

THESE
présentée pour obtenir le titre de Docteur
de l'Ecole Nationale Supérieure
des Télécommunications

Olivier COUDERT

**SIAM : Une Boite à Outils Pour la
Preuve Formelle
de Systèmes Séquentiels**

Thèse soutenue le 10 Octobre 1991 devant le jury composé de :

François ANCEAU	Président
Gérard BERRY Jean Claude RAOULT	Rapporteurs
Edmund M. CLARKE Michel DANA Gary D. HACHTEL Gérard HUET	Examineurs

Remerciements

Cette thèse a été effectuée au Centre de Recherche BULL de Septembre 1988 à Octobre 1991. Je remercie François ANCEAU, directeur de cette division, de m'avoir encouragé dans ce travail, et d'avoir présidé le jury.

Je remercie particulièrement Gérard BERRY, non seulement pour m'avoir fait l'honneur d'être rapporteur de cette thèse, mais aussi pour les excellentes relations que j'ai entretenues avec lui lorsque nous avons travaillé ensemble. Je tiens vivement à le remercier pour avoir fait connaître nos travaux.

Je remercie Jean Claude RAOULT pour avoir été rapporteur de cette thèse, et pour avoir fait la chasse aux anglicismes.

Edmund M. CLARKE et Gary D. HACHTEL ont fait le déplacement des U.S.A. pour me faire l'honneur de participer au jury. Je les en remercie spécialement, ainsi que pour la reconnaissance de nos travaux aux U.S.A. à laquelle ils ont contribué.

Je remercie Michel DANA pour sa participation au jury, et pour son point de vue de concepteur de circuits sur le manuscrit.

Je remercie vivement Gérard HUET de m'avoir fait l'honneur de prendre part au jury, et pour m'avoir conseillé dans la version finale du manuscrit.

Je remercie spécialement Jean Paul BILLON, car je dois à son enthousiasme d'être venu au Centre de Recherche BULL. Je remercie Christian BERTHET qui m'a supporté (mot que j'affectionne pour son double sens) au début de cette recherche.

Je remercie Jean Christophe MADRE, avec qui SIAM a été développé, pour la lecture attentive du manuscrit qu'il a dû subir, mais surtout pour l'ambiance de travail particulière (et appréciée de tous...) que j'ai pu *heureusement* trouver avec lui.

Enfin, je remercie les nombreuses personnes qui ont dû supporter mon humeur (toujours également champêtre, tolérante, voire laxiste) et mes écarts (de langage et de conduite) sans trop oser me le faire remarquer. C'est surtout sur ce dernier point que je leur suis gré...

A mes parents

A Toi, si délicieuse...

Table des matières

Introduction	9
Motivations industrielles	9
La vérification formelle de matériel	10
Spécification du problème	12
Plan de lecture	12
Notre contribution	13
I Logique propositionnelle quantifiée	15
1 Logique propositionnelle quantifiée	17
1.1 Formules propositionnelles quantifiées	17
1.1.1 Syntaxe des formules propositionnelles quantifiées	17
1.1.2 Sémantique des formules propositionnelles quantifiées	18
1.1.3 Égalité sur les formules propositionnelles	18
1.1.4 Formules propositionnelles et fonctions booléennes	19
1.2 Élimination des quantificateurs	20
1.2.1 Élimination gauche-droite des quantificateurs	20
1.2.2 Élimination droite-gauche des quantificateurs	21
1.3 Tautologie, antilogie, satisfiabilité	21
1.4 Résolution d'équations	22
1.4.1 Résolution d'équations à une inconnue	23
1.4.2 Résolution d'équations à plusieurs inconnues	25
1.5 Conclusion	26
2 Représentation des formules propositionnelles	27
2.1 Introduction	27
2.2 Expansion de Shannon	28
2.2.1 Propriétés de l'expansion de Shannon	29
2.2.2 Arbre de Shannon	29
2.3 Formes canoniques graphiques	31
2.3.1 Élimination des noeuds redondants	31
2.3.2 Graphes de décision binaires	38
2.3.3 Graphes de décision typés	43
2.4 Manipulations de graphes de décision	48
2.4.1 Graphe de décision d'une formule	48
2.4.2 Combinaisons de graphes de décision	48
2.4.3 Composition de graphes de décision	51

2.4.4	Forme sans quantificateur d'un graphe quantifié	55
2.4.5	Résolution d'équations	57
2.5	Le problème de l'ordre	58
2.6	Comparaison des graphes avec d'autres représentations	60
2.6.1	Graphes et sommes de produits	60
2.6.2	Graphes et sommes exclusives de produits	62
2.6.3	Verdict : avantage aux graphes de décision!	65
2.7	Conclusion	66
II Preuve de systèmes séquentiels		67
3	Problèmes sur les machines séquentielles	69
3.1	Modèle d'une machine séquentielle	69
3.2	Comparaison de machines séquentielles	71
3.2.1	Critères de comparaison	72
3.2.2	Algorithme de comparaison	72
3.2.3	Le terme critique : l'image	74
3.3	Vérification de propriétés temporelles	74
3.3.1	Syntaxe et sémantique de CTL	75
3.3.2	Algorithme de vérification de formules CTL	76
3.3.3	Le terme critique : l'image réciproque	78
3.4	Minimisation de machines séquentielles	79
3.4.1	Minimisation de la logique combinatoire	80
3.4.2	Élimination des variables d'états redondantes	80
3.4.3	Minimisation des variables d'états par réencodage	81
3.5	Conclusion	82
4	Calcul de l'image d'une fonction	85
4.1	Difficulté du calcul de $Img(\vec{f}, \chi)$	85
4.2	Calcul direct de $Img(\vec{f}, \chi)$	87
4.3	Décomposition du calcul de l'image	89
4.4	Restricteur d'image	92
4.5	L'opérateur "Constrain"	95
4.5.1	Présentation logique de la Plus Proche Interpretation	95
4.5.2	Définition topologique de la Plus Proche Interpretation	97
4.5.3	Définition et évaluation de l'opérateur "Constrain"	99
4.6	Choix d'une couverture	100
4.6.1	Utilisation d'un partitionnement du co-domaine	100
4.6.2	Utilisation d'un partitionnement du domaine	101
4.7	Résultats expérimentaux et discussion	102
4.7.1	L'algorithme final	102
4.7.2	Résultats expérimentaux	103
4.8	Conclusion	107

5	Calcul de l'image réciproque d'une fonction	109
5.1	Difficulté du calcul de $Pre(\vec{f}, Cns, \chi)$	109
5.2	Calcul de $Pre(\vec{f}, Cns, \chi)$ à l'aide de Δ	111
5.3	Evaluation de $Pre(\vec{f}, Cns, \chi)$	111
5.3.1	Réduction de taille pour la composition	111
5.3.2	Réduction incrémentale pendant la composition	113
5.3.3	Décomposition de la composition	114
5.4	Résultats expérimentaux et discussion	116
5.5	Conclusion	117
	Conclusion	119
	Annexes	121
A	Terme, réécriture, forme normale et canonique	123
B	Autres graphes typés	125
B.1	Graphe 4-typé	125
B.2	Graphe symétrisé	127
C	Dénotation d'ensembles par des fonctions	131
C.1	Fonction caractéristique	131
C.2	Image d'une fonction	132
C.3	Image réciproque d'une fonction	132
D	Ensemble de fonctions	133
D.1	Représentations d'ensembles de fonctions	133
D.2	Pouvoirs de dénotation	134
D.3	Fonction paramétrée contrainte	136
E	Réduction et minimisation	137
E.1	Introduction	137
E.2	Minimisation d'une fonction partielle	138
E.3	Minimisation d'une fonction paramétrée	140
E.4	Réduction polynomiale : l'opérateur "Restrict"	140
	Bibliographie	143
	Index	153

Liste de Figures

Représentation des formules propositionnelles	27
Figure 1. Arbre de Shannon de $(x_1 \wedge (x_3 \oplus x_4)) \vee (x_2 \wedge (x_3 \Leftrightarrow x_4))$	31
Figure 2. Orthogonalité de la forme de Shannon	31
Figure 3. Arbre réduit de $(x_1 \wedge (x_3 \oplus a_4)) \vee (x_2 \wedge (x_3 \Leftrightarrow x_4))$	32
Figure 4. Distribution normalisée des arbres réduits	35
Figure 5. BDD de $(x_1 \wedge (x_3 \oplus x_4)) \vee (x_2 \wedge (x_3 \Leftrightarrow x_4))$	39
Figure 6. Exemple de BDD dénoté par une chaîne	42
Figure 7. Arbre de décision typé de $((x_1 \wedge (x_3 \oplus x_4)) \vee (x_2 \wedge (x_3 \Leftrightarrow x_4)))$	44
Figure 8. Arbre réduit typé de $((x_1 \wedge (x_3 \oplus x_4)) \vee (x_2 \wedge (x_3 \Leftrightarrow x_4)))$	46
Figure 9. TDG de $((x_1 \wedge (x_3 \oplus x_4)) \vee (x_2 \wedge (x_3 \Leftrightarrow x_4)))$	47
Figure 10. Croissances des sommes de produits, BDD et TDG sur les additionneurs	47
Figure 11. Calcul du TDG de la formule $((a \wedge b) \vee \neg c) \Rightarrow d$	48
Figure 12. Graphe de t_n	54
Figure 13. Graphe de $(t_n \wedge (a \Leftrightarrow b))$	54
Figure 14. Graphe de g	57
Figure 15. Graphe des fonctions symétriques	59
Problèmes sur les machines séquentielles	69
Figure 16. Modèle d’une machine séquentielle	70
Figure 17. Obtention du 6-uplet $(n, m, r, \omega, \delta, Init)$	71
Figure 18. Machine produit pour la comparaison des langages générés	72
Figure 19. Parcours virtuel du graphe de transition	74
Figure 20. Modèle universel de $AG(y \Rightarrow AX(z))$	76
Figure 21. Calcul des états satisfaisant $A[fUG]$	78
Calcul de l’image d’une fonction	85
Figure 22. Graphes de G et de $(\exists \vec{x} G)$	88
Figure 23. Squelette de la fonction “vectochar”	89
Figure 24. Restricteur d’image	90
Figure 25. Fonction “vectochar” utilisant un cache	91
Figure 26. Restricteur d’image strict	94
Figure 27. Fonction “constrain”	99
Figure 28. Exemple d’application de l’opérateur “constrain”	100
Figure 29. Premier algorithme de calcul de Img	101
Figure 30. Deuxième algorithme de calcul de Img	101

Figure 31. Première fonction “compute-valid”	102
Figure 32. Fonction “compute-valid” finale	103
Figure 33. Courbe $\log_{10}(time)$ en fonction de $\log_{10}(\#valid)$	106
Calcul de l’image réciproque d’une fonction	109
Figure 34. Fonction “compose1”	113
Figure 35. Fonction “compose2”	114
Figure 36. Fonction “decompose-composition”	115
Autres graphes typés	125
Figure 37 Graphe symétrisé de $((x_1 \wedge (x_3 \oplus x_4)) \vee (x_2 \wedge (x_3 \Leftrightarrow x_4)))$	129
Réduction et minimisation	137
Figure 38. Fonction “restrict”	141
Figure 39. Exemple d’application de l’opérateur “restrict”	141

Liste de Tableaux

Représentation des formules propositionnelles	27
Table 1. Répartition par taille des arbres réduits de Shannon	34
Table 2. Comparaison des somme de produits, forme de Reed-Muller, et TDG .	66
Problèmes sur les machines séquentielles	69
Table 3. Exemple de variables redondantes	81
Calcul de l'image d'une fonction	85
Table 4. Comportement de $(\vec{f} \downarrow \chi)$	100
Table 5. Résultats du calcul des états valides	104
Table 6. Analyse du test d'identification étendu	104

Introduction

Motivations industrielles

Les défauts qui peuvent apparaître au cours des différentes phases de l'élaboration d'un nouveau produit doivent être éliminés afin d'assurer une qualité maximale de la production. En effet, la correction d'un défaut implique le retour sur la conception ou la production du produit, ce qui nécessite un investissement additionnel. De plus, au retard de mise sur le marché et aux coûts directs de rectification s'ajoutent les manques à gagner dûs à la perte de crédibilité de l'entreprise, si le client est victime de ces rattrapages d'erreurs. Ceci explique les investissements considérables consentis par les entreprises pour mettre en application le concept de qualité totale, qui consiste à maîtriser au cours de la conception toutes les causes de défaillances d'un produit.

Ceci est particulièrement vrai en informatique. D'une part, la technologie et la demande évoluent très vite, ce qui fait que le facteur temps est crucial, et toute erreur ralentit la mise sur le marché d'un nouveau système. D'autre part, le monde digital impose que la réponse des systèmes, logiquement *discrète*, soit *exacte*, à la différence de nombreux autres secteurs où la réponse est *analogique* (électronique de puissance, résistance des matériaux, mécanique, chimie, ...), où une *tolérance* —par nature— est admise sur la réponse du système. De plus, la complexité des multiples paramètres des systèmes informatiques mis sur le marché fait que la prévention, la détection et la correction des défauts deviennent extrêmement difficiles. On se concentrera ici sur la *fonctionnalité* du système, qui consistera en une machine à états finis (circuit, interface, spécification d'un protocole, ...). Le but est de s'assurer que le système réalisera effectivement et correctement un certain nombre de fonctionnalités spécifiées [5].

La génération automatique, ou synthèse, semble la voie la plus sûre pour concevoir des systèmes sans défaut. Partant d'une spécification du circuit à réaliser, un ensemble de programmes en génère l'implémentation. Pour assurer la génération de bons circuits, les programmes de synthèse automatique doivent être corrects. Or ces programmes doivent être régulièrement modifiés pour tirer le profit maximal des technologies d'implantation des circuits, ce qui diminue la confiance que l'on peut leur accorder. Cette situation motive les recherches sur les compilateurs prouvés corrects [99]. En pratique le risque d'erreur résiduelle est trop élevé pour que l'on envoie en production des circuits générés par ces outils automatiques sans les vérifier.

Quant bien même le système de synthèse marcherait correctement (ce qui est heureusement vrai dans la grande majorité des cas), l'écriture de la spécification est une étape

délicate où peuvent s'introduire des différences par rapport au produit désiré. Il faut donc valider la spécification, en montrant qu'elle satisfait certaines propriétés typiques du système. Par exemple, la spécification d'un protocole réglant la communication entre plusieurs processeurs devra être telle que chaque processeur puisse accéder au bus en un temps fini.

La vérification de matériel recouvre deux aspects. Le premier consiste à comparer les comportements observables de deux machines séquentielles, l'une étant une spécification du système (description de haut niveau en VHDL [7, 87, 41] par exemple), l'autre la description de sa réalisation (en VHDL de bas niveau, ou une description en portes, voire le masque du circuit). Le second consiste à valider le comportement observable d'une machine séquentielle en vérifiant qu'il satisfait un certain nombre de propriétés dynamiques.

Une première technique de vérification consiste à réaliser un prototype, qui peut être matériel ou virtuel (par exemple une description algorithmique), et de le tester dans son environnement réel. La simulation consiste à exciter le prototype avec des données provenant de son environnement réel de fonctionnement. Cependant, la simulation ne peut être exhaustive à cause de la combinatoire explosive des systèmes d'informations. En conséquence, cette approche ne peut que valider un *jeux de tests* extrêmement *réduit* au regard de toutes les configurations possibles du système.

Vérifier formellement les systèmes informatiques consiste à remplacer la vérification expérimentale par une preuve, au sens mathématique du terme, de la correction du circuit. En d'autres termes, au lieu de tester qu'un système est correct par rapport à sa spécification (ou qu'il satisfait un ensemble de propriétés temporelles), on démontre l'adéquation de ce système avec sa spécification (ou qu'il est un modèle d'un ensemble de propriétés temporelles).

La vérification formelle de matériel

La logique du premier ordre a semblé être un bon cadre pour la description et la vérification de systèmes séquentiels. L'idée est de décrire le fonctionnement d'un système séquentiel par un ensemble de prédicats récursifs. Le problème est donc de comparer la théorie engendrée par les prédicats qui forment la spécification avec celle engendrée par les prédicats qui décrivent la réalisation, si l'on veut comparer les comportement d'une spécification et d'une réalisation ; ou de montrer que la théorie engendrée par les prédicats qui décrivent un système est consistante avec les prédicats décrivant les propriétés à valider, si on veut montrer qu'un système satisfait un certain nombre de propriétés temporelles.

L'utilisation de systèmes logiques ayant la puissance du premier ordre a connu un essor considérable au cours des années 1980, et des résultats très significatifs ont été obtenus. Quelques circuits actuellement sur le marché portent déjà le label "Vérifié formellement" [42]. Dans ce cadre, on peut distinguer plusieurs approches.

L'idée introduite dans les années 1970 par T. J Wagner [124] et J. Darringer [52] était d'appliquer les méthodes de démonstration développées pour la vérification de logiciel à

la vérification de descriptions fonctionnelles de machines, en particulier la méthode des assertions de Floyd et les techniques d'exécution symbolique de programmes.

W. Hunt a utilisé le démonstrateur de Boyer-Moore pour effectuer la vérification d'un petit microprocesseur [79]. Ce démonstrateur travaille sur un sous-ensemble de la logique du premier ordre (seuls les quantificateurs universels sont autorisés), et permet l'utilisation du principe d'induction dans les preuves. L'utilisation du démonstrateur de D. Boyer-Moore pour la preuve de matériel a été poursuivie, en particulier en Europe par l'équipe de D. Borrione [20].

M. Gordon a développé un autre formalisme, appelé LCF-LSM [67, 68], construit au-dessus du système LCF (Logic of Computable Functions) développé par R. Milner à Edimbourg pour la génération interactive de preuves formelles. La couche LSM (Logic of Sequential Machines) rajoutée par M. Gordon étend le calcul de LCF avec des descriptions comportementales du CCS (Calculus of Communicating Systems). Ce système a lui aussi été utilisé pour vérifier un petit ordinateur [68], mais n'a eu aucune implication industrielle. Le travail de H. G. Barrow [9] est très proche de celui de M. Gordon, excepté qu'il utilise PROLOG comme démonstrateur.

La logique du premier ordre a une grande puissance d'expression qui permet son utilisation directe comme formalisme de description de matériel, mais la puissance d'expression du premier ordre rend impossible l'automatisation des preuves [36]. Un démonstrateur de cette puissance ne peut pas réaliser tout seul la preuve de correction d'un circuit non trivial. L'utilisateur doit être capable de conduire la preuve, le système n'étant là que pour lui éviter de fastidieuses réécritures de formules. Par exemple, le système LCF offre des primitives de constructions de preuves appelées "tactics" et "tacticals". Les premières permettent de décomposer les problèmes à résoudre en sous-problèmes, et les secondes de combiner les premières pour construire des preuves complexes. Les preuves effectuées avec les systèmes cités ci-dessus ont d'ailleurs été faites par les logiciens/mathématiciens créateurs de ces outils, et ont été longues à réaliser. Or il n'est pas envisageable à court terme de demander aux concepteurs de circuits de devenir aussi spécialistes de logique formelle, ni d'associer à chaque équipe de conception un logicien de haut niveau. L'objectif de développer des outils de vérification automatiques a conduit d'autres équipes, dont celle du Centre de Recherche de BULL [5, 6], vers des voies différentes.

Un système logique décidable et assez puissant pour exprimer et résoudre les problèmes de vérification de matériel, à partir du moment où ils se situent sur un domaine d'interprétation fini, est la logique propositionnelle. En 1986, R. E. Bryant propose [27] une nouvelle forme canonique des formules propositionnelles, les Graphes de Décision Binaires, ou BDDs. Cette forme puissante et compacte permet pour la première fois de prouver complètement la correction de gros additionneurs (32 bits et plus). En 1987, une forme améliorée des BDDs, appelée Graphes de Décisions Typés ou TDGs, est proposée par J. P. Billon [18]. Cette forme, plus un processus d'*exécution symbolique*, conçu par J. C. Madre [89, 19], du langage de description de matériel LDS, permet la conception en 1988 de *PRIAM*, premier outil industriel de preuve de circuits *combinatoires*. *PRIAM* a été intégré dans le système de CAO de BULL et est maintenant utilisé quotidiennement

par tous les concepteurs de BULL.

La suite logique de *PRIAM*, limité au combinatoire, était de s’attaquer au problème de la preuve de systèmes séquentiels. Ce travail est l’objet de cette thèse, qui s’est concrétisé en une boîte à outils, *SIAM*, permettant de comparer les comportements observables de deux machines, et de valider une machine par un ensemble de propriétés temporelles. *SIAM* fournit aussi un jeu de primitives permettant de manipuler les machines afin de réduire leur complexité. Ces primitives ont été reprises dans le cadre de la synthèse de circuits combinatoires et séquentiels [17, 85, 110].

Spécification du problème

De nombreux problèmes portant sur les machines séquentielles, par exemple la vérification d’une implémentation par rapport à une spécification, consistent à comparer les comportements observables de deux machines selon certains critères. Nous montrerons que la comparaison de machines peut s’effectuer sans construction de diagramme d’états, ni énumération d’états ou de transitions, ce qui constituait la limite des techniques précédemment connues.

Si la comparaison de machines permet de vérifier l’adéquation du comportement observable d’une machine avec un autre comportement décrit par une autre machine, certaines propriétés observables ne peuvent pas toujours être ainsi exprimées. Par exemple, la propriété “*si à un moment, tel événement survient, alors tel autre événement se produira dans le futur*” ne peut être exprimée par une machine d’états finis. Aussi une autre façon de décrire un comportement observable est d’utiliser une logique temporelle. Nous montrerons comment la vérification de propriétés temporelles exprimées dans la logique CTL (*Computational Tree Logic*), c’est à dire la logique temporelle arborescente, peut être effectuée à partir de la description de la machine, sans jamais construire son diagramme d’états, comme l’imposaient les techniques de vérification proposées dans le passé (toutes dérivées du *model checking* [40]).

Nous définirons tout d’abord un modèle de machine séquentielle sur lequel porteront les preuves. Puis nous proposerons plusieurs algorithmes, basés sur des manipulations de formules booléennes quantifiées, permettant de résoudre les deux problèmes présentés ci-dessus. Les TDGs seront utilisés pour représenter et manipuler les formules booléennes quantifiées. Nous réduirons alors les deux problèmes de vérification à l’étude de deux primitives de manipulation formelle, à savoir le calcul de l’image et le calcul de l’image réciproque d’une fonction vectorielle booléenne.

Plan de lecture

Le Chapitre 1 expose les résultats simples mais essentiels de la logique propositionnelle avec quantificateurs. Nous posons dans ce chapitre les relations entre formules propositionnelles et fonctions booléennes, et nous y donnons la résolution générale et complète des équations booléennes.

Le Chapitre 2 étudie la représentation des fonctions booléennes. Nous y exposons les représentations déjà connues des fonctions booléennes par des graphes, et nous en proposerons quelques autres. Nous avons surtout développé une analyse des complexités de leurs manipulations. Puis nous montrons la supériorité des formes canoniques graphiques sur deux autres représentations très utilisées dans la démonstration automatique, la *forme normale disjonctive* (ou *somme de produits*), et la forme de *Reed-Muller* (ou *somme exclusive de produits*).

Le Chapitre 3 expose les deux aspects de la vérification de machines séquentielles : comparaison de comportements observables de machines et vérification de propriétés temporelles. Nous montrons que la solution de ces deux problèmes repose sur deux primitives, respectivement le calcul de l'image d'une fonction vectorielle, et le calcul de l'image réciproque d'une fonction vectorielle. Nous aborderons aussi, mais succinctement, les autres manipulations formelles fournies par *SIAM* pour la réduction/minimisation de machines. Sur le problème de la réduction de complexité, on trouvera dans l'Annexe E une courte discussion sur la minimisation combinatoire, qui consiste à réduire la représentation d'une fonction partielle.

Le Chapitre 4 étudie le calcul de l'image, qui est la primitive essentielle pour la comparaison de machines. Nous discutons de la complexité de ce calcul, et proposons plusieurs algorithmes pour son évaluation. Puis nous donnons et analysons des résultats expérimentaux sur un ensemble de machines, qui montreront la supériorité de notre approche symbolique sur les techniques de comparaison de comportements observables proposées dans le passé.

Le chapitre 5 étudie le calcul de l'image réciproque, qui est la primitive essentielle pour la preuve de propriétés temporelles. Là encore, nous analyserons la complexité de ce calcul, et nous donnerons des résultats expérimentaux obtenus chez BULL et par d'autres équipes. Ces résultats montrent que l'approche symbolique que nous décrivons permet de résoudre des problèmes inabordables avec les techniques du type *Model Checking*.

Notre contribution

La logique propositionnelle est un formalisme simple supportant la démonstration automatique complète, et permettant de modéliser tout problème de domaine d'interprétation fini. Nous avons trouvé commode d'introduire les quantificateurs, car si la quantification n'ajoute rien à la "puissance" de la logique propositionnelle, elle permet de représenter linéairement des formules dont la forme sans quantificateur est de taille exponentielle, et aussi de décrire simplement un système de résolution complet pouvant remplir de multiples usages [92, 93, 95, 49, 50, 51].

La technique de représentation des fonctions booléennes par les TDGs (*Typed Decision Graphs*, ou graphes de décision typés) a été développée au Centre de Recherche de BULL depuis 1987, parallèlement aux BDDs (*Binary Decision Diagram*) de R. E. Bryant introduits en 1986. Nous avons contribué au développement des graphes en tant que représentation des fonctions booléennes, en étudiant leurs propriétés et complexité. Nous avons proposé et amélioré de nouveaux algorithmes de manipulation. Nous avons aussi

introduit et étudié des formes dérivées des TDGs.

Nous avons défini des algorithmes qui, avec l'utilisation des TDGs, permettent la vérification de machines séquentielles (comparaison et vérification de propriétés temporelles) dont la taille (nombre d'états et de transitions utiles) dépasse largement la capacité des techniques énumératives précédemment connues. En effet, les algorithmes de preuve que nous proposons ont la propriété d'avoir une complexité qui *ne dépend ni* du nombre d'états, *ni* du nombre de transitions des machines traitées. Toute la complexité est reportée sur des manipulations de TDGs. Nous avons défini plusieurs primitives permettant d'améliorer considérablement les calculs nécessaires à cet effet. Ces techniques ont permis d'étendre largement le champ d'application de la preuve formelle de matériel, et ont trouvé de nombreuses utilisations dans le domaine de la synthèse de circuits [17, 85, 110]. Certaines d'entre elles sont récemment entrées dans le monde industriel, en étant incluses dans des produits de CAO mis sur le marché.

Partie I

Logique propositionnelle quantifiée

Chapitre 1

Logique propositionnelle quantifiée

La logique propositionnelle est complète et décidable, donc susceptible de supporter un démonstrateur automatique pour résoudre avec succès les problèmes qui peuvent s’y exprimer. Nous exposons ici le système formel sur lequel nous nous appuyerons dans la suite, la logique propositionnelle quantifiée.

1.1 Formules propositionnelles quantifiées

Dans cette section nous présentons la syntaxe et la sémantique des formules propositionnelles quantifiées, puis nous précisons comment une formule propositionnelle dénote sans ambiguïté une unique fonction booléenne. On trouvera dans l’Annexe A un rappel des concepts de *terme*, de *substitution*, et de *réécriture* que nous utiliserons ici.

1.1.1 Syntaxe des formules propositionnelles quantifiées

Nous définissons ici les termes appelés *formules propositionnelles quantifiées*. On considère un ensemble dénombrable \mathcal{V} de symboles appelés variables propositionnelles. On les notera $x, x_1, x_2, \dots, y, y_1, y_2, \dots, z, \dots$. Les formules propositionnelles quantifiées sont des *termes* construits à partir de \mathcal{V} , du symbole de fonction monadique $\mathcal{F}_1 = \{\neg\}$, et des symboles de fonctions diadiques $\mathcal{F}_2 = \{\vee, \wedge, \Rightarrow, \Leftrightarrow, \oplus, \exists, \forall\}$. Par commodité, les symboles de fonctions seront infixés ou préfixés. Autrement dit, les formules propositionnelles quantifiées, que nous noterons f, g, h, \dots , s’écrivent avec l’alphabet dénombrable $\{\neg, \vee, \wedge, \Rightarrow, \Leftrightarrow, \oplus, (,), \exists, \forall\} \cup \mathcal{V}$, et leur syntaxe $\langle QPF \rangle$ est définie par :

$$\begin{aligned} \langle QPF \rangle & ::= \langle var \rangle \\ & ::= (\langle QPF \rangle) \\ & ::= \langle op_1 \rangle \langle QPF \rangle \\ & ::= \langle QPF \rangle \langle op_2 \rangle \langle QPF \rangle \\ & ::= \langle quantifier \rangle \langle var \rangle \langle QPF \rangle \\ \langle op_1 \rangle & ::= \neg \\ \langle op_2 \rangle & ::= \vee \mid \wedge \mid \Rightarrow \mid \Leftrightarrow \mid \oplus \\ \langle var \rangle & ::= x \mid x_1 \mid x_2 \mid \dots \\ \langle quantifier \rangle & ::= \exists \mid \forall \end{aligned}$$

L'occurrence d'une variable x dans une formule f est dite *libre* si et seulement si cette occurrence n'apparaît pas dans le champ d'un quantificateur portant sur x . L'occurrence non libre d'une variable est dite *liée*. On notera $f(x_1, \dots, x_n)$ une formule propositionnelle dont les variables qui possèdent au moins une occurrence libre sont x_1, \dots, x_n . Par exemple, dans la formule $(\forall y f(x, y, z) \Rightarrow (\exists x g(x, y)))$, z est libre, les occurrences de y sont liées, et x possède une occurrence libre (la première) et une occurrence liée (la seconde). Une formule est dite *close* si toute occurrence de variable est liée. Si $f(x)$ dénote une formule, on notera $f(t)$ la formule obtenue en remplaçant toute occurrence libre de x dans $f(x)$ par t .

1.1.2 Sémantique des formules propositionnelles quantifiées

On appelle *interprétation* une application i de \mathcal{V} dans $\{0, 1\}$. Toute interprétation i est étendue en une fonction i_* de \mathcal{F} dans $\{0, 1\}$ de la façon suivante. Si x est une variable, $i_*(x) = i(x)$; $i_*(\neg f) = 1$ si et seulement si $i_*(f) = 0$; $i_*(f_1 \vee f_2) = 1$ si et seulement si $i_*(f_1) = 1$ ou $i_*(f_2) = 1$; $i_*(\exists x f(x)) = 1$ si et seulement si $i_*(f(0)) = 1$ ou $i_*(f(1)) = 1$. De plus, on pose les règles de réécriture suivantes, permettant d'étendre i_* sur toutes les formules (dans la suite, on ne distinguera plus i et i_*).

$$\begin{aligned} \forall x f &\rightarrow \neg(\exists x \neg f) \\ f \wedge g &\rightarrow \neg(\neg f \vee \neg g) \\ f \Rightarrow g &\rightarrow \neg f \vee g \\ f \Leftrightarrow g &\rightarrow (f \Rightarrow g) \wedge (g \Rightarrow f) \\ f \oplus g &\rightarrow \neg(f \Leftrightarrow g) \end{aligned}$$

Toute formule f associe à toute interprétation i un élément unique de $\{0, 1\}$, et définit donc une unique fonction $\lambda i. i(f)$ de $(\{0, 1\} \rightarrow \mathcal{V})$ dans $\{0, 1\}$. Pour un ordre total fixé sur \mathcal{V} , une formule f définit une fonction unique de $\{0, 1\}^{|\mathcal{V}|}$ dans $\{0, 1\}$. Une formule f telle que $\lambda i. i(f) = 1$, par exemple $(\neg x \vee x)$, (respectivement une formule telle que $\lambda i. i(f) = 0$, par exemple $(\neg x \wedge x)$), sera appelée une *tautologie* (respectivement une *antilogie*).

1.1.3 Égalité sur les formules propositionnelles

Il y a plusieurs façons d'introduire l'égalité " $=_{\mathcal{B}}$ " sur les formules propositionnelles.

Une présentation très répandue consiste à définir un ensemble récursif de formules appelées *théorèmes* en utilisant un système formel [82]. Puis on dit que $(f =_{\mathcal{B}} g)$ si et seulement si la formule $(f \Leftrightarrow g)$ est un théorème. Il existe de nombreuses axiomatisations de ce système formel, par exemple celle-ci [127] :

$$\begin{aligned} \text{Schéma d'axiomes} & : (f \Rightarrow (g \Rightarrow f)) \\ & ((f \Rightarrow (g \Rightarrow h)) \Rightarrow ((f \Rightarrow g) \Rightarrow (f \Rightarrow h))) \\ & ((\neg f \Rightarrow \neg g) \Rightarrow (g \Rightarrow f)) \end{aligned}$$

Règle d'inférence (*Modus Ponens*) : de f et $(f \Rightarrow g)$, on dérive g .

Une deuxième définition est basée sur la sémantique des formules, en posant $(f =_{\mathcal{B}} g)$ si et seulement si f et g dénotent la même fonction. On peut montrer, en utilisant le théorème de complétude [54], que ceci est équivalent à dire que la formule $(f \Leftrightarrow g)$ est une tautologie ou un théorème.

Une troisième présentation consiste à donner directement la définition de $=_{\mathcal{B}}$ sous la forme d'une théorie équationnelle [73], ou bien d'un système de réécriture. Voici une définition équationnelle [73] :

$$\begin{aligned} \neg\neg f &=_{\mathcal{B}} f \\ (f \vee g) &=_{\mathcal{B}} (g \vee f) \\ ((f \vee g) \vee h) &=_{\mathcal{B}} (f \vee (g \vee h)) \\ \neg(f \vee g) &=_{\mathcal{B}} \neg f \wedge \neg g \\ \neg(f \wedge g) &=_{\mathcal{B}} \neg f \vee \neg g \\ f \wedge (g \vee h) &=_{\mathcal{B}} (f \wedge g) \vee (f \wedge h) \\ f \vee (g \wedge h) &=_{\mathcal{B}} (f \vee g) \wedge (f \vee h) \end{aligned}$$

1.1.4 Formules propositionnelles et fonctions booléennes

On appelle *fonction booléenne* une fonction de $\{0, 1\}^n$ dans $\{0, 1\}$. Nous avons vu dans la Section 1.1.2 qu'une formule propositionnelle f dénote une unique fonction booléenne $\lambda i.i(f)$ de $\{0, 1\}^{|\mathcal{V}|}$ dans $\{0, 1\}$. Réciproquement, le théorème suivant indique que toute fonction booléenne est dénotée par au moins une formule.

Théorème 1.1 *Pour tout n entier fini et toute fonction h de $\{0, 1\}^n$ dans $\{0, 1\}$, il existe une formule propositionnelle sans quantificateur f telle que $h = \lambda i.i(f)$.*

Preuve. Pour tout élément $v = (v_1, \dots, v_n)$ de l'ensemble $\{0, 1\}^n$, on définit la formule $g_v = (\varepsilon_1(x_1) \wedge \varepsilon_2(x_2) \wedge \dots \wedge \varepsilon_n(x_n))$, avec $\varepsilon_k(x_k) = x_k$ si $v_k = 1$, et $\varepsilon_k(x_k) = \neg x_k$ sinon. g_v dénote une fonction de $\{0, 1\}^n \rightarrow \{0, 1\}$ qui prend partout la valeur 0 sauf en v . Soient alors les formules h_v définies pour tout élément v de $\{0, 1\}^n$ par $h_v = (\neg x_1 \wedge x_1)$ si $h(v) = 0$, et $h_v = (\neg x_1 \vee x_1)$ si $h(v) = 1$. Soit f la formule propositionnelle définie par :

$$f = \left(\bigvee_{v \in \{0, 1\}^n} (g_v \wedge h_v) \right).$$

On peut alors vérifier que $i(f) = h(i(x_1), \dots, i(x_n))$ pour toute interprétation i . La formule f est composée de $O(n \times 2^n)$ caractères, donc f est une formule finie quand n est fini. \square

Puisque toute fonction booléenne peut être dénotée par au moins une formule sans quantificateur, on en déduit que toute formule propositionnelle avec quantificateurs est équivalente, i.e. égale au sens de $=_{\mathcal{B}}$, à une formule sans quantificateur. On appellera *forme sans quantificateur* d'une formule quantifiée une formule sans quantificateur qui lui est équivalente.

Comme une formule propositionnelle dénote une unique fonction booléenne, et qu'une fonction booléenne est dénotée par des formules propositionnelles toutes équivalentes entre

elles, on confondra souvent formule et fonction. On rajoute au vocabulaire des formules propositionnelles l'ensemble des constantes $\mathcal{F}_0 = \{0, 1\}$ pour dénoter l'antilogie et la tautologie. Par abus de langage, on écrira “=” pour “= \mathcal{B} ”. Un n -uplet x de $\{0, 1\}^n$ sera noté par un vecteur $[x_1 \dots x_n]$, ou \vec{x} s'il n'y a pas d'ambiguïté sur n . La concaténation sur les vecteurs est notée “@”. On utilisera la curryfication pour alléger l'écriture des fonctions, ce qui permet d'écrire indifféremment $\lambda x_1 \dots \lambda x_n. \lambda y. f$, ou bien $\lambda[x_1 \dots x_n]. \lambda y. f$, ou bien $\lambda \vec{x}. \lambda y. f$, ou encore $\lambda(\vec{x}@[y]). f$.

1.2 Elimination des quantificateurs

Nous montrons ici qu'il existe essentiellement deux procédés d'obtention d'une forme sans quantificateur, l'un consistant à éliminer les quantificateurs de la gauche vers la droite, l'autre consistant à les éliminer de la droite vers la gauche.

On dit qu'une formule est sous forme *prénexe* si elle est de la forme $(Q_1 x_1 \dots Q_n x_n f)$, où Q_k est un quantificateur, et f une formule sans quantificateur (on a éventuellement $n = 0$). Pour toute formule propositionnelle quantifiée, on peut construire un arbre syntaxique dont les feuilles sont des formes prénexes, et les noeuds sont des connecteurs logiques, des symboles de fonctions, ou des quantificateurs. Le problème de l'obtention d'une forme sans quantificateur revient donc à étudier l'élimination des quantificateurs d'une forme prénexe. L'élimination est basée sur le théorème suivant, qui découle de la sémantique des formules.

Théorème 1.2 *Soit f une formule quelconque, et x une variable ne possédant pas d'occurrence liée dans f . On a les identités suivantes, qu'on appellera Q -éliminations. L'identité 1.1 est appelée élimination existentielle ou \exists -élimination, et l'identité 1.2 est appelée élimination universelle ou \forall -élimination.*

$$(\exists x f) = (f[x \leftarrow 0] \vee f[x \leftarrow 1]) \quad (1.1)$$

$$(\forall x f) = (f[x \leftarrow 0] \wedge f[x \leftarrow 1]) \quad (1.2)$$

1.2.1 Elimination gauche-droite des quantificateurs

Soit $(\exists x_1 Q_2 x_2 \dots Q_n x_n f)$ une forme prénexe. Cette formule est équivalente à la formule :

$$(Q_2 x_2 \dots Q_n x_n f[x_1 \leftarrow 0]) \vee (Q_2 x_2 \dots Q_n x_n f[x_1 \leftarrow 1]) \quad (1.3)$$

De même, la formule $(\forall x_1 Q_2 x_2 \dots Q_n x_n f)$ est équivalente à la formule :

$$(Q_2 x_2 \dots Q_n x_n f[x_1 \leftarrow 0]) \wedge (Q_2 x_2 \dots Q_n x_n f[x_1 \leftarrow 1]) \quad (1.4)$$

De cette façon, la première variable quantifiée et son quantificateur sont éliminés.

Il y a ensuite deux façons d'utiliser ce processus d'élimination du quantificateur le plus à gauche d'une forme prénexe.

La première technique consiste à calculer une forme prénexe de la formule obtenue après la première élimination, puis à éliminer les quantificateurs restants. Dans ce cas, la forme prénexe de la formule 1.3 obtenue après la \exists -élimination de x_1 est :

$$(Q_2x_2 \dots Q_nx_n Q_2x'_2 \dots Q_nx'_n f[x_1 \leftarrow 0] \vee f[x_1 \leftarrow 1][x_2 \leftarrow x'_2, \dots, x_n \leftarrow x'_n]) \quad (1.5)$$

La normalisation sous forme prénexe de la formule 1.3 requiert le renommage des $n - 1$ variables d'une des formules de la disjonction. La forme prénexe 1.5 obtenue possède $2n - 2$ variables quantifiées. Cette technique expande donc la forme prénexe et ne peut réduire la formule par Q -élimination, à moins d'utiliser des réécritures sur la formule sans quantificateur ($f[x_1 \leftarrow 0] \vee f[x_1 \leftarrow 1][x_2 \leftarrow x'_2, \dots, x_n \leftarrow x'_n]$). Cette technique ne peut donc s'appliquer sans risquer une expansion coûteuse de la forme prénexe.

La seconde technique consiste à éliminer séparément les quantificateurs des deux formes prénexes ($Q_2x_2 \dots Q_nx_n f[x_1 \leftarrow 0]$) et ($Q_2x_2 \dots Q_nx_n f[x_1 \leftarrow 1]$) obtenues dans 1.3 ou 1.4, pour les recombinaison ensuite. Il est aisé de voir que cet algorithme se termine, mais qu'il requiert 2^n Q -éliminations. Nous montrons dans la section suivante que l'élimination de la droite vers la gauche est moins coûteuse.

1.2.2 Elimination droite-gauche des quantificateurs

On considère la forme prénexe ($Q_1x_1 \dots Q_nx_n f$), où f est sans quantificateur. On définit récursivement la séquence de formules f_k sans quantificateur pour $0 \leq k \leq n$ par :

$$\begin{aligned} f_n &= f \\ \text{Si } Q_k &= \exists \quad \text{alors } f_{k-1} &= f_k[x_k \leftarrow 0] \vee f_k[x_k \leftarrow 1] \\ \text{Si } Q_k &= \forall \quad \text{alors } f_{k-1} &= f_k[x_k \leftarrow 0] \wedge f_k[x_k \leftarrow 1] \end{aligned}$$

On voit que f_k est une forme sans quantificateur de ($Q_{k+1}x_{k+1} f_{k+1}$), qui est aussi une forme sans quantificateur de ($Q_{k+1}x_{k+1} \dots Q_nx_n f$). Aussi f_0 est une forme sans quantificateur de la formule initiale f . La forme sans quantificateur d'une forme prénexe à n variables quantifiées est ainsi obtenue par seulement¹ n Q -éliminations de droite à gauche. En revanche, si aucune réécriture n'est effectuée pour réduire les formules f_k au cours du calcul, alors la taille de la formule f_0 est en $O(2^{|f|})$ si la taille de f est $|f|$.

1.3 Tautologie, antilogie, satisfiabilité

Une formule f est une tautologie (respectivement une antilogie) si et seulement si elle dénote la fonction 1 (respectivement la fonction 0). Une formule est *satisfiable* si et seulement si elle ne dénote pas la fonction 0. Le problème de la preuve consiste à savoir si une formule est une tautologie. Ce problème contient les deux autres, à savoir l'antilogie et la satisfiabilité : une formule f est une antilogie si et seulement si $(\neg f)$ est une tautologie, et f est satisfiable si et seulement si $(\neg f)$ n'est pas une tautologie. Soit $f(x_1, \dots, x_n)$ une formule dont les variables libres sont x_1, \dots, x_n . On a le résultat suivant :

¹Ceci est dû au fait que l'élimination droite-gauche ne brise pas la forme prénexe de la formule, à la différence de l'élimination gauche-droite.

Théorème 1.3 *La formule $f(x_1, \dots, x_n)$ est une tautologie si et seulement si la formule close $(\forall x_1 \dots \forall x_n f(x_1, \dots, x_n))$ est une tautologie.*

Comme la manipulation syntaxique consistant à passer de la formule $f(x_1, \dots, x_n)$ à la formule close $(\forall x_1 \dots \forall x_n f(x_1, \dots, x_n))$, et réciproquement, s'effectue en $O(n)$, déterminer si une formule quelconque est une tautologie est un problème de même complexité que celui de déterminer si une formule close est une tautologie. On s'intéresse donc ici à déterminer si une formule close dénote la fonction tautologie 1.

La forme sans quantificateur d'une formule close ne peut qu'être la fonction constante 0 ou 1. Pour savoir si une formule close est une tautologie, il suffit d'en calculer une forme sans quantificateur, puis de vérifier que celle-ci dénote la fonction 1.

Élimination gauche-droite pour la validité

D'après la Section 1.2.1, l'obtention d'une forme sans quantificateur d'une forme prénex $(Q_1 x_1 \dots Q_n x_n f)$ se fait par 2^n Q -éliminations et combinaisons de la gauche vers la droite. En utilisant les règles de réécritures élémentaires ci-dessous portant uniquement sur les constantes 0 et 1 (une réécriture étant effectuée à chaque retour récursif d'une Q -élimination), une formule close est réduite en la constante 0 ou 1.

$$\begin{aligned} \neg 0 &\rightarrow 1 \\ \neg 1 &\rightarrow 0 \\ 0 \vee t &\rightarrow t \\ 1 \vee t &\rightarrow 1 \end{aligned}$$

Élimination droite-gauche pour la validité

D'après la Section 1.2.2, l'obtention d'une forme sans quantificateur d'une formule close $(Q_1 x_1 \dots Q_n x_n f)$ se fait par n Q -éliminations droite-gauche. La formule obtenue, de taille exponentielle par rapport à la taille de f , ne fait intervenir que les constantes 0 et 1, ainsi que les opérateurs usuels. Le système de réécriture nécessaire pour réduire cette formule en sa constante équivalente est donc le même que précédemment.

1.4 Résolution d'équations

Soit $(Q_1 x_1 \dots Q_n x_n f)$ une formule close, où f est sans quantificateur. Notons x_{k_1}, \dots, x_{k_m} les m variables qui apparaissent quantifiées existentiellement de gauche à droite (c'est à dire $k_1 < \dots < k_m$). Cette formule peut être considérée comme une équation, dont les inconnues sont les variables quantifiées existentiellement. On dira qu'un m -uplet de formules $[f_1 \dots f_m]$ est une *solution* de l'équation si et seulement si :

- La formule f_j ne contient pas d'occurrence des variables x_k pour $k > k_j$.
- La formule² $f[x_{k_1} \leftarrow f_1, \dots, x_{k_m} \leftarrow f_m]$ est une tautologie.

Chaque m -uplet solution de l'équation dénote une fonction booléenne vectorielle de \mathcal{V} dans $\{0, 1\}^m$. L'ensemble des solutions de l'équation est l'ensemble de toutes ces fonctions vectorielles³. Cet ensemble peut être dénoté par un m -uplet de fonctions paramétrées (voir Annexe D), que nous appellerons *fonctions de Skolem* de l'équation, en référence à la skolemization [98]. Si $[s_1 \dots s_m]$ est un m -uplet de formules dénotant ces fonctions de Skolem, on appellera fonction de Skolem associée à la variable x_{k_j} la fonction booléenne dénotée par la formule s_j .

Par exemple, considérons l'équation suivante :

$$\exists x_1 \forall x_2 \exists x_3 \exists x_4 (x_1 \wedge ((x_2 \Leftrightarrow x_4) \vee x_3))$$

La substitution $[x_1 \leftarrow 1, x_3 \leftarrow 0, x_4 \leftarrow x_2]$ est une solution de cette équation. La solution la plus générale σ peut s'écrire:

$$\sigma = [x_1 \leftarrow 1, x_3 \leftarrow x_3, x_4 \leftarrow ((x_3 \wedge x_4) \vee (\neg x_3 \wedge x_2))].$$

Cette solution est unique modulo un renommage des variables (voir Annexe A). Cette substitution est la plus générale, car toute substitution σ' solution de l'équation s'écrit $\sigma' = \rho \circ \sigma$, où ρ est une substitution. Réciproquement, toute substitution de la forme $\rho \circ \sigma$ est une solution de l'équation. En d'autres termes, si V est l'ensemble des variables de l'équation qui sont quantifiées existentiellement, toute substitution σ' solution de l'équation est telle que $\sigma' \preceq [V]\sigma$ (voir Annexe A), c'est à dire que σ' est une *instance* de, ou moins générale que, σ . Par exemple, la substitution $[x_1 \leftarrow 1, x_3 \leftarrow 1, x_4 \leftarrow y]$, qui est une solution de l'équation, s'écrit $\rho \circ \sigma$, où $\rho = [x_3 \leftarrow 1, x_4 \leftarrow y]$.

Résoudre une équation, c'est donner les fonctions de Skolem associées à ses variables quantifiées existentiellement⁴. Nous présentons d'abord la résolution d'équation à une inconnue, puis la résolution dans le cas général.

1.4.1 Résolution d'équations à une inconnue

Soit l'équation à une inconnue, i.e. ne possédant qu'une seule variable quantifiée existentiellement :

$$\forall y_1 \dots \forall y_n \exists x \forall y_{n+1} \dots \forall y_m g(y_1, \dots, y_m, x)$$

La formule $(\forall y_{n+1} \dots \forall y_m g(y_1, \dots, y_m, x))$ est équivalente à une forme sans quantificateur, que nous noterons f . Alors cette équation s'écrit :

$$\forall y_1 \dots \forall y_n \exists x f \tag{1.6}$$

²Comme aucune des formules f_j ne contient d'occurrence des variables x_k pour $k > k_j$, le terme $f[x_{k_1} \leftarrow f_1, \dots, x_{k_m} \leftarrow f_m]$, obtenu grâce à une substitution *simultanée* des x_{k_j} par f_j , est égal au terme $f[x_{k_1} \leftarrow f_1] \dots [x_{k_m} \leftarrow f_m]$, obtenu par une *succession* de substitutions élémentaires effectuées de gauche à droite (voir Annexe A).

³Une équation permet donc de dénoter un ensemble de fonctions, voir Annexe D.

⁴L'unification booléenne [24] n'est qu'un cas particulier de la résolution d'équation.

où f est sans quantificateur. Cette équation s'écrit de façon équivalente :

$$\forall y_1 \dots \forall y_n \exists x (\neg x \wedge f[x \leftarrow 0]) \vee (x \wedge f[x \leftarrow 1]) \quad (1.7)$$

L'équation possède une solution si et seulement si la formule 1.7 est une tautologie, c'est à dire si et seulement si la formule

$$\forall y_1 \dots \forall y_n (f[x \leftarrow 0] \vee f[x \leftarrow 1]) \quad (1.8)$$

est une tautologie. Comme toutes les variables qui apparaissent dans la formule 1.8 sont quantifiées universellement, cette formule est une tautologie si et seulement si la formule $(f[x \leftarrow 0] \vee f[x \leftarrow 1])$ est elle-même une tautologie. Soit $\vec{y} = [y_1 \dots y_n]$. En supposant que l'équation ait une solution, trois cas disjoints recouvrent toutes les interprétations possibles de \vec{y} :

- Si $f[x \leftarrow 0](\vec{y}) = 0$, alors comme par hypothèse $(f[x \leftarrow 0] \vee f[x \leftarrow 1])$ est une tautologie, on a nécessairement $f[x \leftarrow 1](\vec{y}) = 1$. Donc x doit prendre la valeur 1 afin de satisfaire la formule 1.7.
- Si $f[x \leftarrow 0](\vec{y}) = 1$ et $f[x \leftarrow 1](\vec{y}) = 0$, alors x doit prendre la valeur 0 afin de satisfaire la formule 1.7.
- Si $f[x \leftarrow 0](\vec{y}) = 1$ et $f[x \leftarrow 1](\vec{y}) = 1$, alors la formule 1.7 est satisfaite indépendamment de la valeur de x .

Ceci signifie que la valeur que doit prendre x selon la valeur de \vec{y} est définie par :

$$\begin{aligned} x = & (\neg f[x \leftarrow 0](\vec{y}) \wedge 1) \\ & \vee (f[x \leftarrow 0](\vec{y}) \wedge \neg f[x \leftarrow 1](\vec{y}) \wedge 0) \\ & \vee (f[x \leftarrow 0](\vec{y}) \wedge f[x \leftarrow 1](\vec{y}) \wedge p) \end{aligned} \quad (1.9)$$

où p est une variable différente de y_1, \dots, y_n (par exemple la variable x elle-même), dénotant ainsi une valeur quelconque. La condition d'unicité de x est que x ne dépende pas du paramètre p , i.e. $(f[x \leftarrow 0] \wedge f[x \leftarrow 1]) = 0$. La valeur de x donnée par 1.9 s'écrit plus simplement $(\neg f[x \leftarrow 0](\vec{y}) \vee (p \wedge f[x \leftarrow 1](\vec{y})))$. En résumé :

Théorème 1.4 *L'équation $(\forall y_1 \dots \forall y_n \exists x f)$ admet une solution si et seulement si :*

$$(f[x \leftarrow 0] \vee f[x \leftarrow 1]) = 1$$

Cette équation possède une solution unique si et seulement si :

$$(f[x \leftarrow 0] \oplus f[x \leftarrow 1]) = 1$$

Lorsque l'équation admet une solution, la fonction de Skolem s associée à x , dénotant l'ensemble de toutes les solutions, est définie ci-dessous, où $p \notin \{y_1, \dots, y_n\}$:

$$s = (\neg f[x \leftarrow 0] \vee (p \wedge f[x \leftarrow 1]))$$

1.4.2 Résolution d'équations à plusieurs inconnues

On considère maintenant une équation à plusieurs inconnues :

$$Q_1 x_1 \dots Q_n x_n f \quad (1.10)$$

La résolution de cette équation consiste à trouver, si elles existent, les fonctions de Skolem, que nous noterons s_k , associées aux variables x_k qui sont quantifiées existentiellement. Si dans la chaîne " $Q_1 x_1 \dots Q_n x_n$ ", il y a m variables x_{j_1}, \dots, x_{j_m} quantifiées universellement et $k - m - 1$ variables $x_{i_1}, \dots, x_{i_{k-m-1}}$ quantifiées existentiellement qui apparaissent avant l'occurrence de la variable x_k quantifiée existentiellement, alors la fonction de Skolem s_k dépend des m variables x_{j_1}, \dots, x_{j_m} , et de $k - m$ paramètres, constitués des $k - m - 1$ paramètres associés aux variables $x_{i_1}, \dots, x_{i_{k-m-1}}$, plus le paramètre associé à x_k .

