \cup S^{used}$ le type de matériel nécessaire à leur implantation.

Du point de vue de la synthèse on distingue trois catégories de symboles :

- les variables $v \in V_{reg}^{ini}$. Comme nous l'avons vu précédemment pour que le processus soit synthétisable il est nécessaire que ces variables se comportent comme des registres ;
- les variables red_k de l'ensemble V^{used}. Elles sont toujours affectées avant d'être lues et elles ne peuvent en aucun cas se comporter comme des registres ou des barrières trois états. Elles sont implantées à l'aide de portes combinatoires ;
- les signaux s affectés par le processus. Un signal peut se comporter comme un registre, une barrière trois états ou encore comme la sortie d'une porte combinatoire.

Une première classification est effectuée en étudiant le support étendu et les constantes qui apparaissent dans les différentes expressions affectées. Cette classification est effectuée bit par bit puisque tous les bits d'un symbole ne sont pas nécessairement implantés à l'aide du même type de matériel.

On parcourt toutes les affectations (d_j, f_j) de $Trf(t_{red})$ telles que $d_j \in V^{used} \cup S^{used}$ et pour chacun des bits d_j^b de d_j on effectue l'analyse suivante :

1. si d_j^b est un signal alors

- si $d_j^b \in SuppExt^b(f_j)$ alors d_j^b est un registre dont il faut déterminer les différentes propriétés ;
- si $CsteExt^b(f_j)$ contient l'élément $'Z'$ alors il s'agit d'une barrière trois états dont il faut déterminer les propriétés ;
- dans tous les autres cas, d_j^b doit être implanté à l'aide de portes combinatoires. Néanmoins il faut encore vérifier que cette implantation peut se faire en accord avec la liste de sensibilité du processus.

2. si $d_j^b \in V^{ini}$ alors
   - si $d_j^b \in SuppExt^b(f_j)$ alors d_j^b est un registre dont il faut déterminer les différentes propriétés ;
   - dans tous les autres cas d_j^b est une variable lue avant d'être affectée et qui ne se comporte pas comme un registre. Elle ne peut pas être implantée en matériel et l'analyse conclut que le processus n'est pas synthétisable ;
   - on peut remarquer qu'une variable à la différence d'un signal est interne au processus. Sa valeur ne peut pas être modifiée par un autre processus, et par conséquent elle ne peut en aucun cas se comporter comme une barrière trois états.
3. si $d_j^b \in V^{red}$ alors cette variable intervient dans le calcul d'un signal $s \in S^{used}$ ou d'une variable $v \in V^{ini}$. Ce n'est que lors de l'analyse du signal s ou de la variable v qu'on pourra savoir si elle doit ou non être implantée en matériel à l'aide de portes combinatoires.

6.2 L'identification formelle des propriétés des registres

La seconde étape identifie de façon formelle les propriétés des registres qui ont été détectés lors de l'analyse préliminaire des affectations.

6.2.1 L'identification de la condition de mémorisation

L'analyse d'une affectation (d_j, f_j) telle que $d_j^b \in SuppExt^b(f_j)$ doit permettre de trouver la condition c^{mem} qui doit être respectée pour que d_j^b conserve sa valeur. Par la suite on notera r le symbole d_j^b et f l'expression dont on a extrait seulement la partie qui concerne le bit b. L'utilisation des *ROBDDs*[Bry86] pour représenter la fonction f et les différentes propriétés du registre r est implicite dans toute notre méthode d'analyse formelle de r. Les *ROBDDs* sont présentés dans l'annexe B.

La figure 6.15 présente la description d'un processus *P1*, puis les modifications qui lui ont été appliquées avant la réduction et enfin la liste des affectations du graphe de

P1 : PROCESS(CK)
BEGIN
 IF ((CK ='1') AND
     (WEN='1'))
 THEN R <= A OR B;
 END IF;
END PROCESS P1;
	P1 : PROCESS(CK)
	 VARIABLE V_Rdrv : BIT;
	 VARIABLE red_0 : BIT;
	 VARIABLE red_1 : BIT;
	 VARIABLE red_2 : BOOLEAN;
	BEGIN
	 red_0 := R;
	 V_Rdrv := red_0;
	 red_2 := ((CK ='1') AND
	           (WEN='1'));
	 IF (red_2)
	 THEN
	   red_1 := A OR B;
	   V_Rdrv := red_1;
	 END IF;
	 R <= V_Rdrv;
	END PROCESS P1;
		red_0 := R;
		red_1 := A OR B;
		red_2 := (CK='1') AND (WEN='1');
		red_3 := REDUCE(IFT(red_2 , red_1),
		                 IFT(NOT(red_2), red_0));
		V_Rdrv := red_3;
		R <= V_Rdrv;

Figure 6.15: L'identification d'un registre.

contrôle réduit. Le signal r dépend de lui même et l'analyse doit identifier toutes les propriétés du registre r.

L'ajout d'un front dans l'affectation d'un registre

La méthode d'identification des propriétés d'un registre que nous proposons (et en particulier la distinction entre un registre à écriture sur front et un registre à écriture sur état) nécessite que le front du signal d'horloge soit explicite.

Or, lorsque la liste de sensibilité $w(t_{res})$ ne contient qu'un seul signal s alors l'exécution des affectations de $Trf(t_{exe})$ est implicitement conditionnée par la présence d'un front sur le signal s.

De façon à faire apparaître explicitement ce front on transforme l'affectation (r, f) en un nouvelle affectation (r, f') équivalente. L'expression f' est construite de la façon suivante : $f' = REDUCE(IFT(s'Event, f), IFT(NOT(s'Event, r)))$.

```
red_0 := R;
red_1 := A OR B;
red_2 := (CK='1') AND (WEN='1');
red_3 := REDUCE(IFT(    red_2 , red_1),
                 IFT(NOT(red_2), red_0));
V_Rdrv := red_3;
R <= V_Rdrv;
```

$$\Downarrow$$

```
red_0 := R;
red_1 := A OR B;
red_2 := (CK='1') AND (WEN='1');
red_3 := REDUCE(IFT(    red_2 , red_1),
                 IFT(NOT(red_2), red_0));
V_Rdrv := red_3;
R <= REDUCE(IFT(    CK'Event , V_Rdrv),
            IFT(NOT(CK'Event), R     ));
```

Figure 6.16: L'ajout d'un front dans l'affectation d'un registre.

La figure 6.16 illustre la modification de l'affectation du signal r pour faire apparaître explicitement la présence d'un front sur le signal ck.

La substitution des variables

Lors de l'analyse on cherche à conserver les variables intermédiaires car elles peuvent servir pour un partage efficace des ressources matérielles. Cependant certaines d'entres elles doivent être substituées avant l'analyse.

Afin de retrouver la condition de mémorisation c^{mem} il est indispensable que toutes les occurences de r apparaissent directement dans l'expression de f. Soit V_f^{subst} l'ensemble des variables $red_k \in V^{red} \cap SuppExt(f)$ et telles que $r \in SuppExt(f_{red_k})$ où f_{red_k} est l'expression qui est affectée à la variable red_k. On substitue de façon itérative dans f toutes les occurences des variables red_k de l'ensemble V_f^{subst} par les expressions f_{red_k}.

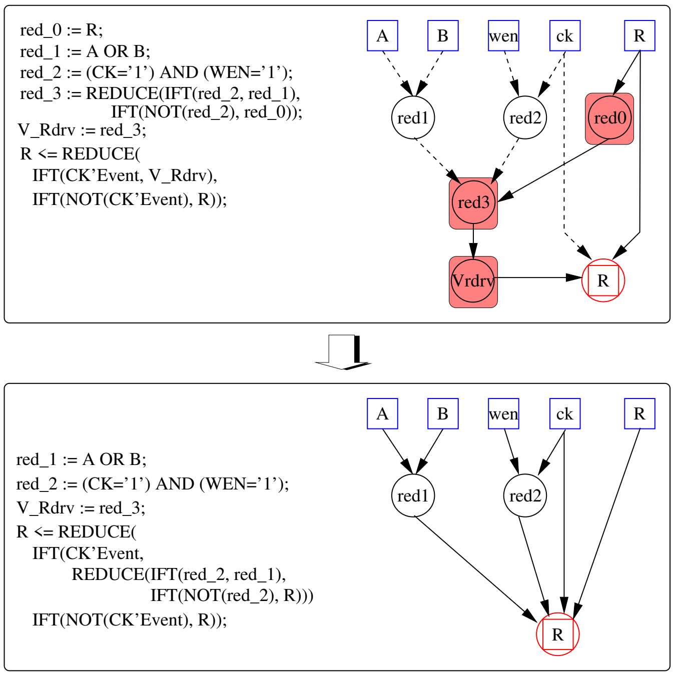

Figure 6.17: La substitution des variables intermédiaires.

La figure 6.17 illustre cette substitution. Les variables v_{Rdrv}, red_3 et red_0 interviennent dans le calcul de r et leur support étendu contient le signal r. Elles doivent par conséquent être éliminées dans l'expression affectée à r.

La condition de mémorisation

La décomposition de Shannon permet d'écrire la fonction f de la façon suivante :

- $f(r, x_2, \ldots, x_n) = (r \wedge f_r^+(x_2, \ldots, x_n)) \vee (\overline{r} \wedge f_r^-(x_2, \ldots, x_n))$,
  avec $f_r^+(x_2, \ldots, x_n) = f(1, x_2, \ldots, x_n)$ et $f_r^-(x_2, \ldots, x_n) = f(0, x_2, \ldots, x_n)$.

La condition c^{mem} est la condition qui doit être respectée pour que le registre r conserve sa valeur. Ainsi lorsque $c^{mem} = 1$, quelque soit la valeur de r on a $f = r$.

La condition c^{mem} peut donc s'écrire : $c^{mem} = f_r^+(x_2, \ldots, x_n) \wedge \overline{f_r^-(x_2, \ldots, x_n)}$.

Cette condition est calculée en construisant tout d'abord le graphe de décision binaire réduit et ordonné de la fonction f, puis en calculant sur ce graphe les deux fonctions $f_r^+(x_2, \ldots, x_n)$ et $f_r^-(x_2, \ldots, x_n)$.

```
red_0 := A OR B;
red_2 := (CK='1') AND (WEN='1');
R <= REDUCE(IFT(CK'Event,REDUCE(IFT(    red_2 ,red_0),
                                IFT(NOT(red_2),R     ))),
            IFT(NOT(CK'Event), R));
```

$$f_r^- = red_2 \wedge red_0$$
$$f_r^+ = \overline{red_2} \vee red_0$$
$$c^{mem} = \overline{red_2 \wedge ck'Event}$$

Figure 6.18: L'identification de la condition de mémorisation.

La figure 6.18 illustre le calcul de la condition de mémorisation du registre r.

On étudie ensuite la valeur de la condition de mémorisation :

- si c^{mem} est égale à 1 alors c'est qu'on a $f = r$. Le symbole r est une constante dont la valeur est celle qui lui a été donnée lors de l'initialisation ;
- si c^{mem} est égale à 0, alors le symbole r n'est pas un registre puisqu'il ne peut pas conserver sa valeur. Le symbole r représente la sortie de la fonction combinatoire f. Il peut néanmoins dépendre de $\overline{r}$ et on doit étudier cette dépendance de façon à détecter la présence éventuelle d'un phénomène oscillatoire ;
- dans tous les autres cas, la condition de mémorisation c^{mem} doit être étudiée de façon précise afin de déterminer toutes les propriétés du registre r.

On voit que l'identification des propriétés du registre repose sur l'analyse de la fonction c^{mem}. Pour permettre cette analyse les variables $red_k \in V^{red}$ qui interviennent dans c^{mem} doivent être substituées dans f par l'expression f_{red_k} qui leur est affectée. On substitue également dans f toutes les variables $red_k \in V^{red}$ dont le support étendu $SuppExt(f_{red_k})$ fait intervenir des signaux $s \in SuppExt(c^{mem})$. La figure 6.19 présente sur notre exemple le résultat de cette substitution.

```
red_0 := A OR B;
R <= REDUCE(IFT(  (CK='1') AND CK'Event AND (WEN='1') ,red_0),
            IFT(NOT((CK='1') AND CK'Event AND (WEN='1')),R));
```

$$f_r^- = ck \wedge ck'Event \wedge wen \wedge red_0$$
$$f_r^+ = (\overline{ck \wedge ck'Event \wedge wen}) \vee red_0$$
$$c^{mem} = \overline{ck \wedge ck'Event \wedge wen}$$

Figure 6.19: La substitution des variables dans la condition de mémorisation.

La présence d'un phénomène oscillatoire

On définit c^{oscil} la condition pour que le symbole r prenne la valeur $\overline{r}$, ainsi lorsque $c^{oscil} = 1$, quelque soit la valeur de r on a $f = \overline{r}$. Cette condition peut donc s'écrire : $c^{oscil} = \overline{f_r^+(x_2, \ldots, x_n)} \wedge f_r^-(x_2, \ldots, x_n)$.

Si des variables $red_k \in V^{red}$ interviennent dans c^{oscil} alors elles doivent être substituées dans f par l'expression f_{red_k} qui leur est affectée. On substitue également dans f toutes les variables $red_k \in V^{red}$ dont le support étendu $SuppExt(f_{red_k})$ fait intervenir des signaux $s \in SuppExt(c^{oscil})$.

On étudie ensuite la valeur de la condition d'oscillation :

- si c^{oscil} est différente de 0 alors il existe un phénomène d'oscillation qui n'est pas souhaitable en synthèse. On distingue deux possibilités :
  - si cette condition nécessite la présence d'un front d'horloge pour être vérifiée alors la synthèse peut continuer l'identification du registre r ;
  - dans tous les autres cas le processus est jugé non synthétisable.
- si c^{oscil} est égale à 0 alors on peut continuer l'analyse du symbole r.

6.2.2 L'identification des propriétés d'un registre

Afin d'identifier formellement les propriétés d'un registre r on utilise l'expansion d'une fonction booléenne sur une base orthogonale. Soit un ensemble ϕ_i de n fonctions booléennes telles que $\sum_{i=1}^{n} \phi_i = 1$ et $\phi_i \wedge \phi_j = 0, \forall\, i \neq j$. L'expansion orthogonale de la fonction f est : $f = \sum_{i=1}^{n} (f \downarrow \phi_i) \wedge \phi_i$ où le terme $f \downarrow \phi_i$ est appelé le cofacteur généralisé de f sachant ϕ_i.

Contrairement à la décomposition de Shannon cette décomposition n'est pas unique. En effet toute fonction booléenne f_{ϕ_i} telle que $f \wedge f_{\phi_i} \subseteq f_{\phi_i} \subseteq f \vee \overline{f_{\phi_i}}$ est un cofacteur généralisé de f sachant ϕ_i. Puisque le terme $f \downarrow \phi_i$ n'est pas unique, il faut choisir parmi toutes les solutions possibles, une fonction qui a une expression simple du point de vue de la synthèse.

On prend la base $\phi_1 = c^{mem}$ et $\phi_2 = c^{write}$ où $c^{write} = \overline{c^{mem}}$, et on peut mettre f sous la forme $f = (c^{mem} \wedge (f \downarrow c^{mem})) \vee (c^{write} \wedge (f \downarrow c^{write}))$. On sait par définition de la condition c^{mem} qu'on peut remplacer $f \downarrow c^{mem}$ par r. On note $d^{data} = f \downarrow c^{write}$ et on obtient finalement l'expression suivante : $f = (c^{mem} \wedge r) \vee (c^{write} \wedge d^{data})$. La fonction $f^{mem} = c^{mem} \wedge r$ représente le comportement du registre lorsqu'il est dans une phase de mémorisation et la fonction $f^{write} = c^{write} \wedge d^{data}$ représente le comportement du registre lorsqu'il est en phase d'écriture d'une donnée d^{data}.

A ce stade les seules fonctions complètement définies sont f, c^{mem} et c^{write}. La fonction f^{write} est facilement calculée à partir de f et c^{write} puisqu'on a $f^{write} = f \wedge c^{write}$. Par contre la fonction d^{data} n'est pas définie de façon unique.

La modélisation d'un registre

Les propriétés que l'on cherche à analyser pour un registre sont plus précises qu'une donnée à mémoriser et une condition d'écriture. La figure 6.20 présente l'élément mémorisant synthétisable le plus général qu'on puisse décrire en VHDL. On distingue deux types de comportement pour ce registre r :

- un comportement asynchrone qui permet au registre de changer de valeur à tout moment, dès qu'un évènement intervient sur la condition de mémorisation ou sur la donnée à mémoriser. Ce changement de valeur n'est pas conditionné par la présence d'un front d'horloge. Il est défini par une condition c^{as} de remise à un asynchrone, une condition c^{ar} de remise à zéro asynchrone, une condition c^{aw} d'écriture asynchrone et une donnée d^{aw} ;

- un comportement synchrone qui permet au registre de changer de valeur uniquement sur un front d'horloge. Il est défini par un front c^{edge}, une condition c^{ss} de remise à un synchrone, une condition c^{sr} de remise à zéro synchrone, une condition c^{sw} d'autorisation en écriture synchrone et une donnée d^{sw}.

```
P1 : PROCESS( AS, AR, AW, DAW, CK )
BEGIN
  IF    (AS='1') THEN R <= '1';       -- Remise à 1 asynchrone
  ELSIF (AR='1') THEN R <= '0';       -- Remise à 0 asynchrone
  ELSIF (AW='1') THEN R <= DAW;       -- Écriture asynchrone

  ELSIF (CK='1' AND CK'EVENT)         -- Front d'horloge

  THEN IF    (SS='1') THEN R <= '1'; -- Remise à 1 synchrone
       ELSIF (SR='1') THEN R <= '0'; -- Remise à 0 synchrone
       ELSIF (SW='1') THEN R <= DSW; -- Écriture synchrone
       END IF;
  END IF;
END PROCESS P1;
```

Figure 6.20: Le modèle complet d'un élément mémorisant.

Le comportement du registre de la figure 6.20 est très général. Tous les registres synthétisables sont des versions plus ou moins allégées de ce comportement générique. La forme syntaxique utilisée sur la figure 6.20 est totalement arbitraire et elle ne constitue qu'une des innombrables variantes équivalentes permises par VHDL.

Toutes les propriétés d'un registre sont identifiées en analysant f^{write} et c^{write}. La figure 6.21 présente l'expression affectée au registre r de la figure 6.20 et une expression pour les deux fonctions c^{write} et f^{write}.

```
R <= REDUCE(IFT(AS='1','1'),
     REDUCE(IFT(AS='0' AND AR='1','0'),
     REDUCE(IFT(AS='0' AND AR='0' AND AW='1', DAW),
            IFT(AS='0' AND AR='0' AND AW='0'
                      AND CK='1' AND CK'Event,
      REDUCE(IFT(SS='1','1')
      REDUCE(IFT(SS='0' AND SR='1','1'),
             IFT(SS='0' AND SR='0' AND SW='1', DSW)))))));
```

$$\Downarrow$$

$$c^{write} = as \vee ar \vee aw \vee \Big(ck \wedge ck'Event \wedge (ss \vee sr \vee sw)\Big)$$

$$f^{write} = as \vee \left(\overline{ar} \wedge aw \wedge daw\right) \vee \left(\overline{as} \wedge \overline{ar} \wedge \overline{aw} \wedge ck \wedge ck'Event \wedge \Big(ss \vee (\overline{sr} \wedge sw \wedge dsw)\Big)\right)$$

Figure 6.21: Les fonctions représentant l'écriture dans un registre.

Le comportement asynchrone du registre r est défini par une condition d'écriture asynchrone c^{async} et une fonction d'affectation f^{async}. Son comportement synchrone est défini par un front c^{edge}, une condition d'écriture synchrone c^{sync} et une fonction d'affectation f^{sync}.

Notre méthode d'identification des propriétés du registre r consiste à décomposer successivement f^{write} et c^{write} afin de trouver une expression pour chacune des fonctions c^{as}, c^{ar}, c^{aw}, d^{aw}, c^{edge}, c^{ss}, c^{sr}, c^{sw} et d^{sw}. On identifie tout d'abord les propriétés asynchrones, puis le type de front d'horloge et enfin les propriétés synchrone du registre r.

Comme nous l'avons vu précédemment, la fonction $d^{data} = f \downarrow c^{write}$ qui intervient dans f^{write} n'est pas unique. Par conséquent il existe plusieurs solutions pour les différentes fonctions qu'on veut calculer. Notre méthode cherche donc à trouver les solutions qui font intervenir le minimum de symboles.

Les propriétés asynchrones

On s'intéresse tout d'abord au comportement asynchrone du registre r. On calcule la fonction f^{async} et c^{async} en procédant de la façon suivante :

- si il existe plusieurs signaux s tel que $s'Event$ apparaisse dans la condition c^{write} alors ce registre ne peut pas être implanté en matériel et le processus n'est pas synthétisable ;
- si il existe un signal s tel que $s'Event$ apparaisse dans la condition c^{write} alors puisqu'on sait que le comportement asynchrone du registre r n'est soumis à aucun front d'horloge on a : $c^{async} = c^{write}(s'Event \leftarrow 0)$ et $f^{async} = f^{write}(s'Event \leftarrow 0)$;

- si il la condition c^{write} ne contient pas de front $s'Event$ alors le registre r est strictement asynchrone et on a : $c^{async} = c^{write}$ et $f^{async} = f^{write}$.

La figure 6.22 illustre sur l'exemple de la figure 6.21 le calcul des fonctions c^{async} et f^{async} à partir des deux fonctions c^{write} et f^{write} précédemment obtenues.

$$c^{write} = as \lor ar \lor aw \lor \Big(ck \land ck'Event \land (ss \lor sr \lor sw)\Big)$$
$$f^{write} = as \lor \left(\overline{ar} \land aw \land daw\right) \lor \left(\overline{as} \land \overline{ar} \land \overline{aw} \land ck \land ck'Event \land \Big(ss \lor (\overline{sr} \land sw \land dsw)\Big)\right)$$
$$\Downarrow$$
$$c^{async} = c^{write}(ck'Event \leftarrow 0) = as \lor ar \lor aw$$
$$f^{async} = f^{write}(ck'Event \leftarrow 0) = as \lor \Big(\overline{ar} \land aw \land daw\Big)$$

Figure 6.22: Les fonctions représentant l'écriture asynchrone dans un registre.

On identifie séquentiellement toutes les propriétés asynchrones du registre à partir des deux fonctions c^{async} et f^{async}. On utilise pour cela les opérateurs de quantification universelle et existentielle.

La quantification universelle d'une fonction f pour une variable x est définie par $(\forall x, f) = f_x^+ \land f_x^-$. La quantification universelle sur un ensemble $X = \{x_1, \ldots, x_n\}$ de variables est définie comme l'application successive de la quantification universelle sur chacune des n variables. On a donc $(\forall X, f) = \forall x_0, (\ldots(\forall x_n, f)\ldots)$.

La quantification existentielle d'une fonction f pour une variable x est définie par : $(\exists x, f) = \overline{\forall x, \overline{f}} = f_x^+ \lor f_x^-$, et pour un ensemble de variables on a $(\exists X, f) = \overline{\forall X, \overline{f}}$.

Les propriétés asynchrones du registre r sont calculées de la façon suivantes :

1. la condition asynchrone de remise à un $c^{as} \subseteq c^{async}$ est définie comme la condition qui doit être vérifiée pour que, quelque soit la valeur de la donnée à mémoriser on ait toujours $f^{async} = 1$. Les symboles qui interviennent dans l'expression de la donnée à mémoriser sont au minimum tout ceux qui apparaissent dans l'ensemble $X = Supp(f^{async}) - Supp(c^{async})$.

   Ainsi pour extraire la condition c^{as} de c^{async} on effectue une quantification universelle sur cet ensemble de symboles : $c^{as} = \forall X, c^{async} \land f^{async}$;

2. la condition asynchrone de remise à zéro $c^{ar} \subseteq c^{async}$ est définie comme la condition qui doit être vérifiée pour que quelque soit la valeur de la donnée à mémoriser on ait toujours $f^{async} = 0$.

   Ainsi pour extraire la condition c^{ar} de c^{async} on effectue une quantification universelle sur l'ensemble X : $c^{ar} = \forall X, c^{async} \land \overline{f^{async}}$;

3. la condition asynchrone d'autorisation en écriture $c^{aw} \subseteq c^{async}$ est définie comme la condition qui doit être vérifiée pour que la donnée d^{aw} puisse être écrite dans le registre r. Cette condition est donc simplement : $c^{aw} = c^{async} \land (\overline{c^{as} \lor c^{ar}})$.

La donnée d^{aw} associée à la condition asynchrone d'autorisation d'écriture c^{aw} doit être extraite de la fonction f^{async}. Comme nous l'avons vu précédemment il existe plusieurs solutions pour la fonction d^{aw}.

On cherche à calculer une fonction δ^{aw} dont le support est disjoint de celui de c^{async}, pour minimiser la complexité de l'expression d^{aw}. Ainsi pour extraire la donnée de f^{async} on effectue une quantification existentielle sur l'ensemble des symboles $Y = Supp(c^{async})$: $\delta^{aw} = \exists Y, c^{aw} \wedge f^{async}$.

Cependant la fonction δ^{aw} n'est pas toujours une solution acceptable et il faut vérifier l'identité : $f^{async} = c^{as} \vee (\overline{c^{ar}} \wedge c^{aw} \wedge \delta^{aw})$.

Si cette identité est respectée alors on a $d^{aw} = \delta^{aw}$. Dans le cas contraire, cela signifie que les supports de d^{aw} et de c^{async} ne sont pas disjoints et il faut utiliser une méthode plus générale. On cherche à simplifier la fonction f^{async} sachant que la condition c^{aw} est vraie, et on a ainsi $d^{aw} = f^{async} \downarrow c^{aw}$ où $\downarrow$ est un opérateur de restriction qui cherche à minimiser le nombre de symboles intervenant dans l'expression de d^{aw}. Cet opérateur est appliqué sur le graphe de décision binaire de f^{async} et c^{aw}, et il est décrit dans l'annexe B.

La figure 6.23 illustre sur l'exemple de la figure 6.21 le calcul des expressions de c^{as}, c^{ar}, c^{aw} et d^{aw} à partir des deux fonctions c^{async} et f^{async} précédemment obtenues.

$$\begin{array}{l} c^{async} = c^{write}(ck'Event \leftarrow 0) = as \vee ar \vee aw \\ f^{async} = f^{write}(ck'Event \leftarrow 0) = as \vee \left(\overline{ar} \wedge aw \wedge daw\right) \end{array} \quad \Longrightarrow \quad \begin{array}{l} c^{as} = as \\ c^{ar} = \overline{as} \wedge ar \\ c^{aw} = \overline{as} \wedge \overline{ar} \wedge aw \\ d^{aw} = daw \end{array}$$

Figure 6.23: L'identification des propriétés asynchrones d'un registre.

Les propriétés synchrones

On s'intéresse ensuite au comportement synchrone du registre r et on note s le signal tel que $s'Event$ apparaisse dans la condition $c'^{write} = c^{write} \wedge \overline{c^{async}}$. On étudie l'influence de s et $s'Event$ sur la condition c'^{write} de la façon suivante :

- $c_+^{rising} = c'^{write}(s \leftarrow 1, s'Event \leftarrow 1)$ et $c_-^{rising} = c'^{write}(s \leftarrow 1, s'Event \leftarrow 0)$;
- $c_+^{falling} = c'^{write}(s \leftarrow 0, s'Event \leftarrow 1)$ et $c_-^{falling} = c'^{write}(s \leftarrow 0, s'Event \leftarrow 0)$.

On note $f'^{write} = f^{write} \wedge \overline{c^{async}}$, $c'^{sync} = \overline{c^{async}} \wedge c^{sync}$ et $f'^{sync} = \overline{c^{async}} \wedge f^{sync}$. On identifie la condition c^{edge} et on calcule c'^{sync} et f'^{sync} en procédant de la façon suivante :

- si $c_+^{rising} \neq 0$, $c_-^{rising} = 0$, $c_+^{falling} = 0$ et $c_-^{falling} = 0$ alors il s'agit d'un front montant : $c^{edge} = s \wedge s'Event$, $c'^{sync} = c_+^{rising}$, $f'^{sync} = f'^{write}(s \leftarrow 1, s'Event \leftarrow 1)$;

- si $c_+^{falling} \neq 0$, $c_-^{falling} = 0$, $c_-^{rising} = 0$ et $c_+^{rising} \neq 0$ alors il s'agit d'un front descendant : $c^{edge} = \overline{s} \wedge s'Event$, $c'^{sync} = c_+^{falling}$, $f'^{sync} = f'^{write}(s \leftarrow 0, s'Event \leftarrow 1)$;
- dans tous les autres cas le registre r n'est pas synthétisable.

La figure 6.24 présente sur l'exemple de la figure 6.23 le résultat de l'identification du type de front d'horloge.

$$
\begin{aligned}
c^{write} &= as \vee ar \vee aw \vee \Big(ck \wedge ck'Event \wedge (ss \vee sr \vee sw)\Big) \\
f^{write} &= as \vee \left(\overline{ar} \wedge aw \wedge daw\right) \vee \left(\overline{as} \wedge \overline{ar} \wedge \overline{aw} \wedge ck \wedge ck'Event \wedge \Big(ss \vee (\overline{sr} \wedge sw \wedge dsw)\Big)\right) \\
c^{async} &= c^{write}(ck'Event \leftarrow 0) = as \vee ar \vee aw
\end{aligned}
$$

$$\Downarrow$$

$$
\begin{aligned}
c'^{write} &= \overline{as} \wedge \overline{ar} \wedge \overline{aw} \wedge ck \wedge ck'Event \wedge (ss \vee sr \vee sw) \\
f'^{write} &= \overline{as} \wedge \overline{ar} \wedge \overline{aw} \wedge ck \wedge ck'Event \wedge \Big(ss \vee (\overline{sr} \wedge sw \wedge dsw)\Big)
\end{aligned}
$$

$$\Downarrow$$

$$
\begin{aligned}
c_+^{rising} &= c'^{write}(ck \leftarrow 1, ck'Event \leftarrow 1) = \overline{as} \wedge \overline{ar} \wedge \overline{aw} \wedge (ss \vee sr \vee sw) \\
c_-^{rising} &= c'^{write}(ck \leftarrow 1, ck'Event \leftarrow 0) = 0 \\
c_+^{falling} &= c'^{write}(ck \leftarrow 0, ck'Event \leftarrow 1) = 0 \\
c_-^{falling} &= c'^{write}(ck \leftarrow 0, ck'Event \leftarrow 0) = 0
\end{aligned}
$$

$$\Downarrow$$

$$c^{edge} = ck \wedge ck'Event$$

Figure 6.24: L'identification du type de front d'horloge.

Il s'agit ensuite d'extraire c^{sync} de c'^{sync} et f^{sync} de f'^{sync}.

On cherche à calculer deux fonctions δ^{sync} et γ^{sync} dont les supports sont disjoints de c^{async}, pour minimiser les expressions de c^{sync} et f^{sync}. On effectue une quantification existentielle sur l'ensemble des symboles $Y = Supp(c^{async})$: $\delta^{sync} = \exists Y, c'^{sync}$ et $\gamma^{sync} = \exists Y, f'^{sync}$.

Cependant les fonctions δ^{sync} et γ^{sync} ne sont pas toujours des solutions acceptables et il faut vérifier les deux identités : $c'^{sync} = \overline{c^{async}} \wedge \delta^{sync}$ et $f'^{sync} = \overline{c^{async}} \wedge \gamma^{sync}$.

Si ces deux identités sont respectées alors on a $c^{sync} = \delta^{sync}$ et $f^{sync} = \gamma^{sync}$. Dans le cas contraire, cela signifie que les supports de c^{sync} et de c^{async} ne sont pas disjoints et il faut utiliser une méthode plus générale. On cherche à simplifier les deux fonction c'^{sync} et f'^{sync} sachant que la condition $\overline{c^{async}}$ est vraie. On a ainsi $c^{sync} = c'^{sync} \downarrow \overline{c^{async}}$ et $f^{sync} = f'^{sync} \downarrow \overline{c^{async}}$, où $\downarrow$ est un opérateur de restriction qui cherche à minimiser le nombre de symboles intervenant dans les expressions de c^{sync} et f^{sync}.

La figure 6.25 illustre sur l'exemple de la figure 6.21 le calcul des fonctions c^{sync} et f^{sync} à partir des deux fonctions c'^{write} et f'^{write} précédemment obtenues.

Si le registre r est synthétisable alors on a $f^{sync} = c^{ss} \vee (\overline{c^{sr}} \wedge c^{sw} \wedge d^{sw})$. On identifie à partir des deux fonctions c^{sync} et f^{sync} toutes les propriétés synchrones en procédant exactement de la même façon que lors de l'extraction des propriétés asynchrones de r.

$$
\begin{aligned}
c'^{write} &= \overline{as} \wedge \overline{ar} \wedge \overline{aw} \wedge ck \wedge ck'Event \wedge (ss \vee sr \vee sw)\\
f'^{write} &= \overline{as} \wedge \overline{ar} \wedge \overline{aw} \wedge ck \wedge ck'Event \wedge \Big(ss \vee (\overline{sr} \wedge sw \wedge dsw)\Big)
\end{aligned}
$$

$$\Downarrow$$

$$
\begin{aligned}
c^{sync} &= ss \vee sr \vee sw\\
f^{sync} &= ss \vee (\overline{sr} \wedge sw \wedge dsw)
\end{aligned}
$$

Figure 6.25: Les fonctions représentant l'écriture synchrone dans un registre.

La figure 6.26 illustre sur l'exemple de la figure 6.21 le calcul des expressions de c^{ss}, c^{sr}, c^{sw} et d^{sw} à partir des deux fonctions c^{sync} et f^{sync}.

$$
\begin{aligned}
c^{sync} &= ss \vee sr \vee sw\\
f^{sync} &= ss \vee (\overline{sr} \wedge sw \wedge dsw)
\end{aligned}
\quad \Rightarrow \quad
\begin{aligned}
c^{ss} &= ss\\
c^{sr} &= \overline{ss} \wedge sr\\
c^{sw} &= \overline{ss} \wedge \overline{sr} \wedge sw\\
d^{sw} &= dsw
\end{aligned}
$$

Figure 6.26: L'identification des propriétés synchrones d'un registre.

La vérification de la cohérence des propriétés du registre

Après avoir identifié les propriétés asynchrones et synchrones du registre r on vérifie que les identités suivantes sont bien respectées :

- $c^{async} = c^{as} \vee c^{ar} \vee c^{aw}$ et $c^{sync} = c^{ss} \vee c^{sr} \vee c^{sw}$;
- $f^{async} = c^{as} \vee (\overline{c^{ar}} \wedge c^{aw} \wedge d^{aw})$ et $f^{sync} = c^{ss} \vee (\overline{c^{sr}} \wedge c^{sw} \wedge d^{sw})$;
- $f = f^{async} \vee (\overline{c^{async}} \wedge c^{edge} \wedge f^{sync}) \vee (\overline{(c^{async} \vee c^{edge} \vee c^{sync})} \wedge r)$.

Dans le cas où l'une de ces identités n'est pas respectée alors r n'est pas synthétisable. Dans le cas contraire, on analyse la cohérence de toutes les propriétés du registre r avec la liste de sensibilité du processus. Cette vérification est développée dans la section 6.4.

6.3 L'analyse formelle des barrières trois états

On procède de façon similaire pour identifier formellement les propriétés des barrières trois états qui ont été détectées lors de l'analyse préliminaire des affectations.

6.3.1 L'identification de la condition de mise en haute impédance

L'analyse d'une affectation (d_j, f_j) telle que l'élément $'Z' \in \ CsteExt^b(f_j)$ doit permettre de trouver la condition $c^{trs}_{d^b_j}$ qui doit être respectée pour que d^b_j prenne la valeur $'Z'$. Par la suite on notera d le symbole d^b_j et f l'expression f_j dont on a extrait seulement la partie qui concerne le bit b.

```
P1 : PROCESS( A,B,SEL )
BEGIN
 IF (SEL='1')
 THEN S <= A OR B;
 ELSE S <= 'Z';
 END IF;
END PROCESS P1;
```

```
P1 : PROCESS( A,B,SEL )
 VARIABLE V_Ddrv : BIT;
 VARIABLE red_0 : BIT;
 VARIABLE red_1 : BIT;
 VARIABLE red_2 : BIT;
 VARIABLE red_3 : BOOLEAN;
BEGIN
 red_0 := D;
 V_Ddrv := red_0;
 red_3 := (SEL='1');
 IF (red_3)
 THEN
   red_1 := A OR B;
   V_Ddrv := red_1;
 ELSE
   red_2 := 'Z';
   V_Ddrv := red_2;
 END IF;
 D <= V_Ddrv;
END PROCESS P1;
```

```
red_1 := A OR B;
red_2 := 'Z';
red_3 := (SEL='1');
red_4 := REDUCE(IFT(    red_3 , red_1),
                IFT(NOT(red_3), red_2));
V_Ddrv := red_4;
D <= V_Ddrv;
```

Figure 6.27: L'identification d'une barrière trois états.

La figure 6.27 présente la description d'un processus *P1*, puis les modifications initiales qui lui ont été appliquées avant réduction et enfin la liste des affectations du graphe de contrôle réduit. Le signal d dépend de $'Z'$ et l'analyse doit identifier toutes les propriétés de cette barrière trois états.

La substitution des variables

Pour retrouver la condition c^{trs} il est indispensable que toutes les occurences de l'élément $'Z'$ apparaissent directement dans l'expression de f. Soit V^{subst}_f l'ensemble des variables $red_k \in V^{red} \cap SuppExt(f)$ et telles que $'Z' \in \ CsteExt(f_{red_k})$ où f_{red_k} est l'expression qui est affectée à la variable red_k. On substitue de façon itérative dans f toutes les occurences des variables red_k de l'ensemble V^{subst}_f par les expressions f_{red_k}.

La figure 6.28 illustre cette substitution. Les variables v_{Ddrv}, red_4 et red_2 interviennent dans le calcul de d et elles font intervenir directement ou indirectement l'élément $'Z'$. Elles doivent par conséquent être substituées dans l'expression affectée à d.

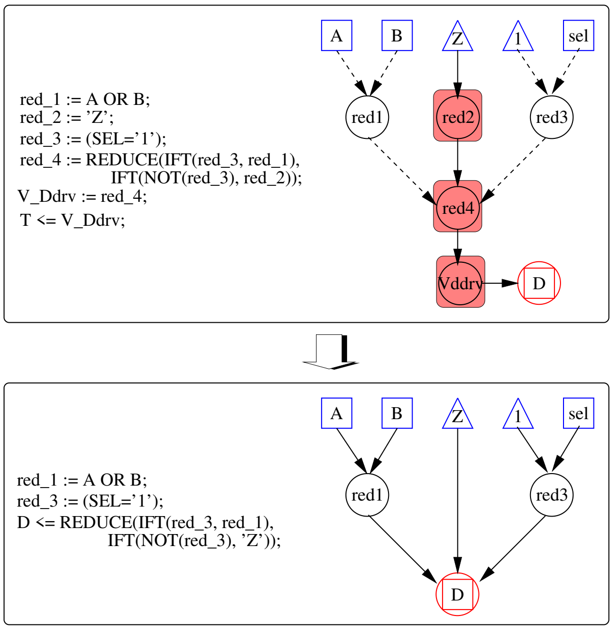

Figure 6.28: La substitution des variables intermédiaires.

L'utilisation incorrecte de la constante $'Z'$

L'utilisation de la constante $'Z'$ est soumise à une règle très stricte mais légitime en synthèse. Dans la description initiale, elle ne doit apparaître que dans le membre de droite d'une affectation de variable ou de signal. De plus elle ne doit pas intervenir directement ou indirectement dans une expression logique ou arithmétique.

La figure 6.29 présente deux exemples d'utilisation incorrecte de la constante $'Z'$ du point de vue de la synthèse. Dans le cas du processus *P1* l'affectation du signal *D* fait intervenir la constante $'Z'$ dans une expression logique. Dans le cas du processus *P2* la constante $'Z'$ est utilisée de façon correcte dans l'affectation de la variable *V*, mais cette variable intervient ensuite dans une expression logique et ce processus n'est donc pas synthétisable.

```
P1 : PROCESS( SEL, I0 )
BEGIN
  D <= (SEL AND I0) OR
       (NOT(SEL) AND 'Z');
END PROCESS P1;
```

```
P2 : PROCESS( SEL, I0 )
  VARIABLE V : STD_LOGIC;
BEGIN
  V := 'Z';
  D <= (SEL AND I0) OR
       (NOT(SEL) AND V);
END PROCESS P1;
```

Figure 6.29: L'utilisation incorrecte de la constante $'Z'$.

