EXTENSION DES DIAGRAMMES DE DECISIONS BINAIRES

POUR LA REPRESENTATION DE PROGRAMMES VHDL

EN VUE DE LEUR VERIFICATION.

Guillaume Decuq[1] et Emmanuelle Encrenaz-Tiphène

Laboratoire d'Informatique de Paris VI
Université Pierre et Marie Curie (Paris VI)
4, place Jussieu
75252 Paris cedex 05
France

contact : Emmanuelle.Encrenaz@lip6.fr

Résumé : Ce document présente une adaptation des Diagrammes de Décision de Données (DDD ou D3) développés au LABRI, pour la représentation de programmes VHDL utilisant des variables booléennes et entières. Il décrit le modèle des D3 utilisés, leur interprétation pour la représentation d'ensemble d'états et de relation de transitions, ainsi que les principaux algorithmes de parcours d'espace d'états nécessaires à la vérification par model-checking.

[1]Ce travail a été réalisé pendant que G. Decuq était scientifique du contingent affecté au LIP6, de septembre 1999 à juin 2000, dans le cadre du projet DGA-99-34-024 réunissant le LIP6, le LaBRI (Université Bordeaux 1) et la DGA.

1. INTRODUCTION

Le langage VHDL permet de modéliser des systèmes matériels de différentes manières, adaptées aux différents points de vue des concepteurs de circuits électroniques. Ainsi, une même entité (définissant l'interface d'un composant électronique) peut disposer de différentes descriptions de ses fonctionnalités. Citons

- la vue structurelle (une interconnexion d'entités de moindre complexité),
- la vue flot de données (un réseau de portes logiques élémentaires, ou réseau booléen), avec ou (plus généralement) sans spécification de délais de portes et de transmission des lignes,
- la vue transfert de registres (des automates d'états finis cadencés sur une ou (plus rarement) plusieurs horloges, implantés en réseaux booléens interconnectant des éléments mémorisants, les registres),
- la vue algorithmique permettant d'utiliser des structures de données complexes, des schémas itératifs, des appels de fonctions, … mais également de définir des systèmes partiellement ou totalement asynchrones, ainsi que des paramètres temporels variables.

Cette dernière vue offre la plus grande liberté de description des fonctionnalités d'un système, mais elle se heurte à un manque d'outils de synthèse et de vérification pour pouvoir être pleinement exploitée. Si la synthèse de haut niveau, actuellement en plein essor, offre quelques solutions pour la synthèse de systèmes décrits à un niveau algorithmique, les outils de vérification automatiques – efficaces pour les systèmes purement booléens– ont du mal à appréhender des systèmes décrits à ce niveau d'abstraction.

Il nous semble qu'un des problèmes importants est la représentation de systèmes manipulant d'une part des données booléennes représentant des parties de chemin de données : décaleurs, …, routeurs, fft, …, et d'autre part des données entières représentant d'autres parties de chemins de données décrites à un niveau abstrait : additionneur, multiplieur,… mais également des paramètres temporels du système.

La recherche de « bonnes » représentations de tels systèmes n'est pas nouvelle : dès 1994, [HP 94] proposait une modélisation de programmes VHDL incluant des données entières (et bornées) basée sur les diagrammes de décisions binaires, ou BDD [Bryant 86]. Mais à notre connaissance (et de notre propre expérience …), ces représentations n'ont pas apporté les performances escomptées. Il faut coder chaque intervalle entier et réécrire les opérateurs entiers (et temporels) dans le domaine booléen ; ces représentations deviennent trop grosses pour être efficacement manipulables dès que les intervalles sont grands.

D'autres représentations arborescentes avec partage maximal des structures isomorphes ont été proposées, citons les BMD [BC 95], les EVBDD [LPV 96], les OKFDD [DB 98], et les MTBDD [CMZFY 93]. La première a été développée pour la vérification de circuits arithmétiques. La seconde consiste en l'adjonction d'expressions à évaluer aux arcs 0 et 1 des nœuds des BDD. La troisième est basée sur différentes décompositions des fonctions booléennes (Shannon, Reed-Muller forme positive ou négative), et permet de mélanger dans un même OKFDD les différentes décompositions, mais ceci reste confiné aux expressions booléennes (et éventuellement entières transformées en booléennes). Les MTBDD quant à eux sont une extension des BDD à des nœuds terminaux n-aires et non plus binaires. Ils sont utilisés dans les systèmes RAVEN [RK 00] et VERUS [CCM 97] pour la vérification de systèmes temps-réels (dans lesquels le temps est représenté par une variable discrète, contrairement au système KRONOS [HNSY 92] se basant sur un temps continu). Ces trois outils manipulent des systèmes décrits sous la forme d'automates temporisés.

Notre objectif est de définir et étudier une structure de données arborescente, présentant un partage maximal des sous-structures isomorphes, et permettant une représentation efficace (1) de données entières bornées (mais dont on ne connaît pas a priori la borne), modélisant des parties de chemins de données arithmétiques ou des paramètres temporels, et (2) de données booléennes modélisant d'autres parties de circuits. Pour ce faire, nous particularisons deux structures de données coopérantes proposées dans [CGS 98] pour la modélisation et la vérification de réseaux de Petri à files, les *Diagrammes de Décision de Données* (DDD ou D3) et les *Diagrammes de Décision et d'Opérations* (DDO ou D2O). L'étude de ces structures et son impact sur les algorithmes de vérification seront réalisés conjointement avec les auteurs de [CGS 98] dans le cadre du projet DGA-99-34-024 réunissant le LIP6, le LaBRI (Université Bordeaux 1) et la DGA.

Le présent document est structuré comme suit : La section 2 fait un rappel des définitions des D3 et D2O de [CGS 98]. La section 3 particularise les D3 pour la représentation de programmes VHDL, en définissant les VHDL-D3. La notion de canonicité y est explicitée, car elle n'apparaît pas dans [CGS 98] mais est importante pour les algorithmes de vérification. Les opérations sur les VHDL-D3 sont définies. Cette section décrit également la structure du D2O représentant la fonction de transitions d'un programme VHDL, et présente un exemple. La section 4 rappelle les principes des algorithmes de vérification symbolique de propriétés CTL et définit les adaptations nécessaires de ces algorithmes pour nos structures. La section 5 présente quelques extensions envisagées à court terme. La section 6 conclut ce document.

2. LES D3 ET LES D20 (RAPPELS DE [CGS'98])

Les D3 (DDD – diagrammes de décision de données) sont une structure de données permettant la représentation d'ensembles finis, mais dont on ne connaît pas a priori la borne. Cette structure est un graphe acyclique avec partage maximal des sous-graphes isomorphes se rapprochant des BDD, mais est étendue à des variables de domaine fini (et non plus booléen). Les D2O (DDO – diagramme de décision et d'opérations) sont une représentation, sous forme de graphe, de séquences d'opérations que l'on peut appliquer sur les D3. La présente section reprend les principales définitions et propriétés des D3 et D2O, décrites dans [CGS 98].

Au cours de cette présentation, nous explicitons les choix, ou au contraire les absences de choix, effectués par les auteurs de [CGS 98]. De fait, les choix non explicités dans cette partie, qui permettent de définir des D3 extrêmement généraux, seront affinés pour nos besoins dans la section suivante dévolue à la particularisation des D3 pour la modélisation et la vérification de programmes VHDL. Notons que dans [CGS 98] il n'est pas fait mention de notion d'ordre sur les variables, ni d'unicité d'occurrence d'une variable menant de la racine à 1. De plus, la réduction des arbres partagés, ainsi que la canonicité ne sont pas abordées.

Soit E un ensemble de variables e typées et de domaine Dom(e). On dénote 0 et 1 les constantes VRAI et FAUX booléennes.

Définition 1: d est un D3 ssi :

- soit $d \in \{0,1\}$: nœud terminal booléen
- soit $d = (e, a(t)_{t\in \mathrm{Dom}(e)})$. C'est un nœud intermédiaire repéré par une étiquette e, et un ensemble non vide d'arcs étiquetés par des valeurs du domaine de définition de e, et pointant sur des D3 :
    - $e \in E$
    - $x \in \mathrm{Dom}(e)$, a(x) est un D3 et $\{x \in \mathrm{Dom}(e) , a(x) \neq 0\}$ est fini.

Un D3 est nul s'il est composé de l'unique nœud 0, ou bien si toutes ses feuilles sont le nœud 0.
(Dans ce dernier cas, le D3 serait réductible à l'unique nœud 0, mais [CGS 98] ne fait aucune référence à la réduction de D3).

Définition 2: un D3 d est dit nul ssi :

- soit d = 0,
- soit d = (e, $a(t)_{t\in \mathrm{Dom}(e)}$) tq $\forall$ t $\in$ Dom(e), a(t) est un D3 nul.

Comme il n'y a aucune contrainte sur le nombre et l'ordre d'occurrence des variables dans le long de chemins du D3, [CGS 98] distinguent les notions de compatibilité entre D3 (même ensembles de chemins), et d'équivalence entre D3 (isomorphisme). En fait, l'équivalence revient à la compatibilité lorsque les D3 sont réduits (présentent un partage maximal des sous-D3 qui les composent).

Deux D3 d et d' sont compatibles si il n'existe pas de chemins les distinguant.