La résolution d'une équation à plusieurs inconnues reprend le résultat présenté précédemment. Le principe est le suivant. On cherche d'abord la variable la plus à gauche quantifiée existentiellement. Soit x_{k_1} cette variable, et soit f_{k_1} la forme sans quantificateur de $(Q_{k_1+1} x_{k_1+1} \dots Q_n x_n f)$. La formule 1.10 est alors équivalente à la formule :

$$\forall x_1 \dots \forall x_{k_1-1} \exists x_{k_1} f_{k_1}. \quad (1.11)$$

Cette équation ne possède qu'une inconnue. Si elle ne possède pas de solution, alors l'équation 1.10 n'a pas de solution non plus, et réciproquement. Sinon, on sait déterminer la fonction de Skolem s_{k_1} de x_{k_1} . Cette fonction de Skolem, qui s'écrit sous la forme $(\neg f_{k_1}[x_{k_1} \leftarrow 0] \vee (p_{k_1} \wedge f_{k_1}[x_{k_1} \leftarrow 1]))$, dépend des variables $x_1 \dots x_{k_1-1}$, et d'un paramètre p_{k_1} différent de ces variables. On peut donc prendre comme symbole de paramètre la variable x_{k_1} . On considère alors la nouvelle équation

$$\forall x_1 \dots \forall x_{k_1} Q_{k_1+1} x_{k_1+1} \dots Q_n x_n f[x_{k_1} \leftarrow s_{k_1}], \quad (1.12)$$

sur laquelle on itère le processus qui vient d'être décrit. A chaque pas de la résolution, la fonction de Skolem de la variable quantifiée existentiellement placée la plus à gauche est calculée, et cette variable devient quantifiée universellement dans l'équation suivante.

Le Théorème 1.5 formalise cette résolution par un processus récursif utilisant les formes sans quantificateurs $f_k, L_k, H_k, L'_k, H'_k$ ($0 \leq k \leq n$). Dans la définition de ces formes, $g = \perp$ signifie que la valeur de g n'est pas déterminée. Les formes f_k, L_k , et H_k sont calculées (à partir de $Q_{k+1}, f_{k+1}, L_{k+1}$, et H_{k+1}) pour k décroissant, et les formes L'_k et H'_k sont calculées (à partir de Q_k, L_k, H_k, L'_{k-1} , et H'_{k-1}) pour k croissant. La formule f_k est la forme sans quantificateur de $(Q_{k+1} x_{k+1} \dots Q_n x_n f)$, et la fonction de Skolem s_k associée à chaque variable x_k quantifiée existentiellement est calculée à partir de L'_{k-1} et H'_{k-1} . En substituant des termes quelconques aux paramètres (ici, le paramètre utilisé par la fonction de Skolem s_k associée à une variable quantifiée existentiellement x_k est cette même variable x_k) introduits dans les fonctions de Skolem $[s_1, \dots, s_m]$, on obtient un m -uplet de formules solution de l'équation. Si l'équation 1.10 est à m inconnues et qu'elle admet une solution, alors les fonctions de Skolem des m variables quantifiées existentiellement sont ainsi obtenues en n Q -éliminations et m substitutions.

Théorème 1.5 *Les fonctions de Skolem s_j de l'équation à plusieurs inconnues $(Q_1x_1 \dots Q_nx_n f)$ sont définies à partir des formes sans quantificateurs $f_k, L_k, H_k, L'_k, H'_k$, pour $0 \leq k \leq n$:*

$$\begin{aligned} f_n &= f \\ L_n &= \perp \\ H_n &= \perp \\ L'_0 &= L_0 \\ H'_0 &= H_0 \end{aligned}$$

Si il existe k tel que $f_k = 0$, alors l'équation n'a pas de solution

Si $Q_k = \forall$, alors :

$$\begin{aligned} f_{k-1} &= f_k[x_k \leftarrow 0] \wedge f_k[x_k \leftarrow 1] \\ L_{k-1} &= L_k \\ H_{k-1} &= H_k \\ L'_k &= L'_{k-1} \\ H'_k &= H'_{k-1} \end{aligned}$$

Si $Q_k = \exists$, alors :

$$\begin{aligned} f_{k-1} &= f_k[x_k \leftarrow 0] \vee f_k[x_k \leftarrow 1] \\ L_{k-1} &= f_k[x_k \leftarrow 0] \\ H_{k-1} &= f_k[x_k \leftarrow 1] \\ s_k &= (\neg L'_{k-1} \vee (x_k \wedge H'_{k-1})) \\ L'_k &= L_k[x_k \leftarrow s_k] \\ H'_k &= H_k[x_k \leftarrow s_k] \end{aligned}$$

1.5 Conclusion

Nous avons présenté la logique propositionnelle quantifiée. Les quantificateurs permettent d'une part, de décrire de façon compacte des expressions dont la forme sans quantificateur est de taille exponentielle ; d'autre part, de définir implicitement des ensembles de fonctions comme étant des instances des fonctions de Skolem d'une équation. Sur ce formalisme, nous avons décrit le processus d'élimination des quantificateurs et la résolution complète des équations booléennes. Cette présentation ne suppose aucune représentation particulière des formules propositionnelles, aussi ces résultats pourront être directement appliqués aux représentations proposées dans le Chapitre 2.

Ce système permet de dénoter et de manipuler des objets de tout système dont le domaine d'interprétation est fini. Il sera utilisé dans toute la suite pour formaliser les problèmes et décrire leurs solutions.

Chapitre 2

Représentation des formules propositionnelles

2.1 Introduction

Le chapitre précédent montrait comment représenter des fonctions booléennes par des formules propositionnelles. Nous nous intéressons maintenant à la représentation et à la manipulation des formules propositionnelles.

Notons que dans le passé, la manipulation de formules a été essentiellement dirigée par la preuve, c'est à dire que la fonction première des systèmes n'était pas de représenter des formules, mais de montrer qu'une formule est une tautologie. Ces systèmes de preuve sont orientés soit vers la preuve de tautologie (utilisation de la sémantique des formules, comme le fait la méthode de Quine [26]), soit vers la preuve de théorème à travers des manipulations syntaxiques (méthode des séquents de Gentzen [65], mise sous forme de clauses [54]). C'est dans cette dernière approche qu'on trouve la nécessité de "bien" représenter les formules propositionnelles. Bien représenter les formules signifie que les formules subissent un traitement qui permettront de faciliter la preuve de tautologie, mais aussi de *combiner* ces formules avec des algorithmes simples. Bien entendu, se pose la question de la *complexité* de la réécriture, de la preuve de tautologie, et des combinaisons.

Ainsi la technique de preuve basée sur la mise en forme de clause utilise un système de réécriture normal (voir Annexe A), et un principe de résolution [54] appelé *règle de coupure*. Le système ci-dessous permet de transformer toute formule en une forme normale, appelée *forme normale disjonctive*, ou encore *somme de produits*, analogue à la forme clausale à une négation près.

$$\begin{aligned}\neg\neg f &\rightarrow f \\ \neg(f \vee g) &\rightarrow \neg f \wedge \neg g \\ \neg(f \wedge g) &\rightarrow \neg f \vee \neg g \\ f \wedge (g \vee h) &\rightarrow (f \wedge g) \vee (f \wedge h) \\ (f \vee g) \wedge h &\rightarrow (f \wedge h) \vee (g \wedge h)\end{aligned}$$

Ce système n'est pas canonique. Par exemple, les formules $((\neg x \wedge y) \vee (\neg y \wedge z) \vee (\neg z \wedge x))$ et $((x \wedge \neg y) \vee (y \wedge \neg z) \vee (z \wedge \neg x))$ sont deux formes disjonctives réduites, syntaxiquement

différentes mais équivalentes. Nous étudierons les performances de cette représentation dans la Section 2.6.1, à la fin de ce chapitre.

Un système de réécriture canonique permet de montrer qu'une formule est valide en la réduisant à sa forme canonique, et en vérifiant que celle-ci est syntaxiquement égale à celle de la tautologie, que nous noterons 1. Jusqu'à récemment on ne connaissait que relativement peu de formes canoniques de la logique propositionnelle : les tables de vérité, la forme canonique de *Blake* [26], la forme *polynomiale*, appelée aussi *somme exclusive de produits*, ou forme de *Reed-Muller* [26, 109] (voir Section 2.6.2). Or ces formes présentent toutes le défaut d'être très peu compactes, dans le sens où le nombre de caractères nécessaires pour écrire la forme canonique d'une formule f peut être très grand par rapport au nombre de caractères de f . Plus récemment, de nouvelles formes canoniques ont été définies, les *graphes de décision* [27] (BDD pour *Binary Decision Diagrams*) et les *graphes de décision typés* [18] (TDG pour *Typed Decision Graphs*).

Nous présentons ici ces deux formes canoniques, BDDs et TDGs. Nous présenterons aussi d'autres formes canoniques dérivées de l'expansion de Shannon. Nous donnons ensuite les complexités des algorithmes manipulant les BDDs et TDGs. Dans la suite, sauf exception précisée, on ne considère que des formules sans quantificateur.

2.2 Expansion de Shannon

Soit f une formule propositionnelle. On appelle *expansion de Shannon* de f par rapport à la variable x_k le couple de formules $f[x_k \leftarrow 0]$ et $f[x_k \leftarrow 1]$. L'expansion de Shannon¹ possède la propriété suivante :

$$f = ((\neg x_k \wedge f[x_k \leftarrow 0]) \vee (x_k \wedge f[x_k \leftarrow 1])) \quad (2.1)$$

Nous étendons ces définitions aux fonctions booléennes grâce à la correspondance existant entre les formules et les fonctions. L'expansion de Shannon de la fonction booléenne $\lambda[x_1 \dots x_n].f(x_1, \dots, x_n)$ par rapport à la variable x_1 est le couple de fonctions $(\lambda[x_2 \dots x_n].f(0, x_2, \dots, x_n), \lambda[x_2 \dots x_n].f(1, x_2, \dots, x_n))$. Le théorème de Shannon ci-dessous montre que l'expansion de Shannon est l'unique solution de l'équation 2.1.

Théorème 2.1 *Soit f une fonction booléenne de $\{0, 1\}^n$ dans $\{0, 1\}$. Il existe un couple unique (f_0, f_1) de fonctions booléennes de $\{0, 1\}^{n-1}$ dans $\{0, 1\}$ telle que*

$$f(x_1, x_2, \dots, x_n) = (\neg x_1 \wedge f_0(x_2, \dots, x_n)) \vee (x_1 \wedge f_1(x_2, \dots, x_n)).$$

Preuve. L'existence d'un tel couple de fonctions a déjà été montrée. Montrons maintenant son unicité. Supposons qu'il existe un couple (f'_0, f'_1) possédant la même propriété. Pour tous x_2, \dots, x_n , on a :

$$f(0, x_2, \dots, x_n) = (\neg 0 \wedge f_0(x_2, \dots, x_n)) \vee (0 \wedge f_1(x_2, \dots, x_n))$$

¹Par abus de langage, le terme $((\neg x_k \wedge f[x_k \leftarrow 0]) \vee (x_k \wedge f[x_k \leftarrow 1]))$ est aussi appelé expansion de Shannon de la formule f par rapport à x_k . La formule $f[x_k \leftarrow 1]$ (respectivement la formule $f[x_k \leftarrow 0]$) est aussi appelée le cofacteur de f par rapport à x_k (respectivement par rapport à $\neg x_k$).

$$\begin{aligned}
&= f_0(x_2, \dots, x_n) \\
&= (\neg 0 \wedge f'_0(x_2, \dots, x_n)) \vee (0 \wedge f'_1(x_2, \dots, x_n)) \\
&= f'_0(x_2, \dots, x_n)
\end{aligned}$$

donc les fonctions f_0 et f'_0 sont égales. On montre de même que f_1 et f'_1 sont elles aussi égales, d'où l'unicité du couple (f_0, f_1) . \square

2.2.1 Propriétés de l'expansion de Shannon

L'expansion de Shannon possède la propriété fondamentale décrite par le théorème suivant, qu'on appellera propriété d'orthogonalité.

Théorème 2.2 *Soient g et f deux formules, x et y deux variables différentes. Alors on a les deux identités*

$$\begin{aligned}
g[y \leftarrow f] &= (\neg f \wedge g[y \leftarrow 0]) \vee (f \wedge g[y \leftarrow 1]) \\
&= (\neg x \wedge g[x \leftarrow 0][y \leftarrow f]) \vee (x \wedge g[x \leftarrow 1][y \leftarrow f])
\end{aligned}$$

L'orthogonalité de l'expansion de Shannon permet de ramener la substitution d'une variable par une formule à une expansion de Shannon et des combinaisons logiques élémentaires. Si l'on interprète les formules par les fonctions qu'elles dénotent, la substitution d'une variable par une formule n'est rien d'autre que la composition de deux fonctions. Ceci fonde l'évaluation de la composition, qui permet d'accéder à n'importe quelle transformation fonctionnelle. En effet, un corollaire de ce résultat est le suivant :

Corollaire 2.1 *Soit $g(y_1, \dots, y_n)$ une formule dont les seules variables sont y_1, \dots, y_n . Alors pour toutes formules f_1, \dots, f_n et toute variable x , on a :*

$$\begin{aligned}
g(f_1, \dots, f_n) &= (\neg x \wedge g(f_1[x \leftarrow 0], \dots, f_n[x \leftarrow 0])) \vee \\
&\quad (x \wedge g(f_1[x \leftarrow 1], \dots, f_n[x \leftarrow 1]))
\end{aligned}$$

En particulier, \star étant l'un des opérateurs $\vee, \wedge, \Rightarrow, \Leftrightarrow, \oplus$, on a :

$$\begin{aligned}
\neg f &= (\neg x \wedge \neg f[x \leftarrow 0]) \vee (x \wedge \neg f[x \leftarrow 1]) \\
(f \star g) &= (\neg x \wedge (f[x \leftarrow 0] \star g[x \leftarrow 0])) \vee (x \wedge (f[x \leftarrow 1] \star g[x \leftarrow 1]))
\end{aligned}$$

Ceci signifie que l'expansion de Shannon de $(\neg f)$, de $(f \star g)$, ou de $g(f_1, \dots, f_n)$ se calcule directement à partir des expansions de Shannon des opérands de la combinaison booléenne considérée. C'est ce principe qui rend simple à réaliser les combinaisons des formes de Shannon, que nous présenterons Section 2.4.

2.2.2 Arbre de Shannon

Considérons une fonction booléenne f de $\{0, 1\}^n$ dans $\{0, 1\}$. L'expansion de Shannon de f par rapport à x_1 fournit un couple unique de fonctions notées (f_0, f_1) , de $\{0, 1\}^{n-1}$ dans $\{0, 1\}$, telles que

$$f = \lambda[x_1 \dots x_n].((\neg x_1 \wedge f_0(x_2, \dots, x_n)) \vee (x_1 \wedge f_1(x_2, \dots, x_n))).$$

Si cette expansion est réitérée sur les fonctions f_0 et f_1 , on obtient un arbre binaire, appelé *arbre de Shannon*, dont les feuilles sont les fonctions constantes 0 et 1.

La syntaxe des arbres de Shannon est définie à partir des symboles de variable x_1, \dots, x_n , et d'un nouveau symbole Δ . Voici une syntaxe contenant les arbres de Shannon :

$$\begin{aligned} \langle tree \rangle &::= \Delta(\langle var \rangle, \langle tree \rangle, \langle tree \rangle) \\ &::= 0 \mid 1 \\ \langle var \rangle &::= x_1 \mid \dots \mid x_n \end{aligned}$$

L'arbre de Shannon d'une formule f est obtenue par l'évaluation du terme $\mathbf{Tree}(1, f)$, où la fonction \mathbf{Tree} est décrite par :

$$(1 \leq k \leq n) \Rightarrow \tag{2.2}$$

$$\mathbf{Tree}(k, f) = \Delta(x_k, \mathbf{Tree}(k + 1, f[x_k \leftarrow 0]), \mathbf{Tree}(k + 1, f[x_k \leftarrow 1]))$$

$$\mathbf{Tree}(n + 1, 0) = 0 \tag{2.3}$$

$$\mathbf{Tree}(n + 1, 1) = 1 \tag{2.4}$$

La règle 2.2 est l'expansion de Shannon par rapport à x_k . Comme chaque expansion de Shannon donne un couple de fonctions unique, ce système de réécriture est canonique. Par exemple, l'arbre² de décomposition de la formule $f = (x_1 \wedge (x_3 \oplus x_4)) \vee (x_2 \wedge (x_3 \Leftrightarrow x_4))$ est représenté par la Figure 1. Mais l'arbre canonique de Shannon d'une formule de n variables possède 2^n feuilles et $2^n - 1$ structures $\Delta(-, -, -)$, ce qui ne constitue pas une bonne représentation.

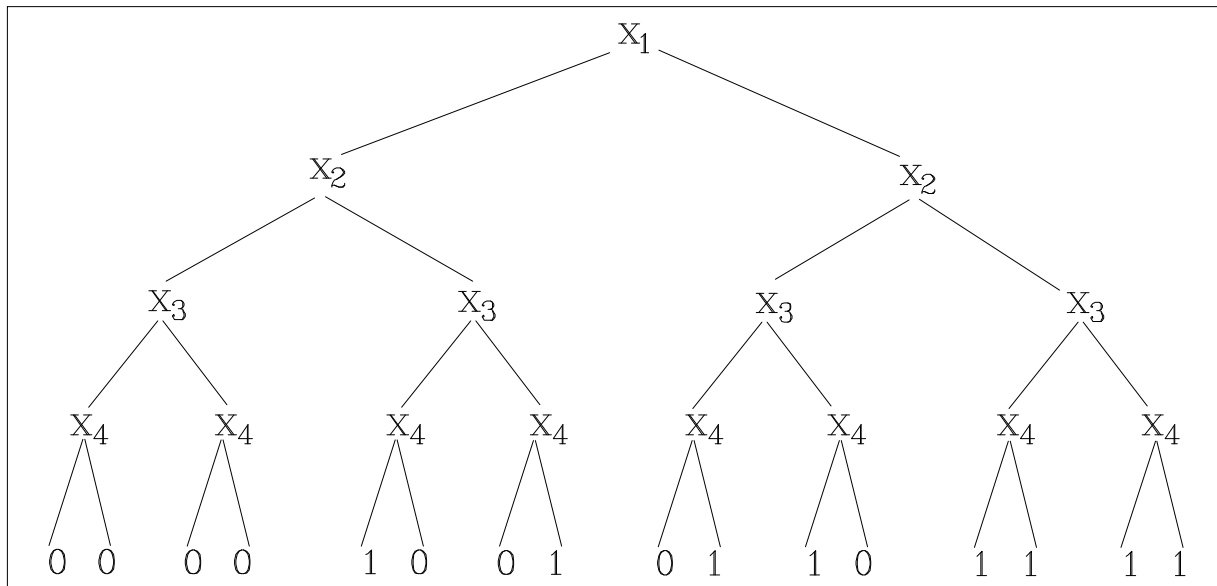


Figure 1. Arbre de Shannon de $(x_1 \wedge (x_3 \oplus x_4)) \vee (x_2 \wedge (x_3 \Leftrightarrow x_4))$.

²Cet arbre permet de calculer graphiquement la valeur de f pour toute interprétation i de ses variables, tout comme une table de vérité. On descend dans l'arbre en partant de sa racine, en choisissant à chaque noeud $\Delta(x_k, L, H)$ soit la branche gauche L si $i(x_k) = 0$, soit la branche droite H si $i(x_k) = 1$. La valeur de la feuille atteinte en fin de parcours est la valeur $i(f)$ recherchée.

Les opérations booléennes classiques peuvent être directement effectuées sur les arbres de Shannon. On utilise la propriété d'orthogonalité décrite par le Corollaire 2.1, illustrée par la Figure 2, qui est satisfaite pour tout opérateur booléen \star .

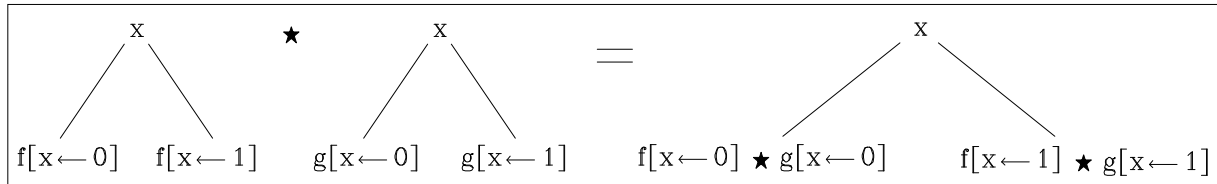


Figure 2. Orthogonalité de la forme de Shannon.

Voici un système de réécriture qui réalise les combinaisons booléennes, où les opérateurs \wedge , \Rightarrow , \Leftrightarrow et \oplus sont supposés être réécrits en fonction de \neg et \vee . Dans ces règles, t , L , H , L' , et H' dénotent des arbres de Shannon.

$$\begin{aligned} \neg \Delta(x, L, H) &\rightarrow \Delta(x, (\neg L), (\neg H)) \\ \neg 0 &\rightarrow 1 \\ \neg 1 &\rightarrow 0 \end{aligned}$$

$$\Delta(x, L, H) \vee \Delta(x, L', H') \rightarrow \Delta(x, L \vee L', H \vee H') \quad (2.5)$$

$$0 \vee t \rightarrow t \quad (2.6)$$

$$1 \vee t \rightarrow 1 \quad (2.7)$$

2.3 Formes canoniques graphiques

La forme de Shannon est une forme canonique peu intéressante, puisqu'elle est de taille $2^n - 1$ si n est le nombre de variables de la formule. Nous présentons ici deux processus de réduction qui permettent de transformer les arbres de Shannon en graphes orientés acycliques (DAGs pour *Directed Acyclic Graphs*), de taille beaucoup plus raisonnable. L'application de cette double réduction sur les arbres de Shannon conduit à une forme canonique compacte, les BDDs (*Binary Decision Diagrams* ou graphes de décision binaires). Nous présenterons ensuite les TDGs (*Typed Decision Graphs* ou graphes de décision typés), qui sont une amélioration des BDDs.

2.3.1 Élimination des noeuds redondants

Dans un arbre de Shannon, si toutes les feuilles accessibles à partir d'un noeud portent la même valeur, alors ce noeud est redondant, car il n'apporte aucune information sur la fonction, et il peut être remplacé par une feuille portant cette valeur. Nous appellerons *arbre réduit de Shannon* d'une fonction f l'arbre obtenu par élimination de tous les noeuds redondants de l'arbre de Shannon de f . L'arbre réduit de Shannon est évidemment canonique. La syntaxe donnée précédemment contient aussi les arbres réduits de Shannon. Pour obtenir l'arbre réduit, il suffit d'appliquer jusqu'à saturation la règle 2.8.

$$\Delta(-, H, H) \rightarrow H \quad (2.8)$$

Il faut noter que, contrairement à celle de l'arbre de Shannon, la taille de l'arbre réduit d'une formule dépend de l'ordre des variables selon lequel est effectuée l'expansion. La Figure 3 montre l'arbre réduit de Shannon de la formule $(x_1 \wedge (x_3 \oplus a_4)) \vee (x_2 \wedge (x_3 \Leftrightarrow x_4))$.

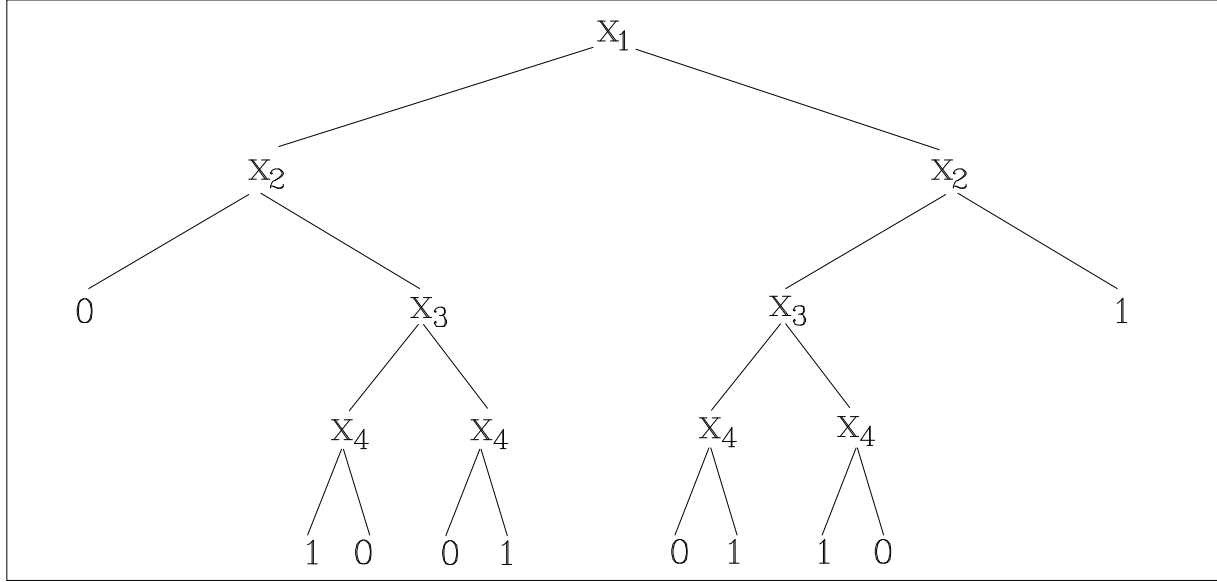


Figure 3. Arbre réduit de $(x_1 \wedge (x_3 \oplus a_4)) \vee (x_2 \wedge (x_3 \Leftrightarrow x_4))$.

La négation sur les arbres réduits est effectuée avec les mêmes règles de réécriture que celles utilisées pour les arbres de Shannon. Pour évaluer la disjonction de deux arbres réduits, on ajoute la règle suivante, et on utilisera la commutativité de la disjonction. Dans cette règle, on définit \preceq par : $x_k \preceq 0$; $x_k \preceq 1$; $x_k \preceq \Delta(x_j, -, -)$ si et seulement si $k \leq j$.

$$(x \prec t) \Rightarrow \\ t \vee \Delta(x, L, H) \rightarrow \Delta(x, (t \vee L), (t \vee H))$$

On notera dans la suite par $|t|$ la taille d'une représentation t , c'est à dire le nombre de structures $\Delta(-, -, -)$ contenues dans t . Le calcul des arbres réduits reste évidemment exponentiel, comme le montre le théorème suivant.

Théorème 2.3 *Le calcul de l'arbre réduit de Shannon d'une formule propositionnelle est exponentiel. Plus précisément, il existe au moins une formule de n variables, de taille $O(n)$, dont l'arbre réduit est de taille $2^n - 1$, et ceci quel que soit l'ordre choisi sur les variables pour effectuer l'expansion de Shannon.*

Preuve. Tout arbre réduit a une taille bornée par $2^n - 1$, où n est son nombre de variables. Soit la formule $(x_1 \oplus x_2 \oplus \dots \oplus x_n)$. Elle utilise n variables et sa taille est en $O(n)$. Nous allons montrer que la taille de son arbre réduit est $2^n - 1$.

Soit f_k la formule $(x_k \oplus x_{k+1} \oplus \dots \oplus x_n)$. Il est clair que $f_k[x_k \leftarrow 0] = f_{k+1}$, et que $f_k[x_k \leftarrow 1] = \neg f_{k+1}$. Donc x_k est forcément non redondante dans la formule f_k , et l'arbre réduit T_k de celle-ci s'écrit $T_k = \Delta(x_k, L_k, H_k)$, où L_k (respectivement H_k) est l'arbre réduit de f_{k+1} (respectivement $\neg f_{k+1}$). Soit s_k la taille de l'arbre T_k . On a

$s_k = |L_k| + |H_k| + 1$. Comme $L_k = \neg H_k$, on a $|L_k| = |H_k| = |T_{k+1}|$, d'où $s_k = 2s_{k+1} + 1$. Comme $T_n = \Delta(x_n, 0, 1)$, on a $s_n = 1$, et donc $s_k = 2^{n-k+1} - 1$. Finalement, la taille de la formule f_1 est $2^n - 1$.

On peut alors remarquer que $f_1(x_1, \dots, x_n) = 1$ si et seulement si il y a un nombre impair de 1 dans $[x_1 \dots x_n]$. Ceci signifie que la fonction f_1 est symétrique, c'est à dire invariante pour toute permutation de ses variables. En particulier, quel que soit l'ordre choisi sur les variables, la taille de son arbre réduit est $2^n - 1$. \square

La complexité du calcul des arbres réduits est liée à la complexité du problème consistant à savoir si une variable est redondante dans une formule. Une variable x est redondante dans une formule f si et seulement si $(f[x \leftarrow 0] \Leftrightarrow f[x \leftarrow 1])$. Or tester si une variable x est redondante sur un arbre réduit consiste simplement à parcourir l'arbre pour vérifier l'absence d'occurrence de x , ce qui est fait linéairement par rapport à la taille de l'arbre réduit, alors que le test de redondance est NP-complet sur une formule.

Théorème 2.4 *Savoir si une variable est redondante dans une formule propositionnelle est un problème NP-complet.*

Preuve. Soit f une formule propositionnelle de longueur n . Considérons la formule f' définie par $(f \wedge x)$, où la variable x n'a pas d'occurrence dans f . Cette formule f' est construite en $O(n)$. On a évidemment $f'[x \leftarrow 0] = 0$, et $f'[x \leftarrow 1] = f$. Alors f est satisfiable si et seulement si x n'est pas redondante dans f' , ce qui montre que le problème de la redondance est NP-difficile.

Réciproquement, soit f une formule de longueur n . On construit la formule f' définie par $(f[x \leftarrow 0] \oplus f[x \leftarrow 1])$ en $O(n)$. Alors x est redondante dans f si et seulement si f' n'est pas satisfiable. Donc le problème de la redondance se réduit polynomialement au problème de la satisfiabilité. \square

L'élimination des noeuds redondants diminue considérablement la taille des arbres de Shannon dans bien des cas. Une façon d'étudier l'intérêt des arbres réduits par rapport aux arbres de Shannon est de calculer la fonction de répartition de leurs tailles, ainsi que leur taille moyenne.

Soit N_s^n le nombre d'arbres réduits de taille s utilisant tout ou partie des variables $\{x_1, \dots, x_n\}$ avec un ordre fixé. On dispose des identités suivantes. Les fonctions admettant un arbre réduit de taille nulle sont les fonctions qui ne possèdent aucune variable non redondante, c'est à dire les deux fonctions constantes 0 et 1, d'où $N_0^n = 2$. Les fonctions sur n variables x_1, \dots, x_n qui possèdent exactement une seule variable non redondante sont les fonctions du type $\lambda \vec{x}.(x_k)$ ou $\lambda \vec{x}.(\neg x_k)$ pour $1 \leq k \leq n$, d'où $N_1^n = 2n$. Une fonction sans variable est constante, donc de taille nulle, d'où $N_s^0 = 0$ pour $s > 0$. La taille maximale d'un arbre réduit sur n variables est $2^n - 1$, d'où $N_s^n = 0$ pour $s > 2^n - 1$. Le nombre de fonctions de $\{0, 1\}^n$ vers $\{0, 1\}$ est 2^{2^n} , d'où $(\sum_{s=0}^{\infty} N_s^n) = 2^{2^n}$.

Le nombre N_s^n est la somme de N_s^{n-1} , qui est le nombre d'arbres réduits de taille s utilisant tout ou partie de $\{x_1, \dots, x_{n-1}\}$, et du nombre d'arbres réduits de taille s utilisant la variable x_n . On construit un arbre réduit $\Delta(x_n, t, t')$ de taille s utilisant la variable x_n

$s \setminus n$	0	1	2	3	4	5
0	2	2	2	2	2	2
1		2	4	6	8	10
2			8	24	48	80
3			2	46	172	420
4				72	544	2000
5				72	1464	8648
6				32	3392	34464
7				2	6520	125148
8					10472	418504
9					13296	1284144
10					13408	3619744
11					9904	9356792
12					4896	22163360
13					1280	48026690
14					128	95006460
15					2	170962500
16						278598300
17						408770400
18						536272800
19						623837700
20						636879100
21						563235300
22						424271100
23						266115100
24						134750100
25						52805790
26						15122820
27						2930112
28						347264
29						21376
30						512
31						2
s moyen	0	0.5	1.625	4.085938	9.152008	19.30384

Table 1. Répartition par taille des arbres réduits de Shannon.

en choisissant deux arbres réduits t et t' sur $\{x_1, \dots, x_{n-1}\}$, tels que $|t| + |t'| = s - 1$ et $t \neq t'$. On en déduit les équations suivantes :

$$N_{2s}^n = N_{2s}^{n-1} + \left(\sum_{k=0}^{2s-1} N_k^{n-1} \times N_{2s-k-1}^{n-1} \right) \quad (2.9)$$

$$N_{2s+1}^n = N_{2s+1}^{n-1} + \left(\sum_{k=0}^{2s} N_k^{n-1} \times N_{2s-k}^{n-1} \right) - N_s^{n-1} \quad (2.10)$$

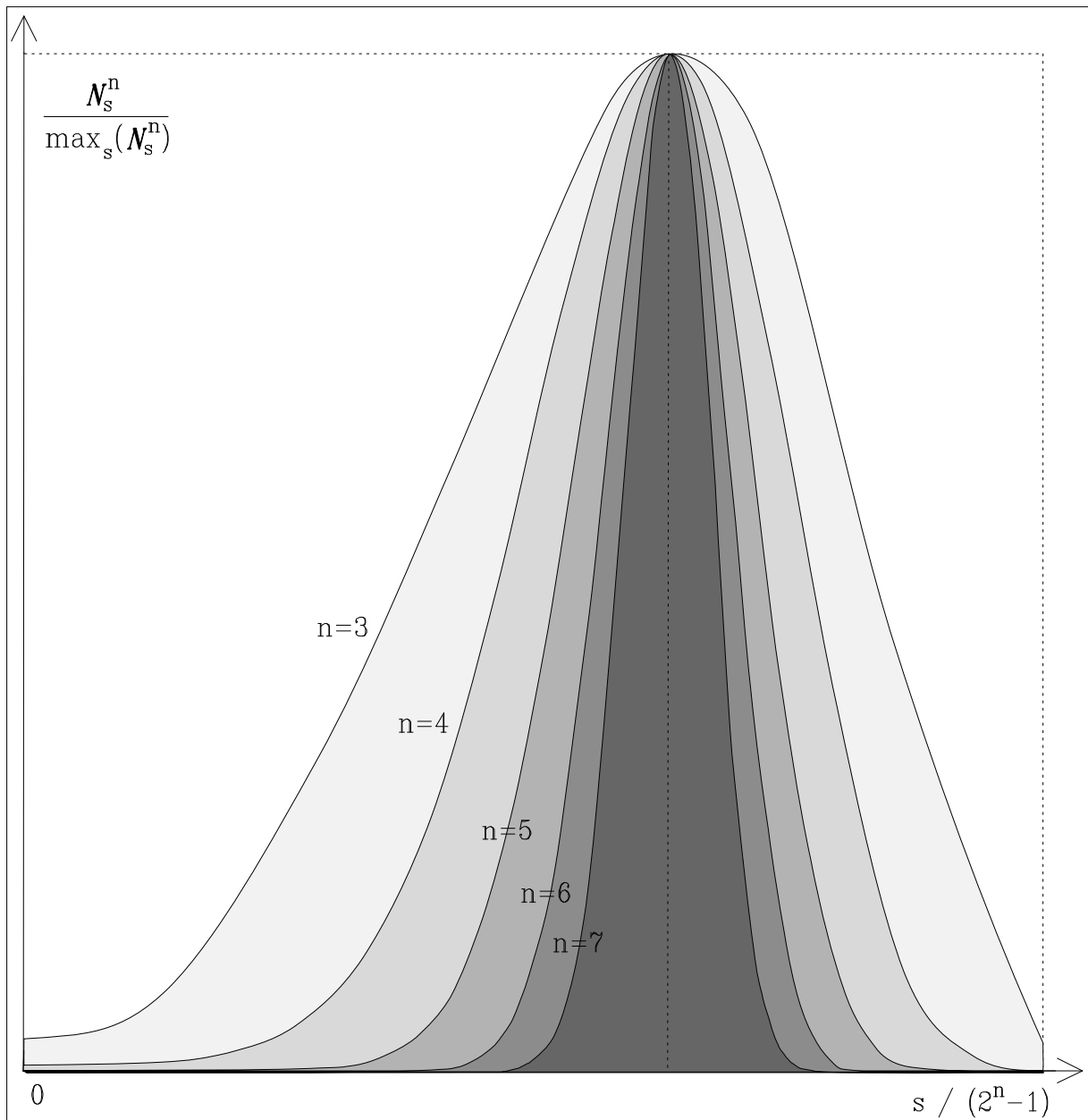


Figure 4. Distribution normalisée des arbres réduits.

La Table 1 donne les valeurs de N_s^n pour $0 \leq n \leq 5$. La taille moyenne pour $n = 6$ est 39.60765, avec un maximum ($1.951548 \cdot 10^{18}$) d'arbres de taille 40. La taille moyenne pour $n = 7$ est 80.21515, avec un maximum ($2.537742 \cdot 10^{37}$) d'arbres de taille 81.

La Figure 4 montre les distributions normalisées des arbres réduits de Shannon construits sur n variables en fonction de leur taille, pour $3 \leq n \leq 7$. La distribution N_s^n a été normalisée en recadrant les valeurs pertinentes de N_s^n dans le carré $[0, 1] \times [0, 1]$, ce qui est réalisé en traçant les points (x, y) paramétrés par s pour n fixé, et définis par :

$$x = \frac{s}{2^n - 1}$$

$$y = \frac{N_s^n}{\max_s(N_s^n)}$$

Ces courbes de distribution normalisées montrent que le maximum de la fonction $\lambda_s(N_s^n)$, qui semble en $2^{2^{\Theta(n)}}$, est obtenu pour une valeur de $(s/(2^n - 1))$ qui converge très vite vers une constante C' , et dont la valeur expérimentale est assez proche de la constante C définie dans le Théorème 2.5 présenté dans la suite. La distribution normalisée semble converger vers une fonction nulle partout sauf en C' , mais nous n'en avons pas une preuve explicite.

Nous pourrions expliciter N_s^n en calculant $g_n(y) = (\sum_{k=0}^{\infty} N_k^n y^k)$, qui est la fonction génératrice [69] associée à $(N_k^n)_{k \geq 0}$. L'équation de la fonction génératrice peut s'obtenir à partir des équations 2.9 et 2.10, ou plus directement à partir de l'égalité

$$\mathcal{A}_n = \mathcal{A}_{n-1} \cup \{x_n\} \times (\mathcal{A}_{n-1} \times \mathcal{A}_{n-1} - \{(t, t) / t \in \mathcal{A}_{n-1}\}),$$

et l'utilisation des lemmes élémentaires de dénombrement [69, 122], d'où on obtient :

$$g_n(y) = g_{n-1}(y) + y(g_{n-1}^2(y) - g_{n-1}(y^2))$$

Nous ne sommes pas parvenus à résoudre cette équation. La difficulté du problème vient, entre autres, que les arbres réduits ne peuvent pas se décrire avec un système de réécriture régulier [39]. Sans doute des chercheurs plus aguerris y parviendront [122, 115, 61, 62, 2, 3, 33]. Cependant, nous pouvons obtenir la taille moyenne d'un arbre réduit de Shannon sans expliciter la fonction N_s^n , comme le montre le théorème suivant.

Théorème 2.5 *Soit t_n la taille moyenne d'un arbre réduit utilisant tout ou partie des variables x_1, \dots, x_n . On a l'égalité :*

$$t_n = 2^n \left(\prod_{k=0}^{n-1} \left(1 - \frac{1}{2^{2^k+1}}\right) \right) - 1,$$

c'est à dire :

$$t_n \approx 0.63449 \times 2^n,$$

qui est donc exponentielle par rapport à n .

Preuve. Soit \mathcal{A}_n l'ensemble des arbres réduits utilisant tout ou partie des variables x_1, \dots, x_n , et t_n la taille moyenne d'un élément de \mathcal{A}_n . Par définition, on a l'égalité $t_n = (\sum_{t \in \mathcal{A}_n} |t|) / 2^{2^n}$, d'où on obtient :

$$\begin{aligned} 2^{2^n} t_n &= \sum_{t \in \mathcal{A}_n} |t| \\ &= \sum_{t \in \mathcal{A}_{n-1}} |t| + \sum_{\substack{t \in \mathcal{A}_{n-1} \\ t' \in \mathcal{A}_{n-1} \\ t \neq t'}} (|t| + |t'| + 1) \\ &= 2^{2^{n-1}} t_{n-1} + \sum_{t \in \mathcal{A}_{n-1}} \sum_{\substack{t' \in \mathcal{A}_{n-1} \\ t' \neq t}} (|t| + |t'| + 1) \\ &= 2^{2^{n-1}} t_{n-1} + \sum_{t \in \mathcal{A}_{n-1}} \left(\left(\sum_{t' \in \mathcal{A}_{n-1}} |t| + |t'| + 1 \right) - 2|t| - 1 \right) \end{aligned}$$

$$\begin{aligned}
&= 2^{2^{n-1}} t_{n-1} + \sum_{t \in \mathcal{A}_{n-1}} \left(2^{2^{n-1}} - 1 + (2^{2^{n-1}} - 2)|t| + \sum_{t' \in \mathcal{A}_{n-1}} |t'| \right) \\
&= 2^{2^{n-1}} t_{n-1} + 2^{2^{n-1}} (2^{2^{n-1}} - 1) + (2^{2^{n-1}} - 2) \left(\sum_{t \in \mathcal{A}_{n-1}} |t| \right) + 2^{2^{n-1}} \left(\sum_{t' \in \mathcal{A}_{n-1}} |t'| \right) \\
&= 2^{2^{n-1}} t_{n-1} + 2^{2^{n-1}} (2^{2^{n-1}} - 1) + (2^{2^{n-1}+1} - 2) 2^{2^{n-1}} t_{n-1} \\
&= 2^{2^{n-1}} (2^{2^{n-1}+1} - 1) t_{n-1} + 2^{2^{n-1}} (2^{2^{n-1}} - 1)
\end{aligned}$$

Cette égalité s'écrit aussi $2^{2^n} t_n = 2^{2^{n-1}} (2^{2^{n-1}} - 1) (2t_{n-1} + 1) + 2^{2^{n-1}} t_{n-1}$, ce qui exprime la conservation de la moyenne de la taille des arbres réduits de Shannon par construction de \mathcal{A}_n à partir de \mathcal{A}_{n-1} . On obtient finalement $2^{2^n} t_n = (2^{2^{n-1}+1} - 1) t_{n-1} + 2^{2^{n-1}} - 1$. Posons $t_n = q_n - 1$. On a alors :

$$2^{2^{n-1}} q_n = (2^{2^{n-1}+1} - 1) q_{n-1},$$

c'est à dire, comme $q_0 = 1$:

$$t_n = 2^n \left(\prod_{k=0}^{n-1} \left(1 - \frac{1}{2^{2^k+1}} \right) \right) - 1$$

Le terme $(\prod_{k=0}^{n-1} (1 - \frac{1}{2^{2^k+1}}))$ converge, quand n tend vers l'infini, vers une constante $C \approx 0.63449$. \square

En conclusion, si l'élimination des noeuds redondants permet certes de réduire la taille de la représentation, la taille moyenne reste exponentielle par rapport au nombre de variables, et la courbe de répartition $-N_s^n$ en fonction de s pour n fixé — montre qu'une majeure partie des arbres ont une taille non polynomiale. Ceci est montré par le Théorème 2.6, qui donne un résultat sur la répartition des tailles des arbres réduits, similaire à celui obtenu [109] sur la répartition des tailles des formules³.

Théorème 2.6 *Pour tout $\epsilon > 0$ fixé, les arbres réduits de Shannon des fonctions de n variables ont presque partout une taille supérieure ou égale à*

$$(1 - \epsilon) \frac{2^n}{\log_2 n}.$$

Preuve. A tout arbre nous associons une unique chaîne de caractères, construite sur l'alphabet $\{x_1, \dots, x_n, 0, 1\}$, et produite par le parcours en profondeur d'abord et à gauche d'abord de cet arbre. Ceci correspond à une notation préfixée des arbres, en sachant que 0 et 1 sont les terminaux. Par exemple, la chaîne $x_1 x_2 0 1 x_2 x_3 0 1 1$ dénote l'arbre $\Delta(x_1, \Delta(x_2, 0, 1), \Delta(x_2, \Delta(x_3, 0, 1), 1))$.

³Cependant, comme le dénombrement effectué dans la preuve du Théorème 2.6 ne tient pas compte de la contrainte de répartition des variables dans l'arbre (toute variable qui occure dans les sous-arbres L et H de l'arbre $\Delta(x, L, H)$ doit avoir un ordre strictement supérieur à celui de x), il se pourrait qu'il existe une borne inférieure (presque partout) plus précise, par exemple en $O(2^n)$, au vu de la courbe de répartition, et du fait que la taille moyenne est en $C \times 2^n$ et que l'arbre de taille maximale est en 2^n . Bien que cela semble peu vraisemblable, la question reste ouverte.

Un arbre de s noeuds possède $2s$ branches, une partie de ces branches pointant sur $s - 1$ noeuds distincts (construisant ainsi un arbre de taille s avec un noeud racine), les autres branches, au nombre de $2s - (s - 1)$, pointant sur un terminal 0 ou 1. Un arbre réduit de Shannon de taille s possède exactement $s + 1$ feuilles, donc la chaîne le dénotant a une taille $2s + 1$.

Soit N_s^n le nombre d'arbres réduits de Shannon de taille s construits sur n variables. Une majoration de N_s^n consiste à dénombrer le nombre de chaînes sur l'alphabet $\{x_1, \dots, x_n, 0, 1\}$, de taille $2s + 1$, comprenant s occurrences de symboles de variable pris dans l'ensemble $\{x_1, \dots, x_n\}$. Il suffit pour cela de choisir la place de s symboles parmi $2s + 1$, d'assigner chacune de ces s places à l'une des n variables, puis d'assigner chacune des $s + 1$ places restantes à la constante 0 ou 1. On a donc :

$$N_s^n \leq \binom{2s+1}{s} n^s 2^{s+1}$$

Par la formule de Stirling $k! \sim \sqrt{2\pi k} \left(\frac{k}{e}\right)^k$, on calcule un équivalent du membre droit de l'inégalité pour obtenir :

$$N_s^n \leq \frac{4}{\sqrt{\pi s}} n^s 2^{3s}$$

Ceci permet de majorer le terme $\mathcal{N}_s^n = (\sum_{k=0}^s N_k^n)$, c'est à dire le nombre d'arbres réduits de Shannon construits sur n variables de taille inférieure ou égale à s , par le terme $\frac{32}{\sqrt{\pi}} n^{s+1} 2^{3s}$. En posant $s = \alpha \frac{2^n}{\log_2 n}$, on obtient :

$$\mathcal{N}_s^n \leq \frac{32}{\sqrt{\pi}} 2^{\alpha 2^n (1 + \frac{3}{\log_2 n})}$$

Pour $\alpha < 1$ fixé, il suffit de prendre n assez grand pour avoir $\mathcal{N}_s^n \ll 2^{2^n}$, ce qui signifie que le nombre de fonctions de n variables ayant un arbre réduit de Shannon de taille inférieure ou égale à $\alpha \frac{2^n}{\log_2 n}$ est négligeable devant le nombre total de fonctions à n variables.

Remarquons que dans la majoration de \mathcal{N}_s^n , on ne tient pas compte de la contrainte de l'ordre des variables. Ceci signifie que ce théorème tient si on associe à chaque fonction son arbre réduit de Shannon *minimal*, c'est à dire celui construit avec un ordre *optimal*. \square

2.3.2 Graphes de décision binaires

La taille de la représentation des formules propositionnelles qui a été présentée dans les pages précédentes peut encore être réduite par partage des sous-arbres identiques [102, 27]. Le partage des sous-arbres d'un arbre réduit transforme celui-ci en un graphe acyclique orienté, où les noeuds sont des structures $\Delta(-, -, -)$. En supposant que le processus de réécriture d'une formule en un arbre de Shannon numérote les structures $\Delta(-, -, -)$ qui sont créées (c'est à dire lors de l'utilisation d'une règle de type 2.2) par un indice croissant, le partage des sous-arbres identiques se fait en utilisant la règle :

$$\begin{aligned} \text{si } \Delta_i(x, L, H) \text{ et } \Delta_j(x, L, H) \text{ ont été créés, et si } i < j, \text{ alors} \\ \Delta_j(x, L, H) \rightarrow \Delta_i(x, L, H) \end{aligned} \quad (2.11)$$

On appelle *graphe de décision binaire*, ou BDD, d'une fonction f le graphe obtenu en partageant tous les sous-arbres identiques de l'arbre de Shannon réduit de f . La Figure 5 montre le BDD de la formule $(x_1 \wedge (x_3 \oplus x_4)) \vee (x_2 \wedge (x_3 \Leftrightarrow x_4))$. Modulo l'ordre des variables, les BDDs [27] sont une représentation canonique des fonctions booléennes. Bien entendu, la taille du graphe obtenu dépend de l'ordre des variables suivant lequel est effectuée l'expansion de Shannon. Cette transformation de l'arbre réduit en graphe permet de diminuer considérablement la taille de la représentation d'une formule. Par exemple, la formule $(x_1 \Leftrightarrow (x_2 \Leftrightarrow (x_3 \Leftrightarrow \dots (x_{n-2} \Leftrightarrow (x_{n-1} \Leftrightarrow x_n)) \dots)))$ qui possède un arbre réduit de taille $2^n - 1$ pour tout ordre, admet un BDD de taille $4n - 1$ pour n'importe quel ordre.

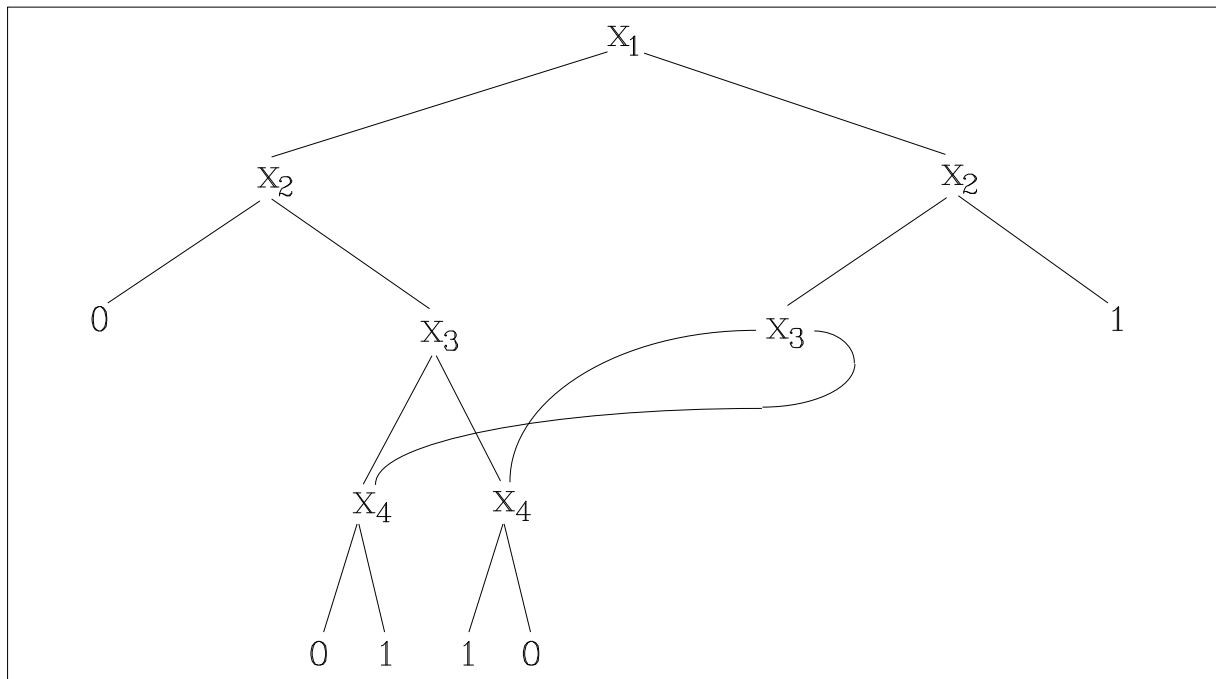


Figure 5. BDD de $(x_1 \wedge (x_3 \oplus x_4)) \vee (x_2 \wedge (x_3 \Leftrightarrow x_4))$.

Le théorème 2.7 affirme que la taille d'un graphe de décision est non polynomiale dans le pire cas.

Théorème 2.7 Soit f le graphe de décision d'une fonction de $\{0, 1\}^n$ dans $\{0, 1\}$. On a :

$$|f| = O\left(\frac{2^n}{n}\right)$$

Preuve. Nous allons construire un BDD de taille maximale, utilisant n variables, et montrer que sa taille est en $O(2^n/n)$. Considérons un arbre de Shannon à n variables $\{x_1, \dots, x_n\}$ complètement expansé. Il possède $2^n - 1$ noeuds, et chaque variable x_k apparaît 2^{k-1} fois dans l'arbre. La taille maximale est obtenue en étudiant le nombre maximal de graphes dont les variables sont incluses dans $\{x_k, \dots, x_n\}$, pour $1 \leq k \leq n$. Autrement dit, on va chercher le nombre maximal de graphes qui peuvent être construits sur le système $\{x_k, \dots, x_n\}$, ce qui nous donnera un graphe réduit de taille maximale sur les parties inférieures du graphe global, et nous indiquera la limite à partir de laquelle il

n'y a plus de partage possible. Soit N_k le nombre de graphes utilisant tout ou partie des variables $\{x_k, \dots, x_n\}$. C'est aussi le nombre de fonctions à $n - k + 1$ variables, donc :

$$N_k = 2^{2^{n-k+1}}$$

Un graphe de taille maximale est construit ainsi. Sur l'arbre de Shannon complètement expansé, on part du niveau n et on remonte vers la racine de telle façon qu'au niveau k , on construit —si c'est possible— tous les N_k graphes existants sur les variables $\{x_k, \dots, x_n\}$, afin de maximiser la taille du graphe résultant. On peut construire les N_k graphes au niveau k tant qu'il y a suffisamment de noeuds x_k dans l'arbre de Shannon. Comme il y a au plus 2^{k-1} noeuds x_k , ce processus continue tant que $N_k < 2^{k-1}$, et cesse pour le plus grand entier s tel que $N_s \geq 2^{s-1}$. Comme N_k décroît et 2^{k-1} croît quand k augmente, s est la partie entière du réel x , avec x satisfaisant $N_x = 2^{x-1}$. En notant $maxsize(n)$ la taille maximale d'un BDD utilisant n variables, on a :

$$\begin{aligned} maxsize(n) &= \sum_{k=1}^s 2^{k-1} + \sum_{k=s+1}^n N_k \\ &= 2^s - 1 + 2^{2^{n-s}} + \sum_{k=s+2}^n 2^{2^{n-k+1}} \\ &= 2^s + 2^{2^{n-s}} + o(2^{2^{n-s}}) \end{aligned}$$

L'équation $N_x = 2^{x-1}$ s'écrit $2^{n-x+1} = x - 1$, soit $n = x - 1 + \log_2(x - 1)$ pour $n > 0$. Posons $x = 2^y + 1$. On a donc la solution de l'équation $N_x = 2^{x-1}$ décrite par un paramètre réel y déterminé de façon unique par n :

$$\begin{aligned} n &= 2^y + y, & \text{et} \\ x &= 2^y + 1. \end{aligned}$$

Comme $s = \lfloor x \rfloor$, s est un entier tel que $2^y < s \leq 2^y + 1$, soit $s = 2^y + \alpha(y)$, avec $0 < \alpha(y) \leq 1$. On a donc :

$$maxsize(n) = 2^{2^y + \alpha(y)} + 2^{2^{y-\alpha(y)}} + o(2^{2^{y-\alpha(y)}})$$

On déduit immédiatement :

$$2^{-2^y} \times maxsize(n) = 2^{\alpha(y)} + 2^{2^{y-\alpha(y)-2^y}} + o(2^{2^{y-\alpha(y)-2^y}})$$

Il est aisé de voir que $2^{2^y} \sim \frac{2^n}{n}$ quand $y \rightarrow +\infty$. On a donc :

$$maxsize(n) \sim \frac{2^n}{n} (2^{\alpha(y)} + 2^{2^{y-\alpha(y)-2^y}})$$

On a donc $maxsize(n) = O(2^n/n)$, puisque $0 < \alpha(y) \leq 1$. Comme il n'existe qu'un facteur constant entre les graphes binaires et typés, la taille du graphe maximal sur n variables est en $O(2^n/n)$.