Comme nous l'avons vu dans le chapitre 5.2 la fonction IFT est utilisée lors de la réduction pour regrouper une condition d'affectation et une valeur à affecter. Par conséquent on vérifie dans l'expression f affectée à d que la constante $'Z'$ intervient uniquement et directement dans le membre de droite d'une fonction IFT.

La condition de mise en haute impédance

Afin d'identifier la condition de mise à l'état haute impédance, on utilise une technique similaire à celle employée pour identifier la condition de mémorisation d'un registre. Comme nous l'avons vu dans la section 6.2, dans le cas d'un registre r il s'agit de trouver la condition qui doit être vérifiée pour que $f = r$. Dans le cas d'une barrière trois états il s'agit de trouver la condition qui doit être vérifiée pour que $f =' Z'$. Cependant la constante $'Z'$ ne peut pas être directement représentée à l'aide d'un graphe de décision binaire. Néanmoins puisque la constante $'Z'$ est utilisée correctement, c'est à dire qu'elle n'intervient dans aucune expression logique ou arithmétique, elle peut être symbolisée par une variable v_z. Ainsi le problème devient identique à celui posé lors de l'identification des propriétés d'un registre.

On note f' la fonction f dans laquelle toutes les occurences de l'élément $'Z'$ ont été remplacées par v_z. La décomposition de Shannon permet d'écrire la fonction f' de la façon suivante :

- $f'(v_z, x_2, \ldots, x_n) = (v_z \wedge {f'}^{+}_{z}(x_2, \ldots, x_n)) \vee (\overline{v_z} \wedge {f'}^{-}_{z}(x_2, \ldots, x_n))$,
  avec ${f'}^{+}_{z}(x_2, \ldots, x_n) = f'(1, x_2, \ldots, x_n)$ et ${f'}^{-}_{z}(x_2, \ldots, x_n) = f'(0, x_2, \ldots, x_n)$.

La condition c^{trs} est la condition qui doit être respectée pour que la barrière trois états d prenne la valeur $'Z'$. La condition c^{trs} peut s'écrire :
$c^{trs} = {f'}^{+}_{z}(x_2, \ldots, x_n) \wedge \overline{{f'}^{-}_{z}(x_2, \ldots, x_n)}$.

Cette condition est calculée en construisant le graphe de décision binaire réduit et ordonné de la fonction f', puis en calculant les deux fonctions ${f'}^{+}_{z}$ et ${f'}^{-}_{z}$.

```
red_1 := A OR B;
red_3 := (SEL='1');
D <= REDUCE(IFT(    red_3 , red_1),
            IFT(NOT(red_3), 'Z'   )))
```

$${f'}^{-}_{z} = red_3 \wedge\ red_1$$
$${f'}^{+}_{z} = \overline{red_3} \vee\ red_1$$
$$c^{trs} = \overline{red_3}$$

Figure 6.30: L'identification de la condition c^{trs}.

La figure 6.30 illustre le calcul de la condition c^{trs} de la barrière trois états d. On remarque que c^{trs} fait apparaître une variable intermédiaire red_3.

Les variables $red_k \in V^{red}$ qui interviennent dans c^{trs} doivent être substituées dans f' par l'expression f_{red_k} qui leur est affectée afin de pouvoir extraire précisément les propriétés de la barrière trois états d. La figure 6.31 présente sur notre exemple le résultat de cette substitution.

On étudie ensuite la valeur de la condition c^{trs} :

`red_1 := A OR B;`
`D <= REDUCE(IFT(SEL='1', red_1),`
`IFT(NOT(SEL='1'), 'Z'));`
	$f'^{-}_{z} = sel \wedge red_1$
	$f'^{+}_{z} = (\overline{sel} \vee red_1$
	$c^{trs} = \overline{sel}$

Figure 6.31: La substitution des variables qui interviennent dans c^{trs}.

- si c^{trs} est égale à 0, alors le symbole d n'est pas une barrière trois états puisqu'il ne peut pas prendre la valeur $'Z'$. Le symbole d représente la sortie de la fonction combinatoire f.
- dans tous les autres cas, la condition c^{trs} représente la condition de mise en haute impédance de la barrière trois états d. La donnée qui doit être émise lorsqu'on a $c^{trs} = 0$ doit être calculée.

6.3.2 L'identification des propriétés de la barrière trois états

Une barrière trois états d est défini par une condition d'émission c^{emi} et une fonction combinatoire d^{emi} représentant la donnée à émettre. L'analyse de l'affectation (d, f) doit permettre de séparer la donnée à émettre et la condition d'émission $c^{emi} = \overline{c^{trs}}$ de la barrière trois états d.

Si d est une barrière trois états alors la fonction f' affectée à d peut s'écrire de la façon suivante : $f' = f^{emi} \vee f^{trs}$, où la fonction f^{emi} représente le comportement de la barrière lorsqu'elle est dans une phase d'émission et la fonction f^{trs} représente le comportement de la barrière lorsqu'elle est dans un état de haute impédance.

La phase d'émission d'une barrière trois états est conditionnée par c^{emi} alors que la phase de haute impédance est conditionnée par c^{trs}. On a donc $f^{emi} = f' \wedge c^{emi}$ et $f^{trs} = f' \wedge c^{trs}$, ou plus précisément $f^{emi} = c^{emi} \wedge d^{emi}$ et $f^{trs} = c^{trs} \wedge v_z$.

Toutes les propriétés de d sont identifiées en analysant f^{emi} et c^{emi}. On peut remarquer que si $c^{emi} = 0$ alors aucune donnée ne sera jamais émise sur le bus d et on peut donc attribuer la valeur arbitraire 0 à la donnée d^{emi}.

La donnée d^{emi} associée à la condition c^{emi} doit être extraite de la fonction f^{emi}. On utilise la technique qui a été présentée dans les paragraphes 6.2.2 lors de l'identification des propriétés asynchrones et synchrones d'un registre.

`P1 : PROCESS( A,B,SEL )`
`BEGIN`
` D <= 'Z';`
` IF (SEL='1')`
` THEN D <= A OR B;`
` END IF;`
`END PROCESS P1;`
	`red_1 := A OR B;`
	`D <= REDUCE(IFT(SEL='1',red_1),`
	`IFT(NOT(SEL='1'),'Z'));`
		$c^{emi} = sel$
		$d^{emi} = red_1$

Figure 6.32: L'identification formelle des propriétés d'une barrière trois états.

La figure 6.32 illustre le résultat de l'application de notre méthode d'identification formelle sur un exemple.

6.4 La cohérence avec la liste de sensibilité

La dernière étape vérifie la cohérence des résultats précédemment obtenus avec la liste de sensibilité suivant les trois types d'implantation matérielle.

6.4.1 Les symboles de type combinatoire

Pour tous les symboles $d \in V^{used} \cup S^{used}$ qui sont de type combinatoire on vérifie :

- si $d \in V^{ini}$ alors le processus n'est pas synthétisable, puisque d doit nécessairement être un registre ;
- si $d \in S^{used}$ alors :
  - si il existe une variable $v \in V_{reg}^{ini}$ dans $SuppExt^b(f_d)$ telle que v soit affectée après le signal d alors le processus n'est pas synthétisable. En effet le symbole d ne respecte pas les contraintes liées à la lecture d'une variable de V_{reg}^{ini} et qui ont été définies dans les paragraphes 6.1.6 ;
  - si $SuppExt^b(f_d) \cap S \subseteq w(t_{wait})$ alors d peut être implanté à l'aide de portes combinatoires ;
  - dans tous les autres cas, le processus n'est pas synthétisable car certains signaux manquent dans la liste de sensibilité.

6.4.2 Les symboles de type élément mémorisant

Pour tous les symboles d qui sont de type registre on vérifie :

- si $c^{async} \neq 0$ alors pour que d soit synthétisable il faut que : $(SuppExt(f^{async}) \cap S) \subseteq w(t_{wait})$;
- si $c^{edge} \neq 0$ alors pour que d soit synthétisable il faut que : $SuppExt(c^{edge}) \subseteq w(t_{wait})$;
- si il existe une variable $v \in V_{reg}^{ini}$ dans $SuppExt^b(f_d)$ telle que v soit affectée après le signal d, et si $v \in SuppExt(c^{mem})$ ou si $c^{edge} = 0$ alors le processus n'est pas synthétisable puisqu'il ne respecte pas les contraintes liées à la lecture d'une variable de V_{reg}^{ini}.

6.4.3 Les symboles de type barrière trois états

Pour tous les symboles d qui sont de type barrière trois états on fait :

- si $d \in V^{ini}$ alors le processus n'est pas synthétisable, puisque d doit nécessairement être un registre ;
- si $d \in S^{used}$ alors :
  - si il existe une variable $v \in V_{reg}^{ini}$ dans $SuppExt^{b}(f_d)$ telle que v soit affectée après le signal d alors le processus n'est pas synthétisable puisqu'il ne respecte pas les contraintes liées à la lecture d'une variable de V_{reg}^{ini} ;
  - si $SuppExt^{b}(f_d) \cap S \subseteq w(t_{wait})$ alors d peut être implanté à l'aide d'une barrière trois états ;
  - dans tous les autres cas, le processus n'est pas synthétisable car certains signaux manquent dans la liste de sensibilité.

6.5 Conclusion

Au cours de ce chapitre nous avons présenté une méthode d'analyse de l'unique liste d'affectations du graphe de contrôle réduit d'un processus à un seul point de suspension permettant de résoudre complètement les problèmes qui ont été exposés dans les chapitres 2 et 5.

Il s'agit en particulier de savoir si un processus est synthétisable ou non, et d'identifier formellement, pour tous les symboles affectés par un processus, le type d'élément matériel à utiliser pour obtenir un résultat de synthèse en conformité avec la sémantique de simulation : porte combinatoire, registre ou barrière trois états.

Nous avons tout d'abord introduit la notion de graphe de dépendance des affectations. Nous avons ensuite montré comment une analyse préliminaire des dépendances entre les symboles affectés permettait d'identifier *a priori* le type de matériel nécessaire à leur implantation.

Nous avons présenté une technique d'analyse formelle permettant d'identifier les registres et les barrières trois états, ainsi que toutes leur caractéristiques matérielles.

Enfin nous avons montré comment vérifier la cohérence des résultats de l'analyse formelle avec la liste de sensibilité du processus.

CHAPITRE 7

Une méthode d'analyse du graphe de contrôle des processus à plusieurs points de suspension

Sommaire

Nous proposons dans ce chapitre une méthode d'analyse permettant de transformer un processus à plusieurs points de suspension en un automate d'états finis synchrone équivalent. Nous étudions tout d'abord les difficultés à résoudre puis nous détaillons une méthode d'analyse permettant d'atteindre notre objectif.

Ce travail est effectué dans un but d'exhaustivité afin de montrer que les processus à plusieurs points de suspension peuvent être traités avec des techniques similaires à celles qui ont été employées pour les processus à un seul point de suspension. Notre étude est assez académique car il est assez rare que les architectes utilisent des processus à plusieurs points de suspension pour décrire en VHDL le comportement d'un circuit.

7.1 Présentation des différents problèmes à résoudre

L'unique façon de synthétiser un processus à plusieurs points de suspension est de le représenter sous la forme d'une machine à états. L'analyse d'un tel processus doit permettre, lorsque c'est possible, de construire le graphe d'un automate d'états finis synchrone reproduisant son comportement. Tous les processus à plusieurs points de suspension qui ne se comportent pas comme un automate synchrone sont considérés comme non synthétisables par l'analyse. La figure 7.1 présente le cas d'un tel processus.

```
UN_SYNTH : PROCESS
BEGIN
  Q <= A OR B;
  WAIT ON A,B;
  Q <= A AND B;
  WAIT ON A,B;
END PROCESS UN_SYNTH;
```

Figure 7.1: Un processus non synthétisable.

Comme nous l'avons vu dans le chapitre 2, pour tous les processus à plusieurs points de suspension il existe une variable d'état implicite contenant leur prochain point de réveil. Chaque point de suspension du processus correspond donc à un état de l'automate et chaque signal affecté représente une sortie de l'automate. Lorsque le processus se réveille il se comporte exactement comme un processus à un seul point de suspension, en empruntant un chemin d'exécution particulier suivant les valeurs des signaux et de ses variables internes avant de s'endormir à nouveau sur un point de suspension. Le passage d'un point de suspension à un autre correspond au passage d'un état de l'automate à un autre. Il est soumis à un ensemble de conditions qui représentent une des fonctions de transition de l'automate. Lors de cette exécution le processus peut modifier la valeur de ses variables internes et de certains signaux. Ces modifications représentent la fonction de génération associée à l'une des transitions de l'automate.

On distingue deux catégories de processus à plusieurs points de suspension se comportant comme des automates d'états finis synchrones :

1. ceux qui affectent des symboles uniquement sur un front d'horloge.
2. ceux qui modifient la valeur des symboles dès qu'un évènement intervient sur un signal présent dans la partie droite des affectations ;

Les processus *P1* et *P2* de la figure 7.2 font respectivement partie de la première et de la seconde catégorie. On peut remarquer que l'écriture de la description d'un processus à plusieurs points de suspension de la seconde catégorie est extrêmement fastidieuse et complexe et qu'aucun outils de synthèse n'accepte ce type de processus.

Par la suite on étudiera donc seulement le cas des processus de la première catégorie, c'est à dire ceux qui se comportent comme des automates d'états finis synchrones avec des registres en sortie.

```
P1 : PROCESS
BEGIN
  WAIT UNTIL CK='1';
  Q <= I1 OR I2;
  IF (I1='1')
  THEN WAIT UNTIL CK='1';
       Q <= I1 AND I2;
  ELSE WAIT UNTIL CK='1';
       Q <= I1 XOR I2;
  END IF;
END PROCESS P1;
```

```
P2 : PROCESS
BEGIN
  L1 : LOOP
    Q <= I1 OR I2;
    WAIT ON CK,I1,I2;
    EXIT L1 WHEN (CK='1' AND CK'EVENT);
  END LOOP L1;
  IF (I0='1')
  THEN L2 : LOOP
         Q <= I1 AND I2;
         WAIT ON CK,I1,I2;
         EXIT L2 WHEN (CK='1' AND CK'EVENT);
       END LOOP L2;
  ELSE L3 : LOOP
         Q <= I1 XOR I2;
         WAIT ON CK,I1,I2;
         EXIT L3 WHEN (CK='1' AND CK'EVENT);
       END LOOP L3;
  END IF;
END PROCESS P2;
```

Figure 7.2: Les deux catégories de processus synthétisables.

Les difficultés majeures de l'analyse des processus à plusieurs points de suspension réside d'une part dans la construction du graphe d'un automate d'états finis synchrone équivalent, et d'autre part dans l'identification des propriétés du registre d'états et des registres de sorties.

La méthode permettant de construire un automate d'états finis synchrone équivalent doit être indépendante de la structure initiale du graphe de contrôle et donc de la description VHDL originale. Elle s'effectue en deux étapes. Le but de la première étape est de déterminer à partir du graphe de contrôle du processus, comment et sous quelles conditions les variables internes et les signaux peuvent être modifiés entre deux points de suspension. Cette première étape ne cherche pas à construire un automate, elle vise simplement à rassembler les affectations présentes sur l'ensemble des différents chemins d'exécution qui relient deux points de suspension. La seconde étape quant à elle, permet d'extraire un graphe d'états équivalent et d'identifier la condition de mémorisation du registre d'état et des registres de sorties. Elle doit assurer par construction que l'automate d'états finis synchrone résultant est en conformité avec le comportement permanent du processus initial. Si cette construction s'avère impossible, alors le processus n'est pas synthétisable. Cette seconde étape exploite directement le résultat de la première, et elle est donc indépendante de la structure du graphe de contrôle du processus.

7.1.1 L'exploration du graphe de contrôle d'un processus à plusieurs points de suspension

Étant donnés deux points de suspension w_i et w_j du processus, l'exploration de l'ensemble des chemins possibles du graphe de contrôle entre w_i et w_j doit permettre d'obtenir :

- la condition qui doit être vérifiée pour passer de w_i à w_j ;
- l'ensemble des affectations qui peuvent effectuées pour chaque symbole s_k et les conditions qui doivent être respectées pour appliquer à s_k une des ces modifications lors du passage de w_i à w_j.

Cette exploration doit être effectuée pour tous les couples $\{w_i, w_j\}$ pour lesquels il existe un chemin d'exécution permettant de passer de w_i à w_j.

Le processus *P1* de la figure 7.3 décrit le comportement d'un compteur synchrone de trois uns successifs. Il est composé de trois points de suspension correspondant aux trois états du compteur. La sortie Q est mise à $'1'$ uniquement lorsque l'entrée *I0* est restée à $'1'$ pendant au moins trois cycles d'horloge.

```
P1 : PROCESS
BEGIN
  L1 : LOOP
    Q <= '0';
    WAIT UNTIL CK='1';    -- W1
    EXIT L1 WHEN I0='0';
    Q <= '0';
    WAIT UNTIL CK='1';    -- W2
    EXIT L1 WHEN I0='0';
    Q <= '0';
    L2 : LOOP
      WAIT UNTIL CK='1';  -- W3
      EXIT L1 WHEN I0='0';
      Q <= '1';
    END LOOP L2;
  END LOOP L1;
END PROCESS P1;
```

Figure 7.3: Un compteur de trois uns successifs.

Sur le graphe de contrôle de *P1* l'analyse doit par exemple permettre d'extraire les informations suivantes concernant le passage de w_1 à w_1 :

- lorsque l'horloge ck passe à $'0'$ ou que l'entrée $i0$ est à $'0'$ et que l'horloge passe à $'1'$ alors on sort de w_1 pour retourner sur w_1.
- lors du passage de w_1 à w_1, la sortie q conserve sa valeur lorsque ck vaut $'0'$, et elle prend la valeur $'0'$ lorsque ck vaut $'1'$ et *I0* vaut $'0'$.

Les problèmes à résoudre lors de cette exploration sont les mêmes que dans le cas des processus à un seul point de suspension. Ainsi pour tous les chemins d'exécution permettant de passer d'un point de suspension à un autre, l'analyse ne doit conserver que la dernière affectation pour chacun des signaux. D'autre part seules les valeurs des variables qui sont effectivement lues et qui interviennent dans le calcul du pilote d'un signal doivent être prises en compte. Sur la figure 7.4 par exemple, l'affectation *ASG 3* de la variable v_3 doit être ignorée car cette variable n'intervient nulle part dans le calcul du pilote d'un signal.

```
P1 : PROCESS
  VARIABLE V1, V2, V3 : BIT;
BEGIN
  WAIT UNTIL CK='1';  -- W1
  V1 := I0;           -- ASG 1
  V2 := I1;           -- ASG 2
  V3 := I0 AND I1;    -- ASG 3
  WAIT UNTIL CK='1';  -- W2
  Q <= V1 OR V2;
END PROCESS P1;
```

Figure 7.4: L'affectation des variables.

Une contrainte supplémentaire doit être prise en compte lors de l'analyse. Soient trois points de suspension w_i, w_j, w_k tels qu'il existe un chemin d'exécution reliant w_i à w_j et un autre reliant w_i à w_k. S'il existe une affectation commune sur ces deux chemins d'exécution il faut pouvoir conserver cette information lors de l'analyse, pour donner aux étapes ultérieures de la synthèse les moyens de partager efficacement des ressources matérielles critiques.

Sur la figure 7.5 par exemple l'affectation *ASG 1* est exécutée à la fois lors du passage de w_1 à w_2 et lors du passage de w_1 à w_3. L'expression qui est affectée au signal q doit donc être partagée.

```
P1 : PROCESS
BEGIN
  WAIT UNTIL CK='1';       -- W1
  Q <= A + B;              -- ASG 1
  IF ( SEL = '1' )
  THEN WAIT UNTIL CK='1';  -- W2
       Q <= A - B;         -- ASG 2
  ELSE WAIT UNTIL CK='1';  -- W3
       Q <= B - A;         -- ASG 3
  END IF;
END PROCESS P1;
```

Figure 7.5: Le partage d'une affectation de signal.

La complexité de cette première étape d'analyse dépend du nombre de chemins d'exécution possibles entre les différents points de suspension du processus. Notre méthode cherche à réduire le nombre de chemins d'exécution à explorer entre deux points de suspension. Les techniques de réduction employées sont similaires aux techniques appliquées aux processus à un seul point de suspension. Ces techniques ont les mêmes objectifs, c'est à dire de transformer le graphe de contrôle d'un processus élaboré en un graphe de contrôle canonique. Ce graphe de contrôle réduit est tel qu'il existe au plus un chemin d'exécution entre deux points de suspension. De plus tous les chemins d'exécution du graphe réduit comportent au plus une affectation à un symbole donné. La figure 7.6 présente la structure d'un graphe de contrôle complètement réduit.

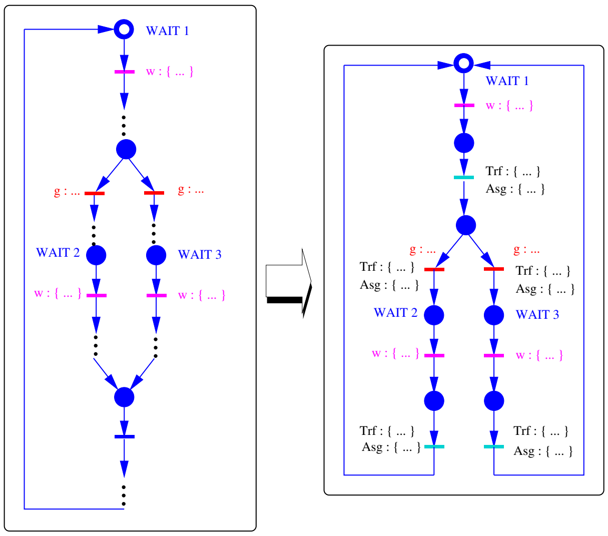

Figure 7.6: La réduction du graphe de contrôle d'un processus.

7.1.2 La construction d'un automate d'états finis

La seconde étape consiste à analyser le graphe de contrôle réduit afin de construire un automate d'états finis synchrone équivalent.

Du point de vue de la synthèse, on ne s'intéresse qu'à un seul type de processus à plusieurs points de suspension. Il s'agit des processus qui se comportent comme un automate d'états finis synchrone, avec des registres en sortie. C'est le cas du processus *P1* de la figure 7.7, puisque la sortie q est mise à jour uniquement sur front montant de l'horloge ck. Les variations de l'entrée $i0$ ne sont donc pas visibles au cours du cycle.

```
P1 : PROCESS
BEGIN
  L1 : LOOP
    Q <= '0';
    WAIT UNTIL CK='1';     -- W1
    EXIT L1 WHEN I0='0';
    Q <= '0';
    WAIT UNTIL CK='1';     -- W2
    EXIT L1 WHEN I0='0';
    Q <= '0';
    L2 : LOOP
      WAIT UNTIL CK='1';   -- W3
      EXIT L1 WHEN I0='0';
      Q <= '1';
    END LOOP L2;
  END LOOP L1;
END PROCESS P1;
```

Figure 7.7: Un exemple de processus synthétisable.

L'analyse doit déterminer si le graphe de contrôle réduit peut être implanté en utilisant l'architecture présentée sur la figure 7.8.

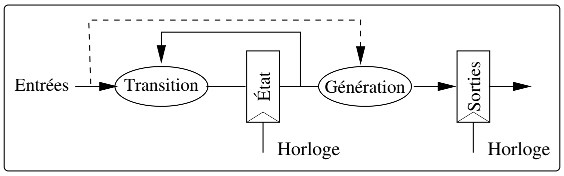

Figure 7.8: L'implantation d'un processus à plusieurs points de suspension.

Elle doit identifier toutes les propriétés des différents registres et construire le graphe d'un automate d'états finis synchrone équivalent. La figure 7.9 présente le résultat attendu après analyse d'un processus à plusieurs points de suspension.

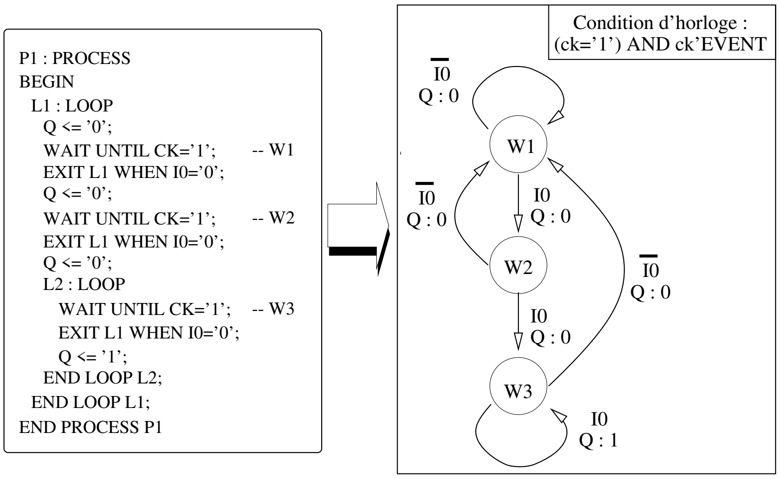

Figure 7.9: Le graphe de contrôle d'un automate d'états finis synchrone équivalent.

7.2 La réduction de notre modèle en vue de la synthèse

Comme nous l'avons vu précédemment l'objectif de la phase de réduction est d'obtenir pour chacun des processus de la description un graphe de contrôle minimal, c'est à dire tel qu'il existe au plus un chemin d'exécution entre deux points de suspension. La méthode de réduction d'un processus à plusieurs points de suspension est semblable à celle utilisée pour les processus à un seul point de suspension.

La phase de réduction se déroule en trois étapes. Les deux premières étapes sont strictement identiques pour les processus à un point de suspension et pour les processus à plusieurs points de suspension. Il s'agit tout d'abord de modifier les affectations présentes sur le graphe de contrôle afin de faciliter la réduction. Il s'agit ensuite de simplifier le graphe de contrôle du processus en appliquant des techniques de réduction locale. Ces deux étapes ont été détaillées dans le chapitre 5 dans la section 5.2. La dernière étape consiste à appliquer une technique de réduction globale permettant d'obtenir un graphe de contrôle minimal .

7.2.1 Le traitement initial des signaux et des variables

De la même façon que lors de la réduction d'un processus à un seul point de suspension on cherche ici à remplacer les signaux par des variables pour obtenir un modèle homogène, et à rendre explicite la "durée de vie" des variables internes du processus analysé.

La technique utilisée pour atteindre ces deux objectifs est celle qui a été décrite dans les sections 5.2.1 et 5.2.2. La figure 7.10 présente le résultat de cette étape préliminaire de la phase de réduction. Le processus *P1* est constitué de deux affectations du signal s notées *ASG 1* et *ASG 2*.

```
ENTITY EX IS
PORT ( A  : IN  BIT;
       B  : IN  BIT;
       Q  : OUT BIT );
END EX;

ARCHITECTURE BEH_1 OF EX IS
  SIGNAL S : BIT;
BEGIN
  P1 : PROCESS
  BEGIN
    S <= A OR B;  -- ASG 1
    WAIT UNTIL CK='1';
    S <= A AND B; -- ASG 2
    WAIT UNTIL CK='1';
  END PROCESS P1;
  Q <= S;
END BEH_1;
```

⇒

```
ENTITY EX IS
PORT ( A  : IN  BIT;
       B  : IN  BIT;
       Q  : OUT BIT );
END EX;

ARCHITECTURE BEH_2 OF EX IS
  SIGNAL S : BIT;
BEGIN
  P2 : PROCESS
    VARIABLE V_Sdrv : BIT;
  BEGIN
    V_Sdrv := A OR B;  -- ASG 3
    S      <= V_Sdrv;  -- ASG 4
    WAIT UNTIL CK='1';
    V_Sdrv := S;       -- ASG 5
    V_Sdrv := A AND B; -- ASG 6
    S      <= V_Sdrv;  -- ASG 7
    WAIT UNTIL CK='1';
    V_Sdrv := S;       -- ASG 8
  END PROCESS P2;
  Q <= S;
END BEH_2;
```

⇒

```
ENTITY EX IS
PORT ( A  : IN  BIT;
       B  : IN  BIT;
       Q  : OUT BIT );
END EX;

ARCHITECTURE BEH_3 OF EX IS
  SIGNAL S : BIT;
BEGIN
  P3 : PROCESS
    VARIABLE V_Sdrv, red_0,
             red_1, red_2,
             red_3 : BIT;
  BEGIN
    red_0  := A OR B;  -- ASG  9
    V_Sdrv := red_0;   -- ASG 10
    S      <= V_Sdrv;  -- ASG 11
    WAIT UNTIL CK='1';
    red_1  := S;       -- ASG 12
    V_Sdrv := red_1;   -- ASG 13
    red_2  := A AND B; -- ASG 14
    V_Sdrv := red_2;   -- ASG 15
    S      <= V_Sdrv;  -- ASG 16
    WAIT UNTIL CK='1';
    red_3  := S;       -- ASG 17
    V_Sdrv := red_3;   -- ASG 18
  END PROCESS P3;
  Q <= S;
END BEH_3;
```

Figure 7.10: La modification des affectations avant réduction.

Il est transformé en un processus *P2* équivalent pour lequel la modification de la valeur du pilote du signal s est explicite. Une variable locale v_{Sdrv} a été introduite pour représenter la valeur du pilote de s. Les deux affectations du signal s ont été transformées en deux affectations *ASG 3* et *ASG 6* de la variable v_{Sdrv}. Les affectations notées *ASG 4* et *ASG 7* donnent à s la valeur de la variable représentant son pilote. Lorsqu'un signal n'est pas affecté il conserve sa valeur et les affectations notées *ASG 5* et *ASG 8* assure ainsi l'initialisation de la variable v_{Sdrv} à la valeur du signal s.

Le processus *P2* est ensuite transformé en un processus *P3* équivalent dans lequel on a introduit des variables intermédiaires dans toutes les affectations de variables. Ainsi les valeurs que peut prendre la variable v_{Sdrv} sont rendues explicites à travers les variables intermédiaires red_0, red_1, red_2 et red_3.

7.2.2 L'application de la réduction locale

La réduction locale réduit chaque sous graphe correspondant à une instruction VHDL. Elle est appliquée une fois que les affectations de variables et de signaux présentes sur le graphe de contrôle du processus ont été modifiées.

La figure 7.11 présente la réduction d'un processus décrivant le comportement d'un automate d'états finis synchrone. La description de ce processus est traitée de la façon suivante :

1. le processus est traduit sur le modèle *XVPN* ;
2. l'étape initiale de la réduction introduit des variables intermédiaires dans les affectations de signaux et de variables. Une variable v_{Sdrv} a été ajoutée pour représenter le pilote du signal s et les variables intermédiaires $red_0, \ldots, red_5$ ont été introduites pour représenter les valeurs que peuvent prendre les variables v_{Sdrv}, v_1 et v_2 ;
3. les instructions du corps du processus sont localement réduites en commençant par l'instruction d'affectation située juste après la place *BEGIN*.

L'application de la réduction locale sur cet exemple ne nous a pas permis d'obtenir un graphe de contrôle minimal, mais elle a permis de diviser le nombre de places et de transitions par deux.

En pratique les processus à plusieurs point de suspension peuvent rarement être complètement réduit par l'application de la réduction locale. Il est donc nécessaire d'élaborer une autre technique de réduction, permettant d'obtenir une forme minimale et canonique quelle que soit la structure initiale du graphe de contrôle.

7.2.3 La réduction globale du graphe de contrôle

La réduction globale d'un processus à plusieurs points de suspension est effectuée en adoptant la même stratégie que celle employée pour les processus à un seul point de suspension. Dans un premier temps on cherche à ramener le graphe de contrôle du processus sous une forme semi-réduite équivalente. On applique dans un deuxième temps, les techniques de réduction locale qui ont été développées dans les sections 5.2.3 et 5.2.4 du chapitre 5, afin d'obtenir un graphe de contrôle minimal complètement réduit.

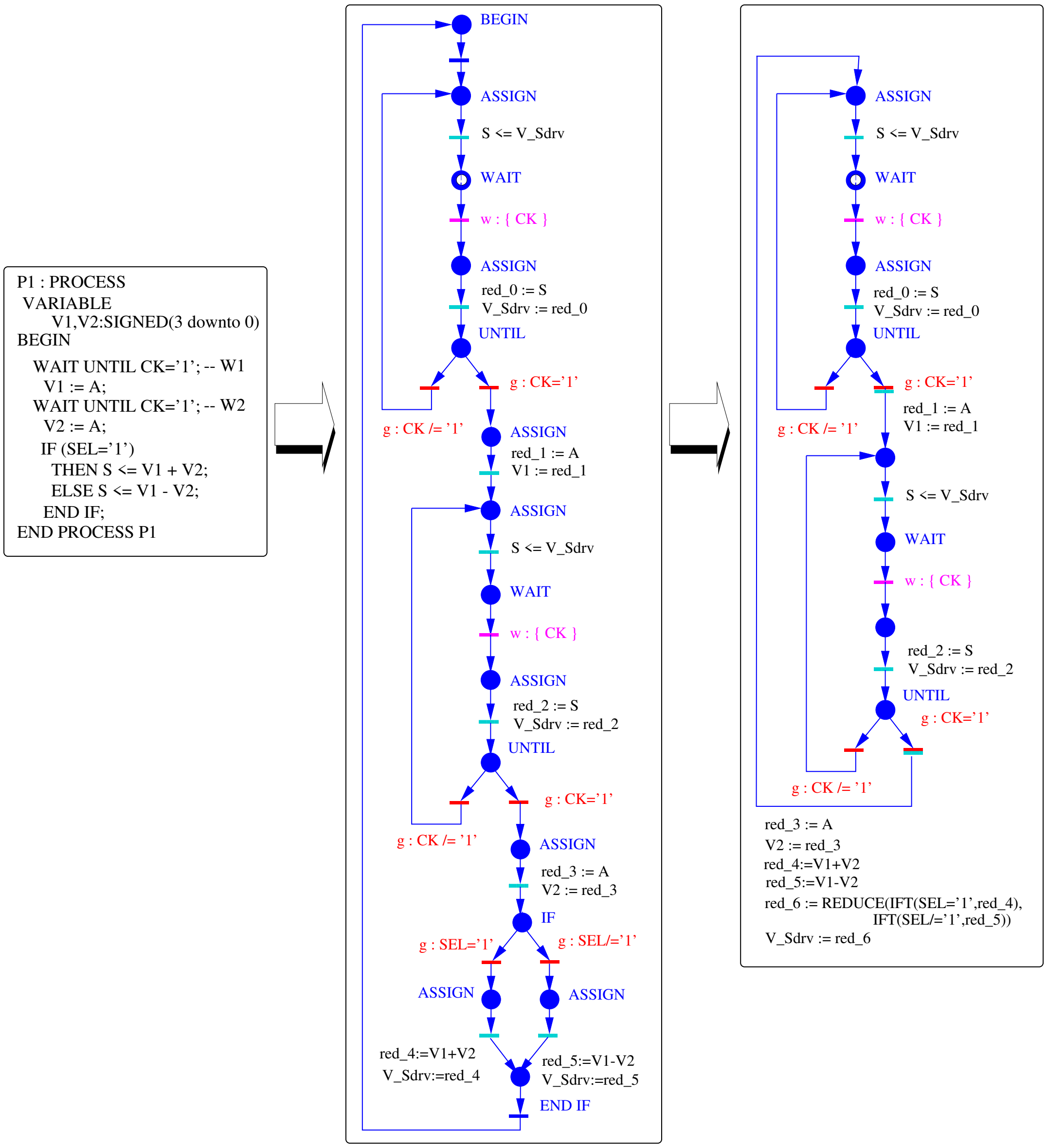

Figure 7.11: La réduction locale d'un processus à plusieurs points de suspension.

La construction du graphe de contrôle semi-réduit

Il faut tout d'abord d'identifier tous les chemins d'exécution possibles entre deux points de suspension. Il s'agit ensuite de rassembler pour chacun d'entre eux toutes les listes d'affectations et toutes les conditions de franchissement présentes afin de les réduire en une unique transition équivalente.

Soit une transition $t^i_{wait} \in T^{res}$ représentant un point de suspension du processus. On note $Succ(t^i_{wait})$ l'ensemble des transitions $t^j_{wait} \in T^{res}$ telles qu'il existe au moins un chemin d'exécution $Path(t^i_{wait}, t^j_{wait})$. Soit $\mathcal{H}^i$ le sous-graphe constitué des places, des transitions et des arcs qui sont situés sur tous les chemins permettant d'aller de la place p^i_{wait} aux transitions de $Succ(t^i_{wait})$.

La réduction globale d'un processus à plusieurs points de suspension consiste à construire pour chacun des sous-graphes $\mathcal{H}^i$ un nouveau sous-graphe semi réduit $\mathcal{H'}^i$ équivalent et facilement réductible par les techniques de réduction locale. La structure d'un sous-graphe de contrôle semi-réduit est illustrée sur la figure 7.12.

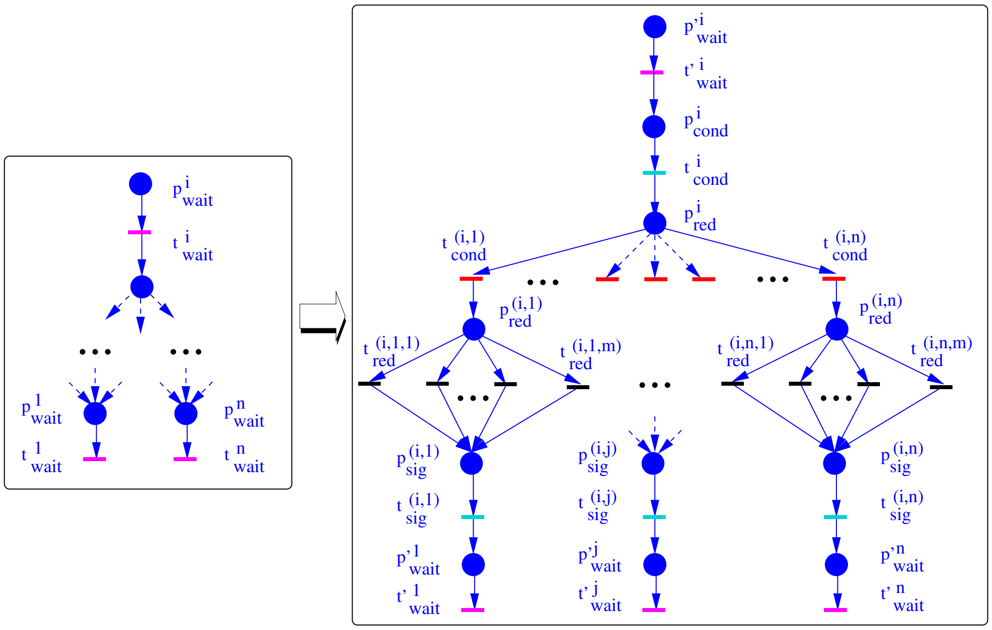

Figure 7.12: La structure d'un sous-graphe de contrôle semi-réduit.

On procède à la création d'un nouvel ensemble de places et de transitions pour chacun des points de suspension du processus :

1. des places p'^i_{wait} identiques aux places p^i_{wait} sont ajoutées à l'ensemble P ;

2. des transition t'^i_{wait} de type T_{res} identiques aux transitions t^i_{wait} sont ajoutées au réseau de Pétri. Chacune d'entre elles est ensuite connectées à la place p'^i_{wait} correspondante.

On cherche à regrouper les chemins $Path^k(t^i_{wait}, t^j_{wait})$ avec $1 \leq \ k \leq \ m_{(i,j)}$ et à réduire chacun de ces chemins en une transition équivalente $t^{(i,j,k)}_{red}$. Le sous graphe de contrôle semi-réduit compris entre $t^i_{wait} \in T^{res}$ et les transitions $t^j_{wait} \in T^{res}$ est construit de la façon suivante :

1. une place p^i_{cond} et une transition t^i_{cond} de type T_{exe} sont créées, ainsi qu'un arc reliant p^i_{cond} à t^i_{cond}. La liste $Trf(t^i_{cond})$ regroupe toutes les affectations de variables $red \in V^{red}$ qui sont effectuées sur les chemins d'exécution $Path^k(t^i_{wait}, t^j_{wait})$. L'algorithme de création de $Trf(t^i_{cond})$ est détaillé dans les paragraphes suivants ;

2. pour chacun des n points de suspension accessibles t^j_{wait} on crée une transition $t^{(i,j)}_{cond}$ de type T_{exe}. Chaque transition $t^{(i,j)}_{cond}$ possède une condition booléenne de franchissement $g(t^{(i,j)}_{cond})$ qui est le produit de toutes les conditions de franchissement présentes sur les $m_{(i,j)}$ chemins $Path^k(t^i_{wait}, t^j_{wait})$.