Définition 3: d et d' sont compatibles ssi

- soit d = 0 ou d' = 0
- soit d = d' = 1
- soit d = (e, $a(t)_{t\in \mathrm{Dom}(e)}$) et d' = (e', $a'(t)_{t\in \mathrm{Dom}(e)}$) tq :
  $\forall$ t $\in$ Dom(e), a(t) et a'(t) sont compatibles.

Deux D3 d et d' sont équivalents si, après suppression des sous-D3 nuls, d et d' sont isomorphes.

Définition 4: d et d' sont équivalents ssi :

- soit d et d' sont des D3 nuls,
- soit d, d' $\in$ {0,1}, d = d'
- soit d = (e, $a(t)_{t\in \mathrm{Dom}(e)}$) et d' = (e', $a'(t)_{t\in \mathrm{Dom}(e)}$) tq :
  $\forall$ t $\in$ Dom(e), a(t) et a'(t) sont équivalents.

On peut alors définir des opérations sur les D3.

Définition 5: opérations sur les D3 :

Soit op une opération booléenne telle que 0 op 0 = 0. L'opération op définit une opération compatible sur les D3. d1 op d2 = d3 ssi :

- si d1 = 0 alors d3 = (si 0 op 1) alors d2 sinon 0,
- si d2 = 0 alors d3 = (si 1 op 0) alors d1 sinon 0,
- si d1 = d2 = 1 alors d3 = 1 op 1,
- si d1 = (e, $a1(t)_{t\in \mathrm{Dom}(e)}$) et d2 = (e, $a2(t)_{t\in \mathrm{Dom}(e)}$) alors d3 = (e, $(a1(t) \text{ op } a2(t))_{t\in \mathrm{Dom}(e)}$)

Remarque 1: bien qu'il ne soit nulle part fait mention dans [CGS 98] de la notion d'ordre d'occurrence des variables le long de chemin menant de la racine vers les feuilles des D3, la définition précédente suppose que d1 et d2 contiennent les mêmes jeux de variables, et ces dernières apparaissent *toutes* et *dans le même ordre* le long des chemins de d1 et d2.

L'application de cette opération aux opérateurs booléens OU, ET, DIFF permet de construire l'union, l'intersection, la différence et la différence symétrique ensemblistes de deux D3. Ces opérations sont illustrées sur la Figure 1.

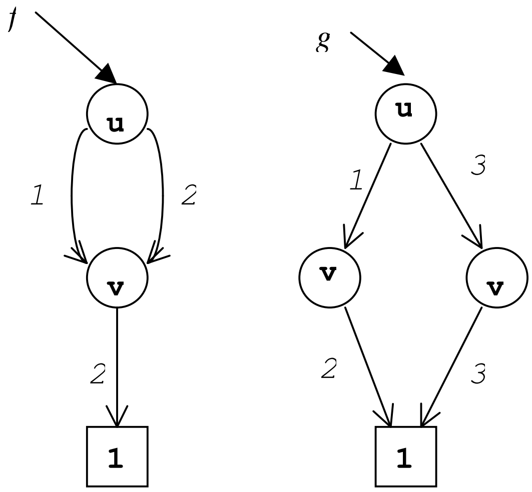

Figure 1 (a) : deux D3 f et g.

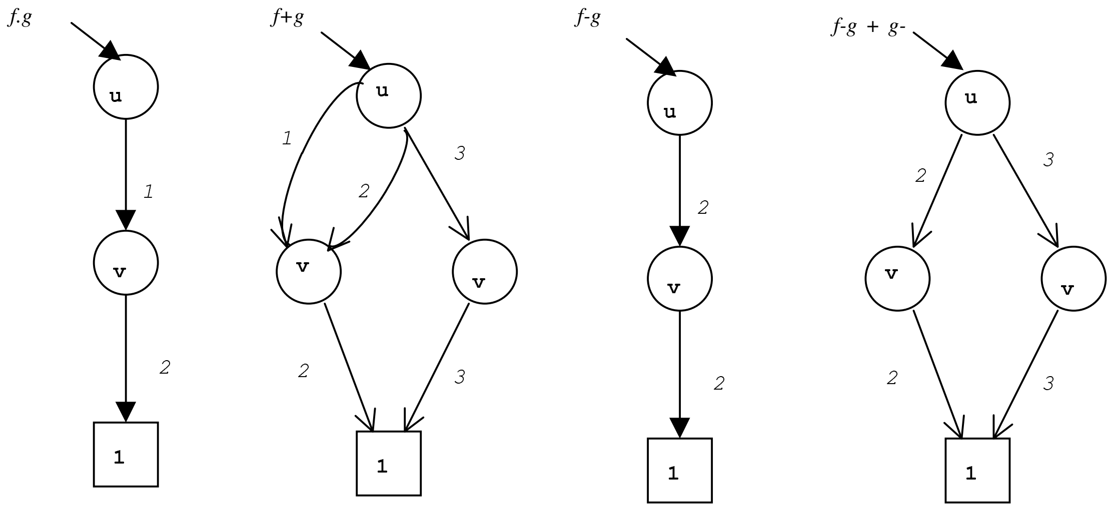

Figure 1 (b) : les opérations ensemblistes sur f et g.

Définition 6: soient deux D3 d et d', la concaténation de d et d', notée d $\rightarrow$ d', est le D3 défini tel que :

- si d = 0, d $\rightarrow$ d' = 0
- si d = 1, d $\rightarrow$ d' = d'
- si d = $(e,a(t)_{t\in Dom(e)})$, d $\rightarrow$ d' = $(e,(a(t) \rightarrow d')_{t\in Dom(e)})$

Remarque 2: Là encore, [CGS 98] ne fait aucune hypothèse sur les jeux de variables impliqués : ou bien d et d' travaillent sur deux jeux de variables distincts, ou bien il peut y avoir plusieurs occurences de e le long d'un chemin d'un D3, ou bien il faut définir plus précisément ce qui se passe lorsque la variable racine de d' est égale à la variable racine de d.

La définition de nouvelles opérations sur les D3 fait appel aux fonctions linéaires sur les D3.
Définition 7: une application O de D3 dans D3 est une fonction ssi :

- d, d' $\in$ D3, et d et d' sont compatibles => O(d) et O(d') sont compatibles.

Proposition 1: Les applications suivantes sont des fonctions :

• l'identité	Id	: $\forall$ d$\in$ D3, Id(d) = d
• constante	c	: $\forall$ d$\in$ D3, c(d) = c
• composition	O1 • O2	: $\forall$ d$\in$ D3, (O1 • O2)(d) = O1(d) • O2(d)
• concaténation	O1 $\rightarrow$ O2	: $\forall$ d$\in$ D3, (O1 $\rightarrow$ O2)(d) = O1(d) $\rightarrow$ O2(d)
• opération booléenne	op	: $\forall$ d$\in$ D3, (O1 op O2)(d) = O1(d) op O2(d)

Définition 8: Une fonction O sur les D3 est linéaire ssi:

- d, d' $\in$ D3, O(d+d') = O(d) + O(d') : le "+" désigne l'union ensembliste.
- O(0) = 0

Proposition 2: soient O1 et O2 des fonctions linéaires sur les D3 et c un D3. Les applications Id, O1•O2, c $\rightarrow$ O1, O1 + O2 sont des fonctions linéaires sur les D3.
NB : si c=(e -x$\rightarrow$ 1), la concaténation c $\rightarrow$ O1 sera notée e-x$\rightarrow$O1.

Proposition 3: Soient $(O(e,x))_{e\in E,\ x\in Dom(e)}$ une famille de fonctions linéaires et c un D3 sur E, alors la fonction O définie telle que :

- si d = 0, O(d) = 0
- si d = 1, O(d) = c
- si d = $(e,a(t)_{t\in Dom(e)})$, O(d) = $\Sigma_{t\in Dom(e)}$ O(e,t)(a(t)),

est une fonction linéaire. (Σ désigne l'union ensembliste)

Proposition 4: soient O1 et O2 des fonctions linéaires sur les D3 et a une variable de E. La fonction O définie récursivement par

- O(e,x) = si (e=a) alors O1•(e-x$\rightarrow$ O2) sinon (e-x$\rightarrow$ O)
- O(1) = 0

est une fonction linéaire notée a-O1$\rightarrow$ O2.

Nous pouvons maintenant définir le D2O (diagramme de décisions et d'opérations), qui est la représentation arborescente de la composition de fonctions linéaires.

Définition 9: Θ est un D2O ssi :

- soit Θ = 1,
- soit $\Theta = (a, (Oi,\Theta_i)_{i\,\in\,I\ fini})$

Définition 10: Soit Θ un D2O. Θ définit une fonction linéaire f(Θ):

- si Θ = 1 alors Id
- si $\Theta = (a, (Oi,\Theta_i)_{i\in I}$ alors $\Sigma_{i\in I}(a\text{-}\Theta_i\rightarrow f(\Theta_i))$

Ainsi, les D3 permettent de représenter des ensembles finis (avec des variables de domaine borné mais dont on ne connaît pas a priori la borne), sur lesquels on peut appliquer les opérations ensemblistes classiques à l'exception notoire de la complémentation (qui nécessiterait justement de disposer d'une borne sur le domaine des variables). En outre, il est possible de construire des fonctions applicables sur (les variables d')un D3. La composition de ces fonctions peut être modélisée par un graphe de décisions et d'opérations (D2O). Le parcours conjoint d'un D3 et d'un D2O permet alors de construire de nouveaux D3.