Pour être plus précis, nous montrons maintenant que la fonction $(2^{\alpha(y)} + 2^{2^{y-\alpha(y)-2^y}})$ (avec $0 < \alpha(y) \leq 1$) est asymptotiquement incluse et dense dans $[1, 2]$ pour $y \rightarrow +\infty$. Ceci montrera que la taille maximale d'un BDD sur n variables varie asymptotiquement, en équivalence, et continûment, entre $\frac{2^n}{n}$ et $\frac{2^{n+1}}{n}$.

Prenons une séquence d'entiers n tendant vers l'infini, telle que $0 < \epsilon \leq \alpha(y) \leq 1$. Sur cette séquence, on a évidemment $\text{maxsize}(n) \sim 2^{\alpha(y)} \times 2^n/n$. Si de plus $\alpha(y)$ converge vers une constante α , alors tout point de $[2^\epsilon, 2]$ peut être atteint par $2^{\alpha(y)}$ à l'infini. Par exemple, la borne supérieure 2 est atteinte pour $\alpha(y)$ constamment égal à 1, ce qui est réalisé entre autres pour la séquence $n = 2^p + p$ avec $y = p$ entier tendant vers l'infini.

Étudions maintenant le comportement de $(2^{\alpha(y)} + 2^{2^y - \alpha(y) - 2^y})$ pour une séquence d'entiers n tendant vers l'infini, telle que $\alpha(y) \rightarrow 0$. On a alors :

$$2^{\alpha(y)} + 2^{2^y - \alpha(y) - 2^y} \underset{\substack{y \rightarrow +\infty \\ \alpha(y) \rightarrow 0}}{=} 1 + 2^{-\alpha(y) \frac{2^y}{\log 2}}$$

Pour $\alpha(y) = f(y)/2^y$ avec $f(y) = o(2^y)$ (car $\alpha(y) \rightarrow 0$), et $\lim_{y \rightarrow +\infty} f(y) = +\infty$, le membre de droite converge vers 1, donc la borne inférieure est bien asymptotiquement atteinte. Cependant, comme la pratique impose des valeurs de n relativement petite (de l'ordre de 100), le BDD maximal obtenu en pratique est “plutôt” de taille $\frac{2^{n+1}}{n}$. \square

Les résultats sur la taille moyenne des arbres réduits de Shannon donnés Section 2.3.1 montrent que la taille moyenne d'un BDD d'une fonction de $\{0, 1\}^n$ dans $\{0, 1\}$ est aussi non polynomiale en n , car le gain du partage ne peut être suffisant pour passer de l'exponentiel au polynomial. Ce dernier point s'explique en relevant qu'un arbre sans label de taille n ne peut espérer en moyenne qu'une taille $O(n/\sqrt{\log n})$ après partage des sous-arbres identiques [62].

Le Théorème 2.8 donne un résultat sur la répartition des tailles des BDDs, similaire à celui obtenu [109] sur la répartition des tailles des circuits. Comme la taille maximale d'un BDD est en $\beta(n) \frac{2^n}{n}$ avec $1 \leq \beta(n) \leq 2$, le résultat du Théorème 2.8 est *optimal*⁴. On déduit de ce théorème que la taille moyenne des BDDs est asymptotiquement (en équivalence) comprise entre $\frac{2^n}{n}$ et $\frac{2^{n+1}}{n}$.

Théorème 2.8 *Pour tout $\epsilon > 0$ fixé, les BDDs des fonctions de n variables ont presque partout une taille supérieure ou égale à*

$$(1 - \epsilon) \frac{2^n}{n}.$$

Preuve. La preuve est similaire à celle du Théorème 2.6, au dénombrement près. Sur l'alphabet $\{[k : x_j] / 1 \leq j \leq n, 1 \leq k \leq 2^n\} \cup \{[k] / 1 \leq k \leq 2^n\} \cup \{0, 1\}$, on construit une unique chaîne de caractères produite par le parcours d'un BDD en profondeur d'abord et à gauche d'abord, avec étiquetage des noeuds atteints. Ceci permet d'écrire un graphe en notation préfixée, en sachant que 0, 1, et $[k]$ sont les terminaux. Par exemple, la chaîne

⁴La courbe de répartition des BDDs présente une accumulation majoritaire au-dessus de la valeur moyenne : assez curieusement, le gain dû au partage est surtout effectif pour les “petits” arbres, la compaction des “gros” arbres, qui constituent la majorité, ne contribuant que très peu au gain de la représentation sous forme de BDD. Encore plus curieusement, pour les valeurs de n telles que la taille maximale se rapproche de $\frac{2^n}{n}$ (i.e. $\beta(n)$ tend vers 1), la valeur moyenne est aussi en $\frac{2^n}{n}$, ce qui montre l'existence d'un pic d'accumulation autour de la valeur maximale : pour une telle séquence de n , la probabilité —après avoir ramené l'espace R^2 sur un carré $[0, 1]^2$ — d'avoir un graphe de taille non nulle tend vers zéro!

$[1 : x_1][2 : x_3][3 : x_4][4 : x_5]10[5 : x_5]01[6 : x_4][5][4][7 : x_2][2][8 : x_3][9 : x_4][5]1[6]$ dénote le graphe de la figure suivante.

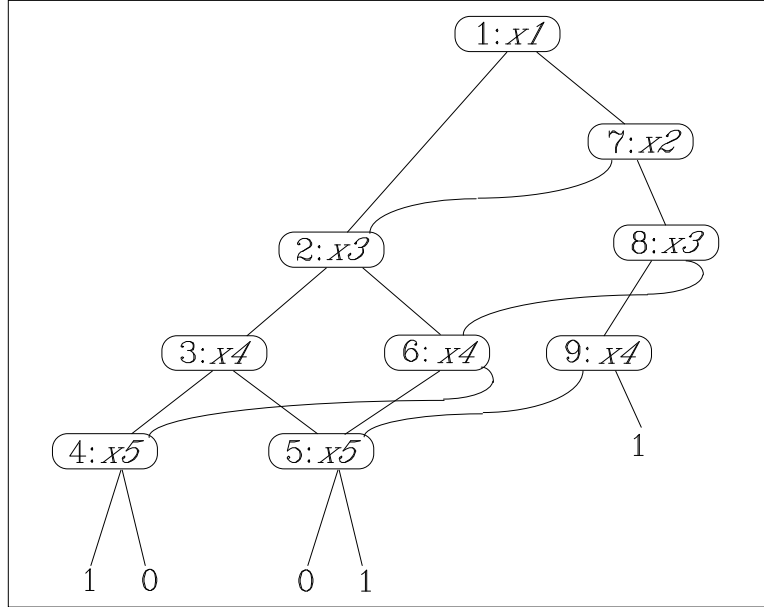


Figure 6. Exemple de BDD dénoté par une chaîne.

Un graphe de taille s est ainsi dénoté par une chaîne de taille $2s + 1$, constituée de l'étiquetage des s noeuds (du type $[k : x_j]$) et des $s + 1$ terminaux (du type $0, 1$, ou $[k]$). Soit N_s^n le nombre de BDDs de taille s construits sur n variables. Une majoration de N_s consiste à dénombrer le nombre de chaînes construites sur l'alphabet décrit plus haut, de taille $2s + 1$, comprenant s occurrences de symboles $[k : x_j]$. Il suffit pour cela de choisir s places parmi $2s + 1$, d'assigner ces s places à l'une des n variables (l'étiquetage étant fait par ordre croissant), puis d'assigner les $s + 1$ places restantes aux terminaux $0, 1$, ou $[k]$, avec $1 \leq k \leq s - 1$ (car tout noeud peut être référencé, c'est à dire partagé, sauf la racine). On a donc :

$$N_s^n \leq \binom{2s + 1}{s} n^s (s + 1)^{s+1}$$

Par la formule de Stirling $k! \sim \sqrt{2\pi k} \left(\frac{k}{e}\right)^k$, on obtient un équivalent du terme de droite, ce qui donne :

$$N_s^n \leq \sqrt{\frac{e}{\pi s}} n^s 2^s (2s + 1)^{s+1}$$

Ceci permet de majorer asymptotiquement le terme $\mathcal{N}_s^n = (\sum_{k=0}^s N_s^n)$, c'est à dire le nombre de BDDs de taille inférieure ou égale à s , par le terme de droite de la dernière inégalité, à une constante c près. En posant $s = \alpha \frac{2^n}{n}$, on obtient :

$$\mathcal{N}_s^n \leq c 2^{\alpha 2^n (1 + \frac{\log_2 4\alpha}{n})}$$

Pour $\alpha < 1$ fixé, il suffit de prendre n assez grand pour avoir $\mathcal{N}_s^n \ll 2^{2^n}$, ce qui signifie que le nombre de fonctions de n variables ayant un BDD de taille inférieure ou égale à $\alpha \frac{2^n}{n}$ est négligeable devant le nombre total de fonctions à n variables.

Remarquons que dans la majoration de \mathcal{N}_s^n , on ne tient pas compte de la contrainte de l'ordre des variables. Ceci signifie que ce théorème est vrai si on associe à chaque fonction son BDD *minimal*, c'est à dire celui construit avec un ordre *optimal*. \square

2.3.3 Graphes de décision typés

L'arbre de Shannon de la fonction $(\neg f)$ est obtenu à partir de l'arbre de Shannon de f en effectuant une copie de ce dernier, et en complétant ses feuilles (les feuilles 0 sont changées en 1 et réciproquement). La négation sur les arbres de Shannon s'effectue donc en $O(|f|)$. La justification des arbres typés de Shannon est de fournir un processus de négation beaucoup plus efficace, en $O(1)$.

L'idée utilisée, proposée initialement par S. B. Akers [4], est d'indiquer les négations au lieu de les réaliser. Ceci consiste à typer les arcs des arbres de Shannon. On obtient ainsi une nouvelle représentation arborescente des fonctions booléennes, dans laquelle une structure $\Delta(-, -, -)$ typée "+" dénote l'arbre de Shannon, cette même structure typée "-" indiquant qu'il faut compléter l'arbre représenté pour obtenir l'arbre de Shannon correspondant. Voici une syntaxe contenant les arbres typés de Shannon :

$$\begin{aligned} \langle tree \rangle & ::= \langle type \rangle \langle vertex \rangle \\ \langle vertex \rangle & ::= \Delta(\langle var \rangle, \langle tree \rangle, \langle tree \rangle) \\ & ::= 1 \\ \langle type \rangle & ::= + \mid - \\ \langle var \rangle & ::= x_1 \mid \dots \mid x_n \end{aligned}$$

L'interprétation des arbres typés de Shannon exprimée avec la forme de Shannon est la suivante :

$$\begin{aligned} +\Delta(x, L, H) & = \Delta(x, L, H) \\ -\Delta(x, L, H) & = \neg\Delta(x, L, H) \end{aligned}$$

Le problème est qu'il y a plusieurs façons de disposer les types pour dénoter une même fonction. Il faut donc trouver des règles qui rendent cette représentation canonique. Un moyen de rendre canonique cette forme est présenté ici [18, 89].

Une fonction f est dite *positive* si $f(1, 1, \dots, 1) = 1$, sinon elle est dite *négative*⁵. On peut aussi définir inductivement les fonctions positives et négatives : la fonction 1 est

⁵La négation réalise un isomorphisme entre $(\mathcal{F}_+^n, \vee, \wedge)$ et $(\mathcal{F}_-^n, \wedge, \vee)$, où \mathcal{F}_+^n (respectivement \mathcal{F}_-^n) est l'ensemble des fonctions positives (respectivement négatives) de $(\{0, 1\}^n \rightarrow \{0, 1\})$. \mathcal{F}_+^n est stable pour toute fonction f k -aire sur \mathcal{F}^n telle que $f(1, 1, \dots, 1) = 1$. Par exemple, \mathcal{F}_+^n est stable pour les opérations $\wedge, \vee, \Rightarrow$, et \Leftrightarrow , mais pas pour l'opération \oplus . Ainsi, la conjonction de fonctions positives est positive, donc différent de 0, et donc satisfiable. Ceci permet de résoudre le problème de la satisfiabilité ou de la validité à partir des types des différentes formules utilisées pour décrire une formule f , sans réaliser totalement la réécriture canonique de toute la formule f .

positive ; la fonction 0 est négative ; pour toute fonction f et toute variable x , f est positive (respectivement négative) si et seulement si la fonction $f[x \leftarrow 1]$ est positive (respectivement négative). On dénote alors explicitement le type d'une fonction par le type de son arbre associé.

L'arbre typé de Shannon d'un arbre de Shannon t s'obtient par évaluation de $\text{Type}(t)$, où la fonction Type est définie par :

$$\begin{aligned} \text{Type}(\Delta(x, L, H)) &= \text{TypeUp}(x, \text{Type}(L), \text{Type}(H)) \\ \text{Type}(0) &= -1 \\ \text{Type}(1) &= +1 \end{aligned}$$

$$\begin{aligned} \text{TypeUp}(x, +L, +H) &= +\Delta(x, +L, +H) \\ \text{TypeUp}(x, -L, +H) &= +\Delta(x, -L, +H) \\ \text{TypeUp}(x, +L, -H) &= -\Delta(x, -L, +H) \\ \text{TypeUp}(x, -L, -H) &= -\Delta(x, +L, +H) \end{aligned}$$

On montre que ce système est canonique parce qu'il n'autorise l'occurrence du type négatif que sur les branches gauches. La Figure 7 montre l'arbre typé de Shannon de la formule $((x_1 \wedge (x_3 \oplus x_4)) \vee (x_2 \wedge (x_3 \Leftrightarrow x_4)))$.

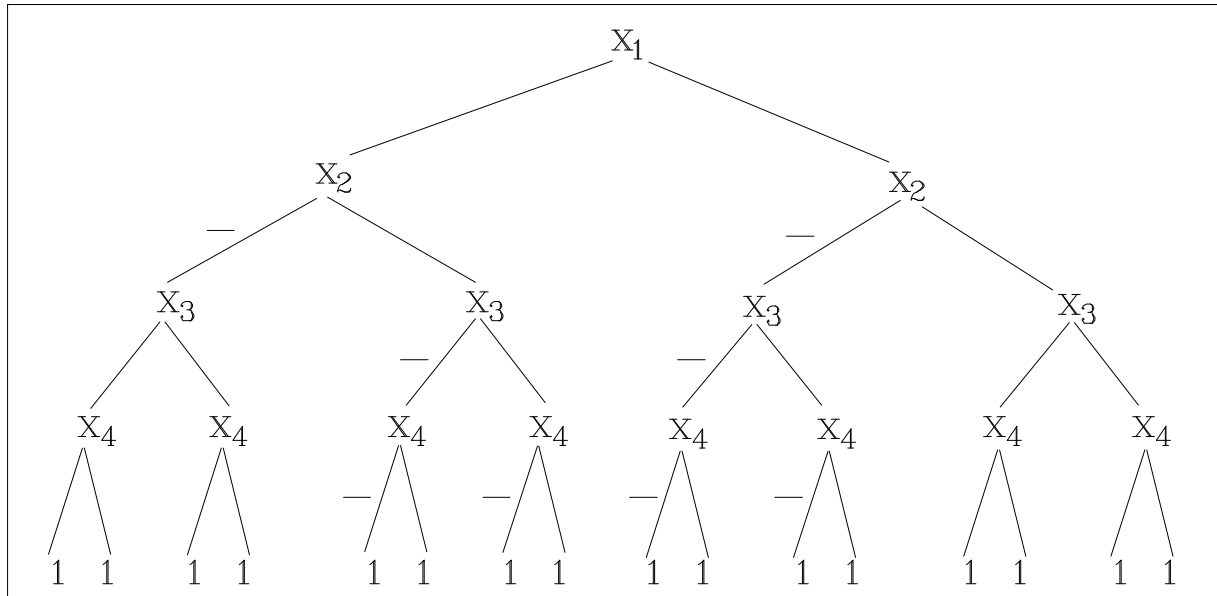


Figure 7. Arbre de décision typé de $((x_1 \wedge (x_3 \oplus x_4)) \vee (x_2 \wedge (x_3 \Leftrightarrow x_4)))$.

La propriété fondamentale des arbres typés de Shannon, en plus de leur canonicité, est la négation en $O(1)$. Pour obtenir l'arbre typé de Shannon de $(\neg f)$, il suffit d'inverser le type de l'arbre typé de Shannon de f , c'est à dire que la négation sur les arbres typés de Shannon s'exprime par les règles :

$$\begin{aligned} (\neg t) &\rightarrow -t \\ -+ &\rightarrow - \\ -- &\rightarrow + \end{aligned}$$

Quant aux combinaisons booléennes sur les arbres typés, il suffit de remplacer les règles 2.5 à 2.7 par les suivantes :

$$\begin{aligned}
 +\Delta(x, L, H) \vee +\Delta(x, L', H') &\rightarrow +\Delta(x, L \vee L', H \vee H') \\
 -\Delta(x, L, H) \vee +\Delta(x, L', H') &\rightarrow +\Delta(x, -L \vee L', -H \vee H') \\
 +\Delta(x, L, H) \vee -\Delta(x, L', H') &\rightarrow +\Delta(x, L \vee -L', H \vee -H') \\
 -\Delta(x, L, H) \vee -\Delta(x, L', H') &\rightarrow +\Delta(x, -L \vee -L', -H \vee -H') \\
 (-1) \vee t &\rightarrow t \\
 (+1) \vee t &\rightarrow +1
 \end{aligned}$$

Tout comme l'élimination des noeuds redondants et le partage des sous-graphes de la forme de Shannon conduit à la forme graphique canonique des BDDs, le même processus de réduction peut être appliqué sur la forme typée de Shannon, conduisant aux graphes de décision typés, ou TDGs. La Figure 8 montre l'arbre typé réduit de la formule $((x_1 \wedge (x_3 \oplus x_4)) \vee (x_2 \wedge (x_3 \Leftrightarrow x_4)))$, et la Figure 9 montre son TDG associé.

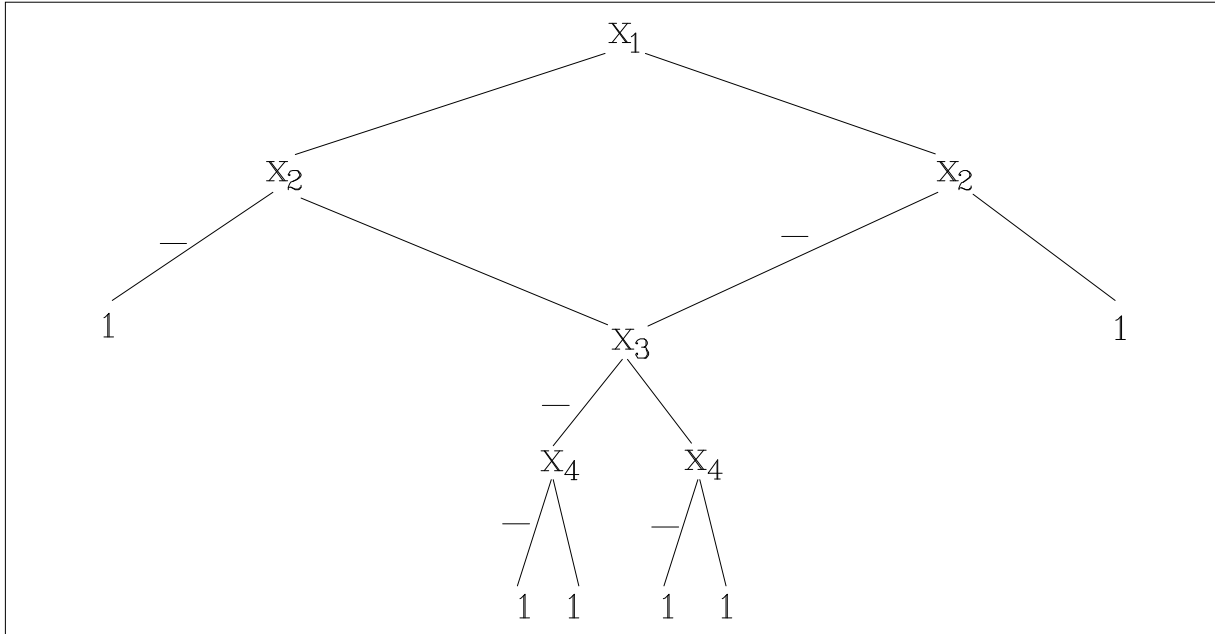


Figure 8. Arbre réduit typé de $((x_1 \wedge (x_3 \oplus x_4)) \vee (x_2 \wedge (x_3 \Leftrightarrow x_4)))$.

Comme les TDGs permettent de partager une fonction et sa négation, ils constituent une représentation plus dense que les BDDs. Par exemple, la formule $(x_1 \oplus x_2 \oplus \dots \oplus x_n)$ est représentée par un BDD de taille $2n - 1$ et un TDG de taille n , pour n'importe quel ordre. Ceci est aussi montré par la Figure 10 qui donne la croissance des BDDs et TDGs associés aux sorties d'un additionneur n -bits. La croissance des BDDs est en $9n$ [27] alors que celle des TDGs est en $5n$. Cependant le rapport entre la taille du BDD et celle du TDG d'une fonction ne peut être supérieur à 2, puisque la forme typée de Shannon ne permet d'économiser que la construction des fonctions négatives.

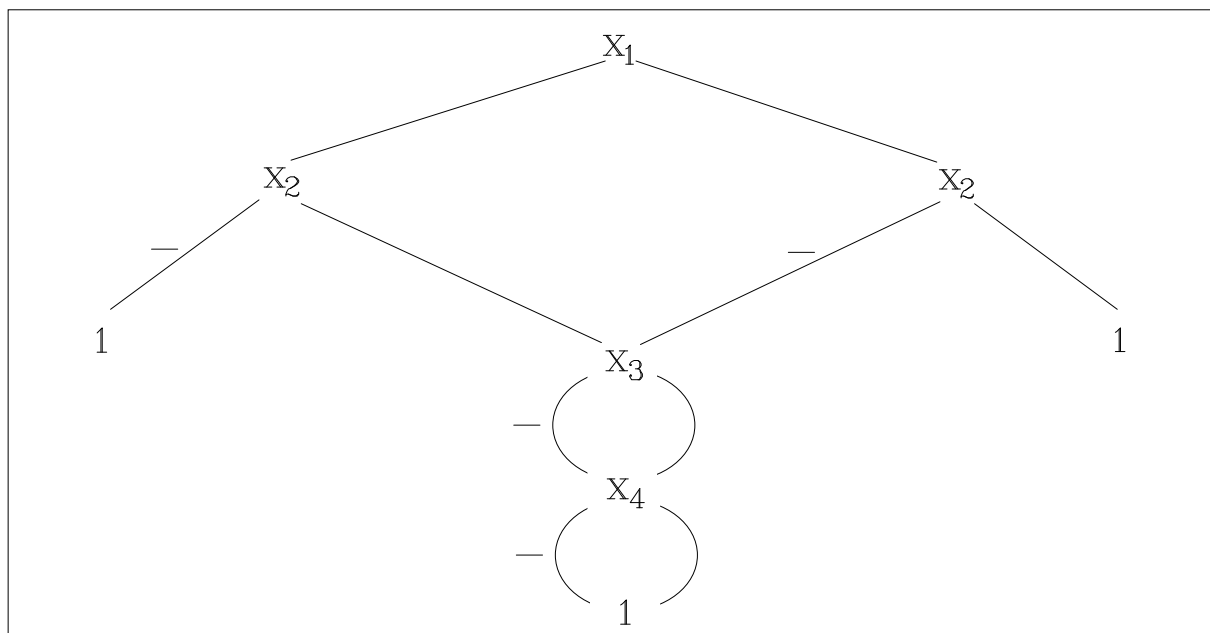
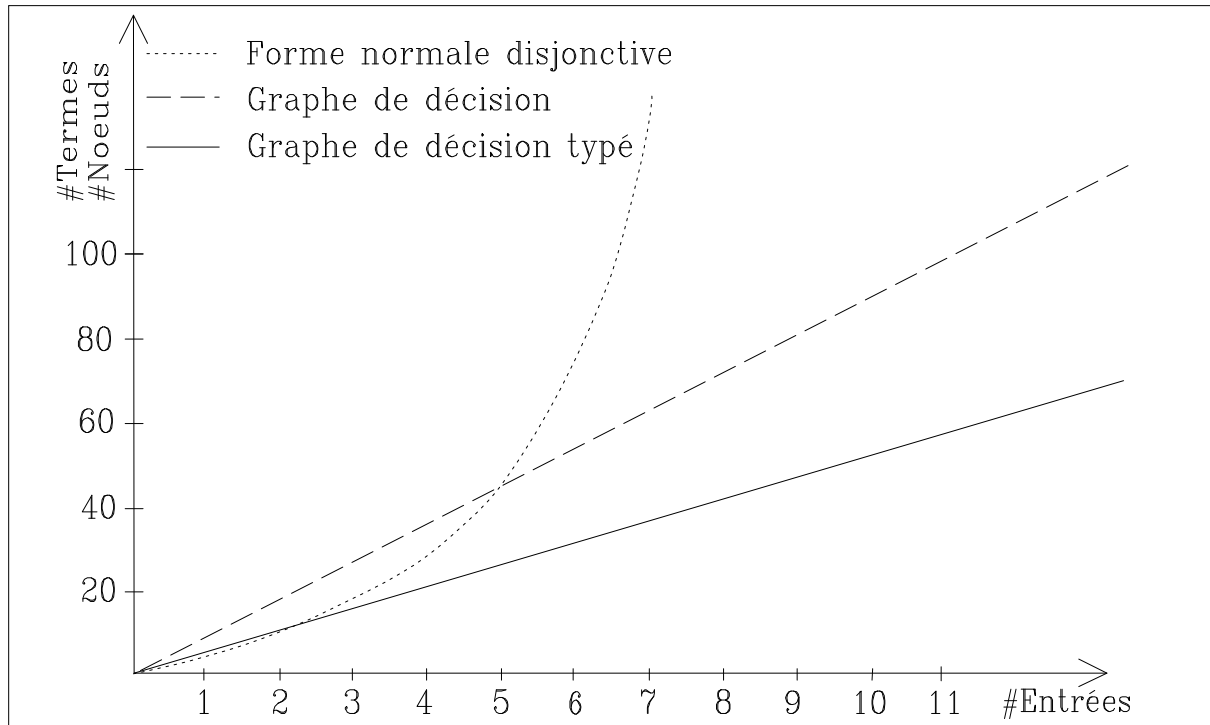
Figure 9. TDG de $((x_1 \wedge (x_3 \oplus x_4)) \vee (x_2 \wedge (x_3 \Leftrightarrow x_4)))$.

Figure 10. Croissances des sommes de produits, BDD et TDG sur les additionneurs.

La compaction de la représentation des formules propositionnelles peut être poursuivie en enrichissant le type des structures $\Delta(-, -, -)$, ce qui permet de jouer sur le codage et la répartition des négations. Nous présentons dans l'Annexe B deux autres formes canoniques graphiques typées, les *graphes 4-typés*, et les *graphes symétrisés*. Ces deux

formes sont plus compactes que les TDGs, d'environ un rapport 2. Si ces deux formes sont intéressantes pour illustrer la variété du typage des formes graphiques canoniques, le gain qu'elles apportent ne semble pas justifier leur utilisation pratique [101].

2.4 Manipulations de graphes de décision

A partir de maintenant, nous ne ferons la distinction entre BDDs et graphes typés (TDGs, graphes 4-typés, graphes symétrisés) que lorsque ce sera explicitement spécifié. Par commodité, nous confondrons souvent une fonction et son graphe.

2.4.1 Graphe de décision d'une formule

La canonisation d'une formule est évidemment un problème NP-complet, il en est donc de même du calcul du graphe d'une formule. Le calcul du graphe d'une formule f se fait à partir de l'arbre syntaxique de f , en remontant des feuilles vers la racine. La Figure 11

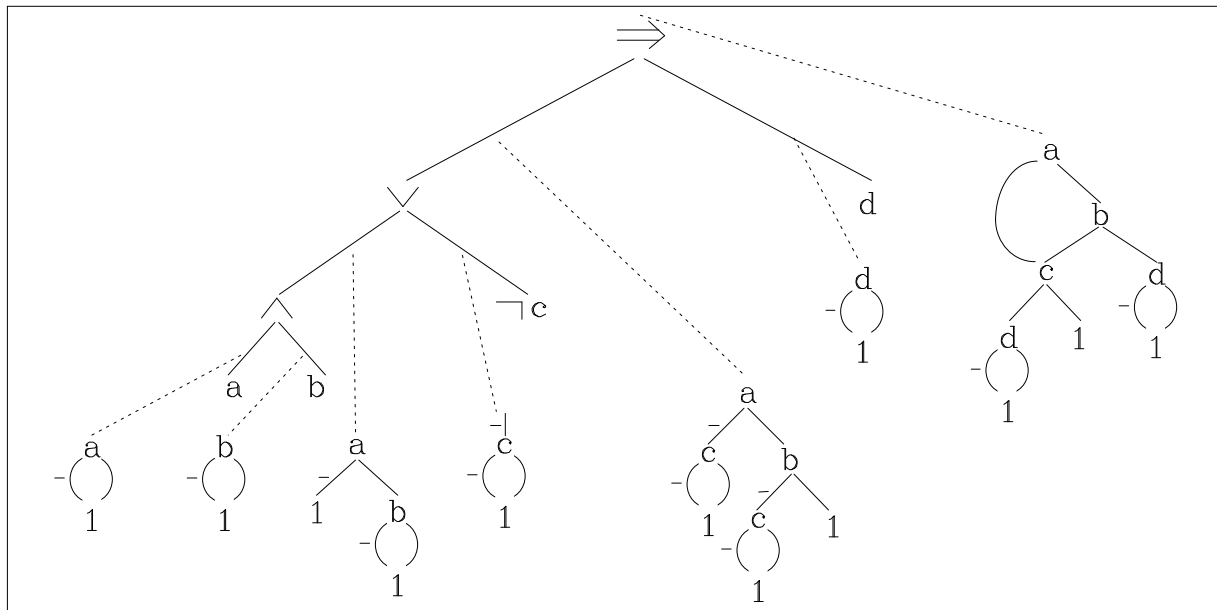


Figure 11. Calcul du TDG de la formule $((a \wedge b) \vee \neg c) \Rightarrow d$.

montre la construction du TDG de la formule $((a \wedge b) \vee \neg c) \Rightarrow d$. Chacune des feuilles est substituée par son graphe élémentaire (constantes 0 et 1, ou graphe $\Delta(x_k, 0, 1)$ pour une variable x_k). Puis les graphes sont combinés en remontant vers la racine, par application des opérateurs booléens intervenant dans l'arbre syntaxique de f .

2.4.2 Combinaisons de graphes de décision

Un exposé détaillé de l'implémentation des algorithmes effectuant les combinaisons booléennes classiques ($\neg, \vee, \wedge, \Rightarrow, \Leftrightarrow, \oplus$) sur les graphes de décision binaires (respectivement les graphes de décision typés) peut être trouvé dans [27] (respectivement dans [94]). Nous ne rappelons ici que les principes et les complexités de ces opérations.

On peut distinguer deux techniques pour évaluer les combinaisons booléennes usuelles $\neg, \vee, \wedge, \Rightarrow, \Leftrightarrow, \oplus$. La première exprime ces combinaisons en fonction d'un opérateur triadique [27], que nous noterons **ifn**. Intuitivement, **ifn** réalise un *ifnot-then-else*, c'est à dire que $\mathbf{ifn}(f, g, h) = ((\neg f \wedge g) \vee (f \wedge h))$. A partir de cet opérateur, on peut définir les règles de combinaison suivantes sur les BDDs :

$$\begin{aligned}
\mathbf{ifn}(0, t, _) &\rightarrow t \\
\mathbf{ifn}(1, _, t) &\rightarrow t \\
(k = \min_{\leq}(t, t', t'')) &\Rightarrow \\
\mathbf{ifn}(t, t', t'') &\rightarrow \Delta(x_k, \mathbf{ifn}(t_{[x_k \leftarrow 0]}, t'_{[x_k \leftarrow 0]}, t''_{[x_k \leftarrow 0]}), \mathbf{ifn}(t_{[x_k \leftarrow 1]}, t'_{[x_k \leftarrow 1]}, t''_{[x_k \leftarrow 1]}))
\end{aligned} \tag{2.12}$$

$$\begin{aligned}
\Delta(x_k, L, _)_{[x_k \leftarrow 0]} &\rightarrow L \\
\Delta(x_k, _, H)_{[x_k \leftarrow 1]} &\rightarrow H \\
(x_k < t) &\Rightarrow \\
t_{[x_k \leftarrow _]} &\rightarrow t \\
(k > j) &\Rightarrow \\
\Delta(x_j, L, H)_{[x_k \leftarrow v]} &\rightarrow \Delta(x_j, L_{[x_k \leftarrow v]}, H_{[x_k \leftarrow v]})
\end{aligned}$$

$$\begin{aligned}
\neg t &\rightarrow \mathbf{ifn}(t, 0, 1) \\
t \vee t' &\rightarrow \mathbf{ifn}(t, t', 1) \\
t \wedge t' &\rightarrow \mathbf{ifn}(t, 0, t') \\
t \Rightarrow t' &\rightarrow \mathbf{ifn}(t, 1, t') \\
t \Leftrightarrow t' &\rightarrow \mathbf{ifn}(t, \neg t', t') \\
t \oplus t' &\rightarrow \mathbf{ifn}(t, t', \neg t') \\
(\exists x_k t) &\rightarrow \mathbf{ifn}(t_{[x_k \leftarrow 1]}, t_{[x_k \leftarrow 0]}, 1) \\
(\forall x_k t) &\rightarrow \mathbf{ifn}(t_{[x_k \leftarrow 1]}, 0, t_{[x_k \leftarrow 0]}) \\
t[x_k \leftarrow t'] &\rightarrow \mathbf{ifn}(t', t_{[x_k \leftarrow 0]}, t_{[x_k \leftarrow 1]})
\end{aligned} \tag{2.13}$$

A ce système on ajoute les identités suivantes qui permettent d'accélérer le calcul :

$$\begin{aligned}
\mathbf{ifn}(_, t, t) &\rightarrow t \\
\mathbf{ifn}(t, 0, 1) &\rightarrow t \\
\mathbf{ifn}(t, t, 1) &\rightarrow t \\
\mathbf{ifn}(t, 0, t) &\rightarrow t \\
\mathbf{ifn}(t, t, 0) &\rightarrow 0 \\
\mathbf{ifn}(t, 1, t) &\rightarrow 1
\end{aligned}$$

L'évaluation du terme $\mathbf{ifn}(f, g, h)$ requiert au plus $|f| \times |g| \times |h|$ applications de la règle 2.12 avant d'atteindre les règles terminales. La complexité de l'évaluation de $\mathbf{ifn}(f, g, h)$ est donc majorée par $O(|f| \times |g| \times |h|)$. La négation est faite en $O(|f|)$, donc les opérations usuelles voient leur complexité bornée par une fonction quadratique. On montrera dans le Théorème 2.9 que le pire cas, c'est à dire quadratique, est effectivement atteint.

La seconde technique [94] consiste à descendre l'opération à effectuer jusqu'aux feuilles des DAGs grâce aux trois premières règles ci-dessous, puis on applique les règles terminales de combinaison associées à chaque opérateur. Là encore, on ajoute à ce système des règles de simplification directe, par exemple l'idempotence des opérateurs $\vee, \wedge, \Rightarrow, \Leftrightarrow$.

$$\text{apply}(o, \Delta(x_k, L, H), \Delta(x_k, L', H')) \rightarrow \Delta(x_k, \text{apply}(o, L, L'), \text{apply}(o, H, H'))$$

$$(x_k \prec t) \Rightarrow$$

$$\text{apply}(o, \Delta(x_k, L, H), t) \rightarrow \Delta(x_k, \text{apply}(o, L, t), \text{apply}(o, H, t))$$

$$(x_k \prec t) \Rightarrow$$

$$\text{apply}(o, t, \Delta(x_k, L, H)) \rightarrow \Delta(x_k, \text{apply}(o, t, L), \text{apply}(o, t, H))$$

$$\text{apply}(\vee, 0, t) \rightarrow t$$

$$\text{apply}(\vee, t, 0) \rightarrow t$$

$$\text{apply}(\vee, 1, t) \rightarrow 1$$

$$\text{apply}(\vee, t, 1) \rightarrow 1$$

$$\text{apply}(\wedge, 0, t) \rightarrow 0$$

$$\text{apply}(\wedge, t, 0) \rightarrow 0$$

$$\text{apply}(\wedge, 1, t) \rightarrow 1$$

$$\text{apply}(\wedge, t, 1) \rightarrow 1$$

$$\text{apply}(\Rightarrow, 0, t) \rightarrow 1$$

$$\text{apply}(\Rightarrow, t, 0) \rightarrow \neg t$$

$$\text{apply}(\Rightarrow, 1, t) \rightarrow t$$

$$\text{apply}(\Rightarrow, t, 1) \rightarrow 1$$

$$\text{apply}(\Leftrightarrow, 0, t) \rightarrow \neg t$$

$$\text{apply}(\Leftrightarrow, t, 0) \rightarrow \neg t$$

$$\text{apply}(\Leftrightarrow, 1, t) \rightarrow t$$

$$\text{apply}(\Leftrightarrow, t, 1) \rightarrow t$$

$$\neg 0 \rightarrow 1$$

$$\neg 1 \rightarrow 0$$

$$\neg \Delta(x, L, H) \rightarrow \Delta(x, \neg L, \neg H)$$

$$t \vee t' \rightarrow \text{apply}(\vee, t, t')$$

$$t \wedge t' \rightarrow \text{apply}(\wedge, t, t')$$

$$t \Rightarrow t' \rightarrow \text{apply}(\Rightarrow, t, t')$$

$$t \Leftrightarrow t' \rightarrow \text{apply}(\Leftrightarrow, t, t')$$

$$t \oplus t' \rightarrow \neg \text{apply}(\Leftrightarrow, t, t')$$

Théorème 2.9 *Soient f et g deux graphes de décision binaires. La complexité du calcul du graphe de décision binaire de $(\neg f)$ est en $O(|f|)$, celui du graphe de décision binaire de $(f \star g)$ est en $O(|f| \times |g|)$, où \star est un opérateur booléen diadique quelconque (par exemple $\vee, \wedge, \Rightarrow, \Leftrightarrow$, ou \oplus).*

Preuve. La négation d'un BDD f nécessite une recopie de celui-ci avec inversion des feuilles, d'où une négation en $O(|f|)$. Nous avons vu que l'obtention du BDD de $(f \star g)$ requiert au plus $|f| \times |g|$ applications de la règle 2.12. Montrons maintenant que la taille du BDD de $(f \star g)$ est en $O(|f| \times |g|)$. Cette preuve est due à Bryant [28]. Par nature, la taille du BDD de $(f \star g)$ est bornée par $|f| \times |g|$. Considérons maintenant $2n + 2m$ variables $x_1 < \dots < x_{2n+2m}$. Soient les formules

$$f = \left(\bigvee_{k=1}^n (x_k \wedge x_{n+m+k}) \right) \quad \text{et} \quad (2.14)$$

$$g = \left(\bigvee_{k=1}^m (x_{n+k} \wedge x_{2n+m+k}) \right). \quad (2.15)$$

Leurs BDDs ont respectivement les tailles 2^{n+1} et 2^{m+1} . On a :

$$f \vee g = \left(\bigvee_{k=1}^{n+m} (x_k \wedge x_{n+m+k}) \right),$$

dont le BDD est de taille 2^{n+m+1} , c'est à dire $(|f| \times |g|)/2$. La complexité du calcul de $(f \star g)$ est donc en $O(|f| \times |g|)$ pour l'application des opérateurs booléens $\vee, \wedge, \Rightarrow, \Leftrightarrow, \oplus$. \square

Théorème 2.10 *Soient f et g deux graphes de décision typés. La complexité du calcul du graphe de décision typé de $(\neg f)$ est en $O(1)$, celui du graphe de décision typé de $(f \star g)$ est en $O(|f| \times |g|)$, où \star est un opérateur booléen quelconque.*

Les opérateurs booléens usuels ont donc des coûts tout à fait raisonnables sur les graphes, puisque polynomiaux par rapport aux tailles des graphes. Pour implémenter l'évaluation d'une fonction n -aire g , il suffit de généraliser l'opération \star au cas n -aire, et donner les réécritures de g dans les cas terminaux (potentiellement 2^n cas terminaux sont nécessaires pour décrire complètement g). La complexité de l'évaluation de $g(f_1, \dots, f_n)$ sera finalement bornée par $O(\prod_{k=1}^n |f_k|)$. En résumé, l'évaluation d'une opération n -aire implémentée "en dur" se fait au pire polynomialement de degré n .

2.4.3 Composition de graphes de décision

Nous abordons ici l'évaluation de $g(f_1, \dots, f_n)$ dans le cas où g est donnée sous la forme d'un graphe. Nous étudions d'abord la substitution d'une variable d'un graphe par un autre graphe, puis le problème de la composition en général.

D'après la règle 2.13, la substitution d'une variable dans un graphe g par un graphe f s'effectue avec une complexité bornée par $O(|g|^2 \times |f|)$. D'un autre côté, il est clair

que la substitution d'une variable d'un graphe g par un graphe f peut être au moins en $O(|g| \times |f|)$. Il suffit de considérer les variables $x_0 < x_1 < \dots < x_{2n+2m}$, et prendre

$$\begin{aligned} g &= (x_0 \vee \left(\bigvee_{k=1}^n (x_k \wedge x_{n+m+k}) \right)) \quad \text{et} \\ f &= \left(\bigvee_{k=1}^m (x_{n+k} \wedge x_{2n+m+k}) \right). \end{aligned}$$

La substitution de la variable x_0 du graphe g par le graphe f conduit à calculer $(\bigvee_{k=1}^{n+m} (x_k \wedge x_{n+m+k}))$, dont la preuve du Théorème 2.9 a montré la taille en $O(|f| \times |g|)$. Cependant, le problème de savoir si le pire cas est en $O(|g| \times |f|)$ ou en $O(|g|^2 \times |f|)$ reste ouvert.

Le problème de la composition est une généralisation du problème de la substitution. Ce problème est le suivant : étant donnés les graphes des formules f_1, \dots, f_n et g , calculer le graphe de la formule $g[x_1 \leftarrow f_1, \dots, x_n \leftarrow f_n]$. La composition est une opération intrinsèquement plus complexe que celles vues précédemment, comme le montre le théorème suivant.

Théorème 2.11 *La composition sur une forme canonique est NP-difficile.*

Preuve. Nous donnons la preuve sur le cas particulier des graphes de décision, son extension à une forme canonique quelconque ne demandant que des développements techniques. Soit $C = (\bigwedge_{k=1}^n c_k)$ une 3-forme normale conjonctive composée de n clauses. Chaque clause est une disjonction de trois littéraux, donc son graphe est construit en un temps constant. Ainsi le calcul des graphes f_1, \dots, f_n des n clauses est en $O(n)$. Soit g le graphe de la fonction $\lambda[x_1 \dots x_n]. (\bigwedge_{k=1}^n x_k)$. Ce graphe est un peigne construit en $O(n)$. Tester si un graphe est différent de zéro est en $O(1)$. Comme C est satisfiable si et seulement si le graphe de $g(f_1, \dots, f_n)$ est différent de 0, calculer $g(f_1, \dots, f_n)$ est NP-difficile. \square

On obtient un théorème plus précis sur les graphes avec l'hypothèse d'un ordre des variables fixé.

Théorème 2.12 *Etant donné un ordre fixé des variables, la composition est un problème de complexité au moins exponentielle vis-à-vis de la taille des graphes et/ou du nombre de variables utilisées. Plus précisément, il existe des graphes f_1, \dots, f_n, g ayant $2n$ variables, dont la taille cumulée est en $O(n)$, tels que le graphe de $g(f_1, \dots, f_n)$ est en $O(2^n)$.*

Preuve. Nous allons exhiber des graphes g et f_1, \dots, f_n , construits sur $2n$ variables, tels que $(|g| + \sum_{k=1}^n |f_k|) = O(n)$, et tels que $|g(f_1, \dots, f_n)| = O(2^n)$. On pose l'ordre des variables suivant : $y_1 < y_2 < \dots < y_{2n}$. Soit

$$g = \lambda[x_1 \dots x_n]. \left(\bigwedge_{k=1}^n x_k \right)$$

Le graphe de cette fonction est une branche de profondeur n , donc de taille en $O(n)$. Soient les n fonctions f_1, \dots, f_n définies par

$$f_k = \lambda[y_1 \dots y_{2n}]. (y_k \Leftrightarrow y_{2n-k+1}),$$

pour $1 \leq k \leq n$. Les BDDs des fonctions f_k sont de taille 3 (de taille 2 pour les TDGs), donc $(\sum_{k=1}^n |f_k|) = O(n)$. On déduit finalement que $(|g| + \sum_{k=1}^n |f_k|)$ est en $O(n)$. La composition $g(f_1, \dots, f_n)$ est la fonction $\lambda \vec{y}. (\bigwedge_{k=1}^n f_k(\vec{y}))$, qui s'écrit donc

$$\lambda[y_1 \dots y_{2n}]. \left(\bigwedge_{k=1}^n (y_k \Leftrightarrow y_{2n-k+1}) \right)$$

Le graphe de la formule $((y_1 \Leftrightarrow y_{2n}) \wedge (y_2 \Leftrightarrow y_{2n-1}) \wedge \dots \wedge (y_n \Leftrightarrow y_{n+1}))$ est en $O(2^n)$, comme le montre le Lemme 2.1. \square

Lemme 2.1 *La formule $((y_1 \Leftrightarrow y_{2n}) \wedge (y_2 \Leftrightarrow y_{2n-1}) \wedge \dots \wedge (y_n \Leftrightarrow y_{n+1}))$ possède un graphe de décision en $O(2^n)$ pour l'ordre $y_1 < y_2 < \dots < y_{2n}$.*

Preuve. Nous montrons que le BDD de $((y_1 \Leftrightarrow y_{2n}) \wedge (y_2 \Leftrightarrow y_{2n-1}) \wedge \dots \wedge (y_n \Leftrightarrow y_{n+1}))$ est de taille $3(2^n - 1)$ par récurrence. Une preuve analogue montre que le TDG de cette formule est de taille $3(2^n - 1) - 1$.

Considérons le graphe t_n montré Figure 12. Il utilise $2n$ variables $y_1 < y_2 < \dots < y_{2n}$, et est totalement expansé sur les $n + 1$ premières variables. Par définition, les 2^{n-1} sous graphes G_j^n ($1 \leq j \leq 2^{n-1}$) sont tous différents. Soit s_n le nombre de noeuds nécessaires pour représenter ces 2^{n-1} sous graphes G_j^n . La taille de t_n est donc $(s_n + \sum_{k=1}^{n+1} 2^{k-1})$, soit $|t_n| = 2^{n+1} + s_n - 1$.

On effectue alors le produit de t_n avec $(a \Leftrightarrow b)$, tel que $y_n < a < b < y_{n+1}$. Le graphe de $(a \Leftrightarrow b)$ doit être introduit au milieu du graphe t_n . On obtient la nouvelle structure de graphe donnée par la Figure 13.

Cette structure est analogue à celle de t_n , à savoir que l'expansion du graphe est faite sur les $n + 2$ premières variables (i.e. y_1, \dots, y_n, a, b), et qu'à partir d'un graphe G_j^n , on construit les deux graphes $G_{2j-1}^{n+1} = (\neg y_{n+1} \wedge G_j^n)$ et $G_{2j}^{n+1} = (y_{n+1} \wedge G_j^n)$, c'est à dire $G_{2j-1}^{n+1} = \Delta(y_{n+1}, G_j^n, 0)$ et $G_{2j}^{n+1} = \Delta(y_{n+1}, 0, G_j^n)$. Les 2^n nouveaux graphes G_j^{n+1} ont un nombre de noeuds $s_{n+1} = s_n + 2^n$. Donc le graphe t_{n+1} est de taille $2^{n+2} + s_{n+1} - 1$.

Pour $n = 1$, on a $s_1 = 0$, donc $s_n = (\sum_{k=1}^{n-1} 2^k)$, c'est à dire $s_n = 2^n - 2$. Donc la taille de t_n , soit la formule $((y_1 \Leftrightarrow y_{2n}) \wedge (y_2 \Leftrightarrow y_{2n-1}) \wedge \dots \wedge (y_n \Leftrightarrow y_{n+1}))$, est égale à $2^{n+1} + 2^n - 2 - 1$, c'est à dire $3(2^n - 1)$. \square

La composition est donc au moins exponentielle si l'ordre des variables est fixé. Cependant, le lecteur pourra objecter que la formule $((y_1 \Leftrightarrow y_{2n}) \wedge (y_2 \Leftrightarrow y_{2n-1}) \wedge \dots \wedge (y_n \Leftrightarrow y_{n+1}))$ utilisée dans la preuve du Théorème 2.12 possède un graphe en $O(n)$ avec l'ordre $y_1 < y_{2n} < y_2 < y_{2n-1} < \dots < y_n < y_{n+1}$. On peut alors légitimement se poser la question suivante : si l'on est capable de modifier dynamiquement l'ordre des variables de façon à minimiser la taille des graphes manipulés, y-a-t-il un moyen d'obtenir un algorithme de composition non exponentiel en mémoire? Quelques remarques suffisent pour répondre par la négative. Premièrement, le problème d'obtenir un "bon ordre", c'est à dire un ordre qui minimise la taille du graphe d'une fonction, est lui-même NP-difficile. Deuxièmement, il existe des fonctions (voir Théorème 2.20, Section 2.5) dont le graphe est exponentiel vis-à-vis du nombre de variables utilisées, quel que soit l'ordre choisi sur celles-ci. Pour cette dernière raison, il est certain que la composition reste exponentielle, même si un oracle était capable de fournir dynamiquement un ordre optimal.

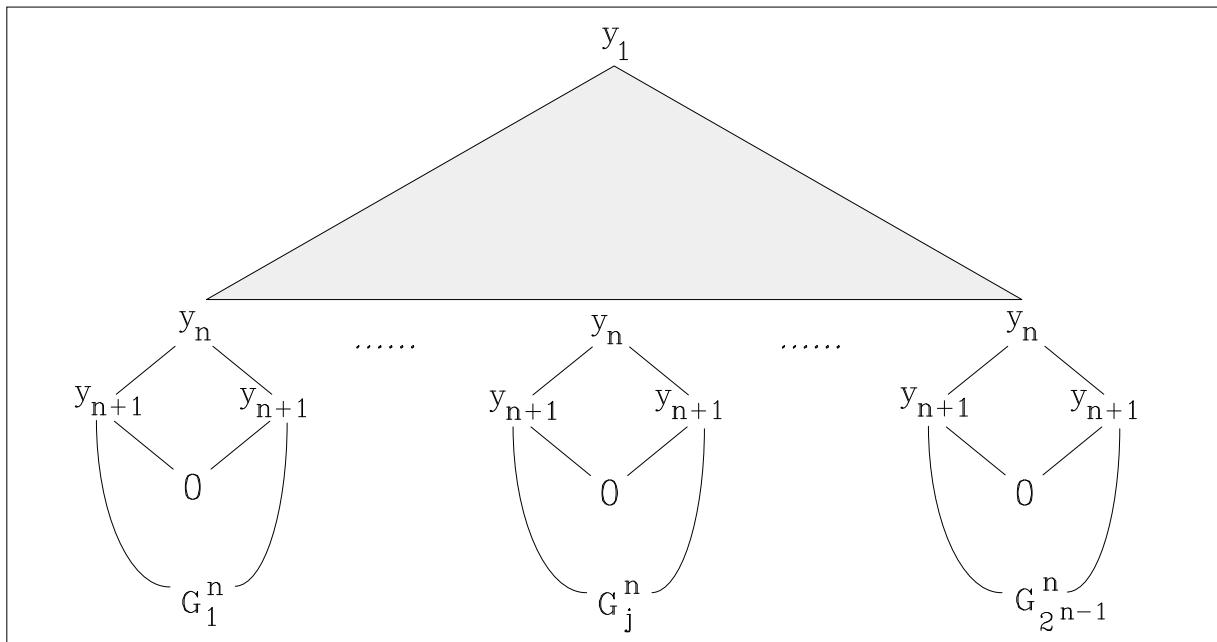


Figure 12. Graphe de t_n .