   Les conditions $g(t^{(i,j)}_{cond})$ peuvent dépendre de la valeur des variables affectées sur les chemins $Path^k(t^i_{wait}, t^j_{wait})$ et les affectations de variables $red \in V^{red}$ présentes sur la transition t^i_{cond} doivent permettre d'en calculer la valeur. La condition de franchissement $g(t^{(i,j)}_{cond})$ est la somme de toutes les conditions $g(t^{(i,j,k)}_{red})$;

3. pour chaque chemin $Path^k(t^i_{wait}, t^j_{wait})$ on crée une transition $t^{(i,j,k)}_{red}$ de type T_{exe} qui produit le même comportement que l'exécution séquentielle de toutes les transitions du chemin $Path^k(t^i_{wait}, t^j_{wait})$. Chaque transition $t^{(i,j,k)}_{red}$ possède une condition booléenne de franchissement $g(t^{(i,j,k)}_{red})$ qui est le produit de toutes les conditions de franchissement présentes sur le chemin $Path^k(t^i_{wait}, t^j_{wait})$. L'algorithme de création des listes $Trf(t^{(i,j,k)}_{red})$ et des conditions $g(t^{(i,j,k)}_{red})$ est détaillé dans les paragraphes suivants ;

4. une place $p^{(i,j)}_{sig}$ et une transition $t^{(i,j)}_{sig}$ de type T_{exe} sont créées, ainsi que deux arcs : l'un reliant $p^{(i,j)}_{sig}$ à $t^{(i,j)}_{sig}$ et l'autre reliant $t^{(i,j)}_{sig}$ à p'^j_{wait}. Les $m_{(i,j)}$ transitions $t^{(i,j,k)}_{red}$ sont ensuite reliées par des arcs à la place $p^{(i,j)}_{sig}$. La liste $Trf(t^{(i,j)}_{sig})$ regroupe toutes les affectations de signaux $s \Leftarrow v_{Sdrv}$ qui sont effectuées entre les transitions t^i_{wait} et t^j_{wait}. L'algorithme de création de $Trf(t^{(i,j)}_{sig})$ est détaillé dans les paragraphes suivants.

La construction des affectations du sous-graphe semi-réduit

La phase initiale de la réduction a transformé toutes les affectations de signaux s en affectations de variables v_{Sdrv}. Elle a ensuite ajouté une unique affectation $s \ \Leftarrow \ v_{Sdrv}$ pour chacun des signaux s, juste avant tous les points de suspension du processus. Aussi toutes les affectations de signaux sont identiques sur tous les chemins d'exécution. Pour construire la liste des affectations $Trf(t^{(i,j)}_{sig})$ des transitions $t^{(i,j)}_{sig}$ il suffit donc simplement de parcourir un des chemins $Path^k(t^i_{wait}, t^j_{wait})$ et de recopier toutes les affectations de signaux sur chacune des transitions $t^{(i,j)}_{sig}$.

La construction des conditions $g(t^{(i,j,k)}_{red})$ et des listes d'affectations $Trf(t^{(i,j,k)}_{red})$ des transitions $t^{(i,j,k)}_{red}$ est effectuée à l'aide de la technique d'expansion et de réduction des chemins $Path^k(t^i_{wait}, t^j_{wait})$ qui a été décrite dans la section 5.2.6. L'unique différence est que les affectations de variables $red \in V^{red}$ ne sont pas placées sur la transition $t^{(i,j)}_{cond}$ mais sur la transition t^i_{cond}.

Une fois que les listes d'affectations et les conditions de franchissement de toutes les transitions $t^{(i,j,k)}_{red}$ ont été construites, toutes les affectations de variables $red \in V^{red}$ sont présentes sur la transition t^i_{cond}.

La réduction finale d'un sous-graphe semi-réduit

La réduction finale du sous-graphe de contrôle semi-réduit est effectuée en appliquant les techniques de réduction locale définies dans les sections 5.2.3 et 5.2.4 du chapitre 5 :

- on cherche à appliquer une réduction latérale entre les deux places $p^{(i,j)}_{red}$ et $p^{(i,j)}_{sig}$. Les conditions de franchissement $g(t^{(i,j,k)}_{red})$ sont mutuellement exclusives, mais il est possible d'avoir $\sum_k g(t^{(i,j,k)}_{red}) \neq 1$. Cependant on a $g(t^{(i,j)}_{cond}) = \sum_k g(t^{(i,j,k)}_{red})$ où $t^{(i,j)}_{cond}$ est la transition qui précède les transitions $t^{(i,j,k)}_{red}$. Lorsque la transition $t^{(i,j)}_{cond}$ est franchissable, c'est que $g(t^{(i,j)}_{cond}) = 1$ et donc que $\sum_k g(t^{(i,j,k)}_{red}) = 1$. Par conséquent on peut appliquer une réduction latérale sur toutes les transitions $t^{(i,j,k)}_{red}$ puisque toutes les conditions d'application de cette réduction définies dans la section 5.2.3 sont respectées ;
- on applique ensuite successivement deux réductions séquentielles entre les deux places p^i_{red} et $p^{(i,j)}_{sig}$, puis entre les deux places p^i_{red} et p^j_{wait}.

Le résultat final des modifications structurelles apportées au sous-graphe semi-réduit est illustré sur la figure 7.13.

Tout les variables v_{Sdrv} de V^{Sdrv} qui ont été ajoutées lors de la phase d'initialisation de la réduction sont affectées une seule fois et ce toujours avant d'être lues. Une fois que tous les sous-graphes ont été réduits les variables v_{Sdrv} sont donc ajoutées à l'ensemble V^{red}.

Le graphe de contrôle d'un processus peut être réduit en utilisant exclusivement la méthode de réduction globale. Cependant la complexité de cette méthode dépend du nombre de chemins d'exécution et il est intéressant de chercher à minimiser leur nombre en utilisant les techniques de réduction locale.

La figure 7.14 présente l'application de la réduction locale sur un exemple. Le processus *P1* attend sur son premier point de suspension que l'entrée $start$ passe à '1'. Il

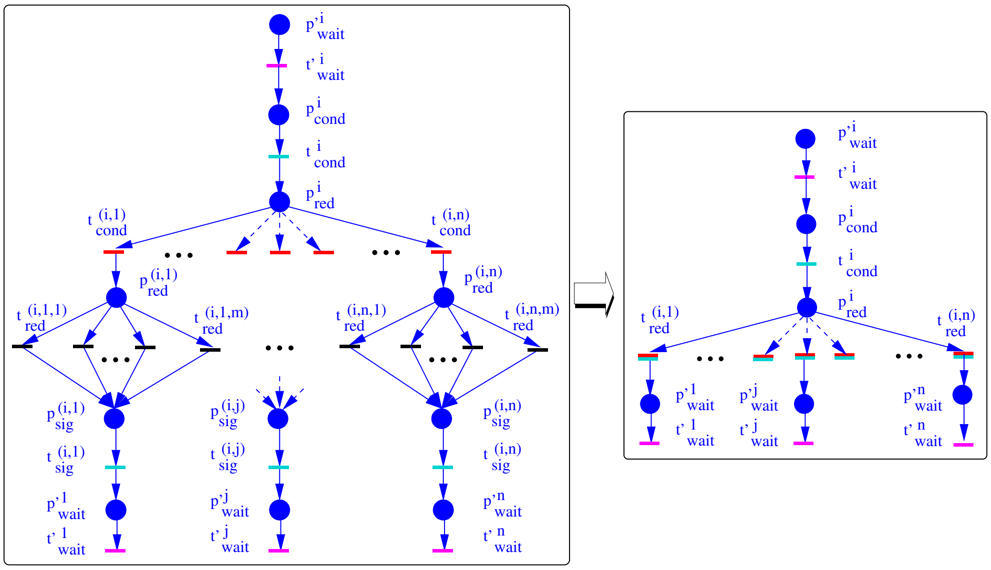

Figure 7.13: La réduction finale d'un sous-graphe semi-réduit.

effectue ensuite une série d'additions entre l'accumulateur v et l'entrée d de façon synchrone, tant que l'entrée $cont$ est à $'1'$. Lorsque l'entrée $cont$ est à $'0'$ le contenu de l'accumulateur v est recopié dans le registre de sortie s. Les deux graphes de contrôle qui sont présentés, sont ceux obtenus respectivement après l'étape initiale de la réduction et après l'application de la réduction globale.

7.3 L'analyse après réduction

L'analyse du graphe de contrôle réduit d'un processus à plusieurs points de suspension est effectuée en deux étapes : un étape d'analyse préliminaire, puis une étape de construction d'un automate d'états finis synchrone équivalent.

7.3.1 L'analyse préliminaire du graphe de contrôle réduit

L'analyse préliminaire consiste tout d'abord à substituer les occurences des constantes par leur valeur, puis à déterminer parmi tous les symboles affectés ceux qui doivent être pris en compte par la synthèse.

La substitution des constantes

Comme nous l'avons vu au chapitre 4 dans la section 4.4, certains signaux et certaines variables doivent être considérés comme des constantes lorsqu'ils ne sont plus

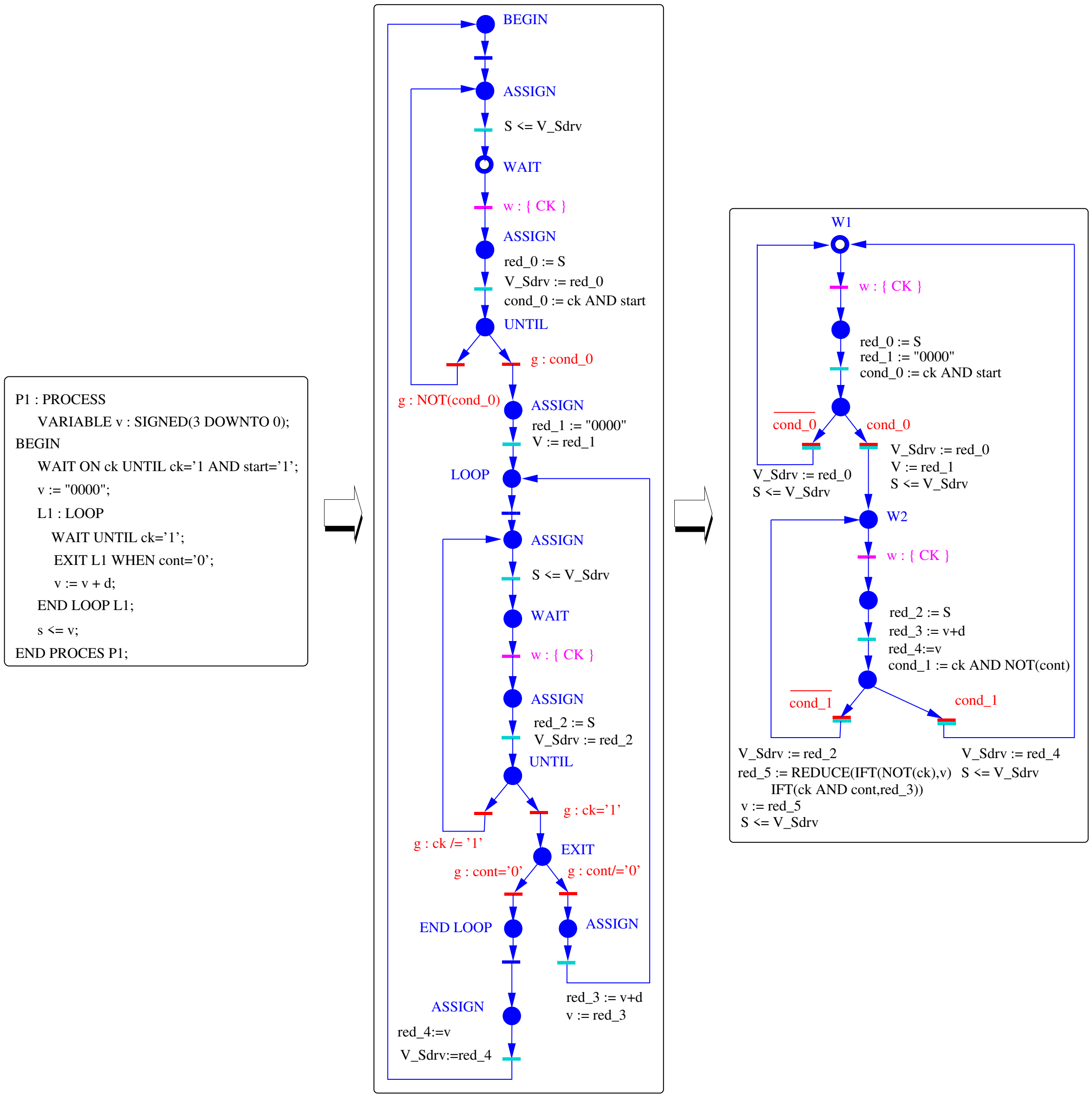

Figure 7.14: L'application de la réduction globale sur un exemple.

affectés après la phase d'initialisation de notre modèle.

On parcourt toutes toutes les transitions du graphe de contrôle réduit et on remplace dans toutes les affectations et les conditions de franchissement, ces signaux et ces variables par leur valeur calculée lors de l'initialisation de notre modèle.

Les symboles utiles à la synthèse

Comme nous l'avons vu dans la section 7.1, la synthèse ne doit prendre en compte que les variables qui interviennent dans le calcul du pilote d'un signal.

L'ensemble $V^{used} = V^{used}_{asg} \cup V^{used}_{cond}$ des variables que la synthèse doit prendre en compte est défini de la façon suivante :

- $V^{used}_{asg} = \bigcup(SuppExt(f_s) \cap \ V)$ où f_s est l'expression affectée au signal s sur une des transitions $t \in \ T^{exe}$ du graphe de contrôle réduit ;

- $V^{used}_{cond} = \bigcup(SuppExt(g(t)) \cap V)$ où $g(t)$ est la condition de franchissement d'une des transitions $t \in \ T^{exe}$ du graphe de contrôle réduit.

7.3.2 La construction d'un automate d'états finis synchrone équivalent

Du point de vue de la synthèse un automate d'états finis synchrone Fsm avec des registres en sortie est défini par : $< State, In, Out, Trans, Gen, c^{write} >$. Les ensembles $State$, In et Out sont respectivement les états, les entrées et les sorties de l'automate. L'ensemble $Trans$ contient toutes les conditions $trans(state_i, state_j)$ qui doivent être respectées par les entrées pour passer de l'état $state_i$ à l'état $state_j$. L'ensemble Gen contient les listes d'affectations concurrentes $gen(state_i, state_j)$ qui affectent les sorties lors du passage de $state_i$ à $state_j$. La condition c^{write} est la condition d'écriture des registres d'états et des registres de sorties.

L'ensemble des états de l'automate

L'ensemble des états de l'automate est construit en créant pour chaque point de suspension $t^i_{wait} \in \ T^{res}$ du graphe de contrôle réduit un état correspondant $state_i$. Le résultat de cette opération est illustré sur la figure 7.15, pour un point de suspension t^i_{wait} et ses n successeurs.

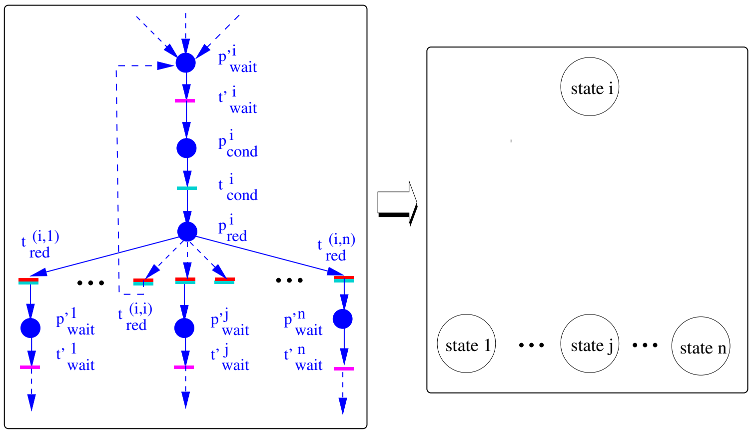

Figure 7.15: La construction de l'ensemble des états de l'automate.

L'ensemble des sorties de l'automate

L'ensemble Out des sorties de l'automate est défini comme l'ensemble des variables $v \in V^{used} \cap V^{ini}$ et des signaux s affectés sur les différentes transitions $t_{red}^{(i,j)}$.

L'identification de la condition d'écriture

Il s'agit ensuite de déterminer la condition d'écriture des registres d'états et des registres de sorties c^{write} de l'automate. On note $c^{mem} = \overline{c^{write}}$ la condition de mémorisation des différents registres.

Si le processus est synthétisable sous la forme d'un automate d'états finis synchrone, alors on a soit $c^{write} = s_{ck} \wedge s'_{ck}Event$, soit $c^{write} = \overline{s_{ck}} \wedge s'_{ck}Event$ où s_{ck} est un signal d'horloge. Ce signal est unique et il est nécessairement présent sur toutes les listes de sensibilité de toutes les transitions t_{wait}^{i}.

On note $g'(t_{red}^{(i,j)})$ la condition de franchissement d'une transition $g(t_{red}^{(i,j)})$ dans laquelle on a substitué toutes les variables $red \in V^{red}$ telles que $s_{ck} \in SuppExt(f_{red})$ où f_{red} est l'expression affectée à la variable v_{red} sur la transition t_{cond}^{i}.

L'analyse d'un processus à plusieurs points de suspension doit s'assurer que les règles suivantes sont bien respectées :

- le graphe de contrôle possède pour tous les points de suspension t_{wait}^{i} un chemin d'exécution $Path(t_{wait}^{i}, t_{wait}^{i})$;

- la condition $g'(t_{red}^{(i,i)})$ peut se mettre sous la forme :
  $g'(t_{red}^{(i,i)}) = c^{mem} \vee c_{(i,i)}^{trans}$ où $c_{(i,i)}^{trans}$ est une condition booléenne indépendante du signal s_{ck} ;

- les conditions $g'(t_{red}^{(i,j)})$ avec $(i \neq j)$ peuvent se mettre sous la forme : $g'(t_{red}^{(i,j)}) = c^{write} \wedge \ c_{(i,j)}^{trans}$ où $c_{(i,j)}^{trans}$ est une condition booléenne indépendante du signal s_{ck}.

On cherche à identifier le signal d'horloge s_{ck} qui apparaît dans la condition c^{write}. On distingue deux cas pour chacun des points de suspension t_{wait}^{i} afin d'identifier s_{ck} :

1. la liste de sensibilité $w(t_{wait}^{i})$ ne contient qu'un seul signal s. Pour que le processus soit synthétisable il est nécessaire d'avoir $s_{ck} = s$;
2. la liste de sensibilité $w(t_{wait}^{i})$ comporte plusieurs signaux s. Pour que le processus soit synthétisable il est nécessaire que l'un d'entre eux soit le signal s_{ck}. Les expressions des différentes conditions $g(t_{red}^{(i,j)})$ doivent contenir directement ou indirectement l'expression $s'_{ck}Event$. On vérifie aussi que : $g'(t_{red}^{(i,j)})(s'_kEvent \leftarrow\ 0) = 0$ avec $i \neq j$ et que $g'(t_{red}^{(i,i)})(s'_kEvent \leftarrow\ 0) = 1$.

Après avoir trouvé le signal s_{ck} et vérifié son unicité, on cherche à identifier le type du front d'horloge :

- si $\forall\ i$, $\forall\ j \neq i$ on a $g'(t_{red}^{(i,j)})(s_{ck}\ \leftarrow\ 0) = 0$ alors il s'agit d'un front montant et $c^{write} = s_{ck} \wedge\ s'_{ck}Event$;
- si $\forall\ i$, $\forall\ j \neq i$ on a $g'(t_{red}^{(i,j)})(s_{ck}\ \leftarrow\ 1) = 0$ alors il s'agit d'un front descendant et $c^{write} = \overline{s_{ck}} \wedge\ s'_{ck}Event$;
- dans tous les autres cas le processus n'est pas synthétisable.

Les transitions de l'automate

L'ensemble *Trans* des transitions de l'automate est construit en créant pour chaque $t_{red}^{(i,j)}$ du graphe de contrôle réduit une transition correspondante allant de l'état $state_i$ à l'état $state_j$.

La condition booléenne $trans(state_i, state_j)$ est égale à $c_{(i,j)}^{trans}$ et elle est calculée de la façon suivante :

- si $i \neq j$ alors on a $g'(t_{red}^{(i,j)}) = c^{write} \wedge\ c_{(i,j)}^{trans}$ et la condition $c_{(i,j)}^{trans}$ est calculée en fonction du type de front d'horloge :
  - si c'est un front montant : $c_{(i,j)}^{trans} = g'(t_{red}^{(i,j)})(s_{ck} \leftarrow 1, s'_{ck}Event \leftarrow 1)$;
  - si c'est un front descendant : $c_{(i,j)}^{trans} = g'(t_{red}^{(i,j)})(s_{ck} \leftarrow 0, s'_{ck}Event \leftarrow 1)$.
- si $i = j$ alors on a $g'(t_{red}^{(i,i)}) = \overline{c^{write}} \vee\ c_{(i,i)}^{trans}$ et la condition $c_{(i,i)}^{trans}$ est calculée en fonction du type de front d'horloge :

– si c'est un front montant : $c^{trans}_{(i,i)} = g'(t^{(i,i)}_{red})(s_{ck} \leftarrow 0, s'_{ck}Event \leftarrow 0)$;

– si c'est un front descendant : $c^{trans}_{(i,i)} = g'(t^{(i,i)}_{red})(s_{ck} \leftarrow 1, s'_{ck}Event \leftarrow 0)$.

Le résultat de cette opération est illustré sur la figure 7.16, pour un point de suspension t^i_{wait} et ses n successeurs.

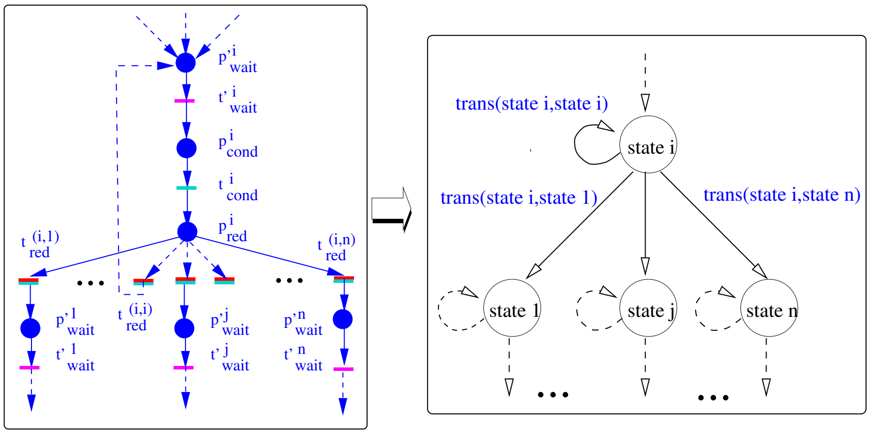

Figure 7.16: La construction de l'ensemble des transitions de l'automate.

Les fonctions de génération de l'automate

L'ensemble Gen des affectations présentes sur les différentes transitions de l'automate est construit en créant pour chaque liste $Trf(t^{(i,j)}_{red})$ une liste d'affectations concurrentes équivalente sur la transition allant de l'état $state_i$ à l'état $state_j$.

On vérifie dans un premier temps que les variables $v \in V^{ini}$ et les signaux s affectés sur les transitions $t^{(i,i)}_{red}$ conservent leur valeur lorsque la condition c^{mem} est vraie. Pour cela on parcourt chacune des listes $t^{(i,i)}_{red}$.

Soit une affectation (d, f) de $Trf(t^{(i,j)}_{red})$, on note f' l'expression affectée à d dans laquelle on a substitué toutes les variables $red \in V^{red}$ telles que $s_{ck} \in SuppExt(f_{red})$, où f_{red} est l'expression affectée à la variable v_{red} sur la transition t^i_{cond}. On note f'' l'expression affectée à d dans laquelle on a substitué toutes les variables $red \in V^{red}$ telles que $d \in SuppExt(f_{red})$ ou $s_{ck} \in SuppExt(f_{red})$.

Pour chacun des bits b de toutes les affectations $(d, f) \in Trf(t^{(i,i)}_{red})$ telles que $d \in S \cup (V^{used} \cap V^{ini})$ on vérifie que :

- $f''^b(s_{ck} \leftarrow 0, s'_{ck}Event \leftarrow 0) = d^b$ dans le cas d'une écriture sur front montant ;
- $f''^b(s_{ck} \leftarrow 1, s'_{ck}Event \leftarrow 0) = d^b$ dans le cas d'un front descendant.

Les variables $red \in V^{red} \cap V^{used}$ affectées sur les transitions t^i_{cond} et $t^{(i,j)}_{red}$ doivent être implantées à l'aide de portes combinatoires.

Les signaux s et les variables $v \in V^{used} \cap V^{ini}$ affectées sur les transitions $t_{red}^{(i,j)}$ représentent les sorties de l'automate. Ainsi pour tous les bits b d'une affectation $(d, f) \in Trf(t_{red}^{(i,j)})$ telle que $d \in S \cup (V^{used} \cap V^{ini})$ on construit une affectation sur un bit (d^b, g^b) avec :

- $g^b = f'^b(s_{ck} \leftarrow 1, s'_{ck}Event \leftarrow 1)$ dans le cas d'une écriture sur front montant ;
- $g^b = f'^b(s_{ck} \leftarrow 0, s'_{ck}Event \leftarrow 1)$ dans le cas d'un front descendant.

Après avoir calculé les différentes expressions g^b, on ajoute une affectation concurrente (d, g) dans la liste $gen(state_i, state_j)$, où g est la concaténation des différentes expressions g^b.

Lorsqu'une sortie $d \in Out$ de l'automate n'est pas affectée sur une transition alors c'est qu'elle conserve sa valeur. Dans un tel cas, on ajoute à la liste $gen(state_i, state_j)$ une affectation concurrente (d, d).

L'ensemble des entrées de l'automate

L'ensemble In des entrées de l'automate est défini comme l'ensemble des variables et des signaux qui apparaissent dans la partie droite des affectations de Gen ainsi que ceux qui apparaissent sur les conditions de transition de $Trans$.

7.3.3 Un exemple complet

La figure 7.17 montre le résultat de l'analyse d'un processus à plusieurs points de suspension. Le processus *P1* est celui qui a été présenté dans la sous-section 7.2.3 sur la figure 7.14.

Ce processus est tout d'abord traduit sur notre modèle, il est ensuite complètement réduit en utilisant la méthode présentée dans la section 7.2. Il est ensuite analysé afin de construire un automate d'états finis synchrone avec des registres en sortie équivalent $Fsm =< State, In, Out, Trans, Gen, c^{write} >$.

Le graphe de contrôle possède deux transitions t_{wait}^1 et t_{wait}^2 de type T^{res} et on a donc $State = \{state_1, state_2\}$. L'ensemble des sorties Out de l'automate est uniquement constitué du signal s et de la variable v. L'identification du signal d'horloge est effectuée en analysant les deux transitions t_{wait}^1 et t_{wait}^2. Ces deux transitions ne possèdent qu'un seul et même signal ck sur leur liste de sensibilité et il s'agit donc nécessairement d'une horloge. L'identification de la condition de mémorisation c^{write} est effectuée en analysant les différentes conditions $g(t_{red}^{(1,1)})$, $g(t_{red}^{(1,2)})$, $g(t_{red}^{(2,2)})$ et $g(t_{red}^{(2,1)})$.

Après avoir substitué les variables intermédiaires $cond_0$ et $cond_1$ de façon à faire apparaître explicitement l'horloge ck on obtient :

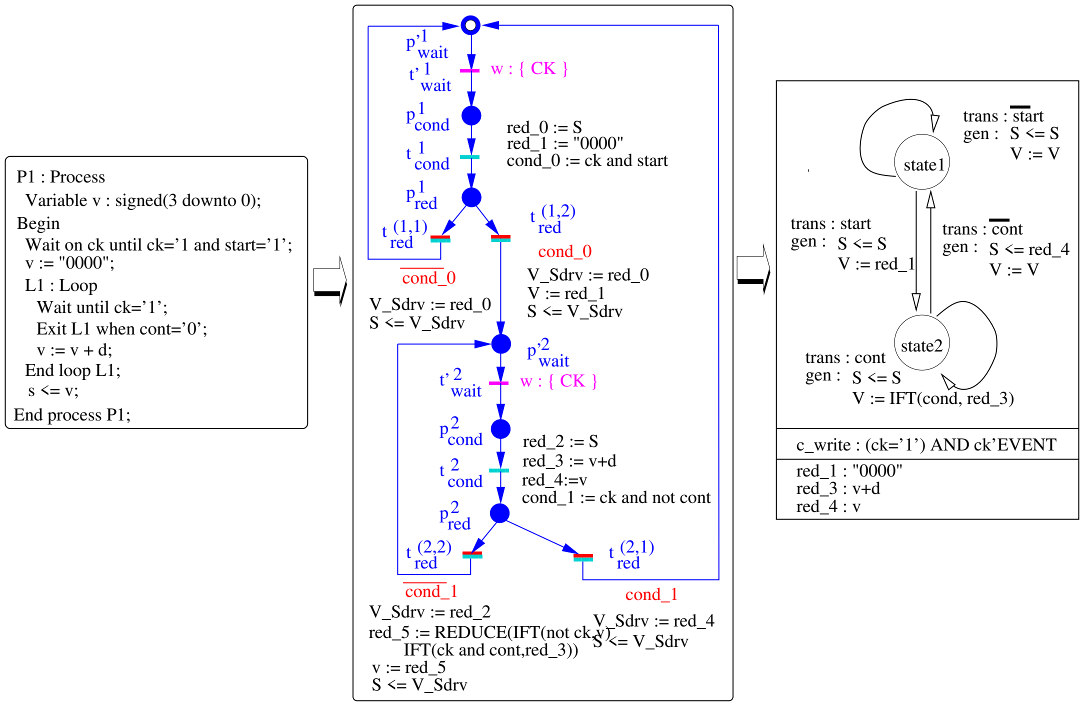

Figure 7.17: L'analyse d'un processus à plusieurs points de suspension.

- $g'(t_{red}^{(1,1)}) = \overline{ck \wedge\ start}$ et $g'(t_{red}^{(1,2)}) = ck \wedge\ start\,;$
- $g'(t_{red}^{(2,2)}) = \overline{ck \wedge\ cond}\,;$ et $g'(t_{red}^{(2,1)}) = ck \wedge\ cond.$

La condition d'écriture est donc $c^{write} = ck \wedge\ ck'EVENT$ puisque :

- $g'(t_{red}^{(1,1)})(ck \leftarrow 0) = 1$ et $g'(t_{red}^{(1,2)})(ck \leftarrow 0) = 0\,;$
- $g'(t_{red}^{(2,2)})(ck \leftarrow 0) = 1$ et $g'(t_{red}^{(2,1)})(ck \leftarrow 0) = 0.$

On construit l'ensemble des transitions $Trans$ de l'automate à partir des conditions de franchissement $g'(t_{red}^{(i,j)})$ et on obtient les résultats suivants :

- $trans(state_1, state_1) = g'(t_{red}^{(1,1)})(ck \leftarrow 0, ck'EVENT \leftarrow 0) = \overline{start}\,;$
- $trans(state_1, state_2) = g'(t_{red}^{(1,2)})(ck \leftarrow 1, ck'EVENT \leftarrow 1) = start\,;$
- $trans(state_2, state_2) = g'(t_{red}^{(2,2)})(ck \leftarrow 0, ck'EVENT \leftarrow 0) = cont\,;$
- $trans(state_2, state_1) = g'(t_{red}^{(2,1)})(ck \leftarrow 1, ck'EVENT \leftarrow 1) = \overline{cont}.$

On construit ensuite l'ensemble des listes d'affectations Gen de l'automate en vérifiant que les sorties conservent leur valeur lorsque la condition d'écriture c^{write} est fausse, et on obtient les résultats suivants :

- $gen(state_1, state_1) = \{(s, s), (v, v)\}$;
- $gen(state_1, state_2) = \{(s, s), (v, red_1)\}$;
- $gen(state_2, state_2) = \{(s, s), (v, IFT(cond, red_3)\}$;
- $gen(state_2, state_1) = \{(s, red_4), (v, v)\}$;

L'ensemble In des entrées de l'automate est construit à partir de Gen et $Trans$ et on obtient $In = \{start, cont, s, v, red_1, red_3, red_4\}$.

7.4 Conclusion

Au cours de ce chapitre nous avons présenté les solutions aux deux principaux problèmes à résoudre lors de l'analyse d'un processus à plusieurs points de suspension, à savoir :

1. comment extraire, quelle que soit la structure du graphe de contrôle, les affectations qui peuvent être appliquées à un symbole entre deux points de suspension et les conditions associées à chacune de ces affectations?
2. comment construire à partir de ces conditions et de ces affectations un automate d'états finis synchrone équivalent?

Nous avons proposé une méthode de réduction du graphe de contrôle d'un processus élaboré permettant de résoudre complètement le premier problème de l'analyse. Cette réduction nous a permis d'obtenir un graphe de contrôle réduit ayant les propriétés suivantes :

- sa structure est canonique et il a le même comportement que le graphe de contrôle initial non réduit ;
- entre deux transitions t^i_{wait} et t^j_{wait} de type T^{res} il n'y a que deux transitions : t^i_{cond} et $t^{(i,j)}_{red}$ de type T^{exe} ;
- les listes d'affectations séquentielles des t^i_{cond} et des t^i_{red} sont telles qu'un symbole est affecté au plus une fois et que toutes les variables intermédiaires sont conservées.
- la transition t^i_{cond} ne contient que des affectations de variables $red \in V^{red}$.

Nous avons ensuite exploré une méthode d'analyse du graphe de contrôle réduit exploitant ces propriétés et permettant de résoudre le second problème de l'analyse. En effet cette méthode permet de savoir si un processus peut être implanté sous la forme d'un automate d'états finis synchrone avec des registres en sortie, et surtout elle permet de construire le graphe d'états de cet automate et d'identifier formellement la condition d'écriture des différents registres.

Grâce à cette approche, nous avons pu traiter les processus à plusieurs points de suspension en se basant uniquement sur la sémantique de simulation du langage VHDL.

CHAPITRE 8

Mise en œuvre logicielle

Sommaire

Ce chapitre décrit un prototype logiciel d'analyse sémantique de descriptions VHDL en vue de la synthèse. Ce prototype appelé *VASY* (VHDL Analyzer for SYnthesis) a pour objectif de valider la méthode qui a été exposée dans les chapitres précédents.

Nous explorons tout d'abord son architecture logicielle, nous présentons ensuite les deux outils logiciels *XvpnView* et *XvpnSim* qui ont été implantés pour faciliter la mise au point du prototype. Nous précisons enfin les différentes utilisations possibles du prototype *VASY*.

8.1 L'architecture logicielle de l'analyseur *VASY*

La figure 8.1 présente l'architecture logicielle de l'analyseur sémantique de descriptions VHDL en vue de la synthèse de niveau *RTL*.

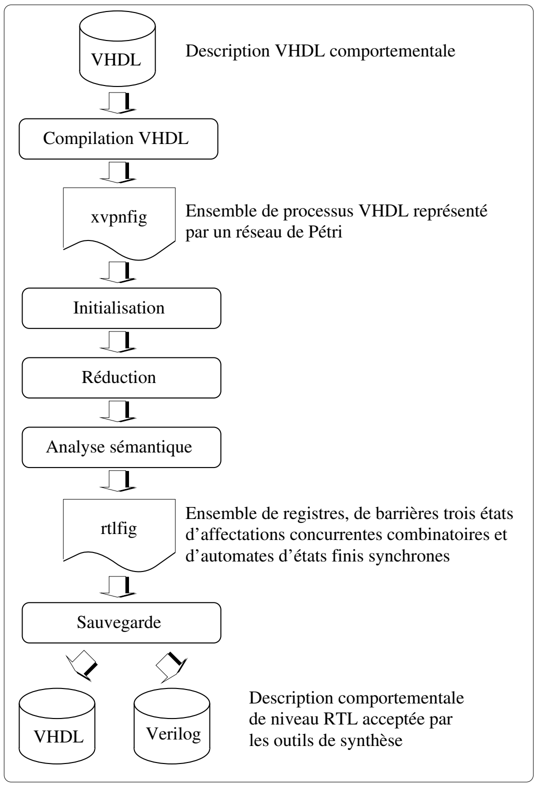

Figure 8.1: L'architecture logicielle de l'analyseur *VASY*.

Le prototype logiciel *VASY* procède en cinq étapes distinctes qui sont décrites dans les sections suivantes.

8.1.1 La compilation d'une description VHDL

La première étape est illustrée sur la figure 8.2. Elle consiste à construire en mémoire une structure de données appelée *vbhfig* (VHDL BeHavioral FIGure) . Cette structure de données est détaillée dans l'annexe D. Elle est plus riche qu'un simple ensemble d'arbres syntaxiques et elle permet de représenter :

- toutes les déclarations de types, de constantes, de signaux, de variables, de fonctions et de procédures de la description VHDL ;
- toutes les instructions séquentielles du corps des fonctions, des procédures et des processus ;
- toutes les instructions d'affectations concurrentes.

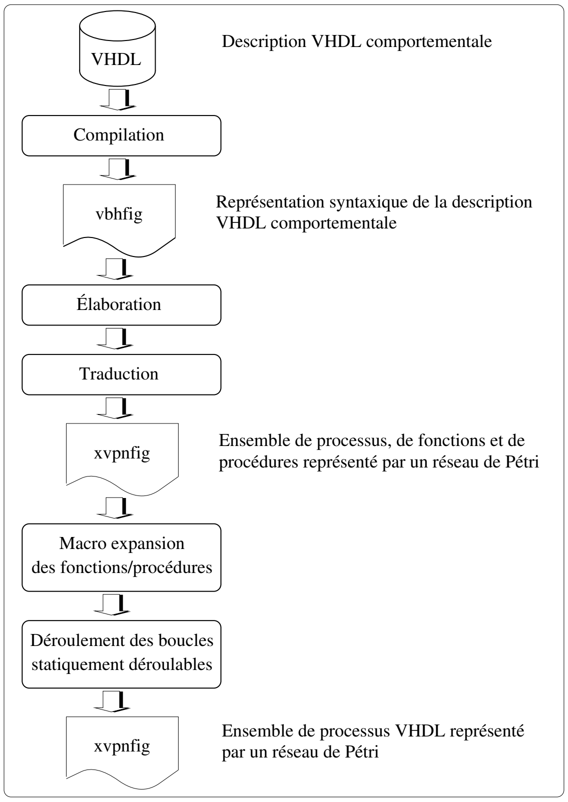

Figure 8.2: La compilation d'une description VHDL.

Lors de sa construction on vérifie que toutes les règles grammaticales et syntaxiques du langage VHDL ainsi que toutes les règles simples et légitimes imposées par la synthèse sont bien respectées. Nous avons implanté un compilateur capable de traiter un sous-ensemble spécifique à la synthèse et suffisamment large pour pouvoir valider notre prototype sur des descriptions de circuits réels. Ce sous ensemble est détaillé dans l'annexe A.

Lors de la compilation toutes les déclarations de signaux et de variables sont ramenées au niveau bit ou vecteur de bits, en conservant cependant toutes les informations sur leur type d'origine. Les expressions qui apparaissent dans les instructions séquentielles et concurrentes de la description sont représentées à l'aide d'une structure de données *vex* (VHDL EXpression) qui est détaillée dans l'annexe D.

On procède à l'élaboration de cette description en construisant un processus équivalent pour chacune des instructions concurrentes de la description. La structure de données *vbhfig* ainsi modifiée est traduite vers le modèle *XVPN* qui a été présenté dans le chapitre 4 en appliquant les techniques de construction qui ont été définies dans la section 4.3. La structure de données cible *xvpnfig* (eXtended VHDL Petri Nets FIGure) représentant le modèle *XVPN* est détaillée dans l'annexe D.