Nous allons maintenant particulariser ces notions à la modélisation de programmes VHDL en vue de leur vérification par model-checking.

3. LES VHDL-D3

Les VHDL-D3 sont une particularisation des D3 permettant la modélisation de systèmes de transitions manipulant des données booléennes et entières. Ce sont des arbres partagés, structurés en deux parties : une partie est composée de nœuds n-aires et représente des variables entières, de type compteur; l'autre partie contient des nœuds binaires et représente des fonctions à variables booléennes. En fait, cette structure en deux niveaux peut être vue comme un ensemble de BDD (représentant les variables booléennes) interconnectés à un niveau supérieur par un arbre de variables entières, dont les feuilles sont les BDD.

3.1. Le modèle des VHDL-D3

Les objets que nous représentons sont des variables booléennes, des variables entières modélisant des compteurs, et des types énumérés. Ces derniers sont assimilés au type entier (bien qu'ils s'en distinguent par la connaissance a priori que l'on a de leur borne).

3.1.1. Domaine des variables manipulées.

Définition 11: type des variables

$\forall$ e $\in$ E, Dom(e) = Bool ou Dom(e) = Int ou Dom(e) = Int $\times$ Int

Remarque 3: Par concision, les ensembles de valeurs entières contiguës sont représentées par des unions d'intervalles disjoints plutôt que par énumération des valeurs.

Remarque 4: Actuellement, les types produits mixtes (Bool $\times$ Int) ne sont pas autorisés du fait de la représentation par BDD des parties booléennes. De plus, les tableaux sont déstructurés. Ces deux limitations sont provisoires. La section 5 présente brièvement les pistes que nous envisageons pour étendre les types de données utilisés.

3.1.2. Définition des VHDL-D3

Comme mentionné précédemment, [CGS 98] définit deux structures et un mode opératoire permettant de construire des ensembles et de les manipuler. Il n'est nullement question de représentation canonique dans ces définitions. Or la canonicité est une propriété importante des structures de données représentant des ensembles pour les algorithmes de vérification, où il faut fréquemment tester l'égalité de deux ensembles. De fait, nous contraignons la structure des D3 pour la rendre canonique, en introduisant un ordre d'occurrence des variables le long des chemins menant de la racine aux feuilles, en forçant un partage maximal des nœuds, et en imposant une représentation canonique d'ensembles de valeurs entières équivalentes.

On définit une relation d'ordre sur l'ensemble E étendu aux constantes booléennes : les variables de E sont munies d'un index, assurant que les variables entières sont d'index inférieur aux variables booléennes. De plus, l'ordre est total.

Définition 12: relation d'ordre sur E :

- $\forall$ e, e' $\in$ E, Dom(e) = Int et Dom(e') = Bool, index(e) < index(e')
- $\forall$ e' $\in$ E, index(0) > index(e), index(1) > index(e)
- $\forall$e, e' $\in$ E, index(e) < index(e') ou index(e') < index(e)

Un VHDL-D3 est un graphe acyclique de feuilles terminales 0 et/ou 1, et dont les variables apparaissant le long des chemins de la racine vers les feuilles sont ordonnées suivant la relation d'ordre définie ci-dessus. De plus, les variables n'apparaissent pas nécessairement le long de tous les chemins (la valeur de ces variables est alors indéterminée le long des chemins où elles n'apparaissent pas). Enfin, le graphe est déterministe.

Définition 13: d est un VHDL-D3 sur E ssi :

- soit $d \in \{0,1\}$
- soit $d = (e, a(t)_{t \in Dom(e)})$ tq
    - si Dom(e) = Bool, $|d^{\bullet}| = 2$ (l'arité sortante des nœuds booléens est de 2)
    - et $a(t) = \{d' = (e', a(e'))$ tq $index(d) < index(d')\} \cup \{0,1\}$
    - $\exists$ d',d'' tq a(t') = d' et a(t'') = d'' et si t' = t'' alors d' = d'' (système déterministe)

Définition 14: d est réduit (partagé) ssi :

- soit $d \in \{0,1\}$,
- soit $d = (e, a(t)_{t \in Dom(e)})$ tq
    - Dom(e) = Bool et d est un ROBDD
    - Dom(e) = Int et
        - $\exists d' = (e, a'(t))$ tq $\forall t \in Dom(e)$, a(t) = a'(t), alors d = d'
        - la fonction a : Int $\rightarrow$ D3 admet une représentation canonique.

Proposition 5: un VHDL-D3 réduit est une forme canonique.
Cette proposition se déduit directement de la définition d'un VHDL-D3 réduit.

3.1.3. Création d'un VHDL-D3

Un VHDL-D3 représente un ensemble d'états d'un programme VHDL. Définissons maintenant un état d'un tel programme, ainsi que la façon de représenter des ensembles d'états par les VHDL-D3.

3.1.3.1. Définition d'un état d'un programme VHDL.

Un état d'un programme VHDL est un tuple composé des valeurs des objets manipulés par le programme, de l'état courant de chacun des processus décrits dans le programme ainsi que de l'état courant du processus noyau.

Les signaux nécessitent trois informations : leur **valeur effective** (accessible en lecture), leur(s) **valeur**(s) **pilotée**(s) accessible(s) en écriture (il peut y avoir plusieurs valeurs pilotées si plusieurs processus peuvent contribuer à l'élaboration de la valeur effective du signal, et chaque processus peut décrire une suite de valeur à affecter au signal dans le futur)), et l'**attribut événement** correspondant à un changement de la valeur effective du signal. Actuellement, nous nous bornons à une seule valeur pilotée par signal, l'extension de ce modèle est discutée en section 5.

Les variables sont représentées par leur **valeur courante**, accessible en lecture et en écriture par le processus dans lequel elles ont été déclarées (on se borne à la définition de VHDL'87 excluant les variables globales).

L'état courant de chacun des processus est considéré lors d'un état stable. Dans ce cas, seules les **positions correspondant à une attente** (instruction " wait ") sont pertinentes.

L'état courant du processus noyau détermine si le simulateur est dans une phase d'augmentation du temps (virtuel ou réel), de mise à jour des pilotes et valeurs effectives des signaux, de réveil ou d'exécution des processus réveillés. Dans le cas d'une analyse en états stables, le processus noyau est nécessairement dans une étape d'augmentation du temps courant, et **il n'est donc pas nécessaire de le représenter**.

Plus formellement, un état d'un processus VHDL est un triplet : < S,V,P>

- S = $\cup_{si \in signaux}$ (si_eff, si_drv, si_evt)
  - Dom(si_eff) = Dom(si_drv) = si type entier ou énuméré alors Int
    sinon si type bit ou boolean alors Bool
    sinon erreur
  - Dom(si_evt) = Bool
- V = $\cup_{vi \in variables}$ vi
  - Dom(vi) = si type entier ou énuméré alors Int
    sinon si type bit ou boolean alors Bool
    sinon erreur

- P = $\cup_{pi \in processus}$ pij : j est une instruction “ wait ” du processus pi et pi en contient plus d'une
  - Dom(pij) = Bool (cela pourra être assimilé à un type énuméré plus tard)

3.1.3.2. Représentation d'un ensemble d'états par une fonction caractéristique

Un ensemble d'états E peut être représenté en extension, (c'est à dire par l'énumération de tous les états qui le constituent) ou bien par une fonction caractéristique χ_E : E $\rightarrow$ Bool retournant la valeur VRAI pour tous les états de E, et FAUX sinon. Cette représentation est tout à fait adéquate avec les D3 en général et les VHDL-D3 en particulier, qui nous garantissent en plus la canonicité de la représentation.

Pour tous les éléments de S,V et P les ensembles caractérisant un état, on construit le VHDL-D3 associé : $\forall x \in S \cup V \cup P$, val(x) $\in$ Dom(x), d3(x) = x-val(x)$\rightarrow$1

Le D3 représentant l'état construit sur les variables de S,V et P est obtenu par concaténation des D3 de chacune des variables, *en respectant un ordre compatible avec celui défini pour les VHDL-D3.*

D = d3(x1) $\rightarrow$ … $\rightarrow$ d3(xn) $\rightarrow$1

3.1.3.3. Exemple : un programme VHDL, un ensemble d'états.

Soit un processus extrait d'un programme VHDL :

```
sender   : process
         variable nb     : integer;
         variable envoi  : TYPE_MSG;

      begin
         wait on REQ;  -- signal externe indiquant la requête de transmission
         nb := 0;
         while not (nb > nb_msg) loop
              nb := nb + 1;
              envoi.message := MSG; -- MSG[nb] dans une version ultérieure ...
              envoi.num := nb;
              K_in <= envoi;

              wait on L_out until (L_out = ack) for timer1;

              if (L_out'event) then -- wait franchi sur réception de l'ack
                   next;
              else -- wait franchi sur timer1 : pb de transmission du msg
                   nb := nb - 1;
                   next;
              end if;
         end loop;
     end process sender;
```

Il s'agit d'un émetteur devant transmettre nb_msg messages, chacun étant estampillé par son numéro d'ordre. Chaque message doit être acquitté avant que le message suivant soit effectivement émis. Les messages et les acquittements sont éventuellement perdus (ce qui justifie leur numérotation).
L'émetteur transmet son message sur la liaison K (signal K_in) et reçoit éventuellement l'acquittement sur la liaison L (signal L_out). La réception de l'acquittement est bornée par un délai de garde modélisé par la constante entière timer_1.