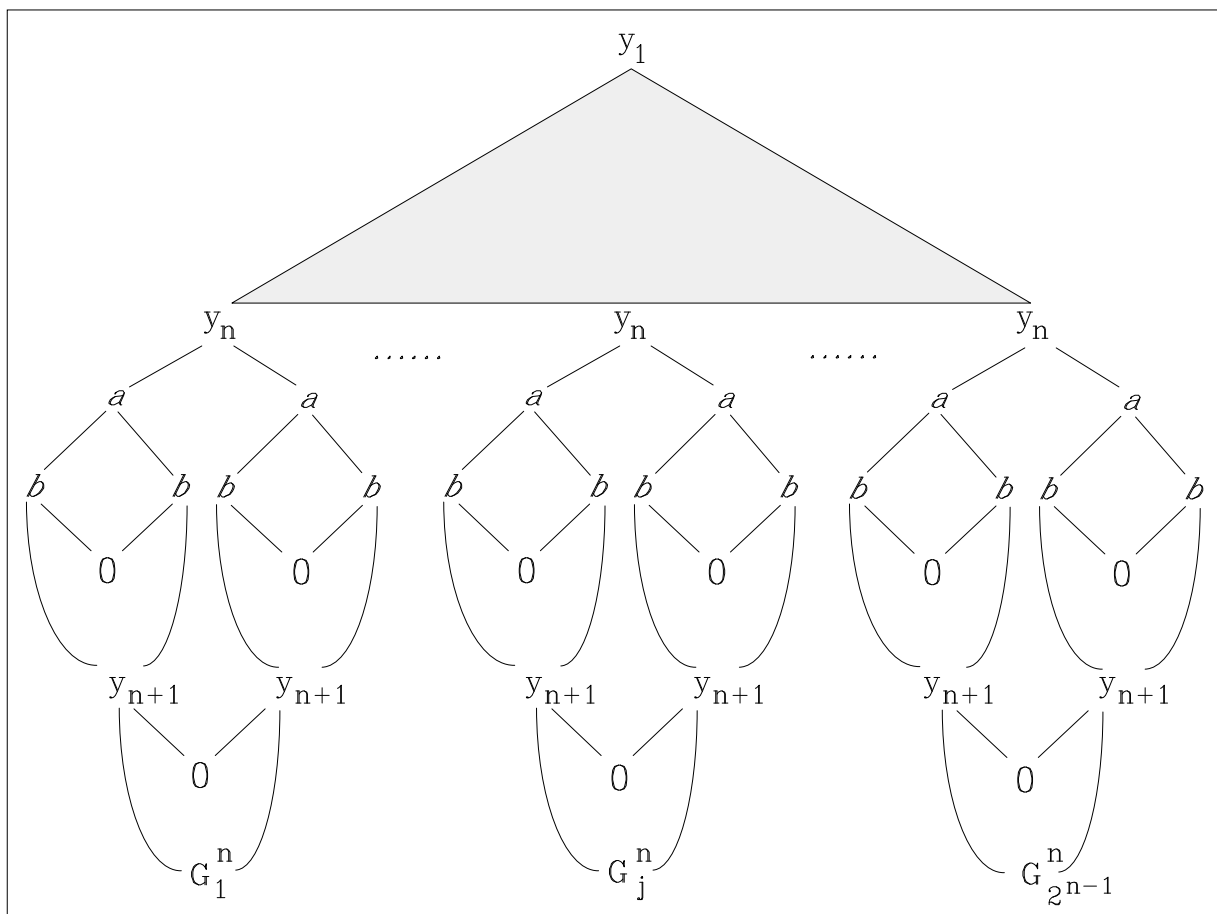


Figure 13. Graphe de $(t_n \wedge (a \Leftrightarrow b))$.

Un cas particulier de composition est celui de la *composition close*. Soit $g(y_1, \dots, y_n)$ un graphe dont le support est inclus dans $\{y_1, \dots, y_n\}$, et soit n graphes f_1, \dots, f_n . Alors $g(f_1, \dots, f_n)$ est une composition close, c'est à dire que toutes les variables de g sont substituées. On peut remarquer que tous les résultats donnés précédemment tiennent pour la composition close, puisque toutes les démonstrations utilisent des compositions closes. Il existe un algorithme de complexité borné par $O(|g| \times \prod_{k=1}^n |f_k|)$ calculant l'évaluation close $g(f_1, \dots, f_n)$. Par contre, et ceci rejoint la remarque sur la substitution d'une variable (est-ce un problème quadratique ou cubique ?), il semble que la composition non close soit en $O(|g|^2 \times \prod_{k=1}^n |f_k|)$.

2.4.4 Forme sans quantificateur d'un graphe quantifié

Nous étudions ici le problème du calcul du graphe de la forme sans quantificateur d'une formule quantifiée $(Q_1 x_1 \dots Q_n x_n f)$, où f est un graphe. Les théorèmes qui suivent montrent que dans le pire cas, cette élimination conduit à un graphe de taille exponentielle par rapport à n .

Théorème 2.13 *Etant donné un graphe f , l'obtention de la forme sans quantificateur de $(Qx f)$, où x est un symbole propositionnel, se fait en $O(|f|^2)$.*

Preuve. La taille de la forme sans quantificateur de $(Qy f)$ est celle du graphe de $(f[y \leftarrow 0] \vee f[y \leftarrow 1])$. Les tailles de $f[y \leftarrow 0]$ et $f[y \leftarrow 1]$ sont au maximum égales à $|f|$, donc la taille de la forme sans quantificateur de $(Qy f)$ est forcément bornée par $|f|^2$. Considérons la fonction

$$\begin{aligned} f &= \lambda[y \ x_1 \dots x_{4n}].((\neg y \wedge g_1(\vec{x})) \vee (y \wedge g_2(\vec{x}))), \quad \text{avec} \\ g_1 &= \lambda\vec{x}. \left(\bigvee_{k=1}^n (x_k \wedge x_{2n+k}) \right), \quad \text{et} \\ g_2 &= \lambda\vec{x}. \left(\bigvee_{k=1}^n (x_{n+k} \wedge x_{3n+k}) \right). \end{aligned}$$

La taille de f est $2^{n+2} + 1$. La forme sans quantificateur de $(\exists y f)$ est $\lambda\vec{x}.(g_1(\vec{x}) \wedge g_2(\vec{x}))$, de taille 2^{2n+1} . Donc la taille de la forme sans quantificateur de $(\exists y f)$ est $(|f| - 1)^2/8$. \square

Théorème 2.14 *Le problème de l'élimination de n variables est non polynomial vis-à-vis de la taille du graphe ou du nombre de variables utilisées.*

En particulier, il existe une fonction f de n variables, telle que :

$$\begin{aligned} |\exists \vec{x} f| &> 2^{|f|^{1/2}}, \quad \text{et} \\ |\exists \vec{x} f| &> 2^{n/4}, \end{aligned}$$

où \vec{x} dénote une partie de ces n variables.

Preuve. Il suffit de considérer un graphe complètement expansé sur les n premières variables, et dont chacun des 2^n sous graphes est celui d'une formule g_k ($1 \leq k \leq 2^n$), utilisant $2n$ variables x_j^k (avec $1 \leq j \leq n$ ou $2n+1 \leq j \leq 3n$) chacune, de la forme

$$g_k = \left(\bigvee_{j=1}^n (x_j^k \wedge x_{2n+j}^k) \right)$$

analogue à celle utilisée dans le théorème précédent. On part donc d'une formule f de taille $(2^n - 1) + (2^n \times 2^{n+1})$, c'est à dire $2^{2n+1} + 2^n - 1$. Cette formule contient $n + (2^n \times 2n)$ variables, c'est à dire $n(2^{n+1} + 1)$ variables. La forme sans quantificateur f_1 de la formule $(\exists x_n f)$ est de taille $(2^{n-1} - 1) + (2^{n-1} \times 2^{2n+1})$. La forme sans quantificateur f_2 de la formule $(\exists x_{n-1} \exists x_n f)$ est de taille $(2^{n-2} - 1) + (2^{n-2} \times 2^{4n+1})$. Par une récurrence immédiate, on montre que la forme sans quantificateur f_k de la formule $(\exists x_{n-k+1} \dots \exists x_n f)$ est de taille $(2^{n-k} - 1) + (2^{n-k} \times 2^{2^k+1})$. On en déduit la taille de la forme sans quantificateur f_n de $(\exists x_1 \dots \exists x_n f)$:

$$|\exists x_1 \dots \exists x_n f| = 2^{n2^{n+1}}.$$

On peut alors montrer que

$$|\exists \vec{x} f| = O(|f|^{\frac{n2^n}{2^{n+1}}}).$$

En particulier, $|\exists \vec{x} f| > 2^{|f|^{1/2}}$. Donc l'élimination est non polynomiale vis-à-vis de la taille du graphe initial. La formule initiale contient $v = n(2^{n+1} + 1)$ variables. On a donc $|\exists \vec{x} f| > 2^{v/4}$. L'élimination est donc aussi exponentielle vis-à-vis du nombre de variables utilisées. \square

La forme sans quantificateur d'une formule close, c'est à dire une formule dont toutes les variables sont quantifiées, est obligatoirement 0 ou 1. Comme le résultat est de taille constante, on peut se demander si le test (ou le calcul) portant sur la forme sans quantificateur d'un graphe (ou d'un graphe clos) est un problème plus facile. Il n'en est évidemment rien dans le cas général, comme le montre le Théorème 2.17.

Théorème 2.15 *Soit f un graphe quelconque. Tester si $(\forall x_1 \dots \forall x_n f) = 1$, ou si $(\exists x_1 \dots \exists x_n f) = 0$, se fait en $O(1)$.*

Preuve. $(\forall x_1 \dots \forall x_n f) = 1$ si et seulement si $f = 1$, et $(\exists x_1 \dots \exists x_n f) = 0$ si et seulement si $f = 0$. Comme les graphes sont canoniques, le résultat est immédiat. \square

Théorème 2.16 *Soit $f(x_1, \dots, x_n)$ un graphe de n variables x_1, \dots, x_n . Calculer les formes sans quantificateur des formules closes $(\forall x_1 \dots \forall x_n f)$ et $(\exists x_1 \dots \exists x_n f)$ se fait en $O(1)$.*

Preuve. La forme sans quantificateur d'une formule close est la constante 0 ou 1. On a $(\forall x_1 \dots \forall x_n f) = 1$ si et seulement si $f = 1$, et $(\exists x_1 \dots \exists x_n f) = 0$ si et seulement si $f = 0$. Comme le graphe de f est une forme canonique, le résultat est immédiat. \square

Théorème 2.17 *Soit $f(x_1, \dots, x_n)$ un graphe de n variables x_1, \dots, x_n . Calculer la forme sans quantificateur du terme clos $(Q_1 x_1 \dots Q_n x_n f)$ est NP-difficile.*

Preuve. Soit $C = (\bigwedge_{k=1}^n c_k)$ une 3-forme normale conjonctive composée de n clauses c_k . Soit n variables y_1, \dots, y_n inférieures à toutes les variables (que l'on dénotera par le vecteur \vec{x}) apparaissant dans C . Soit g le graphe de la formule

$$\begin{aligned}
& ((y_1 \wedge c_1) \vee \\
& \quad (\neg y_1 \wedge ((y_2 \wedge c_2) \vee \\
& \quad \quad (\neg y_2 \wedge ((y_3 \wedge c_3) \vee \\
& \quad \quad \quad \ddots \\
& \quad \quad \quad (\neg y_{n-2} \wedge ((y_{n-1} \wedge c_{n-1}) \vee \\
& \quad \quad \quad \quad (\neg y_{n-1} \wedge (\neg y_n \vee c_n)))) \dots))))
\end{aligned}$$

Le graphe de g est donné ci-dessous. Comme chaque clause c_k est une disjonction de trois littéraux, la taille de g est bornée par $4n$, et il peut être facilement construit en $O(n)$. Considérons alors la formule close $(\exists \vec{x} \forall \vec{y} g)$ qui est construite en $O(n)$.

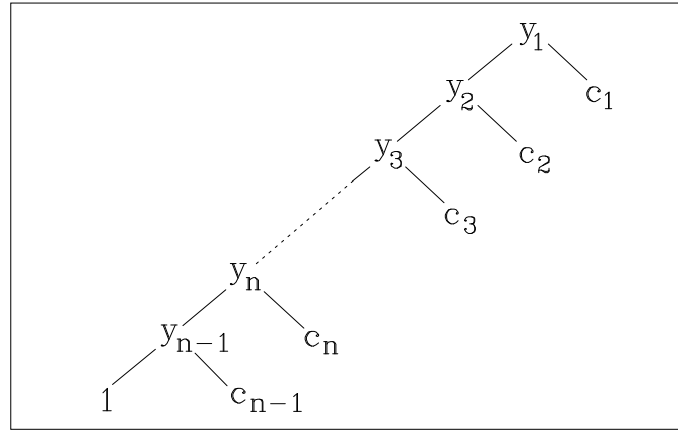


Figure 14. Graphe de g .

La Figure 14 montre que la forme sans quantificateur de $(\forall \vec{y} g)$ est $(\bigwedge_{k=1}^n c_k)$. Donc la forme sans quantificateur de $(\exists \vec{x} \forall \vec{y} g)$ est équivalente à $(\exists \vec{x} C)$. Cette formule est égale à 1 si et seulement si C est satisfiable. Comme $(\exists \vec{x} \forall \vec{y} g)$ est une formule close, on peut calculer sa forme sans quantificateur de la façon suivante : si C est satisfiable, alors $(\exists \vec{x} \forall \vec{y} g) = 1$, sinon $(\exists \vec{x} \forall \vec{y} g) = 0$. Donc calculer la forme sans quantificateur de la formule close $(\exists \vec{x} \forall \vec{y} g)$ est NP-difficile. \square

2.4.5 Résolution d'équations

Il suffit de reprendre les algorithmes définis Section 1.4, et d'étudier leur complexité sur les graphes.

Pour une équation à une inconnue $(\forall y_1 \dots \forall y_n \exists x f)$ où f est un graphe, la fonction de Skolem de x (si elle existe) est calculée en $O(|f|^2)$. Dans le cas particulier où l'ordre sur les variables est tel que $\{y_1, \dots, y_n\} < x$, la fonction de Skolem est calculée en $O(|f|)$ par l'évaluation du terme $\text{solve}(f)$, où solve est définie par :

$$\begin{aligned}
\text{solve}(\Delta(y_k, L, H)) &= \Delta(y_k, \text{solve}(L), \text{solve}(H)) \\
\text{solve}(\Delta(x, 0, 1)) &= 1 \\
\text{solve}(\Delta(x, 1, 0)) &= 0 \\
\text{solve}(1) &= \Delta(p, 0, 1)
\end{aligned}$$

Pour une équation quelconque, la résolution est évidemment non polynomiale en général, puisque le calcul d'une forme sans quantificateur d'un graphe est lui-même un problème non polynomial. Dans le cas particulier d'une équation du type $(\forall y_1 \dots \forall y_n \exists x_1 \dots \exists x_m f)$ où f est un graphe, ce qui correspond au cas de l'unification booléenne [119, 24], avec $\{y_1, \dots, y_n\} < \{x_1, \dots, x_m\}$, les m fonctions de Skolem sont calculées en $O(m \times |f|)$, et chacune d'entre elles a une taille en $O(|f|)$.

2.5 Le problème de l'ordre

Un ordre fixé des variables assure la canonicité des BDDs et TDGs. Mais la taille du graphe d'une formule dépend de cet ordre. Nous étudions ici l'influence de l'ordre des variables sur la taille des graphes.

Considérons une permutation π des n premiers entiers. Pour toute formule f , on note f_π la formule obtenue à partir de f par renommage des variables x_k en $x_{\pi(k)}$. Soit V_π la fonction $\lambda[x_1 \dots x_n]. [x_{\pi(1)} \dots x_{\pi(n)}]$. A une fonction $\lambda \vec{x}. f$ de n variables, on peut associer $n!$ réécritures de la forme $(\lambda \vec{x}. f_\pi) \circ V_{\pi^{-1}}$. On peut représenter f par le graphe de f_π modulo la permutation π , c'est à dire modulo l'ordre sur les variables. Les graphes des $n!$ formules f_π possibles n'ont évidemment pas tous la même taille, et on a vu qu'en choisissant un mauvais ordre sur les variables, on peut faire croître dramatiquement la taille du graphe d'une fonction. Il importe donc de bien choisir l'ordre d'expansion des variables. Il existe des classes de fonctions insensibles à l'ordre, comme les fonctions symétriques :

Théorème 2.18 *Le graphe d'une fonction symétrique de n variables est de taille majorée par $n^2/2$, quel que soit l'ordre des variables.*

Preuve. Soit f une fonction symétrique de n variables. Comme f est symétrique, pour toute permutation π des n premiers entiers, on a $f(x_1, \dots, x_n) = f(x_{\pi(1)}, \dots, x_{\pi(n)})$. Ceci garantit que la taille du graphe de f est insensible à l'ordre. Montrons maintenant qu'il est de taille bornée par $n^2/2$.

Comme f est symétrique, la valeur de $f(x_1, \dots, x_n)$ ne dépend que du nombre de valeurs 1 prises par les variables x_1, \dots, x_n . Ainsi une fonction symétrique de n variables est entièrement décrite en donnant sa valeur v_k lorsque exactement k de ses variables prennent la valeur 1, ceci pour $0 \leq k \leq n$:

$$f(\underbrace{1, \dots, 1}_k, \underbrace{0, \dots, 0}_{n-k}) = v_k$$

Le graphe non complètement réduit de cette fonction est la structure G_1^1 , où $G_j^k = \Delta(x_k, G_j^{k+1}, G_{j+1}^{k+1})$ pour $1 \leq j \leq k \leq n$, et $G_j^{n+1} = v_{j-1}$ pour $1 \leq j \leq n+1$. Ce graphe est donné Figure 15. La taille du graphe réduit correspondant, obtenu en remplaçant les feuilles v_k par les constantes 0 ou 1, est évidemment bornée par $\frac{n(n-1)}{2}$. \square

D'un autre côté, la croissance des graphes décrivant les sorties de l'additionneur n -bits est linéaire en fonction de n dans le meilleur des cas, mais elle est exponentielle dans le pire des cas [27]. Plus directement, $((y_1 \Leftrightarrow y_{2n}) \wedge (y_2 \Leftrightarrow y_{2n-1}) \wedge \dots \wedge (y_n \Leftrightarrow y_{n+1}))$ a un graphe en $O(n)$ pour l'ordre $y_1 < y_{2n} < y_2 < y_{2n-1} < \dots < y_{n-1} < y_n$, mais le Lemme 2.1

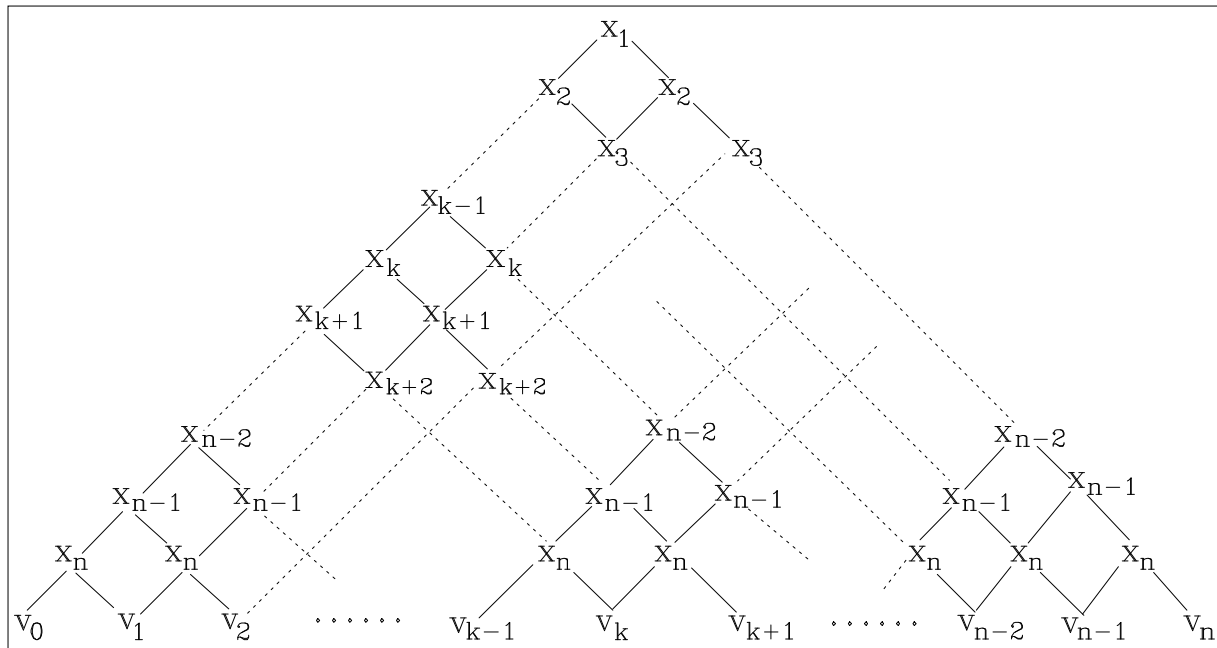


Figure 15. Graphe des fonctions symétriques.

a montré que ce graphe est en $O(2^n)$ pour l'ordre $y_1 < y_2 < y_3 < \dots < y_{2n}$. On peut alors se demander s'il existe un algorithme simple pour trouver un ordre qui minimise la taille du graphe, puis s'il est toujours possible de trouver un graphe de taille polynomiale pour chaque fonction. Les deux théorèmes suivants, issus de [28, 12, 13], répondent par la négative⁶.

Théorème 2.19 *Etant donnée une fonction booléenne f , le problème de trouver un ordre des variables qui minimise la taille du graphe de f est NP-difficile.*

Théorème 2.20 *Il existe des fonctions booléennes pour lesquelles il n'existe pas d'ordre permettant d'obtenir un graphe de taille polynomiale.*

Il existe un algorithme de complexité $O(n^2 3^n)$ permettant de déterminer un ordre des n variables qui minimise le graphe de décision d'une fonction [63]. Cependant, son coût prohibitif lui fait préférer l'utilisation des nombreuses heuristiques [64, 96, 12, 13] proposées dans le cadre de la vérification de circuits combinatoires, toutes basées sur l'analyse topologique du circuit. Cependant, aucune de ces heuristiques n'est supérieure aux autres : là où une heuristique donne un bon ordre pour une fonction pour laquelle échouent les autres heuristiques, il existe réciproquement une fonction sur laquelle la première heuristique va donner un mauvais ordre alors que les autres sauront calculer un ordre raisonnable. Afin de combiner les avantages de chacune de ces heuristiques, il a été proposé [108] de choisir un meilleur ordre parmi ceux calculés. Cette technique de choix est basée sur la construction partielle des graphes, en bornant par une constante le nombre de noeuds autorisé pour chaque graphe, et en construisant le graphe en largeur d'abord afin de conserver le

⁶Le Théorème 2.20 vient de [28], où il est montré qu'il existe une fonction, réalisée à partir d'un multiplicateur n bits, dont le BDD est de taille au moins en $O(1.09^n)$, quel que soit l'ordonnement des variables.

plus d'informations pertinentes possibles. Ceci permet de discriminer rapidement un bon ordre parmi plusieurs ordres proposés, lorsque celui-ci existe. Quant aux fonctions pour lesquelles aucun "bon ordre" n'existe, elles demeurent hors de portée des manipulations complexes.

2.6 Comparaison des graphes avec d'autres représentations

Dans cette section, nous comparons les TDGs avec une forme normale de représentation très utilisée aujourd'hui, à savoir la forme normale disjonctive, encore appelée somme de produits, ainsi qu'avec la forme canonique de Reed-Muller.

2.6.1 Graphes et sommes de produits

Voici une syntaxe contenant les sommes de produits. On appellera un terme de type $\langle product \rangle$ un *produit*.

$$\begin{aligned}
 \langle SumOfProduct \rangle & ::= \langle SumOfProduct \rangle \vee \langle product \rangle \\
 & ::= 0 \mid 1 \mid \langle product \rangle \\
 \langle product \rangle & ::= \langle product \rangle \wedge \langle literal \rangle \\
 & ::= \langle literal \rangle \\
 \langle literal \rangle & ::= \langle var \rangle \mid \neg \langle var \rangle \\
 \langle var \rangle & ::= x_1 \mid \dots \mid x_n
 \end{aligned}$$

Toute formule s'écrit en une somme de produits par le système de réécriture suivant :

$$\begin{aligned}
 f \oplus g & \rightarrow (\neg f \wedge g) \vee (f \wedge \neg g) \\
 f \Leftrightarrow g & \rightarrow (f \wedge g) \vee (\neg f \wedge \neg g) \\
 f \Rightarrow g & \rightarrow \neg f \vee g \\
 \\
 \neg \neg f & \rightarrow f \\
 \neg(f \vee g) & \rightarrow \neg f \wedge \neg g \\
 \neg(f \wedge g) & \rightarrow \neg f \vee \neg g \\
 f \wedge (g \vee h) & \rightarrow (f \wedge g) \vee (f \wedge h) \\
 (f \vee g) \wedge h & \rightarrow (f \wedge h) \vee (g \wedge h)
 \end{aligned}$$

On dira qu'un produit c *contient* un produit c' si l'ensemble des littéraux composant c inclut l'ensemble des littéraux composant c' . Evidemment, c contient c' si et seulement si $(c \Rightarrow c')$ est une tautologie. On appellera *somme de produits réduite* une forme normale obtenue en appliquant les règles de simplification élémentaires utilisant les constantes 0 et 1, et des *règles de recouvrement*, à savoir : si un produit c contient une variable et sa négation, le remplacer par 0 ; si un produit c contient un produit c' , supprimer c . Dans la suite, on ne considèrera que des sommes de produits réduites. Un système de simplification

peut être décrit ainsi, où x est un symbole propositionnel, c un produit, f une somme de produits, \vee et \wedge sont commutatifs :

$$\begin{aligned}
 (\neg x \vee x) &\rightarrow 1 \\
 ((\neg x \wedge c) \vee (x \wedge c)) &\rightarrow c \\
 (c \vee (c \wedge c')) &\rightarrow c \\
 (\neg x \wedge x) &\rightarrow 0 \\
 (0 \wedge c) &\rightarrow 0 \\
 (0 \vee f) &\rightarrow f \\
 (1 \vee f) &\rightarrow 1
 \end{aligned}$$

Une somme de produits s'écrit $(\bigvee_{k=1}^n (\bigwedge_{j=1}^{m_k} q_k^j))$, où q_k^j est un littéral. La taille d'une somme de produits f sera mesurée par le nombre d'occurrences de littéraux dans f (cette mesure est analogue au nombre de caractères de la formule f), c'est à dire $(\sum_{k=1}^n m_k)$. On a les résultats suivants :

Théorème 2.21 *Sur les sommes de produits, la disjonction est linéaire, et la conjonction est quadratique. Obtenir une somme de produits représentant la négation d'une somme de produits f s'effectue en*

$$O\left(\sqrt{|f|}^{\sqrt{|f|+1}}\right).$$

Obtenir une somme de produits représentant l'implication, l'équivalence ou la somme exclusive de deux sommes de produits f et g s'effectue en

$$O\left(\sqrt{|f| + |g|}^{\sqrt{|f|+|g|+1}}\right).$$

Preuve. La disjonction de deux sommes de produits est évidemment linéaire sans simplification, polynomiale avec simplification. La conjonction des deux expressions

$$\left(\bigvee_{k=1}^n \left(\bigwedge_{j=1}^{m_k} q_k^j \right) \right), \quad \text{et} \\
 \left(\bigvee_{k=1}^{n'} \left(\bigwedge_{j=1}^{m'_k} q'_k{}^j \right) \right)$$

conduit à distribuer un produit sur deux sommes, ce qui donne potentiellement une somme de $n \times n'$ produits, chacun ayant $m_i + m'_j$ (avec $1 \leq i \leq n$ et $1 \leq j \leq n'$) littéraux. La conjonction est donc quadratique. Par exemple, considérons deux sommes de produits $f = (\bigvee_{k=1}^n x_k)$ et $f' = (\bigvee_{k=1}^n x'_k)$, de taille n chacune. Le produit $(f \wedge f')$ s'écrit comme la somme de produits

$$\left(\bigvee_{\substack{1 \leq k \leq n \\ 1 \leq j \leq n}} (x_k \wedge x'_j) \right)$$

qui est composée de la somme de n^2 produits de taille 2, soit une taille de $2n^2$.

Calculer la négation de la somme de produit $(\bigvee_{k=1}^n (\bigwedge_{j=1}^{m_k} q_k^j))$ consiste à écrire en somme de produits l'expression $(\bigwedge_{k=1}^n (\bigvee_{j=1}^{m_k} \neg q_k^j))$. Il faut distribuer la conjonction sur tous les produits, pour obtenir potentiellement $(\prod_{k=1}^n m_k)$ produits de taille n , soit une complexité non polynomiale. Par exemple, considérons la somme de produits $(\bigvee_{k=1}^n (\bigwedge_{j=1}^n x_j^k))$, où chaque variable x_j^k n'apparaît qu'une seule fois. Elle contient exactement n^2 variables, et sa taille est n^2 . Sa négation est $(\bigwedge_{k=1}^n (\bigvee_{j=1}^n \neg x_j^k))$, dont la somme de produits s'écrit

$$\left(\bigvee_{(j_1, \dots, j_n) \in [1, n]^n} \left(\bigwedge_{k=1}^n \neg x_{j_k}^k \right) \right)$$

ce qui représente une somme de n^n produits de taille n , soit une somme de produits de taille n^{n+1} . Remarquons de plus que cette formule se représente efficacement par un BDD (ou TDG) de taille n^2 avec l'ordre $x_j^k < x_{j+1}^k$ (avec $1 \leq j \leq n$ et $1 \leq k \leq n$) et $x_n^k < x_1^{k+1}$ (avec $1 \leq k < n$). Ce n'est donc pas la "complexité" de cette formule qui attribue une négation de coût exponentiel à sa somme de produits.

L'implication, l'équivalence et la somme exclusive sont de même complexité, puisque ces opérations peuvent s'exprimer en utilisant la négation (exponentielle) et la disjonction (polynomiale). Par exemple, $(f \Rightarrow 0) = \neg f$, $(f \Leftrightarrow 0) = \neg f$, et $(f \oplus 1) = \neg f$. \square

Théorème 2.22 *Si les fonctions booléennes sont représentées par des sommes de produits, l'élimination existentielle d'un nombre quelconque de variables est linéaire, et l'élimination universelle d'une variable (respectivement d'un nombre quelconque de variables) est quadratique (respectivement exponentiel).*

Etudions les rapports entre la taille d'une somme de produits et celle de son graphe. Si la formule $(\bigwedge_{k=1}^n (x_{2k-1} \Leftrightarrow x_{2k}))$ peut se représenter linéairement par un BDD de taille $3n$, la somme de produits correspondante est exponentielle, car constituée de 2^n produits de taille $2n$. On pourra rétorquer que pour un mauvais ordre, le graphe de cette expression est de taille exponentielle. Considérons alors la formule $(x_1 \oplus x_2 \oplus \dots \oplus x_n)$. Elle dénote une fonction symétrique, égale à 1 si et seulement si un nombre impair de ses variables prennent la valeur 1. Quel que soit l'ordre considéré, la taille de son graphe de décision est en $O(n)$. Par contre, sa somme de produits est constituée de la somme de 2^{n-1} produits de taille n . Il existe donc des fonctions représentées par un graphe polynomial quel que soit l'ordre de ses variables, qui n'admettent qu'une représentation exponentielle en somme de produits. La réciproque, à savoir existe-il des fonctions polynomialement représentées par une somme de produits et qui n'admettent pas de représentation polynomiale sous forme de graphe, est une question qui reste ouverte⁷. De très récents développements semblent montrer que cette réciproque est vraie.

2.6.2 Graphes et sommes exclusives de produits

Nous comparons ici les graphes et la forme *somme exclusive de produits*, ou *forme de Reed-Muller*, dont l'idée originale a été introduite dans [117], et qui a été appliquée à la

⁷On ne peut que constater que le passage d'une somme de produits à un graphe est nécessairement NP-difficile, car savoir si une somme de produits est une tautologie est un problème NP-complet, et les graphes sont canoniques.

démonstration par [76, 77, 78]. Voici une syntaxe contenant la forme de Reed-Muller.

$$\begin{aligned}
 \langle \text{ReedMuller} \rangle &::= \langle \text{constant} \rangle \langle \text{ExclusiveSum} \rangle \\
 \langle \text{constant} \rangle &::= 0 \mid 1 \\
 \langle \text{ExclusiveSum} \rangle &::= \\
 &::= \oplus \langle \text{ExclusiveSum2} \rangle \\
 \langle \text{ExclusiveSum2} \rangle &::= \langle \text{product} \rangle \\
 &::= \langle \text{product} \rangle \oplus \langle \text{ExclusiveSum2} \rangle \\
 \langle \text{product} \rangle &::= \langle \text{var} \rangle \\
 &::= \langle \text{var} \rangle \wedge \langle \text{product} \rangle \\
 \langle \text{var} \rangle &::= x_1 \mid \dots \mid x_n
 \end{aligned}$$

Toute formule peut s'écrire sous la forme d'une somme exclusive de produits par le système de réécriture suivant :

$$\begin{aligned}
 \neg f &\rightarrow 1 \oplus f \\
 f \Leftrightarrow g &\rightarrow 1 \oplus f \oplus g \\
 f \Rightarrow g &\rightarrow 1 \oplus f \oplus (f \wedge g) \\
 f \vee g &\rightarrow f \oplus g \oplus (f \wedge g) \\
 f \wedge (g \oplus h) &\rightarrow (f \wedge g) \oplus (f \wedge h)
 \end{aligned}$$

Il ne reste plus qu'à appliquer les simplifications suivantes, pour obtenir une forme canonique modulo la commutativité et l'associativité de \oplus et \wedge . Cette forme canonique est faite d'une somme exclusive de produits de variables, plus éventuellement une somme exclusive avec la constante 1.

$$\begin{aligned}
 0 \wedge f &\rightarrow 0 \\
 1 \wedge f &\rightarrow f \\
 f \wedge f &\rightarrow f \\
 0 \oplus f &\rightarrow f \tag{2.16}
 \end{aligned}$$

$$f \oplus f \rightarrow 0 \tag{2.17}$$

L'ordonnancement des variables (par exemple l'ordre lexical), qui induit sur les produits une canonisation et un ordre total, permet d'obtenir une forme syntaxique unique. On supposera dans la suite que les produits sont ainsi ordonnés, et que la constante 1 est le produit minimal. Par exemple la forme canonique de Reed-Muller associée à la formule $((x \Rightarrow y) \wedge (y \Rightarrow z) \wedge (z \Rightarrow x))$ est la formule

$$1 \oplus x \oplus y \oplus z \oplus (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$$

Une somme exclusive de produits s'écrit $(1 \oplus (\bigoplus_{k=1}^n (\bigwedge_{j=1}^{m_k} x_k^j)))$ ou $(\bigoplus_{k=1}^n (\bigwedge_{j=1}^{m_k} x_k^j))$, où x_k^j est une variable. La taille d'une somme exclusive de produit f sera mesurée par le nombre d'occurrences de variables dans f (cette mesure est analogue aux nombres de caractères de f), c'est à dire $(\sum_{k=1}^n m_k)$. On a les résultats suivants :

Théorème 2.23 *La négation sur la forme de Reed-Muller est en $O(1)$. L'obtention de la forme de Reed-Muller de l'équivalence ou la somme exclusive de deux formes de Reed-Muller f et g est en*

$$O((|f| + |g|) \log(|f| + |g|)).$$

La conjonction, la disjonction ou l'implication de deux formes de Reed-Muller f et g est en

$$O(|f| \times |g| \times \log(|f| \times |g|)).$$

Preuve. Calculer la négation d'une somme exclusive de produits consiste à appliquer les règles suivantes, ce qui s'effectue en $O(1)$:

$$\begin{aligned} \neg f &\rightarrow 1 \oplus f \\ 1 \oplus 1 &\rightarrow 0 \\ 0 \oplus f &\rightarrow f \end{aligned}$$

Calculer la somme exclusive de deux formes de Reed-Muller f et g consiste à appliquer les règles de simplification 2.16 et 2.17 sur la formule $f \oplus g$, et à ordonner lexicalement les produits des deux formules, ce qui est fait en $O((|f| + |g|) \log(|f| + |g|))$. L'équivalence possède la même complexité, car on a l'identité $(f \Leftrightarrow g) = \neg(f \oplus g)$.

Le produit de deux formes de Reed-Muller $(\bigoplus_{k=1}^n (\bigoplus_{j=1}^{m_k} x_k^j))$ et $(\bigoplus_{k=1}^{n'} (\bigoplus_{j=1}^{m'_k} x_k'^j))$ consiste à distribuer un produit sur deux sommes exclusives, ce qui donne potentiellement une somme exclusive de $n \times n'$ produits, chacun étant constitué de $m_i + m'_j$ occurrences de variables (avec $1 \leq i \leq n$ et $1 \leq j \leq n'$). La conjonction donne donc un nombre de produits quadratique. Par exemple, considérons deux formes de Reed-Muller $f = (\bigoplus_{k=1}^n x_k)$ et $f' = (\bigoplus_{k=1}^{n'} x_k')$, de taille n chacune. Le produit $(f \wedge f')$ s'écrit comme la somme exclusive de produits

$$\left(\bigoplus_{\substack{1 \leq k \leq n \\ 1 \leq j \leq n}} (x_k \wedge x_j') \right)$$

qui est composée de la somme exclusive de n^2 produits de taille 2, soit une taille de $2n^2$. L'ordonnement lexical de la formule ainsi obtenue donne finalement la complexité. Comme on a les égalités $(f \vee g) = \neg(\neg f \wedge \neg g)$ et $(f \Rightarrow g) = (\neg f \vee g)$, la disjonction et l'implication possèdent la même complexité. \square

Théorème 2.24 *L'élimination existentielle ou universelle d'une variable (respectivement d'un nombre quelconque de variables) d'une forme de Reed-Muller f est en $O(|f|^2 \log |f|)$ (respectivement exponentiel).*

Tout comme dans la section précédente avec les sommes de produits, il existe des fonctions (par exemple les sorties d'un additionneur) qui sont représentées exponentiellement avec la somme exclusive de produits, alors qu'elles admettent une représentation polynomiale avec les graphes. La réciproque, à savoir s'il existe des fonctions polynomialement représentées par une somme exclusive de produits et qui n'admettent pas de représentation polynomiale sous forme de graphe, reste ouverte⁸.

⁸Elle est à notre avis fautive, mais nous n'avons pu apporter une preuve complète.

2.6.3 Verdict : avantage aux graphes de décision!

Résumons tout d'abord la situation pour les formes canoniques graphiques : la négation s'effectue linéairement avec les BDDs, en $O(1)$ pour les trois formes typés ; les combinaisons élémentaires sont quadratiques sur toutes les formes graphiques ; la composition, l'élimination des quantificateurs, et la résolution d'équations sont non polynomiales. Il y un rapport borné par 2 (respectivement par 4) entre un BDD et le TDG (respectivement le graphe 4-typé ou symétrisé) équivalent. La Table 2 présente divers critères de comparaison pour la somme de produits, la forme de Reed-Muller, et le TDG.

	SumOfProduct	Reed-Muller	TDG
Calcul de f	$O(f \times 2^{ f })$	$O(f \times 2^{ f })$	$O(2^{ f })$
Satisfiabilité	NP-complet	$O(1)$	$O(1)$
Tautologie	NP-complet	$O(1)$	$O(1)$
$\neg f$	$O\left(\sqrt{ f }^{\sqrt{ f +1}}\right)$	$O(1)$	$O(1)$
$f \Rightarrow g$	$O\left(\sqrt{ f + g }^{\sqrt{ f + g +1}}\right)$	$O(f \times g \times \log(f \times g))$	$O(f \times g)$
$f \Leftrightarrow g$	$O\left(\sqrt{ f + g }^{\sqrt{ f + g +1}}\right)$	$O((f + g) \log(f + g))$	$O(f \times g)$
$f \oplus g$	$O\left(\sqrt{ f + g }^{\sqrt{ f + g +1}}\right)$	$O((f + g) \log(f + g))$	$O(f \times g)$
$f \wedge g$	$O(f \times g)$	$O(f \times g \times \log(f \times g))$	$O(f \times g)$
$f \vee g$	$O(f + g)$	$O(f \times g \times \log(f \times g))$	$O(f \times g)$
$\exists x f$	$O(f)$	$O(f ^2 \times \log f)$	$O(f ^2)$
$\exists \vec{x} f$	$O(f)$	$O(f \times 2^{ f })$	$O(2^{\sqrt{ f }})$
$\forall x f$	$O(f ^2)$	$O(f ^2 \times \log f)$	$O(f ^2)$
$\forall \vec{x} f$	$O(2^{ f })$	$O(f \times 2^{ f })$	$O(2^{\sqrt{ f }})$
taille max.	$n2^n$	$n2^{n-1}$	$O\left(\frac{2^n}{n}\right)$
répartition	$> \frac{2^n}{\log_2 n}$ p.p.	$> \frac{2^n}{\log_2 n}$ p.p.	$> \frac{2^n}{n}$ p.p.

Table 2. Comparaison des somme de produits, forme de Reed-Muller, et TDG.

La représentation des fonctions booléennes sous forme de graphes canoniques est supérieure aux sommes de produits sur la plupart des points. Les algorithmes de combinaisons élémentaires ($\neg, \vee, \wedge, \dots$) sur les graphes sont polynomiaux, et très simples à mettre en oeuvre, car très récursifs. Ce n'est pas le cas pour les sommes de produits, certaines opérations simples étant non polynomiales, et chaque opération logique devant être réalisée par un algorithme spécialisé. La représentation sous forme de graphes est canonique, ce qui simplifie considérablement les algorithmes de preuve.

Les divergences entre graphes et forme de Reed-Muller à propos de la complexité algorithmique semble moins apparentes. Mais des fonctions booléennes usuelles, telles que celles implémentées par un additionneur, admettent une représentation polynomiale avec les graphes mais pas avec la forme de Reed-Muller (ni avec les sommes de produits). En général, les graphes sont une représentation bien plus compacte que ces deux autres formes, plus souple et plus efficace. La meilleure preuve est le succès des BDDs et TDGs rencontré d'abord dans la preuve formelle de matériel depuis 1986, qui touche maintenant d'autres domaines (par exemple les systèmes de maintien de la cohérence).

Le problème essentiel des graphes est qu'ils requièrent un ordonnancement des variables. Un ordre fixe rend les graphes canoniques, mais la taille des graphes est étroitement dépendante de l'ordre des variables, et trouver un bon ordre est un problème NP-difficile. Il apparait que l'ordre doit être dynamique [48] si on veut pouvoir manipuler des fonctions complexes ou avec beaucoup de variables (*sparse functions*). Une voie de recherche, récemment abordée [8, 31], consiste à utiliser une forme graphique "localement" canonique, autorisant plusieurs ordres, ou des ordres partiels.

2.7 Conclusion

La forme canonique graphique, basée sur l'expansion de Shannon et le compactage d'arbres en graphes acycliques orientés, est actuellement la technique de représentation des formules propositionnelles la plus efficace, car très compacte et autorisant des manipulations formelles complexes avec des algorithmes simples à mettre en oeuvre. La souplesse des formes canoniques graphiques et leur compacité ont permis un progrès considérable dans la preuve en logique propositionnelle.

Les deux formes BDDs (Graphes de Décision Binaires) et TDGs (Graphes de Décision Typés) ont été présentées, et nous avons défini plusieurs autres formes canoniques graphiques originales. Nous avons surtout détaillé les comportements de ces formes de représentation (répartition des graphes par rapport à leurs tailles), ainsi que la complexité de leur manipulation sur toutes les opérations que nous avons définies dans le Chapitre 1 (combinaisons élémentaires, élimination des quantificateurs, composition, résolution d'équations), en introduisant de nouveaux résultats permettant de cerner leur efficacité et leur limite.

Partie II

Preuve de systèmes séquentiels

Chapitre 3

Problèmes sur les machines séquentielles

Nous nous intéressons ici aux manipulations formelles susceptibles de résoudre efficacement une large classe de problèmes portant sur les machines à états finis. Nous donnons d'abord le modèle de machine séquentielle sur lequel porteront ces manipulations, puis nous donnons trois grands aspects des problèmes posés sur les machines séquentielles. Nous réduisons alors ces trois problèmes à l'étude de quelques primitives de manipulation formelle, à savoir le calcul de l'image d'une fonction, le calcul de l'image réciproque d'une fonction, la minimisation combinatoire, l'élimination des variables d'états redondantes, et le réencodage.

3.1 Modèle d'une machine séquentielle

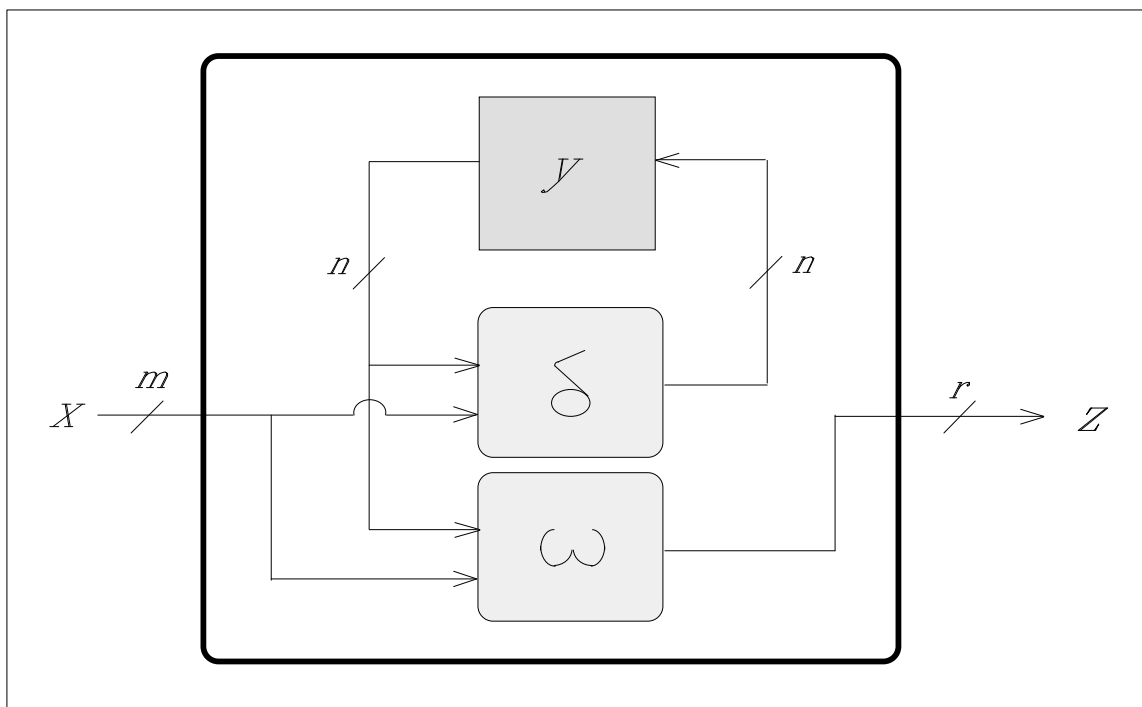


Figure 16. Modèle d'une machine séquentielle.

Nous considérons ici que la machine séquentielle qui est traitée est une machine incomplètement spécifiée [75, 83]. Cette machine \mathcal{M} est définie par un 6-uplet $(n, m, r, \omega, \delta, Init)$, où :

- n est le nombre de variables d'état de \mathcal{M} . Son espace d'états est donc $\{0, 1\}^n$.
- m est le nombre d'entrées de la machine, donc l'ensemble des entrées qui peuvent être utilisées pour calculer les sorties et l'état successeur est $\{0, 1\}^m$.
- ω est le vecteur des r fonctions de sortie partiellement définies de la machine. Chaque des fonctions du vecteur ω est une fonction de $\{0, 1\}^n \times \{0, 1\}^m$ dans $\{0, 1\}^r$.
- δ est la fonction de transition partiellement définie de la machine. δ est une fonction de $\{0, 1\}^n \times \{0, 1\}^m$ dans $\{0, 1\}^n$, et est dénotée par un couple (Cns, \vec{f}) tel que :

$$\delta = \lambda \vec{y}. \lambda \vec{x}. (\mathbf{if} \ Cns(\vec{y}, \vec{x}) = 1 \ \mathbf{then} \ \vec{f}(\vec{y}, \vec{x}) \ \mathbf{else} \ \perp)$$

Cns dénote le domaine où la fonction de transition est définie, et \vec{f} définit sur le domaine Cns la valeur de l'état successeur. Cns est une fonction booléenne de $\{0, 1\}^n \times \{0, 1\}^m$ dans $\{0, 1\}$; \vec{f} est une fonction vectorielle $[f_1 \dots f_n]$, où chaque f_k est une fonction de $\{0, 1\}^n \times \{0, 1\}^m$ vers $\{0, 1\}$.

- $Init$ est l'ensemble des états initiaux de la machine.

Si ω ne dépend que des variables d'état de la machine, on dira que \mathcal{M} est une *machine de Moore*. Sinon, on dira que \mathcal{M} est une *machine de Mealy*. Une machine de Moore a un pouvoir de dénotation équivalent à celui d'une machine de Mealy, car toute machine de Mealy peut se transformer en une machine de Moore équivalente [74, 83].

La figure 17 illustre le processus d'obtention du 6-uplet qui décrit une machine séquentielle, tel qu'il est par exemple mis en oeuvre dans PÂRIS [53]. PÂRIS prend en entrée la description d'un réseau de machines communicantes, écrite dans le langage VHDL [7, 87, 41]. Un processus d'*exécution symbolique*, tel que celui décrit dans [19, 89, 94], permet de calculer les fonctions booléennes décrivant le comportement de chaque machine. Puis un processus de *composition symbolique de machines* [53] permet d'obtenir le comportement global du réseau de machines, sous la forme d'une machine de Mealy décrite par un 6-uplet $(n, m, r, \omega, \delta, Init)$. Durant cette compilation, des vérifications statiques sont faites pour s'assurer que le réseau de machines respecte un certain nombre de règles. Ainsi, il faut s'assurer qu'il n'y a pas de boucle fonctionnelle, sous peine d'avoir un système asynchrone. On vérifie aussi que la sémantique de VHDL n'est pas violée par une association de machines.

On appelle *relation de transition* ou *relation d'accessibilité* de la machine \mathcal{M} la fonction Δ définie de $\{0, 1\}^n \times \{0, 1\}^n$ dans $\{0, 1\}$, telle que $\Delta(\vec{y}, \vec{y}') = 1$ si et seulement si il existe une transition de l'état \vec{y} à l'état \vec{y}' . De même, on appelle *relation de sortie* la fonction Λ définie de $\{0, 1\}^n \times \{0, 1\}^r$ dans $\{0, 1\}$, telle que $\Lambda(\vec{y}, \vec{z}) = 1$ si et seulement si à partir de l'état \vec{y} de la machine, on peut produire la sortie \vec{z} en acceptant une entrée. Plus

formellement, on a :

$$\Delta = \lambda \vec{y}. \lambda y'. (\exists \vec{x} \text{ Cns}(\vec{y}, \vec{x}) \wedge \left(\bigwedge_{k=1}^n (y'_k \Leftrightarrow f_k(\vec{y}, \vec{x})) \right))$$

$$\Lambda = \lambda \vec{y}. \lambda \vec{z}. (\exists \vec{x} \left(\bigwedge_{k=1}^r (\omega_k(\vec{y}, \vec{x}) \Leftrightarrow \vec{z}_k) \right))$$

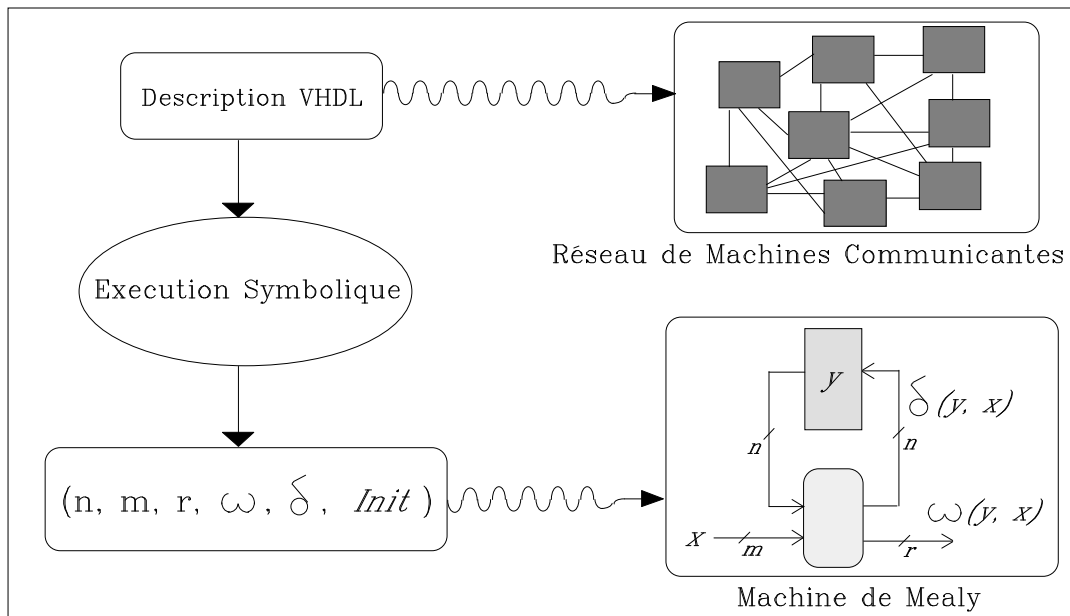


Figure 17. Compilation d'une description algorithmique en un 6-uplet $(n, m, r, \omega, \delta, Init)$.

On dit qu'une séquence d'entrée $\vec{x}_0, \dots, \vec{x}_k$ est *acceptée* par la machine \mathcal{M} si et seulement si pour tout état \vec{y}_0 de *Init*, la séquence d'états $\vec{y}_{j+1} = \delta(\vec{y}_j, \vec{x}_j)$ est bien définie pour $0 \leq j \leq k$. On appellera langage accepté la partie de $(\{0, 1\}^m)^*$ constituée de l'ensemble des séquences d'entrée acceptées. On dira qu'un état est *valide* s'il est atteignable à partir d'un état initial par une séquence acceptée. Pour une séquence d'entrée $\vec{x}_0, \dots, \vec{x}_k$ acceptée par \mathcal{M} , on dit que la séquence de sortie *générée* est $\vec{z}_0, \dots, \vec{z}_k$, où $\vec{z}_j = \omega(\vec{y}_j, \vec{x}_j)$, avec $\vec{y}_{j+1} = \delta(\vec{y}_j, \vec{x}_j)$ et $0 \leq j \leq k$. On appellera *langage généré* la partie de $(\{0, 1, \perp\}^r)^*$ constituée de l'ensemble des séquences générées.

3.2 Comparaison de machines séquentielles

De nombreux problèmes portant sur les machines séquentielles consistent à comparer les comportements observables, i.e. les séquences générées, de deux machines selon certains critères. Nous montrons ici que ce processus de comparaison peut être réalisé sans construire les graphes d'états des machines. Nous réduirons la comparaison de machines à l'évaluation de l'opération *Img* qui calcule l'image d'une fonction booléenne vectorielle.

3.2.1 Critères de comparaison

La notion de comparaison la plus courante est celle de l'équivalence observationnelle. On dira que deux machines sont *équivalentes* si et seulement si, d'une part elles acceptent les mêmes séquences d'entrées, d'autre part à partir de leurs états initiaux respectifs, elles produisent des séquences de sorties identiques quelle que soit la séquence d'entrées acceptée par les deux machines [75, 83].