La structure de données *xvpnfig* est constituée principalement de la liste des signaux, des variables et des constantes qui ont été déclarés, et de la liste des graphes de contrôle des processus élaborés, des procédures et des fonctions. Cette structure de données *xvpnfig* est donc plus riche que la simple mise en œuvre du modèle *XVPN* puisqu'elle permet aussi de représenter l'interface et le corps des fonctions et des procédures.

Chacun des graphes de contrôle associé à un processus possède une liste de places et de transitions interconnectées par des arcs. Tous les objets de la structure de données *xvpnfig* peuvent être retrouvés rapidement grâce à l'utilisation des techniques de hachage. On utilise la structure de données *vex* dans la structure *xvpnfig* pour représenter d'une part la déclaration des symboles de la description et d'autre part toutes les expressions VHDL présentes sur les transitions du graphe de contrôle des processus.

On macro-expanse ensuite tous les appels de fonctions et de procédures définies par l'utilisateur. On obtient ainsi une structure de données *xvpnfig* qui représente exactement le modèle *XVPN* tel qu'il a été défini dans le chapitre 4, c'est à dire sans fonctions ni procédures.

Dans un souci d'homogénéité avec les outils de synthèse industriels, on déroule ensuite chacune des boucles *FOR* dont les bornes sont constantes et peuvent être évaluées statiquement, et dont le corps ne contient pas de point de suspension.

Nous avons implanté une bibliothèque de fonctions d'accès pour les trois structures de données *vbhfig*, *vex* et *xvpnfig* en adoptant une approche de programmation orientée objet. Ces bibliothèques sont constituées de fonctions permettant de créer, de modifier, de rechercher et de détruire chaque objet de la structure de données.

Le compilateur d'une description VHDL sur la structure de données *vbhfig* a également été implanté sous la forme d'une bibliothèque, ainsi que le traducteur vers la structure de données *xvpnfig*. Ce découpage permet d'utiliser le module de compilation dans d'autres contextes d'application comme nous le décrivons dans la section 8.3.

8.1.2 L'initialisation de la description

La seconde étape est illustrée sur la figure 8.3. Elle a pour objectif de calculer une valeur initiale pour tous les symboles de la description et surtout d'identifier le premier point de suspension de chacun des processus.

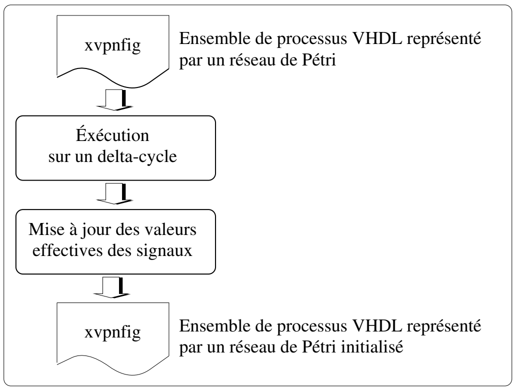

Figure 8.3: L'initialisation du modèle *XVPN*.

Comme nous l'avons vu dans la section 4.4 cette étape consiste à exécuter sur un delta-cycle l'ensemble des graphes de contrôle des processus. Nous avons donc implanté le coeur d'un simulateur respectant la norme VHDL. Ce simulateur repose directement sur la structure de données *xvpnfig* et il permet donc de manipuler uniquement les types définis pour la synthèse.

La technique de simulation utilisée est celle qui a été définie dans la section 4.4. Le simulateur procède par interprétation des conditions de franchissement et des listes d'affectations présentes sur les transitions des différents graphes de contrôle des processus de la description. Ce coeur de simulation est le même que celui utilisé par le simulateur *XvpnSim* décrit dans la section 8.2.3.

8.1.3 La réduction du modèle *XVPN*

La troisième étape est illustrée sur la figure 8.4. Elle a pour objectif de réduire le graphe de contrôle de tous les processus pour leur donner une structure minimale et canonique. Pour cela on applique les techniques qui ont été décrites dans les sections 5.2 et 7.2 sur la structure de données *xvpnfig*.

Les techniques de réduction nécessitent l'identification des différents symboles affectés sur une transition, et surtout de l'endroit où ils sont affectés dans les listes d'affectations. Nous avons donc utilisé des techniques de hachage afin d'effectuer rapidement ce type de recherche.

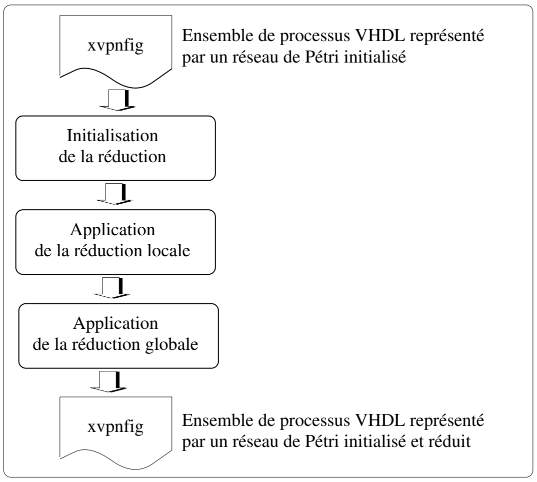

Figure 8.4: La réduction du modèle *XVPN*.

8.1.4 L'analyse sémantique du modèle *XVPN*

La quatrième étape est illustrée sur la figure 8.5. Elle a pour objectif d'analyser le graphe de contrôle réduit de tous les processus.

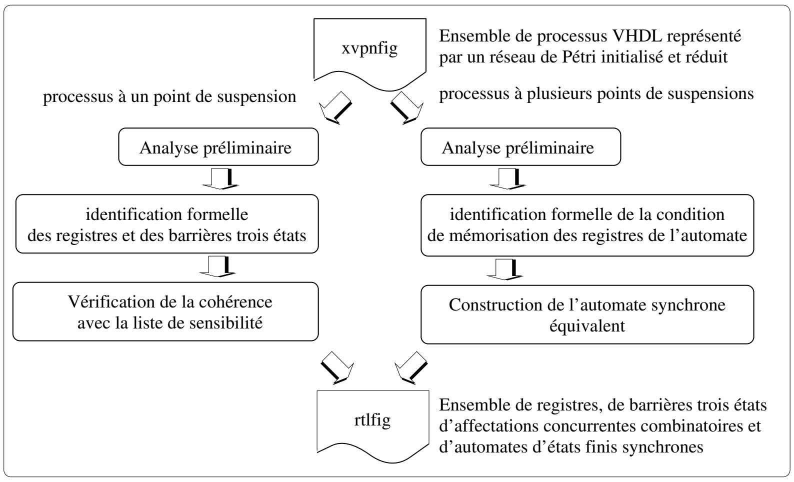

Figure 8.5: L'analyse sémantique d'une figure *xvpnfig* réduite.

La technique d'analyse est différente suivant que le processus possède un ou plusieurs points de suspension, comme nous l'avons développé dans les chapitres 5, 6 et 7. Dans le premier cas il s'agit d'identifier les symboles qui doivent être implantés en matériel avec des registres, des barrières trois états ou des portes combinatoires, alors que dans le deuxième cas il s'agit de construire un automate d'états finis synchrone équivalent.

La structure de données *rtlfig* est la cible de cette phase d'analyse. Elle est détaillée dans l'annexe D et elle est constituée principalement :

- d'une liste de déclarations de constantes et de signaux ;
- d'une liste de registres et de barrières trois états dont les propriétés sont clairement définies ;
- d'une liste d'affectations concurrentes purement combinatoires ;
- d'une liste d'automates d'états finis synchrones avec des registres en sorties.

La phase d'identification formelle des propriétés des registres et des barrières trois états repose sur l'utilisation des graphes de décisions binaires. Comme nous l'avons détaillé dans les section 5, 6 et 7.3 cette identification s'effectue de façon itérative pour chacun des symboles.

La construction des graphes de décision binaire peut être coûteuse et on doit pouvoir à tout moment libérer les nœuds des graphes qui ont été construits lors des itérations précédentes. Nous avons donc implanté une librairies de fonctions permettant de manipuler efficacement les graphes de décision binaire. Nous avons utilisé des techniques classiques de ré-ordonnancement dynamique [Rud93] afin de réduire la taille des graphes lorsqu'elle devient trop importante, et nous avons mis en place un système de calcul exact du nombre de références pour chacun des nœuds afin de pouvoir à tout moment libérer tous ceux qui ne sont plus utilisés.

8.1.5 La sauvegarde des résultats de l'analyse sémantique

La dernière étape est illustrée sur la figure 8.6. Elle consiste à sauvegarder une nouvelle description comportementale de niveau *RTL* à partir d'une figure *rtlfig*.

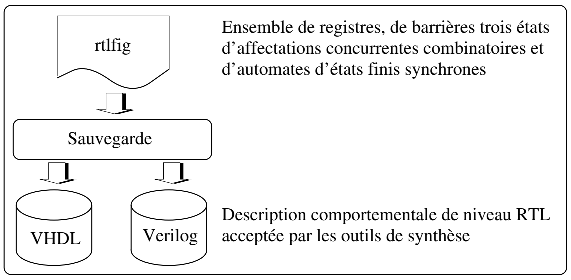

Figure 8.6: La sauvegarde des résultats de l'analyse sémantique.

Comme nous l'avons vu dans les chapitres 2 et 3, les outils de synthèse n'acceptent pas tous le même sous ensemble VHDL et ils n'imposent pas tous les mêmes contraintes syntaxiques.

Le prototype logiciel *VASY* permet donc de choisir l'outil de synthèse cible, afin de sauvegarder la description comportementale de niveau *RTL* en respectant les contraintes syntaxiques imposées par cet outil.

Il existe un style de description VHDL [IEE99] commun à la plupart des outils industriels commerciaux actuels pour décrire, sans ambiguïté pour la synthèse, le comportement des registres et des barrières trois états. C'est ce style qui est utilisé par défaut lors de la sauvegarde de la structure de données *rtlfig*.

L'outils de synthèse *BOP*[Bur93] de la chaîne de *CAO Alliance* [GP93] accepte un sous-ensemble très particulier de VHDL. Afin de pouvoir utiliser directement les résultats de l'analyseur *VASY* en entrée de cet outil de synthèse nous avons implanté un module de sauvegarde spécifique.

Afin d'élargir encore d'avantage la portée du prototype *VASY*, un module logiciel permettant de sauvegarder la description d'une figure *rtlfig* en langage *Verilog* a été implanté par Frédéric Pétrot.

8.2 L'environnement de mise au point de l'analyseur *VASY*

Pour faciliter la mise en œuvre logicielle du prototype *VASY* et en particulier pour vérifier les techniques de construction et de réduction d'une figure *xvpnfig*, nous avons développé un environnement de mise au point. Cet environnement est constitué de deux modules d'échange, d'un outil de visualisation graphique et d'un simulateur.

8.2.1 Les deux modules d'échange

Nous avons implanté les deux modules d'échanges qui sont illustrés sur la figure 8.7.

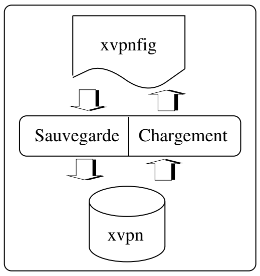

Figure 8.7: Les deux modules d'échanges.

Le premier module permet de sauvegarder une figure *xvpnfig* présente en mémoire dans un fichier sous un format texte. On peut ainsi suivre pas à pas l'exécution d'un algorithme qui modifie une figure *xvpnfig*, en sauvegardant dans des fichiers les différents états de la figure.

Le second module permet d'effectuer l'opération inverse c'est à dire de reconstruire en mémoire une figure *xvpnfig* à partir d'un fichier texte précédemment sauvegardé. Ce module est utilisé par les deux outils logiciels *XvpnView* et *XvpnSim* qui sont détaillés dans les paragraphes suivants.

8.2.2 L'outil de visualisation graphique *XvpnView*

L'outil logiciel *XvpnView* permet de visualiser graphiquement la structure de l'ensemble des graphes de contrôle d'une figure *xvpnfig*. Il permet aussi d'afficher la valeur des champs de n'importe quel élément de la structure de données. La figure 8.8 présente l'architecture logicielle de l'outil *XvpnView*.

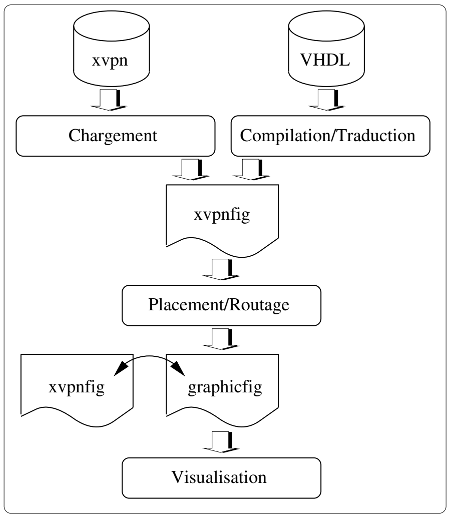

Figure 8.8: L'architecture logicielle de l'outil *XvpnView*.

Le module de chargement décrit précédemment permet de prendre en entrée un fichier texte sauvegardé par une autre application logicielle et représentant un état de la structure de données *xvpnfig*. L'outil logiciel *XvpnView* offre également la possibilité de charger directement une description VHDL qui est alors compilée et traduite en une figure *xvpnfig*.

On construit ensuite un objet graphique pour représenter chaque place, chaque transition et chaque arc de cette figure *xvpnfig* . Ces objets sont ensuite placés et routés afin d'obtenir une figure graphique affichable.

La figure 8.9 présente une utilisation de l'outil logiciel *XvpnView* lors de la mise au point des techniques de réduction implantées dans le prototype *VASY*. On peut voir sur cet exemple les différentes étapes de la réduction locale du graphe de contrôle d'un processus à un point de suspension. Une boîte de dialogue permet d'afficher la valeur de tous les champs de n'importe quel élément de la structure de données *xvpnfig*. On peut ainsi suivre l'évolution lors de la réduction des listes d'affectations et des conditions de franchissement présentes sur les transitions du graphe de contrôle.

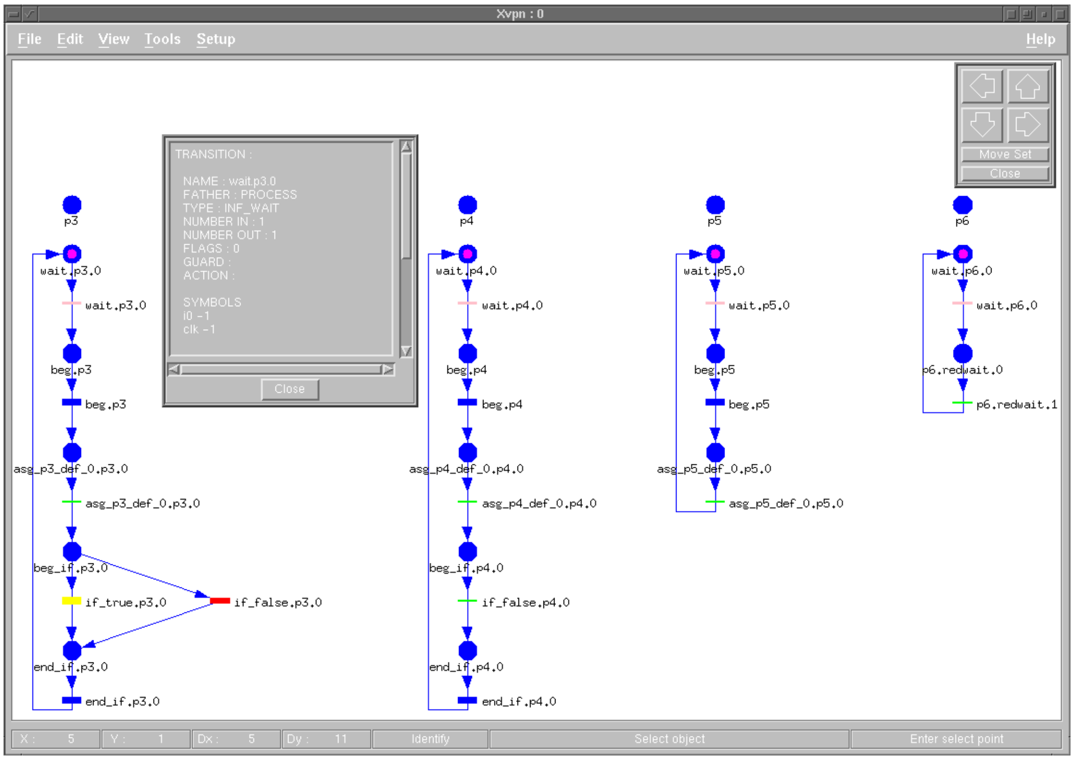

Figure 8.9: L'utilisation de l'outil de visualisation *XvpnView*.

8.2.3 Le simulateur *XvpnSim*

L'outil logiciel *XvpnSim* permet de simuler une figure *xvpnfig* en accord avec la sémantique de VHDL. La figure 8.8 présente l'architecture logicielle du simulateur *XvpnSim*.

De la même façon que l'outil logiciel *XvpnView*, le simulateur *XvpnSim* permet de prendre en entrée soit un fichier texte représentant un état de la structure de données *xvpnfig* sauvegardé par une autre application logicielle, soit une description VHDL qui est alors compilée et traduite en une figure *xvpnfig* équivalente.

Une fois que la figure *xvpnfig* est en mémoire, le simulateur charge l'ensemble des stimuli à appliquer sur les entrées de la description ainsi que les résultats attendus sur les sorties au moment de la simulation. Le format du fichier de stimuli est le même que celui du simulateur VHDL *ASIMUT*[BSV94] de la chaîne de conception Alliance [GP93].

Le simulateur applique ensuite un à un les stimuli et il vérifie que les résultats obtenus sur les sorties de la description sont en accord avec ceux qui ont été spécifiés.

Une des applications directe de l'outil logiciel *XvpnSim* est la vérification par simulation des différentes étapes de construction et de réduction d'une figure *xvpnfig*.

Il est également utilisé pour vérifier que le comportement d'une description VHDL

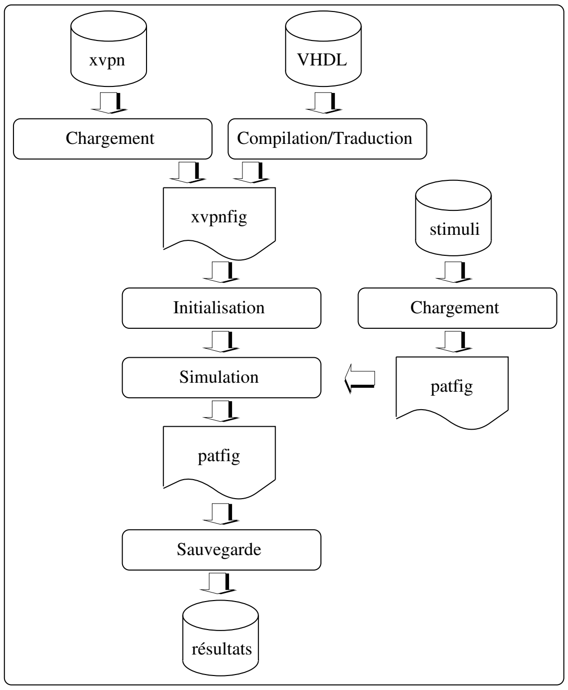

Figure 8.10: L'architecture logicielle du simulateur *XvpnSim*.

est bien le même avant et après l'analyse sémantique.

8.3 L'utilisation du module de compilation de *VASY*

La structure de données *xvpnfig* peut être utilisée dans des contextes autres que l'analyse sémantique de descriptions VHDL pour la synthèse de niveau *RTL*. Il en va de même pour le module de *VASY* qui assure la compilation d'une description VHDL et sa traduction vers la structure de données *xvpnfig*.

La figure 8.11 présente ce module de compilation. La macro-expansion des fonctions et des procédures ainsi que le déroulement des boucles sont deux techniques spécifiques à la synthèse de niveau *RTL*. Elles ne sont donc pas suffisamment générales pour être utilisées dans un autre contexte.

Aussi seule la première partie du module de compilation de *VASY* a été mise sous la forme d'une bibliothèque de fonctions. Les outils logiciel *XvpnView* et *XvpnSim* font d'ailleurs appel aux fonctions de cette bibliothèque pour effectuer le chargement d'une description VHDL en mémoire.

Cette bibliothèque est utilisée également pour la synthèse de haut niveau et la simulation mixte VHDL/C.

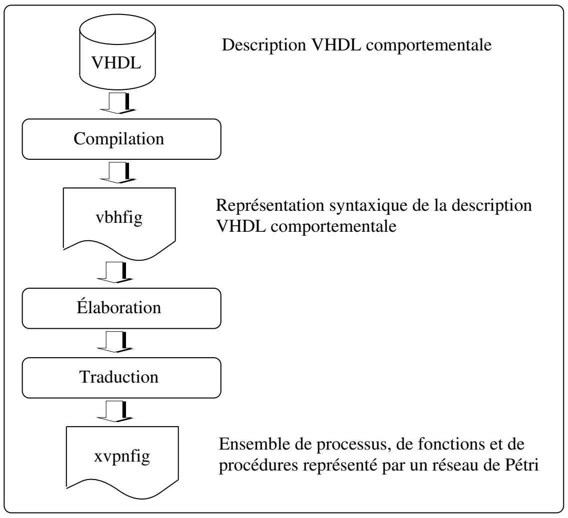

Figure 8.11: Le module de compilation d'une description VHDL.

8.3.1 La synthèse de haut niveau

L'outil de synthèse de haut niveau *UGH*[ABG+97] développé au laboratoire *LIP6* assure la transformation d'un algorithme séquentiel en une description synthétisable de niveau *RTL*. L'architecture logicielle simplifiée de l'outil *UGH* est donnée sur la figure 8.12.

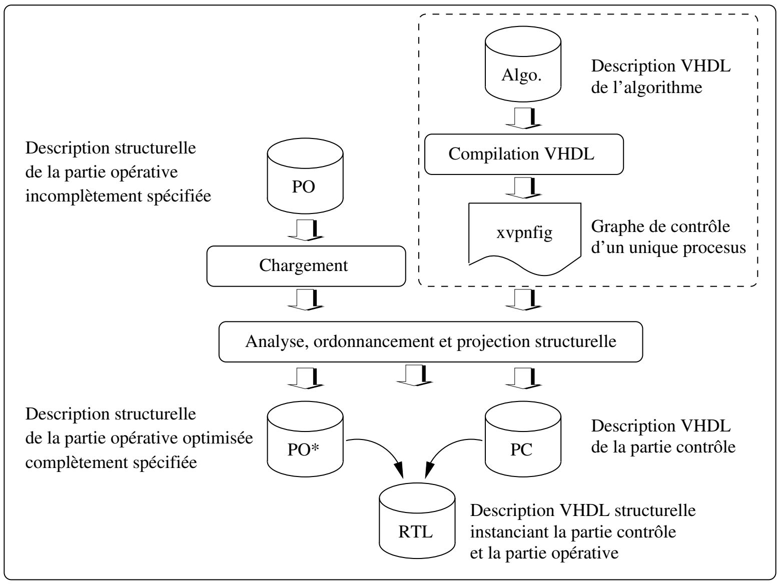

Figure 8.12: L'architecture logicielle simplifiée de l'outil *UGH*.

L'algorithme séquentiel décrivant le comportement à synthétiser est écrit en VHDL sous la forme d'un unique processus. La partie opérative d'entrée représente les différentes ressources matérielles nécessaire à l'implantation de cet algorithme et elle se présente sous la forme d'une interconnexion d'opérateurs et de registres.

L'outil logiciel *UGH* réalise un ordonnancement sous contrainte de temps de cycle. Il analyse le graphe de contrôle de l'unique processus et la partie opérative. Il en déduit l'ensemble des opérations qui doivent être exécutées cycle par cycle sur la partie opérative pour réaliser l'algorithme. Il construit également le graphe d'états de la partie contrôle capable de piloter cette partie opérative. Le résultat est une description VHDL hiérarchique synthétisable de niveau *RTL*.

Le graphe de contrôle de l'unique processus de la description de l'algorithme à synthétiser est représenté en mémoire par une figure *xvpnfig*, et de façon naturelle le compilateur VHDL utilisé est celui qui a été défini dans les paragraphes précédents.

8.3.2 La traduction d'une description VHDL en langage C

Le simulateur *CASS*[PHG97] est un outil d'évaluation de performances précis au niveau cycle, pour systèmes intégrés. Le système à simuler est décrit comme une interconnexion de processeurs ou coprocesseurs synchrones, se comportant comme des automates d'états finis. L'outil *CASS* prend en entrée un modèle comportementale décrit en langage *C* de chacun des coprocesseurs, et il assure la compilation et la simulation.

On souhaite intégrer dans la simulation des modèles de coprocesseurs écrits en VHDL. Pour cela il est nécessaire de transformer la description comportementale VHDL en un modèle décrit en langage *C* acceptable par l'outil *CASS*.

C'est l'objet du traducteur *V2Cass* dont l'architecture logicielle simplifiée est illustrée sur la figure 8.13.

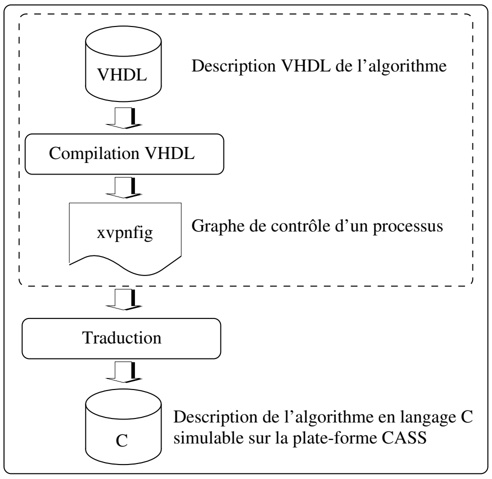

Figure 8.13: L'architecture logicielle simplifiée de l'outil *V2Cass*.

Le traducteur *V2Cass* charge en mémoire le graphe de contrôle de l'unique processus modélisant le coprocesseur sous la forme d'une figure *xvpnfig*. Il utilise pour cela le même module de compilation que l'outil de synthèse de haut niveau *UGH*.

Il traduit ensuite chacun des sous-graphes correspondant à une instruction VHDL du corps du processus, en une ou plusieurs instructions du langage *C*. Toutes les expressions VHDL présentes sur les transitions du graphe de contrôle sont également traduites en expressions équivalentes, à l'aide d'un ensemble de fonctions et de macro-fonctions écrites en langage *C* et représentant les différents opérateurs VHDL.

8.4 Conclusion

Au cours de ce chapitre nous avons tout d'abord présenté l'architecture logicielle du prototype *VASY* qui a été implanté afin de valider notre méthode d'analyse sémantique d'une description VHDL pour la synthèse de niveau *RTL*. Pour chaque module logiciel du prototype *VASY* nous avons introduit les structures de données mises en jeu et leur évolution après chaque étape.

Nous avons présenté un environnement de mise au point du prototype *VASY* constitué de deux modules d'échange, d'un outil de visualisation graphique et d'un simulateur. Nous avons montré l'utilité de cet environnement notamment pour valider l'implantation des techniques de construction et de réduction d'une figure *xvpnfig*.

Puis nous avons montré que la structure de données *xvpnfig* et que la première partie du module de compilation de *VASY* étaient suffisamment générales pour être utilisées dans d'autres contextes d'application. Nous avons décrit ensuite deux exemples concrets pour lesquels ils ont été utilisés avec succès, c'est à dire l'outil de synthèse de haut niveau *UGH* et le traducteur *V2Cass*.

CHAPITRE 9

Résultats et performances

Sommaire

Nous analysons dans ce chapitre les performances du prototype logiciel *VASY*. Notre objectif est de valider notre méthode d'analyse sémantique des descriptions VHDL en vue de la synthèse de niveau *RTL*. Il s'agit d'évaluer la qualité et la précision de l'analyse, ainsi que sa rapidité d'exécution sur des descriptions de circuits réels.

Comme nous l'avons vu dans les chapitres 5, 6 et 7 notre méthode d'analyse est différente pour les processus ne contenant qu'un seul point de suspension et ceux possédant plusieurs points de suspension. Nous précisons tout d'abord le contexte de l'expérimentation et nous traitons ensuite séparément ces deux catégories de processus.

9.1 Le contexte de l'expérimentation

Comme nous l'avons vu dans les chapitres 2 et 3, les outils de synthèse industriels imposent de fortes contraintes syntaxiques afin d'identifier facilement les éléments mémorisants et les barrières trois états. Pour pouvoir comparer notre méthode d'analyse sémantique aux méthodes de reconnaissance syntaxique nous devons définir le type de descriptions VHDL auxquelles nous allons nous intéresser.

Toutes les descriptions VHDL qui font l'objet de notre expérimentation respectent les règles légitimes de la synthèse qui ont été introduites dans le chapitre 2. Elles font parties du sous ensemble VHDL décrit dans l'annexe A.

Les seules contraintes sont les suivantes :

- les processus peuvent être analysés indépendamment les uns des autres ;
- le comportement des processus est complètement défini sur un delta-cycle ;
- les attributs *EVENT* et *STABLE* ne sont utilisés que pour décrire un mécanisme de mémorisation sur front d'horloge.

La figure 9.1 présente différentes descriptions qui ne respectent pas ces propriétés. La description *EX1* utilise la clause *AFTER* qui ne fait pas partie du sous ensemble VHDL défini pour la synthèse.

Pour analyser correctement la description *EX2* il faut prendre en compte les dépendances entre les deux processus *P1* et *P2*, ils ne peuvent donc pas être traités indépendamment.

Le processus *P3* de la description *EX3* utilise l'attribut *EVENT* pour décrire autre chose qu'un front d'horloge, et cette description sort elle aussi du cadre de notre expérimentation.

Les deux descriptions *EX1* et *EX3* sont rejetées à la compilation par tous les outils de synthèse. Dans le cas de la description *EX2* le registre *S* n'est pas identifié puisque la phase d'analyse de la compilation considère chaque processus indépendamment des autres.

```
ENTITY EX1 IS
PORT (
 A, B : in  BIT;
 S    : out BIT )
END EX1;

ARCHITECTURE RTL OF EX1
IS
BEGIN
 S <= A OR B AFTER 1 ns;
END RTL;
```

```
ENTITY EX2 IS
PORT (
 D,SEL : in  BIT;
 S     : out BIT )
END EX2;

ARCHITECTURE RTL OF EX2
IS
 SIGNAL T : BIT;
BEGIN
 P1 : PROCESS( D,T,SEL )
 BEGIN
  IF (SEL='1')
   THEN S <= D;
   ELSE S <= T;
  END IF;
 END PROCESS P1;

 P2 : PROCESS( S )
 BEGIN
  T <= S;
 END PROCESS P2;
END RTL;
```

```
ENTITY EX3 IS
PORT (
 A, B : in  BIT;
 S    : out BIT )
END EX3;

ARCHITECTURE RTL OF EX3
IS
BEGIN
 P3 : PROCESS( A,B )
 BEGIN
  IF (A'EVENT OR B'EVENT)
  THEN S <= A OR B;
  END IF;
 END PROCESS P3;
END RTL;
```

Figure 9.1: Les descriptions qui sortent du cadre de notre expérimentation.

9.2 Les processus à un seul point de suspension

Afin d'évaluer la grande variété des constructions syntaxiques acceptées par notre méthode d'analyse sémantique des processus à un seul point de suspension, nous ne nous intéressons qu'à des descriptions de petite taille, composées d'un ou deux processus VHDL.

Pour évaluer la rapidité d'exécution il est nécessaire de prendre des descriptions de circuits réels composées soit d'un nombre important de processus, soit d'un petit nombre de processus mais d'une grande complexité.

9.2.1 La mesure de la qualité de l'analyse sémantique

Les descriptions VHDL qu'il est possible d'écrire en respectant les règles qui ont été définies précédemment sont de deux types : celles qui sont implantables en matériel et celles qui ne le sont pas.

Pour chacun des outils de synthèse industriels on peut distinguer trois catégories de descriptions qui sont illustrées sur la figure 9.2 :

1. celles qui sont implantables en matériel et qui sont synthétisées correctement ;
2. celles qui sont implantables en matériel et qui ne sont pas synthétisées correctement ;
3. celles qui ne sont pas implantables en matériel et qui pourtant sont acceptées ;

Les deux dernières catégories correspondent évidemment à des comportements anormaux, et nous cherchons à montrer que notre méthode fournit dans tous les cas une réponse correcte.

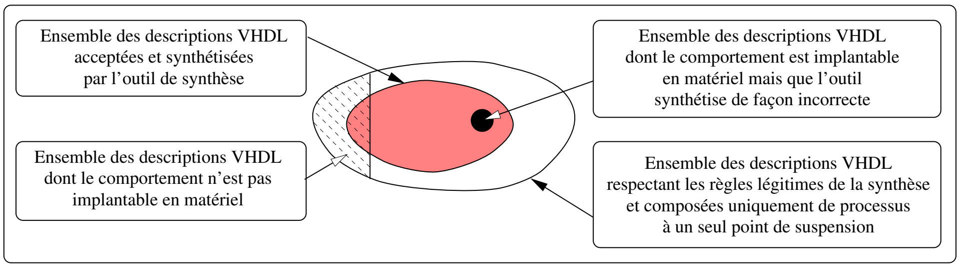

Figure 9.2: Les différentes catégories de descriptions VHDL.

Comme nous l'avons présenté dans les chapitres 2 et 3, les outils de synthèse industriels n'acceptent pas tous les mêmes descriptions. Nous cherchons à montrer que toute description synthétisable est acceptée par *VASY* et qu'il est capable de générer une description VHDL équivalente acceptée par tous les outils commerciaux.

Les descriptions synthétisables

Tout d'abord nous nous intéressons aux descriptions qui sont implantables en matériel, mais qui sont rejetées par au moins un outil de synthèse industriel. Nous nous limitons ici aux quatre outils de synthèse industriels suivants : *Autologic* de *Mentor Graphic*, *Design Compiler* de *Synopsys*, *Asic Synthesizer* de *Compass* et *Synergy* de *Cadence*, que nous noterons par la suite respectivement *AU*, *DC*, *AS* et *SY*.

Le tableau 9.1 présente les résultats de cette première expérience qui a été menée sur 38 descriptions, dont le code source est donné dans l'annexe E. Ces descriptions modélisent toutes le comportement d'un élément matériel très simple, c'est à dire une bascule D, un *latch*, ou encore un multiplexeur.

La première colonne donne le nom de la description. Les quatre colonnes suivantes donnent pour chacun des outils de synthèse le résultat obtenu lors de la compilation. Le symbole ′+′ signifie que la description a été acceptée par l'outil de synthèse, et le symbole ′-′ signifie qu'elle a été rejetée lors de la compilation. Les quatre dernières colonnes donnent pour chacun des outils de synthèse, le résultat obtenu lors de la compilation de la description VHDL équivalente générée par le prototype *VASY*. La dernière ligne du tableau donne pour chacun des outils de synthèse le pourcentage des descriptions qui sont acceptées.

Cette première expérience met en évidence les points suivants :

- il existe au niveau de la compilation une grande disparité entre ces quatre outils de synthèse. Ils n'imposent pas tous les mêmes contraintes syntaxiques et par conséquent une description acceptée par l'un n'est pas nécessairement acceptée par l'autre. Ceci remet gravement en cause la portabilité des descriptions VHDL ;

Nom	Avant analyse				Après analyse			
	AU	DC	AS	SY	AU	DC	AS	SY
d02	-	+	-	-	+	+	+	+
d03	+	-	-	-	+	+	+	+
d04	+	-	-	-	+	+	+	+
d06	+	+	+	-	+	+	+	+
d07	+	-	+	-	+	+	+	+
d08	+	-	+	-	+	+	+	+
d09	+	-	-	+	+	+	+	+
d10	-	-	-	-	+	+	+	+
d11	-	+	+	-	+	+	+	+
d12	+	-	+	-	+	+	+	+
d13	+	+	+	-	+	+	+	+
d14	+	-	-	-	+	+	+	+
d15	-	-	-	-	+	+	+	+
d16	+	-	+	-	+	+	+	+
d17	-	-	-	-	+	+	+	+
d18	-	+	+	-	+	+	+	+
d19	+	+	+	-	+	+	+	+
d20	+	-	+	+	+	+	+	+
d21	+	-	+	+	+	+	+	+
d22	-	-	+	+	+	+	+	+
d23	-	-	+	+	+	+	+	+
d25	-	-	-	+	+	+	+	+
d_Ra01	-	-	-	-	+	+	+	+
d_Ra03	+	+	-	-	+	+	+	+
d_Ra04	-	-	-	-	+	+	+	+
d_ws00	+	+	+	-	+	+	+	+
d_ws01	-	-	+	-	+	+	+	+
d_ws02	-	+	+	-	+	+	+	+
d_ws03	-	-	+	-	+	+	+	+
l00	-	+	-	-	+	+	+	+
l01	-	-	-	-	+	+	+	+
l04	+	-	-	+	+	+	+	+
l06	+	-	+	-	+	+	+	+
l15	-	-	-	-	+	+	+	+
l17	+	-	+	-	+	+	+	+
l18	-	+	-	-	+	+	+	+
mux01	+	-	-	+	+	+	+	+
mux03	-	-	-	-	+	+	+	+
%	50	29	50	21	100	100	100	100

Tableau 9.1: Les descriptions synthétisables.

- le prototype *VASY* accepte toutes ces descriptions et pour chacune d'entre elles, il génère une description VHDL équivalente. Cette nouvelle description est acceptée sans exception par les quatre outils de synthèse.

Les descriptions incorrectement synthétisées

Nous nous intéressons maintenant aux descriptions qui sont implantables en matériel, mais qui ne sont pas synthétisées correctement par au moins un des quatre outils industriels.

Le tableau 9.2 présente les résultats de cette deuxième expérience qui a été menée sur 23 descriptions dont le code source est donné dans l'annexe E. Ces descriptions modélisent toutes le comportement d'un élément matériel très simple, c'est à dire une bascule D, un *latch*, un multiplexeur ou une barrière trois états.

La première colonne donne le nom de la description. Les quatre colonnes suivantes donnent pour chacun des outils le résultat obtenu lors de la synthèse. Le symbole *'X'* signifie que la description n'a pas été acceptée lors de la compilation, le symbole '+' signifie qu'elle a été synthétisée correctement, et le symbole '-' signifie que le résultat de la synthèse n'est pas en accord avec le comportement de la description initiale. Les quatre dernières colonnes donnent pour chacun des outils, le résultat obtenu lors de la synthèse de la description VHDL équivalente générée par le prototype *VASY*. La dernière ligne du tableau donne pour chacun des outils le pourcentage des descriptions qui ont été acceptées mais qui n'ont pas été synthétisées correctement.

Le processus de la figure 9.3 modélise le comportement d'une bascule D. L'outil *Asic Synthesizer* de *Compass* accepte et synthétise correctement cette description. L'outil *Design Compiler* de *Synopsys* accepte également cette description mais le résultat de la synthèse est un *latch* au lieu d'être une bascule D.

Cette deuxième expérience met en évidence les points suivants :

- les règles syntaxiques imposées par les différents outils ne leur permettent pas toujours d'identifier correctement les éléments matériels nécessaire à la synthèse. Ceci est d'autant plus grave qu'il y a, de ce point de vue, une grande disparité entre les quatre outils. En effet une description peut être traitée correctement avec un outil et synthétisée de façon incorrecte avec un autre ;

- le prototype *VASY* accepte toutes ces descriptions et il réussi à trouver pour chacune d'entre elles, une implantation matérielle conforme à son comportement. Les descriptions équivalentes générées après l'analyse sémantique sont acceptées et synthétisées correctement par les quatre outils.

Nom	Avant analyse				Après analyse			
	AU	DC	AS	SY	AU	DC	AS	SY
d06	+	-	+	X	+	+	+	+
d07	-	X	-	X	+	+	+	+
d11	X	-	-	X	+	+	+	+
d13	+	-	+	X	+	+	+	+
d16	-	X	+	X	+	+	+	+
d18	X	-	+	X	+	+	+	+
d19	+	-	+	X	+	+	+	+
d_ws00	+	-	+	X	+	+	+	+
d_ws02	X	-	+	X	+	+	+	+
l00	X	-	X	X	+	+	+	+
l02	+	-	+	+	+	+	+	+
l03	-	+	+	+	+	+	+	+
l06	-	X	-	X	+	+	+	+
l10	X	-	+	+	+	+	+	+
l11	-	-	-	-	+	+	+	+
l12	+	-	+	+	+	+	+	+
l17	-	X	-	X	+	+	+	+
mux02	-	-	+	+	+	+	+	+
mux04	-	-	+	+	+	+	+	+
mux05	-	-	+	+	+	+	+	+
mux06	-	-	+	+	+	+	+	+
trs02	+	-	+	+	+	+	+	+
trs05	-	-	+	+	+	+	+	+
%	47	78	21	4	0	0	0	0

Tableau 9.2: Les descriptions incorrectement synthétisées.

```
library IEEE;
use IEEE.std_logic_1164.all;
entity d06 is port(
        CK : in std_logic;
        I0 : in std_logic;
        O0 : out std_logic
     );
end d06;
ARCHITECTURE BEHAVIOR OF d06 IS
BEGIN
process( CK )
begin
  if ( CK = '0' ) then
     O0 <= I0;
  end if;
end process;
END BEHAVIOR;
```

Figure 9.3: Un exemple de description incorrectement synthétisée.