Un état de ce processus est déterminé par une configuration des variables et signaux qui l'affectent, ainsi que la position du compteur ordinal (repéré sur les instructions `wait`, et, pour des raisons de clarté, en entrée de la boucle).

état de sender [2]:

- `P` $\in$ {0,1,2} (0 :wait_on_REQ, 1 : while, 2 :wait_on_L_out...)
- `Req` $\in$ {0,1} (0 : pas de requête, 1 : soumission d'une requête)
- `nb` $\in$ [0,nb_msg]
- `envoi, K_in_drv, K_in_eff` $\in$ {VIDE,MSG}$\times$ [0,nb_msg]
- `L_out_evt, L_out_eff` $\in$ {VIDE,ACK}
- `timer1` $\in$ {0,1} 0 : non déclenché, 1: déclenché

Remarque 5: la variable `envoi` ne sert qu'à construire la valeur à affecter au signal K_in. Dans la pratique, à la fin de chaque cycle de simulation (fin de phase d'exécution), les valeurs de `envoi` et la valeur pilotée de `K_in` sont identiques. La valeur effective de `K_in` est décalée de un cycle de simulation.

Remarque 6: en réalité, il ne convient pas de représenter timer1 par une variable booléenne, mais bien le temps restant à s'écouler entre l'instant actuel et l'instant de fin de délai, déterminé à l'instant de son armement par :
$\text{instant\_courant}_{\text{armement}} + \text{timer1}$.
La variable booléenne timer1 représente ici le fait que le délai de garde vient d'être franchi ou non.

Remarque 7: ici, les soumissions de requêtes qui ne sont pas vues instantanément sont perdues.

L'état initial de ce processus (et des signaux qui l'entourent) est le suivant[3]:

```
P = 0
req = 0
nb = 0
envoi = K_in_drv = K_in_eff = (VIDE,0)
K_in_evt = 0
L_out_eff = VIDE
L_out_evt = 0
timer1 = 0
```

Définition d'un ordre sur les variables :
Les variables entières (et assimilées) ont un index inférieur aux variables booléennes, et l'ordre est total.
index (nb) < index(envoi) < index(K_in_drv) < index(K_in_eff) < index(L_out_eff) < index(P) < index(req) < index(K_in_evt) < index(L_out_evt) < index(timer1) < index(0) < index(1)

---

[2]Pour les signaux, les suffixes _drv, _eff et _evt désignent respectivement les valeurs pilotée (cas de signaux à un seul pilote et affectation lors du delta cycle suivant), effective et les événements qui leurs sont associés.
[3]Lors de la phase d'élaboration, VHDL affecte automatiquement la valeur la plus à gauche de l'ensemble de définition des types énumérés (booléens et entiers compris). Les valeurs effectives sont égales aux valeurs pilotées (dans le cas d'un pilote unique), il n'y a pas d'événement, le compteur ordinal pointe sur la première instruction suivant le mot-clé begin.

le D3 correspondant à l'état initial est représenté à gauche sur la figure suivante. La structure du graphe reprend bien la définition d'un VHDL-D3 : les variables entières sont vers la racine, et implicitement, seules les valuations des variables support de chemins menant vers la feuille 1 sont représentées. Les variables booléennes sont rassemblées vers les feuilles du graphe, et forment un BDD (ou une forêt de BDD) classique(s), dans lequel les chemins menant à 0 et 1 sont explicitement représentés.

Les états représentant "qu'un message à été acquitté et que le second vient d'être envoyé" est le suivant :

```
P = 1
req = 0 ou 1  (le signal peut être retombé ou non, il peut même être
              retombé puis être remonté, signalant une autre requête, qui
              ne sera pas prise en compte à ce niveau)
nb = 2        (on envoie le 2e message)
envoi = K_in_drv = K_in_eff = (MSG,2)
K_in_evt = 1
L_out_eff = VIDE (le récepteur n'a pas encore acquitté le message)
L_out_evt = 0
timer1 = 0 (on suppose le délai non nul)
```

Ces deux états sont représentés par le D3 de droite sur la Figure 2.

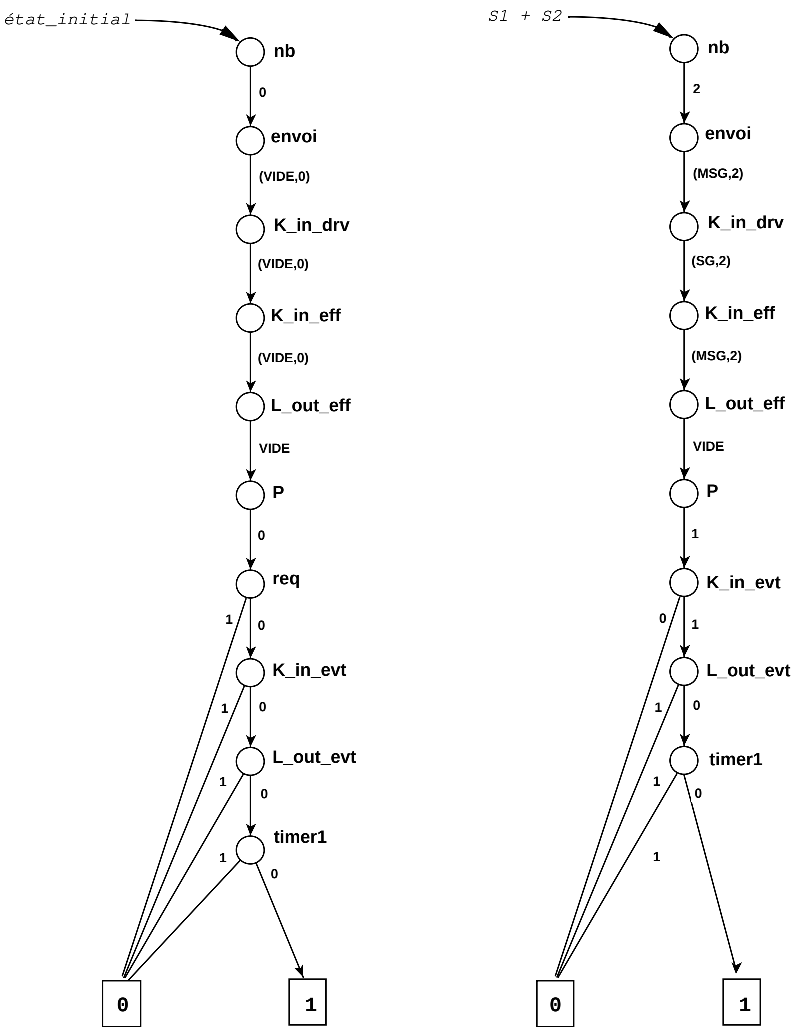

Figure 2. Représentation de l'état initial (à gauche) et d'un ensemble d'états (à droite) relatifs au programme VHDL.

3.2. Les opérations sur les VHDL-D3

3.2.1. Opérations ensemblistes

Les opérations ensemblistes sont réalisées par application de l'opérateur booléen défini sur les D3. La définition est néanmoins précisée en ce sens qu'elle fait référence à l'ordre des variables, et traite le cas des variables muettes (n'apparaissant pas le long de tous les chemins). En fait, elle reprend la définition de la fonction APPLY [Bryant 86] étendue à des nœuds n-aires.

Définition 15: application d'un opérateur op (vérifiant 0 op 0 = 0): d1 op d2 = d3
- si d1 = 0 alors d3 = (si 0 op 1) alors d2 sinon 0,
- si d2 = 0 alors d3 = (si 1 op 0) alors d1 sinon 0,
- si d1 = d2 = 1 alors d3 = 1 op 1,
- si d1 = (e1, $a1(t)_{t\in \mathrm{Dom}(e1)}$) et d2 = (e2, $a2(t)_{t\in \mathrm{Dom}(e2)}$) alors

  d3 = si (index(e1) < index(e2)) alors (e1, $\Sigma_{t\in \mathrm{Dom}(e1)}$ (a1(t) op d2))
  sinon, si (index(e1) > index(e2)) alors (e2, $\Sigma_{t\in \mathrm{Dom}(e2)}$ (d1 op a2(t)))
  sinon, (e1, $\Sigma_{t\in \mathrm{Dom}(e1)}$ (a1(t) op a2(t))

Les opérations ensemblistes sur les VHDL-D3 se déduisent de cette définition, en particularisant l'opérateur op :
- union : d3 = d1 $\cup$ d2 = APPLY(or, d1, d2)
- intersection : d3 = d1 $\cap$ d2 = APPLY(and, d1, d2)
- différence : d3 = d1 \ d2 = APPLY(function a b $\rightarrow$ a and (not b), d1, d2)
- différence symétrique: d3 = d1 \ d2 $\cup$ d2 \ d1

L'appartenance d'un élément à un ensemble ne présente aucune difficulté.

Définition 16: soit un D3 d définissant la fonction caractéristique χ_E d'un ensemble E inclus dans un univers plus vaste Ω. e de Ω est un élément de E ssi $\chi_E(e) = 1$.

Le mode opératoire est également très simple ; il revient à rechercher dans d s'il existe un chemin, dont les variables ont les valeurs définissant l'élément, menant de la racine au nœud terminal 1.