On peut définir un grand nombre de critères de comparaison de machines en dehors de l'équivalence, par exemple, l'inclusion de langage accepté ou généré, et toutes les variations qui s'ensuivent. Il est montré dans [121] comment, à partir du critère de comparaison entre deux machines \mathcal{M}_1 et \mathcal{M}_2 , on peut construire une machine composée du produit de \mathcal{M}_1 et de \mathcal{M}_2 , avec l'ajout d'une machine à états finis prenant comme entrée les sorties de $\mathcal{M}_1 \times \mathcal{M}_2$. La Figure 18 illustre ce processus de comparaison. Le problème est alors de montrer que la machine produit génère le langage 1^* .

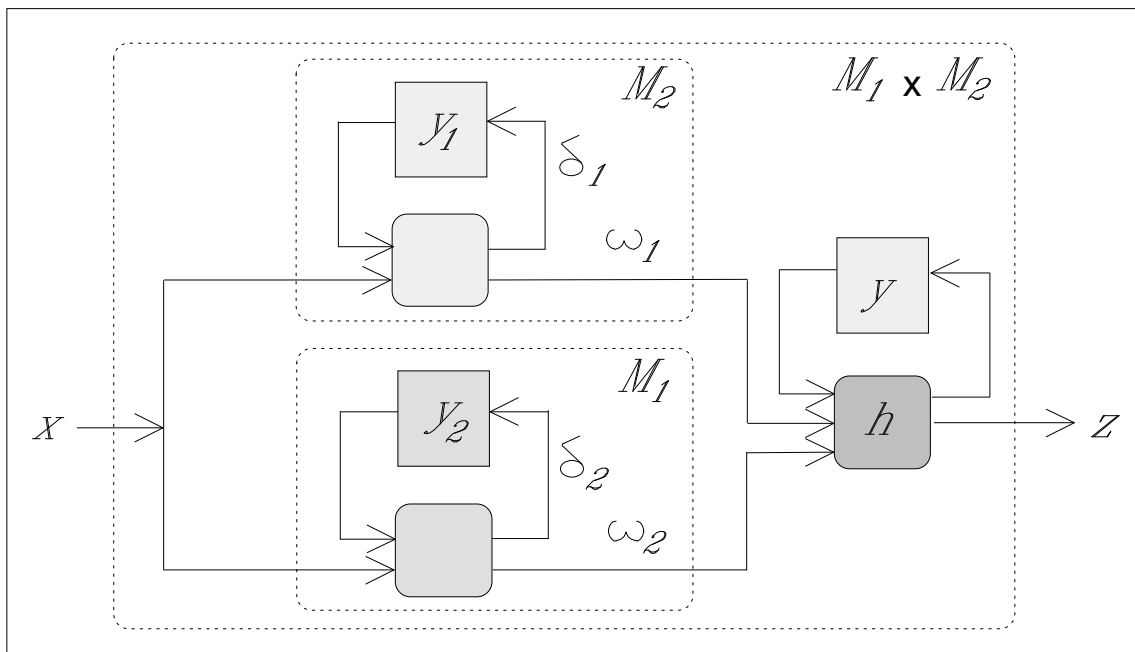


Figure 18. Machine produit pour la comparaison des langages générés.

3.2.2 Algorithme de comparaison

La comparaison des langages générés par deux machines est ramené au problème de prouver qu'une machine génère le langage 1^* . Pour cela, il suffit de calculer l'ensemble des états valides de la machine produit [75] montrée Figure 18, et de vérifier que ceux-ci ne peuvent générer que la sortie 1.

La seule technique connue jusqu'à récemment pour comparer les comportements observables de deux machines consistait à simuler la machine produit à partir de ses états

produits initiaux, pour énumérer l'ensemble de ses états et transitions valides, et vérifier que ceux-ci ne peuvent produire que la sortie 1.

Le problème est évidemment l'explosion combinatoire : une machine $(n, m, \omega, \delta, Init)$ possède potentiellement 2^n états et 2^{n+m} transitions. La simulation est donc vite dépassée par le nombre prohibitif de cas à envisager. Cette technique fut ensuite améliorée [118] en n'énumérant que des cubes représentant des groupes de transitions. Cependant, l'énumération des états valides demeure. Les limites des méthodes présentées ci-dessus découlent du fait que les ensembles d'états sont représentés en *extention*, et donc que les états sont manipulés individuellement.

Afin de se libérer de ces limites, nous proposons ici de manipuler des ensembles d'états, représentés en *intention* par leur fonction caractéristique. L'ensemble des états valides d'une machine $(n, m, r, \omega, \delta, Init)$ peut être défini comme la limite de la suite d'ensembles $(V_k)_{k \geq 0}$ définie par les équations

$$V_0 = Init, \quad \text{et} \quad (3.1)$$

$$V_{k+1} = V_k \cup \{\vec{y}' / \exists \vec{y} \in V_k, \exists \vec{x} \in \{0, 1\}^m, \vec{y}' = \delta(\vec{y}, \vec{x})\}. \quad (3.2)$$

La preuve se fait en montrant que, pour $k \geq 0$, on a l'égalité

$$\{\omega(\vec{y}, \vec{x}) / \vec{y} \in V_k, \vec{x} \in \{0, 1\}^m\} = \{1\}. \quad (3.3)$$

Les équations ci-dessus peuvent s'écrire avec l'opération "*Img*", où $Img(g, A)$ est l'ensemble $\{g(x) / x \in A\}$, c'est à dire l'image de la fonction g sur l'ensemble A . Le second terme droite de 3.2 est $Img(\delta, V_k \times \{0, 1\}^m)$, c'est à dire $Img(\vec{f}, \lambda \vec{y}. \lambda \vec{x}. (V_k(\vec{y}) \wedge Cns(\vec{y}, \vec{x})))$. De même, le terme gauche de l'équation 3.3 est $Img(\omega, V_k \times \{0, 1\}^m)$, c'est à dire $Img(\omega, V_k)$. Le calcul des états valides devient donc :

$$V_0 = Init, \quad \text{et} \quad (3.4)$$

$$V_{k+1} = \lambda \vec{y}. \left(V_k(\vec{y}) \vee Img(\vec{f}, \lambda \vec{y}. \lambda \vec{x}. (V_k(\vec{y}) \wedge Cns(\vec{y}, \vec{x}))) (\vec{y}) \right). \quad (3.5)$$

Cette équation de point fixe correspond à un parcours *virtuel en avant et en largeur d'abord* du diagramme d'états de la machine séquentielle \mathcal{M} . La figure ci-dessous montre un calcul des états valides, effectué en 4 étapes en utilisant les équations 3.1 et 3.2. Chacun des ensembles grisés est l'ensemble des états nouvellement découverts à chaque itération de l'équation 3.2, qui est égal à $(V_{k+1} \setminus V_k)$.

Figure 19. Parcours virtuel du graphe de transition.

3.2.3 Le terme critique : l’image

Les équations 3.4 et 3.5 n’utilisent que des combinaisons booléennes élémentaires ($\neg, \vee, \wedge, \dots$), plus l’opération *Img*. Comme les premières sont polynomiales vis-à-vis de la taille des DAGs, la complexité du calcul dépend directement du coût de l’opération *Img*. Nous étudierons dans le Chapitre 4 la complexité de l’évaluation de *Img*, et nous montrerons qu’elle est la seule opération non polynomiale dans cet algorithme. Nous proposerons alors plusieurs techniques d’évaluation de *Img*.

3.3 Vérification de propriétés temporelles

Si la comparaison de machines permet de vérifier l’adéquation du comportement observable d’une machine avec un autre comportement décrit par une autre machine, certaines propriétés observables ne peuvent pas toujours être ainsi exprimées. Par exemple, la propriété observable “si à un moment, tel événement survient, alors tel autre événement se produira dans le futur” ne peut être exprimée par une machine d’états finis. Une discussion approfondie sur les puissances d’expression de propriétés observables sera trouvée dans [59].

Une autre façon de décrire un comportement observable est d’utiliser un système formel possédant une sémantique adéquate sur un domaine apte à modéliser le temps. On appelle logique temporelle un tel système. Plusieurs systèmes de logique temporelle ont été élaborés : logique temporelle axiomatisée dans le premier ordre, logique des intervalles, logique temporelle linéaire, logique arborescente, ... Il n’est pas de notre propos de présenter ici toutes ces approches, on trouvera dans [59] un exposé de ces différentes logiques. Nous ne retiendrons qu’un système, le système CTL (*Computational Tree Logic*), c’est à dire la logique temporelle arborescente.

Plusieurs méthodes ont été proposées pour vérifier qu’une machine séquentielle satisfait une propriété temporelle. L’algorithme “Model Checking” [40] est un ensemble unifié d’algorithmes qui permet de vérifier automatiquement des propriétés exprimées dans le

formalisme de la logique temporelle arborescente. Cette technique de vérification requiert la construction partielle ou totale du graphe de transition de la machine, et est basée sur des algorithmes de parcours de ce graphe de transition. Cette technique est limitée par la taille du graphe de transition, car coûteuse en mémoire et en temps de par l'explosion combinatoire.

Nous montrons ici que les techniques de manipulations symboliques que nous avons décrites dans les deux premiers chapitres peuvent être utilisées pour la vérification de formules temporelles, sans qu'aucun diagramme d'état ne soit construit. Nous décrivons d'abord la syntaxe et la sémantique des formules CTL à vérifier, et le type de machines qui seront traitées. Puis nous donnons les algorithmes de vérification et nous montrons comment ceux-ci s'appuient sur l'opération *Pre*.

3.3.1 Syntaxe et sémantique de CTL

Le machine séquentielle \mathcal{M} que nous traitons est une machine de Moore incomplètement spécifiée. Elle est définie par un 6-uplet $(n, m, r, \omega, \delta, Init)$, où ω ne dépend que de l'état de la machine.

On suppose que la formule $(\forall \vec{y} \exists \vec{x} Cns(\vec{y}, \vec{x}))$ est une tautologie, c'est à dire que tout état possède au moins un successeur. Ceci n'est pas restrictif, puisque la plupart des formules CTL n'ont aucun sens sur des états sans successeur [40]. Si cette condition n'est pas vérifiée, la fonction $\lambda \vec{y}.(\forall \vec{x} \neg Cns(\vec{y}, \vec{x}))$ est la fonction caractéristique de l'ensemble *EtatPuit* des états sans successeur. Soit *VersEtatPuit* l'ensemble des états sans successeur ou conduisant nécessairement à un état sans successeur. L'ensemble *VersEtatPuit* est l'ensemble des états satisfaisant la formule CTL $AF(EtatPuit)$, et peut donc se calculer par une équation de point fixe, comme on le verra ci-dessous. Le nouveau domaine à considérer pour la fonction δ est alors $\lambda \vec{y}.\lambda \vec{x}.(\neg VersEtatPuit(\vec{y}) \wedge Cns(\vec{y}, \vec{x}))$.

CTL permet d'exprimer des propriétés sur les états et les transitions d'une machine séquentielle [40]. Les formules temporelles considérées pour la vérification sont les formules d'état de CTL, qui décrivent des propriétés relatives aux états de la machine. Les formules d'état de CTL et leur sémantique vis-à-vis d'une machine de Moore $\mathcal{M} = (n, m, r, \omega, \delta, Init)$ sont les suivantes:

1. $y_1, y_2, \dots, y_n, z_1, \dots, z_r$ sont des formules d'état. Pour tout état \vec{y} de la machine, $\vec{y} \models y_k$ si et seulement si $y_k = 1$. $\vec{y} \models z_k$ si et seulement si la valeur de la k -ième composante de $\omega(\vec{y})$ est 1.
2. Si f et g sont des formules d'état, alors $(\neg f)$, $(f \wedge g)$, $(f \vee g)$, $(f \Leftrightarrow g)$, et $(f \Rightarrow g)$ sont des formules d'état. Les connecteurs logiques ont leur sens usuel, par exemple $s \models (f \wedge g)$ si et seulement si $s \models f$ et $s \models g$.
3. Si f est une formule d'état, alors $EX(f)$ et $AX(f)$ sont des formules d'état. Pour tout état \vec{y} , $\vec{y} \models EX(f)$ si et seulement si il existe une entrée \vec{x} acceptée par la machine telle que $\delta(\vec{y}, \vec{x}) \models f$. On définit la formule $AX(f)$ par $AX(f) = \neg EX(\neg f)$.

4. Si f et g sont des formules d'état, alors $E[fUg]$ est une formule d'état. Pour tout état \vec{y} , $\vec{y} \models E[fUg]$ si et seulement si il existe un chemin $(\vec{y}_0, \vec{y}_1, \dots)$, tel que $\vec{y}_0 = \vec{y}$ et $\exists k ((\vec{y}_k \models g) \wedge (\forall j (0 \leq j < k \Rightarrow \vec{y}_j \models f))$.
5. Si f et g sont des formules d'état, alors $A[fUg]$ est une formule d'état. Pour tout état \vec{y} de M , $\vec{y} \models A[fUg]$ si et seulement si pour tout chemin $(\vec{y}_0, \vec{y}_1, \dots)$ tel que $\vec{y}_0 = \vec{y}$, et $\exists k ((\vec{y}_k \models g) \wedge (\forall j (0 \leq j < k \Rightarrow \vec{y}_j \models f))$.

Des abréviations commodes sont introduites : $EF(f) = E[1Uf]$, i.e. il existe un chemin comprenant un état satisfaisant f ; $AF(f) = A[1Uf]$, i.e. sur tout chemin il existe un état où f est satisfaite; $EG(f) = \neg AF(\neg f)$, i.e. il existe un chemin où f est toujours satisfaite; $AG(f) = \neg EF(\neg f)$, i.e. f est vraie le long de tout chemin.

3.3.2 Algorithme de vérification de formules CTL

On peut vérifier qu'une machine séquentielle \mathcal{M} satisfait une *propriété de sûreté* f en comparant la machine avec un automate obtenu [22] à partir de la formule f . L'automate \mathcal{M}_f associé à une propriété de sûreté f est tel que pour toute machine \mathcal{M} , on a $\mathcal{M} \models f$ si et seulement si \mathcal{M} et \mathcal{M}_f sont équivalentes sur les signaux observables de \mathcal{M}_f . En substance, \mathcal{M}_f est le plus petit modèle de f qui décrit tous les comportements possibles des modèles de f . \mathcal{M}_f est aussi appelé *modèle universel*. Dans le cas des propriétés de sûreté, la preuve peut donc se faire directement par comparaison de machines. C'est cette approche qui a été retenue par l'équipe du langage LUSTRE de Grenoble [71]. Par exemple, pour vérifier la propriété "à tout moment, si $y = 1$ alors à l'instant d'après on a forcément $z = 1$ ", qui correspond à la formule CTL $AG(y \Rightarrow AX(z))$, il suffit de comparer les signaux y et z de \mathcal{M} avec ceux définis par la machine de Moore dont le graphe de transition est montré Figure 20. Mais la puissance d'expression de CTL est strictement supérieure à celle des propriétés de sûreté, ce qui nécessite un algorithme plus général.

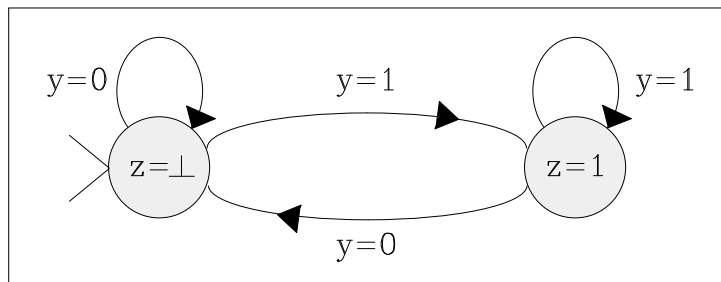


Figure 20. Modèle universel de $AG(y \Rightarrow AX(z))$.

L'algorithme "Model Checking" [40] permet de vérifier si $(\vec{y} \models f)$ pour un état \vec{y} de la machine et une formule CTL f . Le Model Checking est basé sur des parcours du graphe d'états et de transitions de la machine, et requiert la construction de celui-ci. Cette technique est donc limitée par la taille du graphe d'états de la machine, car la preuve de $(\vec{y} \models f)$ se fait en $O(\text{length}(f) \times (S + R))$, où S est le nombre d'états et R le nombre de transitions [40].

Afin de s'affranchir de cette limite, les ensembles d'états seront considérés non pas en *extention*, mais en *intention*, dénotés par leurs fonctions caractéristiques. Au lieu de

vérifier directement par énumération que $(\vec{y} \models f)$, on calcule la fonction caractéristique \mathbf{F} des états qui satisfont f , puis on vérifie que $\mathbf{F}(\vec{y}) = 1$. Nous décrivons ci-après un algorithme basé sur cette technique, qui ne nécessite aucune construction de graphe d'état.

L'algorithme prend comme entrées une machine \mathcal{M} décrite par son 6-uplet $(n, m, r, \omega, \delta, Init)$ et la formule temporelle f à valider. Il calcule récursivement l'ensemble d'états de la machine \mathcal{M} qui satisfont la formule f à partir des ensembles d'états satisfaisant les sous-formules de f . A chaque étape, il n'y a que cinq cas à considérer, correspondant aux cinq types de formules donnés dans la section précédente.

Les formules de type propositionnelle

Les ensembles d'états sont représentés par les DAGs de leurs fonctions caractéristiques. Ainsi les formules de type (1) et (2) sont triviales à traiter. Par exemple, si $f = (f_1 \wedge f_2)$, et \mathbf{F}_1 et \mathbf{F}_2 sont les fonctions caractéristiques des ensembles d'états satisfaisant respectivement f_1 et f_2 , alors \mathbf{F} est $\lambda\vec{y}.(\mathbf{F}_1(\vec{y}) \wedge \mathbf{F}_2(\vec{y}))$. Le coût de calcul du DAG de \mathbf{F} est en $O(|\mathbf{F}_1| \times |\mathbf{F}_2|)$. De la même façon, $\vec{y} \models f$ si et seulement si $\mathbf{F}(\vec{y}) = 1$, ce qui est calculé en $O(n)$. Finalement, $(\mathcal{M} \models f)$ si et seulement si $(Init \Rightarrow \mathbf{F})$ est une tautologie, ce qui est calculé en $O(|Init| \times |\mathbf{F}|)$. Les autres types de formules temporelles sont moins immédiats à traiter, et leur traitement repose sur des équations de points fixes [46, 29, 30].

Les formules de type EX et AX

Soit f une formule et \mathbf{F} l'ensemble d'états satisfaisant f . L'ensemble d'états \mathbf{EX} qui satisfont $EX(f)$ est défini par $\mathbf{EX} = \{\vec{y} / \exists \vec{x} \delta(\vec{y}, \vec{x}) \in \mathbf{F}\}$. Sa fonction caractéristique est :

$$\mathbf{EX} = \lambda\vec{y}.(\exists \vec{x} Cns(\vec{y}, \vec{x}) \wedge \mathbf{F}(\vec{f}(\vec{y}, \vec{x}))). \quad (3.6)$$

Comme $AX(f) = \neg EX(\neg f)$, la fonction caractéristique \mathbf{AX} de l'ensemble d'états satisfaisant $AX(f)$ est :

$$\mathbf{AX} = \lambda\vec{y}.(\neg(\exists \vec{x} Cns(\vec{y}, \vec{x}) \wedge \neg \mathbf{F}(\vec{f}(\vec{y}, \vec{x}))). \quad (3.7)$$

Les formules de type EU et AU

Soient f et g deux formules et \mathbf{F} et \mathbf{G} les ensembles d'états qui satisfont respectivement f et g . L'ensemble \mathbf{EU} d'états de la machine \mathcal{M} qui satisfont la formule $E[fUg]$ est défini comme la limite de la suite d'ensembles (\mathbf{E}_k) suivante [40, 21, 46, 29] :

$$\begin{aligned} \mathbf{E}_0 &= \mathbf{G}, & \text{et} \\ \mathbf{E}_{k+1} &= \mathbf{E}_k \cup \{\vec{y} / (\vec{y} \in \mathbf{F}) \wedge (\exists \vec{x} \delta(\vec{y}, \vec{x}) \in \mathbf{E}_k)\}. \end{aligned}$$

Les fonctions caractéristiques de ces ensembles sont les suivantes :

$$\mathbf{E}_0 = \mathbf{G}, \quad \text{et} \quad (3.8)$$

$$\mathbf{E}_{k+1} = \lambda\vec{y}. \left(\mathbf{E}_k(\vec{y}) \vee \left(\mathbf{F}(\vec{y}) \wedge (\exists \vec{x} Cns(\vec{y}, \vec{x}) \wedge \mathbf{E}_k(\vec{f}(\vec{y}, \vec{x}))) \right) \right). \quad (3.9)$$

De la même façon, l'ensemble \mathbf{AU} des états de la machine \mathcal{M} qui satisfont la formule $A[fUg]$ est défini comme la limite de la suite d'ensembles (\mathbf{A}_k) suivante :

$$\begin{aligned} \mathbf{A}_0 &= \mathbf{G}, & \text{et} \\ \mathbf{A}_{k+1} &= \mathbf{A}_k \cup \{\vec{y} / (\vec{y} \in \mathbf{F}) \wedge (\forall \vec{x} \delta(\vec{y}, \vec{x}) \in \mathbf{A}_k)\}. \end{aligned}$$

Les fonctions caractéristiques de ces ensembles s'écrivent :

$$\mathbf{A}_0 = \mathbf{G}, \quad \text{et} \quad (3.10)$$

$$\mathbf{A}_{k+1} = \lambda \vec{y}. \left(\mathbf{A}_k(\vec{y}) \vee \left(\mathbf{F}(\vec{y}) \wedge (\forall \vec{x} \text{Cns}(\vec{y}, \vec{x}) \Rightarrow \mathbf{A}_k(\vec{f}(\vec{y}, \vec{x}))) \right) \right). \quad (3.11)$$

Ces équations de points fixes correspondent à *un parcours virtuel en arrière et en largeur d'abord* du graphe d'état de la machine, c'est à dire que la relation d'accéssibilité utilisée pendant le parcours est "être antécédant de". La Figure 21 montre le calcul de l'ensemble des états satisfaisant $A[fUg]$. Dans cette exemple, l'ensemble des états satisfaisant f (respectivement g) est donné en pointillé et étiqueté \mathbf{F} (respectivement \mathbf{G}). Les flèches sont les transitions d'état à état. En gris sont représentées les quatres parties constituant l'ensemble des états satisfaisant $A[fUg]$, trouvés en quatre étapes à partir des équations de point fixe 3.10 et 3.11.

Figure 21. Calcul des états satisfaisant $A[fUG]$.

3.3.3 Le terme critique : l'image réciproque

Les équations 3.6 à 3.11 peuvent se réécrire à partir d'une fonction "Pre", telle que $Pre(\vec{f}, \text{Cns}, \chi)$ est la fonction caractéristique des états qui possèdent au moins un successeur appartenant à l'ensemble χ . Pre est définie par :

$$Pre(\vec{f}, \text{Cns}, \chi) = \lambda \vec{y}. (\exists \vec{x} \text{Cns}(\vec{y}, \vec{x}) \wedge \chi(\vec{f}(\vec{y}, \vec{x}))).$$

De cette façon, on réécrit les équations données précédemment en :

$$\mathbf{EX} = Pre(\vec{f}, Cns, \mathbf{F}) \quad (3.12)$$

$$\mathbf{AX} = \neg Pre(\vec{f}, Cns, \neg \mathbf{F}) \quad (3.13)$$

$$\mathbf{E}_0 = \mathbf{G}, \quad \text{et} \quad (3.14)$$

$$\mathbf{E}_{k+1} = \lambda \vec{y}. (\mathbf{E}_k(\vec{y}) \vee (\mathbf{F}(\vec{y}) \wedge Pre(\vec{f}, Cns, \mathbf{E}_k)(\vec{y}))). \quad (3.15)$$

$$\mathbf{A}_0 = \mathbf{G}, \quad \text{et} \quad (3.16)$$

$$\mathbf{A}_{k+1}(\vec{y}) = \lambda \vec{y}. (\mathbf{A}_k(\vec{y}) \vee (\mathbf{F}(\vec{y}) \wedge \neg Pre(\vec{f}, Cns, \mathbf{A}_k)(\vec{y}))) \quad (3.17)$$

D'un point de vue calculatoire, cela signifie que ces 5 cas peuvent être traités en utilisant les opérateurs booléens usuels ($\neg, \vee, \wedge, \dots$), de complexité polynomiale sur les DAGs, plus le calcul de la fonction Pre . Le coût global de la validation d'une formule temporelle sur une machine séquentielle dépend donc du calcul de Pre . Nous étudierons dans le Chapitre 5 la complexité de cette opération, et nous montrerons qu'elle est non polynomiale. Nous proposerons alors plusieurs techniques d'évaluation de Pre .

3.4 Minimisation de machines séquentielles

La difficulté de la synthèse de matériel à partir d'une description fonctionnelle d'une machine vient de ce que ce problème recouvre simultanément plusieurs problèmes d'optimisation souvent contradictoires. On doit synthétiser un circuit de taille aussi petite que possible, il faut donc diminuer le nombre de portes logiques et s'attacher à ce que l'assemblage de celles-ci occupe une surface minimale. Le circuit doit être suffisamment rapide, il faut donc que les connexions soient les plus courtes possibles, et que la profondeur du circuit (nombre maximal de portes logiques entre les entrées et les sorties) soit la plus faible possible. Il y a des contraintes électriques et de consommation, il faut donc que le nombre de connexions à l'entrée d'une porte (*fan-in*) et à sa sortie (*fan-out*) respectent des lois bien précises. Si l'implémentation du circuit se fait sur un PLA (*Programmable Logic Array*), fournissant des portes et connexions prédéfinies (à charge de l'utilisateur de construire le circuit en ajoutant ou coupant certaines connexions), il faut aussi tenir compte du cadre non modifiable de cette structure pour l'optimisation des divers critères cités.

Cependant, un aspect à peu près commun à toutes ces optimisations est la minimisation logique. Informellement, ceci pose la question de la "quantité" de logique (i.e. le nombre de portes) nécessaire pour implémenter une fonctionnalité donnée, éventuellement vectorielle et/ou séquentielle. Nous expliquons ici quelles sont les minimisations qui peuvent intervenir pour réduire la "complexité" (en terme de circuit, ou plus formellement en terme de complexité combinatoire et séquentielle au sens de [109]) d'une machine séquentielle.

3.4.1 Minimisation de la logique combinatoire

La minimisation de la logique combinatoire consiste à trouver un circuit “minimal” implémentant une fonction booléenne donnée. Cette minimalité doit être comprise vis-à-vis d’une mesure de complexité. Celle-ci est définie comme le nombre de portes du circuit, où ces portes sont des instances d’un ensemble de fonctions booléennes élémentaires. Le problème de la minimisation combinatoire est évidemment NP-difficile. Nous abordons succinctement dans L’Annexe E le problème de la minimisation de fonctions booléennes partielles, et nous proposons plusieurs techniques de réduction, certaines basées sur des réécriture de DAGs, d’autres sur une minimisation de la somme de produits associée. Il faut noter que les techniques de minimisation que nous avons définies ont été reprises par nous-mêmes et d’autres équipes pour améliorer des algorithmes de calcul sur les DAGs, notamment la composition.

3.4.2 Elimination des variables d’états redondantes

Le but de l’élimination des variables d’états redondantes est de réduire la mémoire de la machine sans modifier son comportement observable. On dit qu’une variable d’état d’une machine est redondante si elle n’apporte aucune information sur l’état de la machine lorsque celle-ci évolue dans son espace d’états valides. Autrement dit, une variable d’état y_k est redondante si et seulement si, pour tout état valide, la valeur de y_k est entièrement déterminée par la valeur des autres variables d’états. Ceci implique qu’il existe une fonction des autres variables d’états qui permet de calculer la valeur y_k sur l’espace des états valides. *Valid* étant la fonction caractéristique des états valides, nous disons [17] qu’une variable d’état y_k est redondante dans le système des n variables d’états y_1, \dots, y_n si et seulement si la formule

$$\forall y_1 \dots \forall y_{k-1} \forall y_{k+1} \dots \forall y_n \exists f \forall y_k (Valid(\vec{y}) \Rightarrow (f \Leftrightarrow y_k))$$

est une tautologie. Si tel est le cas, la fonction de Skolem de f donne la fonction de réécriture de y_k en fonction des autres variables d’états. En reprenant les résultats acquis au Chapitre 1, et à partir de cette équation, nous montrons [17] que y_k est redondante si et seulement si :

$$Valid[y_k \leftarrow 0] \wedge Valid[y_k \leftarrow 1] = 0$$

Dans ce cas, la fonction de réécriture de y_k est la fonction de Skolem définie par :

$$Valid[y_k \leftarrow 1] \vee (p \wedge \neg Valid[y_k \leftarrow 0])$$

Ainsi le test de redondance et le calcul de la fonction de réécriture est quadratique par rapport à la taille du DAG de *Valid*.

L’itération aveugle du processus d’élimination (test de redondance et réécriture) que nous avons défini ne conduit pas à une minimisation optimale, comme il a été remarqué dans [86]. En effet, le nombre de variables d’états éliminées dépend de l’ordre dans lequel est effectué l’élimination. Considérons l’exemple suivant, où 5 variables d’états y_k codent 4 états s_j :

	y_1	y_2	y_3	y_4	y_5
s_1	0	0	0	1	1
s_2	0	1	0	0	1
s_3	1	1	1	0	1
s_4	1	0	0	0	0

Table 3. Exemple de variables redondantes.

On voit que $\{y_1, y_2\}$ sont redondantes. Mais la meilleure solution est $\{y_3, y_4, y_5\}$. Si on élimine d'abord y_1 , alors la meilleure solution ne pourra être trouvée. Il faut donc considérer tous les choix d'élimination afin d'obtenir une solution optimale. Cette approche est évidemment exponentielle. Cependant, un algorithme *branch and bound* permet d'obtenir une solution optimale en un temps raisonnable [86] pour certaines machines.

Chaque occurrence d'une variable d'état redondante est substituée par sa fonction de réécriture dans la fonction de transition et la fonction de sortie de la machine. A chaque élimination, une variable disparaît, on peut ainsi espérer diminuer la taille du circuit correspondant. Mais sur la machine réelle, ce processus de substitution correspond à l'ajout d'une partie combinatoire de codage/décodage, ce qui peut complexifier le circuit. Aussi le meilleur choix est-il difficile à faire, et il n'est pas toujours souhaitable de réduire optimalement la mémoire de la machine.

Cette technique d'élimination des variables redondantes, plus la minimisation de la partie combinatoire sur l'espace des états valides d'une machine séquentielle, permet de réduire la complexité d'un circuit séquentiel. C'est cette approche qui a été utilisée par Gérard Berry pour optimiser les circuits synthétisés à partir de programmes ESTEREL [14, 15, 16], ce qui donne de bonnes performances.

3.4.3 Minimisation des variables d'états par réencodage

L'élimination des variables d'états redondantes était un premier pas vers la diminution de la mémoire. La minimisation des variables d'états par réencodage revient à calculer une machine minimale équivalente à la machine initiale. Le réencodage permet une minimisation *absolue*. Une machine possédant r classes d'états équivalents peut être réencodée sur une machine possédant $\lceil \log_2 r \rceil$ variables d'état. Cependant, cette approche risque de complexifier considérablement la partie combinatoire du circuit, comme nous l'avons précisé ci-dessus. Une approche non optimale en terme du nombre de variables d'état, mais optimale en terme du nombre d'états valides, consiste à réencoder la machine sur ses classes d'états équivalents.

Le réencodage peut se faire symboliquement en concevant une fonction de projection qui va projeter une classe d'équivalence sur un état unique de cette classe. Cette projection est obtenue en utilisant l'hyper-distance "plus proche interprétation" sur les interprétations (voir Section 4.5.2). Dans [86] est proposée une telle projection. Le calcul des classes d'états équivalents pour la réduction d'une machine séquentielle \mathcal{M} , ou pour l'obtention du modèle minimal équivalent à \mathcal{M} [23], s'effectue grâce à l'opérateur *Pre* décrit dans le Chapitre 5.

3.5 Conclusion

Nous avons distingué trois grandes problématiques sur les machines séquentielles : la comparaison de langages générés par deux machines ; la validation de propriétés temporelles sur une machine ; la réduction et/ou minimisation d’une machine. A partir d’une description algorithmique d’une machine à états finis, on sait générer un modèle décrit par des fonctions booléennes. C’est sur ce modèle que nous avons exprimé la résolution des deux premiers problèmes. Nous avons aussi étudié quelques aspects de la réduction et minimisation d’une machine.

Les méthodes de résolution proposées dans le passé pour la comparaison de machines et la validation de propriétés temporelles consistent en un traitement explicite des états et des transitions, et nécessitent donc l’énumération de ceux-ci, voire une construction explicite du graphe d’états et de transitions de la machine. Ces méthodes sont donc limitées par la taille (i.e. nombre d’états et de transitions) des machines traitées. Plus précisément, le temps et la mémoire que requièrent ces algorithmes sont linéairement dépendants de la taille de la machine.

Nous avons montré comment la comparaison de machines et la validation de propriétés temporelles peuvent être exprimées par des équations booléennes. L’idée est de remplacer le traitement individuel des états et des transitions par des manipulations de formules propositionnelles. Celles-ci sont des fonctions caractéristiques dénotant des ensembles d’états et de transitions. Les algorithmes que nous avons proposés correspondent donc à des calculs sur des ensembles *implicitement* représentés. Par exemple, la comparaison de machines correspond au calcul du plus petit ensemble d’états contenant un sous-ensemble d’états initiaux, clos par la relation d’accessibilité de la machine. Si il n’y pas de relation entre la taille de la représentation d’une formule propositionnelle et le nombre d’éléments qu’elle représente en tant que fonction caractéristique¹, on peut s’attendre à des algorithmes de preuve dont la complexité ne *dépend pas* du nombre d’états et de transitions de la machine. La complexité sera reportée sur la “complexité intrinsèque” des formules propositionnelles en présence.

Nous représenterons les fonctions booléennes par des TDGs (graphes de décision typés), présentés au Chapitre 2. La taille d’un DAG ne dépend pas du nombre d’interprétations qui le satisfait. Ceci permet, avec les équations booléennes que nous avons décrites, de nous affranchir de l’étroite dépendance avec la taille de la machine que subissent les techniques basées sur un parcours explicite du diagramme d’états de la machine. Dans nos équations booléennes sont utilisées les combinaisons booléennes usuelles ($\neg, \vee, \wedge, \dots$), plus deux opérateurs complexes que nous avons distingué : la comparaison de machines utilise l’opérateur *Img*, qui correspond au calcul de l’image d’une fonction, et la validation de propriétés temporelles utilise l’opérateur *Pre*, qui correspond au calcul d’une projection de l’image réciproque d’une fonction. Comme les combinaisons booléennes usuelles sont effectuées au pire quadratiquement sur les DAGs, il ne nous reste plus qu’à réaliser le calcul de *Img* et *Pre* pour obtenir deux algorithmes de preuve symboliques, dont les complexités dépendront de celles de *Img* et *Pre*. C’est ce qui constitue la matière des deux chapitres

¹C’est le nombre d’interprétations satisfiant la formule.

suivants.

Chapitre 4

Calcul de l'image d'une fonction

Nous présentons ici des algorithmes de calcul de l'image d'une fonction vectorielle booléenne. Etant donné les DAGs d'une fonction vectorielle \vec{f} de $\{0, 1\}^m$ dans $\{0, 1\}^n$, et le DAG d'une fonction χ de $\{0, 1\}^m$ dans $\{0, 1\}$, nous établissons tout d'abord la complexité d'évaluation de la fonction $Img(\vec{f}, \chi)$, définie de $\{0, 1\}^n$ dans $\{0, 1\}$. Puis nous introduisons la notion de restricteur d'image, qui permet de décomposer le calcul de l'image. Enfin nous donnons le schéma d'un algorithme permettant de calculer Img , et nous proposons deux instances de ce schéma. Les résultats présentés seront valables pour les deux représentations graphiques BDD et TDG, sauf lorsque cela sera explicitement spécifié.

Nous adoptons les notations suivantes. La fonction booléenne ε est l'identité ou la négation. Une variable du domaine $\{0, 1\}^m$ (respectivement du codomaine $\{0, 1\}^n$) est notée \vec{x} (respectivement \vec{y}). Une fonction booléenne est confondue avec son DAG. Les vecteurs de fonctions booléennes sont notés \vec{f} ou $[f_1 \dots f_n]$. Nous étendons les opérateurs booléens aux vecteurs, de façon à obtenir une structure d'espace vectoriel. Par exemple, $[f_1 \dots f_n] \vee [f'_1 \dots f'_n] = [(f_1 \vee f'_1) \dots (f_n \vee f'_n)]$, et le vecteur $\chi \wedge [f_1 \dots f_n]$ est égal à $[(\chi \wedge f_1) \dots (\chi \wedge f_n)]$.

4.1 Difficulté du calcul de $Img(\vec{f}, \chi)$

La difficulté intrinsèque du problème est indiquée par le théorème suivant. Calculer $Img(\vec{f}, 1)$ est NP-difficile¹. Il est même certain que le calcul de $Img(\vec{f}, 1)$ n'appartient pas à NP. En utilisant l'hypothèse que l'ordre des variables est fixé, on obtient le théorème 4.2 qui précise la complexité du problème.

Théorème 4.1 *Etant donné la forme canonique d'une fonction vectorielle \vec{f} , la canonisation de $Img(\vec{f}, 1)$ est NP-difficile.*

Preuve. Nous ne donnons que la preuve sur les DAGs. Soit $C = (\bigwedge_{k=1}^n c_k)$ une 3-forme normale conjonctive composée de n clauses c_k . Chaque clause c_k est une disjonction de trois littéraux, donc le calcul des DAGs de toutes les clauses se fait en $O(n)$.

¹On a $Img(\vec{f}, \chi) = \lambda \vec{y}. Img(\vec{f} @ [\chi], 1)(y_1, \dots, y_n, 1)$. Donc $Img(\vec{f}, \chi) = Img(\vec{f} @ [\chi], 1)[y_{n+1} \leftarrow 1]$, le terme de droite étant calculé en $O(|Img(\vec{f} @ [\chi], 1)|)$. Le calcul de $Img(\vec{f}, 1)$ est donc aussi difficile que celui de $Img(\vec{f}, \chi)$.

Soit $\chi = \text{Img}(\vec{f}, 1)$. Evaluer le terme $\chi(1, \dots, 1)$ se fait en $O(n)$. Or C est satisfiable si et seulement si $\chi(1, \dots, 1) = 1$, donc calculer χ est NP-difficile. \square

Théorème 4.2 *Le problème de la composition se réduit à celui du calcul de l'image.*

Preuve. Rappelons que $\vec{x} = [x_1 \dots x_m]$, et $\vec{y} = [y_1 \dots y_n]$. Soient les DAGs des fonctions $\lambda\vec{x}.f_1(\vec{x})$, \dots , $\lambda\vec{x}.f_n(\vec{x})$, et $\lambda\vec{y}.g(\vec{y})$. Le problème de la composition consiste à calculer le DAG de la fonction $\lambda\vec{x}.(g(f_1(\vec{x}), \dots, f_n(\vec{x})))$. Soit la fonction vectorielle \vec{F} définie par :

$$\vec{F} = \lambda(\vec{x}@\vec{y}).[x_1 \dots x_m (y_1 \Leftrightarrow f_1(\vec{x})) \dots (y_n \Leftrightarrow f_n(\vec{x})) g(\vec{y})].$$

Les DAGs de \vec{F} se construisent en $O(m + |g| + \sum_{k=1}^n |f_k|)$. On a :

$$\begin{aligned} \text{Img}(\vec{F}, 1) &= \lambda[z_1 \dots z_{m+n+1}].(\exists\vec{x} \exists\vec{y} \left(\bigwedge_{k=1}^m (z_k \Leftrightarrow x_k) \right) \wedge \\ &\quad \left(\bigwedge_{k=1}^n (z_{m+k} \Leftrightarrow (y_k \Leftrightarrow f_k(\vec{x}))) \right) \wedge \\ &\quad (z_{m+n+1} \Leftrightarrow g(\vec{y}))) \end{aligned}$$

On a donc :

$$\begin{aligned} \text{Img}(\vec{F}, 1)(z_1, \dots, z_m, 1, \dots, 1) &= \exists\vec{x} \exists\vec{y} \left(\bigwedge_{k=1}^m z_k \Leftrightarrow x_k \right) \wedge \left(\bigwedge_{k=1}^n y_k \Leftrightarrow f_k(\vec{x}) \right) \wedge g(\vec{y}) \\ &= \exists\vec{y} \left(\bigwedge_{k=1}^n (y_k \Leftrightarrow f_k(z_1, \dots, z_m)) \right) \wedge g(\vec{y}) \\ &= g(f_1(z_1, \dots, z_m), \dots, f_n(z_1, \dots, z_m)) \end{aligned}$$

La composition $g(f_1, \dots, f_n)$ est égale à $\lambda[x_1 \dots x_m].\text{Img}(\vec{F}, 1)(x_1, \dots, x_m, 1, \dots, 1)$. Ce dernier DAG se calcule en $O(|\text{Img}(\vec{F}, 1)|)$, ou même en $O(n)$ si le DAG de $\text{Img}(\vec{F}, 1)$ sur le système $\{z_1, \dots, z_m, z_{m+1}, \dots, z_{m+n}, z_{m+n+1}\}$ est ordonné de telle façon que $\{z_{m+1}, \dots, z_{m+n}, z_{m+n+1}\} < \{z_1, \dots, z_m\}$. \square

Le Théorème 4.2 montre que la composition se réduit polynomialement au calcul de l'image. Les remarques faites Section 2.4.3 sur la complexité de la composition sont donc applicables. Le calcul de l'image est au moins exponentiel en mémoire dans le pire des cas pour un ordre fixé. Même si un oracle donnait dynamiquement un bon ordre des variables minimisant les DAGs des fonctions manipulées, le calcul de l'image reste au moins exponentiel en mémoire dans le pire des cas.

Ces résultats de complexité exponentielle sont obtenus vis-à-vis de la mémoire nécessaire pour construire le DAG de $\text{Img}(\vec{f}, \chi)$. On peut s'interroger sur la complexité de problèmes faisant intervenir le calcul de l'image avec une sortie bornée (par exemple un problème dont la réponse est "vrai" ou "faux"). Déterminer si $\text{Img}(\vec{f}, 1) = 1$ pour une fonction vectorielle \vec{f} donnée est un de ces problèmes, et un des plus intéressants, car il permettrait d'accélérer considérablement le calcul de l'image (voir Section 4.3 sur l'utilisation du cache). Cependant, les théorèmes suivants affirment la non polynomialité de ce test.

Théorème 4.3 *Etant donnés les DAGs d'une fonction vectorielle \vec{f} , le problème de savoir si $\text{Img}(\vec{f}, 1) = 1$ est NP-difficile.*

Preuve. Soit $C = (\bigwedge_{k=1}^n c_k)$ une 3-forme normale conjonctive composée de n clauses c_k . Le calcul des DAGs f_1, \dots, f_n des n clauses est en $O(n)$. Nous créons alors n variables x_1, \dots, x_n n'ayant pas d'occurrence dans C , ce qui peut se faire polynomialement. Construire les n DAGs des formules $(f_k \wedge x_k)$ est en $O(n)$. $\text{Img}([f_1 \wedge x_1 \dots f_n \wedge x_n], 1) = 1$ si et seulement si le point $[1 \dots 1]$ appartient à $\text{Img}([f_1 \dots f_n], 1)$, c'est à dire si et seulement si C est satisfiable, donc tester si $\text{Img}(\vec{f}, 1) = 1$ est NP-difficile. \square

Théorème 4.4 *Etant donné $\vec{y} \in \{0, 1\}^n$, et étant donnés les DAGs d'une fonction vectorielle \vec{f} , le problème de savoir si $\vec{y} \in \text{Img}(\vec{f}, 1)$ est NP-complet.*

Preuve. Ce test peut être réalisé par l'algorithme non déterministe suivant, qui est polynomial :

```

function belongs-to?( $\vec{y}$ ,  $\vec{f}$ );
choose  $\vec{x}$  in  $\{0, 1\}^m$ ;                               /* en  $O(1)$  */
let  $\vec{y}' = \vec{f}(\vec{x})$  in                                   /* en  $O(n \times m)$  */
    if  $\vec{y}' = \vec{y}$  then success else failure;           /* en  $O(n)$  */

```

Soit $C = (\bigwedge_{k=1}^n c_k)$ une 3-forme normale conjonctive composée de n clauses c_k . Le calcul des DAGs f_1, \dots, f_n des n clauses est en $O(n)$. Alors le point $[1 \dots 1]$ appartient à $\text{Img}([f_1 \dots f_n], 1)$ si et seulement si C est satisfiable. Donc tester si $\vec{y} \in \text{Img}(\vec{f}, 1)$ est NP-difficile. \square

4.2 Calcul direct de $\text{Img}(\vec{f}, \chi)$

Soit $\vec{f} = [f_1 \dots f_n]$ une fonction vectorielle de $\{0, 1\}^m$ vers $\{0, 1\}^n$, et χ une fonction booléenne sur $\{0, 1\}^m$. La fonction $\text{Img}(\vec{f}, \chi)$ est définie par l'équation [43] :

$$\text{Img}(\vec{f}, \chi) = \lambda \vec{y}. (\exists \vec{x} \chi(\vec{x}) \wedge \left(\bigwedge_{k=1}^n y_k \Leftrightarrow f_k(\vec{x}) \right)) \quad (4.1)$$

Une première façon d'obtenir $\text{Img}(\vec{f}, \chi)$ consiste à utiliser les opérateurs de combinaisons et d'élimination de quantificateurs définis dans les Chapitres 1 et 2. On construit d'abord le DAG de $(\chi(\vec{x}) \wedge (\bigwedge_{k=1}^n y_k \Leftrightarrow f_k(\vec{x})))$, puis on élimine le vecteur de variables \vec{x} quantifiées existentiellement. La première étape est en $O(|\chi| \times 2^n \times \prod_{k=1}^n |f_k|)$, et l'élimination des variables \vec{x} est au moins exponentielle dans le pire cas, soit deux étapes non polynomiales. Cependant, l'expérience montre que l'élimination de variables quantifiées est très souvent réalisable, et le coût global du calcul dépend donc essentiellement de la taille du DAG de $(\chi(\vec{x}) \wedge (\bigwedge_{k=1}^n y_k \Leftrightarrow f_k(\vec{x})))$. Si ce DAG peut être construit, alors le calcul de $\text{Img}(\vec{f}, \chi)$ par cette méthode est très efficace [43].

Dans le cas du parcours virtuel d'une machine, le terme $(\bigwedge_{k=1}^n (y_k \Leftrightarrow f_k(\vec{x})))$ représente, à une élimination des variables d'entrée près, la relation de transition de la machine.

Or le DAG de cette relation ne peut pas toujours être construit. Ceci vient du fait que les variables y_j n'ont pas d'occurrence libre dans les formules $f_k(\vec{x})$ ($1 \leq j \leq n$ et $1 \leq k \leq n$), et donc que la construction du produit $(\bigwedge_{k=1}^n (y_k \Leftrightarrow f_k(\vec{x})))$ requiert le calcul de la plupart des 2^n sous-produits $(\bigwedge_{k=1}^n \varepsilon_k(f_k))$. Le graphe G représentant le produit $(\bigwedge_{k=1}^n (y_k \Leftrightarrow f_k(\vec{x})))$ va croître de façon explosive, alors que la fonction $Img(\vec{f}, 1)$ obtenue après la \exists -élimination des variables x_1, \dots, x_m peut être de taille réduite.

Cette différence extrême de taille entre le graphe de $(\bigwedge_{k=1}^n (y_k \Leftrightarrow f_k(\vec{x})))$ et celui de $Img(\vec{f}, 1)$ est traduite par la Figure 22. Par exemple, si les variables y_1, \dots, y_n sont inférieures aux variables x_1, \dots, x_m , on risque d'obtenir un graphe G dont les 2^n premiers noeuds seront tous expansés. On distingue dans ce graphe 3 types de chemins : le chemin de type (a) atteint une feuille 1 sans rencontrer de variable x_k ; le chemin de type (b) atteint une feuille 0 sans rencontrer de variable x_k ; le chemin de type (c) rencontre au moins une variable x_k . Le graphe de $(\exists \vec{x} G)$ montre que les sous graphes de la partie à éliminer, correspondant aux chemins de type (c), contiennent beaucoup trop d'information : ils rassemblent toutes les interprétations de \vec{x} telles que $(\bigwedge_{k=1}^n y_k \Leftrightarrow f_k(\vec{x}))$ soit vrai pour un certain \vec{y} fixé, alors qu'on a juste besoin de savoir si $(\bigwedge_{k=1}^n y_k \Leftrightarrow f_k(\vec{x}))$ est satisfiable pour cet \vec{y} . Alors que l'information de satisfiabilité sur une forme canonique est de taille $O(1)$, il est d'abord demandé de calculer un graphe de relation sur \vec{x} , qui est potentiellement une information de taille $O(2^m/m)$! L'expérience montre [43] que pour $n > 30$, cette méthode ne peut être appliquée à des machines non triviales, ou à des machines dont la fonction de transition possède un graphe insuffisamment régulier [29].

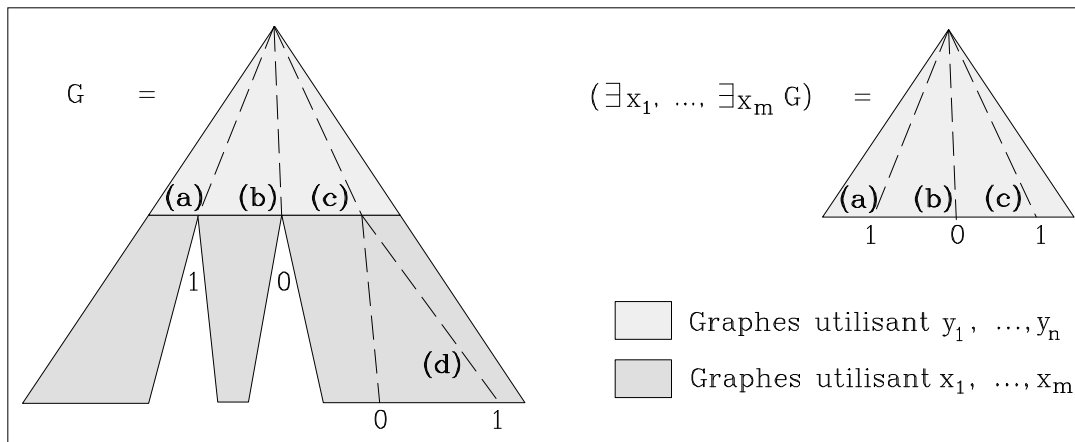


Figure 22. Graphes de G et de $(\exists \vec{x} G)$.

Dans [120], H. Touati *et al.* proposent une technique permettant de réduire le coût du calcul de l'image basé sur l'équation 4.1. L'idée est d'ordonner les variables de \vec{x} et \vec{y} , ainsi que les calculs des termes $(y_k \Leftrightarrow f_k(\vec{x}))$, de façon à éliminer les variables quantifiées aussi tôt que possible, et à garder un produit intermédiaire de taille raisonnable. Cette heuristique est efficace sur beaucoup d'exemples, mais si les composantes du vecteur \vec{f} ont des supports très semblables, alors l'ordonnement obtenu ne permet pas une réduction significative du coût du calcul.

4.3 Décomposition du calcul de l'image

Définition 4.1 Un restricteur d'image est un opérateur \mathbf{rr} tel que, pour toute fonction $\chi \neq 0$, et pour toute fonction vectorielle \vec{f} , on ait :

$$\text{Img}(\vec{f}, \chi) = \text{Img}(\vec{f} \mathbf{rr} \chi, 1).$$

La Figure 24 illustre l'utilisation d'un restricteur d'image. En utilisant un restricteur d'image \mathbf{rr} , le calcul de $\text{Img}(\vec{f}, \chi)$ revient au calcul de $\text{Img}(\vec{f} \mathbf{rr} \chi, 1)$. Nous appellerons une *couverture* un ensemble de fonctions $(h_j)_{j \in J}$ tel que $\lambda \vec{x}.(\bigvee_{j \in J} h_j(\vec{x})) = 1$. Alors pour toute couverture $(h_j)_{j \in J}$, on a l'équation de décomposition :

$$\text{Img}(\vec{f}, 1) = \lambda \vec{y}. \left(\bigvee_{j \in J} \text{Img}(\vec{f} \mathbf{rr} h_j, 1)(\vec{y}) \right).$$

Nous appelons “vectochar” (*vector to characteristic function*) la fonction qui calcule $\text{Img}(\vec{f}, 1)$. Le squelette de cet algorithme est donné ci-dessous.

```

function vectochar( $\vec{f}$ ) : TDG;
if  $\vec{f} = [\varepsilon_1(1) \dots \varepsilon_n(1)]$  then return  $(\bigwedge_{k=1}^n \varepsilon_k(y_k))$ ;
let  $(h_j)_{j \in J}$  such that  $(\bigvee_{j \in J} h_j) = 1$  in
  return  $(\bigvee_{j \in J} \text{vectochar}(\vec{f} \mathbf{rr} h_j))$ ;

```

Figure 23. Squelette de la fonction “vectochar”.

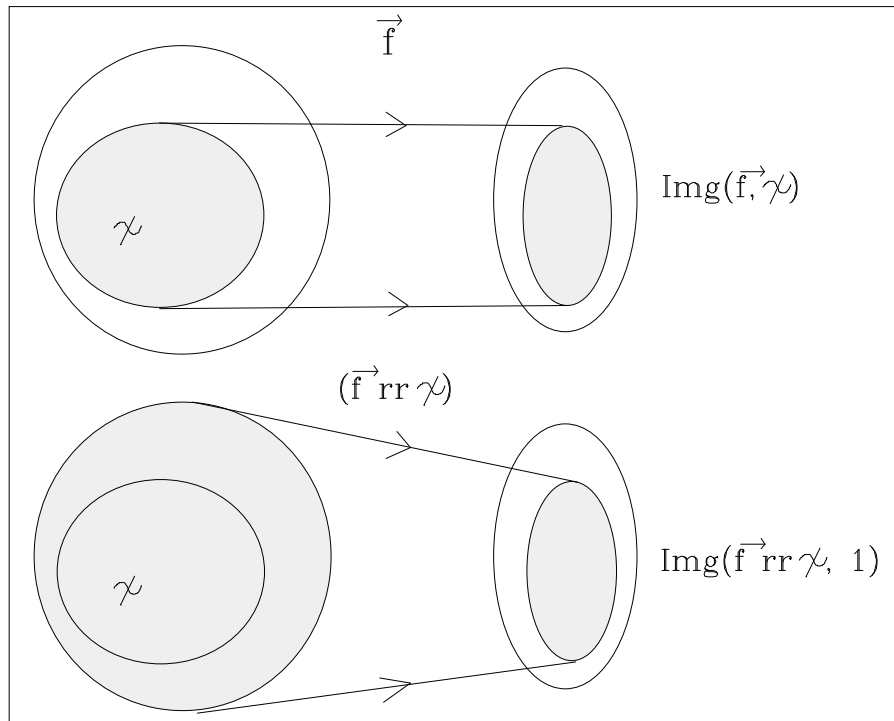


Figure 24. Restricteur d'image.