Les descriptions non synthétisables

Enfin nous nous intéressons aux descriptions qui ne sont pas synthétisables mais qui sont pourtant acceptées par au moins un outil de synthèse industriel. Le tableau 9.2 présente les résultats de cette troisième expérience qui a été menée sur 9 descriptions dont le code source est donné dans l'annexe E. Ces descriptions modélisent toutes un comportement qui n'est pas implantable en matériel.

Nom	Rejetée par				Rejetée par *VASY*
	AU	DC	AS	SY	
nmat00	-	-	+	-	+
nmat01	-	-	+	-	+
nmat02	+	-	+	+	+
nmat03	-	-	-	-	+
nmat04	-	-	-	-	+
nmat05	+	-	-	+	+
nmat06	-	-	+	-	+
nmat07	+	-	+	+	+
nmat08	-	-	+	-	+
%	33	0	66	33	100

Tableau 9.3: Les descriptions non synthétisables.

La première colonne donne le nom de la description. Les quatre colonnes suivantes donnent pour chacun des outils de synthèse le résultat obtenu lors de la compilation. Le symbole ′+′ signifie que la description a été rejetée par l'outil de synthèse, et le symbole ′-′ signifie qu'elle a été acceptée à tort lors de la compilation. La dernière colonne donne le résultat obtenu lors de l'analyse de la description par le prototype *VASY*. La dernière ligne du tableau donne pour chacun des outils de synthèse le pourcentage des descriptions qui ont été rejetées.

Le processus de la figure 9.4 modélise un comportement qui ne peut pas être implanté en matériel. Cette description est rejetée par l'outil *Asic Synthesizer* de *Compass* alors qu'elle est acceptée à tort par l'outil *Design Compiler* de *Synopsys*.

```
library IEEE;
use IEEE.std_logic_1164.all;
entity nmat00 is port(
 CK,I1,I0 : in std_logic;
       O0 : out std_logic
      );
end nmat00;
ARCHITECTURE BEHAVIOR OF nmat00 IS
BEGIN
  process( CK, I0, I1 )
    variable V : std_logic;
  begin
    if ( CK = '1' )
    then V := I0;
    end if;
    O0 <= V;
    V := I1;
  end process;
END BEHAVIOR;
```

Figure 9.4: Un exemple de description non synthétisable.

Cette dernière expérience met en évidence les points suivants :

- les règles syntaxiques imposées par les différents outils de synthèse ne leur permettent pas de s'assurer qu'une description est effectivement implantable en matériel ;
- le prototype *VASY* quand à lui, rejette sans exception toutes les descriptions non synthétisables.

Les apports de notre méthode d'analyse sémantique

Notre méthode d'analyse sémantique permet d'une part de rejeter les descriptions qui ne sont pas implantables en matériel. Elle permet d'autre part de transformer une description VHDL synthétisable en une description équivalente acceptée et synthétisée correctement par tous les outils de synthèse. L'analyse sémantique permet donc de résoudre complètement tous les problèmes liés à l'identification des registres et des barrières trois états dans des descriptions composées de processus à un seul point de suspension.

9.2.2 La mesure du temps d'exécution de l'analyse sémantique

Nous cherchons à évaluer le temps d'exécution de notre méthode d'analyse sur un ensemble de descriptions VHDL de grande taille. Les mesures sont effectuées sur une machine de type *Ultra 5* cadencée à 333 MHz et possédant 256 Mo de mémoire vive.

Les descriptions de blocs matériels réels

On s'intéresse tout d'abord à des descriptions VHDL réelles, de complexité relativement importante et composées d'un nombre conséquent de processus. Le tableau 9.4 présente les résultats de l'analyse qui a été menée sur 14 descriptions. Ces descriptions modélisent le comportement soit de chemins de données, soit d'un banc de registres multi-ports, soit de blocs de logique de contrôle faisant intervenir des registres.

Nom	Li	Pr	Tr	Pl	Reg	Trs	Co	Tr	RL	RG	An	Tot
amd2901	510	52	571	446	68	8	0	1	0	-	1	2
itctldat	915	86	1351	1158	76	0	0	0	1	-	1	2
regpci	716	32	606	502	76	0	0	0	0	-	2	2
hadalu	212	16	168	142	140	0	0	0	0	-	1	1
regrx	416	17	469	390	144	0	0	0	0	-	4	4
dmarpcidat	320	25	379	316	153	72	0	0	0	-	11	11
regtx	816	41	1099	917	186	0	0	0	1	-	6	7
pcicdat	309	32	283	250	221	0	1	0	0	-	5	6
dmawpcidat	465	48	653	550	225	72	0	0	0	-	7	7
prxdat	718	65	846	713	306	0	0	0	0	-	16	17
dmarpidat	285	28	395	330	327	0	0	0	0	-	9	9
ptxdat	894	61	960	745	405	0	0	0	0	-	16	17
dmawpidat	485	40	693	542	428	0	0	0	0	-	20	20
dlxmdpt	737	44	909	660	1152	32	1	0	0	-	60	61

Tableau 9.4: Le temps d'exécution de *VASY* sur des descriptions *RTL*.

La première colonne donne le nom de la description, la seconde précise le nombre de lignes du code source VHDL et la troisième le nombre de processus. Les deux colonnes suivantes représentent respectivement le nombre de transitions et le nombre de places du modèle *XVPN* correspondant. Les deux colonnes suivantes donnent le nombre de registres et de barrières trois états qui ont été identifiés lors de l'analyse. Les colonnes suivantes donnent le temps en seconde pour chacune des étapes : la compilation, la traduction sur le modèle *XVPN*, la réduction locale, la réduction globale et l'analyse sémantique du modèle réduit. La dernière colonne donne le temps d'exécution total en seconde.

Les descriptions *itctldat*, *dmarpcidat*, *pcicdat*, *dmawpcidat*, *dmarpidat* et *dmawpidat* sont des blocs du circuit *NOE*, qui est en phase finale de conception au laboratoire *LIP6*. Les descriptions *regpci*, *regrx*, *regtx prxdat* et *ptxdat* sont des blocs de logique de contrôle du circuit *PCI-DDC*[WDS+97] qui a également été conçu dans notre laboratoire. Les descriptions *amd2901*, *hadalu* et *dlxmdpt* modélisent respectivement le comportement d'un *AMD2901*, de l'unité arithmétique et logique du processeur de traitement d'image *Hadamard* et du chemin de données du processeur 32 bits *DLXm*[HP90].

Les résultats obtenus font ressortir les points suivants :

- l'application de la réduction locale des processus à un seul point de suspension a permis d'obtenir dans tous les cas une forme complètement réduite. Il n'a donc pas été nécessaire d'employer la méthode de réduction globale qui a été exposée dans la section 5.2.6 ;
- le temps d'exécution de l'analyse sémantique est tout à fait raisonnable puisqu'il est, dans les cas les plus complexes, de quelques dizaines de secondes ;
- le temps consommé par les quatre premières étapes est négligeable puisqu'il ne dépasse jamais plus d'une seconde.
- l'étape la plus coûteuse est l'étape d'analyse sémantique puisqu'elle prend entre 50% et 99% du temps d'exécution total. Le temps pris par l'analyse sémantique dépend de façon prédominante du nombre de registres et de barrières trois états à identifier. Comme nous l'avons vu dans le chapitre 6, l'analyse des propriétés d'un registre ou d'une barrière trois états utilise les graphes de décision binaire. Les techniques de construction et de manipulation de ces graphes de décision binaire sont connues pour être d'une complexité importante.

Les descriptions d'automates d'états finis

On s'intéresse maintenant à des descriptions VHDL d'automates d'états finis dont le nombre d'états varie de 10 pour le plus petit à 109 pour le plus grand. Ces descriptions sont toutes composées de deux processus : le premier représente le changement d'état sur un front d'horloge, et le second représente les fonctions de transition et de génération de l'automate. Le tableau 9.5 présente les résultats de l'analyse qui a été menée sur 11 descriptions.

Nom	Li	Pr	Tr	Pl	Reg	Co	Tr	RL	RG	An	Tot
dmawpifsm	369	2	158	120	4	0	0	0	0	0	0
hadfsm	293	2	89	65	4	0	1	0	0	0	1
r3fsm	255	2	135	105	4	0	0	0	0	0	0
r3iocfsm	520	2	174	135	4	0	1	0	0	0	1
dmawpcifsm	775	2	234	178	5	0	0	0	0	0	0
ptxfsm	657	2	169	126	5	0	0	0	0	0	0
dlxmfsm	1826	2	1127	866	6	0	0	1	0	1	2
pcisfsm	1785	2	514	388	6	0	0	1	0	0	1
prxfsm	1785	2	514	388	6	0	0	0	0	0	0
pcicmfsm	3243	2	749	569	6	0	0	0	0	1	1
pciddcfsm	10908	2	1706	1300	7	0	1	0	0	1	2

Tableau 9.5: Le temps d'exécution de *VASY* sur des automates d'états finis.

Les automates *dmawpifsm*, *dmawpcifsm*, *pcisfsm* et *pcicmfsm* font partie du circuit *NOE*. Les descriptions *ptxfsm*, *prxfsm* et *pciddcfsm* sont les automates du circuit *PCI-DDC*[WDS+97].

Les automates *r3fsm* et *r3iocfsm* sont les automates du routeur *R3*[ZRP+96]. Les descriptions *hadfsm*, *dlxmfsm* sont respectivement les séquenceurs des processeurs *Hadamard* et *DLXm*[HP90].

Le nombre de registres à identifier est très faible en comparaison avec des chemins de données, et par conséquent malgré la complexité de certaines de ces descriptions le temps total d'exécution reste faible.

9.3 Les processus à plusieurs points de suspension

On procède de façon similaire pour évaluer les performances de l'analyse sémantique des processus à plusieurs point de suspension. La qualité et la précision de notre méthode d'analyse des processus à plusieurs points de suspension est évaluée sur un ensemble de petites descriptions. Par contre pour évaluer la rapidité d'exécution de notre méthode il est nécessaire de prendre des descriptions de circuits réels.

9.3.1 La mesure de la qualité de l'analyse sémantique

Comme nous l'avons évoqué précédemment, les descriptions VHDL qu'il est possible d'écrire en respectant les règles qui ont été définies précédemment sont de deux types : celles qui sont implantables en matériel et celles qui ne le sont pas.

En ce qui concerne les descriptions composées de processus à plusieurs points de suspension, les outils de synthèse commerciaux imposent des règles extrêmement strictes et précises et par conséquent il n'est pas possible qu'ils synthétisent de façon incorrecte une description, où qu'ils acceptent une description non synthétisable. Par contre, ils n'acceptent pas nécessairement les mêmes descriptions. Cette configuration est illustrée sur la figure 9.5.

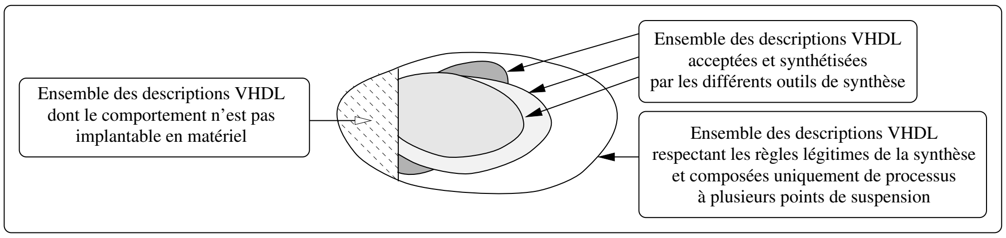

Figure 9.5: Les différentes catégories de descriptions VHDL.

Nous nous intéressons aux processus qui sont implantables en matériel sous la forme d'un automate d'états finis synchrone avec des registres en sorties, mais qui sont rejetées à la compilation par au moins un outil de synthèse.

Le tableau 9.6 présente les résultats de cette première expérience qui a été menée sur 7 descriptions, dont le code source est donné dans l'annexe F.

Nom	Avant analyse			Après analyse		
	DC	AS	SY	DC	AS	SY
mw00	+	+	+	+	+	+
mw01	-	+	-	+	+	+
mw02	+	+	-	+	+	+
mw03	+	-	-	+	+	+
mw04	-	+	-	+	+	+
mw05	+	+	-	+	+	+
mw06	+	+	+	+	+	+
%	71	85	28	100	100	100

Tableau 9.6: Les descriptions synthétisables.

La première colonne donne le nom de la description. Les trois colonnes suivantes donnent pour chacun des outils de synthèse le résultat obtenu lors de la compilation. Le symbole '+' signifie que la description a été acceptée par l'outil de synthèse, et le symbole '-' signifie qu'elle a été rejetée lors de la compilation. Les trois dernières colonnes donnent pour chacun des outils de synthèse, le résultat obtenu lors de la compilation de la description VHDL équivalente générée par *VASY* et composée exclusivement de processus à un seul point de suspension. La dernière ligne du tableau donne pour chacun des outils de synthèse le pourcentage des descriptions qui sont acceptées.

Cette expérience met en évidence les points suivants :

- tout comme les descriptions composées de processus à un seul point de suspension, il existe du point de vue de la compilation une grande disparité entre ces trois outils de synthèse. Ils n'imposent pas tous les mêmes contraintes syntaxiques et par conséquent une description acceptée par l'un n'est pas nécessairement acceptée par l'autre ;
- le prototype *VASY* accepte toutes ces descriptions et pour chacune d'entre elles, il génère une description équivalente. Cette nouvelle description est acceptée sans exception par les trois outils de synthèse.

La figure 9.6 illustre de façon générale les apports de notre méthode d'analyse sémantique des processus à plusieurs points de suspension.

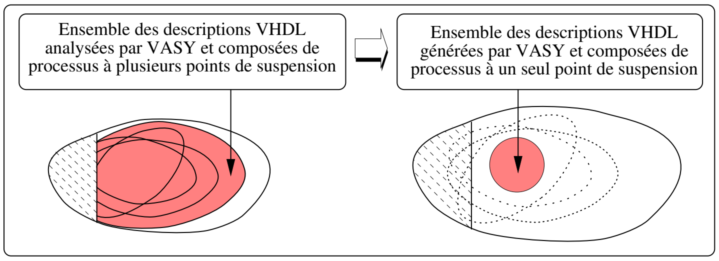

Figure 9.6: Le résultat de l'analyse sémantique du prototype *VASY*.

Elle permet d'une part de rejeter les descriptions qui ne sont pas implantables en matériel sous la forme d'un automate d'états finis synchrone avec des registres en sortie.

Elle permet d'autre part de transformer une description VHDL synthétisable en une description équivalente acceptée et synthétisée correctement par tous les outils de synthèse. L'analyse sémantique permet donc de résoudre complètement tous les problèmes de compatibilité entre les différents outils de synthèse.

9.3.2 La mesure du temps d'exécution de l'analyse sémantique

Nous cherchons maintenant à évaluer le temps d'exécution de notre méthode sur un ensemble de descriptions VHDL composées d'un unique processus à plusieurs points de suspension.

Le tableau 9.7 présente les résultats de l'analyse qui a été menée sur 4 descriptions, dont le code source est donné dans l'annexe F. Ces descriptions modélisent toutes le comportement d'un algorithme séquentiel, par exemple la résolution d'une équation différentielle.

La première colonne donne le nom de la description, la seconde précise le nombre de lignes du code source VHDL. Les deux colonnes suivantes représentent respectivement le nombre de transitions et le nombre de places du modèle *XVPN* correspondant.

Les colonnes suivantes donnent le temps en seconde pour chacune des étapes : la compilation, la traduction sur le modèle *XVPN*, la réduction locale, la réduction globale et l'analyse sémantique du modèle réduit. La dernière colonne donne le temps d'exécution total en seconde.

Nom	Li	Tr	Pl	Co	Tr	RL	RG	An	Tot
crc	57	22	19	0	0	0	0	0	0
diffeq	32	18	15	0	0	0	0	0	0
multi	72	35	29	0	0	0	0	0	0
pgcd	67	36	30	0	0	0	0	0	0

Tableau 9.7: Le temps d'exécution de *VASY* sur des exemples réels.

Les résultats obtenus font ressortir les points suivants :

- dans tous les cas, l'application de la réduction locale n'a pas permis d'obtenir un graphe de contrôle minimal, et il a donc été nécessaire d'appliquer la méthode de réduction globale qui a été exposée dans la section 7.2.3 ;
- le temps d'exécution de l'analyse sémantique est négligeable puisqu'elle ne dépasse jamais une seconde.

9.4 Conclusion

Dans ce chapitre nous avons mené en ensemble d'expériences sur un total de 100 descriptions VHDL, afin de montrer les performances de notre méthode d'analyse sémantique tant sur le plan qualitatif que sur le plan quantitatif.

Nous nous sommes tout d'abord intéressés aux descriptions VHDL ne comportant que des processus à un seul point de suspension. Nous avons mené trois séries d'expériences qui ont mis en évidence les problèmes suivants :

- les outils de synthèse industriels n'imposent pas tous les mêmes contraintes syntaxiques, et ils n'acceptent donc pas tous les mêmes descriptions. Cette disparité pose un problème grave de portabilité lorsqu'on cherche à faire migrer la description d'un circuit, d'une plate-forme de synthèse vers une autre ;
- les outils industriels commettent des erreurs d'interprétation lors de la compilation, et le résultat de la synthèse n'est pas nécessairement en accord avec le comportement de la description initiale.
- les outils industriels ne sont pas non plus capables de discerner avec certitude, les descriptions qui sont synthétisables de celles qui ne le sont pas ;

Nous avons montré expérimentalement que notre méthode d'analyse des processus à un seul point de suspension permettait de résoudre ces trois problèmes graves.

Nous avons effectué deux séries d'expériences afin de valider les performances temporelles de *VASY*. Nous avons donc confronté notre prototype à des descriptions de circuits réels composées soit d'un nombre important de processus, soit d'un petit nombre de processus mais d'une grande complexité.

Les résultat obtenus montrent que l'analyse sémantique prend moins d'une seconde sur la plupart des exemples, et qu'elle prend quelques dizaines de secondes sur les descriptions les plus complexes composées essentiellement de chemins de données. Ces temps d'exécution sont donc tout à fait satisfaisants.

Nous nous sommes ensuite intéressés aux descriptions VHDL ne comportant que des processus à plusieurs points de suspension. Nous avons mené une autre série d'expériences qui ont une fois de plus mis en évidence la disparité des sous ensembles VHDL acceptés par les différents outils.

Nous avons montré que notre méthode d'analyse des processus à un plusieurs points de suspension permettait de résoudre ce problème, en générant une description équivalente composée uniquement de processus à un seul point de suspension et acceptée par tous les outils. Les résultat obtenus ont montré que l'analyse sémantique ne prenait pas plus d'une seconde sur tous les exemples réels traités.

CHAPITRE 10

Conclusions et perspectives

Sommaire

Dans ce chapitre nous dressons tout d'abord le bilan de la méthode proposée dans le cadre de cette thèse. Nous donnons ensuite quelques perspectives de recherches qui peuvent être entreprises à la suite de nos travaux.

10.1 Le bilan des travaux effectués

Dans cette thèse nous avons tout d'abord analysé les particularités du langage de description de matériel VHDL et nous avons insisté sur le fait qu'il a été initialement défini pour la simulation. Nous avons ensuite mis en évidence les problèmes posés lors de son utilisation comme langage de spécification de la synthèse de niveau *RTL*.

Nous avons montré que la phase d'analyse de la compilation en vue de la synthèse devait permettre de répondre principalement aux trois questions suivantes :

- comment analyser et vérifier qu'une description VHDL est synthétisable?
- comment extraire d'un processus VHDL séquentiel un ensemble d'affectation concurrentes, tout en restant en accord avec le comportement défini par la sémantique de simulation du langage?
- comment identifier pour chaque symbole de la description, le type d'élément matériel (registre, barrière trois états, porte combinatoire) nécessaire à son implantation?

Nous avons distingué deux approches possibles pour résoudre ces différents problèmes : les méthodes d'analyse syntaxique et les méthodes d'analyse sémantique. Nous avons montré que les méthodes d'analyse syntaxique sont certes simple à mettre en œuvre d'un point de vue logiciel, mais qu'elles ne permettent pas toujours d'atteindre les objectifs de la phase de compilation de la synthèse. Nous avons ensuite mis en évidence les limitations des méthodes d'analyse sémantique existantes.

Les méthodes d'analyse sémantique s'appuient sur un modèle formel de VHDL. Nous avons choisis le modèle *VPN*[Baw96] qui a été défini par l'équipe d'algorithmique numérique du laboratoire *LIP6* et qui a été utilisé avec succès pour la vérification de propriété et la preuve formelle de descriptions VHDL[Enc95, Baw96].

Nous avons montré la nécessité d'étendre ce modèle pour l'utiliser dans le cadre de la synthèse de niveau *RTL*. Nous avons donné une définition formelle du modèle étendu *XVPN* adapté aux besoins de la synthèse, et nous avons développé toutes les techniques de traduction d'une description VHDL sur ce modèle.

Nous avons distingué deux catégories de processus VHDL : ceux qui comportent un seul point de suspension et ceux qui en comportent plusieurs. Nous avons insisté sur le fait que les objectifs de l'analyse étaient différents pour ces deux catégories de processus. Nous avons ainsi mis en évidence la nécessité d'élaborer une méthode d'analyse sémantique spécifique pour ces deux catégories de processus, afin d'atteindre complètement les objectifs de la synthèse.

Nous avons détaillé une méthode d'analyse sémantique des processus à un seul point de suspension, constituée de deux principales étapes :

- la première étape consiste à appliquer un ensemble de techniques de réduction sur le graphe de contrôle du processus à analyser afin d'obtenir une forme canonique et minimale. Ces techniques de réduction préservent le comportement du processus vis à vis de la sémantique de simulation de VHDL. Elles permettent également de conserver toutes les informations pertinentes pour les étapes ultérieures de la synthèse ;

- la seconde étape consiste à analyser les affectations du graphe de contrôle réduit d'un processus, afin d'identifier formellement et précisément le type d'élément matériel (registre, barrière trois états, ou porte combinatoire) nécessaire à l'implantation de chacun des symboles affectés. Cette seconde étape permet également de s'assurer qu'un processus est effectivement synthétisable.

Nous avons ensuite détaillé une méthode d'analyse sémantique des processus à plusieurs points de suspension, constituée elle aussi de deux principales étapes :

- la première étape est comparable a celle utilisée lors de l'analyse des processus à un seul point de suspension, et elle assure la réduction du graphe de contrôle d'un processus afin d'obtenir une structure canonique ;

- la seconde étape consiste à transformer le graphe de contrôle réduit d'un processus en un automate d'états finis synchrone équivalent, et à identifier formellement la condition de mémorisation du registre d'états.

Nous avons présenté en détail l'architecture logicielle du prototype *VASY* qui a été implanté afin de valider notre méthode d'analyse sémantique. Puis nous avons montré que le module de compilation de ce prototype était suffisamment général pour être utilisé dans d'autres contextes d'application, et nous en avons décrit deux exemples concrets.

Enfin nous avons mené en ensemble d'expériences sur une centaine de descriptions VHDL et nous avons montré les bonnes performances de notre méthode d'analyse sémantique tant sur le plan qualitatif que sur le plan quantitatif.

En effet, contrairement aux méthodes d'analyse syntaxique notre méthode permet d'atteindre tous les objectifs de la phase de compilation de la synthèse, avec un temps d'exécution tout à fait satisfaisant même sur des descriptions d'une grande complexité.

10.2 Les perspectives de recherches

Comme nous l'avons évoqué dans les chapitres 2 et 6, de même que les méthodes utilisées par les outils de synthèse commerciaux, notre méthode considère chacun des processus indépendamment des autres, et elle analyse le comportement permanent d'un processus sur un unique delta-cycle.

On peut imaginer relâcher ses deux contraintes. Ceci permettrait d'offrir une plus grande liberté d'écriture aux concepteurs, mais surtout de détecter précisément les éventuelles boucles combinatoires présentes dans une description.

Notre méthode d'analyse sémantique d'une description VHDL en vue de la synthèse de niveau *RTL* peut être utilisée en particulier dans le domaine de la preuve formelle.

La complexité des techniques formelles dépend principalement du nombre de symboles à représenter et de la complexité des expressions qui leur sont affectées.

Yee-Wing Hsieh et Steven P. Levitan ont proposé une méthode d'analyse préliminaire d'une description VHDL[HL97], afin d'identifier tous les symboles qui sont susceptibles de conserver leur valeur. D'une façon générale, cette méthode consiste à parcourir les différents chemins d'exécution d'un processus VHDL, et à établir la liste des symboles qui ne sont pas affectés sur tous les chemins.

Une fois cette identification effectuée, on peut considérer tous les autres symboles internes de la description comme des symboles intermédiaires. Lors de la construction de la relation de transition qui sert de base à la vérification formelle, on peut substituer les symboles intermédiaires par la fonction qui leur est affectée, sans altérer le comportement global du système. Cette analyse préliminaire permet ainsi de réduire le nombre des symboles, et par conséquent le nombre des états du système, facilitant ainsi l'application des techniques de vérification formelle.

Notre méthode d'analyse sémantique identifie en particulier les symboles qui conservent leur valeur. On peut donc chercher à l'appliquer dans ce domaine afin d'obtenir des résultats d'une meilleure précision.

Bibliographie

[ABG+97] I. Augé, R.K. Bawa, P. Guerrier, A. Greiner, L. Jacomme, and F. Pétrot. User guided high level synthesis. In *International Conference on Very Large Scale Integration (VLSI'97)*, pages 464–475, Gramado, Brésil, Aout 1997.

[Ake78] S.B. Akers. Binary decision diagram. In *Transaction on computers*, volume c-27, USA, Juin 1978. IEEE.

[Baw96] R.K. Bawa. *Un environnement intégré pour la vérification formelle et l'analyse des systèmes décrits en VHDL*. PhD thesis, Université Pierre et Marie Curie, laboratoire MASI, Paris, France, Décembre 1996.

[BRB90] K. Brace, R. Rudell, and R. Bryant. Efficient implementation of a bdd package. In *27th Design Automotion Conference*, pages 40–45. ACM/IEEE, 1990.

[Bry86] R. Bryant. Graph-based algorithm for boolean function manipulation. In *Transaction on computers*, volume c-35, pages 677–691. IEEE, Aout 1986.

[BSV94] P. Bazargan-Sabet and H.N. Vuong. Trade-offs in designing a vhdl simulation tool for digital vlsi. In *26th IEEE Southeastern Symposium on System Theory*, pages 380–384, Athens, Ohio, USA, Mars 1994.

[Bur93] L. Burgun. *Synthèse logique de circuits intégrés VLSI: Utilisation de graphes de décision binaire dans la phase d'optimisation comportementale*. PhD thesis, Université Pierre et Marie Curie, Paris, France, Décembre 1993.

[BVB97] D. Borrione, F. Vestman, and H. Bouamama. An approach to verilog-vhdl interoperability for synchronous designs. In *IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods CHARME'97*, Montreal, Canada, Octobre 1997.

[Cad97] Cadence Design Systems Incorporation. *Synergy 3.0-p001*, 1997.

[Com96] Compass Design Automation Incorporation. *AsicSynthesizer v4.2*, 1996.

[Cou91] O. Coudert. *SIAM: Une boite à outils pour la preuve formelle de systèmes séquentiels*. PhD thesis, École Nationale Supérieure des Télécommunications, Paris, France, Octobre 1991.

[DB93a] D. Déharbe and D. Borrione. A qualitative finite subset of vhdl and semantics. Technical report, IMAG Institute, 1993.

[DB93b] D. Déharbe and D. Borrione. Symbolic model checking of vhdl design entities. Technical report, IMAG Institute, Décembre 1993.

[DB95] D. Déharbe and D. Borrione. Semantic of a verification-oriented subset of vhdl'87. In *IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods, CHARME'95*, page Octobre, Frankfurt, Allemagne, 1995.

[Döh94] G. Döhmen. Petri nets as intermediate representation between vhdl and symbolic transition systems. In *EURO-VHDL*, pages 572–577, Grenoble, France, 1994.

[DJP97] J. Dunoyer, L. Jacomme, and F. Pétrot. Stratégies de codage des automates pour les applications basse-consommation : Expérimentations et interprétations. In *Journées d'études Faible Tension Faible Consommation*, pages 145–152, Paris, France, Novembre 1997.

[DO98] E. Dumitrescu and P. Ostier. Identification of non-redundant memorizing elements in vhdl synchronous designs for formal verification tools. In *SMACD'98*, Kaiserslautern, Octobre 1998.

[DPJ97a] J. Dunoyer, F. Pétrot, and L. Jacomme. Intrinsic limitations of logarithmic encodings for low power finite state machines. In *Mixed Design of VLSI Circuits Conference*, pages 613–618, Poznan, Pologne, Juin 1997.

[DPJ97b] J. Dunoyer, F. Pétrot, and L. Jacomme. Limitations of logarithmic encodings for low power finite state machines. In *4th IEEE International Conference on Electronics Circuits and Systems (ICECS'97)*, pages 522–528, Le Caire, Égypte, Décembre 1997.

[EBC93] E. Encrenaz, R.K. Bawa, and J.M. Couvreur. Validation of vhdl systems based on petri net modeling. In *Workshop on Design Methodologies for Microelectronics and Signal Processing*, Pologne, 1993.

[EE95] A. Evans and E. Encrenaz. An approach to the synthesis of vhdl concurrent processes as a fsm. In *IFIP WG 10.5 Workshop on Logic and Architecture Synthesis*, pages 115–124, Grenoble, France, 1995.

[Enc95] E. Encrenaz. *Une méthode de vérification de propriétés de programmes VHDL basée sur des modèles formels de réseaux de Petri*. PhD thesis, Université Pierre et Marie Curie, laboratoire MASI, Paris, France, Décembre 1995.

[GDW92] D. Gajski, N.D. Dutt, and A. Wu. *High Level Synthesis : Introduction to Chip and System Design*. Kluwer Academic Publishers, 1992.

[GP93] A. Greiner and F. Pécheux. Alliance : A complete set of cad tools for teaching vlsi design. In *3rd Eurochip Workshop on VLSI Design Training*, pages 230–237, 1993.

[HL97] Y-W. Hsieh and S.P. Levitan. Control data-flow analysis for vhdl semantic extraction. In *4th Asia-Pacific Conference on Hardware Description Language*, pages 68–75, 1997.

[HP90] J.L. Hennessy and D.A. Patterson. *Computer architecture, a quantitative approach*. Morgan Kaufmann Publisher Inc., 1990.

[IEE87] IEEE. *Standard VHDL Language Reference Manual 1076*, 1987.

[IEE93a] IEEE. *Standard Multivalue Logic System for VHDL Model Interoperability 1164*, 1993.

[IEE93b] IEEE. *Standard VHDL Language Reference Manual 1076*, 1993.

[IEE95] IEEE. *Standard Hardware Description Language based on the Verilog HDL, Standard 1364*, 1995.

[IEE97] IEEE. *Standard VHDL Synthesis Packages 1076.3*, 1997.

[IEE99] IEEE. *Standard for VHDL Register Transfert Level Synthesis, Draft 1.12a 1076.6*, 1999.

[JP96] L. Jacomme and F. Pétrot. Du codage des automates en vue de la synthèse sur silicium : comparaison systématique et critique des approches existantes. In *IIème Conférence Nationale sur la Résolution Pratique des Problèmes NP-Complets (CNPC'96)*, pages 293–302, Dijon, France, Mars 1996.

[KB95] C. Delgado Kloos and P.T. Breuer. *Formal Semantics for VHDL*. Kluwer Academic Publishers, 1995.

[MB96] J. Mirkowski and K. Bilinski. Petri net modelling of vhdl simulation cycle for high level synthesis purposes. In *VHDL User Forum in Europe SIG-VHDL Spring'96 working conference*, Dresden, 1996.

[Mek98] G.E. Mekenkamp. *A New Approach to VHDL-Based Synthesis*. PhD thesis, University of Twente, Janvier 1998.

[Men95] Mentor Graphic Incorporation. *Autologic II*, 1995.

[MM95] E. Molenkamp and G.E. Mekenkamp. Sil: an intermediate for syntax based vhdl synthesis. In *VHDL International Users Forum*, pages 5.1–5.9, Avril 1995.

[MMM+96] G. Mekenkamp, P.F.A. Middelhoek, B.E. Molenkamp, J. Hofstede, and T. Krol. A syntax based vhdl to cdfg translation model for high-level synthesis. In *VHDL International Users Forum Spring'96*, pages 89–97, Santa Clara, USA, Février 1996.

[Mor82] B.M.E. Moret. Decision tree and diagrams. In *Computing Surveys*, volume 14, pages 593–623, Décembre 1982.

[Mur89] A. Murata. Petri nets: Properties analysis and applications. In *IEEE*, volume 77, pages 541–580, Avril 1989.

[OC93] S. Olcoz and J.M. Colom. A petri net approach for the analysis of vhdl descriptions. In *CHARME*, pages 15–26, 1993.

[Olc95] S. Olcoz. A petri net model of vhdl: A base to apply formal methods to hardware design. In *VHDL International Users Forum*, pages 10.21–10.37, Avril 1995.

[PHG97] F. Pétrot, D. Hommais, and A. Greiner. A simulation environment for core based embedded systems. In *30th Annual Simulation Symposium*, pages 86–91, Atlanta, Géorgie, Avril 1997.

[RB98] D. Rahal-Besseghir. *Réutilisation de descriptions VHDL synthétisables: Migration mutli-plateformes*. PhD thesis, Université Louis Pasteur, Strasbourg, France, Avril 1998.

[Rud93] R. Ruddel. Dynamic variable reordering for binary decision diagrams. In *International Conference on Computer-Aided Design*, pages 42–47, Novembre 1993.

[SEV95] M. Selz, W. Ecker, and E. Villar. Vhdl synthesis description portability: the need for level-x synthesis subsets. In *VHDL Forum for CAD in Europe Spring'95 Working Conference*, 1995.

[Syn98] Synopsys Incorporation. *Design Compiler Version 1998.08*, Juin 1998.

[WDS+97] F. Wajsbürt, J. Desbarbieux, C. Spasevski, S. Penain, and A. Greiner. An integrated pci component for ieee 1355. In *European Multimedia, Microprocessor Systems and Electronic Commerce Conference and Exhibition (EMMSEC'97)*, Florence, Italie, Novembre 1997.

[X-T97] X-Tek Corporation. *XHDL 3.0*, Aout 1997.

[ZRP+96] B. Zerrouk, V. Reibaldi, F. Potter, A. Greiner, and A. Derieux. Rcube : A gigabit serial links low latency adaptive router. In *Records of the IEEE Hot Interconnects IV*, Palo Alto CA, USA, Aout 1996.

Publications personnelles sur l'analyse sémantique de descriptions VHDL en vue de la synthèse

R.K. Bawa, P. Guerrier, L. Jacomme, and P. Bazargan-Sabet. An approach to behavioral synthesis from a formal model of vhdl. In *VHDL International Users Forum*, pages 117–126, Durham, Caroline du Nord, USA, Octobre 1996.

L. Jacomme and R.K. Bawa. Synthèse de descriptions comportementales séquentielles en conformité avec la sémantique vhdl. In *Actes du 1er Colloque CAO de Circuits Intégrés et Systèmes*, pages 303–306, Villard de Lans , France, Janvier 1997.

L. Jacomme, F. Pétrot, and R.K. Bawa. Formal extraction of memorizing elements for sequential vhdl synthesis. In *24th Euromicro International Conference on Digital Sytems*, pages 317–320, Vasteras, Suède, Aout 1998. IEEE.

L. Jacomme, F. Pétrot, and R.K. Bawa. Formal analysis of single wait vhdl processes for semantic based synthesis. In *12th International Conference on VLSI Design*, pages 151–156, Goa, Inde, Janvier 1999. IEEE.

ANNEXE **A**

Un sous ensemble de VHDL adapté à la synthèse

Cette annexe présente un sous ensemble du langage VHDL adapté à la synthèse, et suffisamment riche pour expérimenter et valider notre méthode. Ce sous ensemble n'est soumis à aucune contrainte liée aux méthodes de reconnaissance syntaxique.

En effet dans le cadre de nos travaux nous cherchons à construire un ensemble d'affectations concurrentes à partir d'une description VHDL sans imposer de restrictions syntaxiques c'est pourquoi notre sous ensemble ne doit être soumis qu'aux véritables contraintes de synthèse.

A.1 Les types, les fonctions et les attributs prédéfinis

Les types de notre sous ensemble sont ceux admis par les outils de synthèse industriels c'est-à-dire :

- les types prédéfinis *BIT*, *BOOLEAN*, *BIT_VECTOR*, et *INTEGER* et les opérateurs associés ;
- les types, les opérateurs et les fonctions de conversion définis par les deux normes IEEE-1164 et IEEE-1076.3 ;
- les types énumérés définis par l'utilisateur.

Les attributs de notre sous ensemble sont tous ceux ayant un sens dans le cadre de la synthèse, c'est-à-dire :

- les attributs utiles pour la spécification d'une condition de mémorisation sur front : *EVENT*, *STABLE* et *LAST_VALUE* ;

- les attributs permettant de manipuler les intervalles et les bornes d'un type scalaire ou d'un vecteur.

A.2 Les instructions VHDL exploitables en synthèse

Il existe deux types d'instruction en VHDL : les instructions concurrentes et les instructions séquentielles. Il convient de définir pour chacune de ces deux catégories les restrictions que nous souhaitons imposer.

A.2.1 Les instructions concurrentes

Les instructions concurrentes permettent :

- l'instanciation d'un composant ;
- la définition de directives génériques ;
- l'affectation conditionnelle ou sélective d'un signal ;
- l'appel à une procédure concurrente ;
- la définition d'un processus ;
- le regroupement d'un bloc d'instructions concurrentes ;

Les instructions concurrentes trouvent assez facilement une implantation matérielle combinatoire à l'exception des blocs. En effet il est possible de définir une condition de garde à l'intérieur d'un bloc qui peut servir à décrire le comportement d'un registre ou d'une barrière trois états. Du point de vue d'un outil de synthèse ceci impose la mise en oeuvre d'un ensemble de motifs spécifiques, afin de pouvoir reconnaître syntaxiquement les éléments matériels équivalents.

La prise en compte de la totalité ou d'une grande partie de cet ensemble d'instructions est cruciale pour un outil de synthèse industriel. En effet c'est un critère de qualité indéniable que d'offrir une grande souplesse et un grand confort de description aux concepteurs de circuits.

Dans le cadre de nos travaux les enjeux ne sont pas les mêmes ; il ne s'agit pas de développer un compilateur industriel, mais de trouver un sous ensemble VHDL suffisamment élaboré pour rendre compte des problèmes posés par la reconnaissance sémantique des éléments matériels.

Aussi on élimine toutes les instructions qui n'entrent pas en jeu dans une méthode de reconnaissance syntaxique, c'est à dire les instructions d'instanciation de composants et les directives génériques.

A.2.2 Les instructions séquentielles

Les instructions séquentielles sont utilisées pour décrire le comportement du corps des processus, des fonctions ou des procédures VHDL. Ces instructions peuvent être de différents types :

- les affectations de signaux et de variables ;
- les instructions conditionnelles ;
- les instructions de gestion de boucles ;
- l'instruction de suspension d'un processus.
- l'instruction vide.

L'ensemble de ces instructions séquentielles est accepté par les outils de synthèse industriel à l'exception parfois de certaines instructions de boucles. Le problème étant de pouvoir faire correspondre à l'exécution d'une boucle séquentielle un ensemble d'équations concurrentes implantables en matériel.