L'égalité de deux ensembles est également triviale. Puisque les ensembles sont représentés par des VHDL-D3 qui sont canoniques et implantés sur des arbres partagés, les ensembles sont égaux si et seulement si ils sont représentés par le même VHDL-D3.

Proposition 7: Soient E et E' deux ensembles, caractérisés respectivement par les fonctions χ_E et $\chi_{E'}$, elles même représentées par des VHDL-D3 d et d', alors E = E' $\Leftrightarrow \chi_E = \chi_{E'} \Leftrightarrow$ d = d'.

3.2.2. Opérations de sélection et d'affectation sur les VHDL-D3

La sélection permet de construire un D3 sous-ensemble d'un autre D3. L'affectation permet de construire de nouveaux ensembles d'états à partir d'ensembles antérieurs dans lesquels on a affecté de nouvelles valeurs à certaines variables d'états.

Définition 17: Soit A $\subseteq$ E, une expression de sélection est de type A $\rightarrow$ Bool.

Définition 18: Soit *exp* une expression de sélection sur E et *d* un D3 sur E. L'opération de sélection *sel(exp,d)* est définie telle que :

- si *exp* = VRAI, *sel(exp,d)* = d
- si *exp* = FAUX, *sel(exp,d)* = 0
- si d = 0, *sel(exp,d)* = 0
- si d = $(e, a(t)_{t\in Dom(e)})$, *sel(exp,d)* = $(e, (sel(exp|_{e\leftarrow t}, a(t))_{t\in Dom(e)})$

Définition 19: Soit A $\subseteq$ E, une expression d'affectation de e $\in$ E est de type : A -> Dom(e)

Définition 20: Soit *exp* une expression à affecter à une variable *x* sur un D3 *d*, cette opération d'affectation est notée *aff(exp,x,d)* et définie telle que :

- si d = 0, *aff(exp,x,d)* = 0
- si d = $(e, a(t)_{t\in Dom(e)})$ et x •e, *aff(exp,x,d)* = $(e, (aff(exp|_{e\leftarrow t}, a(t))_{t\in Dom(e)})$
- si d = $(x, a(t)_{t\in Dom(e)})$, *aff(exp,x,d)* = $(x, a(t'))_{t'\in Dom(e) \text{ et } t' = eval(exp|_{x<-t}, a(t))}$)

La fonction *eval(exp,d)* retourne les valeurs de l'expression *exp* évaluée dans les environnements représentés par *d*. L'évaluation se fait par substitution des variables de l'expression par leur valeur lors du parcours du D3 et application des opérateurs, jusqu'à ce que l'expression soit réduite à une valeur de son co-domaine. Une définition plus rigoureuse est donnée ci-après.

Définition 21: *eval(exp,d)* avec d $\neq$ 0 =

Si d = 1 alors exp /* elle est supposée ne plus contenir de variable libre */

Si d = $(e, a(t)_{t\in Dom(e)})$, alors $\cup_{t\in Dom(e)}$: $red\,(eval(exp|_{e\leftarrow t}, a(t))$

Note : la fonction *red(exp)* applique les opérandes ayant une valeur aux opérateurs dans l'expression.

Exemple : soit à affecter l'expression « *x + y + z* » à la variable *x* dans le D3 *d* décrit sur la Figure 3. Le D2O correspondant à cette affectation ainsi que le D3 résultant sont représentés.

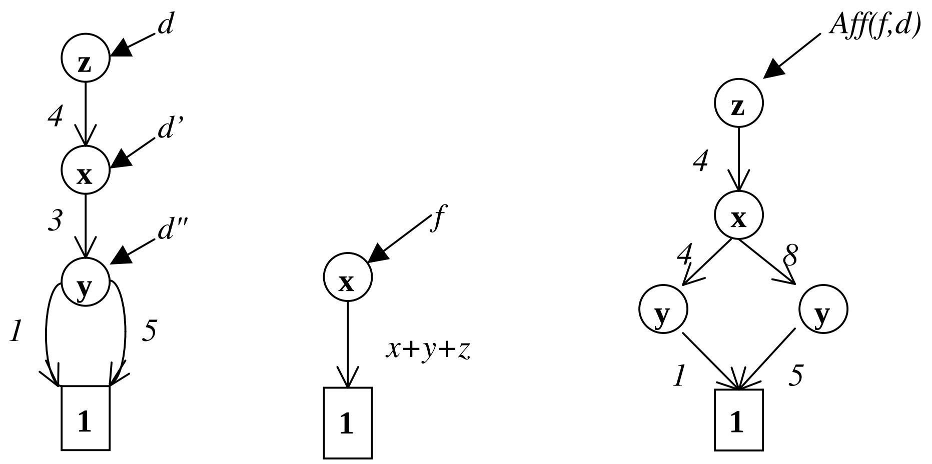

Figure 3. Affectation de nouvelles valeurs à certaines variables d'un D3.

3.2.3. D2O représentant la relation de transitions

Les relations de transitions sont une partie d'un automate décrivant les changements d'états de ce dernier. Nous définissons tout d'abord les modèles d'automates qui nous intéressent, puis nous définissons le D2O d'une transition, et montrons que le D2O représentant la relation de transitions dans son ensemble est une union ensembliste des D2O de chacune des transitions.

Définition 22: un automate déterministe A est un tuple $<S,I,O,\delta,\lambda,so>$ dont chaque composante est définie tel que :

- S : ensemble fini d'états
- I : ensemble fini des configurations d'entrée
- O :ensemble fini des configurations de sortie
- δ : fonction de transition : $S \times I \rightarrow S$
- λ : fonction de sortie : $S \times I \rightarrow O$
- so : état initial

NB : il s'agit ici de machine de Mealy, les machines de Moore ont une fonction de sortie de signature $S \rightarrow O$.

Ainsi, les variables composant un état VHDL, tel que nous l'avons défini dans la section précédente, se décomposent en trois ensembles : les variables d'entrée, qui sont uniquement lues, les variables de sortie, qui sont uniquement écrites, et les variables d'état, représentant la mémoire du programme, et qui peuvent être lues et écrites. La partition de l'ensemble des variables en ces trois parties est réalisée par une analyse statique du programme.

Définition 23: soit E_{var} l'ensemble des variables composant un état VHDL,

- $E_{var} = E_{var\_in} \cup E_{var\_out} \cup E_{var\_mem}$
- $E_{var\_in} \cap E_{var\_out} = \emptyset$, $E_{var\_in} \cap E_{var\_mem} = \emptyset$, $E_{var\_out} \cap E_{var\_mem} = \emptyset$
- $E_{var\_in} = \{x \in <S,V,P>$ tq x est lue uniquement$\}$
- $E_{var\_out} = \{x \in <S,V,P>$ tq x est écrite uniquement$\}$
- $E_{var\_mem} = \{x \in <S,V,P>$ tq x est lue et écrite$\}$

Définition 24: une configuration c sur un ensemble X est un mot composé de la concaténation (au sens de concaténation de chaînes de caractères et non de D3 ou de D2O), notée •, d'une valeur de chaque élément de X. On appelle C(E) l'ensemble des configurations sur E.

Soit $x \in X$, $val(x) \in Dom(x)$, $c = \bullet_{x \in X} val(x)$

$C(X) = \{c$ tq c est une configuration de $X\}$

Il s'ensuit que la définition de l'automate A peut être particularisée en précisant la définition de I, O et S :

- $I = C(E_{var\_in})$
- $O = C(E_{var\_out})$
- $S = C(E_{var\_mem})$

3.2.3.1. cas synchrone

La fonction de transitions δ est un ensemble de fonctions affectant obligatoirement une nouvelle valeur (éventuellement identique à la précédente) à chaque élément de E_{var_mem}.

$\delta : \{f_i : \forall v_i \in E_{var\_mem}, val(v_i) = f_i(C(E_{var\_mem}),C(E_{var\_in}))\}$

Remarque 8: la fonction f_i peut être représentée par une fonction gardée : dans ce cas, elle est composée d'une garde γ_i et d'une partie opérative ϕ_i. La partie opérative n'est appliquée à son argument que si la garde est vraie avant l'application de la partie opérative. De fait, f_i peut être réécrite en : $f_i =$ si γ_i alors ϕ_i sinon Id.

Chaque manipulation de v_i est représentable par un D2O : $d2o(v_i) = v_i\text{-}f_i \rightarrow 1$.

Le D2O représentant la fonction de transitions δ est alors réalisée par une concaténation de D2O :

$d2o(\delta) : v_1\text{-}f_1 \rightarrow v_2\text{-}f_2 \rightarrow \dots v_n\text{-}f_n \rightarrow 1\ \forall v_i\ , i \in [1..n]$, tq $v_i \in E_{var\_mem}$.

De façon similaire, le D2O représentant la fonction de sortie λ est défini par :

$\lambda : \{g_i : \forall\ v_i \in E_{var\_out}, val(v_i) = g_i(C(E_{var\_mem}),C(E_{var\_in}))\}$

$d2o(\lambda) : v_1\text{-}f_1 \rightarrow v_2\text{-}g_2 \rightarrow \dots v_n\text{-}g_n \rightarrow 1\ \forall v_i\ , i \in [1..n]$, tq $v_i \in E_{var\_out}$.