La complexité de cet algorithme dépend du choix de la couverture $(h_j)_{j \in J}$. Si à chaque étape la couverture $(h_j)_{j \in J}$ contient au moins deux ensembles h_1 et h_2 tels que $h_1 \not\subseteq h_2$ et $h_2 \not\subseteq h_1$, l'algorithme se termine. Si à chaque étape la couverture $(h_j)_{j \in J}$ est une partition, le nombre de récursions est borné par le nombre d'éléments du domaine, c'est à dire 2^m . En utilisant un cache qui conserve les calculs intermédiaires [38], ce nombre de récursions peut être radicalement réduit. Ce cache conserve les couples (\vec{f}, χ) tels que $\chi = \text{Img}(\vec{f}, 1)$. Au lieu de prendre en compte le vecteur \vec{f} en entier, cet algorithme de cache peut être raffiné en utilisant les propriétés suivantes [47] :

Théorème 4.5 *La fonction caractéristique de $\text{Img}([f_1 \dots f_{k-1} \varepsilon(1) f_{k+1} \dots f_n], \chi)$ est*

$$\lambda \vec{y}.(\varepsilon(y_k) \wedge \chi'(y_1, \dots, y_{k-1}, y_{k+1}, \dots, y_n)),$$

où $\chi' = \text{Img}([f_1 \dots f_{k-1} f_{k+1} \dots f_n], \chi)$.

Théorème 4.6 *La fonction caractéristique de $\text{Img}([f_1 \dots f_{k-1} \varepsilon(f_j) f_{k+1} \dots f_n], \chi)$ est*

$$\lambda \vec{y}.((y_j \Leftrightarrow \varepsilon(y_k)) \wedge \chi'(y_1, \dots, y_{k-1}, y_{k+1}, \dots, y_n)),$$

où $\chi' = \text{Img}([f_1 \dots f_{k-1} f_{k+1} \dots f_n], \chi)$.

Théorème 4.7 *Soit χ'_1 et χ'_2 les fonctions caractéristiques de $\text{Img}(\vec{f}_1, \chi_1)$ et $\text{Img}(\vec{f}_2, \chi_2)$. Alors la fonction caractéristique de $\text{Img}(\lambda(\vec{x}_1 @ \vec{x}_2).(\vec{f}_1(\vec{x}_1) @ \vec{f}_2(\vec{x}_2)), \chi_1 \times \chi_2)$ est :*

$$\lambda(\vec{y}_1 @ \vec{y}_2).(\chi'_1(\vec{y}_1) \wedge \chi'_2(\vec{y}_2)).$$

Le Théorème 4.5 permet d'éliminer les composantes constantes de \vec{f} . Le Théorème 4.6 affirme que si plusieurs composantes de \vec{f} sont égales ou complémentaires, toutes ces composantes sauf une peuvent être éliminées. Ces deux processus d'élimination réduisent le nombre d'entrées dans le cache, et augmente les possibilités de reconnaissance, d'où réduction du nombre de recursions. Le Théorème 4.7 signifie que si on peut trouver une partition $P = \{\vec{f}_1, \dots, \vec{f}_q\}$ des composantes de \vec{f} , telle que les vecteurs \vec{f}_k aient des supports de variables disjoints, alors calculer $\text{Img}(\vec{f}, 1)$ revient à calculer les q fonctions caractéristiques des ensembles $\text{Img}(\vec{f}_k, 1)$, puis à construire le produit cartésien (i.e. conjonction avec renommage des variables) de ces fonctions caractéristiques [44]. De cette façon, le nombre maximum de récursions est réduit de 2^m à $(\sum_{k=1}^q 2^{\text{sup}_k})$, où sup_k est la taille du support de \vec{f}_k .

Soit "eliminate-and-partition" la fonction qui, appliquée à un vecteur \vec{f} , retourne un couple (c, P) , tel que : c est la fonction caractéristique relative aux composantes éliminées de \vec{f} (par convention, $c = 1$ si aucune composante de \vec{f} ne peut être éliminée), et P est la partition des composantes restantes de \vec{f} en vecteurs de supports disjoints. Notons que le processus d'élimination utilisant les Théorèmes 4.5 et 4.6 peut être effectué en $O(n \log n)$ avec les TDGs, mais il est plus coûteux avec les BDDs (voir la discussion sur le test d'identification étendu page suivante). Le partitionnement des composantes restantes

```

function vectochar( $\vec{f}$ ) : TDG;
var  $c'$  : TDG;
let ( $c, P$ ) = eliminate-and-partition( $\vec{f}$ ) in {
  case  $P$  of {
    {} :  $c' = 1$ ;
    { $\vec{f}_1, \dots, \vec{f}_q$ } :  $c' = \lambda\vec{y} \cdot (\bigwedge_{k=1}^q \text{vectochar-cache}(\vec{f}_k)(\vec{y}))$ ;
  }
}
return  $\lambda\vec{y} \cdot (c(\vec{y}) \wedge c'(\vec{y}))$ ;

function vectochar-cache( $\vec{f}$ );
if  $\vec{f} = []$  or  $\vec{f} = [-]$  then return 1;
if is-in-cache?( $\vec{f}$ ) then return get-in-cache( $\vec{f}$ );
let  $c = \text{vectochar-recurse}(\vec{f})$  in {
  put-in-cache( $\vec{f}, c$ );
  return  $c$ ;
}

function vectochar-recurse( $\vec{f}$ );
let ( $h_j$ ) $_{j \in J}$  such that  $\lambda\vec{x} \cdot (\bigvee_{j \in J} h_j(\vec{x})) = 1$  in
  return  $\lambda\vec{y} \cdot (\bigvee_{j \in J} \text{vectochar}(\vec{f} \text{ rr } h_j)(\vec{y}))$ ;

```

Figure 25. Fonction “vectochar” utilisant un cache.

peut être effectué en $O((m \log m) \times (n + |\vec{f}|))$. La Figure 25 montre alors la nouvelle fonction “vectochar”.

Au lieu d'utiliser une identification exacte des vecteurs de fonctions, telle que celle proposée dans [38], nous utilisons une identification étendue [47] basée sur les deux théorèmes suivants.

Théorème 4.8 *Si $\mathcal{X}' = \text{Img}([f_1 \dots f_k], \mathcal{X})$, alors on a l'identité suivante, où ε_k est la fonction identité ou négation :*

$$\text{Img}([\varepsilon_1 \circ f_1 \dots \varepsilon_k \circ f_k], \mathcal{X}) = \lambda\vec{y} \cdot \mathcal{X}'(\varepsilon_1(y_1), \dots, \varepsilon_k(y_k)).$$

Théorème 4.9 *Si $\mathcal{X}' = \text{Img}([f_1 \dots f_k], \mathcal{X})$, alors on a l'identité suivante, où π est une permutation des k premiers entiers :*

$$\text{Img}([f_{\pi(1)} \dots f_{\pi(k)}], \mathcal{X}) = \lambda\vec{y} \cdot \mathcal{X}'(y_{\pi(1)}, \dots, y_{\pi(k)}).$$

Ceci signifie qu'un couple $([f_1 \dots f_k], \chi)$ dans le cache dénote l'image de $(k! \times 2^k)$ vecteurs de longueur k . La complexité du test d'identification étendue dépend directement de la représentation des fonctions booléennes. En utilisant les BDDs, tester si une fonction f est la négation de g est en $O(\max(|f|, |g|))$. Donc pour tout ordre total " \preceq " sur les BDDs, si la fonction $\lambda f. \lambda g. ((f \preceq g) \wedge (g \preceq f))$ et la fonction $\lambda f. \lambda g. ((f = g) \vee (f = \neg g))$ sont égales, alors $\lambda f. \lambda g. (f \preceq g)$ peut être évaluée au mieux en $O(\max(|f|, |g|))$. Ceci implique que les propriétés traduites par les Théorèmes 4.8 et 4.9 sont relativement coûteuses à tester avec les BDDs : le test d'identité étendue entre deux vecteurs \vec{f} et \vec{g} de dimension k est en $O(k \log k \times (|\vec{f}| + |\vec{g}|))$.

Ceci est entièrement différent avec les TDGs. Tester si une fonction f est la négation de g est en $O(1)$. Les composantes d'un vecteur $[f_1 \dots f_k]$ sont ordonnées selon l'ordre total " \preceq " défini par : $(f \preceq g)$ si et seulement si $\text{add}(f) \leq \text{add}(g)$, où $\text{add}(f)$ est l'adresse en mémoire du graphe de f si f est positive, et $\text{add}(f)$ est l'adresse du graphe de $\neg f$ si f est négative. Notons que cet ordonnancement fait que le processus de réécriture d'un vecteur \vec{f} en un couple (c, P) est canonique. Cet ordonnancement, qui peut être directement fait dans la fonction "eliminate-and-partition", s'effectue en $O(k \log k)$. Une fois le vecteur ordonné, les deux propriétés 4.8 et 4.9 sont testées en $O(k)$, ce qui fait que le test d'identité étendue est effectué en $O(k \log k)$. Ce test sur les TDGs ne dépend que de la longueur du vecteur, et pas de la taille des graphes.

Nous devons maintenant étudier le choix d'un restricteur d'image qui puisse décomposer efficacement le calcul de l'image, ce qui nous conduira à introduire le restricteur d'image "constrain".

4.4 Restricteur d'image

Comme le coût des manipulations des DAGs est directement lié à leur taille, le meilleur restricteur d'image **rr** est celui qui minimise la taille des DAGs de $(\vec{f} \text{ rr } \chi)$. Mais le théorème suivant indique que le restricteur d'image **rr** qui minimise la taille du graphe de $(\vec{f} \text{ rr } \chi)$ est beaucoup trop coûteux.

Théorème 4.10 *Etant donnés les arbres de Shannon réduits (ou les DAGs) d'un vecteur \vec{f} , trouver un vecteur \vec{f}' satisfaisant $\text{Img}(\vec{f}', 1) = \text{Img}(\vec{f}, 1)$ et ayant des arbres de Shannon minimaux (ou des DAGs minimaux) est NP-difficile.*

Preuve. Nous ne donnons la preuve que sur les arbres de Shannon réduits, la preuve pour les DAGs étant analogue. D'une part, tout vecteur \vec{f}' de n composantes tel que $\text{Img}(\vec{f}', 1) = 1$ doit disposer de n variables indépendantes, donc $|\vec{f}'| \geq n$. D'autre part, la fonction vectorielle identité Id est de taille n , et $\text{Img}(Id, 1) = 1$. Donc tout vecteur \vec{f}' de n composantes tel que $\text{Img}(\vec{f}', 1) = 1$ et qui minimise la forme de Shannon est de taille n . Alors on voit que tout vecteur \vec{f}' de n composantes tel que $\text{Img}(\vec{f}', 1) = 1$ et qui minimise la forme de Shannon est égal à l'une des $n! \times 2n$ fonctions $\lambda \vec{x}. [\varepsilon_1(x_{\pi(1)}) \dots \varepsilon_n(x_{\pi(n)})]$, où π est une permutation des n premiers entiers. Alors tester si $\text{Img}(\vec{f}, 1) = 1$ revient d'abord à calculer le vecteur minimal \vec{f}' tel que $\text{Img}(\vec{f}', 1) = \text{Img}(\vec{f}, 1)$, puis vérifier que \vec{f}' est d'une des formes décrites ci-dessus, ce qui peut être fait en $O(n^2)$. Une preuve similaire à celle

du Théorème 4.3 montre facilement que tester si $Img(\vec{f}, 1) = 1$, où la forme de Shannon de \vec{f} est donnée, est NP-difficile. \square

Le problème est alors de définir un restricteur d'image qui puisse être évalué polynomialement. Un restricteur d'image polynomial sur la structure récursive des formules propositionnelles (arbres syntaxiques, formes de Shannon, DAGs) doit pouvoir être appliqué récursivement sur ces structures. Comme ces structures peuvent être exprimées à partir de la négation et de la disjonction, notre problème peut se traduire par :

- (1) Déterminer les restricteurs d'image \mathbf{rr} tels que, pour toutes fonctions vectorielles \vec{f} et \vec{g} , pour toute fonction χ , on ait :

$$\begin{aligned} ((\neg \vec{f}) \mathbf{rr} \chi) &= \neg(\vec{f} \mathbf{rr} \chi), \text{ et} \\ ((\vec{f} \vee \vec{g}) \mathbf{rr} \chi) &= ((\vec{f} \mathbf{rr} \chi) \vee (\vec{g} \mathbf{rr} \chi)) \end{aligned}$$

Etant donné que toute composition s'exprime à travers la décomposition de Shannon par des négations et des disjonctions (propriété d'orthogonalité), ce problème peut se reformuler ainsi :

- (2) Déterminer les restricteurs d'image \mathbf{rr} tels que, pour toutes fonctions vectorielles \vec{f} et \vec{g} , pour toute fonction χ , on ait :

$$((\vec{f} \circ \vec{g}) \mathbf{rr} \chi) = (\vec{f} \circ (\vec{g} \mathbf{rr} \chi))$$

Comme toute fonction f s'écrit $\vec{f} \circ Id$, où Id est la fonction identité, un restricteur d'image satisfaisant (2) est tel que $(\vec{f} \mathbf{rr} \chi) = (\vec{f} \circ (Id \mathbf{rr} \chi))$ pour toute fonction χ . Aussi un tel restricteur d'image s'écrit $\lambda \vec{f}. \lambda \chi. \lambda \vec{x}. (\vec{f}(Fun(\chi, \vec{x})))$, où $Fun(\chi, \vec{x}) = (Id \mathbf{rr} \chi)(\vec{x})$ ne dépend pas de \vec{f} . D'après la définition des restricteurs d'image, Fun est telle que pour toute fonction $\chi \neq 0$, on a $Img(\lambda \vec{x}. Fun(\chi, \vec{x}), 1) = \chi$.

Nous nous inspirons de cette propriété de Fun pour justifier l'utilisation des projecteurs. Un *projecteur* sur une fonction $\chi \neq 0$ est une fonction qui envoie le domaine sur χ . En d'autres termes, un projecteur P sur $\chi \neq 0$ est une fonction de $(\{0, 1\}^m \rightarrow \{0, 1\}) \times \{0, 1\}^m$ dans $\{0, 1\}^m$ telle que $Img(\lambda \vec{x}. P(\chi, \vec{x}), 1) = \chi$. Alors pour tout projecteur P , la fonction $\lambda \vec{f}. \lambda \chi. \lambda \vec{x}. (\vec{f}(P(\chi, \vec{x})))$ est un restricteur d'image satisfaisant (2). Tout revient donc à choisir un projecteur.

On distingue les projecteurs *stricts*, qui sont des projecteurs égaux à l'identité sur χ , c'est à dire que $\chi(\vec{x}) = 1$ implique $P(\chi, \vec{x}) = \vec{x}$. On appellera *restricteur d'image strict* un restricteur d'image engendré par un projecteur strict². La figure 26 montre la transformation qu'effectue un restricteur d'image strict \mathbf{srr} sur une fonction \vec{f} . Un restricteur

²Un restricteur d'image strict est une extension du concept d'évaluation et de cofacteur. En effet, pour toute fonction h , on a :

$$h(\vec{v}) = (\lambda \vec{y}. h(\vec{y} \mathbf{srr} \left(\bigwedge_{k=1}^n \varepsilon_k(y_k) \right))),$$

où ε_k est la négation si $v_k = 0$ et ε_k est l'identité si $v_k = 1$. De plus, pour toute fonction χ non constante, on a $h = (\neg \chi \wedge (h \mathbf{srr} \neg \chi)) \vee (\chi \wedge (h \mathbf{srr} \chi))$, analogue à l'expansion de Shannon.

Figure 26. Restricteur d'image strict.

d'image strict **srr** est tel que $\chi(\vec{x}) = 1$ implique $(\vec{f} \mathbf{srr} \chi)(\vec{x}) = \vec{f}(\vec{x})$. Autrement dit, les comportements de \vec{f} et $(\vec{f} \mathbf{srr} \chi)$ sur χ sont identiques. En particulier, $(\vec{f} \mathbf{srr} 1) = \vec{f}$.

Théorème 4.11 *Pour tout $\chi \neq 0$, et tout restricteur d'image strict **srr**, on a :*

$$\text{si } (\chi \Rightarrow f) \quad \text{alors } (f \mathbf{srr} \chi) = 1 \quad (4.2)$$

$$\text{si } (\chi \Rightarrow \neg f) \quad \text{alors } (f \mathbf{srr} \chi) = 0 \quad (4.3)$$

$$(f \mathbf{srr} 1) = f \quad (4.4)$$

$$(\chi \mathbf{srr} \chi) = 1 \quad (4.5)$$

$$((\neg\chi) \mathbf{srr} \chi) = 0 \quad (4.6)$$

$$(1 \mathbf{srr} \chi) = 1 \quad (4.7)$$

$$(0 \mathbf{srr} \chi) = 0 \quad (4.8)$$

Le comportement d'un restricteur d'image strict est entièrement caractérisé par son projecteur. Par exemple, nous pouvons prendre comme projecteur P la fonction $\lambda\chi.\lambda\vec{x}.((\chi(\vec{x}) \wedge \vec{x}) \vee (\neg\chi(\vec{x}) \wedge \vec{x}_0))$, où \vec{x}_0 est un élément de $\{0, 1\}^m$ qui satisfait χ [44]. Un restricteur d'image strict **srr** basé sur ce projecteur est :

$$\mathbf{srr} = \lambda\vec{f}.\lambda\chi.\lambda\vec{x}.((\chi(\vec{x}) \wedge \vec{f}(\vec{x})) \vee (\neg\chi(\vec{x}) \wedge \vec{f}(\vec{x}_0)))$$

Donc étant donnée une fonction vectorielle \vec{f} , calculer les DAGs de $(\vec{f} \mathbf{srr} \chi)$ consiste à : trouver un élément \vec{x}_0 du domaine $\{0, 1\}^m$ tel que $\chi(\vec{x}_0) = 1$, ce qui est en $O(|\chi|)$; puis calculer l'élément $[v_1 \dots v_n]$ du codomaine $\{0, 1\}^n$ égal à $\vec{f}(\vec{x}_0)$, ce qui est en $O(n \times m)$;

enfin construire les n DAGs $(\chi(\vec{x}) \wedge f_k(\vec{x})) \vee (\neg\chi(\vec{x}) \wedge v_k)$, ce qui est en $O(|\mathcal{X}| \times \sum_{k=1}^n |f_k|)$. Finalement, calculer les DAGs de $(\vec{f} \text{ srr } \chi)$ est en $O(|\mathcal{X}| \times \sum_{k=1}^n |f_k|)$, ce qui est très satisfaisant (cette complexité est similaire à celle de n'importe quel opérateur booléen usuel). Mais nous voulons un restricteur d'image strict **srr** tel que la taille des DAGs de $(\vec{f} \text{ srr } \chi)$ demeurent relativement petite comparée à celle des DAGs de \vec{f} et χ .

Un projecteur P qui minimise la taille des graphes de $(\vec{f} \circ \lambda\vec{x}.P(\chi, \vec{x}))$ n'est certainement pas polynomial. Nous allons donc étudier le problème sur les arbres de Shannon. Bien sûr, la seule relation entre la taille de l'arbre de Shannon et la taille de son graphe associé est une inégalité. Ceci signifie que minimiser les arbres de Shannon de $(\vec{f} \circ \lambda\vec{x}.P(\chi, \vec{x}))$ ne donne pas une solution pour minimiser les graphes de $(\vec{f} \circ \lambda\vec{x}.P(\chi, \vec{x}))$, mais cela peut constituer une assez bonne approximation en pratique. Nous présentons dans la suite un restricteur d'image strict appelé "constrain", qui est généré par un projecteur bien adapté.

4.5 L'opérateur "Constrain"

Afin de présenter le restricteur d'image strict "constrain", nous définissons d'abord le projecteur *PPI* (pour *Plus Proche Interpretation*). Cette fonction est définie sur l'ensemble des interprétations vis-à-vis d'une fonction booléenne $\chi \neq 0$. Nous donnons ici deux présentations de *PPI*. La première, opérationnelle (elle inspire directement l'algorithme de l'opérateur "constrain"), est basée sur les définitions de satisfiabilité et de validité. La seconde, plus intuitive, utilise la définition d'une hyper-distance sur les interprétations.

4.5.1 Présentation logique de la Plus Proche Interpretation

Puisque χ est une formule de longueur finie, son support est fini. Comme les variables propositionnelles $\{x_1, x_2, \dots\}$ sont ordonnées, il existe un indice m tel que ce support soit inclus dans x_1, \dots, x_m . Pour la première présentation, nous démontrons d'abord le Lemme suivant :

Lemme 4.1 *Soit $\chi \neq 0$, et $[x_1 \dots x_m] \in \{0, 1\}^m$ tel que $\chi(x_1, \dots, x_m) = 0$. Alors il existe une unique indice k ($1 \leq k \leq m$), tel que les formules suivantes soient valides :*

$$\begin{aligned} &(\forall y_{k+1} \dots \forall y_m \neg\chi(x_1, \dots, x_k, y_{k+1}, \dots, y_m)), \quad \text{et} \\ &(\exists y_{k+1} \dots \exists y_m \chi(x_1, \dots, x_{k-1}, \neg x_k, y_{k+1}, \dots, y_m)). \end{aligned}$$

Preuve. Supposons qu'un tel indice k n'existe pas. Alors pour $1 \leq j \leq m$, la formule

$$\begin{aligned} \wp_j = &(\exists y_{j+1} \dots \exists y_m \chi(x_1, \dots, x_j, y_{j+1}, \dots, y_m)) \vee \\ &(\forall y_{j+1} \dots \forall y_m \neg\chi(x_1, \dots, x_{j-1}, \neg x_j, y_{j+1}, \dots, y_m)) \end{aligned}$$

est valide. Pour $j = m$, ceci signifie que $(\chi(x_1, \dots, x_m) \vee \neg\chi(x_1, \dots, x_{m-1}, \neg x_m))$ est valide. Comme par hypothèse $\chi(x_1, \dots, x_m) = 0$, la formule $\chi(x_1, \dots, x_{m-1}, \neg x_m)$ doit être égale à 0. On en déduit que $(\forall y_m \neg\chi(x_1, \dots, x_{m-1}, y_m))$ est valide. En combinant ce résultat

avec la validité de (\wp_{j-1}) , on obtient que $(\forall y_m \neg \chi(x_1, \dots, x_{m-2}, \neg x_{m-1}, y_m))$ est valide. On en déduit que la formule $(\forall y_{m-1} \forall y_m \neg \chi(x_1, \dots, x_{m-2}, y_{m-1}, y_m))$ est valide.

En poursuivant ce raisonnement pour j décroissant, on montre que la formule $(\forall y_1 \dots \forall y_m \neg \chi(y_1, \dots, y_m))$ est valide, c'est à dire que $\chi = 0$, ce qui est contradictoire. Donc un tel indice k existe.

Supposons maintenant qu'il existe deux indices k et k' satisfaisant la propriété donnée. La formule $(\forall y_{k+1} \dots \forall y_m \neg \chi(x_1, \dots, x_k, y_{k+1}, \dots, y_m))$ est valide par définition de k , ainsi que la formule $(\exists y_{k'+1} \dots \exists y_m \chi(x_1, \dots, x_{k'-1}, \neg x_{k'}, y_{k'+1}, \dots, y_m))$ par définition de k' , ce qui implique que $(\exists y_{k'} \dots \exists y_m \chi(x_1, \dots, x_{k'-1}, y_{k'}, \dots, y_m))$ est valide. Donc on doit avoir $k' < k + 1$. De la même façon, on obtient $k < k' + 1$, d'où $k = k'$. \square

A partir de ce lemme, nous définissons la suite de points $(\vec{x}^n)_{n \geq 0}$, associée à un point $\vec{x} = (x_1, x_2, \dots)$ de $\{0, 1\}^{\mathbb{N}}$ et une fonction $\chi \neq 0$, par :

- $\vec{x}^0 = \vec{x}$
- si \vec{x}^n satisfait χ , alors $\vec{x}^{n+1} = \vec{x}^n$
- si $\vec{x}^n = (x_1^n, x_2^n, \dots)$ ne satisfait pas χ , alors comme $\chi \neq 0$, le Lemme 4.1 nous assure qu'il existe un unique indice k tel que

$$\begin{aligned} & (\forall y_{k+1} \dots \forall y_m \neg \chi(x_1^n, \dots, x_k^n, y_{k+1}, \dots, y_m)), \quad \text{et} \\ & (\exists y_{k+1} \dots \exists y_m \chi(x_1^n, \dots, x_{k-1}^n, \neg x_k^n, y_{k+1}, \dots, y_m)). \end{aligned}$$

On pose alors : $\vec{x}_{n+1} = (x_1^n, x_2^n, \dots, x_{k-1}^n, \neg x_k^n, x_{k+1}^n, \dots)$.

Théorème 4.12 *Etant donnée une formule finie $\chi \neq 0$ et un point \vec{x} de $\{0, 1\}^{\mathbb{N}}$, la suite $(\vec{x}^n)_{n \geq m}$ associée à \vec{x} est constante, et est telle que $\chi(\vec{x}^m) = 1$.*

Preuve. Si il existe un indice n tel que $n < m$ et que \vec{x}^n satisfait χ , alors par définition la suite de points devient constante et \vec{x}^m satisfait χ . Supposons maintenant que pour les indices $n, n < m$, les points \vec{x}^n ne satisfont pas χ . Quand \vec{x}^n ne satisfait pas χ , \vec{x}^{n+1} est obtenue en inversant la k -ième composante de \vec{x}^n , où l'indice k est défini de façon unique par le Lemme 4.1. Soit $\vec{x}^n = (x_1^n, x_2^n, \dots)$ et $\vec{x}^{n+1} = (x_1^n, x_2^n, \dots, x_{k-1}^n, \neg x_k^n, x_{k+1}^n, \dots)$. Si \vec{x}^{n+1} ne satisfait pas χ , on doit trouver l'indice k' permettant de calculer \vec{x}^{n+2} . Par définition de k , la formule

$$(\exists y_{k+1} \dots \exists y_m \chi(x_1^n, \dots, x_{k-1}^n, \neg x_k^n, y_{k+1}, \dots, y_m))$$

est valide. Ceci implique que pour $k' < k$, la formule

$$(\forall y_{k'+1} \dots \forall y_m \neg \chi(x_1^n, \dots, x_{k'}^n, y_{k'+1}, \dots, y_m))$$

est une antilogie, ainsi que la formule

$$(\forall y_{k'+1} \dots \forall y_m \neg \chi(x_1^n, \dots, x_{k-1}^n, \neg x_k^n, y_{k'+1}, \dots, y_m))$$

pour $k' = k$. Comme k' est tel que $(\forall y_{k'+1} \dots \forall y_m \neg \chi(x_1^n, \dots, y_{k'+1}, \dots, y_m))$, on doit avoir $k' > k$. Notons $Ind(\vec{x}^n)$ l'indice k qui est utilisé pour calculer \vec{x}^{n+1} à partir de \vec{x}^n . On a $Ind(\vec{x}^n) \leq m$, et nous venons de montrer que $Ind(\vec{x}^{n+1}) > Ind(\vec{x}^n)$, c'est à dire

$Ind(\vec{x}^{n+1}) \geq Ind(\vec{x}^n) + 1$. Comme $Ind(\vec{x}^0) > 0$, on obtient $Ind(\vec{x}^n) > n$. Il est évident que si $Ind(\vec{x}^n) = m$, alors \vec{x}^{n+1} satisfait χ . Comme on a $Ind(\vec{x}^{m-1}) \geq m$, \vec{x}^m doit satisfaire χ , et donc la séquence de points $(\vec{x}^n)_{n \geq m}$ devient constante. \square

Le théorème précédent nous permet de définir la fonction *PPI* de façon opérationnelle.

Définition 4.2 Soit χ une fonction non nulle de $\{0, 1\}^m$ dans $\{0, 1\}$. Soit \vec{x} un point de $\{0, 1\}^m$, et soit $(\vec{x}^n)_{n \geq 0}$ la séquence associée à \vec{x} et χ . On pose :

$$PPI(\chi, \vec{x}) = \vec{x}^m. \quad (4.9)$$

4.5.2 Définition topologique de la Plus Proche Interpretation

La définition de la série d'interprétations $(\vec{x}^n)_{0 \leq n \leq m}$ et la preuve du Lemme 4.1 suggèrent une autre présentation de la *Plus Proche Interpretation*, sans doute plus intuitive, car utilisant la topologie de l'espace des interprétations. Nous définissons la fonction d sur $\{0, 1\}^{\mathbb{N}} \times \{0, 1\}^{\mathbb{N}}$ par :

$$d(\vec{x}, \vec{y}) = \sum_{k=1}^{\infty} \frac{|x_k - y_k|}{k2^k}.$$

Lemme 4.2 La fonction d est une hyper-distance³, c'est à dire une distance telle que pour toutes interprétations \vec{x}, \vec{y} et \vec{z} , $d(\vec{x}, \vec{z}) = d(\vec{y}, \vec{z})$ si et seulement si $\vec{x} = \vec{y}$.

Preuve. d est une distance : la fonction d est positive et symétrique ; $d(\vec{x}, \vec{y})$ est nul si et seulement si $x_k = y_k$ pour tout k , c'est à dire si et seulement si $\vec{x} = \vec{y}$; comme on a :

$$\begin{aligned} |x_k - y_k| &= |(x_k - z_k) - (y_k - z_k)| \\ &\leq |x_k - z_k| + |y_k - z_k|, \end{aligned}$$

on obtient $d(\vec{x}, \vec{y}) \leq d(\vec{x}, \vec{z}) + d(\vec{z}, \vec{y})$. Montrons maintenant que d est une hyper-distance. Supposons que $\vec{x} \neq \vec{y}$. Soit donc n l'indice minimal k tel que $x_k \neq y_k$. On obtient les minoration suivantes :

$$\begin{aligned} |d(\vec{x}, \vec{z}) - d(\vec{y}, \vec{z})| &= \left| \sum_{k=1}^{\infty} \frac{(|x_k - z_k| - |y_k - z_k|)}{k2^k} \right| \\ &\geq \frac{1}{n2^n} - \left| \sum_{k=n+1}^{\infty} \frac{(|x_k - z_k| - |y_k - z_k|)}{k2^k} \right| \\ &\geq \frac{1}{n2^n} - \sum_{k=n+1}^{\infty} \frac{||x_k - z_k| - |y_k - z_k||}{k2^k} \end{aligned}$$

³La propriété d'hyper-distance est perdue si le facteur $1/k$ est ôté dans la définition de d . Ainsi, la fonction $d'(\vec{x}, \vec{y}) = \sum_{k=1}^{\infty} |x_k - y_k|/2^k$ est une distance, mais pas une hyper-distance. Par exemple, pour $\vec{x} = (111\dots)$, $\vec{y} = (000\dots)$ et $\vec{z} = (011\dots)$, on a $\vec{x} \neq \vec{y}$, mais $d'(\vec{x}, \vec{z}) = d'(\vec{y}, \vec{z}) = 1/2$. L'affectation d'un poids $1/k2^k$ à la k -ième composante de l'interprétation nous assure que "l'importance" des composantes d'indice supérieur ou égal à $k + 1$ est strictement inférieure à "l'importance" de la k -ième composante.

$$\begin{aligned}
&\geq \frac{1}{n2^n} - \sum_{k=n+1}^{\infty} \frac{1}{k2^k}, \quad \text{car } ||v_k - u_k| - |w_k - u_k|| \leq 1 \\
&> \frac{1}{n2^n} - \frac{1}{n+1} \left(\sum_{k=n+1}^{\infty} \frac{1}{2^k} \right) \\
&= \frac{1}{2^n} \left(\frac{1}{n} - \frac{1}{n+1} \right) \\
&> 0
\end{aligned}$$

d'où $d(\vec{x}, \vec{z}) \neq d(\vec{y}, \vec{z})$. Comme d est une distance, la réciproque est immédiate. \square

Lemme 4.3 Soit \vec{x} un point de $\{0, 1\}^{\aleph}$. La relation notée $\leq_{\vec{x}}$, et définie pour tous points \vec{y} et \vec{z} par $\vec{y} \leq_{\vec{x}} \vec{z}$ si et seulement si $d(\vec{x}, \vec{y}) \leq d(\vec{x}, \vec{z})$, est un ordre total sur $\{0, 1\}^{\aleph}$.

Preuve. Comme d est une distance bornée, on a $d(\vec{x}, \vec{y})$ et $d(\vec{x}, \vec{z})$ finis pour tous points \vec{y} et \vec{z} , donc on a $\vec{y} \leq_{\vec{x}} \vec{z}$ ou $\vec{z} \leq_{\vec{x}} \vec{y}$. La relation $\leq_{\vec{x}}$ est transitive puisque d est une distance. Si $\vec{y} \leq_{\vec{x}} \vec{z}$ et $\vec{z} \leq_{\vec{x}} \vec{y}$, alors ceci implique par définition que $d(\vec{x}, \vec{y}) = d(\vec{x}, \vec{z})$, ce qui implique que $\vec{y} = \vec{z}$ puisque d est une hyper-distance. \square

Lemme 4.4 Les formules propositionnelles finies sont continues⁴ vis-à-vis de l'hyper-distance d , c'est à dire que si une suite de points $(\vec{x}^n)_{n>0}$ converge au sens de d , alors pour toute formule f de taille finie, $\lim_{n \rightarrow \infty} f(\vec{x}^n) = f(\lim_{n \rightarrow \infty} \vec{x}^n)$.

Preuve. Soit $\vec{l} = \lim_{n \rightarrow \infty} \vec{x}^n$. Montrons que la suite $(f(\vec{x}^n))_{n>0}$ converge pour d , et que sa limite est $f(\vec{l})$. La formule f est de longueur finie, donc de support fini, et il existe m tel que ce support est inclus dans $\{x_1, \dots, x_m\}$. Comme $(\vec{x}^n)_{n>0}$ converge vers \vec{l} et que $d(\vec{x}^n, \vec{x}^{n+1}) \leq d(\vec{x}^n, \vec{l}) + d(\vec{l}, \vec{x}^{n+1})$, il existe un entier k tel que pour $n > k$, on ait $d(\vec{x}^n, \vec{x}^{n+1}) < 1/m2^m$. Ceci signifie que pour $n > k$, les m premières composantes de \vec{x}^n restent constantes, et que ces m premières composantes seront aussi celles de \vec{l} . Comme le support de f est inclus dans $\{x_1, \dots, x_m\}$, cela veut dire aussi que la suite $(f(\vec{x}^n))_{n>k}$ est constante, et égale à $f(\vec{l})$. \square

Théorème 4.13 Soit \vec{x} un point de $\{0, 1\}^{\aleph}$ et une formule $\chi \neq 0$. Il existe un unique point de $\{0, 1\}^{\aleph}$ qui satisfait χ et qui minimise la distance d avec \vec{x} .

Preuve. Soit $S = \{\vec{y} / \chi(\vec{y}) = 1\}$. L'ensemble S réunit les interprétations satisfaisant χ . Cet ensemble est non vide puisque $\chi \neq 0$. Vu le Lemme 4.3, la relation $\leq_{\vec{x}}$ est un ordre total sur S , et comme la distance d est finie, S possède une borne inférieure \vec{x}_{\min} vis-à-vis de $\leq_{\vec{x}}$. Par définition de la borne inférieure, il existe une suite d'interprétation $(\vec{x}^n)_{n>0}$ de S qui converge vers \vec{x}_{\min} au sens de d . D'après le Lemme 4.4, on a les égalités :

$$\begin{aligned}
\chi(\vec{x}_{\min}) &= \chi(\lim_{n \rightarrow \infty} \vec{x}^n) \\
&= \lim_{n \rightarrow \infty} \chi(\vec{x}^n).
\end{aligned}$$

⁴Cette continuité est perdue sur les formules de taille infinie. Par exemple, avec la formule infinie $f = (\bigvee_{k=1}^{\infty} x_k)$ et la suite d'interprétations $(\vec{x}^n)_{n>0}$ définie par $x_n^n = 1$ et $x_k^n = 0$ si $k \neq n$, on a $\lim_{n \rightarrow \infty} \vec{x}^n = (00\dots)$ au sens de d , et donc $f(\lim_{n \rightarrow \infty} \vec{x}^n) = 0$, alors que $f(\vec{x}^n)$ est constante et égale à 1. Ceci n'est pas étonnant, puisque l'ensemble $(\aleph \rightarrow \{0, 1\})$ a la puissance du continu, et donc que l'ensemble des fonctions caractéristiques de \aleph qui sont récursives est de mesure nulle sur $(\aleph \rightarrow \{0, 1\})$. Ceci revient aussi à dire que dans l'espace des formules infinies, l'ensemble des formules calculables (c'est à dire de sémantique évaluable) est aussi de mesure nulle.

Comme $(\vec{x}^n)_{n>0}$ est dans S , la suite $(\chi(\vec{x}^n))_{n>0}$ est constante et égale à 1, donc $\lim_{n \rightarrow \infty} \chi(\vec{x}^n) = 1$, et donc la borne inférieure x_{\min} satisfait χ . \square

On peut ainsi donner une deuxième définition de la *Plus Proche Interpretation*, et montrer facilement qu'elle est équivalente à celle donnée par la définition 4.2.

Définition 4.3 Soit une fonction $\chi \neq 0$, et une interprétation \vec{x} . L'unique interprétation notée $PPI(\chi, \vec{x})$ est la plus proche interprétation de \vec{x} , au sens de d , qui satisfait χ .

4.5.3 Définition et évaluation de l'opérateur "Constrain"

Définition 4.4 Nous définissons l'opérateur "constrain", noté " \downarrow ", comme :

$$\downarrow = \lambda \vec{f}. \lambda \chi. \lambda \vec{x}. \vec{f}(PPI(\chi, \vec{x})).$$

On montre facilement que PPI est un projecteur strict, donc que " \downarrow " est un restricteur d'image strict. L'algorithme ci-dessous calcule le graphe de $\lambda \vec{x}. \vec{f}(PPI(\chi, \vec{x}))$ à partir des graphes des deux fonctions f et χ [44]. Cet algorithme utilise la définition 4.2 de PPI , et il exploite les propriétés des restricteurs d'image stricts données par le Théorème 4.11. Si un cache est utilisé pour éviter les calculs redondants, alors sa complexité est en $O(|f| \times |\chi|)$. Etant donné un vecteur $\vec{f} = [f_1 \dots f_n]$, le calcul de $(\vec{f} \downarrow \chi)$ est effectué en construisant le vecteur $[(f_1 \downarrow \chi) \dots (f_n \downarrow \chi)]$, ce qui est fait en $O(|\vec{f}| \times |\chi|)$.

```

function constrain( $f, \chi$ ) : TDG;
if  $\chi = 0$  then error;
return cnst( $f, \chi$ );

function cnst( $f, \chi$ ) : TDG;
if  $\chi = 1$  or  $f = 0$  or  $f = 1$  then return  $f$ ;
if  $f = \chi$  then return 1;
if  $f = \neg \chi$  then return 0;
let  $x = \chi.root$  in {
  if  $\chi[x \leftarrow 0] = 0$  then return cnst( $f[x \leftarrow 1], \chi[x \leftarrow 1]$ );
  if  $\chi[x \leftarrow 1] = 0$  then return cnst( $f[x \leftarrow 0], \chi[x \leftarrow 0]$ );
  return ( $\neg x \wedge cnst(f[x \leftarrow 0], \chi[x \leftarrow 0])$ )  $\vee$  ( $x \wedge cnst(f[x \leftarrow 1], \chi[x \leftarrow 1])$ );
}

```

Figure 27. Fonction "constrain".

La Figure 28 montre un exemple d'application de l'opérateur "constrain" sur des arbres réduits de Shannon, et la Table 4 montre comment les valeurs de \vec{x} , $\vec{f}(\vec{x})$, et $PPI(\chi, \vec{x})$ sont implicitement combinées pour mener à $(\vec{f} \downarrow \chi)(\vec{x}) = \vec{f}(PPI(\chi, \vec{x}))$. On vérifie aussi que $Img(\vec{f}, \chi)$ et $Img(\vec{f} \downarrow \chi, 1)$ sont égaux à $\lambda \vec{y}. (\neg y_2 \wedge ((\neg y_1 \wedge (\vec{y}_3 \vee y_4)) \vee (y_1 \wedge \neg y_3 \wedge y_4)))$, qui dénote l'ensemble $\{[0001], [0010], [0011], [1001]\}$.

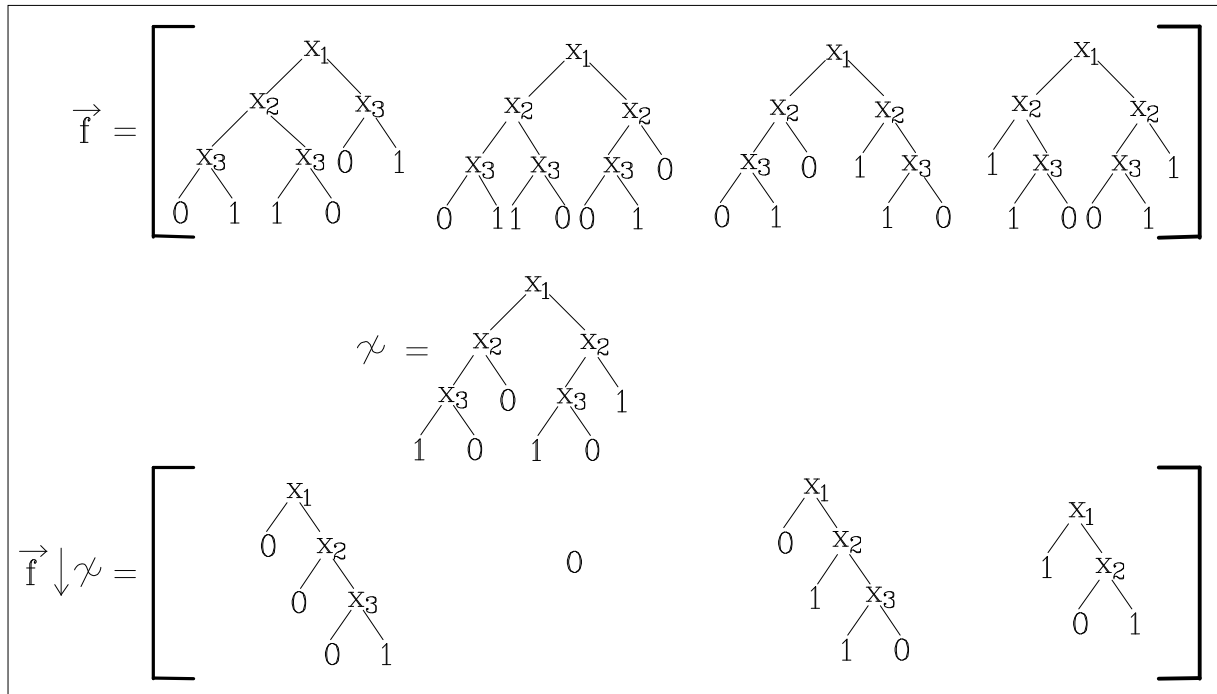


Figure 28. Exemple d'application de l'opérateur "constrain".

\vec{x}	$\vec{f}(\vec{x})$	$\chi(\vec{x})$	$PPI(\chi, \vec{x})$	$(\vec{f} \downarrow \chi)(\vec{x})$
[000]	[0001]	1	[000]	[0001]
[001]	[1111]	0	[000]	[0001]
[010]	[1101]	0	[000]	[0001]
[011]	[0000]	0	[000]	[0001]
[100]	[0010]	1	[100]	[0010]
[101]	[1111]	0	[100]	[0010]
[110]	[0011]	1	[110]	[0011]
[111]	[1001]	1	[111]	[1001]

Table 4. Comportement de $(\vec{f} \downarrow \chi)$.

4.6 Choix d'une couverture

4.6.1 Utilisation d'un partitionnement du co-domaine

Pour obtenir une partition $(h_j)_{j \in J}$ du domaine, on peut soit travailler directement sur le domaine, soit utiliser une partition du codomaine. Considérons une partition $(g_j)_{j \in J}$ du codomaine. La fonction $g_j \circ \vec{f}$ dénote les éléments du domaine tel que leur image par \vec{f} appartient à g_j . Donc $(g_j \circ \vec{f})_{j \in J}$ est une partition du domaine.

Mais la composition est un problème NP-difficile. Nous devons donc soigneusement définir $(g_j)_{j \in J}$ afin d'obtenir une composition de coût modéré. Choisissons [44]

la partition (g_0, g_1) telle que $g_0 = \lambda\vec{y}.(\neg y_k)$ et $g_1 = \lambda\vec{y}.y_k$. Ce couple sépare le codomaine en deux parties de même taille. On a $(g_0 \circ \vec{f}) = \neg f_k$, et $(g_1 \circ \vec{f}) = f_k$. Si **srr** est un restricteur d'image strict, alors le vecteur $(\vec{f} \text{ srr } \varepsilon(f_k))$ est égal à

$$([f_1 \dots f_{k-1}] \text{ srr } \varepsilon(f_k)) @ [\varepsilon(1)] @ ([f_{k+1} \dots f_n] \text{ srr } \varepsilon(f_k)).$$

Ainsi la fonction “vectochar-recurse” devient :

```

function vectochar-recurse( $\vec{f}$ ) : TDG;
let  $[f_1 \dots f_n] = \vec{f}$  and
       $k = \text{choose-index}(\vec{f})$  and
       $\vec{f}' = [f_1 \dots f_{k-1} \ f_{k+1} \dots f_n]$  in
return  $\lambda\vec{y}.(\neg y_k \wedge \text{vectochar}(\vec{f}' \text{ srr } \neg f_k)(y_1, \dots, y_{k-1}, y_{k+1}, \dots, y_n)) \vee$ 
       $(y_k \wedge \text{vectochar}(\vec{f}' \text{ srr } f_k)(y_1, \dots, y_{k-1}, y_{k+1}, \dots, y_n));$ 

```

Figure 29. Premier algorithme de calcul de *Img*.

Dans cet algorithme, la fonction “choose-index” retourne un index k . Par exemple, “choose-index” pourrait toujours rendre 1. Cette fonction peut plus intelligemment utiliser des heuristiques afin d’obtenir un index k , qui, dans la mesure du possible, permettra d’accélérer l’apparition de constantes dans $(\vec{f}' \text{ srr } \varepsilon(f_k))$. On peut par exemple choisir de retourner l’index de la composante qui possède le plus petit graphe, ou l’index de la composante ayant le moins de variables.

Grâce a cette fonction “vectochar-recurse”, le nombre de récursion de “vectochar” est borné par le nombre d’éléments de $\text{Img}(\vec{f}, 1)$. Donc si à chaque étape, la machine n’atteint qu’un relativement petit nombre d’états, cet algorithme est très efficace.

4.6.2 Utilisation d’un partitionnement du domaine

Choisissons [47] la partition (h_0, h_1) tel que $h_0 = \lambda\vec{x}.(\neg x_k)$ et $h_1 = \lambda\vec{x}.x_k$. Ce couple sépare le domaine en deux parties de tailles égales. Si **srr** est un restricteur d’image strict, nous avons $(\vec{f} \text{ srr } h_0) = \vec{f}[x_k \leftarrow 0]$ et $(\vec{f} \text{ srr } h_1) = \vec{f}[x_k \leftarrow 1]$, qui sont calculées en $O(|\vec{f}|)$. Ainsi la fonction “vectochar-recurse” devient :

```

function vectochar-recurse( $\vec{f}$ ) : TDG;
let  $x$  be a variable occurring in  $\vec{f}$  in
return  $\lambda\vec{y}.(\text{vectochar}(\vec{f}[x \leftarrow 0]) \vee \text{vectochar}(\vec{f}[x \leftarrow 1]));$ 

```

Figure 30. Deuxième algorithme de calcul de *Img*.

La variable x n’apparaît pas dans les graphes de $(\vec{f} \text{ srr } \neg x)$ et de $(\vec{f} \text{ srr } x)$. Si nous voulons exploiter le cache au maximum, nous devons éviter que les vecteurs qui y sont conservés soient composés de nouveaux noeuds. En choisissant la variable de plus petit ordre dans \vec{f} , le calcul de $\vec{f}[x \leftarrow 0]$ et $\vec{f}[x \leftarrow 1]$ est seulement en $O(k)$, où k est le nombre de composantes de \vec{f} , et aucun noeud n’est créé. Ce processus correspond à un parcours parallèle en profondeur d’abord des graphes de \vec{f} . Les seuls noeuds qui sont créés sont

ceux utilisés par les fonctions caractéristiques conservées dans le cache. Donc si les graphes du vecteur initial \vec{f} sont relativement petits, ou si il y a beaucoup de partages entre ces graphes, alors cet algorithme est très efficace. Comme les vecteurs pris comme entrée par la fonction “vectochar-recurse” sont construits à partir des sous-graphes du vecteur initial \vec{f} , le nombre de recursions de “vectochar” est borné par $\prod_{k=1}^n |f_k|$, c’est à dire exponentiel par rapport à n . En fait, nous pensons que le nombre de recursions ne dépend que de $|\vec{f}|$, évidemment de façon non polynomiale.

4.7 Résultats expérimentaux et discussion

4.7.1 L’algorithme final

L’algorithme qui calcule l’ensemble des états valides d’une machine \mathcal{M} est donné ci-dessous. Le nombre d’itérations de la boucle **while** est le plus long chemin acyclique (partant d’un état initial) du graphe d’états de la machine.

```

function compute-valid( $n, m, r, \omega, (Cns, \vec{f}), Init$ ) : TDG;
var Valid, New, Current : TDG;
Valid =  $Init$ ;
New =  $Init$ ;
while ( $New \wedge Cns$ )  $\neq 0$  do {
    Current = New;
    let Range = vectochar( $\vec{f} \downarrow (Current \wedge Cns)$ ) in {
        New = Range  $\wedge \neg Valid$ ;
        Valid = Valid  $\vee$  New;
    }
}
return Valid;

```

Figure 31. Première fonction “compute-valid”.

A chaque étape, $Current$ contient tous les états nouveaux dernièrement découverts (réunis dans l’ensemble New). On peut ajouter à $Current$ des états ayant déjà été découverts (ces états appartiennent à $Valid$) sans modifier la correction de *calcule-valid*. Donc l’algorithme demeure correct si on choisit $Current$ tel que $New \subset Current \subset Valid$. Comme la taille du vecteur ($\vec{f} \downarrow (Current \wedge Cns)$) est en $O(|\vec{f}| \times |Current| \times |Cns|)$, une idée [43, 44] naturelle pour réduire le coût de l’algorithme est de choisir un graphe $Current$ aussi petit que possible, tout en conservant la correction de l’algorithme.

Pour réduire le coût de l’algorithme il suffit de choisir le graphe $Current$ minimal tel que ($New \Rightarrow Current \Rightarrow Valid$). Mais ceci est un problème NP-difficile (voir le problème de la minimisation Section E.2). Nous utilisons donc une fonction “between” non optimale [43] possédant une complexité en $O(|New| + |Valid|)$. De cette façon, on obtient l’algorithme

```

function calcule-valid( $n, m, r, \omega, (Cns, \vec{f}), Init$ ) : TDG;
var Valid, New, Current : TDG;
Valid = Init;
New = Init;
while (New  $\wedge$  Cns)  $\neq$  0 do {
    Current = between(New, Valid);
    let Range = vectochar( $\vec{f} \downarrow (Current \wedge Cns)$ ) in {
        New = Range  $\wedge$   $\neg$  Valid;
        Valid = Valid  $\vee$  New;
    }
}
return Valid;

```

Figure 32. Fonction “compute-valid” finale.

final⁵ présenté Figure 32.

4.7.2 Résultats expérimentaux

Nous comparons ici les deux algorithmes proposés dans les Sections 4.6.1 et 4.6.2. Le premier est basé sur le restricteur d’image strict “constrain” et une partition du codomaine. Le second utilise une partition du domaine et est correct pour tout restricteur d’image strict.

Les deux algorithmes sont écrits en C, et les temps CPU en secondes ont été obtenus sur Sun SPARC station 1. La Table 5 donne les temps CPU nécessaires pour calculer l’ensemble des états valides de quelques circuits séquentiels. Pour chaque circuit, **#in** est le nombre de ses entrées, **#reg** le nombre de ses variables d’état, **depth** est le nombre d’itérations, **#valid** est le nombre d’états valides, **time-codp** (respectivement **time-dp**) est le temps CPU de l’algorithme basé sur une partition du codomaine (respectivement de l’algorithme basé sur une partition du domaine). La Table 6 présente pour les deux méthodes le nombre de récursions de “vectochar-recurse” (colonne **#rec**) requis par le calcul des états valides. Elle donne aussi le nombre d’identifications obtenues, réparties en identifications exactes (colonne **#same**), et en identifications étendues utilisant les propriétés des Théorèmes 4.8 ou 4.9 (colonne **#diff**). Ceci montre que pour la plupart des machines, le gain apporté par l’identification étendue est important.

⁵Afin d’améliorer “vectochar”, on peut utiliser cette idée [43] : comme les seuls états intéressants sont ceux qui n’appartiennent pas déjà à *Valid*, nous pouvons déterminer le domaine D où seuls de nouveaux états peuvent être trouvés. Ce domaine est $\lambda(\vec{y}@\vec{x}).(Current(\vec{y}) \wedge Cns(\vec{y}, \vec{x}) \wedge \neg Valid(\vec{f}(\vec{y}, \vec{x})))$. L’ensemble des nouveaux états *New* est directement égal à $vectochar(\vec{f} \downarrow D)$. Dans certain cas, cette méthode réduit considérablement le nombre de recursions de la fonction “vectochar”. De plus, il est alors suffisant de tester si D est égal à 0 pour pouvoir stopper l’itération, ce qui économise la dernière évaluation de *Img*. Mais le calcul de D requiert la composition $Valid \circ \vec{f}$, ce qui peut être très coûteux. Pour cette raison, nous avons abandonné cette technique, au contraire de [38].