Ces instructions ont une importance prédominantes puisque c'est sur elles que s'appuies les méthodes d'identification des éléments matériels dans une description VHDL. Notre sous ensemble se doit donc de prendre en compte la totalité de ces instructions avec cependant les restrictions suivantes admises dans le cadre de la synthèse :

- les affectations de signaux et de variables ne peuvent pas être suivies de la clause *AFTER* pour stipuler un temps de propagation ;
- l'instruction de suspension *WAIT* ne peut pas être succédée de la clause *FOR* pour préciser un temps d'arrêt.

A.3 La grammaire de notre sous ensemble VHDL

Nous présentons dans les pages suivantes, la grammaire complète de notre sous ensemble VHDL dans un format similaire au *Backus-Naur Form*.

```
design_file
  : design_unit
    ..design_unit..
  ;
..design_unit..
  :
  | design_unit
    ..design_unit..
  ;
design_unit
  : context_clause
    library_unit
  ;
library_unit
  : primary_unit
  | secondary_unit
  | error
  ;
primary_unit
  : entity_declaration
  ;
secondary_unit
  : architecture_body
  ;
library_clause
  : LIBRARY
    logical_name_list
    Semicolon_ERR
  ;
logical_name_list
  : Identifier
    ...logical_name..
  ;
...logical_name..
  :
  | ...logical_name..
    Comma
    Identifier
  ;
context_clause
  : ..context_item..
  ;
..context_item..
  :
  | ..context_item..
    context_item
  ;
context_item
  : library_clause
  | use_clause
  ;
use_clause
  : USE
    selected_name
    ...selected_name..
    Semicolon_ERR
  ;
attribute_declaration
  : ATTRIBUTE
    Identifier
    Colon
    simple_name
    Semicolon_ERR
  ;
attribute_specification
  : ATTRIBUTE
    simple_name
    OF
    simple_name
    Colon
    entity_class
    IS
    StringLit
    Semicolon_ERR
  ;
entity_class
  : ENTITY
  | ARCHITECTURE
  | FUNCTION
  | PROCEDURE
  | SUBTYPE
  | CONSTANT
  | VARIABLE
  | SIGNAL
  | LABEL
  | TYPE
  ;
selected_name
  : prefix
    Dot
    suffix
  ;
suffix
  : simple_name
  | ALL
  ;
prefix
  : simple_name
  | prefix
    Dot
    simple_name
  ;
entity_declaration
 : ENTITY
```

```
   simple_name
   IS
   .port_clause.
   END_ERR
   .simple_name.
   Semicolon_ERR
 | ENTITY
   error
  ;
.constraint.
 :
  | constraint
  ;
constraint
 : range_constraint
  |
   LeftParen
   range
   RightParen_ERR
  ;
range_constraint
 : RANGE
   range
  ;
range
 : attribute_name
 | simple_expression
   direction
   simple_expression
  ;
direction
 :  TO
  | DOWNTO
  ;
.port_clause.
 :
 | port_clause
  ;
port_clause
 : PORT
   LeftParen
   formal_port_list
   RightParen_ERR
 | PORT
   error
   Semicolon_ERR
  ;
formal_port_list
 : formal_port_element
   ...formal_port_element..
  ;
...formal_port_element..
 :
 | ...formal_port_element..
   Semicolon_ERR
   formal_port_element
  ;
formal_port_element
 : .SIGNAL.
   identifier_list
   Colon
   .mode.
   type_mark
   .constraint.
   .signal_kind.
   .VarAsgn__expression.
 | error
  ;
.SIGNAL.
 :
  ;
.mode.
  :
  | IN
  | OUT
  | INOUT
  ;
architecture_body
 : ARCHITECTURE
   Identifier
   OF
   simple_name
   IS
   architecture_declarative_part
   BEGIN
   architecture_statement_part
   END_ERR
   .simple_name.
   Semicolon_ERR
 | ARCHITECTURE
   error
  ;
architecture_declarative_part
 : ..block_declarative_item..
  ;
..block_declarative_item..
 :
 | ..block_declarative_item..
   block_declarative_item
  ;
block_declarative_item
 : signal_declaration
 | variable_declaration
 | type_declaration
```

```
    | subtype_declaration
    | attribute_declaration
    | attribute_specification
    | constant_declaration
    | subprogram_declaration
    | subprogram_body
    | error
      Semicolon_ERR
    ;
designator
  : Identifier
  ;
.procedure_parameter_list.
  :
  | LeftParen
    procedure_parameter_element
    ...procedure_parameter_element..
    RightParen_ERR
  | LeftParen
    error
    RightParen_ERR
  ;
...procedure_parameter_element..
  :
  | ...procedure_parameter_element..
    Semicolon_ERR
    procedure_parameter_element
  ;
procedure_parameter_element
  : .procedure_parameter_object_class.
    identifier_list
    Colon
    .procedure_parameter_mode.
    type_mark
    .constraint.
  ;
.procedure_parameter_object_class.
  :
  | VARIABLE
  | CONSTANT
  | SIGNAL
  ;
.procedure_parameter_mode.
  :
  | IN
  | OUT
  | INOUT
  ;
.function_parameter_list.
  :
  | LeftParen
    function_parameter_element
    ...function_parameter_element..
    RightParen_ERR
  | LeftParen
    error
    RightParen_ERR
  ;
...function_parameter_element..
  :
  | ...function_parameter_element..
    Semicolon_ERR
    function_parameter_element
  ;
function_parameter_element
  : .function_parameter_object_class.
    identifier_list
    Colon
    .function_parameter_mode.
    type_mark
    .constraint.
  ;
.function_parameter_object_class.
  :
  | CONSTANT
  | SIGNAL
  ;
.function_parameter_mode.
  :
  | IN
  ;
subprogram_declarative_part
  :
  |
    subprogram_declarative_part
    subprogram_declarative_item
  ;
subprogram_declarative_item
  : constant_declaration
  | variable_declaration
  | type_declaration
  | subtype_declaration
  | attribute_declaration
  | attribute_specification
  ;
subprogram_declaration
  : subprogram_specification
    Semicolon_ERR
  ;
subprogram_specification
  : PROCEDURE
    designator
    .procedure_parameter_list.
  | FUNCTION
```

```
    designator
    .function_parameter_list.
    RETURN
    type_mark
  ;
subprogram_body
  : subprogram_specification
    IS
    subprogram_declarative_part
    BEGIN
    sequence_of_statements
    END_ERR
    .designator.
    Semicolon_ERR
  ;
.designator.
  :
  | designator
  ;
constant_declaration
 : CONSTANT
   Identifier
   Colon
   type_mark
   .constraint.
   constant_VarAsgn__expression
   Semicolon_ERR
  ;
constant_VarAsgn__expression
 : VarAsgn
   expression
  ;
.VarAsgn__expression.
  :
  | VarAsgn
    expression
  ;
variable_declaration
  : VARIABLE
    identifier_list
    Colon
    type_mark
    .constraint.
    .VarAsgn__expression.
    Semicolon_ERR
  ;
signal_declaration
 : SIGNAL
   identifier_list
   Colon
   type_mark
   .constraint.
```

```
   .signal_kind.
   .VarAsgn__expression.
   Semicolon_ERR
  ;
type_declaration
  : full_type_declaration
  ;
full_type_declaration
  : TYPE
    Identifier
    IS
    type_definition
    Semicolon_ERR
  ;
type_definition
  : scalar_type_definition
  | composite_type_definition
  | error Semicolon_ERR
  ;
subtype_declaration
  : SUBTYPE
    Identifier
    IS
    subtype_indication
    Semicolon_ERR
  ;
subtype_indication
 : type_or_function_name
   type_mark
   .constraint.
 | type_mark
   .constraint.
  ;
type_or_function_name
  : simple_name
  ;
scalar_type_definition
  : enumeration_type_definition
  ;
enumeration_type_definition
  : LeftParen
    enumeration_literal
    ...enumeration_literal..
    RightParen_ERR
  ;
...enumeration_literal..
  :
  | ...enumeration_literal..
    Comma
    enumeration_literal
  ;
enumeration_literal
```

```
  : Identifier
  | CharacterLit
  ;
composite_type_definition
  : array_type_definition
  ;
array_type_definition
  : unconstrained_array_definition
  | constrained_array_definition
  ;
unconstrained_array_definition
  : ARRAY
    LeftParen
    index_subtype_definition
    RightParen_ERR
    OF
    subtype_indication
  ;
constrained_array_definition
  : ARRAY
    index_constraint
    OF
    subtype_indication
  ;
index_subtype_definition
  : name
    RANGE
    Box
  ;
index_constraint
  : LeftParen
    discrete_range
    RightParen_ERR
  ;
discrete_range
  : range
  | name
    range_constraint
  | type_mark
  ;
type_mark
    :
     simple_name
  ;
.signal_kind.
 :
 | BUS
 | REGISTER
  ;
architecture_statement_part
 : ..concurrent_statement..
  ;
..concurrent_statement..
 :
 | ..concurrent_statement..
   concurrent_statement
  ;
concurrent_statement
 : block_statement
 | process_statement
 | concurrent_signal_assignment_statement
 | error
   Semicolon_ERR
  ;
block_statement
  : a_label
    BLOCK
    .guard_expression.
    BEGIN
    set_of_statements
    END_ERR
    BLOCK
    .label.
    Semicolon_ERR
  ;
set_of_statements
  : ..concurrent_statement..
  ;
concurrent_signal_assignment_statement
  : a_label
    unl_conditional_signal_assignment
  | unl_conditional_signal_assignment
  | a_label
    unl_selected_signal_assignment
  | unl_selected_signal_assignment
  ;
unl_conditional_signal_assignment
  : target
    LESym
    .GUARDED.
    ..waveform__WHEN__condition__ELSE..
    waveform
    Semicolon_ERR
  ;
..waveform__WHEN__condition__ELSE..
  :
  | ..waveform__WHEN__condition__ELSE..
    waveform
    WHEN
    expression
    ELSE
  ;
unl_selected_signal_assignment
  : WITH
```

```
    expression
    SELECT
    target
    LESym
    .GUARDED.
    waveform__WHEN__choices
    ...waveform__WHEN__choices..
    Semicolon_ERR
  ;
...waveform__WHEN__choices..
  :
  | ...waveform__WHEN__choices..
    Comma
    waveform__WHEN__choices
  ;
waveform__WHEN__choices
  : waveform
    WHEN
    choices
  ;
process_statement
 : a_label
   unl_process_statement
   unl_process_statement
  ;
unl_process_statement
 : PROCESS
   .sensitivity_list.
 process_declarative_part
 BEGIN
 sequence_of_statements
   END_ERR
   PROCESS
   .label.
   Semicolon_ERR
  ;
process_declarative_part
  : ..process_declarative_item..
  ;
..process_declarative_item..
  :
  | ..process_declarative_item..
    process_declarative_item
  ;
process_declarative_item
  : constant_declaration
  | variable_declaration
  | type_declaration
  | subtype_declaration
  | attribute_declaration
  | attribute_specification
  | subprogram_declaration
  | subprogram_body
  ;
.sensitivity_list.
  :
  | sensitivity_list
  ;
sensitivity_list
 : LeftParen
   signal_list
  ;
signal_list
  : name
    ...name..
  ;
...name..
  :
  | Comma
    name
    ...name..
  ;
sequence_of_statements
 :
 | sequence_of_statements
   sequential_statement
  ;
sequential_statement
 :  variable_assignment_statement
  | signal_assignment_statement
  | if_statement
  | case_statement
  | loop_statement
  | next_statement
  | exit_statement
  | return_statement
  | null_statement
  | procedure_call_statement
  | wait_statement
  ;
loop_statement
  : a_label
    unl_loop_statement
  |

    unl_loop_statement
  ;
unl_loop_statement
  : .iteration_scheme.
    LOOP
    sequence_of_statements
    END_ERR
    LOOP
    .label.
    Semicolon_ERR
```

```
  ;
.iteration_scheme.
  :
  | iteration_scheme
  ;
iteration_scheme
  : WHILE
    expression
  | FOR
    loop_parameter_specification
  ;
loop_parameter_specification
  : Identifier
    IN
    discrete_range
  ;
wait_statement
 : WAIT
   .sensitivity_clause.
   .condition_clause.
   Semicolon_ERR
  ;
.sensitivity_clause.
 :
 | sensitivity_clause
  ;
sensitivity_clause
 : ON
   signal_list
  ;
.condition_clause.
 :
 | condition_clause
  ;
condition_clause
 : UNTIL
   expression
  ;
next_statement
 : NEXT
   .label.
   .WHEN__condition.
   Semicolon_ERR
  ;
exit_statement
 : EXIT
   .label.
   .WHEN__condition.
   Semicolon_ERR
  ;
.WHEN__condition.
  :
  | WHEN
    condition
  ;
return_statement
  : RETURN
    .expression.
    Semicolon_ERR
  ;
.expression.
  :
  | expression
  ;
variable_assignment_statement
  : target
    VarAsgn
    expression
    Semicolon_ERR
  ;
procedure_call_statement
  : name
    Semicolon_ERR
  ;
signal_assignment_statement
 : target
   LESym
   waveform
   Semicolon_ERR
  ;
waveform_element
 : expression
  ;
if_statement
 : IF
   condition
   THEN
   sequence_of_statements
   ..ELSIF__THEN__seq_of_stmts..
   .ELSE__seq_of_stmts.
   END_ERR
   IF
   Semicolon_ERR
  ;
..ELSIF__THEN__seq_of_stmts..
 :
 | ..ELSIF__THEN__seq_of_stmts..
   ELSIF
   condition
   THEN
   sequence_of_statements
  ;
.ELSE__seq_of_stmts.
 :
```

```
 | ELSE
   sequence_of_statements
  ;
case_statement
 : CASE
   expression
   IS
   case_statement_alternative
   ..case_statement_alternative..
   END_ERR
   CASE
   Semicolon_ERR
  ;
..case_statement_alternative..
 :
 | ..case_statement_alternative..
   case_statement_alternative
  ;
case_statement_alternative
 : WHEN
   choices
   Arrow
   sequence_of_statements
  ;
waveform
 : waveform_element
  ;
choices
 : choice
   ..Bar__choice..
  ;
null_statement
  : NULL
  ;
..Bar__choice..
 :
 | ..Bar__choice..
   Bar
   choice
  ;
choice
 : literal
 | OTHERS
 | name
  ;
condition
  : expression
  ;
expression
 : relation..AND__relation..
 | relation..OR__relation..
 | relation.NAND_NOR_NXOR_relation.
 | relation..XOR__relation..
  ;
relation..AND__relation..
 : relation
   AND
   relation
 | relation..AND__relation..
   AND
   relation
  ;
relation..OR__relation..
 : relation
   OR
   relation
 | relation..OR__relation..
   OR
   relation
  ;
relation.NAND_NOR_NXOR_relation.
 : relation
 | relation
   NAND
   relation
 | relation
   NOR
   relation
 | relation
   NXOR
   relation
 | relation
   IFT
   relation
  ;
relation..XOR__relation..
 : relation
   XOR
   relation
 | relation..XOR__relation..
   XOR
   relation
  ;
relation
 : simple_expression
 | simple_expression
   relational_operator
   simple_expression
  ;
simple_expression
 : .sign.term..add_op__term..
  ;
.sign.term..add_op__term..
  : term
```

```
    | sign
      term
    | .sign.term..add_op__term..
      adding_operator
      term
    ;
adding_operator
    : Plus
    | Minus
    | Ampersand
    ;
sign
    : Plus
    | Minus
    ;
multiplying_operator
    : Star
    | Slash
    | MOD
    | REM
    ;
term
  : factor
  | term
    multiplying_operator
    factor
    ;
factor
  : primary
  | NOT
    primary
    ;
primary
  : literal
    | aggregate
    | name
    ;
relational_operator
  : EQSym
  | NESym
  | GESym
  | GTSym
  | LESym
  | LTSym
    ;
literal
  : AbstractLit
  | CharacterLit
  | StringLit
  | BitStringLit
    ;
aggregate
    : LeftParen
      element_association
      ...element_association..
      RightParen_ERR
    ;
...element_association..
    :
    | ...element_association..
      Comma
      element_association
    ;
element_association
    : expression
    | OTHERS
      Arrow
      expression
    ;
name
  : simple_name
  | indexed_name
  | slice_name
  | attribute_name
    ;
indexed_name
  :  simple_name
     aggregate
    ;
slice_name
  : simple_name
    LeftParen
    simple_expression
    direction
    simple_expression
    RightParen_ERR
    ;
attribute_name
    : name
      Apostrophe
      attribute_designator
    ;
attribute_designator
    : STABLE
    | EVENT
    | LEFT
    | RIGHT
    | LOW
    | HIGH
    | LENGTH
    | RANGE
    | REV_RANGE
    ;
identifier_list
```

```
 : Identifier
   ...identifier..
  ;
...identifier..
 :
 | ...identifier..
   Comma
   Identifier
 ;
.label.
 :
 | label
 ;
.guard_expression.
  :
  | guard_expression
  ;
guard_expression
  : LeftParen
    expression
    RightParen_ERR
  ;
.GUARDED.
  :
  | GUARDED
  ;
.simple_name.
 :
 | simple_name
  ;
simple_name
 : Identifier
  ;
target
 : name
 | aggregate
  ;
a_label
 : label
   Colon
  ;
label
 : Identifier
RightParen_ERR
 : RightParen
Semicolon_ERR
 : Semicolon
END_ERR
 : END
```

ANNEXE **B**

Les graphes de décision binaire réduits et ordonnés

Dans cette annexe, nous présentons tout d'abord de façon générale les graphes de décision binaire. Nous détaillons ensuite l'implantation des opérateurs de quantification et de restriction qui ont été évoqués dans le chapitre 6.

Les graphes de décision binaire a été initialement définis par S.B. Akers [Ake78]. Plus tard Randall Bryant en a donné une représentation canonique [Bry86] ainsi qu'un ensemble de techniques de construction et de manipulation [BRB90].

B.1 L'arbre de Shannon

La décomposition de Shannon permet d'écrire une fonction booléenne f de la façon suivante :

- $f(x_1, x_2, \ldots, x_n) = (x_1 \wedge f^+_{x_1}(x_2, \ldots, x_n)) \vee (\overline{x_1} \wedge f^-_{x_1}(x_2, \ldots, x_n))$,
  avec $f^+_{x_1}(x_2, \ldots, x_n) = f(1, x_2, \ldots, x_n)$ et $f^-_{x_1}(x_2, \ldots, x_n) = f(0, x_2, \ldots, x_n)$.

Un arbre de Shannon, appelé aussi un arbre de décision binaire, est une arborescence relative à la décomposition d'une fonction booléenne par la formule de Shannon, selon un ordre fixé des variables $\{x_1, \ldots, x_n\}$ [Mor82].

Les feuilles de l'arbre de Shannon représentent une des deux valeurs booléennes $\{0, 1\}$. Les nœuds non terminaux sont associés à une des variables x_i. Chaque nœud $node$, associé à la variable x_i représente une fonction booléenne f_{node}. Un nœud possède un fils droit et un fils gauche représentant respectivement la fonction $f_{node}(x_i = 1)$ et $f_{node}(x_i = 0)$.

La figure B.1 présente l'exemple d'un arbre de Shannon pour la fonction :

- $f = (a \wedge b) \vee (\overline{a} \wedge c) \vee (\overline{b} \wedge c)$, avec l'ordre des variables a, b, c.

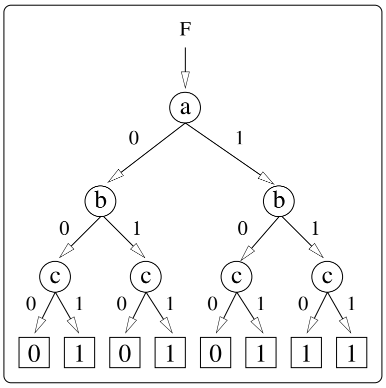

Figure B.1: Un exemple d'arbre de Shannon.

Un arbre de décision binaire est une représentation canonique d'une fonction booléenne, pour un ordonnancement donné des variables. Ainsi pour prouver l'équivalence de deux fonctions booléennes f_1 et f_2, il suffit simplement de construire les arbres de f_1 et de f_2 et de vérifier leur isomorphisme.

Le nombre de nœuds d'un arbre de Shannon évolue de façon exponentielle en fonction du nombre de variables. L'arbre de Shannon peut cependant être réduit pour donner un arbre minimal, en éliminant les nœuds redondants. Un nœud $node$ associé à une variable x_i est redondant si l'arbre droit représentant la fonction $f_{node}(x_i = 1)$ et l'arbre gauche représentant la fonction $f_{node}(x_i = 0)$ sont identiques.

La figure B.2 donne la version réduite de l'arbre présenté sur la figure B.1.

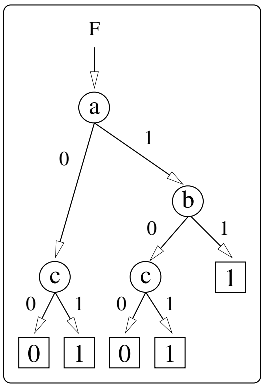

Figure B.2: Un exemple d'arbre de Shannon réduit.

B.2 Le graphe de décision binaire

On peut réduire encore le nombre de nœuds en transformant un arbre de Shannon réduit en un graphe de décision binaire. Ce graphe est obtenu en regroupant les sous-arbres isomorphes.

La figure B.3 donne le graphe de décision binaire correspondant à l'arbre de Shannon réduit présenté sur la figure B.2.

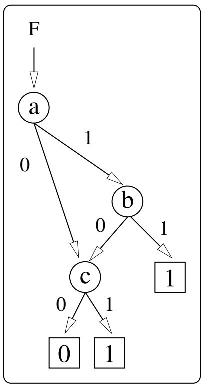

Figure B.3: Un exemple de graphe de décision binaire réduit et ordonné.

Pour représenter un ensemble de fonctions booléennes f_i on utilise la notion de multi-graphe de décision binaire réduit et ordonné.

La figure B.4 donne le multi-graphe de décision binaire représentant les fonctions booléennes suivantes : $f_1 = a \oplus b$, $f_2 = \overline{b}$, $f_3 = \overline{a \oplus b} \vee c$, $f_4 = b \vee c$.

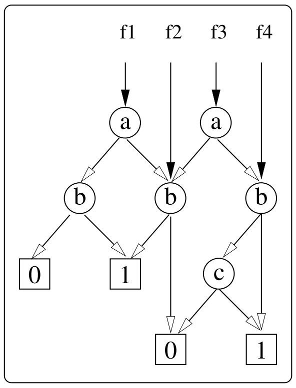

Figure B.4: Un exemple de multi-graphe de décision binaire réduit et ordonné.

B.3 La quantification existentielle et universelle

La structure de données définie sur la figure B.5 permet de représenter un nœud du multi-graphe de décision binaire.

La quantification existentielle d'une fonction f pour une variable x est définie par $(\exists x, f) = f_x^+ \vee f_x^-$. La quantification existentielle sur un ensemble $X = \{x_1, \ldots, x_n\}$ de

```
typedef struct bddnode {
  struct bddnode *HIGH;    /* noeud droit             */
  struct bddnode *LOW;     /* noeud gauche            */
  short          INDEX;    /* index de la variable    */
  short          REF_INT;  /* nombre de pères         */
  short          REF_EXT;  /* nombre de réf. externes */
} bddnode;
```

Figure B.5: La définition de la structure de données *bddnode*.

variables est définie comme l'application successive de la quantification existentielle sur chacune des n variables. On a donc $(\exists X, f) = \exists x_0, (\ldots(\exists x_n, f)\ldots)$.

La quantification universelle d'une fonction f pour une variable x est définie par : $(\forall x, f) = \overline{\exists x, \overline{f}} = f_x^+ \wedge f_x^-$, et pour un ensemble de variables on a $(\forall X, f) = \overline{\exists X,\ \overline{f}}$.

L'algorithme de quantification existentielle d'un graphe de décision binaire réduit et ordonné sur un ensemble X de variables est donné en langage C sur la figure B.6. La quantification universelle peut s'exprimer comme une quantification existentielle et elle effectuée en utilisant le même algorithme.

```
bddnode *ExistBddNode( bddnode *BddNode, bddset *X )
{
  if ( IsBddVarInSet( BddNode, X ) ) {
    return( ApplyBddNodeOr( ExistBddNode( BddNode->HIGH ),
                            ExistBddNode( BddNode->LOW  ) ) );
  }
  else {
    return( AddBddNode( ExistBddNode( BddNode->HIGH ),
                        ExistBddNode( BddNode->LOW  ) ) );
  }
}
```

Figure B.6: L'algorithme de quantification existentielle sur un *ROBDD*.

B.4 Le cofacteur généralisé

Comme nous l'avons vu dans le chapitre 6, afin d'identifier formellement les propriétés d'un registre on utilise l'expansion d'une fonction booléenne sur une base orthogonale. Soit un ensemble ϕ_i de n fonctions booléennes telles que $\sum_{i=1}^{n}\phi_i = 1$ et $\phi_i \wedge \phi_j = 0, \forall\, i \neq j$. L'expansion orthogonale de la fonction f est : $f = \sum_{i=1}^{n}(f \downarrow \phi_i) \wedge \phi_i$ où le terme $f \downarrow \phi_i$ est appelé le cofacteur généralisé de f sachant ϕ_i.

Contrairement à la décomposition de Shannon cette décomposition n'est pas unique. En effet toute fonction booléenne f_{ϕ_i} telle que $f \wedge f_{\phi_i} \subseteq f_{\phi_i} \subseteq f \vee \overline{f_{\phi_i}}$ est un cofacteur généralisé de f sachant ϕ_i.

Puisque le terme $f \downarrow \phi_i$ n'est pas unique, il faut choisir parmi toutes les solutions possibles, une fonction qui a une expression simple du point de vue de la synthèse.

Un algorithme de calcul d'un cofacteur généralisé sur un graphe de décision binaire réduit et ordonné a été initialement proposé par Olivier Coudert [Cou91]. Il est décrit en langage C sur la figure B.7. Son résultat dépend de l'ordonnancement des variables, cependant il est facile à implanter en logiciel et il donne souvent de bons résultats (c'est à dire qu'il minimise de façon significative le nombre de nœuds du graphe).

```
bddnode *CofactorBddNode( bddnode *BddNode, bddnode *BddPhi )
{
  short Index;

  if ( BddPhi == BDD_ZERO ) return( (bddnode *)0 );
  if ( BddPhi == BDD_ONE  ) return( BddNode      );

  if ( ( BddNode == BDD_ZERO ) ||
       ( BddNode == BDD_ONE  ) ) return( BddNode );

  if ( IsBddVarEqual( BddNode, BddPhi ) ) {
    Index   = BddNode->INDEX;
    BddHigh = CofactorBddNode( BddNode->HIGH, BddPhi->HIGH );
    BddLow  = CofactorBddNode( BddNode->LOW , BddPhi->LOW  );
  }
  else
  if ( IsBddVarTop( BddPhi, BddNode ) ) {
    Index   = BddPhi->INDEX;
    BddHigh = CofactorBddNode( BddNode, BddPhi->HIGH );
    BddLow  = CofactorBddNode( BddNode, BddPhi->LOW  );
  }
  else {
    Index   = BddNode->INDEX;
    BddHigh = CofactorBddNode( BddNode->HIGH, BddPhi );
    BddLow  = CofactorBddNode( BddNode->LOW , BddPhi );
  }

  if ( BddHigh == (bddnode *)0 ) BddNode = BddLow;
  else
  if ( BddLow  == (bddnode *)0 ) BddNode = BddHigh;
  else
  BddNode = AddBddNode( Index, BddHigh, BddLow );

  return( BddNode );
}
```

Figure B.7: L'algorithme de calcul d'un cofacteur généralisé sur un *ROBDD*.

ANNEXE C

L'application de la réduction locale sur le graphe de contrôle d'un processus

La réduction locale réduit chaque sous graphe correspondant à une instruction VHDL. Elle est appliquée une fois que les affectations de variables et de signaux présentes sur le graphe de contrôle du processus ont été modifiées.

Comme nous l'avons vu au chapitre 4, lors de la compilation d'une description VHDL, chaque instruction séquentielle d'un processus est traduite sur notre modèle sous la forme d'un sous réseau de Pétri.

La réduction locale exploite directement la structure de l'instruction et il existe pour chaque type d'instruction une technique particulière de réduction locale.

C.1 Les instructions d'affectation

La figure C.1 présente le cas des instructions d'affectation. Une instruction d'affectation est constituée d'une unique place p_{Assign} et d'une unique transition t_{Assign}. Cette forme est déjà minimale et la réduction locale essaye simplement d'effectuer une réduction séquentielle entre la transition t_1 qui précède la place p_{Assign} et la transition t_{Assign}.

Lorsque cette réduction est possible, on obtient une unique transition t_{red} équivalente à l'exécution séquentielle de t_1 et de t_{Assign}.

Si la place p_1 n'a pas d'autre successeur que la nouvelle transition t_{red} on peut alors considérer que la place p_1 et la transition t_{red} constituent une nouvelle instruction d'affectation qui doit être localement réduite.

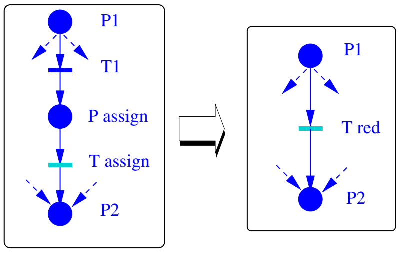

Figure C.1: La réduction locale d'une affectation.

C.2 Les instructions conditionnelles

La figure C.2 présente le traitement appliqué à une instruction *IF*. Il est effectué en trois étape :

1. tout d'abord il s'agit de réduire toutes les instructions situées sur les deux branches *THEN* et *ELSE* du *IF*. Lorsque cette réduction réussie, la transition t_{then} et toutes les instructions de la branche *THEN* sont remplacées par une unique transition t_{red1}, et la transition t_{else} et toutes les instructions de la branche *ELSE* sont remplacées par une transition t_{red2} ;
2. on peut dès lors appliquer une réduction latérale entre les deux places p_{if} et p_{endif}. Lorsque cette réduction réussie, on obtient une transition équivalente t_{red3} ;
3. on applique ensuite deux fois la réduction séquentielle sur les trois transitions t_1, t_{red3} et t_2. Lorsque ces deux réductions sont possibles, on obtient une transition t_{red4} équivalente à l'exécution de l'instruction *IF* ;

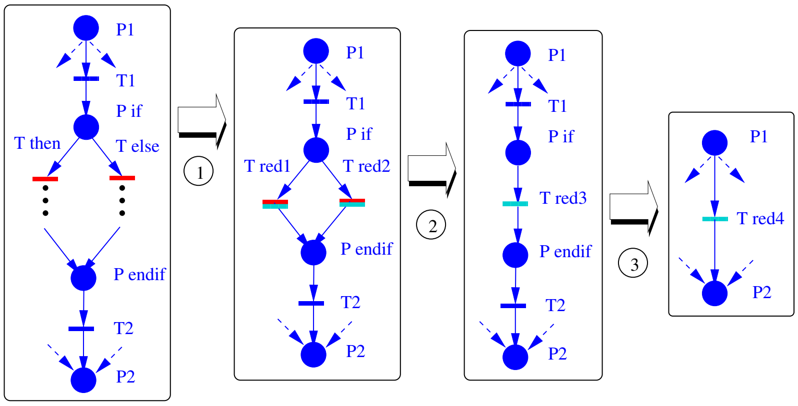

Figure C.2: La réduction locale d'une instruction *IF*.

L'instruction *CASE* est traitée de la même façon, en effectuant les deux premières étapes sur toutes les branches *WHEN*.

Si la place p_1 n'a pas d'autre successeur que la nouvelle transition t_{red4} on peut alors considérer que la place p_1 et la transition t_{red4} constituent une instruction d'affectation qui doit être localement réduite.

C.3 Les instructions de boucle

La figure C.3 présente les deux réductions locales possibles, communes aux trois instruction *LOOP*, *WHILE* et *FOR*.

On réduit tout d'abord toutes les instructions du corps de la boucle. Si la place $p_{endloop}$ a un unique prédécesseur, alors elle peut être considérée comme une instruction d'affectation et être réduite en conséquence. Si la place $p_{endloop}$ a plusieurs prédécesseurs, et si la place p_2 n'a qu'un unique successeur, on peut déplacer tous les arcs entrants de la place $p_{endloop}$ sur la place p_2. On peut alors détruire la place $p_{endloop}$ et la transition $t_{endloop}$.

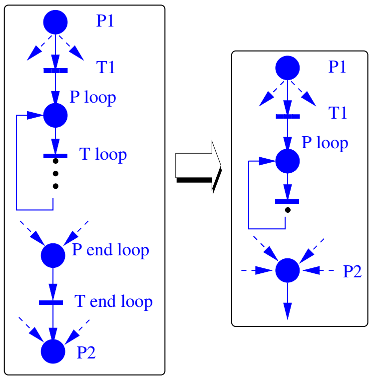

Figure C.3: La réduction locale d'une instruction *LOOP*.

La figure C.4 présente les réductions locales qui peuvent être effectuées sur l'instruction *FOR*. Les deux places p_{for} et p_{inc} représentant respectivement l'initialisation et l'incrémentation du compteur de boucle, peuvent être considérée comme des instructions d'affectation. Elles sont donc réduites en deux transitions équivalentes t_{red1} et t_{red4}.

La figure C.5 présente les réductions locales qui peuvent être effectuées sur l'instruction *WHILE*. On applique une réduction locale séquentielle sur les deux transitions t_{true} et t_{loop} afin d'obtenir une transition t_{red1} équivalente.

La figure C.6 présente la réduction locale d'une instruction *NEXT*. La seule réduction possible consiste à appliquer une réduction locale séquentielle sur les deux transition t_{false} et $t_{endnext}$.

La figure C.7 présente la réduction locale d'une instruction *EXIT*. La seule réduction possible consiste à appliquer une réduction locale séquentielle sur les deux transitions

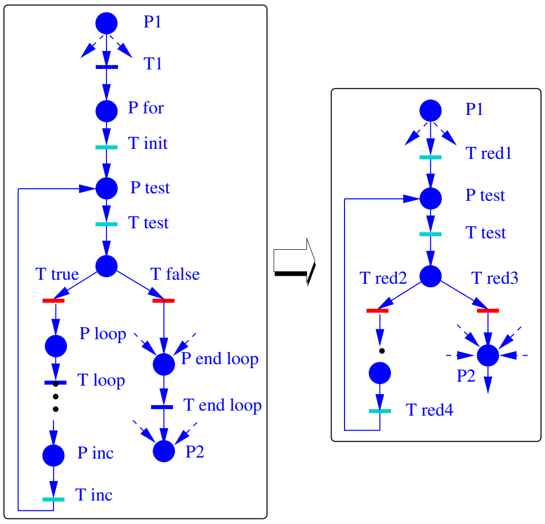

Figure C.4: La réduction locale d'une instruction *FOR*.

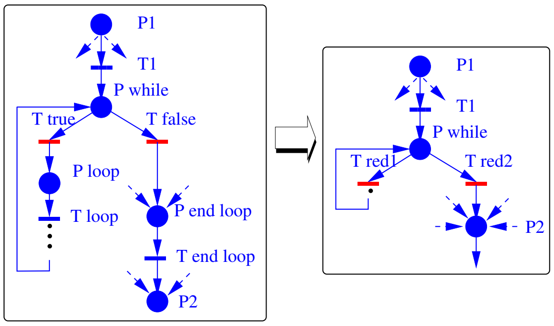

Figure C.5: La réduction locale d'une instruction *WHILE*.

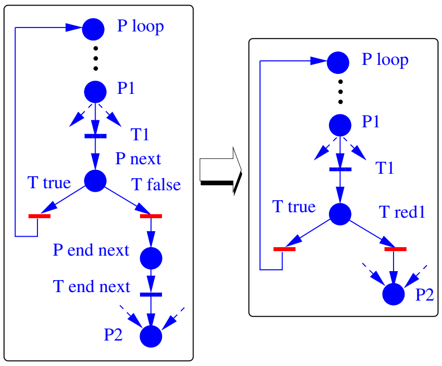

Figure C.6: La réduction locale d'une instruction *NEXT*.

t_{false} et $t_{endexit}$.

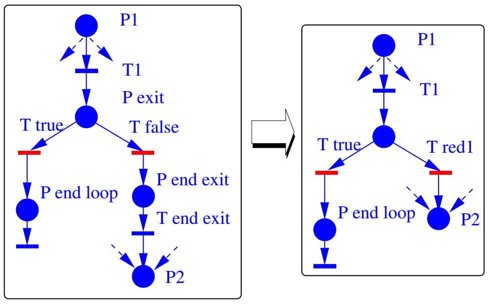

Figure C.7: La réduction locale d'une instruction *EXIT*.

C.4 Les instructions de suspension

La figure C.8 présente le cas des instructions de suspension *WAIT* et *WAIT UNTIL*.

L'instruction *WAIT* ne peut pas être réduite. Dans le cas de l'instruction *WAIT UNTIL*, la seule réduction possible consiste à appliquer une réduction locale séquentielle sur les deux transitions t_{true} et $t_{enduntil}$.

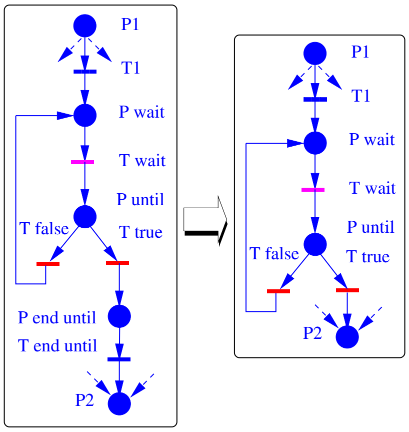

Figure C.8: La réduction locale d'une instruction *WAIT UNTIL*.

En pratique on réduit tout d'abord localement toutes les instructions du processus et on revient ensuite sur l'instruction de suspension *WAIT UNTIL* pour essayer d'effectuer une réduction latérale entre les deux transitions t_{true} et t_{false}.

ANNEXE **D**

Les structures de données

Cette annexe présente l'ensemble des structures de données que nous avons définies lors de la mise en œuvre logicielle de notre méthode. L'utilisation de ces différentes structures de données a été détaillée dans le chapitre 8.

D.1 Les expressions VHDL

Afin de représenter les expressions VHDL nous utilisons la structure de données *vexexpr* décrite sur la figure D.1.

Cette structure de données permet de manipuler des symboles au niveau de bit ou vecteurs de bits. Le champs *TYPE* contient principalement les deux informations suivantes : le type du nœud (un atome, un opérateur ou une fonction) et le type VHDL associé au nœud (*BIT*, *BIT_VECTOR*, *SIGNED* etc ...).

Les structures de données *chain_list* et *ptype_list* permettent de construire des listes d'éléments (typés ou non typés) et elles sont utilisées dans de nombreux contextes, et en particulier ici pour construire la liste des opérandes d'une fonction ou d'un opérateur.

La figure D.2 présente une expression VHDL et la structure de données *vexexpr* correspondante.

D.2 La représentation syntaxique d'une description VHDL

La structure de données *vbhhfig* assure la représentation en mémoire d'une description VHDL de niveau *RTL*. Elle est composée d'une liste des types, d'une liste des déclarations des ports et des signaux, et d'une liste des processus et des fonctions de la description.

```
typedef struct chain_list {
  struct chain_list *NEXT; /* maillon suivant            */
  void              *DATA; /* valeur associée au maillon */
} chain_list;

typedef struct ptype_list {
  struct ptype_list *NEXT; /* maillon suivant            */
  long               TYPE  /* type du maillon            */
  void              *DATA; /* valeur associée au maillon */
} ptype_list;

typedef struct vexexpr {
  chain_list *OPERAND;     /* liste des opérandes        */
  void       *VALUE;       /* valeur associée au noeud   */
  short       TYPE;        /* type du noeud              */
  short       LEFT;        /* borne gauche du vecteur    */
  short       RIGHT;       /* borne droite du vecteur    */
  short       WIDTH;       /* largeur du vecteur         */
} vexexpr;
```

Figure D.1: La définition de la structure de données *vexexpr*.