3.2.3.2. cas asynchrone

Dans ce cas, chaque transition de δ n'affecte pas nécessairement toutes les variables de E_{var_mem}, mais seulement un sous-ensemble. δ est alors l'union de toutes les transitions du modèle.

On nomme P(E) l'ensemble des parties de E.
Le franchissement d'une transition t est modélisé par la fonction ft définie telle que :
$f_t : \forall\ v_i \in E'_t \subseteq P(E_{var\_mem}), (\bullet_{vi \in E'}\ (val(v_i) = f_i(C(E_{var\_mem}),C(E_{var\_in})))$

Ici, t affecte un sous-ensemble des variables de E_{var_mem}, et la fonction f_t retourne la configuration des valeurs des variables affectées (la configuration étant obtenue par concaténation des symboles, au sens de la concaténation de chaîne de caractères).

Comme précédemment, t peut être représentée par un D2O qui est la concaténation des D2O représentant les modifications de chacune des variables affectée par la transition.

$d2o(t) : v_1\text{-}f_1 \rightarrow v_2\text{-}f_2 \rightarrow \dots v_n\text{-}f_n \rightarrow 1\ \forall v_i\ , i \in [1..n]$, tq $v_i \in E'_t$

Remarque 9: la fonction f_i peut être représentée par une fonction gardée : dans ce cas, elle est composée d'une garde γ_i et d'une partie opérative ϕ_i. La partie opérative n'est appliquée à son argument que si la garde est vraie avant l'application de la partie opérative. De fait, f_i peut être réécrite en : $f_i =$ si γ_i alors ϕ_i sinon 0.

Le D2O de δ correspond alors à l'union des D2O des transitions qui la constituent :

$d2o(\delta) = \Sigma_{t \in T\ fini}\ d2o(t)$

3.2.4. Exemple du programme précédent

L'automate représentant le processus `sender` de l'exemple précédent est donné ci-après.

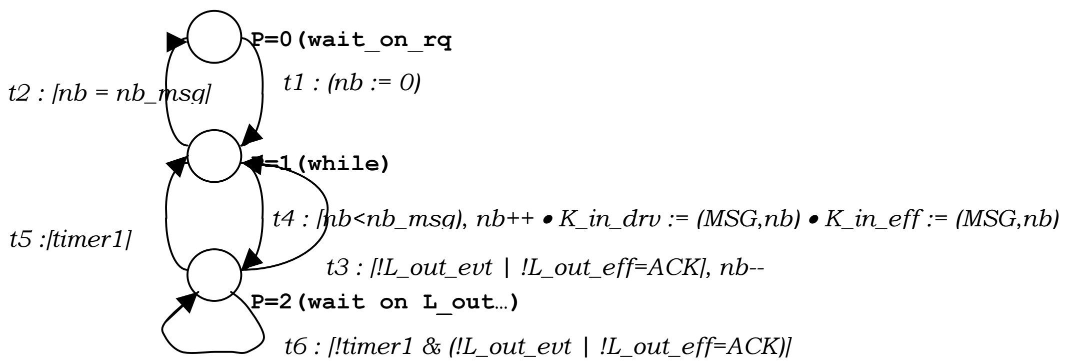

Figure 4. Automate correspondant au processus `sender` du programme VHDL.

(les crochets [] indiquent l'expression de garde, le point • représente la composition d'opérations à effectuer, due au regroupement d'instructions séquencées).

Le D2O de cet automate est représenté sur la Figure 5 :

L'ordre n'est pas identique à celui défini pour les D3, mais ce n'est pas problématique : l'utilisation d'un même ordre est utile pour comparer deux D3. La non imposition d'un ordre pour le D2O se justifie car le D2O représente la composition de fonctions opérant sur des variables le long des chemins d'un D3 ; de fait, l'évaluation de chaque fonction peut nécessiter des parcours le long de chemins dans le D3. (Il est clair que si toutes les gardes et fonctions ont pour argument la variable courante, alors le parcours pourra être optimisé si les ordres d'occurrences des variables dans le D3 et le D2O sont identiques, et ce parfois au détriment du partage des nœuds dans le D2O).

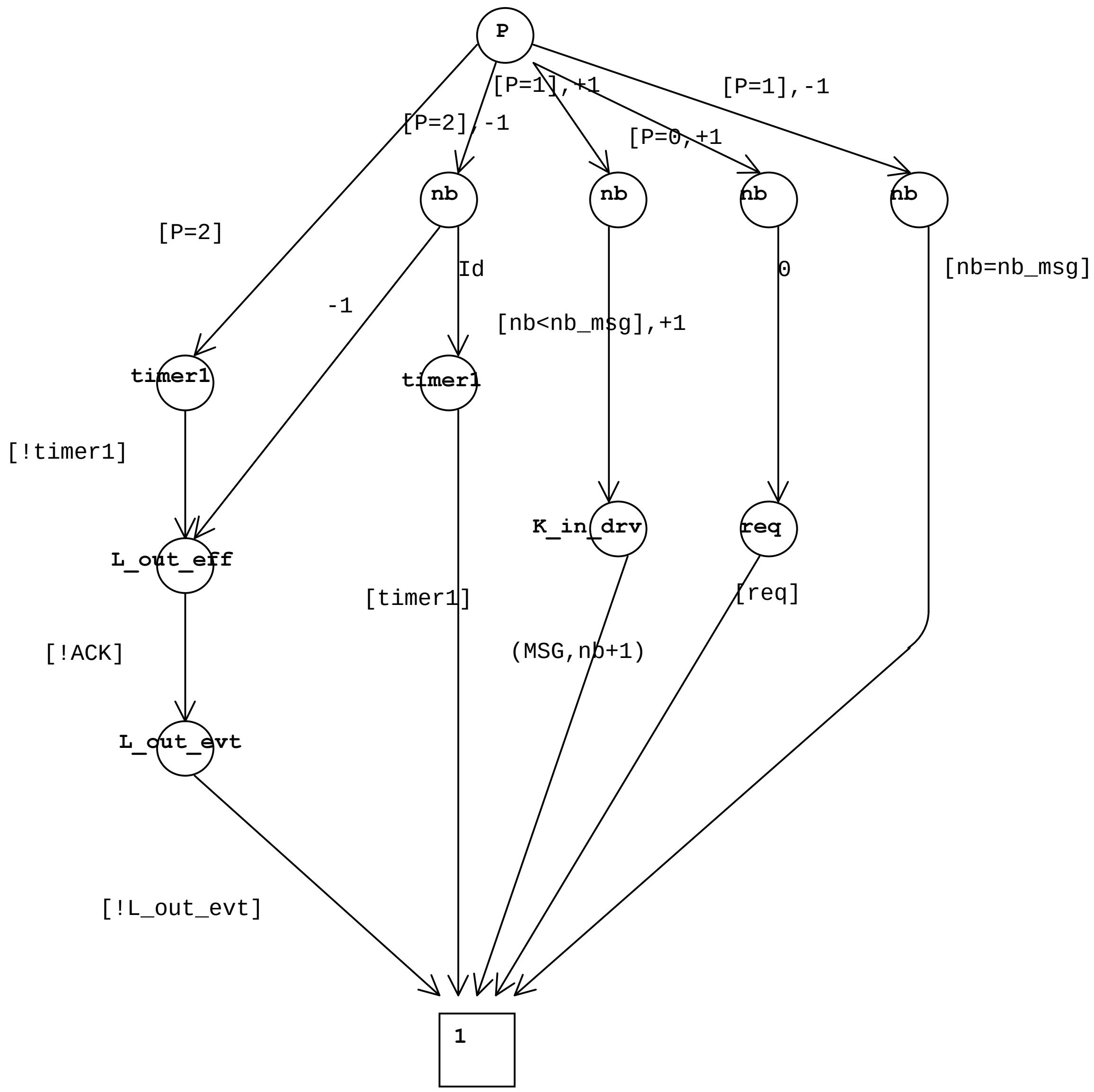

Figure 5. Le D2O de l'automate de la figure 4.

4. ADAPTATION DES ALGORITHMES DE VERIFICATION

Cette section rappelle les algorithmes de vérification de propriétés CTL, apparaissant comme des manipulations d'ensembles, représentés par des fonctions booléennes, et montre comment étendre cette démarche en s'appuyant sur les D3 et D2O définis précédemment.

4.1. Principe de fonctionnement de la vérification par modèle symbolique

La vérification symbolique se base sur la représentation par point fixe des opérateurs CTL:

$\exists \mathbf{G}(p) = p \vee \exists \mathbf{X} \exists \mathbf{G}(p))$ initialisée avec VRAI

$\exists \mathbf{U}(p,q) = q \wedge (p \vee \exists \mathbf{X}(\exists \mathbf{U}(p,q))$ initialisée avec FAUX

Remarque 10: Les autres opérateurs peuvent se réécrire en $\exists \mathbf{G}$ et $\exists \mathbf{U}$.

La vérification de modèle symbolique est fondée sur le calcul de point fixe des opérateurs de CTL. Ce calcul manipule des ensembles d'états, ainsi que des opérateurs sur ces ensembles, notamment le calcul des ensemble d'états prédécesseurs et successeurs d'un ensemble d'états donné. Il s'agit alors d'adopter une bonne représentation symbolique des ensembles d'états, sur laquelle les opérations correspondants au calcul des prédécesseurs et successeurs est aisée.