\mathcal{M}	caractéristiques			calcul états valides		
	#regs	#ins	#valid	depth	time-codp	time-dp
<i>s298</i>	14	3	218	19	0.5	0.3
<i>s344</i>	15	9	2625	7	4.6	3.5
<i>s349</i>	15	9	2625	7	4.6	3.4
<i>s382</i>	21	3	8865	151	8.3	7
<i>s400</i>	21	3	8865	151	8.2	6.9
<i>s420</i>	16	19	17	17	0.3	0.3
<i>s444</i>	21	3	8865	151	8.3	6.9
<i>s526</i>	21	3	8868	151	7.9	7.0
<i>s641</i>	19	35	1544	7	13.1	8.1
<i>s713</i>	19	35	1544	7	12.5	8.3
<i>s838</i>	32	35	17	17	0.5	0.5
<i>cbp16</i>	16	17	$6.5 \cdot 10^4$	0.4	0.9	0.3
<i>cbp32</i>	32	33	$4.3 \cdot 10^9$	2	1.8	1.6
<i>key</i>	56	62	$7.2 \cdot 10^{16}$	2	2.8	1.1
<i>stage</i>	64	113	$1.8 \cdot 10^{19}$	2	mem full	183.6
<i>sbc</i>	28	40	$1.5 \cdot 10^5$	10	3188	444.1
<i>clm1</i>	33	14	$3.8 \cdot 10^5$	397	88.6	229.5
<i>clm2</i>	33	388	$1.6 \cdot 10^5$	412	101.1	189.7
<i>clm3</i>	32	382	$3.3 \cdot 10^6$	279	571.3	mem full
<i>mm10</i>	30	13	$1.8 \cdot 10^8$	4	mem full	5.5
<i>mm20</i>	60	23	$1.9 \cdot 10^{17}$	4	mem full	21.7
<i>mm30</i>	90	33	$2 \cdot 10^{26}$	4	mem full	56.1

Table 5. Résultats du calcul des états valides.

Certains circuits ne peuvent être traités que par un seul des deux algorithmes. A chaque étape durant le calcul des états valides de *clm2*, seul un petit nombre d'états est atteint, ce qui fait que la méthode basée sur le partitionnement du codomaine est efficace, contrairement à celle basée sur le partitionnement du domaine. Les circuits *MinMax* [44] (*mm20*, *mm25*, *mm30*) ne peuvent être traités que par une partition du domaine : le taux d'identification est très élevé pour cet algorithme, ce qui n'est pas le cas pour l'autre algorithme, et le nombre d'états qui sont atteints à chaque étape est très grand.

Comme ces deux algorithmes possèdent le même squelette (i.e. “vectochar”) et utilisent le cache de la même façon, ils peuvent être utilisés conjointement. Le problème est alors de trouver une fonction qui décidera à chaque récursion quelle technique doit être utilisée. Une autre voie de recherche consiste à définir des instances spécialisées de l'algorithme “vectochar” pour traiter certaines classes de machines.

La Figure 33 montre comment varie le logarithme du temps de calcul symbolique des états valides en fonction du logarithme du nombre de ces états valides. La droite de pente 1 correspond aux algorithmes d'énumération classiques, c'est à dire une dépendance linéaire

\mathcal{M}	codomaine split			domaine split		
	#rec	#matches	#extended	#rec	#matches	#extended
<i>s298</i>	32	24	0	32	27	11
<i>s344</i>	814	190	118	780	431	261
<i>s349</i>	814	190	118	780	431	261
<i>s382</i>	674	403	136	376	363	151
<i>s400</i>	674	403	136	376	363	151
<i>s420</i>	23	8	0	35	7	0
<i>s444</i>	672	399	161	376	363	185
<i>s526</i>	840	457	188	709	563	146
<i>s641</i>	1109	237	1	2184	1716	748
<i>s713</i>	1103	239	2	2349	1832	543
<i>s838</i>	23	8	0	35	7	0
<i>cbp16</i>	15	0	0	16	0	0
<i>cbp32</i>	61	40	0	32	13	0
<i>key</i>	109	106	0	2	1	0
<i>stage</i>	–	–	–	7339	10520	1391
<i>sbc</i>	122310	33343	4815	63762	63829	17376
<i>clm1</i>	2153	1764	344	19390	20361	1355
<i>clm2</i>	2363	2047	386	15972	17863	1381
<i>clm3</i>	29776	21981	5754	–	–	–
<i>mm10</i>	–	–	–	494	419	45
<i>mm20</i>	–	–	–	1094	969	105
<i>mm30</i>	–	–	–	1694	1519	165

Table 6. Analyse du test d'identification étendu.

du temps avec le nombre d'états valides. On peut vérifier expérimentalement qu'il n'y a *pas de dépendance* entre le nombre d'états valides et le temps de calcul symbolique de ceux-ci. Ceci est en accord avec l'analyse de complexité, car la complexité du calcul symbolique des états valides n'est liée qu'à la taille des TDGs manipulés, et il n'y a *aucune* relation entre le nombre d'éléments d'un ensemble et la taille du TDG de sa fonction caractéristique. Cette indépendance de la complexité de notre algorithme avec le nombre d'états valides n'a rien de curieuse : la complexité change d'aspect, elle est transférée sur les DAGs, plus exactement sur leurs tailles.

La taille des DAGs manipulés dépend de l'ordonnement des variables. Un bon ordre est celui pour lequel un ensemble de fonctions admettent des DAGs de tailles raisonnables, et trouver un bon ordre est un problème difficile (voir Section 2.5). Ici, le problème se complexifie, car on peut très bien utiliser un ordre des variables qui soit bon pour les DAGs décrivant la machine (i.e. *Init*, *Cns*, \vec{f} , ω), mais inadapté pour représenter le DAG *Valid*. Il est quasi-impossible de prévoir si l'ordre initial choisit pour les DAGs décrivant la machine conviendra à *Valid*, et les heuristiques d'ordonnement pour le traitement de machines séquentielles [81] deviennent complexes et assez aléatoires. Une idée simple

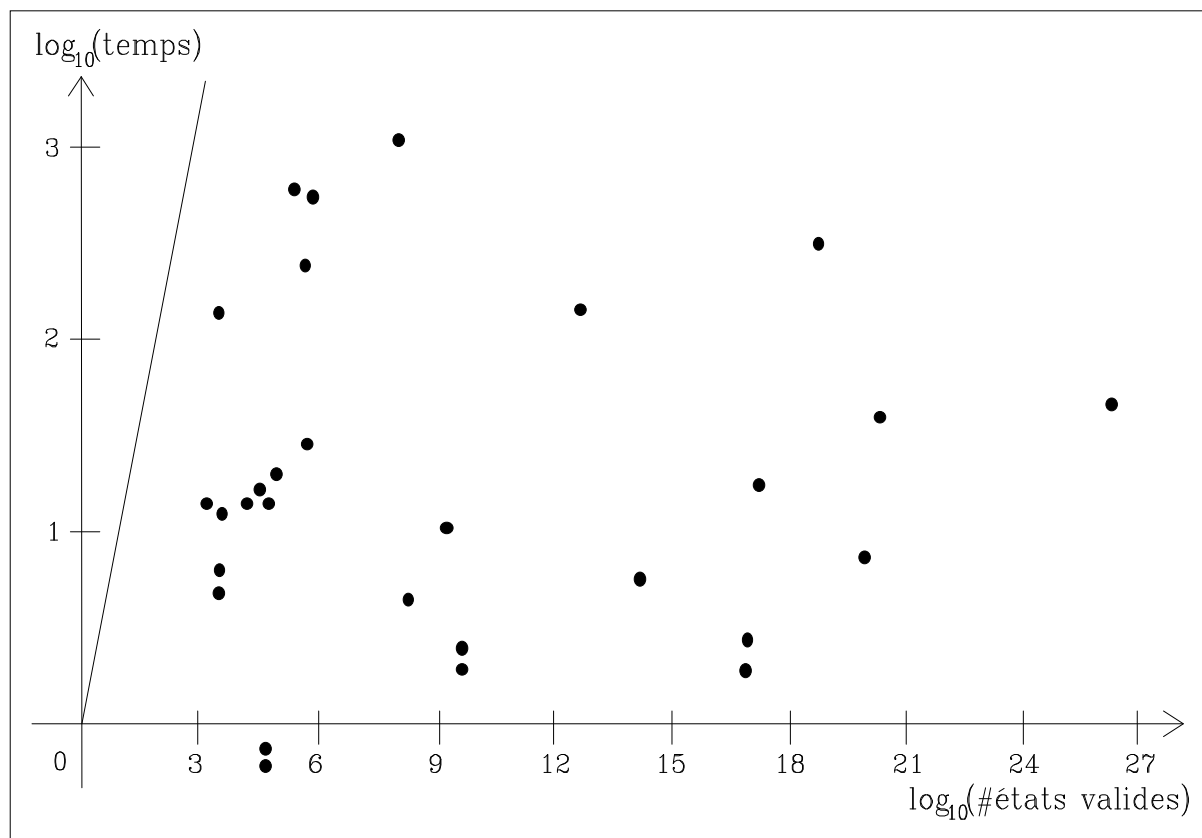


Figure 33. Courbe $\log_{10}(\text{time})$ en fonction de $\log_{10}(\#\text{valid})$.

pouvant éviter ce problème est la suivante : si pendant l'itération du calcul du point fixe, le graphe *Valid* devient trop gros, on recalcule un ordre des variables prenant en compte ce nouveau graphe, et on recalcule tous les graphes nécessaires (c'est à dire ceux décrivant la machine, plus les graphes *Valid* et *New*) d'après ce nouvel ordre. Cette modification dynamique de l'ordre des variables adapte le comportement de l'algorithme de preuve aux fluctuations de la taille du graphe *Valid*.

Cette technique ne peut évidemment pas toujours être efficace, car il existe des couples de fonctions ayant des bons ordres incompatibles, car contradictoires : si un ordre est bon (polynomial) pour une fonction, il peut être mauvais (non polynomial) pour l'autre, et réciproquement. Si *Valid* et l'une des fonctions intervenant dans la description de la machine forment un tel couple, l'ordonnancement dynamique n'atteindra pas son but. Il faut alors utiliser des ordres partiels pour faire cohabiter plusieurs ordres sur différents DAGs (voir Section 2.5).

De plus, on sait qu'il existe des fonctions pour lesquelles il n'existe aucun bon ordre. L'algorithme de calcul symbolique que nous avons proposé est forcément limité par des machines ayant un ensemble d'états valides dont la fonction caractéristique est une fonction de ce type, "intrinsèquement" très complexe.

4.8 Conclusion

Nous avons montré que notre calcul symbolique des états valides d’une machine séquentielle dépendait essentiellement de l’opération “*Img*”, car cette opération intervenant dans le calcul du point fixe est la seule qui soit NP-difficile vis-à-vis de la taille des DAGs qui représentent la machine séquentielle. Nous avons proposé un schéma d’algorithme, basé sur un restricteur d’image et une partition, qui permet de décomposer récursivement le calcul de “*Img*”.

Plusieurs types de restricteurs d’image ont été étudiés, et nous avons défini le restricteur d’image strict “constrain”. Ce nouvel opérateur sur les DAGs combine l’avantage d’être polynomial et l’utilisation de la notion de “plus proche interprétation”. Cette dernière notion permet d’éliminer des variables et de faire apparaître des constantes dans les DAGs traités, afin d’éviter de trop les expander.

Puis nous avons proposé deux instances du schéma d’algorithme en choisissant une partition et un restricteur d’image adapté. Le premier algorithme est entièrement basé sur notre opérateur “constrain”. Le second utilise “constrain” et effectue un parcours parallèle en profondeur d’abord d’un ensemble de DAGs de décision. Nous avons fourni une analyse de complexité et de comportement pour chacune de ces instances.

Les performances des algorithmes que nous avons proposés pour le calcul des états valides d’une machine ont été montrées. Elles dépassent celles des techniques précédemment connues, basées sur l’énumération, qui sont limitées à 1 million d’états valides, avec un traitement de 100 états par seconde [74]. Un fait essentiel est que la complexité de nos algorithmes ne dépend plus du nombre d’états valides. Notre méthode permet de résoudre des problèmes [44, 48, 38, 120] faisant intervenir des machines ayant typiquement 50 variables d’états et 50 variables d’entrées, ce qui est inabordable avec des techniques d’énumération.

Chapitre 5

Calcul de l'image réciproque d'une fonction

Etant donnés les DAGs des fonctions \vec{f} de $\{0, 1\}^m$ dans $\{0, 1\}^n$, de Cns de $\{0, 1\}^m$ dans $\{0, 1\}$, et de χ de $\{0, 1\}^n$ dans $\{0, 1\}$, nous étudions ici des algorithmes calculant la fonction $Pre(\vec{f}, Cns, \chi)$, définie de $\{0, 1\}^n$ dans $\{0, 1\}$. Après avoir présenté quelques résultats de complexité, nous comparons ce problème à celui du calcul de l'image, et montrons qu'il est plus complexe. Nous proposons alors quelques techniques de calcul, en discutant leurs intérêts respectifs.

5.1 Difficulté du calcul de $Pre(\vec{f}, Cns, \chi)$

La difficulté intrinsèque du problème est indiquée par le théorème suivant, qui montre que le calcul de $Pre(\vec{f}, Cns, \chi)$ est NP-difficile¹. On obtient le Théorème 5.2 plus précis quant aux graphes de décision.

Théorème 5.1 *Etant données les formes canoniques de \vec{f} et de χ , la canonisation de $Pre(\vec{f}, 1, \chi)$ est NP-difficile.*

Preuve. Nous ne donnons que la preuve sur les DAGs. Soit $C = (\bigwedge_{k=1}^n c_k)$ une 3-forme normale conjonctive composée de n clauses c_k . Chaque clause c_k est une disjonction de trois littéraux, donc le calcul des graphes de toutes les clauses se fait en $O(n)$. Soit χ le graphe de la fonction $\lambda[x_1 \dots x_n].(\bigwedge_{k=1}^n x_k)$. Ce graphe est un peigne, construit en $O(n)$. Tester si le graphe de $Pre(\vec{f}, 1, \chi)$ est différent de 0 se fait en $O(1)$. Or C est satisfiable si et seulement si $Pre(\vec{f}, 1, \chi) \neq 0$, donc calculer $Pre(\vec{f}, 1, \chi)$ est NP-difficile. \square

Théorème 5.2 *Etant donnés les DAGs de \vec{f} et χ , le calcul de $Pre(\vec{f}, 1, \chi)$ pour un ordre des variables fixé est non polynomial.*

¹On a $Pre(\vec{f}, Cns, \chi) = Pre(\vec{f}@[Cns], 1, \lambda(\vec{y}@[y_{n+1}]).(\chi(\vec{y}) \wedge y_{n+1}))$. Comme le graphe de $\lambda(\vec{y}@[y_{n+1}]).(\chi(\vec{y}) \wedge y_{n+1})$ est obtenu à partir du graphe de χ par un produit en $O(|\chi|)$, donc polynomial, le calcul de $Pre(\vec{f}, Cns, \chi)$ et $Pre(\vec{f}, 1, \chi)$ sont de même classe de complexité.

Preuve. Il suffit de constater que la composition se réduit à un calcul de $Pre(\vec{f}, 1, \chi)$. Tout ensemble de fonctions f_1, \dots, f_n, g peut s'écrire sous la forme $\lambda(\vec{y}@\vec{x}).f_1(\vec{y}), \dots, \lambda(\vec{y}@\vec{x}).f_n(\vec{y}), \lambda\vec{y}.g(\vec{y})$. Il suffit pour cela d'étendre le système des variables utilisées, ce qui est fait en $O(|g| + \sum_{k=1}^n |f_k|)$, donc polynomialement. Alors on a $Pre(\lambda(\vec{y}@\vec{x}).[f_1(\vec{y}) \dots f_n(\vec{y})], 1, \lambda\vec{y}.g(\vec{y})) = \lambda\vec{y}.(\exists \vec{x} g(f_1(\vec{y}), \dots, f_n(\vec{y})))$, ce dernier terme s'écrivant directement $\lambda\vec{y}.g(f_1(\vec{y}), \dots, f_n(\vec{y}))$, c'est à dire la composition de g et des fonctions f_1, \dots, f_n . \square

Nous sommes donc de nouveau confronté, comme pour la fonction Img dans le Chapitre 4, à un problème au moins exponentiel en mémoire dans le pire cas pour un ordre à priori fixé, ou même avec un ordre optimal donné dynamiquement. En fait, ce nouveau problème est intrinsèquement plus complexe que celui qui consiste à évaluer Img . Ceci est montré par le Théorème 5.3, qui affirme qu'on peut transformer polynomialement le calcul de Img en un calcul de Pre , alors que la réciproque est fautive. En effet, en reprenant les notations de la preuve de 5.3, il est impossible de décrire la composition $\chi'(\vec{g}(\vec{y}, \vec{x}))$ à partir des seuls composants \vec{g} , χ' , et Img . Pour cela, constatons que $Img(\vec{f}, \chi)$ calcule l'image de \vec{f} sur un ensemble χ , et $Pre(\vec{f}, 1, \chi)$ son image réciproque sur χ : si on veut utiliser Img pour traduire Pre , il faut disposer de la fonction réciproque \vec{f}^{-1} . Mais le calcul de \vec{f}^{-1} est lui-même NP-difficile (on peut même montrer que le calcul de l'image de \vec{f} sur 1 se réduit au calcul de \vec{f}^{-1}). Un argument plus simple est de voir que si \vec{f} est une fonction injective de $\{0, 1\}^m$ dans $\{0, 1\}^n$, alors on obtient une fonction (évidemment partielle) \vec{f}^{-1} de $\{0, 1\}^n$ dans $\{0, 1\}^m$. Mais si \vec{f} n'est pas injective, alors il faut considérer \vec{f}^{-1} comme une fonction de $\{0, 1\}^{2^n}$ dans $\{0, 1\}^m$. C'est ici que se produit un saut exponentiel, qui montre la non-réduction polynomiale du problème Pre à celui de Img . Aussi calculer $Pre(\vec{f}, 1, \chi)$ est strictement plus difficile que calculer $Img(\vec{f}, \chi)$.

Théorème 5.3 *Le problème de l'évaluation de Img se réduit linéairement à celui de l'évaluation de Pre .*

Preuve. Soit $\vec{f} = [f_1 \dots f_n]$ de $\{0, 1\}^m$ dans $\{0, 1\}^n$, et χ une fonction booléenne sur $\{0, 1\}^m$ non nulle. Etant donnés les DAGs de \vec{f} et χ , nous allons montrer que le calcul de $Img(\vec{f}, \chi)$ se transforme polynomialement en l'évaluation de $Pre(\vec{g}, 1, \chi')$, où \vec{g} et χ' sont précisées ci-après. Dans la suite, \vec{y} dénotera un point $[y_1 \dots y_{2n+1}]$ de $\{0, 1\}^{2n+1}$, et \vec{x} dénotera un point de $\{0, 1\}^m$. Soit \vec{g} la fonction de $\{0, 1\}^{2n+1} \times \{0, 1\}^m$ dans $\{0, 1\}^{2n+1}$ définie par :

$$\vec{g} = \lambda\vec{y}.\lambda\vec{x}.[y_1 \dots y_n f_1(\vec{x}) \dots f_n(\vec{x}) \chi(\vec{x})]$$

Soit $\chi' = \lambda\vec{y}.(y_{2n+1} \wedge (\bigwedge_{k=1}^n (y_k \Leftrightarrow y_{n+k})))$. C'est une fonction non nulle sur $\{0, 1\}^{2n+1}$. Alors on a :

$$\begin{aligned} Pre(\vec{g}, 1, \chi') &= \lambda\vec{y}.(\exists \vec{x} \chi'(\vec{g}(\vec{y}, \vec{x}))) \\ &= \lambda\vec{y}.(\exists \vec{x} \chi(\vec{x}) \wedge \left(\bigwedge_{k=1}^n y_k \Leftrightarrow f_k(\vec{x}) \right)) \\ &= \lambda\vec{y}.Img(\vec{f}, \chi)(y_1, \dots, y_n) \end{aligned}$$

Finalement, comme $Pre(\vec{g}, 1, \chi')(\vec{y})$ ne dépend que des variables y_1, \dots, y_n , on peut écrire $Img(\vec{f}, \chi) = \lambda[y_1 \dots y_n].Pre(\vec{g}, 1, \chi')(y_1, \dots, y_n, 0_1, \dots, 0_{n+1})$. Choisissons les $n+1$ variables

d'état y_{n+1}, \dots, y_{2n+1} ajoutées aux n variables d'état y_1, \dots, y_n de telle façon qu'on ait $y_1 < y_{n+1} < y_2 < y_{n+2} < \dots < y_n < y_{2n} < y_{2n+1}$. Avec cet ordre, le graphe de la fonction χ' a une taille de $3n + 1$. Ainsi le graphe de χ' est construit en $O(n)$. De même, le graphe de \vec{g} est construit en $O(n)$, puisqu'il suffit de générer n graphes élémentaires. Donc le passage des graphes de \vec{f} et χ à ceux de \vec{g} et χ' est linéaire avec n . Enfin, comme le terme $Pre(\vec{g}, 1, \chi')(\vec{y})$ ne dépend que des variables y_1, \dots, y_n , le graphe de $Img(\vec{f}, \chi)$ est directement celui de $Pre(\vec{g}, 1, \chi')$. \square

Nous avons montré que l'opération Pre est le goulot d'étranglement de la procédure de vérification de formules CTL. Nous étudions dans la suite comment cette résolution peut être effectuée de façon efficace.

5.2 Calcul de $Pre(\vec{f}, Cns, \chi)$ à l'aide de Δ

Une façon directe de calculer $Pre(\vec{f}, Cns, \chi)$ passe par la construction de la relation de transition Δ définie à partir de \vec{f} et Cns , puis par l'utilisation de l'identité suivante [29, 21].

$$Pre(\vec{f}, Cns, \chi) = \lambda \vec{y}. (\exists \vec{y}' \Delta(\vec{y}, \vec{y}') \wedge \chi(\vec{y}'))$$

Cependant, on sait que le graphe de $\lambda \vec{y}. \lambda \vec{x}. \lambda \vec{y}'. (\bigwedge_{k=1}^n y'_k \Leftrightarrow f_k(\vec{y}, \vec{x}))$, préalable à la construction de Δ , est construit en $O(2^n \times \prod_{k=1}^n |f_k|)$, et l'expérience montre que le pire cas est "souvent" atteint (voir Section 4.2). Comme la relation de transition ne peut être raisonnablement construite pour des machines trop complexes ou ayant trop de variables d'état, il nous faut trouver une résolution qui ne requiert pas cette construction.

5.3 Evaluation de $Pre(\vec{f}, Cns, \chi)$

Le terme $Pre(\vec{f}, Cns, \chi)$ est défini par :

$$Pre(\vec{f}, Cns, \chi) = \lambda \vec{y}. (\exists \vec{x} Cns(\vec{y}, \vec{x}) \wedge \chi(\vec{f}(\vec{y}, \vec{x})))$$

A partir de cette équation, le calcul de $Pre(\vec{f}, Cns, \chi)$ peut s'effectuer en deux étapes. La première consiste à construire le graphe de $(Cns(\vec{y}, \vec{x}) \wedge \chi(\vec{f}(\vec{y}, \vec{x})))$, ce qui est non polynomial à cause de la composition. La deuxième étape consiste à éliminer les variables quantifiées \vec{x} . Cette dernière étape est aussi de complexité non polynomiale, mais l'expérience montre qu'elle reste souvent faisable. La difficulté de l'évaluation de Pre vient de la composition $\chi(\vec{f}(\vec{y}, \vec{x}))$ qui y apparait. Il faut donc étudier comment évaluer efficacement cette composition.

5.3.1 Réduction de taille pour la composition

L'idée [43, 46] qui nous conduit à l'utilisation d'un opérateur de réduction est similaire à celle discutée Section 4.7.1. Elle consiste à tenir compte du fait que Pre intervient dans des équations très particulières, ce qui fait que le terme Pre peut être légèrement modifié sans altérer la validité des équations de point fixe.

Le calcul de la composition $(\chi \circ \vec{f})$ est au moins en $O(|\chi| \times \prod_{k=1}^n |f_k|)$ (voir Section 2.4.3). Il faut donc réduire au maximum la taille des arguments : on est conduit à étudier la minimisation des graphes de \vec{f} dans le terme $Pre(\vec{f}, Cns, \chi)$.

Rappelons l'équation 3.9 permettant de calculer l'ensemble des états satisfaisant une formule CTL de type EU :

$$\mathbf{E}_{k+1} = \lambda \vec{y}. \left(\mathbf{E}_k(\vec{y}) \vee \left(\mathbf{F}(\vec{y}) \wedge (\exists \vec{x} \ Cns(\vec{y}, \vec{x}) \wedge \mathbf{E}_k(\vec{f}(\vec{y}, \vec{x}))) \right) \right). \quad (5.1)$$

Lorsque que $\mathbf{E}_k(\vec{y}) = 1$, ou $\mathbf{F}(\vec{y}) = 0$, ou $Cns(\vec{y}, \vec{x}) = 0$, la valeur de $\mathbf{E}_{k+1}(\vec{y})$ ne dépend plus du terme $\mathbf{E}_k(\vec{f}(\vec{y}, \vec{x}))$. Aussi la fonction \vec{f} qui apparaît dans 5.1 peut être remplacée par n'importe quelle fonction \vec{f}' qui est égale à \vec{f} sur le domaine D défini par $D = \lambda \vec{y}. \lambda \vec{x}. (\neg \mathbf{E}_k(\vec{y}) \wedge \mathbf{F}(\vec{y}) \wedge Cns(\vec{y}, \vec{x}))$, sans changer la valeur de la fonction \mathbf{E}_{k+1} . Cette remarque tient aussi pour les autres formules 3.6 à 3.11 définissant les points fixes des formules CTL. Les domaines D respectifs sont :

$$\begin{aligned} D &= Cns && \text{pour 3.6 et 3.7,} \\ D &= \lambda \vec{y}. \lambda \vec{x}. (\neg \mathbf{E}_k(\vec{y}) \wedge \mathbf{F}(\vec{y}) \wedge Cns(\vec{y}, \vec{x})) && \text{pour 3.9,} \\ D &= \lambda \vec{y}. \lambda \vec{x}. (\neg \mathbf{A}_k(\vec{y}) \wedge \mathbf{F}(\vec{y}) \wedge Cns(\vec{y}, \vec{x})) && \text{pour 3.11.} \end{aligned}$$

Comme on cherche à valider des formules CTL sur les états atteignables de la machine, on peut restreindre le processus de calcul aux seuls états valides. Il est donc souhaitable de d'abord calculer l'ensemble des états valides grâce à Img (Section 3.2.2 et Chapitre 4), puis d'utiliser cette connaissance afin de ne considérer que des états valides durant le calcul de Pre . Ceci revient à rajouter dans le domaine D défini ci-dessus l'information $Valid(\vec{y})$ par un produit, c'est à dire que le nouveau domaine à considérer est $\lambda \vec{y}. \lambda \vec{x}. (Valid(\vec{y}) \wedge D(\vec{y}, \vec{x}))$. Autrement dit, les équations 3.12 à 3.17 deviennent :

$$\mathbf{EX} = \lambda \vec{y}. (Valid(\vec{y}) \wedge Pre(\vec{f}, Cns, \mathbf{F}))$$

$$\mathbf{AX} = \lambda \vec{y}. (Valid(\vec{y}) \wedge \neg Pre(\vec{f}, Cns, \neg \mathbf{F})(\vec{y}))$$

$$\mathbf{E}_0 = \lambda \vec{y}. (Valid(\vec{y}) \wedge \mathbf{G}(\vec{y})), \quad \text{et}$$

$$\mathbf{E}_{k+1} = \lambda \vec{y}. \left(Valid(\vec{y}) \wedge \left(\mathbf{E}_k(\vec{y}) \vee (\mathbf{F}(\vec{y}) \wedge Pre(\vec{f}, Cns, \mathbf{E}_k)(\vec{y})) \right) \right).$$

$$\mathbf{A}_0 = \lambda \vec{y}. (Valid(\vec{y}) \wedge \mathbf{G}(\vec{y})), \quad \text{et}$$

$$\mathbf{A}_{k+1}(\vec{y}) = \lambda \vec{y}. \left(Valid(\vec{y}) \wedge \left(\mathbf{A}_k(\vec{y}) \vee (\mathbf{F}(\vec{y}) \wedge \neg Pre(\vec{f}, Cns, \mathbf{A}_k)(\vec{y})) \right) \right)$$

Nous proposons différentes techniques de minimisation d'une fonction partielle (D, f) dans l'Annexe E. Par exemple, si on utilise la fonction "restrict" [43] (voir Section E.4), alors on calculera la composition $(\chi \circ (\vec{f} \Downarrow D))$ au lieu de $(\chi \circ \vec{f})$, en garantissant que chaque graphe $(f_k \Downarrow D)$ est de taille inférieure ou égale à celui de f_k , ce qui peut considérablement diminuer le temps de la composition.

Des expériences [29] utilisant l'idée que nous avons proposée (reprise dans [29] sous l'appellation *frontier set minimization*) ont montré que le temps de calcul du point fixe de formules CTL sur certains exemples (opérateurs séquentiels en pipe-line) pouvait être réduit de 40%.

5.3.2 Réduction incrémentale pendant la composition

Un algorithme de composition [27] “compose1” directement inspiré de l'expansion de Shannon s'écrit :

```

function compose1( $g, l$ ) : TDG;
if  $g = 1$  or  $g = 0$  or  $l = ()$  then return  $g$ ;
let  $((y_1, f_1), \dots, (y_k, f_k)) = l$  and
       $x = g.root$  in {
  if  $x < y_1$  then return  $((\neg x \wedge \text{compose1}([g \leftarrow 0], l)) \vee (x \wedge \text{compose1}([g \leftarrow 1], l)))$ ;
   $l = ((y_2, f_2) \dots (y_k, f_k))$ ;
  if  $x > y_1$  then return  $\text{compose1}(g, l)$ ;
  return  $((\neg f_1 \wedge \text{compose1}([g \leftarrow 0], l)) \vee (f_1 \wedge \text{compose1}([g \leftarrow 1], l)))$ ;
}

```

Figure 34. Fonction “compose1”.

Sous l'hypothèse que $y_1 < \dots < y_n$, “compose1($g, ((y_1, f_1), \dots, (y_n, f_n))$)” calcule le graphe de la formule $g[y_1 \leftarrow f_1, \dots, y_n \leftarrow f_n]$. A cet algorithme on peut associer un cache pour éviter les calculs redondants, ce qui fait que sa complexité est au pire en $O(|g|^2 \times \prod_{k=1}^n |f_k|)$ (voir Section 2.4.3).

L'idée de la réduction de taille incrémentale pendant la composition est la suivante. Pendant la descente récursive, on peut simplifier les termes f_2, \dots, f_k qui seront substitués aux variables y_2, \dots, y_k en utilisant la connaissance du chemin pris dans l'arbre des récursions, c'est à dire en utilisant la connaissance de l'interprétation de f_1 . Un nouvel algorithme de composition “compose2” [106] utilisant cette simplification incrémentale (avec l'opérateur “restrict” par exemple) s'écrit simplement :

```

function compose2( $g, l$ ) : TDG;
if  $g = 1$  or  $g = 0$  or  $l = ()$  then return  $g$ ;
let  $((y_1, f_1), \dots, (y_k, f_k)) = l$  and
       $x = g.root$  in {
  if  $x < y_1$  then return  $((\neg x \wedge \text{compose2}([g \leftarrow 0], l)) \vee (x \wedge \text{compose2}([g \leftarrow 1], l)))$ ;
  if  $x > y_1$  then return  $\text{compose2}(g, ((y_2, f_2) \dots (y_k, f_k)))$ ;
  let  $l_0 = ((y_2, f_2 \Downarrow \neg f_1) \dots (y_k, f_k \Downarrow \neg f_1))$  and
       $l_1 = ((y_2, f_2 \Downarrow f_1) \dots (y_k, f_k \Downarrow f_1))$  in
  return  $((\neg f_1 \wedge \text{compose2}([g \leftarrow 0], l_0)) \vee (f_1 \wedge \text{compose2}([g \leftarrow 1], l_1)))$ ;
}

```

Figure 35. Fonction “compose2”.

Des expériences [106] utilisant cet algorithme “compose2” ont montré qu’il pouvait réduire de 10% à 50% le temps de composition par rapport à l’algorithme de composition “compose1”.

5.3.3 Décomposition de la composition

Les algorithmes de composition “compose1” et “compose2” présentés dans la Section précédente traversent le graphe g en profondeur d’abord, et calculent du bas vers le haut la composition $g[y_1 \leftarrow f_1, \dots, y_n \leftarrow f_n]$. Un cache est associé à “compose1” pour associer à chaque noeud le résultat partiel de la composition obtenue, et faire ainsi un nombre de récursion en $O(|g|)$. Le problème est qu’avec cet algorithme, les graphes intermédiaires associés aux noeuds peuvent devenir très grands et saturer la mémoire, d’où l’impossibilité de parvenir au résultat final.

L’idée utilisée ici [46] est d’exprimer le terme $\chi(\vec{f}(\vec{y}, \vec{x}))$ comme une disjonction de formules $h_k(\vec{y}, \vec{x})$ ($1 \leq k \leq q$) dont les graphes sont plus petits que celui de $\chi(\vec{f}(\vec{y}, \vec{x}))$. L’intérêt de cette décomposition est la transformation suivante :

$$\begin{aligned}
 Pre(\vec{f}, Cns, \chi) &= \lambda \vec{y}. (\exists \vec{x} \ Cns(\vec{y}, \vec{x}) \wedge \chi(\vec{f}(\vec{y}, \vec{x}))) \\
 &= \lambda \vec{y}. (\exists \vec{x} \ Cns(\vec{y}, \vec{x}) \wedge \left(\bigvee_{k=1}^q h_k(\vec{y}, \vec{x}) \right)) \\
 &= \lambda \vec{y}. (\exists \vec{x} \ \left(\bigvee_{k=1}^q Cns(\vec{y}, \vec{x}) \wedge h_k(\vec{y}, \vec{x}) \right)) \\
 &= \lambda \vec{y}. \left(\bigvee_{k=1}^q (\exists \vec{x} \ Cns(\vec{y}, \vec{x}) \wedge h_k(\vec{y}, \vec{x})) \right)
 \end{aligned}$$

Le terme final signifie que Pre peut être décomposé en la somme de plusieurs termes beaucoup plus simple. En effet, il n’y a ici plus de composition, et les variables \vec{x} peuvent être \exists -éliminées indépendamment dans chaque composante de la somme. Le problème est donc de trouver les fonctions h_k .

Etant donné un graphe g et une substitution $[y_1 \leftarrow f_1, \dots, y_n \leftarrow f_n]$, le calcul des graphes h_1, \dots, h_q dont la somme est égale à $g[y_1 \leftarrow f_1, \dots, y_n \leftarrow f_n]$ est effectué par la fonction “decompose-composition” donnée Figure 36.

Chaque noeud du graphe de g est accessible par plusieurs chemins². Un chemin est défini par un unique produit (c’est l’argument *path* dans l’algorithme), constitué des variables rencontrées de la racine du graphe jusqu’au noeud atteint. Le principe est de déterminer dans chaque noeud du graphe la somme S de tous les chemins (i.e. les interprétations) qui permettent d’accéder à celui-ci, de calculer la composition $(S \circ \vec{f})$, puis de propager ce résultat partiel aux noeuds fils. Le regroupement des chemins permet de factoriser des calculs qui seraient éclatés et reconduits plusieurs fois avec l’algorithme “compose1” ou “compose2” de la Section précédente.

²un chemin dans un graphe dénote un ensemble d’interprétations

```

var SetFun : list of TDG;

function decompose-composition( $g, l$ ) : list of TDG;
set-nbpath-and-path( $g$ );
SetFun = ();
produce-functions( $g, l, 1$ );
return SetFun;

function produce-functions( $g, l, path$ ) : void;
if  $g = 0$  then return;
if  $g = 1$  then {
  SetFun = append-to-list(SetFun,  $path$ );
  return;
}
let  $((y_1, f_1) \dots (y_n, f_n)) = l$  and
       $x = g.root$  in {
  if  $x > y_1$  then {
    produce-functions( $g, ((y_2, f_2) \dots (y_n, f_n)), path$ );
    return;
  }
  if  $x = y_1$  then {
     $g.nbpath = g.nbpath - 1$ ;
     $g.path = g.path \vee path$ ;
    if  $g.nbpath = 0$  then {
      produce-functions( $g[x \leftarrow 0], ((y_2, f_2) \dots (y_n, f_n)), \neg f_1 \wedge g.path$ );
      produce-functions( $g[x \leftarrow 1], ((y_2, f_2) \dots (y_n, f_n)), f_1 \wedge g.path$ );
    }
  }
}
}

```

Figure 36. Fonction “decompose-composition”.

Dans l’algorithme, le champ $path$ d’un noeud est destiné à recevoir le terme $(S \circ \vec{f})$. Il est initialisé à 0 par le fonction “set-nbpath-and-path”, et mis à jour chaque fois qu’on atteint un noeud par un nouveau chemin. Afin de savoir si tous les chemins permettant d’accéder à un noeud ont bien été considérés, le nombre des chemins restant à traverser est rangé dans le champ $nbpath$ associé au noeud. Sa valeur initiale est le nombre de tous les chemins atteignant le noeud, et est calculé par la fonction “set-nbpath-and-path”. La fonction “set-nbpath-and-path” qui initialise les deux champs $path$ et $nbpath$ des noeuds du graphe g consiste en un simple parcours du graphe, elle est donc en $O(g)$. Le champ $nbpath$ d’un noeud est décrémenté lorsque la fonction “compose” y accède.

Lorsque le champ $nbpath$ d’un noeud devient nul, le champ $path$ associé à ce noeud contient le terme $(S \circ \vec{f})$. On récurse alors en propageant le terme partiel $(S \circ \vec{f})$ (argument

path de la fonction “produce-functions”), de façon à calculer les termes partiels du type $(S \circ \vec{f})$ des noeuds fils.

Chaque fois qu'une feuille 1 est atteinte, le graphe d'une des fonctions h_k est généré. Ce graphe est celui du terme $(S_1 \circ \vec{f})$, où S_1 est la somme de tous les chemins accédant à 1 en passant par une unique branche terminale. Comme la formule $g[y_1 \leftarrow f_1, \dots, y_n \leftarrow f_n]$ est aussi la somme de toutes ses interprétations (i.e. chemins) la satisfaisant, on a bien :

$$g[y_1 \leftarrow f_1, \dots, y_n \leftarrow f_n] = \left(\bigvee_{h_k \in \text{SetFun}} h_k \right).$$

L'éclatement de la composition $(\chi \circ \vec{f})$ en une liste de graphes h_1, \dots, h_q dont on *sait* que la somme est $(\chi \circ \vec{f})$, mais qui n'est pas *réalisée*, permet ensuite d' \exists -éliminer les variables \vec{x} *indépendamment* sur chaque graphe h_k . Cet algorithme pourrait encore être amélioré en éliminant directement les variables \vec{x} dans les termes partiels $(S \circ \vec{f})$, de façon à réduire le nombre de variables le plus tôt possible.

5.4 Résultats expérimentaux et discussion

La technique de vérification de formules CTL présentée ici a été appliquée à deux machines chez BULL. La machine *clm1* conçue chez BULL (voir Section 4.7.2, Table 5) décrit une partie de l'interface entre deux bus. Elle est faite de 33 registres d'état et de 14 entrées. Les graphes qui représentent globalement sa fonction de transition ont plus de 10000 noeuds, et cinq des registres ont une fonction de transition de plus de 2500 noeuds. Cette machine possède environ $3.8 \cdot 10^5$ états valides parmi les 2^{33} états potentiels. Le temps CPU nécessaire pour effectuer le calcul symbolique de ses états valides est de 88.5 secondes sur un Sun SPARC Station 1. La propriété temporelle à valider sur cette machine requiert le calcul d'un seul point fixe demandant 38 itérations. Nous n'avons pu construire le graphe de la relation de transition de cette machine, aussi la technique décrite Section 5.2 ne peut être utilisée. La composition $\chi(\vec{f}(\vec{y}, \vec{x}))$ utilisée à chaque itération a cessé d'être raisonnablement calculable avec l'algorithme de composition standard après seulement quelques itérations. Si on utilise la composition standard et la simplification décrite Section 5.3.1, le point fixe est calculé en 1502 secondes. La fonction “restrict” (utilisée conjointement avec le domaine D défini Section 5.3.1) est indispensable pendant ce calcul, puisqu'il réduit les graphes de \vec{f} utilisés dans la composition de façon à ce que pendant le calcul du point fixe, aucun de ces graphes n'a une taille supérieure à 186, soit un gain de taille allant jusqu'à 10 pour chacune des fonctions de transition f_k des variables d'état. Le temps CPU est ramené à 623 secondes si on utilise en plus la minimisation incrémentale de la Section 5.3.2 pour évaluer la composition.

La machine séquentielle *sync* est un synchronisateur ayant 21 registres d'états et 4 entrées. La propriété à valider exprime que la valeur d'une sortie *ok* est égale à 1 pour tous les états valides de la machine. Ceci est traduit par la formule $(Init \models AG(ok))$. Le temps CPU nécessaire pour prouver cette formule avec la composition standard et sans utilisation des états valides est de 2080 secondes et le point fixe est atteint en 9 étapes. L'opérateur “restrict” n'a ici été d'aucune utilité puisqu'il n'a pu réduire la taille des

graphes de la fonction de transition. Cependant, si on utilise l'ensemble des états valides *Valid* pour simplifier les arguments de la composition, la validation est *instantanée*, car la fonction *ok* est égale à 1 sur l'ensemble *Valid*, ce qui implique que $(ok \Downarrow Valid) = 1$.

Il est à remarquer que la formule $AG(ok)$ est une propriété de sûreté, donc elle appartient à la classe de formules qui peuvent être vérifiées en comparant la machine sur laquelle doit être validée cette formule avec un automate construit à partir de la formule. On peut donc utiliser la technique de vérification présentée dans la Section 3.2. Dans ce cas précis, il suffit de vérifier que la sortie *ok* est toujours égale à 1 sur l'espace des états valides de la machine. Valider la propriété en utilisant cette méthode ne requiert que 58 secondes, le calcul des états valides étant effectué en 20 étapes.

Cette exemple semble montrer que si la formule peut être validée en utilisant une comparaison de machines (l'une étant le modèle universel), alors il est préférable d'appliquer cette technique plutôt que l'algorithme général de vérification des formules CTL. En pratique, on trouve en effet des machines qui ne peuvent être traversées en arrière, alors que leur ensemble d'états valides peut être symboliquement calculé. Ceci n'est qu'une illustration du Théorème 5.3, où nous avons montré que calculer l'image réciproque est intrinsèquement plus difficile que calculer l'image³.

Parallèlement à notre technique de preuve symbolique de formule CTL sur une machine séquentielle, l'équipe d'Edmund Clarke de *Carnegie Mellon University* a développé une technique de preuve très similaire [29]. Elle diffère de la notre en ce qu'elle utilise la *relation* de transition de la machine pour la résolution des équations de points fixes (voir Section 5.2), alors que nous utilisons la composition sur la *fonction* de transition (voir Section 5.3). Cette technique a été utilisée pour la vérification d'opérateurs arithmétiques "pipeline" [30], possédant jusqu'à 10^{120} états utiles, ce qui rend inapplicables les techniques énumératives. Cette méthode a été aussi utilisée pour valider des protocoles décrits en CCS [60]. Retenons surtout ce qui me semble être l'application la plus probante, la détection d'erreurs dans un protocole de communication entre les caches mémoires d'une machine multi-processeurs, erreurs qui n'avaient pas été détectées durant la période de test [88].

5.5 Conclusion

Nous avons présenté dans le Chapitre 3 une procédure automatique de validation de formules CTL sur une machine séquentielle, qui s'appuie sur des manipulations de formules propositionnelles quantifiées. Cette procédure de preuve ne requiert ni la construction du diagramme d'état de la machine, ni la construction de la relation de transition de la machine.

³A propos des deux stratégies possibles pour vérifier une propriété de sûreté, certains ont affirmé que le nombre d'itérations requis pour atteindre le point fixe par un parcours en arrière est inférieur à celui requis par un parcours en avant. C'est évidemment faux. Pour $n_1 \geq 0$ et $n_2 \geq 0$ quelconques, on peut exhiber un modèle sur lequel le nombre d'itérations nécessaire pour atteindre le point fixe en avant (respectivement en arrière) est égal à n_1 (respectivement à n_2). Il n'y a *aucune* relation entre n_1 et n_2 .

Nous avons montré que cet algorithme de vérification symbolique de formules temporelles (Section 3.3.2) est essentiellement tributaire de l'évaluation de *Pre*, puisque c'est la seule opération NP-difficile utilisée durant le calcul des points fixes. La difficulté de *Pre* est due à l'occurrence d'une composition dans son expression. Nous avons donc présenté plusieurs techniques afin d'évaluer au mieux cette composition.

L'algorithme de preuve symbolique que nous avons défini permet de vérifier des propriétés temporelles sans construire aucun diagramme d'états, au contraire des techniques proposées dans le passé, telle que le *Model Checking*. Il est essentiel que la complexité de notre algorithme ne dépend pas du nombre d'états valides. Il peut donc être appliqué à des machines ayant un nombre d'états utiles dépassant largement la capacité des techniques précédemment connues.

Conclusion

La vérification formelle devient peu à peu une réalité industrielle. Le centre de recherche de Bull a été particulièrement novateur en la matière : les graphes de décision typés y ont été conçus en 1987, et le premier outil industriel de preuve formelle de circuits combinatoires, *PRIAM*, a été intégré dans le système de CAO de Bull en 1988.

La suite logique consistait à étudier la preuve formelle de systèmes séquentiels. C'est ce travail qui a été présenté ici. Sa concrétisation est une boîte à outils, *SIAM*, remplissant de multiples tâches.

Nous avons défini des algorithmes symboliques permettant de comparer deux machines séquentielles, et de valider une machine en prouvant qu'elle satisfait des propriétés temporelles. Les algorithmes de *SIAM* sont novateurs dans le sens où leurs complexités ne *dépendent pas* du nombre d'états et de transitions des machines traitées, ce qui constituait la limite des techniques précédemment proposées. *SIAM* permet donc de comparer ou de valider des machines dont le très grand nombre d'états utiles interdit l'utilisation de techniques basées sur des parcours de diagramme d'états.

SIAM a suscité une extension considérable des primitives de manipulation des graphes de décision. Nous avons introduit de nouvelles fonctionnalités au système de preuve existant (élimination de quantificateurs, résolution d'équations), pour accroître la puissance et la souplesse de l'expression et du calcul. Nous avons aussi créé des opérations de manipulations non logiques, initialement destinées à des algorithmes spécialisés (réduction, minimisation, intégration de contraintes, élimination de redondances). Ces divers opérateurs et fonctionnalités disponibles dans *SIAM* ont trouvé des applications en dehors de la preuve de matériel, en particulier dans le monde de la synthèse de circuits.

Les techniques et algorithmes que nous avons proposés ici ont été repris et/ou améliorés par d'autres équipes de recherche dans le monde. On commence à voir sur le marché des produits de CAO intégrant certaines techniques introduites dans *SIAM*.

SIAM ne constitue pas un outil pouvant être directement intégré dans la chaîne de CAO de Bull. Un système séquentiel industriel complexe, tel qu'un protocole de communication, possède assez souvent plusieurs centaines de variables (propositionnelles) d'état. En pratique, *SIAM* ne peut pas traiter une telle machine en son entier. Les algorithmes de parcours symboliques peuvent sans doute être améliorés, mais on atteint la limite de l'exploitation de ces techniques.

Cependant, les systèmes séquentiels conséquents sont décrits par l'assemblage de "petites" machines séquentielles qui échangent des informations entre elles. Il est donc souhaitable de tirer partie de cette structure, car chacune des sous-machines est d'une taille qui rentre dans le domaine d'application de *SIAM*. On peut par exemple décomposer une preuve en plusieurs preuves partielles sur des sous-machines, ou faire abstraction d'une partie du comportement du système, ceci sans modifier la validité de la preuve [84, 128, 1, 121, 56]. C'est un nouvel objectif dans l'optique d'un système de preuve automatique de systèmes séquentiels pouvant être intégré dans une chaîne de CAO industrielle.

Annexes

Annexe A

Terme, réécriture, forme normale et canonique

Terme

Considérons un ensemble dénombrable \mathcal{V} , appelé ensemble des symboles de variable, et les ensembles dénombrables \mathcal{F}_k pour $k \geq 0$, où l'ensemble \mathcal{F}_k est appelé ensemble des symboles de fonction d'arité k . Nous appellerons aussi \mathcal{F}_0 l'ensemble des constantes. A partir de ces ensembles, on définit inductivement l'ensemble des *termes* \mathcal{T} : toute variable est un terme ; soit t_1, \dots, t_n des termes et f un symbole de \mathcal{F}_n , alors $f(t_1, \dots, t_n)$ est un terme.

Substitution

On appelle *substitution* une fonction de \mathcal{V} dans \mathcal{T} presque partout égale à l'identité. On représente une substitution en énumérant tous les cas où elle diffère de l'identité. Par définition, $[x_1 \leftarrow t_1, \dots, x_n \leftarrow t_n]$ est la substitution σ telle que $\sigma(x_k) = t_k$ pour $1 \leq k \leq n$, et $\sigma(x) = x$ pour $x \notin \{x_1, \dots, x_n\}$. Les substitutions seront postfixées sur leur argument.

On appelle substitution élémentaire (ou remplacement) une substitution égale à l'identité sauf pour un seul élément de \mathcal{V} . Toute substitution peut s'écrire comme la composition de substitutions élémentaires. Plus précisément, il faut disposer d'un ensemble dénombrable de nouveaux symboles \mathcal{W} disjoint de \mathcal{V} , ce qui permet de décomposer toute substitution de $(\mathcal{V} \rightarrow \mathcal{T})$ en une composition de substitutions élémentaires prises dans $((\mathcal{V} \cup \mathcal{W}) \rightarrow \mathcal{T})$. On évite ainsi les "effets de bord". Par exemple, la substitution $[x \leftarrow y, y \leftarrow x]$ qui échange *parallèlement* les variables x et y se décompose sous la forme $[x \leftarrow U][y \leftarrow x][U \leftarrow y]$ avec $U \in \mathcal{W}$. Celle-ci ne peut pas s'écrire $[x \leftarrow z][y \leftarrow x][z \leftarrow y]$ avec $z \in \mathcal{V}$, car $f(x, y, z)$ est un terme appartenant au langage, et $f(x, y, z)[x \leftarrow z][y \leftarrow x][z \leftarrow y]$ est égal à $f(y, x, y)$, au lieu du terme $f(y, x, z)$ escompté.

Toute substitution σ s'étend en une fonction σ_* de \mathcal{T} dans \mathcal{T} : pour toute variable x , $\sigma_*(x) = \sigma(x)$; pour un terme $f(t_1, \dots, t_n)$, $\sigma_*(f(t_1, \dots, t_n)) = f(\sigma_*(t_1), \dots, \sigma_*(t_n))$. Dans la suite, on ne distinguera plus σ et σ_* . On dira que les substitutions bijectives sont des renommages de variables.

Unification

On définit un ordre partiel sur \mathcal{T} par : $t \preceq t'$ si et seulement si il existe une substitution σ telle que $\sigma(t) = t'$. Intuitivement, ceci revient à dire que t est “plus général” ou “moins instancié” que t' . Pour un ensemble de variables V , on définit aussi un ordre partiel sur les substitutions par : $\sigma' \preceq [V]\sigma$ si et seulement si il existe une substitution ϱ telle que pour toute variable x de V , $\sigma'(x) = \varrho(\sigma(x))$. On dit que deux termes t et t' sont unifiables si et seulement si il existe une substitution σ tel que $\sigma(t) = \sigma(t')$. Si V est un ensemble de variables contenant les variables qui apparaissent dans t et t' , on peut alors montrer qu’il existe un unificateur minimal au sens de $\preceq [V]$, unique modulo un renommage des variables.

Réécriture

Une réécriture consiste à appliquer une transformation syntaxique sur un terme. Essentiellement, une règle de réécriture est une relation sur \mathcal{T} . On notera $(l \rightarrow r)$ une règle de réécriture, où l et r sont des termes (éventuellement construits à partir d’un sur-ensemble de \mathcal{V} et $\bigcup_{k \geq 0} \mathcal{F}_k$). On appelle système de réécriture \mathcal{R} un ensemble dénombrable de règles de réécriture. Un système de réécriture \mathcal{R} définit une relation sur \mathcal{T} , notée $\xrightarrow{\mathcal{R}}$, constituée par la plus petite relation sur \mathcal{T} contenant \mathcal{R} , close par substitutions et remplacements, comme indiqué ci-dessous. On confondra dans la suite cette relation avec sa fermeture transitive.

$$\begin{aligned} (l \rightarrow d) \in \mathcal{R} &\Rightarrow l \xrightarrow{\mathcal{R}} d \\ t \xrightarrow{\mathcal{R}} t' &\Rightarrow \sigma(t) \xrightarrow{\mathcal{R}} \sigma(t') \\ t \xrightarrow{\mathcal{R}} t' &\Rightarrow t''[x \leftarrow t] \xrightarrow{\mathcal{R}} t''[x \leftarrow t'] \end{aligned}$$

Forme compatible, normale, canonique

On dit qu’un système de réécriture \mathcal{R} est compatible avec une relation d’équivalence $=_{\mathcal{T}}$ sur \mathcal{T} si pour tous termes t, t', t'' , $t \xrightarrow{\mathcal{R}} t'$ et $t \xrightarrow{\mathcal{R}} t''$ implique $t' =_{\mathcal{T}} t''$. Etant donné un système de réécriture \mathcal{R} , on dit qu’un terme t est réécrit sous une forme normale t' si $t \xrightarrow{\mathcal{R}} t'$, et tout terme t'' tel que $t' \xrightarrow{\mathcal{R}} t''$ satisfait $t'' = t'$. On dit qu’un système de réécriture est normal si tout terme possède une forme normale. On dit qu’un système de réécriture est canonique si tout terme possède une forme normale unique. On notera \hat{t} la forme canonique du terme t , et $=_{\mathcal{R}}$ la relation d’équivalence engendrée par un système de réécriture canonique \mathcal{R} .

Une forme canonique \mathcal{R} compatible avec une relation d’équivalence $=_{\mathcal{T}}$ sur \mathcal{T} est telle que $=_{\mathcal{R}} =_{\mathcal{T}}$ coïncident. Ceci signifie que $t =_{\mathcal{T}} t'$ si et seulement si $\hat{t} = \hat{t}'$, c’est à dire si \hat{t} et \hat{t}' sont syntaxiquement égaux. Ainsi une forme canonique compatible avec $=_{\mathcal{T}}$ permet de reporter le processus de décision lié à l’expression $t =_{\mathcal{T}} t'$ sur un processus de transformation syntaxique. L’existence d’un système de réécriture canonique ou normal, et compatible, avec certaines propriétés (décidabilité, finitude, terminaison, tractabilité) sont des problèmes directement liés à ce qui est connu sous l’expression générique “démonstration automatique”.