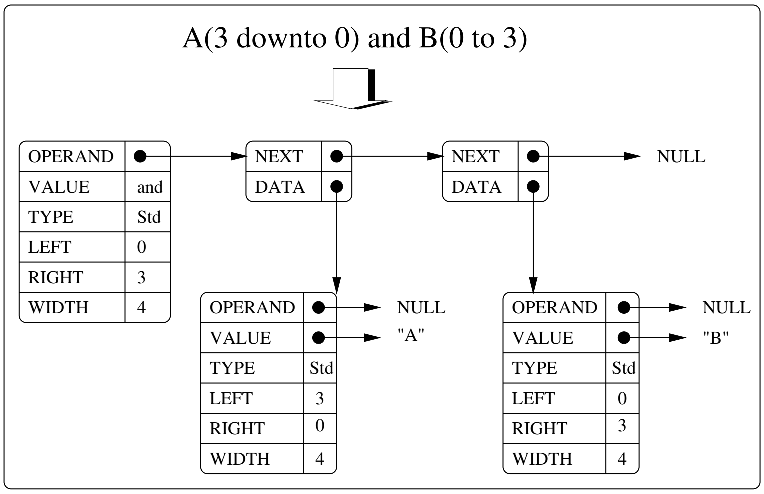

Figure D.2: La représentation d'une expression VHDL.

```
typedef struct vbhfig {
  struct vbhfig *NEXT;      /* figure suivante                  */
  char          *NAME;      /* nom de la description            */
  vbhtyp_list   *BETYP;     /* liste des types                  */
  vbhpor_list   *BEPOR;     /* déclaration des ports            */
  vbhcst_list   *BECST;     /* déclaration des constantes       */
  vbhaux_list   *BEAUX;     /* déclaration des signaux internes */
  vbhpcs_list   *BEPCS;     /* liste des processus              */
  vbhfun_list   *BEFUN;     /* liste des fonctions              */
} vbhfig_list;
```

Figure D.3: La structure de données *vbhfig*.

Les structures de données *vbhtyp*, *vbhaux*, *vbhcst* et *vbhpor* permettent de représenter respectivement les déclarations des types, des signaux, des constantes et des ports de la description. Ces différentes structures de données sont présentées sur la figure D.4.

La structure de données *vbhpcs* permet de représenter un processus VHDL, c'est à dire son éventuelle liste de sensibilité, les déclarations de ses variables locales, ainsi que toutes les instructions de son corps. La structure de données *vbhfun* permet de représenter une fonction VHDL, et elle est donc composée d'une liste des arguments (*vbharg*) et des variables locales (*vbhvar*) de la fonction, ainsi que de l'ensemble des instructions de son corps. Ces différentes structures de données sont présentées sur la figure D.5.

Les instructions du corps d'un processus ou d'une fonction sont représentées à l'aide de la structure de données *ptype_list*. Le champs *TYPE* contient le type de l'instruction (*CASE*, *IF*, etc ...), le champs *DATA* pointe sur la structure de données associée à l'instruction et le champs *NEXT* donne l'instruction suivante.

Les structures de données associées aux différentes instructions sont présentées sur les figure D.6, D.7, D.8, D.9 et D.10.

```
typedef struct vbhtyp {
  struct vbhtyp  *NEXT;    /* type suivant                       */
  char           *NAME;    /* nom du type                        */
  long            LEFT;    /* borne gauche                       */
  long            RIGHT;   /* borne droite                       */
  char          **VALUE;   /* les valeurs d'un type énuméré      */
  unsigned int    SIZE;    /* nombre des valeurs                 */
  char            CLASS;   /* classe (tableau,scalaire etc..)    */
  unsigned char   INDEX;   /* numéro d'identification du type    */
  struct vbhtyp  *BASE;    /* type de base pour les sous types   */
  struct vbhfun  *RESOLV;  /* fonction de résolution             */
} vbhtyp_list;

typedef struct vbhaux {
  struct vbhaux *NEXT;     /* déclaration suivante               */
  vexexpr       *TARGET;   /* nom du signal                      */
  vexexpr       *VEX;      /* valeur initiale                    */
  vbhtyp_list   *TYPE;     /* type du signal                     */
  unsigned char  KIND;     /* catégorie (register/bus)           */
} vbhaux_list;

typedef struct vbhcst {
  struct vbhcst  *NEXT;    /* déclaration suivante               */
  vexexpr        *TARGET;  /* nom de la constante                */
  vexexpr        *VEX;     /* valeur de la constante             */
  vbhtyp_list    *TYPE;    /* type de la constante               */
} vbhcst_list;

typedef struct vbhpor {
  struct vbhpor  *NEXT;    /* déclaration suivante               */
  vexexpr        *TARGET;  /* nom du port                        */
  vexexpr        *VEX;     /* valeur initiale                    */
  vbhtyp_list    *TYPE;    /* type du port                       */
  unsigned char   DIR;     /* direction du port (IN,OUT,INOUT)   */
  unsigned char   KIND;    /* catégorie (register/bus)           */
} vbhpor_list;
```

Figure D.4: Les structures de données représentant les déclarations.

```
typedef struct vbhvar {
  vexexpr        *TARGET;     /* nom de la variable       */
  vexexpr        *VEX;        /* valeur initiale          */
  vbhtyp_list    *TYPE;       /* type de la variable      */
} vbhvar_list;

typedef struct vbhpcs {
  struct vbhpcs *NEXT;        /* processus suivant        */
  char          *LABEL;       /* nom du processus         */
  chain_list    *SENSITIVITY; /* liste de sensibilité     */
  ptype_list    *VARIABLE;    /* liste des variables      */
  ptype_list    *INSTRUCTION; /* liste des instructions   */
} vbhpcs_list;

typedef struct vbharg {
  struct vbharg *NEXT;        /* argument suivant         */
  vexexpr       *TARGET;      /* nom de l'argument        */
  vbhtyp_list   *TYPE;        /* type de l'argument       */
  char           DIR;         /* direction de l'argument  */
  char           CLASS;       /* constant/signal/variable */
} vbharg_list;

typedef struct vbhfun {
  struct vbhfun *NEXT;        /* fonction suivante        */
  char          *NAME;        /* nom de la fonction       */
  struct vbharg *ARGUMENT;    /* liste des arguments      */
  struct vbharg *RETURN;      /* type/valeur retournée    */
  ptype_list    *VARIABLE;    /* liste des variables      */
  ptype_list    *INSTRUCTION; /* liste des instructions   */
} vbhfun_list;
```

Figure D.5: La représentation des processus et des fonctions.

```
typedef struct vbhasg {
  vexexpr       *TARGET;  /* nom du signal affecté       */
  vexexpr       *VEX;     /* expression affectée         */
  vbhtyp_list   *TYPE;    /* type du signal              */
} vbhasg_list;

typedef struct vbhvar {
  vexexpr       *TARGET;  /* nom de la variable affectée */
  vexexpr       *VEX;     /* expression affectée         */
  vbhtyp_list   *TYPE;    /* type de la variable         */
} vbhvar_list;

typedef struct vbhagr {
  ptype_list   *ASSIGN;   /* affectations de l'agrégat   */
} vbhagr_list;
```

Figure D.6: La représentation des instructions d'affectation.

```
typedef struct vbhifs {
  vexexpr      *CND;       /* expression de la condition  */
  ptype_list   *CNDTRUE;   /* liste des instructions THEN */
  ptype_list   *CNDFALSE;  /* liste des instructions ELSE */
} vbhifs_list;

typedef struct vbhcas {
  vexexpr      *VEX;       /* expression du CASE          */
  vbhtyp_list  *TYPE;      /* type de l'expression        */
  vbhcho_list  *CHOICE;    /* table des choix WHEN        */
  unsigned int  SIZE;      /* taille de la table          */
} vbhcas_list;

typedef struct vbhcho {
  char       *VALUE;       /* valeur du choix WHEN        */
  ptype_list *INSTRUCTION; /* liste des instructions      */
} vbhcho_list;
```

Figure D.7: Les instructions conditionnelles.

```
typedef struct vbhwhi {
  char        *LABEL;       /* nom de la boucle WHILE     */
  vexexpr     *CND;         /* expression de la condition */
  ptype_list *INSTRUCTION; /* liste des instructions     */
} vbhwhi_list;

typedef struct vbhlop {
  char        *LABEL;       /* nom de la boucle LOOP      */
  ptype_list *INSTRUCTION; /* liste des instructions     */
} vbhlop_list;

typedef struct vbhfor {
  char        *LABEL;       /* nom de la boucle FOR       */
  vexexpr     *VARIABLE;    /* nom du compteur de boucle  */
  vexexpr     *LEFT;        /* borne gauche du FOR        */
  vexexpr     *RIGHT;       /* borne droite du FOR        */
  char         UP;          /* sens de parcourt           */
  ptype_list *INSTRUCTION; /* liste des instructions     */
} vbhfor_list;

typedef struct vbhext {
  char        *LABEL;       /* nom de la boucle pour EXIT */
  vexexpr     *CND;         /* expression de la condition */
} vbhext_list;

typedef struct vbhnxt {
  char        *LABEL;       /* nom de la boucle pour NEXT */
  vexexpr     *CND;         /* expression de la condition */
} vbhnxt_list;
```

Figure D.8: La représentation des instructions de boucle.

```
typedef struct vbhwas {
  chain_list *SENSITIVITY; /* liste de sensibilité */
  vexexpr    *CND;         /* expression du UNTIL  */
} vbhwas_list;
```

Figure D.9: La représentation d'un point de suspension.

```
typedef struct vbhret {
  vexexpr     *RET;        /* expression retournée */
} vbhret_list;

typedef struct vbhcal {
  vexexpr     *CALL;       /* appel d'un procédure */
} vbhcal_list;
```

Figure D.10: Les instructions d'appel et de retour de fonction.

D.3 Le modèle *XVPN*

La structure de données *xvpnfig* représente le modèle *XVPN* qui a été défini dans le chapitre 4. Elle est essentiellement composée d'une liste de symboles, d'une liste de processus et d'une liste de fonctions.

```
typedef struct xvpnfig {
  struct xvpnfig *NEXT;          /* figure suivante              */
  char           *NAME;          /* nom de la figure             */
  xvpnproc_list  *PROCESS;       /* liste des processus          */
  xvpnfunc_list  *FUNCTION;      /* liste des fonctions          */
  xvpndecl_list  *DECLAR         /* liste des décl. triées       */
  [ VPN_MAX_DECLAR_TYPE ];       /* par type (port/signal/etc)   */
  authtable      *HASH_DECLAR    /* tables de hachage des décl. */
  [ VPN_MAX_DECLAR_TYPE ];       /* triées par type              */
  authtable      *HASH_PROCESS;  /* table de hachage des proc.  */
  authtable      *HASH_FUNCTION; /* table de hachage des fonct. */
  authtable      *HASH_TRANS;    /* table de hachage des trans. */
  authtable      *HASH_PLACE;    /* table de hachage des places */
  auth2table     *HASH_ARC;      /* table de hachage des arcs   */
} xvpnfig_list;
```

Figure D.11: La structure de données *xvpnfig*.

Les déclarations de symboles sont représentées à l'aide de la structure de données *xvpndecl*. Afin de manipuler seulement certains bits d'un vecteur, les symboles sont également représentés au niveau bit à l'aide de la structure *xvpnsym*. La figure D.12 donne la définition de ces deux structures de données.

```
typedef struct xvpnsym {
  char            *NAME;      /* nom du symbole               */
  struct xvpndecl *DECL;      /* déclaration du symbole       */
  short            INDEX;     /* numéro du bit du symbole     */
} vpnsym;

typedef struct xvpndecl {
  struct xvpndecl  *NEXT;     /* déclaration suivante         */
  struct xvpndecl **PREV;     /* déclaration précédente       */
  vexexpr          *VEX_ATOM; /* nom du symbole               */
  vexexpr          *VEX_INIT; /* valeur initiale              */
  xvpnsym          *DECL_SYM; /* tableau des bits du symbole */
  unsigned char     DIR;      /* direction IN/OUT/INOUT       */
  unsigned char     TYPE;     /* port,signal,argument etc     */
  unsigned char     KIND;     /* catégorie : bus/register     */
  unsigned char     CLASS;    /* argument CONSTANT etc        */
  unsigned char     BASE;     /* bit, bit_vector,etc          */
```

Figure D.12: La représentation des symboles de la description.

Les processus et les fonctions sont constitués d'une liste de places et de transitions représentant leur graphe de contrôle. Ils possèdent également la liste de leurs variables locales ainsi qu'un pointeur sur la première place de leur graphe de contrôle (la place *BEGIN*).

Les fonctions possèdent la liste de leurs arguments et une expression *VEX_RET* représentant le type de la valeur qu'elles retournent. Ces deux structures de données sont présentées sur la figure D.13.

```
typedef struct xvpnproc {
  struct xvpnproc  *NEXT;      /* processus suivant   */
  struct xvpnproc **PREV;      /* processus précédent */
  char             *NAME;      /* nom du processus    */
  xvpnplace_list   *FIRST;     /* place BEGIN         */
  xvpnplace_list   *SIMUL;     /* place marquée       */
  xvpnplace_list   *PLACE;     /* liste des places    */
  xvpntrans_list   *TRANS;     /* liste des trans     */
  chain_list       *VAR_DECL;  /* liste des variables */
} xvpnproc_list;

typedef struct xvpnfunc {
  struct xvpnfunc  *NEXT;      /* fonction suivante   */
  struct xvpnfunc **PREV;      /* fonction précédente */
  char             *NAME;      /* nom de la fonction  */
  xvpnplace_list   *FIRST;     /* place BEGIN         */
  xvpnplace_list   *SIMUL;     /* place marquée       */
  xvpnplace_list   *PLACE;     /* liste des places    */
  xvpntrans_list   *TRANS;     /* liste des trans     */
  vexexpr          *VEX_RET;   /* expression retour   */
  chain_list       *ARG_DECL;  /* liste des arguments */
  chain_list      **LAST_ARG;  /* dernier argument    */
  chain_list       *VAR_DECL;  /* liste des variables */
} xvpnfunc_list;
```

Figure D.13: La représentation des processus et des fonctions.

Les places, les transitions et les arcs du graphe de contrôle d'un processus ou d'une fonction sont représentés respectivement par les structures de données *xvpnplace*, *xvpntrans* et *xvpnarc*. Ces structures de données sont présentées sur la figure D.14.

Les deux structures de données *xvpnplace* et *xvpntrans* contiennent la liste des arcs entrants et la liste des arcs sortants.

Une place *xvpnplace* connaît le type de l'instruction qu'elle représente à savoir un *IF*, un *CASE* etc ... Le champs *LINK* permet de connaître la place qui correspond à la fin d'une instruction. Ainsi pour une place de type *IF* le champs *LINK* pointera sur la place *END IF* correspondante. Cette information est essentielle pour pouvoir appliquer la méthode de réduction locale exposée dans la section 5.2.

Une transition lorsqu'elle est de type *EXECUTE* possède une condition de franchissement contenue dans le champs *VEX_GUARD* et une liste séquentielle des actions

```
typedef struct xvpnact {
  struct xvpnact  *NEXT;      /* action suivante    */
  struct xvpnact **PREV;      /* action précédente  */
  vexexpr         *VEX_ATOM;  /* membre gauche      */
  vexexpr         *VEX_EXPR;  /* membre droit       */
  unsigned char    TYPE;      /* affectation, appel */
} xvpnact_list;               /* de procédure etc   */


typedef struct xvpnarc {
  void           *SOURCE;     /* place/transition source      */
  void           *TARGET;     /* place/transition destination */
  unsigned char   TYPE;       /* place-trans, trans-place     */
} xvpnarc;

typedef struct xvpnplace {
  struct xvpnplace  *NEXT;       /* place suivante             */
  struct xvpnplace **PREV;       /* place précédente           */
  char              *NAME;       /* nom de la place            */
  unsigned char      FATHER;     /* processus/fonction         */
  unsigned char      TYPE;       /* instruction qu'elle        */
                                 /* représente IF/WAIT etc     */
  unsigned short     TOKEN;      /* valeur de sa marque        */
  unsigned short     NUMBER_IN;  /* nombre d'arcs entrants     */
  unsigned short     NUMBER_OUT; /* nombre d'arcs sortants     */
  chain_list        *TRANS_IN;   /* liste des trans entrantes  */
  chain_list        *TRANS_OUT;  /* liste des trans sortantes  */
  struct xvpnplace  *LINK;       /* place de fin d'instruction */
} xvpnplace_list;                /* END IF/END LOOP etc        */

typedef struct xvpntrans {
  struct xvpntrans  *NEXT;       /* transition suivante        */
  struct xvpntrans **PREV;       /* transition précédente      */
  char              *NAME;       /* nom de la transition       */
  unsigned char      FATHER;     /* processus/fonction         */
  unsigned char      TYPE;       /* EXECUTE/RESUME             */
  chain_list        *WAIT_SYM;   /* liste de sensibilité       */
  unsigned short     NUMBER_IN;  /* nombre d'arc entrant       */
  unsigned short     NUMBER_OUT; /* nombre d'arc sortant       */
  chain_list        *PLACE_IN;   /* liste des places entrantes */
  chain_list        *PLACE_OUT;  /* liste des places sortantes */
  xvpnact_list      *ACT;        /* liste des actions          */
  xvpnact_list     **LAST_ACT;   /* dernière action            */
  vexexpr           *VEX_GUARD;  /* condition franchissement   */
} xvpntrans_list;
```

Figure D.14: La représentation des arcs, des places et des transitions.

(*xvpnact*) qui doivent être exécutées lorsqu'elle franchie. Une action peut être soit un appel de procédure, soit une affectation.

Une transition lorsqu'elle est de type *RESUME* possède une liste *WAIT_SYM* des signaux pour lesquels elle attend un évènement.

D.4 La représentation d'une description de niveau *RTL*

La structure de données *rtlfig* représente le comportement d'un circuit sous la forme d'affectations concurrentes et d'automates d'états finis synchrone avec des registres en sorties. Elle est la structure de données cible de l'analyse sémantique.

```
typedef struct rtlfig {
  struct rtlfig *NEXT;        /* figure suivante           */
  char          *NAME;        /* nom de la figure          */
  rtlasg_list   *ASSIGN;      /* liste des affectations    */
  rtlfsm_list   *FSM;         /* liste des FSM             */
  rtldecl_list  *DECLAR       /* liste des déclarations    */
  [ RTL_MAX_DECLAR_TYPE ];    /* triées par catégories     */
  authtable     *HASH_DECLAR; /* table de hachage des decl */
} rtlfig_list;
```

Figure D.15: La structure de données *rtlfig*.

Les déclarations de symboles sont représentées à l'aide de la structure de données *rtldecl*. Afin de manipuler seulement certains bits d'un vecteur, les symboles sont également représentés au niveau bit à l'aide de la structure *rtlsym*. La figure D.16 donne la définition de ces deux structures de données.

Les affectations concurrentes modélisent des registres, des barrières trois états dont les propriétés sont clairement identifiées, ainsi que des sorties de blocs combinatoires. Elles sont représentées par la structure de données *rtlasg*.

Lorsqu'il s'agit d'une affectation représentant la sortie d'un bloc combinatoire, alors les champs *VEX_ATOM* et *VEX_DATA* représentent respectivement le nom du symbole affecté et l'expression combinatoire.

Lorsqu'il s'agit d'un registre ou d'une barrière trois états, alors le champs *VEX_ATOM* est le nom du symbole affecté, et les différentes propriétés sont données par une liste de *rtlbivex*.

La figure D.17 donne la définition de ces deux structures de données.

La figure D.18 donne l'ensemble des structures de données qui sont utilisées pour représenter un automate d'états finis synchrone avec des registres en sorties.

```
typedef struct rtlsym {
  char           *NAME;     /* nom du symbole              */
  struct rtldecl *DECL;     /* déclaration du symbole      */
  short           INDEX;    /* numéro du bit du symbole    */
} rtlsym;

typedef struct rtldecl {
  struct rtldecl *NEXT;     /* déclaration suivante        */
  vexexpr        *VEX_ATOM; /* nom du symbole              */
  vexexpr        *VEX_INIT; /* valeur initiale             */
  vpnsym         *DECL_SYM; /* tableau des bits du symbole */
  unsigned char   DIR;      /* direction IN/OUT/INOUT      */
  unsigned char   TYPE;     /* port,signal,argument etc    */
  unsigned char   KIND;     /* catégorie : bus/register    */
  unsigned char   BASE;     /* bit, bit_vector,etc         */
} rtldecl_list;
```

Figure D.16: La représentation des symboles de la description.

```
typedef struct rtlbivex {
  struct rtlbivex *NEXT;      /* bi-vex suivant             */
  vexexpr         *VEX_COND;  /* condition                  */
  vexexpr         *VEX_DATA;  /* valeur                     */
  unsigned char    TYPE;      /* type du bi-vex             */
} rtlbivex_list;              /* SET_ASYNC,FALLING_EDGE etc */

typedef struct rtlasg {
  struct rtlasg  *NEXT;       /* affectation suivante       */
  vexexpr        *VEX_ATOM;   /* membre gauche de l'affect. */
  vexexpr        *VEX_DATA;   /* membre droit de l'affect.  */
  rtlbivex_list  *BIVEX;      /* bi-vex pour les registres  */
                                 les tri-states et les mux  */
  rtlbivex_list **LAST_BIVEX; /* dernier bi-vex             */
  unsigned char   TYPE;       /* REGISTER/TRISTATE/MUX etc  */
  unsigned char   REG_TYPE;   /* type du registre : ASYNC   */
} rtlasg_list;                /* SYNC ou MIXED              */
```

Figure D.17: Les affectations concurrentes.

```
 typedef struct rtlfsmasg
 {
   struct rtlfsmasg *NEXT;      /* affectation suivante        */
   vexexpr          *VEX_ATOM;  /* membre gauche de l'affect.  */
   vexexpr          *VEX_DATA;  /* membre droit de l'affect.   */
 } rtlfsmasg_list;

typedef struct rtlfsmstate {
  struct rtlfsmstate *NEXT;     /* état suivant                */
  char               *NAME;     /* nom de l'état               */
  chain_list         *FROM;     /* liste des trans sortantes   */
  chain_list         *TO;       /* liste des trans entrantes   */
  rtlfsmasg_list     *ASSIGN;   /* liste des affectations      */
} rtlfsmstate_list;

typedef struct rtlfsmtrans {
  struct rtlfsmtrans *NEXT;     /* transition suivante         */
  struct rtlfsmstate *FROM;     /* état de départ              */
  struct rtlfsmstate *TO;       /* état d'arrivée              */
  rtlfsmasg_list     *ASSIGN;   /* liste des affectations      */
  vexexpr            *VEX_COND; /* condition de franchissement */
} rtlfsmtrans_list;

typedef struct rtlfsm {
  struct rtlfsm    *NEXT;       /* FSM suivante                */
  char             *NAME;       /* nom de la FSM               */
  rtlfsmstate_list *STATE;      /* liste des états             */
  rtlfsmtrans_list *TRANS;      /* liste des transitions       */
  authtable        *HASH_STATE; /* table de hachage des états  */
  rtlbivex_list    *BIVEX;      /* propriétés du reg d'état    */
  rtlbivex_list   **LAST_BIVEX;
} rtlfsm_list;
```

Figure D.18: La représentation d'un automate d'états finis synchrone.

ANNEXE E

Le code source des processus à un seul point de suspension

Cette annexe donne le code source des descriptions VHDL qui ont été utilisées lors de l'expérimentation présentée dans le chapitre 9. Ces descriptions sont composées exclusivement de processus à un seul point de suspension.

E.1 Les descriptions synthétisables

Toutes les descriptions présentées dans cette section modélisent le comportement d'un élément matériel simple, comme une bascule D, un latch, un multiplexeur ou une barrière trois états. Pour chacune de ces descriptions, un tableau récapitule les résultats qui ont été obtenus pour les différents outils de synthèse. La première colonne donne le nom de l'outil de synthèse. La seconde donne le résultat obtenu à la compilation : OK ou KO. La dernière donne le résultat obtenu lors de la synthèse : NO signifie que la synthèse n'a pas pu être effectuée, OK signifie que la description a été synthétisée correctement, et KO signifie que l'implantation matérielle obtenue ne correspond pas au comportement initial.

E.1.1 Les descriptions d'une bascule D

Les figures E.1, E.2, E.3, E.4, E.5 donnent le code sources de toutes les descriptions d'une bascule D qui ont été expérimentées dans le chapitre 9.

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  OK  |  OK  |
--| Design Compiler  |  OK  |  OK  |
--| Asic Synthesizer |  OK  |  OK  |
--| Synergy          |  OK  |  OK  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity d00 is port(
  CLK : in std_logic;
  I0 : in std_logic;
  O0 : out std_logic);
end d00;
ARCHITECTURE BEHAVIOR OF d00 IS
BEGIN
 process
 begin
   wait
   until (clk='0' and clk'event);
   O0 <= I0;
 end process;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  OK  |  OK  |
--| Design Compiler  |  OK  |  OK  |
--| Asic Synthesizer |  OK  |  OK  |
--| Synergy          |  OK  |  OK  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity d01 is port(
  CLK : in std_logic;
  I0 : in std_logic;
  O0 : out std_logic);
end d01;
ARCHITECTURE BEHAVIOR OF d01 IS
BEGIN
 process
 begin
   wait until clk='0';
   O0 <= I0;
 end process;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  KO  |  NO  |
--| Design Compiler  |  OK  |  OK  |
--| Asic Synthesizer |  KO  |  NO  |
--| Synergy          |  KO  |  NO  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity d02 is port(
  CLK : in std_logic;
  I0 : in std_logic;
  O0 : out std_logic);
end d02;
ARCHITECTURE BEHAVIOR OF d02 IS
BEGIN
 process
 begin
   wait on clk,i0
   until (clk='0' and clk'event);
   O0 <= I0;
 end process;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  OK  |  OK  |
--| Design Compiler  |  KO  |  NO  |
--| Asic Synthesizer |  KO  |  NO  |
--| Synergy          |  KO  |  NO  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity d03 is port(
        CLK : in std_logic;
        I0 : in std_logic;
        O0 : out std_logic
     );
end d03;
ARCHITECTURE BEHAVIOR OF d03 IS
BEGIN
 process
 begin
   if (clk='0' and clk'event)
   then
     O0 <= I0;
   end if;
   wait on clk,i0;
 end process;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  OK  |  OK  |
--| Design Compiler  |  KO  |  NO  |
--| Asic Synthesizer |  KO  |  NO  |
--| Synergy          |  KO  |  NO  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity d04 is port(
  CLK : in std_logic;
  I0 : in std_logic;
  O0 : out std_logic
     );
end d04;
ARCHITECTURE BEHAVIOR OF d04 IS
BEGIN
 process
 begin
   wait on clk;
   if ( clk='0')
   then
     O0 <= I0;
   end if;
 end process;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  OK  |  OK  |
--| Design Compiler  |  OK  |  OK  |
--| Asic Synthesizer |  OK  |  OK  |
--| Synergy          |  OK  |  OK  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity d05 is port(
        CK : in std_logic;
        I0 : in std_logic;
        O0 : out std_logic
     );
end d05;
ARCHITECTURE BEHAVIOR OF d05 IS
BEGIN
process( CK )
begin
  if ( CK = '0' and ck'event ) then
     O0 <= I0;
   end if;
end process;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  OK  |  OK  |
--| Design Compiler  |  OK  |  KO  |
--| Asic Synthesizer |  OK  |  OK  |
--| Synergy          |  KO  |  NO  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity d06 is port(
        CK : in std_logic;
        I0 : in std_logic;
        O0 : out std_logic
     );
end d06;
ARCHITECTURE BEHAVIOR OF d06 IS
BEGIN
process( CK )
begin
  if ( CK = '0' ) then
     O0 <= I0;
   end if;
end process;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  OK  |  KO  |
--| Design Compiler  |  KO  |  NO  |
--| Asic Synthesizer |  OK  |  NO  |
--| Synergy          |  KO  |  NO  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity d07 is port(
        CK : in std_logic;
        I0 : in std_logic;
        O0 : out std_logic
     );
end d07;
ARCHITECTURE BEHAVIOR OF d07 IS
BEGIN
process( CK )
begin
  if (ck'event)
  then
    if ( CK = '0' ) then
       O0 <= I0;
    end if;
  end if;
end process;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  OK  |  OK  |
--| Design Compiler  |  KO  |  NO  |
--| Asic Synthesizer |  OK  |  OK  |
--| Synergy          |  KO  |  NO  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity d08 is port(
        CK : in std_logic;
        I0 : in std_logic;
        O0 : out std_logic
     );
end d08;
ARCHITECTURE BEHAVIOR OF d08 IS
BEGIN
process( CK )
begin
  if (not(CK)='1' and ck'event)
  then
     O0 <= I0;
  end if;
end process;
END BEHAVIOR;
```

Figure E.1: Les descriptions d'une bascule D (1).

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  OK  |  OK  |
--| Design Compiler  |  KO  |  NO  |
--| Asic Synthesizer |  KO  |  NO  |
--| Synergy          |  OK  |  OK  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity d09 is port(
  CK,I0 : in std_logic;
  O0 : out std_logic);
end d09;
ARCHITECTURE BEHAVIOR OF d09 IS
BEGIN
process( CK )
  variable V : std_logic;
begin
  if ( CK = '0' and ck'event )
  then V := I0;
  end if;
  O0 <= V;
end process;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  KO  |  NO  |
--| Design Compiler  |  KO  |  NO  |
--| Asic Synthesizer |  KO  |  NO  |
--| Synergy          |  KO  |  NO  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity d10 is port(
  CK,I0 : in std_logic;
  O0 : out std_logic);
end d10;
ARCHITECTURE BEHAVIOR OF d10 IS
  signal S : std_logic;
BEGIN
process( CK )
  variable V : std_logic;
begin
  V := S;
  if ( CK = '0' and ck'event )
  then V := I0;
  end if;
  S <= V;
end process;
 O0 <= S;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  KO  |  NO  |
--| Design Compiler  |  OK  |  KO  |
--| Asic Synthesizer |  OK  |  KO  |
--| Synergy          |  KO  |  NO  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity d11 is port(
  CK,I0 : in std_logic;
  O0 : out std_logic);
end d11;
ARCHITECTURE BEHAVIOR OF d11 IS
  signal S : std_logic;
BEGIN
process( CK )
begin
  S <= (I0 AND NOT(CK)) OR
       (S AND CK);
end process;
O0 <= S;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  OK  |  OK  |
--| Design Compiler  |  KO  |  NO  |
--| Asic Synthesizer |  OK  |  OK  |
--| Synergy          |  KO  |  NO  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity d12 is port(
  CK,I0 : in std_logic;
  O0 : out std_logic);
end d12;
ARCHITECTURE BEHAVIOR OF d12 IS
  signal S : std_logic;
BEGIN
process( CK )
begin
  if ( CK = '0' and ck'event )
  then S <= I0;
  else S <= S;
  end if;
end process;
 O0 <= S;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  OK  |  OK  |
--| Design Compiler  |  OK  |  KO  |
--| Asic Synthesizer |  OK  |  OK  |
--| Synergy          |  KO  |  NO  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity d13 is port(
  CK,I0 : in std_logic;
  O0 : out std_logic);
end d13;
ARCHITECTURE BEHAVIOR OF d13 IS
  signal S : std_logic;
BEGIN
process( CK )
begin
  if (CK = '0')
  then S <= I0;
  else S <= S;
  end if;
end process;
 O0 <= S;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  OK  |  OK  |
--| Design Compiler  |  KO  |  NO  |
--| Asic Synthesizer |  KO  |  NO  |
--| Synergy          |  KO  |  NO  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity d14 is port(
  CK,I0 : in std_logic;
  O0 : out std_logic);
end d14;
ARCHITECTURE BEHAVIOR OF d14 IS
BEGIN
process ( CK )
  VARIABLE B : boolean;
begin
  B := (CK = '0') AND CK'EVENT;
  IF (B)
  THEN  O0 <= I0;
  END IF;
end process;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  KO  |  NO  |
--| Design Compiler  |  KO  |  NO  |
--| Asic Synthesizer |  KO  |  NO  |
--| Synergy          |  KO  |  NO  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity d15 is port(
  CK,I0 : in std_logic;
  O0 : out std_logic
     );
end d15;
ARCHITECTURE BEHAVIOR OF d15 IS
BEGIN
process
begin
  LOOP
    WAIT ON CK;
    EXIT WHEN CK='0';
  END LOOP;
  O0 <= I0;
end process;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  OK  |  KO  |
--| Design Compiler  |  KO  |  NO  |
--| Asic Synthesizer |  OK  |  OK  |
--| Synergy          |  KO  |  NO  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity d16 is port(
  CK,I0 : in std_logic;
  O0 : out std_logic);
end d16;
ARCHITECTURE BEHAVIOR OF d16 IS
BEGIN
process ( CK )
begin
  IF NOT(NOT(CK'EVENT) OR CK /='0')
  THEN  O0 <= I0;
  END IF;
end process;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  KO  |  NO  |
--| Design Compiler  |  KO  |  NO  |
--| Asic Synthesizer |  KO  |  NO  |
--| Synergy          |  KO  |  NO  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity d17 is port(
  CK,I0 : in std_logic;
  O0 : inout std_logic);
end d17;
ARCHITECTURE BEHAVIOR OF d17 IS
BEGIN
process ( CK )
  VARIABLE V : std_logic;
begin
  IF (NOT(CK'EVENT) OR CK /='0')
  THEN  V := O0;
  ELSE  V := I0;
  END IF;
  O0 <= V;
end process;
END BEHAVIOR;
```

Figure E.2: Les descriptions d'une bascule D (2).

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  KO  |  NO  |
--| Design Compiler  |  OK  |  KO  |
--| Asic Synthesizer |  OK  |  OK  |
--| Synergy          |  KO  |  NO  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity d18 is port(
  CK,I0,O0 : in std_logic;
  O0 : inout std_logic);
end d18;
ARCHITECTURE BEHAVIOR OF d18 IS
BEGIN
process ( CK )
  VARIABLE V : std_logic;
begin
  IF (CK /='0')
  THEN  V := O0;
  ELSE  V := I0;
  END IF;
  O0 <= V;
end process;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  OK  |  OK  |
--| Design Compiler  |  OK  |  KO  |
--| Asic Synthesizer |  OK  |  OK  |
--| Synergy          |  KO  |  NO  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity d19 is port(
  CK,I0 : in std_logic;
  O0 : out std_logic);
end d19;
ARCHITECTURE BEHAVIOR OF d19 IS
BEGIN
process ( CK )
  VARIABLE B : boolean;
begin
  B := (CK = '0');
  IF (B) THEN  O0 <= I0;
  END IF;
end process;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  OK  |  OK  |
--| Design Compiler  |  KO  |  NO  |
--| Asic Synthesizer |  OK  |  OK  |
--| Synergy          |  OK  |  OK  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity d20 is port(
  CK,I0,I1 : in std_logic;
  O0,O1 : out std_logic);
end d20;
ARCHITECTURE BEHAVIOR OF d20 IS
BEGIN
process ( CK )
begin
  IF (CK='1' AND CK'EVENT)
  THEN O0 <= I0;
  END IF;
  IF (CK='1' AND CK'EVENT)
  THEN O1 <= I1;
  END IF;
end process;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  OK  |  OK  |
--| Design Compiler  |  KO  |  NO  |
--| Asic Synthesizer |  OK  |  OK  |
--| Synergy          |  OK  |  OK  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity d21 is port(
  CK,I0 : in std_logic;
  O0 : out std_logic);
end d21;
ARCHITECTURE BEHAVIOR OF d21 IS
BEGIN
process ( CK )
  VARIABLE V : std_logic;
begin
  IF (CK='1' AND CK'EVENT)
  THEN O0 <= V;
  END IF;
  IF (CK='1' AND CK'EVENT)
  THEN V := I0;
  END IF;
end process;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  KO  |  NO  |
--| Design Compiler  |  KO  |  NO  |
--| Asic Synthesizer |  OK  |  OK  |
--| Synergy          |  OK  |  OK  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity d22 is port(
  CK,I0 : in std_logic;
  O0 : out std_logic);
end d22;
ARCHITECTURE BEHAVIOR OF d22 IS
BEGIN
process ( CK )
  VARIABLE V : std_logic;
begin
  IF (CK='1' AND CK'EVENT)
  THEN V := I0;
  END IF;
  IF (CK='0' AND CK'EVENT)
  THEN O0 <= V;
  END IF;
end process;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  KO  |  NO  |
--| Design Compiler  |  KO  |  NO  |
--| Asic Synthesizer |  OK  |  OK  |
--| Synergy          |  OK  |  OK  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity d23 is port(
  CK,I0 : in std_logic;
  O0 : out std_logic);
end d23;
ARCHITECTURE BEHAVIOR OF d23 IS
BEGIN
process ( CK )
  VARIABLE V : std_logic;
begin
  IF (CK='1' AND CK'EVENT)
  THEN V := I0;
  END IF;
  IF (CK='0' AND CK'EVENT)
  THEN O0 <= V;
  END IF;
end process;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  OK  |  OK  |
--| Design Compiler  |  OK  |  OK  |
--| Asic Synthesizer |  OK  |  OK  |
--| Synergy          |  OK  |  OK  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity d24 is port(
  CK,I0 : in std_logic;
  O0 : out std_logic);
end d24;
ARCHITECTURE BEHAVIOR OF d24 IS
BEGIN
process ( CK )
  VARIABLE V : std_logic;
begin
  IF (CK='1' AND CK'EVENT)
  THEN O0 <= V; V := I0;
  END IF;
end process;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  KO  |  NO  |
--| Design Compiler  |  KO  |  NO  |
--| Asic Synthesizer |  KO  |  NO  |
--| Synergy          |  OK  |  OK  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity d25 is port(
  CK,I0,I1 : in std_logic;
  O1,O0 : out std_logic);
end d25;
ARCHITECTURE BEHAVIOR OF d25 IS
BEGIN
process ( CK )
  VARIABLE V : std_logic;
begin
  V := I0;
  IF (CK='1' AND CK'EVENT)
  THEN O0 <= V; V := I1;
       O1 <= V;
  END IF;
end process;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  OK  |  OK  |
--| Design Compiler  |  OK  |  OK  |
--| Asic Synthesizer |  OK  |  OK  |
--| Synergy          |  OK  |  OK  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity d_ra00 is port(
  CK,I0,R : in std_logic;
  O0 : out std_logic);
end d_ra00;
ARCHITECTURE BEHAVIOR OF d_ra00 IS
BEGIN
 process( CK, R )
 begin
   IF (R='1')
   THEN O0 <= '0';
   ELSIF (CK='0' and CK'EVENT)
   THEN O0 <= I0;
   END IF;
 end process;
END BEHAVIOR;
```

Figure E.3: Les descriptions d'une bascule D (3).

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  KO  |  NO  |
--| Design Compiler  |  KO  |  NO  |
--| Asic Synthesizer |  KO  |  NO  |
--| Synergy          |  KO  |  NO  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity d_ra01 is port(
  CK,I0,R : in std_logic;
  O0 : out std_logic);
end d_ra01;
ARCHITECTURE BEHAVIOR OF d_ra01 IS
BEGIN
 process( CK, R )
 begin
   IF (R=’1’)
   THEN O0 <= ’0’;
   END IF;
   IF (R=’0’ AND CK=’0’ and CK’EVENT)
   THEN O0 <= I0;
   END IF;
 end process;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  OK  |  OK  |
--| Design Compiler  |  OK  |  OK  |
--| Asic Synthesizer |  OK  |  OK  |
--| Synergy          |  OK  |  OK  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity d_ra02 is port(
  CK,I0,R : in std_logic;
  O0 : out std_logic);
end d_ra02;
ARCHITECTURE BEHAVIOR OF d_ra02 IS
BEGIN
 process( CK, R )
 begin
   IF (CK=’0’ and CK’EVENT)
   THEN O0 <= I0;
   END IF;
   IF (R=’1’) THEN O0 <= ’0’;
   END IF;
 end process;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  OK  |  OK  |
--| Design Compiler  |  OK  |  OK  |
--| Asic Synthesizer |  KO  |  NO  |
--| Synergy          |  KO  |  NO  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity d_ra03 is port(
  CK,I0,R : in std_logic;
  O0 : out std_logic);
end d_ra03;
ARCHITECTURE BEHAVIOR OF d_ra03 IS
BEGIN
 process( CK, R )
   VARIABLE V : std_logic;
 begin
   IF (CK=’0’ and CK’EVENT)
   THEN V := I0;
   END IF;
   IF (R=’1’)
   THEN V := ’0’;
   END IF;
   O0 <= V;
 end process;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  KO  |  NO  |
--| Design Compiler  |  KO  |  NO  |
--| Asic Synthesizer |  KO  |  NO  |
--| Synergy          |  KO  |  NO  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity d_ra04 is port(
  CK,I0,R : in std_logic;
  O0 : out std_logic);
end d_ra04;
ARCHITECTURE BEHAVIOR OF d_ra04 IS
BEGIN
 process( CK, R )
 begin
   IF ((R=’1’) OR (CK=’0’ and CK’EVENT))
   THEN O0 <= I0 AND NOT(R);
   END IF;
 end process;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  OK  |  OK  |
--| Design Compiler  |  OK  |  KO  |
--| Asic Synthesizer |  OK  |  OK  |
--| Synergy          |  KO  |  NO  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity d_ws00 is port(
  CK,WE,I0 : in std_logic;
  O0 : out std_logic);
end d_ws00;
ARCHITECTURE BEHAVIOR OF d_ws00 IS
BEGIN
 PROCESS( ck )
 BEGIN
   IF ( ck=’0’ )
   THEN
     IF ( we=’0’ )
     THEN O0 <= I0;
     END IF;
   END IF;
 END PROCESS;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  KO  |  NO  |
--| Design Compiler  |  KO  |  NO  |
--| Asic Synthesizer |  OK  |  OK  |
--| Synergy          |  KO  |  NO  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity d_ws01 is port(
  CK,WE,I0 : in std_logic;
  O0 : out std_logic);
end d_ws01;
ARCHITECTURE BEHAVIOR OF d_ws01 IS
BEGIN
 PROCESS( ck )
 BEGIN
   IF ( we=’0’ and
       ck=’0’ and ck’event )
   THEN O0 <= I0;
   END IF;
 END PROCESS;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  KO  |  NO  |
--| Design Compiler  |  OK  |  KO  |
--| Asic Synthesizer |  OK  |  OK  |
--| Synergy          |  KO  |  NO  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity d_ws02 is port(
  CK,WE,I0 : in std_logic;
  O0 : out std_logic);
end d_ws02;
ARCHITECTURE BEHAVIOR OF d_ws02 IS
BEGIN
 PROCESS( ck )
 BEGIN
   IF ( we=’0’ and ck=’0’)
   THEN O0 <= I0;
   END IF;
 END PROCESS;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  KO  |  NO  |
--| Design Compiler  |  KO  |  NO  |
--| Asic Synthesizer |  OK  |  OK  |
--| Synergy          |  KO  |  NO  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity d_ws03 is port(
  CK,WE,I0 : in std_logic;
  O0 : out std_logic);
end d_ws03;
ARCHITECTURE BEHAVIOR OF d_ws03 IS
BEGIN
 PROCESS
 BEGIN
   WAIT on ck
   UNTIL ck=’0’ and we=’0’;
   O0 <= I0;
 END PROCESS;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  OK  |  OK  |
--| Design Compiler  |  OK  |  OK  |
--| Asic Synthesizer |  OK  |  OK  |
--| Synergy          |  OK  |  OK  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity d_ws04 is port(
  CK,WE,I0 : in std_logic;
  O0 : out std_logic);
end d_ws04;
ARCHITECTURE BEHAVIOR OF d_ws04 IS
BEGIN
 PROCESS
 BEGIN
   WAIT ON ck UNTIL ck=’0’;
   IF ( we=’0’)
   THEN O0 <= I0;
   END IF;
 END PROCESS;
END BEHAVIOR;
```

Figure E.4: Les descriptions d’une bascule D (4).