De fait, les ensembles d'états et les relations ou fonctions de transitions sont représentées par des agencements de fonctions caractéristiques, elles-mêmes représentées par des BDD. Les calculs de successeurs et prédécesseurs revient alors à des manipulations de fonctions booléennes.

4.1.1. Représentation d'ensemble d'états d'un automate

Un ensemble d'états $E \subseteq S$ est représenté par sa fonction caractéristique $\chi_E : S \rightarrow B$, définie telle que :

$e \in E \Leftrightarrow \chi_E(e) = 1.$

En pratique, S est fini et l'on définit V un ensemble de variables booléennes = $\{v_0, v_1, ...v_{n-1}$ tq $\lceil \log_2|S| \rceil \leq n \}$ et une fonction de codage $h : S \rightarrow B^n$ permettant de coder les états de S.
De ce fait, la fonction caractéristique de E est redéfinie sur V et non plus sur E : $\chi_E : B^n \rightarrow B$

$e \in E \Leftrightarrow \chi_E(h(e)) = 1.$

4.1.2. Représentation de la relation de transition

Soit T une relation de transitions $\subseteq S \times S$. La fonction caractéristique de T , $\chi_T : S \times S \rightarrow B$, est définie telle que : $(e,e') \in T \Leftrightarrow \chi_T(e,e') = 1.$

En pratique, on introduit un second jeu de variables $V' = \{v'_0, v'_1, ...v'_{n-1} \mid \lceil \log_2|S| \rceil \leq n \}$ afin de pouvoir distinguer les états courants des états futurs, et la fonction caractéristique de T est redéfinie en fonction de ce codage booléen : $\chi_T : B^n \times B^n \rightarrow B$, telle que $(e,e') \in T \Leftrightarrow \chi_T(h(e),h(e')) = 1$

Remarque 11 : De fait, la construction de la relation de transitions peut être prohibitive, car χ_T peut être représenté par un gros BDD, ou alors le BDD final n'est pas trop gros mais les BDD intermédiaires eux le sont. En pratique, χ_T n'est généralement pas représentée par un unique BDD monolithique, mais par une conjonction implicite de BDD, c'est ce qu'on appelle la relation de transition partitionnée

Remarque 12: Dans la suite, par abus de langage et pour simplifier les notations, E et χ_E seront confondus, E(v) désignera l'ensemble E codé sur les variables courantes, E(v') désignera l'ensemble E codé sur les variables futures, χ_T (h(e),h(e')) sera noté R(v,v') (c'est la relation liant des états codé sur les variables courantes à des états codés sur les variables futures).

4.1.3. Représentation de la fonction de transition

Soit δ une fonction de transitions : $S \to S$.
S étant fini, on définit comme précédemment un ensemble de variables booléennes V = {v_0, v_1, ...v_{n-1} tq $\lceil \log_2|S| \rceil \leq n$ } et une fonction de codage h : $S \to B^n$ permettant de coder les états de S.
δ est alors redéfinie sur $B^n \to B^n$.
δ peut ainsi être représentée par un vecteur de fonctions $\delta = [\delta_0,\delta_1,...\delta_{n-1}]$, dont la i-ème position δ_i = $B^n \to B$ représente l'affectation de la variable v_i en fonction de variables de V à l'instant précédent.

Remarque 13: Contrairement à la représentation d'une relation de transitions quelconque qui nécessite deux jeux de variables pour distinguer les variables futures (écrites) des variables courantes (lues), les variables affectée par le franchissement d'une transition sont implicitement repérées par leur position dans le vecteur de fonctions δ.

Remarque 14: Avec cette représentation par vecteur de fonctions, on s'intéresse principalement aux systèmes présentants un fonctionnement synchrone : pour chaque transition de δ, toutes les variables d'états sont affectées (même si leur valeur reste inchangée). De plus, cette représentation est limitée aux systèmes déterministes, du fait même que δ est une composition synchrone de fonctions.

4.1.4. Calcul des successeurs et des prédécesseurs d'un ensemble d'états avec une relation de transitions [Mc Millan 92]

Passer d'un ensemble d'états (From) à l'ensemble des états successeurs (To) revient à calculer *l'image* de From:

$$To(v') = \exists_v \, (R(v,v') \wedge From(v')_{|v' \leftarrow v})$$

L'énumération des états et de leurs successeurs est ramenée à trois opérations booléennes :
la substitution de variables, le ET logique, et la quantification existentielle : $\exists_{xi} f = f_{|xi\leftarrow 1} + f_{|xi\leftarrow 0}$

Le calcul des prédécesseurs (From) d'un ensemble d'états (To) se déduit directement de l'équation précédente:

$$From(v) = \exists_{v'} \, (R(v,v') \wedge To(v)_{|v \leftarrow v'})$$

4.1.5. Calcul des successeurs et des prédécesseurs d'un ensemble d'états avec une fonction de transitions [CBM 89]

O. Coudert, J.C. Madre et R. Berthet ont proposé dans [CBM 90] un algorithme de calcul de l'image d'un espace d'états, nommé "techniques récursive de résolution de contraintes", que nous allons exposer brièvement ici. Cette méthode, plus difficile à appréhender, présente néanmoins l'avantage d'éviter l'utilisation de deux jeux de variables pour construire et manipuler la relation de transitions.

L'opérateur de contrainte est un opérateur permettant de simplifier une fonction booléenne *f* sous une contrainte donnée *c*.

Définition 25: Soient f et c ($c \neq$ FAUX) deux prédicats, le résultat de l'opérateur de contraintes appliqué à f et c, noté $f{\uparrow}c$, est un prédicat vérifiant : $c \Rightarrow (f \Leftrightarrow f{\uparrow}c)$

Cet opérateur se définit récursivement .

```
constrain (f,c)
     si f = VRAI ou f = FAUX alors f
     sinon si c = VRAI alors retourner f
          sinon, ∃ une variable x appartenant au support de f ou de c :
               si (c|x←0 = FAUX) alors constrain(f|x←1,c|x←1)
               sinon, si (c|x←1 = FAUX) alors constrain(f|x←0,c|x←0)
                    sinon (x.constrain(f|x←1,c|x←1) + !x.constrain(f|x←0,c|x←0))
```

Cette définition peut être utilisée pour calculer l'opération de contrainte de manière univoque si les variables sont ordonnées. De plus, la définition de `constrain` peut être étendue aux vecteurs de fonctions. En effet, si $f = [f_1,...f_n]$, $f{\uparrow}c = [\, f_1{\uparrow}c,...,f_n{\uparrow}c \,]$

Théorème : Soit $A \subseteq B^n$, l'image de A par la fonction $\delta = [\delta_1,\delta_2,...\delta_n]$, appelée $\delta(A)$, est égale au co-domaine de $\delta{\uparrow}A$.

De plus, l'opérateur R "range-restrictor" calcule récursivement le co-domaine d'un vecteur de fonctions : soit $\delta = [\delta_1,\delta_2,...\delta_n]$, dont on cherche le co-domaine.

```
R(δ) =
     si δ = [δn] alors
          si δn = VRAI alors vn
          sinon, si δn = FAUX alors !vn
          sinon, VRAI   /* vn + !vn */
     sinon  /* δ = [δk,...δn] */
          si δk = VRAI alors vk.R([δk+1,...,δn])
          sinon, si δk = FAUX alors !vk. R([δk+1,...,δn])
          sinon, vk.R( [δk+1,...δn] ↑ δk) + !vk.R( [δk+1,...δn] ↑ !δk)
```

Symétriquement calculer les prédécesseurs par $\delta = [\delta_1,\delta_1,...\delta_n]$ d'un ensemble d'états A revient à déterminer $\delta^{-1}(A)$, c'est à dire $R^{-1}(A,\delta)$, qui reprend l'algorithme précédent :

```
R-1(A,δ) =
     si A = FAUX alors FAUX
     sinon, si δ = [δn] alors
               si A => vn alors δn,
               sinon, si A => !vn alors !δn,
               sinon, VRAI
          sinon, /* δ = [δk,...δn] */
               si A => vk alors δk. R(A|vk<-1, [δk+1,...,δn])
               sinon, si A => !vk alors !δk. R(A|vk<-0, [δk+1,...,δn])
               sinon, δk.R-1(A|vk<-1, [δk+1,...δn] ) + !δk.R-1(A|vk<-0, [δk+1,...δn] )
```

4.1.6. Algorithmes de parcours symbolique

Le parcours de l'espace d'états consiste à itérer ce calcul d'image, soit à partir de la méthode définie par [McMillan 92], soit par celle définie par [CBM 89], tant que de nouveaux états sont rencontrés.

```
Traverse(R,s0)

     Reached = From = s0
     faire
          To = Image(R,From)
          New = To - Reached
          From =  New
          Reached = Reached + New
     jusqu'à (New = Ø)
     retourner Reached
fin
```

L'évaluation des formules CTL (réécrites selon les trois opérateurs de base EX, EG et EU) suit l'algorithme présenté ci-après :

```
function Eval(p)
     case
          p : proposition atomique     => retourner p
          p = ¬ q                      => retourner ¬ Eval(q)
          p = r ∧ s                    => retourner Eval(r) ∧ Eval(s)
          p = ∃Xq                      => retourner EvalEX(q)
          p = ∃Gq                      => retourner EvalEG(q,true)
          p = ∃U(r,s)                  => retourner EvalEU(r,s,false)
     fin case
fin

function EvalEX(p)
     retourner pred(p)
fin
(avec pred (p) = ∃v' (R ∧ p|v ← v') dans le cas de [McMillan 93]
 et pred(p) = R-1(δ,p) dans le cas de [CBM 89])

function EvalEG(p,y)
     y' ← p ∧ EX(y)
     si y' = y alors
          retourner y
     sinon
          retourner EvalEG(p,y')
fin

function EvalEU(p,q,y)
     y' ← q ∨ ( p ∧ EX(y))
     si y' = y alors
          retourner y
     sinon
          retourner EvalEU(p,q,y')
fin
```

4.2. Adaptation des algorithmes de parcours avant et arrière

On a déjà vu dans la section 3 la définition d'un état d'un programme VHDL et sa représentation à l'aide des VHDL-D3, ainsi que la représentation de la fonction de transitions par un D2O.
Il nous reste à présenter les algorithmes de calcul d'image et de pré-image sur les VHDL-D3. En fait, ces algorithmes comprennent deux parties : la première traite des variables entières en appliquant pour chaque variable l'opération d'affectation définie en section 3.2.2, la seconde utilise le calcul d'image proposé par [CBM 90] (puisque le D2O de la partie booléenne peut être vue comme un vecteur de fonctions booléennes).

4.2.1. Calcul d'image

Définition 26: soit *d*, un VHDL-D3 représentant un ensemble d'états d'un programme et Θ, le D2O représentant la relation de transitions de ce dernier. L'image de *d* par Θ est définie opérationnellement de la façon suivante :

```
Img(d, Θ) :
     • d = 0 ou Θ = 0 : retourner 0
     • d = 1 : retourner 1
     • Θ = 1 : retourner d
     • d = (e,a(t)t∈Dom(e)), Θ= (e', (Oi,Θi)i∈I fini :
     /* les Oi représentent les expressions de la relation de transition à
     affecter à la variable e */

     /* PARTIE ENTIERE : */
     /* si e = e', on exécute Oi(t) (ce qui revient à réaliser une affectation),
     puis on propage le calcul d'image sur les sous-D3 a(t) */
          si e = e' et Dom(e) = Int alors
               retourner (e, (a(Aff(Oi,e,a(t)) ;Img(a(t),Θi))t∈Dom(e), i∈I fini

     /* PARTIE BOOLEENNE : */
     /* On applique le calcul de [CBM 89] rappelé en section 4.1.5 */
          si e = e' et Dom(e) = Bool alors
               retourner R(make_vector(Θ))

     /* sinon, on descend dans l'arborescence de d ou de Θ
          si index(e) < index (e'), retourner (e, Img(a(t)t∈Dom(e) , Θ)
          si index(e) > index (e'), retourner Img(d, Θi)i∈I fini
```

Remarque 15: il s'agit d'un parcours attelé des deux graphes, qui présuppose que les variables du D3 et du D2O sont rangées dans le même ordre. Comme il a déjà été précisé, le respect de cet ordre le long des D3 *et des D2O* n'est pas impératif, mais il simplifie la nature du parcours.

4.2.2. Calcul de pré-image

Il s'agit ici d'appliquer l'algorithme de calcul d'image non plus sur les opérations O_i mais leur inverse O_i^{-1}. Ceci suppose que l'on soit capable de déterminer O_i^{-1}. Si cela ne pose pas de problème pour les incrémentations, décrémentations, …, ou les fonctions booléennes (cf. algorithme de calcul de pré-image de [CBM 89]), le problème demeure dans le cas d'affectations à une valeur constante, ou d'expressions ne dépendant pas directement de la valeur antérieure de la variable affectée. Dans ces cas, l'opération O_i^{-1} réalise une projection sur la variable concernée, qui a, à l'issue de l'affectation, une valeur indéterminée. On représente ainsi un sur-ensemble des états prédécesseurs, ce qui, dans la pratique, est limité en contraignant les parcours arrières à l'ensemble des états accessibles calculé préalablement.

5. QUELQUES EXTENSIONS ENVISAGEES

Le modèle des VHDL-D3 actuel ne permet pas encore de représenter des pilotes de signaux ou des vecteurs. Les extensions que nous envisageons visent à pouvoir représenter ces éléments.

5.1. Le produit mixte : Int × Bool

Actuellement, les valuations des arcs contiennent uniquement des valeurs entières (ou couples de valeurs entières) pour la partie compteur, et les valuations binaires pour la partie booléenne. Il peut être intéressant de pouvoir représenter et manipuler des produits mixtes (dont certains champs sont entiers et d'autres sont booléens). Cette adaptation nécessite quelques aménagements des VHDL-D3 : les nœuds des BDD sont une forme particulière de D3 (on ne représente plus les chemins menant au nœud terminal 0), sur lesquels on applique les algorithmes d'évaluation des D3 et non des BDD. De fait, on peut s'affranchir de l'ordre total imposé aux variables, stipulant que les variables compteur sont d'index inférieur aux variables booléennes.

Une application immédiate de cette représentation est le pilote d'un signal VHDL. Un pilote est une liste de couples (valeur –entière ou booléenne-, date de prise en compte), ordonnée selon les dates de prise en compte croissante.

La représentation de tels objets nécessite quelques aménagements sur les algorithmes de parcours avant et arrière, notamment l'ajout de fonctions permettant la représentation de l'avancée du temps réel, à savoir des recherches de minimum sur des ensembles de valeurs entières, des affectations de champs de couples dans des listes de couples, …

5.2. Le vecteur

Actuellement, les vecteurs (ou tableaux n-dimensionnels) sont déstructurés : chaque élément du tableau est une variable distincte, et les attributs du tableau sont une combinaison des attributs de chacune de ses cases (par exemple, l'attribut 'event d'un tableau est représenté par la disjonction des attributs 'event de chaque case du tableau). Il pourrait être intéressant de disposer dans les VHDL-D3 de variables de type vecteur, dont les valuations des arcs sont des listes de valeurs des cases du tableau représenté. Il est alors nécessaire de disposer de l'opérateur d'indexation permettant d'accéder à un élément particulier du tableau.
Ces structures nécessitent également quelques extensions des algorithmes de parcours : l'affectation groupée de tous les éléments, l'affectation d'un élément d'un vecteur, la lecture d'un élément d'un vecteur…

6. CONCLUSION

Nous avons présenté une première définition des VHDL-D3, une structure de données permettant de représenter des programmes VHDL manipulant simultanément des objets entiers et booléens, basée sur des arbres partagés n-aires. Nous avons donné une représentation des ensembles d'états manipulés ainsi que de la relation de transitions du programme. Nous avons également montré comment adapter les algorithmes de vérification symbolique, basés sur les BDD, à nos structures. Un premier prototype a été réalisé. S'il ne nous a pas encore permis d'évaluer les performances de la représentation que nous proposons vis à vis de représentations purement booléennes, il nous a permis de jeter les bases de ces VHDL-D3.

7. REFERENCES

[Bryant 86] *Graph-based algorithms for boolean function manipulation*. R. Bryant. IEEE Trans on computers, Vol C-35, N°8, 1986.

[BC 95] *Verification of arithmetic circuits with Binary Moment Diagrams*. R. Bryant, Y. Chen. 32th Design Automation Conference, San Fransisco, USA, 1995

[Mc Millan 92] *Symbolic Model Checking*. K. Mc Millan. Kluwer Academics Publishers 1992.

[CBM 89] *Verification of sequential machines using bolean functional vectors*. O. Coudert, C. Berthet, J.C. Madre. IFIP Int Workshop on Applied Formal Methods for Correct VLSI Design, L. Claesen editor, Louvin, Belgique, 1989, North Holland.

[LPV 96] *Formal verification using edge-valued binary decision diagrams*. Y. Lai, M. Pedram, S. Vrudhula. IEEE Trans. On Computers, Vol 45, N°2 (1996), pp 247-255.

[DB 98] *Ordered Kronecker Functional Decision Diagrams, A data structure for representation and manipulation of boolean functions.* R. Dreshler, B. Becker. IEEE Trans. On Computer-Aided Design of Integrated Circuits and Systems, Vol 17, N°5, pp 457-459, 1998.

[CMZFY 93] Spectral Transforms for large boolean functions with application to technologie mapping. E. Clarke, K. McMillan, X. Zhao, M. Fujita, J. Yang, Design Automation Conference, Dallas, USA, 1993

[RK 00] Analysing Real-Time Systems. J. Ruf, T. Kropf. Design, Automation and Test in Europe, Paris, France, 2000

[CCM 97] The verus tool : a quantitative approach for the formal verification of real-time systems. S. Campos, E. Clarke, M. Minea. Int Conf on Computer Aided Verification, 1997

[HNSY 92] Symbolic Model Checking of Real-time systems. T. Henzinger, X. Nicollin, J. Sifakis, S. Yovine. Int Conf on Logics in Computer Sciences, 1992.

[CGS 98] Diagrammes de décision pour la vérification de réseaux à files. J.-M. Couvreur, A. Griffault, D. Sherman. Journée « Vérification » du LSV, Cachan, France 1998

[HP 94] Computing Binary Decision Diagrams for VHDL Data Types. R. Herrmann, H.Pargmann,G. Int Conf EURO-DAC'94, Grenoble, France, 1994.