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  OK  |  OK  |
--| Design Compiler  |  OK  |  OK  |
--| Asic Synthesizer |  OK  |  OK  |
--| Synergy          |  OK  |  OK  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity d_ws05 is port(
  CK,WE,I0 : in std_logic;
  O0 : out std_logic);
end d_ws05;
ARCHITECTURE BEHAVIOR OF d_ws05 IS
BEGIN
 PROCESS( CK )
 BEGIN
   IF (we='0')
   THEN
     IF (ck='0' and ck'event)
     THEN  O0 <= I0;
     END IF;
   END IF;
 END PROCESS;
END BEHAVIOR;
```

Figure E.5: Les descriptions d'une bascule D (5).

E.1.2 Les descriptions d’un latch

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  KO  |  NO  |
--| Design Compiler  |  OK  |  KO  |
--| Asic Synthesizer |  KO  |  NO  |
--| Synergy          |  KO  |  NO  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity l00 is port(
  CLK,I0 : in std_logic;
  O0 : out std_logic);
end l00;
ARCHITECTURE BEHAVIOR OF l00 IS
BEGIN
 process
 begin
   wait on clk,i0 until clk='1';
   O0 <= I0;
 end process;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  KO  |  NO  |
--| Design Compiler  |  KO  |  NO  |
--| Asic Synthesizer |  KO  |  NO  |
--| Synergy          |  KO  |  NO  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity l01 is port(
  CLK,I0 : in std_logic;
  O0 : out std_logic);
end l01;
ARCHITECTURE BEHAVIOR OF l01 IS
BEGIN
 process
 begin
   wait on clk,i0
   until clk='1' or
        (clk'event and clk='1');
   O0 <= I0;
 end process;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  OK  |  OK  |
--| Design Compiler  |  KO  |  NO  |
--| Asic Synthesizer |  KO  |  NO  |
--| Synergy          |  OK  |  OK  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity l04 is port(
  CLK,I0 : in std_logic;
  O0 : out std_logic);
end l04;
ARCHITECTURE BEHAVIOR OF l04 IS
BEGIN
 process
 begin
   wait on clk,i0;
   if ( clk='1')
   then O0 <= I0;
   end if;
 end process;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  OK  |  KO  |
--| Design Compiler  |  OK  |  OK  |
--| Asic Synthesizer |  OK  |  OK  |
--| Synergy          |  OK  |  OK  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity l03 is port(
  CLK,I0 : in std_logic;
  O0 : out std_logic);
end l03;
ARCHITECTURE BEHAVIOR OF l03 IS
  SIGNAL S : std_logic;
BEGIN
 process( CLK, I0 )
 begin
   IF (CLK ='1' AND I0='1')
   THEN S <= '1';
   END IF;
   IF (CLK='1' AND I0='0')
   THEN S <= '0';
   END IF;
 end process;
  O0 <= S;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  OK  |  OK  |
--| Design Compiler  |  OK  |  KO  |
--| Asic Synthesizer |  OK  |  OK  |
--| Synergy          |  OK  |  OK  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity l02 is port(
  CLK,I0 : in std_logic;
  O0 : out std_logic);
end l02;
ARCHITECTURE BEHAVIOR OF l02 IS
  SIGNAL S : std_logic;
BEGIN
 process( CLK, I0 )
 begin
   IF (CLK ='1' AND I0='1')
   THEN S <= '1';
   ELSIF (CLK='1' AND I0='0')
   THEN S <= '0';
   ELSE S <= S;
   END IF;
 end process;
 O0 <= S;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  OK  |  OK  |
--| Design Compiler  |  OK  |  OK  |
--| Asic Synthesizer |  OK  |  OK  |
--| Synergy          |  OK  |  OK  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity l05 is port(
  CK,I0 : in std_logic;
  O0 : out std_logic);
end l05;
ARCHITECTURE BEHAVIOR OF l05 IS
BEGIN
process( CK, I0 )
begin
  if ( CK = '1' )
  then O0 <= I0;
   end if;
end process;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  OK  |  KO  |
--| Design Compiler  |  KO  |  NO  |
--| Asic Synthesizer |  OK  |  KO  |
--| Synergy          |  KO  |  NO  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity l06 is port(
  CK,I0 : in std_logic;
  O0 : out std_logic);
end l06;
ARCHITECTURE BEHAVIOR OF l06 IS
BEGIN
process( CK, I0 )
begin
  if ( CK = '1' AND
      (CK'EVENT OR NOT(CK'EVENT)))
  then O0 <= I0;
  end if;
end process;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  OK  |  OK  |
--| Design Compiler  |  OK  |  OK  |
--| Asic Synthesizer |  OK  |  OK  |
--| Synergy          |  OK  |  OK  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity l08 is port(
  CK,I0 : in std_logic;
  O0 : out std_logic);
end l08;
ARCHITECTURE BEHAVIOR OF l08 IS
BEGIN
process( CK, I0 )
begin
  if ( not(CK) = '0')
  then O0 <= I0;
  end if;
end process;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  OK  |  OK  |
--| Design Compiler  |  OK  |  OK  |
--| Asic Synthesizer |  OK  |  OK  |
--| Synergy          |  OK  |  OK  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity l09 is port(
  CK,I0 : in std_logic;
  O0 : out std_logic);
end l09;
ARCHITECTURE BEHAVIOR OF l09 IS
BEGIN
  process( CK, I0 )
    variable V : std_logic;
  begin
    if (CK = '1') then V := I0;
    end if;
    O0 <= V;
  end process;
END BEHAVIOR;
```

Figure E.6: Les descriptions d’un *Latch*(1).

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  KO  |  NO  |
--| Design Compiler  |  OK  |  KO  |
--| Asic Synthesizer |  OK  |  OK  |
--| Synergy          |  OK  |  OK  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity l10 is port(
  CK,I0 : in std_logic;
  O0 : out std_logic);
end l10;
ARCHITECTURE BEHAVIOR OF l10 IS
  signal S : std_logic;
BEGIN
process( CK, I0 )
  variable V : std_logic;
begin
  V := S;
  if ( CK = '1' )
  then V := I0;
  end if;
  S <= V;
end process;
 O0 <= S;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  OK  |  KO  |
--| Design Compiler  |  OK  |  KO  |
--| Asic Synthesizer |  OK  |  KO  |
--| Synergy          |  OK  |  KO  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity l11 is port(
  CK,I0 : in std_logic;
  O0 : out std_logic);
end l11;
ARCHITECTURE BEHAVIOR OF l11 IS
  signal S : std_logic;
BEGIN
process( CK, I0, S )
begin
  S <= (I0 AND CK) OR
       (S  AND NOT(CK));
end process;
O0 <= S;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  OK  |  OK  |
--| Design Compiler  |  OK  |  KO  |
--| Asic Synthesizer |  OK  |  OK  |
--| Synergy          |  OK  |  OK  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity l12 is port(
  CK,I0 : in std_logic;
  O0 : out std_logic);
end l12;
ARCHITECTURE BEHAVIOR OF l12 IS
  signal S : std_logic;
BEGIN
process( CK, I0, S )
begin
  if ( CK = '1')
  then S <= I0;
  else S <= S;
  end if;
end process;
 O0 <= S;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  KO  |  NO  |
--| Design Compiler  |  KO  |  NO  |
--| Asic Synthesizer |  KO  |  NO  |
--| Synergy          |  KO  |  NO  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity l15 is port(
  CK,I0 : in std_logic;
  O0 : out std_logic);
end l15;
ARCHITECTURE BEHAVIOR OF l15 IS
BEGIN
process
begin
  LOOP
    WAIT ON CK, I0;
    EXIT WHEN CK='1';
  END LOOP;
  O0 <= I0;
end process;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  OK  |  KO  |
--| Design Compiler  |  KO  |  NO  |
--| Asic Synthesizer |  OK  |  KO  |
--| Synergy          |  KO  |  NO  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity l17 is port(
  CK,I0 : in std_logic;
  O0 : out std_logic);
end l17;
ARCHITECTURE BEHAVIOR OF l17 IS
BEGIN
 process( ck, i0 )
 begin
   if ( ck'event and ck='1')
   then O0 <= I0;
   end if;
   if ( not(ck'event) and ck='1' )
   then O0 <= I0;
   end if;
 end process;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  KO  |  NO  |
--| Design Compiler  |  OK  |  OK  |
--| Asic Synthesizer |  KO  |  NO  |
--| Synergy          |  KO  |  NO  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity l18 is port(
  CK,I1,I0 : in std_logic;
  O0 : out std_logic);
end l18;
ARCHITECTURE BEHAVIOR OF l18 IS
  SIGNAL S : std_logic;
BEGIN
 PROCESS( CK, I0 )
   VARIABLE V : STD_LOGIC;
 BEGIN
   IF (CK='1') THEN V := I0;
   END IF;
   IF (CK='0') THEN V := S;
   END IF;
   S <= V;
 END PROCESS;
 O0 <= S;
END BEHAVIOR;
```

Figure E.7: Les descriptions d'un *Latch*(2).

E.1.3 Les descriptions d'un multiplexeur

```
--+-----------------+------+------+
--|       Tool      | Comp | Synt |
--+ Autologic       |  OK  |  OK  |
--| Design Compiler |  OK  |  OK  |
--| Asic Synthesizer|  OK  |  OK  |
--| Synergy         |  OK  |  OK  |
--+-----------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity mux00 is port(
  SEL,I0,I1 : in std_logic;
        O0  : out std_logic);
end mux00;
ARCHITECTURE BEHAVIOR OF mux00 IS
BEGIN
 process( sel, i0, i1 )
 begin
   if (sel='1')
   then O0 <= i0;
   elsif (sel='0')
   then O0 <= i1;
   end if;
 end process;
END BEHAVIOR;
```

```
--+-----------------+------+------+
--|       Tool      | Comp | Synt |
--+ Autologic       |  OK  |  OK  |
--| Design Compiler |  KO  |  NO  |
--| Asic Synthesizer|  KO  |  NO  |
--| Synergy         |  OK  |  OK  |
--+-----------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity mux01 is port(
  SEL,I0,I1 : in std_logic;
        O0  : out std_logic);
end mux01;
ARCHITECTURE BEHAVIOR OF mux01 IS
BEGIN
 process
 begin
   if (sel='1')
   then O0 <= i0;
   else O0 <= i1;
   end if;
   wait on sel,i0,i1;
 end process;
END BEHAVIOR;
```

```
--+-----------------+------+------+
--|       Tool      | Comp | Synt |
--+ Autologic       |  OK  |  KO  |
--| Design Compiler |  OK  |  KO  |
--| Asic Synthesizer|  OK  |  OK  |
--| Synergy         |  OK  |  OK  |
--+-----------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity mux02 is port(
  SEL,I0,I1 : in std_logic;
        O0  : out std_logic);
end mux02;
ARCHITECTURE BEHAVIOR OF mux02 IS
BEGIN
 process( sel, i0, i1 )
 begin
   if (sel='1')
   then O0 <= i0;
   end if;
   if (sel='0')
   then O0 <= i1;
   end if;
 end process;
END BEHAVIOR;
```

```
--+-----------------+------+------+
--|       Tool      | Comp | Synt |
--+ Autologic       |  KO  |  NO  |
--| Design Compiler |  KO  |  NO  |
--| Asic Synthesizer|  KO  |  NO  |
--| Synergy         |  KO  |  NO  |
--+-----------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity mux03 is port(
  SEL,I0,I1 : in std_logic;
        O0  : out std_logic);
end mux03;
ARCHITECTURE BEHAVIOR OF mux03 IS
BEGIN
 process
 begin
   wait on sel,i0,i1
    until sel='0' or sel='1';
   if (sel='1')
   then O0 <= i0;
   else O0 <= i1;
   end if;
 end process;
END BEHAVIOR;
```

```
--+-----------------+------+------+
--|       Tool      | Comp | Synt |
--+ Autologic       |  OK  |  KO  |
--| Design Compiler |  OK  |  KO  |
--| Asic Synthesizer|  OK  |  OK  |
--| Synergy         |  OK  |  OK  |
--+-----------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity mux04 is port(
        CK : in std_logic;
     I1,I0 : in std_logic;
        O0 : out std_logic);
end mux04;
ARCHITECTURE BEHAVIOR OF mux04 IS
  SIGNAL S : std_logic;
BEGIN
 PROCESS( CK, I0, I1 )
   VARIABLE V : STD_LOGIC;
 BEGIN
   IF (CK='1') THEN V := I0;
   END IF;
   IF (CK='0') THEN V := I1;
   END IF;
   S <= V;
 END PROCESS;
 O0 <= S;
END BEHAVIOR;
```

```
--+-----------------+------+------+
--|       Tool      | Comp | Synt |
--+ Autologic       |  OK  |  KO  |
--| Design Compiler |  OK  |  KO  |
--| Asic Synthesizer|  OK  |  OK  |
--| Synergy         |  OK  |  OK  |
--+-----------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity mux05 is port(
  SEL,I0,I1 : in std_logic;
        O0  : out std_logic);
end mux05;
ARCHITECTURE BEHAVIOR OF mux05 IS
BEGIN
 process( sel, i0, i1 )
 begin
   if (sel='1')
   then O0 <= i0;
   elsif (sel/='1')
   then O0 <= i1;
   end if;
 end process;
END BEHAVIOR;
```

```
--+-----------------+------+------+
--|       Tool      | Comp | Synt |
--+ Autologic       |  OK  |  KO  |
--| Design Compiler |  OK  |  KO  |
--| Asic Synthesizer|  OK  |  OK  |
--| Synergy         |  OK  |  OK  |
--+-----------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity mux06 is port(
  SEL,I0,I1 : in std_logic;
        O0  : out std_logic);
end mux06;
ARCHITECTURE BEHAVIOR OF mux06 IS
BEGIN
 process( sel, i0, i1 )
   variable v : std_logic;
 begin
   if (sel='1') then V := i0;
   end if;
   if (sel/='1') then V := i1;
   end if;
   O0 <= V;
 end process;
END BEHAVIOR;
```

Figure E.8: Les descriptions d'un multiplexeur.

E.1.4 Les descriptions d’une barrière trois états

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  OK  |  OK  |
--| Design Compiler  |  OK  |  OK  |
--| Asic Synthesizer |  OK  |  OK  |
--| Synergy          |  OK  |  OK  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity trs00 is port(
  SEL,I0 : in std_logic;
     O0  : out std_logic
     );
end trs00;
ARCHITECTURE BEHAVIOR OF trs00 IS
BEGIN
 process( SEL, I0 )
 begin
   IF ( SEL=’1’) THEN O0 <= I0;
   ELSE               O0 <= ’Z’;
   END IF;
 end process;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  OK  |  OK  |
--| Design Compiler  |  OK  |  OK  |
--| Asic Synthesizer |  OK  |  OK  |
--| Synergy          |  OK  |  OK  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity trs01 is port(
  SEL,I0 : in std_logic;
     O0  : out std_logic
     );
end trs01;
ARCHITECTURE BEHAVIOR OF trs01 IS
BEGIN
 O0 <= I0 WHEN SEL=’1’ ELSE ’Z’;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  OK  |  OK  |
--| Design Compiler  |  OK  |  KO  |
--| Asic Synthesizer |  OK  |  OK  |
--| Synergy          |  OK  |  OK  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity trs02 is port(
  SEL,I0 : in std_logic;
     O0  : out std_logic
     );
end trs02;
ARCHITECTURE BEHAVIOR OF trs02 IS
BEGIN
 PROCESS( SEL, I0 )
 BEGIN
   IF ( SEL=’1’)
   THEN O0 <= I0;
   END IF;
   IF ( SEL/=’1’)
   THEN O0 <= ’Z’;
   END IF;
 END PROCESS;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  OK  |  OK  |
--| Design Compiler  |  OK  |  OK  |
--| Asic Synthesizer |  OK  |  OK  |
--| Synergy          |  OK  |  OK  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity trs03 is port(
  SEL,I0 : in std_logic;
     O0  : out std_logic
     );
end trs03;
ARCHITECTURE BEHAVIOR OF trs03 IS
BEGIN
 PROCESS( SEL, I0 )
   VARIABLE V : std_logic;
 BEGIN
   IF (SEL=’1’)
   THEN V := I0;
   ELSE V := ’Z’;
   END IF;
   O0 <= V;
 END PROCESS;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  OK  |  OK  |
--| Design Compiler  |  OK  |  OK  |
--| Asic Synthesizer |  OK  |  OK  |
--| Synergy          |  OK  |  OK  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity trs04 is port(
  SEL,I0 : in std_logic;
     O0  : out std_logic
     );
end trs04;
ARCHITECTURE BEHAVIOR OF trs04 IS
BEGIN
 PROCESS( SEL, I0 )
 BEGIN
   IF ( SEL=’1’ )
   THEN O0 <= ’Z’;
   ELSIF (SEL=’0’)
   THEN O0 <= I0;
   END IF;
 END PROCESS;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+ Autologic        |  OK  |  KO  |
--| Design Compiler  |  OK  |  KO  |
--| Asic Synthesizer |  OK  |  OK  |
--| Synergy          |  OK  |  OK  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity trs05 is port(
  SEL,I0 : in std_logic;
     O0  : out std_logic
     );
end trs05;
ARCHITECTURE BEHAVIOR OF trs05 IS
BEGIN
 PROCESS( SEL, I0 )
 BEGIN
   IF ( SEL=’1’ )
   THEN O0 <= ’Z’;
   ELSIF (SEL/=’1’)
   THEN O0 <= I0;
   END IF;
 END PROCESS;
END BEHAVIOR;
```

Figure E.9: Les descriptions d’une barrière trois états.

E.2 Les descriptions non-synthétisables

Toutes les descriptions présentées sur la figure E.10 modélisent un comportement qui n’est pas implantable en matériel. Pour chacune de ces descriptions, un tableau récapitule les résultats qui ont été obtenus pour les différents outils de synthèse. La première colonne donne le nom de l’outil de synthèse. La seconde donne le résultat obtenu à la compilation : OK signifie que la description a été acceptée à tort, et KO qu’elle a été rejetée.

```
--+------------------+------+
--|       Tool       | Comp |
--+ Autologic        |  OK  |
--| Design Compiler  |  OK  |
--| Asic Synthesizer |  KO  |
--| Synergy          |  OK  |
--+------------------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity nmat00 is port(
 CK,I1,I0 : in std_logic;
       O0 : out std_logic
     );
end nmat00;
ARCHITECTURE BEHAVIOR OF nmat00 IS
BEGIN
  process( CK, I0, I1 )
    variable V : std_logic;
  begin
    if ( CK = '1' )
    then V := I0;
    end if;
    O0 <= V;
    V := I1;
  end process;
END BEHAVIOR;
```

```
--+------------------+------+
--|       Tool       | Comp |
--+ Autologic        |  OK  |
--| Design Compiler  |  OK  |
--| Asic Synthesizer |  KO  |
--| Synergy          |  OK  |
--+------------------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity nmat01 is port(
  SEL,I0 : in std_logic;
     O0  : out std_logic);
end nmat01;
ARCHITECTURE BEHAVIOR OF nmat01 IS
BEGIN
 PROCESS( SEL, I0 )
   VARIABLE V, VZ : std_logic;
 BEGIN
   O0 <= V;
   IF ( SEL='1' )
   THEN V := 'Z';
   ELSE V := I0;
   END IF;
 END PROCESS;
END BEHAVIOR;
```

```
--+------------------+------+
--|       Tool       | Comp |
--+ Autologic        |  KO  |
--| Design Compiler  |  OK  |
--| Asic Synthesizer |  KO  |
--| Synergy          |  KO  |
--+------------------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity nmat02 is port(
  CK,I1,I0 : in std_logic;
        O0 : out std_logic
      );
end nmat02;
ARCHITECTURE BEHAVIOR OF nmat02 IS
BEGIN
  process( CK, I0 )
  begin
    if ( CK = '1' )
    then O0 <= I0 OR I1;
    end if;
  end process;
END BEHAVIOR;
```

```
--+------------------+------+
--|       Tool       | Comp |
--+ Autologic        |  OK  |
--| Design Compiler  |  OK  |
--| Asic Synthesizer |  OK  |
--| Synergy          |  OK  |
--+------------------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity nmat03 is port(
  CK,I0 : in std_logic;
     O0 : out std_logic
     );
end nmat03;
ARCHITECTURE BEHAVIOR OF nmat03 IS
BEGIN
  process( CK )
    variable V : std_logic;
  begin
    O0 <= V;
    if ( CK'EVENT AND CK='1')
    then V := I0;
    end if;
  end process;
END BEHAVIOR;
```

```
--+------------------+------+
--|       Tool       | Comp |
--+ Autologic        |  OK  |
--| Design Compiler  |  OK  |
--| Asic Synthesizer |  OK  |
--| Synergy          |  OK  |
--+------------------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity nmat04 is port(
  CK,I1,I0 : in std_logic;
        O0 : out std_logic
      );
end nmat04;
ARCHITECTURE BEHAVIOR OF nmat04 IS
BEGIN
  process( CK, I1 )
    variable V : std_logic;
  begin
    O0 <= V OR I1;
    if ( CK'EVENT AND CK='1')
    then V := I0;
    end if;
  end process;
END BEHAVIOR;
```

```
--+------------------+------+
--|       Tool       | Comp |
--+ Autologic        |  KO  |
--| Design Compiler  |  OK  |
--| Asic Synthesizer |  OK  |
--| Synergy          |  KO  |
--+------------------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity nmat05 is port(
  CK,I1,I0 : in std_logic;
        O0 : out std_logic
      );
end nmat05;
ARCHITECTURE BEHAVIOR OF nmat05 IS
BEGIN
  process( CK )
    variable V : std_logic;
  begin
    O0 <= V OR I1;
    IF ( CK='1')
    THEN V := I0;
    END IF;
  end process;
END BEHAVIOR;
```

```
--+------------------+------+
--|       Tool       | Comp |
--+ Autologic        |  OK  |
--| Design Compiler  |  OK  |
--| Asic Synthesizer |  KO  |
--| Synergy          |  OK  |
--+------------------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity nmat06 is port(
  I1,I0 : in std_logic;
     O0 : out std_logic
     );
end nmat06;
ARCHITECTURE BEHAVIOR OF nmat06 IS
BEGIN
  process( I0, I1 )
    variable V : std_logic;
  begin
    O0 <= V OR I1;
    V := I0;
  end process;
END BEHAVIOR;
```

```
--+------------------+------+
--|       Tool       | Comp |
--+ Autologic        |  KO  |
--| Design Compiler  |  OK  |
--| Asic Synthesizer |  KO  |
--| Synergy          |  KO  |
--+------------------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity nmat07 is port(
  CK,I0 : in std_logic;
     O0 : out std_logic
     );
end nmat07;
ARCHITECTURE BEHAVIOR OF nmat07 IS
BEGIN
  process
  begin
    wait on i0 until ck='1';
    O0 <= I0;
  end process;
END BEHAVIOR;
```

```
--+------------------+------+
--|       Tool       | Comp |
--+ Autologic        |  OK  |
--| Design Compiler  |  OK  |
--| Asic Synthesizer |  KO  |
--| Synergy          |  OK  |
--+------------------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity nmat08 is port(
  CK,I1,I0 : in std_logic;
        O0 : out std_logic
      );
end nmat08;
ARCHITECTURE BEHAVIOR OF nmat08 IS
BEGIN
  process( CK, I1, I0 )
    variable V : std_logic;
  begin
    O0 <= V OR I1;
    IF ( CK='1')
    THEN V := I0;
    END IF;
  end process;
END BEHAVIOR;
```

Figure E.10: Les descriptions non synthétisables.

ANNEXE **F**

Le code source des processus à plusieurs points de suspension

Cette annexe donne le code source des descriptions VHDL qui ont été utilisées lors de l'expérimentation présentée dans le chapitre 9. Ces descriptions sont composées exclusivement de processus à plusieurs points de suspension.

F.1 Les descriptions synthétisables

Toutes les descriptions présentées dans cette section modélisent le comportement d'un automate d'états finis synchrone avec des registres en sorties. Pour chacune de ces descriptions, un tableau récapitule les résultats qui ont été obtenus pour les différents outils de synthèse. La première colonne donne le nom de l'outil de synthèse. La seconde donne le résultat obtenu à la compilation : OK ou KO. La dernière donne le résultat obtenu lors de la synthèse : NO signifie que la synthèse n'a pas pu être effectuée, OK signifie que la description a été synthétisée correctement, et KO signifie que l'implantation matérielle obtenue ne correspond pas au comportement initial.

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+------------------+------+------+
--| Design Compiler  |  OK  |  OK  |
--| Asic Synthesizer |  OK  |  OK  |
--| Synergy          |  OK  |  OK  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity mw00 is port(
  CK  : in BIT;
  SEL : in BIT_VECTOR(1 downto 0);
  I0  : in BIT;
  I1  : in BIT;
  I2  : in BIT;
  O0  : out BIT);
end mw00;
ARCHITECTURE BEHAVIOR OF mw00 IS
BEGIN
  PROCESS
  BEGIN
    WAIT UNTIL CK='1';
    CASE SEL IS
      WHEN "00" => O0 <= I0;
      WHEN "01" => O0 <= I1;
      WHEN "10" => O0 <= I2;
                   WAIT UNTIL CK='1';
   O0 <= I1;
      WHEN "11" => O0 <= I0;
                   WAIT UNTIL CK='1';
                   O0 <= I1;
    END CASE;
  END PROCESS;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+------------------+------+------+
--| Design Compiler  |  OK  |  OK  |
--| Asic Synthesizer |  OK  |  OK  |
--| Synergy          |  KO  |  NO  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity mw02 is port(
  CK  : in BIT;
  I0  : in BIT;
  O0  : out BIT;
  O1  : out BIT_vector(1 downto 0));
end mw02;
ARCHITECTURE BEHAVIOR OF mw02 IS
BEGIN
  PROCESS
    VARIABLE RA, RB, SUM : bit;
  BEGIN
    SUM := RA XOR RB;
    O1 <= "10";
    O0 <= SUM;
    WAIT UNTIL CK='1' AND CK'EVENT;
    RA := I0;
    O1 <= "00";
    WAIT UNTIL CK='1' AND CK'EVENT;
    RB := I0;
    O1 <= "01";
    WAIT UNTIL CK='1' AND CK'EVENT;
  END PROCESS;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+------------------+------+------+
--| Design Compiler  |  KO  |  NO  |
--| Asic Synthesizer |  OK  |  OK  |
--| Synergy          |  KO  |  NO  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity mw01 is port(
  CK  : in BIT;
  WEN : in BIT;
  SEL : in BIT_VECTOR(1 downto 0);
  I0  : in BIT;
  I1  : in BIT;
  I2  : in BIT;
  O0  : out BIT);
end mw01;
ARCHITECTURE BEHAVIOR OF mw01 IS
BEGIN
  PROCESS
  BEGIN
    WAIT ON CK UNTIL WEN='1' and CK='1';
    CASE SEL IS
      WHEN "00" => O0 <= I0;
      WHEN "01" => O0 <= I1;
      WHEN "10" => O0 <= I2;
                   WAIT ON CK UNTIL WEN='1' and CK='1';
                   O0 <= I1;
      WHEN "11" => O0 <= I0;
                   WAIT ON CK UNTIL WEN='1' and CK='1';
                   O0 <= I1;
    END CASE;
  END PROCESS;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+------------------+------+------+
--| Design Compiler  |  OK  |  OK  |
--| Asic Synthesizer |  KO  |  NO  |
--| Synergy          |  KO  |  NO  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity mw03 is port(
  CK  : in BIT;
  WEN : in BIT;
  I0  : in BIT;
  Q   : out BIT_VECTOR(1 DOWNTO 0));
end mw03;
ARCHITECTURE BEHAVIOR OF mw03 IS
BEGIN
  PROCESS
  BEGIN
    L1 : LOOP
      WAIT UNTIL WEN='1' AND CK='1' AND CK'EVENT;
      Q <= "00";
      EXIT L1 WHEN I0='0';
      WAIT UNTIL WEN='1' AND CK='1' AND CK'EVENT;
      Q <= "01";
      EXIT L1 WHEN I0='0';
      L2 : LOOP
        WAIT UNTIL WEN='1' AND CK='1' AND CK'EVENT;
        Q <= "10";
        EXIT L1 WHEN I0='0';
      END LOOP L2;
    END LOOP L1;
  END PROCESS;
END BEHAVIOR;
```

Figure F.1: Les processus à plusieurs points de suspension (1)

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+------------------+------+------+
--| Design Compiler  |  KO  |  NO  |
--| Asic Synthesizer |  OK  |  OK  |
--| Synergy          |  KO  |  NO  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity mw04 is port(
  CK  : in BIT;
  WEN  : in BIT;
  I0  : in BIT;
  I1  : in BIT;
  Q   : out BIT);
end mw04;
ARCHITECTURE BEHAVIOR OF mw04 IS
BEGIN
  PROCESS
  BEGIN
    WAIT ON CK UNTIL WEN='1' AND CK='1';
    IF(I0='1')
    THEN
      Q <= I1;
    ELSE
      WAIT UNTIL CK='1';
      Q <= I0;
    END IF;
  END PROCESS;
END BEHAVIOR;

--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+------------------+------+------+
--| Design Compiler  |  OK  |  OK  |
--| Asic Synthesizer |  OK  |  OK  |
--| Synergy          |  OK  |  OK  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity mw06 is port(
  CK  : in BIT;
  I0  : in BIT;
  O0  : out BIT;
  O1  : out BIT_vector(1 downto 0));
end mw06;
ARCHITECTURE BEHAVIOR OF mw06 IS
BEGIN
  PROCESS
    VARIABLE RA, RB, SUM : bit;
  BEGIN
    WAIT UNTIL CK='1' AND CK'EVENT;
    SUM := RA XOR RB;
    O1 <= "10";
    O0 <= SUM;
    WAIT UNTIL CK='1' AND CK'EVENT;
    RA := I0;
    O1 <= "00";
    WAIT UNTIL CK='1' AND CK'EVENT;
    RB := I0;
    O1 <= "01";
  END PROCESS;
END BEHAVIOR;
```

```
--+------------------+------+------+
--|       Tool       | Comp | Synt |
--+------------------+------+------+
--| Design Compiler  |  OK  |  OK  |
--| Asic Synthesizer |  OK  |  OK  |
--| Synergy          |  KO  |  NO  |
--+------------------+------+------+
library IEEE;
use IEEE.std_logic_1164.all;
entity mw05 is port(
  CK  : in BIT;
  I0  : in BIT;
  I1  : in BIT;
  I2  : in BIT;
  Q   : out BIT);
end mw05;
ARCHITECTURE BEHAVIOR OF mw05 IS
BEGIN
  PROCESS
  BEGIN
    L1 : LOOP
      L0 : LOOP
        WAIT UNTIL CK='1';
        EXIT L0 WHEN I0='1';
EXIT L1 WHEN I0='0';
      END LOOP L0;
      Q <= I1;
    END LOOP L1;
    Q <= I2;
    WAIT UNTIL CK='1' AND CK'EVENT;
  END PROCESS;
END BEHAVIOR;
```

Figure F.2: Les processus à plusieurs points de suspension (2)

F.2 Les descriptions algorithmiques

Toutes les descriptions présentées dans cette section modélisent le comportement d'un algorithme implantable sous la forme d'un automate d'états finis synchrone avec des registres en sorties.

```
entity crc is port (
  ck     : in  bit;
  din    : in  bit_vector(31 downto 0);
  dout   : out bit_vector(31 downto 0);
  req    : out bit_vector(1 downto 0);
  fifo   : in  bit);
end crc;

architecture pisim of crc is

constant read  : bit_vector(1 downto 0 ) := "01";
constant write : bit_vector(1 downto 0 ) := "10";
constant nop   : bit_vector(1 downto 0 ) := "00";
constant empty : bit := '0';
constant full  : bit := '1';
constant eof   : bit_vector(31 downto 0) := X"FFFFFFFF";
constant zero  : bit_vector(31 downto 0) := X"00000000";

begin
crc : process

  variable rcr : bit_vector(31 downto 0);
  variable icr : bit_vector(31 downto 0);
  variable car : bit_vector(31 downto 0);

begin
  icr := zero;
  infinity: loop
    rcr := icr;
    req <= read;
    wait on ck until ((ck = '0') and (fifo = full));
    car := din;
    req <= nop;
    exit infinity when (car = eof);
    icr := rcr xor (X"000000" & car(7 downto 0));
    rcr := icr;
    icr := rcr xor (rcr(2 downto 0) & rcr(31 downto 3));
    rcr := icr;
    icr := rcr xor (rcr(22 downto 0) & rcr(31 downto 23));
  end loop;
  req <= write;
  dout <= rcr;
  wait on ck until ((ck = '0') and (fifo = empty));
  req <= nop;
end process;
end pisim;
```

Figure F.3: Les descriptions algorithmiques (1)

```
entity diffeq is port(
  dx_p, a_p, x_p, u_p : in integer range 0 to 255;
  y_p                 : inout integer range 0 to 255;
  ck, go              : in bit );
end diffeq;

architecture behavior of diffeq is
begin
   process
    variable x, y, a, u,
             dx, xl, yl, ul : integer range 0 to 255;
    variable inf : boolean;
   begin
    wait on ck until ck='1' and go = '1';
    x := x_p;
    y := y_p;
    a := a_p;
    u := u_p;
    dx := dx_p;
    inf := x < a;
    while ( inf )
    loop
     wait until ck = '1';
     xl := x + dx;
     ul := u - (3 * x * u * dx) - (3 * y * dx);
     yl := y + (u * dx);
     x  := xl;
     u  := ul;
     y  := yl;
    end loop;
    y_p <= y;
   end process;
end behavior;
```

Figure F.4: Les descriptions algorithmiques (2)

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity MULTI is
port (
  CK     : in  bit;
  DIN    : in  UNSIGNED(7 downto 0);
  DOUT   : out UNSIGNED(15 downto 0);
  REQ    : out bit_vector( 1 downto 0 );
  FULL   : in  bit;
  EMPTY  : in  bit);
end MULTI;

architecture PISIM of MULTI is

CONSTANT READ    : bit_vector (1 downto 0 ):="01" ;
CONSTANT WRITE   : bit_vector (1 downto 0 ):="10" ;
CONSTANT NOP     : bit_vector (1 downto 0 ):="00" ;
CONSTANT ZERO_8  : UNSIGNED(7 downto 0) := "00000000";
CONSTANT ZERO_16 : UNSIGNED(15 downto 0) := ZERO_8 & ZERO_8;

BEGIN

MULTI : PROCESS
  VARIABLE RA  : UNSIGNED(15 downto 0);
  VARIABLE RB  : UNSIGNED(15 downto 0);
  VARIABLE RM  : UNSIGNED(15 downto 0);
BEGIN
-- lecture asynchrone premier opérande
  REQ <= READ;
  WAIT on CK until ( CK = '1' and EMPTY = '0' );
  RA  := ZERO_8 & DIN;
-- lecture asynchrone deuxième opérande
  REQ <= READ;
  WAIT on CK until ( CK = '1' and EMPTY = '0' );
  RB := ZERO_8 & DIN;
-- calcul effectif du MULTI
  REQ <= NOP;
  RM := ZERO_16;
  WHILE ( RA /= ZERO_16 )
  LOOP
    IF ( RA(0) = '1' ) THEN
      RM := RM + RB ;
    END IF;
    RA := '0' & RA(15 downto 1);
    RB := RB(14 downto 0) & '0';
    WAIT until CK='1';
  END LOOP;
-- écriture asynchrone du résultat
  REQ  <= WRITE;
  DOUT <= RM;
  WAIT on CK until ( CK = '1' and FULL = '0' );

END PROCESS;
END PISIM;
```

Figure F.5: Les descriptions algorithmiques (3)

```
entity PGCD is port (
     CK     : in  bit;
     DIN    : in  integer;
     DOUT   : out integer;
     REQ    : out bit_vector( 1 downto 0 );
     FULL   : in  bit;
     EMPTY  : in  bit);
end PGCD;

architecture PISIM of PGCD is

CONSTANT READ  : bit_vector (1 downto 0 ):="01" ;
CONSTANT WRITE : bit_vector (1 downto 0 ):="10" ;
CONSTANT NOP   : bit_vector (1 downto 0 ):="00" ;

BEGIN
PGCD : PROCESS
  VARIABLE RA  : integer;
  VARIABLE RB  : integer;

  VARIABLE NUL  : boolean ;
  VARIABLE INF  : boolean ;
BEGIN
-- lecture asynchrone premier opérande
  REQ <= READ;
  WAIT on CK until ( CK = '1' and EMPTY = '0' );
  RA  := DIN;
-- lecture asynchrone deuxième opérande
  REQ <= READ;
  WAIT on CK until ( CK = '1' and EMPTY = '0' );
  RB  := DIN;
-- calcul effectif du PGCD
  REQ <= NOP;
  NUL := (RA = RB );
  WHILE ( NOT NUL )
  LOOP
    INF := (RA < RB );
    IF ( INF ) THEN
      RB:= RB - RA ;
    ELSE
      RA:= RA - RB ;
    END IF;
    NUL := (RA = RB );
    WAIT until CK='1';
  END LOOP;
-- écriture asynchrone du résultat
  REQ  <= WRITE;
  DOUT <= RA ;
  WAIT on CK until ( CK = '1' and FULL = '0' );
END PROCESS;
END PISIM;
```

Figure F.6: Les descriptions algorithmiques (4)