Annexe B

Autres graphes typés

Nous présentons ici deux autres représentations des formules booléennes sous la forme de graphes canoniques typés, les *graphes 4-typés*, et les *graphes symétrisés*.

B.1 Graphe 4-typé

La compaction de la représentation des formules propositionnelles présentée Section 2.3.3 peut être poursuivie en enrichissant le type des structures $\Delta(-, -, -)$, ce qui permet de jouer sur le codage et la répartition des négations. Considérons par exemple cette forme, que nous appellerons *graphe 4-Typé*. Elle utilise les quatre types ++, +-, +-, --. Une syntaxe contenant cette forme s'écrit :

$$\begin{aligned} \langle tree \rangle & ::= \langle type \rangle \langle vertex \rangle \\ \langle vertex \rangle & ::= \Delta(\langle var \rangle, \langle tree \rangle, \langle tree \rangle) \\ & ::= 1 \\ \langle type \rangle & ::= ++ \mid -+ \mid +- \mid -- \\ \langle var \rangle & ::= x_1 \mid \dots \mid x_n \end{aligned}$$

L'interprétation de ces types exprimée avec la forme de Shannon est la suivante, où L et H dénotent des fonctions positives :

$$\begin{aligned} ++ \Delta(x, L, H) & = \Delta(x, L, H) \\ -+ \Delta(x, L, H) & = \Delta(x, \neg L, H) \\ +- \Delta(x, L, H) & = \Delta(x, L, \neg H) \\ -- \Delta(x, L, H) & = \neg \Delta(x, L, H) \end{aligned}$$

Ainsi la structure $\Delta(-, -, -)$ permet de représenter 4 fonctions différentes par modification de son type supérieur.

Une forme canonique peut être obtenue en imposant que tous les sous-arbres stricts composant un arbre 4-typé dénotent uniquement des fonctions positives. Seul le type de l'arbre principal permettra d'interpréter correctement la fonction. Intuitivement, le type d'une fonction f s'écrivant $((\neg x \wedge L) \vee (x \wedge H))$ correspond aux cas suivants : si L et H sont positifs, f est positive de type ++ ; si L est négatif et H positif, f est positive de

type $-+$; si L est positif et H négatif, f est négative de type $+ -$; enfin si L et H sont négatifs, f est négative de type $--$. Ainsi une fonction est positive si et seulement si elle est de type $++$ ou $-+$, ce seront donc les seuls types autorisés pour les sous-arbres stricts, les types $+ -$ et $--$ dénotant des fonctions négatives ne pouvant apparaître qu'à la racine de l'arbre principal. La forme 4-typée d'un arbre de Shannon t s'obtient en évaluant le terme $\text{Type}(t)$, où la fonction Type est définie par :

$$\begin{aligned}\text{Type}(\Delta(x, L, H)) &= \text{TypeUp}(x, \text{Type}(L), \text{Type}(H)) \\ \text{Type}(1) &= ++1 \\ \text{Type}(0) &= --1\end{aligned}$$

$$\begin{aligned}\text{TypeUp}(x, ++L, ++H) &= ++\Delta(x, ++L, ++H) \\ \text{TypeUp}(x, -+L, ++H) &= ++\Delta(x, -+L, ++H) \\ \text{TypeUp}(x, +-L, ++H) &= -+\Delta(x, +-L, ++H) \\ \text{TypeUp}(x, --L, ++H) &= -+\Delta(x, ++L, ++H)\end{aligned}$$

$$\begin{aligned}\text{TypeUp}(x, ++L, -+H) &= ++\Delta(x, ++L, -+H) \\ \text{TypeUp}(x, -+L, -+H) &= ++\Delta(x, -+L, -+H) \\ \text{TypeUp}(x, +-L, -+H) &= -+\Delta(x, -+L, -+H) \\ \text{TypeUp}(x, --L, -+H) &= -+\Delta(x, ++L, -+H)\end{aligned}$$

$$\begin{aligned}\text{TypeUp}(x, ++L, +-H) &= +- \Delta(x, ++L, -+H) \\ \text{TypeUp}(x, -+L, +-H) &= +- \Delta(x, -+L, -+H) \\ \text{TypeUp}(x, +-L, +-H) &= -- \Delta(x, -+L, -+H) \\ \text{TypeUp}(x, --L, +-H) &= -- \Delta(x, ++L, -+H)\end{aligned}$$

$$\begin{aligned}\text{TypeUp}(x, ++L, --H) &= +- \Delta(x, ++L, ++H) \\ \text{TypeUp}(x, -+L, --H) &= +- \Delta(x, -+L, ++H) \\ \text{TypeUp}(x, +-L, --H) &= -- \Delta(x, -+L, ++H) \\ \text{TypeUp}(x, --L, --H) &= -- \Delta(x, ++L, ++H)\end{aligned}$$

La négation en $O(1)$ est conservée, par les règles suivantes :

$$\begin{aligned}\neg t &\rightarrow \neg t \\ -++ &\rightarrow -- \\ --+ &\rightarrow +- \\ -+- &\rightarrow -+ \\ --- &\rightarrow ++\end{aligned}$$

L'élimination des noeuds redondants et le partage des sous-graphes de cet arbre 4-typé conduit à une forme graphique canonique plus dense que les BDDs et les TDGs. Cependant le rapport entre la taille du BDD (respectivement du TDG) et celle du graphe 4-typé d'une fonction ne peut être supérieur à 2 (respectivement 4), puisque la forme 4-typée ne représente explicitement que des fonctions positives, au type supérieur près.

B.2 Graphe symétrisé

Une autre idée permettant de compacter le graphe de représentation est d'utiliser un type qui dénote l'échange de deux branches. Pour toute variable x , on considère la fonction S_x définie sur toute formule f par :

$$S_x(f) = f[x \leftarrow (\neg x)]$$

Suposons qu'une formule f s'écrive $\Delta(x, L, H)$ en forme de Shannon, alors la forme de Shannon de $S_x(f)$ est $\Delta(x, H, L)$. Afin de conserver la négation en $O(1)$, on va utiliser la séparation en fonctions positives et négatives, et ajouter les types “+” et “-”, tels qu'ils ont été exposés pour les TDGs. Une syntaxe abstraite contenant la *forme symétrisée* est la suivante :

$$\begin{aligned} \langle tree \rangle & ::= \langle type \rangle \langle vertex \rangle \\ \langle vertex \rangle & ::= \Delta(\langle var \rangle, \langle tree \rangle, \langle tree \rangle) \\ & ::= 1 \\ \langle type \rangle & ::= + \mid - \mid +S \mid -S \\ \langle var \rangle & ::= x_1 \mid \dots \mid x_n \end{aligned}$$

L'interprétation de cette forme exprimée avec la forme de Shannon est la suivante :

$$\begin{aligned} +\Delta(x, L, H) & = \Delta(x, L, H) \\ -\Delta(x, L, H) & = \neg\Delta(x, L, H) \\ +S\Delta(x, L, H) & = \Delta(x, H, L) \\ -S\Delta(x, L, H) & = \neg\Delta(x, H, L) \end{aligned}$$

Pour assurer la canonicité d'une forme utilisant l'opérateur S , il suffit de disposer d'un critère pour décider laquelle des deux structures $\Delta(x, L, H)$ ou $\Delta(x, H, L)$ sera prise comme référence. Soit \mathcal{R} une relation d'équivalence sur l'ensemble des fonctions \mathcal{F} , et soit $\leq_{\mathcal{R}}$ un ordre total sur \mathcal{F}/\mathcal{R} . Pour toute fonction f , on note $f_{\mathcal{R}}$ la classe d'équivalence de f relativement à \mathcal{R} . Afin d'obtenir une forme canonique, on peut par exemple imposer que la branche droite dénote une fonction H , telle que $(H_{\mathcal{R}} \leq_{\mathcal{R}} L_{\mathcal{R}})$, où L est la fonction dénotée par la branche de gauche.

Une forme canonique possédant une négation en $O(1)$ est par exemple obtenue en imposant que la branche droite dénote toujours une fonction positive, qui soit inférieure ou égale à la fonction dénotée par la branche gauche. Avec cette forme, une fonction est positive si et seulement si elle est soit de type +, soit de type $+S$ avec la branche gauche positive (notons que c'est alors une fonction de type ++), soit de type $-S$ avec la branche gauche négative (notons que c'est alors une fonction de type -+). Réciproquement, une fonction est négative si et seulement si elle soit de type -, soit de type $+S$ avec la branche gauche négative (notons que c'est alors une fonction de type +-), soit de type $-S$ avec la branche gauche positive (notons que c'est alors une fonction de type --).

Une deuxième forme canonique est obtenue en imposant qu'une fonction positive soit exclusivement dénotée soit par une structure de type +, avec une branche droite positive

inférieure ou égale à la branche gauche, soit par une structure de type $+S$, avec une branche gauche positive strictement supérieure à la branche droite. Une autre façon de formuler cette règle est d'imposer que les structures typées $+$ ou $-$ possèdent une branche droite positive inférieure ou égale à la branche gauche, et que les structures typées $+S$ ou $-S$ possèdent une branche gauche positive strictement supérieure à la branche droite. Alors par définition, une fonction est positive si et seulement si elle est de type $+$ ou $+S$. Cette forme a le bon goût être générée par un système de réécriture très concis. La forme symétrisée d'un arbre de Shannon t s'obtient en évaluant le terme $\mathbf{Type}(t)$, où la fonction \mathbf{Type} est définie par :

$$\begin{aligned}\mathbf{Type}(\Delta(x, L, H)) &= \mathbf{TypeUp}(x, \mathbf{Type}(L), \mathbf{Type}(H)) \\ \mathbf{Type}(1) &= +1 \\ \mathbf{Type}(0) &= -1\end{aligned}$$

$$(H \leq_{\mathcal{R}} L) \Rightarrow$$

$$\begin{aligned}\mathbf{TypeUp}(x, +L, +H) &= +\Delta(x, +L, +H) \\ \mathbf{TypeUp}(x, -L, +H) &= +\Delta(x, -L, +H) \\ \mathbf{TypeUp}(x, +L, -H) &= -\Delta(x, -L, +H) \\ \mathbf{TypeUp}(x, -L, -H) &= -\Delta(x, +L, +H)\end{aligned}$$

$$(L <_{\mathcal{R}} H) \Rightarrow$$

$$\begin{aligned}\mathbf{TypeUp}(x, +L, +H) &= +S\Delta(x, +H, +L) \\ \mathbf{TypeUp}(x, -L, +H) &= +S\Delta(x, +H, -L) \\ \mathbf{TypeUp}(x, +L, -H) &= -S\Delta(x, +H, -L) \\ \mathbf{TypeUp}(x, -L, -H) &= -S\Delta(x, +H, +L)\end{aligned}$$

Une troisième forme canonique est obtenue en imposant que la branche droite soit toujours positive, et que si la branche gauche est aussi positive, alors celle-ci doit être supérieure ou égale à celle de droite. Ceci revient à dire que la branche droite est positive strictement inférieure à celle de gauche (sur les graphes), en posant un ordre sur les fonctions positives et en imposant que toute fonction positive est strictement inférieure à toute fonction négative. Ainsi une fonction est positive si et seulement si elle est de type $+$ ou $+S$. Remarquons qu'avec cette forme, le type $+S$ (respectivement $-S$) est identique au type $++$ (respectivement $--$). Cette forme est aussi générée par un système de réécriture très concis. Elle est définie à partir de la fonction \mathbf{Type} suivante :

$$\begin{aligned}\mathbf{Type}(\Delta(x, L, H)) &= \mathbf{TypeUp}(x, \mathbf{Type}(L), \mathbf{Type}(H)) \\ \mathbf{Type}(1) &= +1 \\ \mathbf{Type}(0) &= -1\end{aligned}$$

$$\begin{aligned}\mathbf{TypeUp}(x, -L, +H) &= +\Delta(x, -L, +H) \\ \mathbf{TypeUp}(x, +L, -H) &= -\Delta(x, -L, +H)\end{aligned}$$

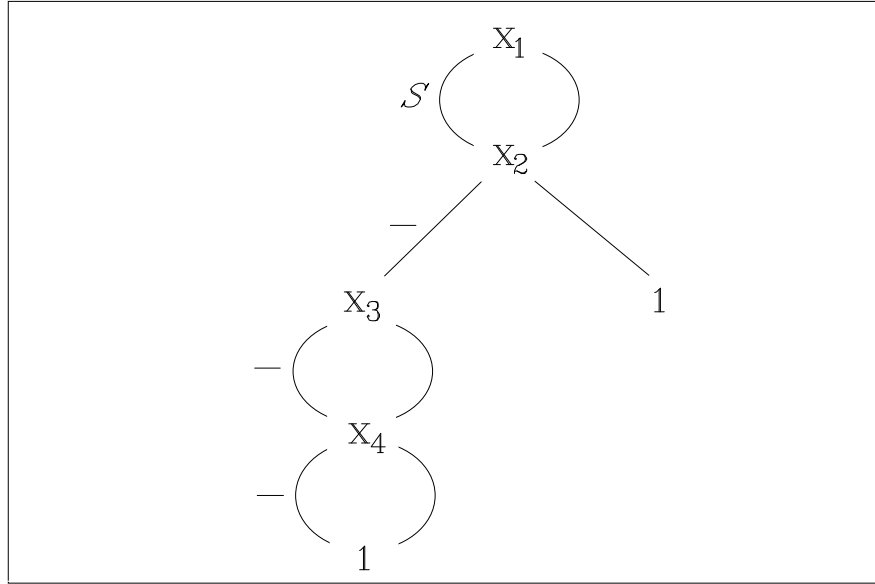


Figure 37. Graphe symétrisé de $((x_1 \wedge (x_3 \oplus x_4)) \vee (x_2 \wedge (x_3 \Leftrightarrow x_4)))$.

$$\begin{aligned}
 (H \leq_{\mathcal{R}} L) &\Rightarrow \\
 \text{TypeUp}(x, +L, +H) &= +\Delta(x, +L, +H) \\
 \text{TypeUp}(x, -L, -H) &= -\Delta(x, +L, +H)
 \end{aligned}$$

$$\begin{aligned}
 (L <_{\mathcal{R}} H) &\Rightarrow \\
 \text{TypeUp}(x, +L, +H) &= +S\Delta(x, +H, +L) \\
 \text{TypeUp}(x, -L, -H) &= -S\Delta(x, +H, +L)
 \end{aligned}$$

De cette façon, ces deux dernière formes canoniques conservent la négation en $O(1)$, car il suffit d'inverser le type supérieur d'une forme symétrisée pour obtenir sa négation. Très exactement, la négation sur la forme symétrisée est définie par :

$$\begin{aligned}
 \neg t &\rightarrow -t \\
 -- &\rightarrow + \\
 -+ &\rightarrow - \\
 -+S &\rightarrow -S \\
 --S &\rightarrow +S
 \end{aligned}$$

L'utilisation du type "S" permet de disposer d'un grand nombre de formes canoniques possédant la négation en $O(1)$: il suffit de choisir une partition \mathcal{R} , munie d'un ordre total $\leq_{\mathcal{R}}$, de l'ensemble des fonctions positives. Pour obtenir la partition la plus fine, et donc la plus grande compaction en permettant le plus de partages possibles, il suffit de définir un ordre total \leq sur les fonctions positives. Afin d'avoir une réécriture canonique efficace, il faut pouvoir déterminer efficacement si $L < H$ pour L et H deux fonctions positives. Ceci peut être réalisé en imposant l'orthogonalité de $<$, c'est à dire $\Delta(x, L, H) < \Delta(x, L', H')$

si et seulement si $H < H'$, ou $H = H'$ et $L < L'$. A partir de l'ordre total sur \mathcal{V} , on pose sur la forme de Shannon : $1 < 0$; si x est une variable, alors $0 < x$; si $x < y$, alors $\Delta(x, -, -) < \Delta(y, -, -)$; on complète cet ordre par orthogonalité. On dit alors que $L < H$, où L et H sont deux fonctions positives, si la forme de Shannon de L est strictement inférieure à celle de H . Cet ordre se teste en $O(\max(|L|, |H|))$. La Figure 37 montre le graphe symétrisé de la formule $((x_1 \wedge (x_3 \oplus x_4)) \vee (x_2 \wedge (x_3 \Leftrightarrow x_4)))$

Un ordre bien plus efficace consiste à ordonner les fonctions positives par les adresses de leur graphe en mémoire. La forme utilisée est alors canonique par construction. L'ordre se testant en $O(1)$, le typage d'une structure est immédiat.

L'élimination des noeuds redondants et le partage des sous-graphes de cette forme symétrisée conduit à une forme graphique canonique plus dense que les BDDs et les TDGs. Cependant le rapport entre la taille du BDD (respectivement du TDG) et celle du graphe 4-typé d'une fonction ne peut être supérieur à 4 (respectivement 2). Les tailles du graphe 4-typé et du graphe symétrisé d'une fonction sont en moyenne égales, puisque ces formes ne font qu'utiliser un isomorphisme de même finesse, mais portant sur deux partitions différentes des fonctions booléennes. Le gain relatif apporté par une telle forme symétrisée par rapport aux TDGs ne semble pas justifier leur utilisation pratique [101].

Annexe C

Dénotation d'ensembles par des fonctions

On étudie ici la représentation des parties de $\{0, 1\}^n$. On distinguera trois façons de dénoter un ensemble à l'aide de fonctions booléennes, à savoir : la fonction caractéristique, l'image d'une fonction, et l'image réciproque d'une fonction. On étudiera aussi comment passer d'une de ces trois représentations à une autre.

C.1 Fonction caractéristique

Soit f une fonction de $(\{0, 1\}^n \rightarrow \{0, 1\})$. On dit que f est la *fonction caractéristique* du sous-ensemble E de $\{0, 1\}^n$ défini par :

$$E = \{\vec{x} \in \{0, 1\}^n / f(\vec{x}) = 1\}$$

Réciproquement, toute partie de $\{0, 1\}^n$ est dénotée par une unique fonction caractéristique. Dans la suite, on notera χ une fonction pour laquelle on s'intéresse à l'ensemble qu'elle dénote. On confondra un ensemble et sa fonction caractéristique. On dispose d'un isomorphisme entre $(\{0, 1\}^n \rightarrow \{0, 1\})$ et l'ensemble des parties de $\{0, 1\}^n$. Plus précisément, on obtient les identités suivantes, où χ_E est la fonction caractéristique de E :

$$\begin{aligned}\chi_\emptyset &= 0 \\ \chi_{\{0,1\}^n} &= 1 \\ \chi_{E \cup F} &= \lambda \vec{x}. (\chi_E(\vec{x}) \vee \chi_F(\vec{x})) \\ \chi_{E \cap F} &= \lambda \vec{x}. (\chi_E(\vec{x}) \wedge \chi_F(\vec{x})) \\ \chi_{E \setminus F} &= \lambda \vec{x}. (\chi_E(\vec{x}) \wedge \neg \chi_F(\vec{x})) \\ \chi_{E \times F} &= \lambda [\vec{x} @ \vec{y}]. (\chi_E(\vec{x}) \wedge \chi_F(\vec{y})) \\ (\vec{x} \in E) &= \chi_E(\vec{x}) \\ (E \subseteq F) &= \lambda \vec{x}. (\chi_E(\vec{x}) \Rightarrow \chi_F(\vec{x}))\end{aligned}$$

C.2 Image d'une fonction

Une autre façon de représenter un ensemble est de considérer l'image d'une fonction booléenne de $\{0, 1\}^m$ dans $\{0, 1\}^n$. Soit \vec{f} une telle fonction. On appelle *image* de cette fonction sur $A \subseteq \{0, 1\}^m$ la partie $Img(\vec{f}, A)$ de $\{0, 1\}^n$ définie par :

$$Img(\vec{f}, A) = \{\vec{y} \in \{0, 1\}^n / \exists \vec{x} \in A, \vec{f}(\vec{x}) = \vec{y}\}$$

Tout ensemble est l'image d'une fonction sur un ensemble (éventuellement vide). Afin d'unifier ce type de représentation, on peut imposer que l'ensemble sur lequel est calculé l'image soit tout le domaine, c'est à dire 1. Dans ce cas, tout ensemble non vide est l'image d'une fonction totale sur 1. Avec ce type de représentation, les opérations ensemblistes classiques sont évidemment réalisables. Par exemple, si \vec{f} et \vec{g} représentent les ensembles $A = Img(\vec{f}, 1)$ et $B = Img(\vec{g}, 1)$, alors $A \cup B$ est l'image sur 1 de la fonction vectorielle dénotée par $((\neg x \wedge \vec{f}) \vee (x \wedge \vec{g}))$, où x est une variable qui n'apparaît ni dans \vec{f} , ni dans \vec{g} . De même, $A \times B$ est l'image de la fonction vectorielle $\lambda \vec{x}_1. \lambda \vec{x}_2. (\vec{f}(\vec{x}_1) @ \vec{g}(\vec{x}_2))$ sur 1. Cependant, la non unicité de la représentation d'un ensemble par une image fait qu'il n'existe pas de morphisme simple avec l'ensemble des parties de $\{0, 1\}^n$.

C.3 Image réciproque d'une fonction

Une troisième façon de représenter un ensemble est par l'*image réciproque* d'une fonction. L'image réciproque de la fonction \vec{f} sur $B \subseteq \{0, 1\}^n$ est la partie $Pre(\vec{f}, 1, B)$ de $\{0, 1\}^m$ définie par :

$$Pre(\vec{f}, 1, B) = \{\vec{x} \in \{0, 1\}^m / \exists \vec{y} \in B, \vec{f}(\vec{x}) = \vec{y}\}$$

Tout ensemble est l'image réciproque d'une fonction sur un ensemble. On peut même imposer que l'ensemble "générateur" soit tout le domaine d'arrivée, ce qui permet de représenter tout ensemble non vide comme l'image réciproque d'une fonction sur 1. Bien entendu, la fonction "génératrice" est en général non unique.

Annexe D

Ensemble de fonctions

D.1 Représentations d'ensembles de fonctions

Nous donnons ici les différents moyens d'expression en intention d'un ensemble de fonctions booléennes, ainsi que leur pouvoir de dénotation. On distinguera cinq façons de représenter un ensemble de fonctions, à savoir : la fonction partielle, l'intervalle, la fonction de Skolem, la fonction paramétrée, et la fonction paramétrée contrainte. On étudiera les relations qu'entretiennent entre elles ces méthodes de représentation. En particulier, on montrera que les pouvoirs de dénotation des quatre premières sont équivalentes (à l'ensemble vide près), et qu'elles ne peuvent représenter que des ensembles de fonctions très particuliers. Seule la dernière méthode permet de représenter intensivement n'importe quel ensemble de fonctions, et en cela elle est strictement plus puissante que les quatre premières.

Fonction partielle

Une fonction *partielle* f_{\perp} est dénotée par un couple de fonctions totales (D, f) , tel que :

$$f_{\perp} = \lambda \vec{x}. (\text{if } D(\vec{x}) = 1 \text{ then } f(\vec{x}) \text{ else } \perp)$$

Il est aisé de voir que tout couple (D', f') dénote cette même fonction f_{\perp} si et seulement si $D = D'$ et $(D \Rightarrow (f' \Leftrightarrow f))$ est une tautologie. On dit alors que la fonction partielle f_{\perp} représente l'ensemble des fonctions f' telle que le couple (D, f') dénote cette même fonction f_{\perp} .

Intervalle

Un *intervalle* est un couple de fonctions (g, h) . On dit qu'il dénote toutes les fonctions f' telle que $(g \Rightarrow f')$ et $(f' \Rightarrow h)$. On écrira $(g \Rightarrow f' \Rightarrow h)$ en abrégé. On dit aussi souvent que g est le *ON-set* de l'ensemble, car toutes les fonctions de l'ensemble valent 1 sur g , et que $(\neg h)$ est le *OFF-set* de l'ensemble, car toutes les fonctions de l'ensemble valent 0 sur $(\neg h)$.

Fonction de Skolem

Une fonction de Skolem, c'est à dire la solution paramétrée d'une équation à une inconnue, représente par définition un ensemble de fonctions, celui de toutes les solutions de la dite

équation. Par exemple, l'ensemble des fonctions dénotées par la fonction partielle (D, f) est aussi la fonction de Skolem de l'équation $(\forall \vec{x} \exists f' (D \Rightarrow (f' \Leftrightarrow f)))$, c'est à dire la fonction paramétrée $((D \wedge f) \vee (\neg D \wedge p))$.

Fonction paramétrée

En généralisant la définition ci-dessus, une fonction paramétrée est une des fonctions de Skolem d'une équation à plusieurs inconnues, c'est à dire une fonction $\lambda \vec{x}. \lambda \vec{p}. g(\vec{x}, \vec{p})$ ayant éventuellement plusieurs paramètres \vec{p} , tous libres de prendre n'importe quelle valeur fonctionnelle sur un espace $(\{0, 1\}^n \rightarrow \{0, 1\})$ fixé. L'ensemble de fonctions dénoté par $\lambda \vec{x}. \lambda \vec{p}. g(\vec{x}, \vec{p})$ est l'ensemble de toutes les compositions $\lambda \vec{x}. g(\vec{x}, \vec{f}(\vec{x}))$, où les fonctions f_k sont des fonctions totales quelconques de $(\{0, 1\}^n \rightarrow \{0, 1\})$.

D.2 Pouvoirs de dénotation

Les trois Théorèmes suivants montrent que les quatres techniques fonction partielle, intervalle, fonction de Skolem, et fonction paramétrée, sont équivalentes (à l'ensemble vide près).

Théorème D.1 *Les pouvoirs de dénotation d'une fonction partielle et d'un intervalle non vide¹ sont égaux.*

Preuve. Soit une fonction partielle dénotée par le couple (D, f) . Une fonction f' est "contenue" par cette fonction partielle si et seulement si $(D \Rightarrow (f' \Leftrightarrow f))$ est une tautologie, donc si et seulement si $((D \wedge f) \Rightarrow (f' \Leftrightarrow f))$ et $((D \wedge \neg f) \Rightarrow (f' \Leftrightarrow f))$, c'est à dire si et seulement si $((D \wedge f) \Rightarrow f')$ et $((D \wedge \neg f) \Rightarrow \neg f')$. Cette dernière formulation s'écrit aussi $((D \wedge f) \Rightarrow f')$ et $(f' \Rightarrow (\neg D \vee f))$. Donc f' est contenue dans l'intervalle $(D \wedge f, \neg D \vee f)$.

Réciproquement, une fonction f' est contenue dans l'intervalle (g, h) si et seulement si $(g \Rightarrow f' \Rightarrow h)$ est une tautologie. Posons $D = (g \vee \neg h)$ et $f = g$, on peut alors aisément vérifier que f' est "contenue" dans la fonction partielle dénotée par le couple (D, f) . \square

Théorème D.2 *Les pouvoirs de dénotation d'une équation à une inconnue et d'un intervalle sont égaux.*

Preuve. Les fonctions contenues dans l'intervalle (g, h) , sont toutes représentées par la fonction de Skolem de l'équation à une inconnue $(\forall \vec{x} \exists f' (g \Rightarrow f' \Rightarrow h))$. Cette équation possède une solution si et seulement si $(g \Rightarrow h)$ est une tautologie, et sa fonction de Skolem est $(g \vee (h \wedge p))$.

Réciproquement, soit l'équation à une inconnue de forme générale $(\forall \vec{x} \exists y f)$. Si une solution existe, alors $(f[y \leftarrow 0] \vee f[y \leftarrow 1])$ est une tautologie, et la fonction de Skolem est $(\neg f[y \leftarrow 0] \vee (p \wedge f[y \leftarrow 1]))$. On peut alors vérifier que dans ce cas, l'intervalle $(\neg f[y \leftarrow 0], \neg f[y \leftarrow 0]) \wedge f[y \leftarrow 1]$ représente le même ensemble. S'il n'y a pas de solution, n'importe quel intervalle vide fait l'affaire. \square

¹Un intervalle peut éventuellement dénoter l'ensemble vide, ce qui n'est pas le cas d'une fonction partielle.

Théorème D.3 *Les pouvoirs de dénotation d'une fonction paramétrée et d'un intervalle non vide sont égaux.*

Preuve. Considérons une fonction à un seul paramètre $f(\vec{x}, p)$. Il est clair que si elle peut se représenter par une fonction partielle, son domaine est l'ensemble des \vec{v} pour lesquels $f(\vec{v}, p)$ ne dépend pas de p , soit $D = \{\vec{v} \in \{0, 1\}^n / f(\vec{v}, 0) = f(\vec{v}, 1)\}$, ou encore $D = (f(\vec{x}, 0) \Leftrightarrow f(\vec{x}, 1))$, et la valeur est la fonction $f(\vec{x}, 0)$. Il reste à s'assurer que $f(\vec{x}, p)$ dénote toujours $((\neg D) \rightarrow \{0, 1\})$ sur $(\neg D)$. Ceci est le cas car pour tout élément $\vec{v} \in (\neg D)$, on a nécessairement, soit $f(\vec{v}, p) = p$, soit $f(\vec{v}, p) = \neg p$. Etant donnée une fonction g quelconque de $((\neg D) \rightarrow \{0, 1\})$, on peut toujours définir la fonction p sur $(\neg D)$ en posant $p(\vec{v}) = g(\vec{v})$ si $f(\vec{v}, p) = p$, et $p(\vec{v}) = \neg g(\vec{v})$ si $f(\vec{v}, p) = \neg p$.

Soit $f(\vec{x}, p_1, \dots, p_m)$ une fonction à plusieurs paramètres. Le même raisonnement s'applique, en définissant la fonction partielle par son domaine $D = ((\forall \vec{p} f) \Leftrightarrow (\exists \vec{p} f))$, et sa valeur $f(\vec{x}, 0, \dots, 0)$. \square

Ces quatre techniques ne peuvent pas représenter tous les ensembles. Par exemple, l'ensemble $\{\Leftrightarrow, \oplus\}$ ne peut pas être décrit par une fonction partielle, car comme les fonctions \Leftrightarrow et \oplus diffèrent pour toutes interprétations, ceci imposerait un domaine $D = 0$, soit une fonction partielle contenant toutes les fonctions. Cette remarque permet de caractériser les ensembles non vides $\{f_1, \dots, f_m\}$ qui peuvent être représentés par une fonction partielle (ou un intervalle, ou une fonction de Skolem, ou une fonction paramétrée) : il faut et il suffit que l'ensemble D (qui est par ailleurs le domaine de la fonction partielle correspondante) défini par

$$D = \left(\bigwedge_{\substack{1 \leq i \leq m \\ 1 \leq j \leq m}} (f_i \Leftrightarrow f_j) \right)$$

soit tel que l'ensemble des fonctions f_k restreintes à $(\neg D)$ est égal à $((\neg D) \rightarrow \{0, 1\})$. Il faut ainsi noter que seuls les ensembles de taille d'une puissance de 2 peuvent être éventuellement représentés par une fonction partielle. Plus précisément, un ensemble non vide de fonctions qui est représentable par une fonction partielle, est entièrement défini par la donnée de trois ensembles disjoints recouvrant $\{0, 1\}^n$: un ensemble D_0 où toute fonction prend la valeur 0, un ensemble D_1 où toute fonction prend la valeur 1, et un ensemble D_\perp tel que toute fonction de $(D_\perp \rightarrow \{0, 1\})$ est la restriction à D_\perp d'une des fonctions de l'ensemble. Une fois fixé D_\perp , il y a $2^{2^n - |D_\perp|}$ façon de choisir D_0 et D_1 . On peut donc représenter

$$\sum_{k=0}^{2^n} \binom{2^n}{k} 2^{2^n - k}$$

ensembles de fonctions, c'est à dire 3^{2^n} ensembles. Aussi sur l'espace des fonctions $(\{0, 1\}^n \rightarrow \{0, 1\})$, seules 3^{2^n} de ses parties peuvent être dénotées par une fonction partielle, ce qui est négligeable devant les $2^{2^{2^n}}$ ensembles possibles.

D.3 Fonction paramétrée contrainte

Si maintenant on s'autorise une fonction paramétrée dont les paramètres ne sont plus libres de prendre toutes valeurs fonctionnelles, mais assignés à prendre uniquement les valeurs 0 et 1, alors on obtient un pouvoir de dénotation supérieur, puisque tout ensemble non vide $E = \{f_0, \dots, f_m\}$ de fonctions de $(\{0, 1\}^n \rightarrow \{0, 1\})$ peut être représenté par une fonction

$$f(\vec{x}, p_1, \dots, p_m) = \left(f_0(\vec{x}) \wedge \left(\bigwedge_{j=1}^m \neg p_j \right) \right) \vee \left(\bigvee_{k=1}^m \left(f_k(\vec{x}) \wedge p_k \wedge \left(\bigwedge_{j=1}^{k-1} \neg p_j \right) \right) \right),$$

avec $\vec{p} \in \{0, 1\}^m$. Bien entendu, il existe une représentation paramétrée (à valeur dans $\{0, 1\}$) n'utilisant que $\lceil \log_2 |E| \rceil$ paramètres. Il suffit pour cela de considérer le codage binaire de l'indice k des fonctions f_k . Le problème de cette technique est le nombre prohibitif (potentiellement 2^{2^n}) de paramètres nécessaires pour représenter un ensemble quelconque de fonctions.

Afin de pallier cet inconvénient, on peut définir un ensemble de fonctions par une fonction paramétrée $f(\vec{x}, \vec{p})$, où le domaine des valeurs que peuvent prendre les paramètres \vec{p} en fonction de \vec{x} est donnée par une fonction caractéristique $\chi(\vec{p}, \vec{x})$. On obtient ici encore un pouvoir de dénotation permettant de représenter tout ensemble de fonctions, mais beaucoup plus efficace, car tout ensemble de fonctions peut être ainsi représenté en utilisant au plus n paramètres.

Cette dernière représentation est la meilleure, mais n'est curieusement *jamais* utilisée dans les problèmes liés à la conception de circuits, tel que la minimisation de fonction. Il faut dire qu'à notre connaissance, les problèmes pratiques de minimisation en conception de circuits semblent toujours pouvoir s'exprimer à l'aide de fonctions partielles.

Annexe E

Réduction et minimisation

E.1 Introduction

Une fonction peut être dénotée en utilisant divers systèmes de représentation : table de vérité, formule propositionnelle, somme de produits, forme de Reed-Muller, BDD, TDG, ... Pour chacun de ces systèmes de représentation, on définit un critère de taille. Le problème de la minimisation consiste à minimiser ce critère de taille dans un espace de fonctions données à priori.

On s'intéressera plus particulièrement à, d'une part, la minimisation des BDDs et TDGs, car la complexité des algorithmes de combinaison de graphes dépendent des tailles des graphes ; d'autre part, à la minimisation des sommes de produits, car de leur taille dépend directement l'efficacité (taille et vitesse du circuit) de l'implémentation d'une fonction en logique à deux niveaux [25]. Le critère de taille d'une somme de produits sera le nombre d'occurrences des littéraux. Comme il y a une dépendance linéaire entre le nombre d'occurrences de littéraux dans une somme de produits et son nombre de caractères, ce critère est satisfaisant en terme de complexité. Il est aussi satisfaisant comme critère d'implémentation efficace d'une fonction booléenne par un circuit [26, 109]. Un exposé détaillé de l'analyse de complexité (complexité dite *taille de formule* et complexité dite *combinatoire*) des fonctions booléennes peut être trouvé dans [109, 57, 125].

Un ensemble de fonctions, comme tout ensemble, peut se représenter soit en extension, soit en intention. S'il est donné en extension, la minimisation consiste en une énumération des tailles des fonctions. On n'étudiera donc que les ensembles de fonctions donnés en intention, ce qui est le cas pour les problèmes pratiques. On trouvera dans l'Annexe D la présentation et la comparaison de cinq techniques d'expression en intention d'un ensemble de fonctions booléennes. A notre connaissance, les problèmes pratiques de minimisation en conception de circuits semblent toujours pouvoir s'exprimer à l'aide de fonctions partielles. Aussi on se concentrera uniquement sur la minimisation de fonctions partielles.

E.2 Minimisation d'une fonction partielle

Rappelons qu'une fonction partielle f_{\perp} est définie par un couple de fonctions (D, f) , où D est le domaine de définition et f la valeur (voir Annexe D), par :

$$f_{\perp} = \lambda \vec{x}.(\text{if } D(\vec{x}) = 1 \text{ then } f(\vec{x}) \text{ else } \perp)$$

Il est aisé de voir que tout couple (D', f') dénote cette même fonction f_{\perp} si et seulement si $D = D'$ et $(D \Rightarrow (f' \Leftrightarrow f))$ est une tautologie. Le problème est alors de déterminer une telle fonction f' qui soit minimale.

Soit “*minimize*” une fonction qui à partir de (D, f) retourne une telle fonction minimale f' . On a les propriétés immédiates pour $D \neq 0$: si $(D \Rightarrow f)$ est une tautologie, alors $\text{minimize}(D, f) = 1$; si $(D \Rightarrow \neg f)$ est une tautologie, alors $\text{minimize}(D, f) = 0$; en particulier, $\text{minimize}(1, f) = f$, $\text{minimize}(D, D) = 1$, $\text{minimize}(D, \neg D) = 0$, $\text{minimize}(D, 1) = 1$, $\text{minimize}(D, 0) = 0$. De par l'Annexe D, on sait que le problème de la minimisation d'une fonction partielle (D, f) est équivalent à celui-ci, où $g = (D \wedge f)$ et $h = (\neg D \vee f)$:

(1) *Etant données les fonctions g et h , trouver une fonction minimale f' telle que*

$$(g \Rightarrow f' \Rightarrow h). \tag{E.1}$$

Prenons une somme de produits g quelconque, et posons $h = g$. Alors g est une tautologie si et seulement si la somme de produits minimale f' satisfaisant $(g \Rightarrow f' \Rightarrow h)$ est égale à 1. Comme déterminer si une somme de produits est une tautologie est un problème NP-complet, la minimisation est NP-difficile sur les sommes de produits. Il en est de même de la minimisation sur les graphes. Afin de minimiser une somme de produits, on se basera sur la minimisation des graphes, à partir desquels on sait générer une somme de produits, elle-même minimale. Noter que ce dernier processus est exponentiel par nature, car la Section 2.6.1 montre que l'obtention d'une somme de produits (minimale ou non) à partir d'un graphe g se fait en $O(2^{|g|})$.

Une phase préliminaire pour obtenir une minimisation est de savoir quels sont les supports minimaux de variables nécessaires pour représenter une fonction f' satisfaisant E.1. De part l'expansion de Shannon, f' satisfait la formule E.1 si et seulement si pour toute variable x on a :

$$(g[x \leftarrow 0] \Rightarrow f'[x \leftarrow 0] \Rightarrow h[x \leftarrow 0]), \quad \text{et} \tag{E.2}$$

$$(g[x \leftarrow 1] \Rightarrow f'[x \leftarrow 1] \Rightarrow h[x \leftarrow 1]). \tag{E.3}$$

Soit x_1, \dots, x_n le support des graphes g et h , c'est à dire les variables non redondantes dans g ou h . Suposons qu'il existe une fonction f' satisfaisant la formule E.1 et utilisant une variable x différente de x_1, \dots, x_n . Cette variable x est redondante dans g et h , on a donc en particulier $g[x \leftarrow 0] = g$ et $h[x \leftarrow 0] = h$, d'où l'on déduit par E.2 que la formule $(g \Rightarrow f'[x \leftarrow 0] \Rightarrow h)$ est une tautologie. Un raisonnement analogue montre que la formule $(g \Rightarrow f'[x \leftarrow 1] \Rightarrow h)$ est une tautologie. Donc les fonctions $f'[x \leftarrow 0]$ et $f'[x \leftarrow 1]$ satisfont aussi E.1, et leur taille (somme de produit ou graphe) est nécessairement strictement inférieure à celle de f' . On cherche donc les supports minimaux de f' inclus

dans $\{x_1, \dots, x_n\}$. On dit qu'une variable x_k est *inutile* si et seulement si il existe un graphe f' qui n'utilise pas x_k . Autrement dit, il existe une fonction f' pour laquelle x_k est redondante. On a alors le résultat suivant :

Théorème E.1 *L'ensemble des fonctions f' n'utilisant pas la variable x_k et satisfaisant $(g \Rightarrow f' \Rightarrow h)$ est égal à l'ensemble des fonctions f' satisfaisant $(\exists x_k g) \Rightarrow f' \Rightarrow (\forall x_k g)$.*

Preuve. Soit f' n'utilisant pas la variable x_k et satisfaisant $(g \Rightarrow f' \Rightarrow h)$. Ceci est équivalent à dire que d'une part $f'[x_k \leftarrow 0] = f'[x_k \leftarrow 1]$, puisque x_k est redondante dans f' , et que d'autre part les formules $(g[x_k \leftarrow 0] \Rightarrow f'[x_k \leftarrow 0] \Rightarrow h[x_k \leftarrow 0])$ et $(g[x_k \leftarrow 1] \Rightarrow f'[x_k \leftarrow 1] \Rightarrow h[x_k \leftarrow 1])$ sont des tautologies, de part E.2 et E.3. On a donc $(g[x_k \leftarrow 0] \Rightarrow f' \Rightarrow h[x_k \leftarrow 0])$, et $(g[x_k \leftarrow 1] \Rightarrow f' \Rightarrow h[x_k \leftarrow 1])$, ce qui est équivalent à $(\exists x_k g) \Rightarrow f' \Rightarrow (\forall x_k g)$. \square

De ce théorème, on déduit aisément le suivant :

Théorème E.2 *L'ensemble des fonctions f' satisfaisant $(g \Rightarrow f' \Rightarrow h)$ et n'utilisant pas les variables \vec{y} est non vide si et seulement si $(\exists \vec{y} g) \Rightarrow (\forall \vec{y} g)$ est une tautologie. Dans ce cas, cet ensemble est égal à l'ensemble des fonctions f' satisfaisant $(\exists \vec{y} g) \Rightarrow f' \Rightarrow (\forall \vec{y} g)$.*

Ce résultat permet de calculer les supports minimaux de f' , ainsi que les fonctions g et h définissant la fonction partielle f' correspondante. Bien entendu, il existe plusieurs supports minimaux, mais tous ne sont pas optimaux. Par exemple, si le support de g et h est $\{x_1, \dots, x_4\}$, on peut avoir les supports localement minimaux $\{x_1, x_2\}$ et $\{x_2, x_3, x_4\}$. Afin d'avoir une solution optimale (c'est à dire le maximum de variables inutiles), un *branch and bound* est nécessaire. Ceci implique un coût exponentiel, qui peut donc être rapidement rédhibitoire.

Une fois les variables inutiles éliminées, le problème de la minimisation est réduit à un certain nombre —un par support minimal— de problèmes qui se formulent comme suit :

(2) *Etant donnés deux graphes g et h , trouver un graphe f' tel que*

$$(g \Rightarrow f' \Rightarrow h). \tag{E.4}$$

Il est ici entendu que le support de f' est nécessairement inclus dans celui de g et h , c'est à dire qu'il n'y pas de variable inutile.

Si $(g \Rightarrow h)$ est une tautologie, alors l'ensemble des fonctions f' satisfaisant $(g \Rightarrow f' \Rightarrow h)$ est non vide, et il s'écrit $(g \vee (h \wedge p))$, où p est un paramètre. On revient donc au problème ci-dessous, et qui fait l'objet de la section suivante :

(3) *Calculer le graphe minimal dans l'ensemble de graphes représenté par la formule paramétrée $f(\vec{x}, p)$.*

E.3 Minimisation d'une fonction paramétrée

Afin de réduire le graphe d'une fonction paramétrée $f(\vec{x}, p)$, on va utiliser des règles basées sur l'unification, où seule la variable p est quantifiée existentiellement. Si le graphe peut s'unifier avec une constante, alors c'est la solution minimale. Sinon, on tente de partager des sous-graphes du graphes. Ces règles sur les TDGs (la règle E.8 est omise pour les BDDs) s'écrivent :

$$\begin{aligned} (\forall \vec{x} \exists p f) &\Rightarrow \\ f &\rightarrow 1 \end{aligned} \tag{E.5}$$

$$\begin{aligned} (\forall \vec{x} \exists p \neg f) &\Rightarrow \\ f &\rightarrow 0 \end{aligned} \tag{E.6}$$

$$\begin{aligned} (s \text{ est la fonction de Skolem de } (\forall \vec{x} \exists p (L \Leftrightarrow H))) &\Rightarrow \\ \Delta(-, L, H) &\rightarrow H[p \leftarrow s] \end{aligned} \tag{E.7}$$

$$\begin{aligned} (s \text{ est la fonction de Skolem de } (\forall \vec{x} \exists p (L \oplus H))) &\Rightarrow \\ \Delta(x, L, H) &\rightarrow \Delta(x, -H[p \leftarrow s], H[p \leftarrow s]) \end{aligned} \tag{E.8}$$

Ce système de réécriture, utilisé conjointement avec l'élimination des variables inutiles, donne de bon résultats. Cependant, sa complexité est exponentielle, et cette technique ne peut être donc utilisée avec profit que lors d'une phase de minimisation complète d'un ensemble de fonctions, par exemple les fonctions de transitions et de sorties d'une machine séquentielle, en vue d'assurer une implémentation non redondante et compacte. Pourtant la minimisation est un problème qui permet d'accélérer considérablement l'évaluation de certaines opérations, par exemple les compositions qui interviennent lors de la vérification de formules CTL (voir Section 5.3.1). La section suivante présente un opérateur polynomial dévolu à cette tâche.

E.4 Réduction polynomiale : l'opérateur "Restrict"

Nous présentons ici un opérateur très simple, appelé "restrict", qui permet de réduire une fonction f sur un domaine d'utilisation D . Si son pouvoir de réduction est moins puissant (écarts de 0% à 60%) que celui de la technique présentée précédemment, "restrict" a l'avantage de s'évaluer quadratiquement, alors que la première technique est exponentielle.

La définition algorithmique de "restrict", noté " \Downarrow ", est donnée Figure 38. Cette fonction opère sur les graphes ou les formes de Shannon des fonctions f et D , et retourne le graphe de $(f \Downarrow D)$. A partir de cette définition algorithmique, on déduit les deux théorèmes suivants, montrant que "restrict" permet de réduire une fonction f sur un domaine d'utilisation D .

Bibliographie

- [1] M. Abadi, L. Lamport, “Composing Specifications”, DEC report, 1990. Also in “Stewise Refinement of Distributed Systems: Models, Formalism, Correctness”, Vol. 430 of *Lectures Notes in Computer Science*, Springer-Verlag, Berlin, 1990.
- [2] L. Albert, “Présentation et Evaluation de la Complexité en Moyenne des Algorithmes d’Unification”, *Rapport de recherche INRIA N° 1212*, avril 1990.
- [3] L. Albert, “Average Case Complexity Analysis of RETE Pattern-Matching Algorithm and Average Size of Join in Databases”, *9th Conference on Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science N°405, Springer-Verlag, 1989.
- [4] S. B. Akers, “Binary Decision Diagrams”, *IEEE Transactions on Computers*, Vol C-27, N°6, 1978.
- [5] F. Anceau, “Design Methodology for Large Custom Processors”, in Proc. of *the 1986 ESSIR Conference*, Delft, USA, 1986.
- [6] F. Anceau, C. Berthet, J. C. Madre, O. Coudert, J. P. Billon, “La Preuve Formelle en Milieu Industriel”, in TSI, Vol. 8, N°6, 1989.
- [7] J. R. Armstrong, *Chip-Level Modeling With VHDL*, Prentice Hall, 1989.
- [8] P. Ashar, A. Ghosh, S. Devadas, A. R. Newton, “Combinational and Sequential Logic Verification Using General Binary Decision Diagrams”, in Proc. of *MCNC workshop*, North Carolina, USA, mai 1991.
- [9] H. G. Barrow, “Verify: A Program for Proving Correctness of Digital Hardware Designs”, *Artificial Intelligence 24*, 1984.
- [10] C. Bayol, J. L. Paillet, “Using Tache for Proving Circuits”, in *Formal VLSI Correctness Verification*, L.J.M. Claesen Editor, North-Holland, pp. 83–87, novembre 1989.
- [11] Beckman F.S., *Mathematical Foundations of Programming*, Addison-Wesley, 1980.
- [12] C. L. Berman, “Circuit Width, Register Allocation, and Reduced Function Graphs”, *IBM Research Report RC 14129*, 1989.
- [13] C. L. Berman, “Ordered Binary Decision Diagrams and Circuit Structure”, in Proc. of *ICCD’89*, USA, septembre 1989.

- [14] G. Berry, "A Hardware Implementation of ESTEREL", in Proc. of *1991 International Workshop on Formal Methods in VLSI Design*, Miami FL, USA, janvier 1991.
- [15] G. Berry, "ESTEREL on Hardware", in Proc. of *The Royal Society Discussion Meeting on Mechanized Reasoning and Hardware Design*, London, UK, octobre 1991.
- [16] G. Berry *et al.*, "The ESTEREL Language", in *IEEE Special Issue on Synchronous Languages*, septembre 1992.
- [17] C. Berthet, O. Coudert, J. C. Madre, "New Ideas on Symbolic Manipulations of Finite State Machines", in Proc. of *ICCD'90*, Cambridge MA, USA, septembre 1990.
- [18] J. P. Billon, "Perfect Normal Forms for Discrete Functions", *BULL Research Report N°87019*, juin 1987.
- [19] J. P. Billon, "Symbolic Execution of Discrete Programs", *BULL Research Report N°87039*, Septembre 1987.
- [20] D. Borrione, J. L. Paillet, L. Pierre, H. Collavizza, "Modélisation Fonctionnelle et Preuve de Circuits Digitaux", TSI, Vol. 8, N°6, pp. 523–544, 1989.
- [21] S. Bose, A. Fisher, "Automatic Verification of Synchronous Circuits Using Symbolic Logic Simulation and Temporal Logic", in *Formal VLSI Correctness Verification*, L.J.M. Claesen Editor, North-Holland, pp. 151–158, novembre 1989.
- [22] A. Bouajjani, J. C. Fernandez, N. Halbwachs, "Verification of Safety Properties", avril 1990.
- [23] A. Bouajjani, J. C. Fernandez, N. Halbwachs, "Minimal Model Generation", in *Computer-Aided Verification'90*, E. M. Clarke and R. P. Kurshan Editors, DIMACS Series, pp. 85–91, juin 1990.
- [24] A. Boudet, J. P. Jouannaud, "Unification in Boolean Rings and Abelian Groups", *special issue of JSC*, mai 1988.
- [25] R. K. Brayton, G. D. Hachtel, C. T. McMullen, A. L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, 1984.
- [26] F. M. Brown, *Boolean Reasoning*, Kluwer Academic Publishers, 1990.
- [27] R. E. Bryant, "Graph-Based Algorithms for Boolean Functions Manipulation", *IEEE Transactions on Computers*, Vol C35, N°8, pp. 677–692, août 1986.
- [28] R. E. Bryant, "On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication", Carnegie Mellon University Research Report, septembre 1988.
- [29] S. Burch, E. M. Clarke, K. L. McMillan, "Symbolic Model Checking: 10^{20} States and Beyond", in Proc. of *LICS*, 1990.
- [30] S. Burch, E. M. Clarke, K. L. McMillan, "Sequential Circuit Verification Using Symbolic Model Checking", in Proc. of *27th DAC*, Orlando FL, USA, juillet 1990.

- [31] S. Burch, "Using BDDs to Verify Multipliers", in Proc. of the *1991 International Workshop on Formal Methods in VLSI Design*, Miami FL, USA, janvier 1991.
- [32] P. Camurati, P. Prinetto, "Formal Verification of Hardware Correctness: Introduction and Survey of Current Research", *IEEE Transactions on Computer*, juillet 1988.
- [33] R. Casas, M. I. Fernández-Camacho, J. M. Steyaert, "Algebraic Simplification in Computer Algebra: An Analysis of Bottom-Up Algorithms", *Theoretical Computer Science, Vol 74*, pp. 273–298, North-Holland, 1990.
- [34] E. Cerny, M. A. Marin, "A Computer Algorithm for the Synthesis of Memoryless Switching Circuits", *IEEE Transactions on Computer*, C-23, mai 1974.
- [35] M. S. Chandrasekhar, J. P. Privitera, K. W. Conradt, "Application of Term Rewriting Techniques to Hardware Design Verification", in Proc. of the *24th DAC*, juillet 1987.
- [36] C.-L. Chang, R. C.-T. Lee, *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, 1973.
- [37] C. Charniak, K. Riesbeck, D. V. McDermott, *Artificial Intelligence Programming*, LEA Publishers 1980.
- [38] H. Cho, G. Hachtel, S. W. Jeong, B. Plessier, E. Schwarz, F. Somenzi, "ATPG Aspect of FSM Verification", in Proc. of *ICCAD'90*, Santa Clara CA, USA, novembre 1990.
- [39] C. Choppy, S. Kaplan, M. Soria, "Complexity Analysis of Term-Rewriting Systems", *Theoretical Computer Science N°67*, North Holland, pp. 261–282, 1989.
- [40] E. M. Clarke, O. Grumbreg, "Research on Automatic Verification of Finite-State Concurrent Systems", *Annual Revue Computing Science*, vol. 2, pp. 269–290, 1987.
- [41] D. R. Coelho, *The VHDL Handbook*, Kluwer Academic Publishers, 1989.
- [42] A. Cohn, "A Proof Of Correctness of the Viper Microprocessor: The First Level", *Technical Report N°104*, University of Cambridge, England, janvier 1987.
- [43] O. Coudert, C. Berthet, J. C. Madre, "Verification of Synchronous Sequential Machines Based on Symbolic Execution", in *Lecture Notes in Computer Science: Automatic Verification Methods for Finite State Systems*, Volume 407, J. Sifakis Editor, Springer-Verlag, pp. 365–373, juin 1989.
- [44] O. Coudert, C. Berthet, J. C. Madre, "Verification of Sequential Machines using Boolean Functional Vectors", in *Formal VLSI Correctness Verification*, L.J.M. Claesen Editor, North-Holland, pp. 179–196, novembre 1989.
- [45] O. Coudert, C. Berthet, J. C. Madre, "Symbolic Manipulations for the Verification of Sequential Machines", in Proc. of the *1st EDAC*, Glasgow, UK, mars 1990.
- [46] O. Coudert, J. C. Madre, "Verifying Temporal Properties of Sequential Machines Without Building their State Diagrams", in *Computer-Aided Verification'90*, E. M. Clarke and R. P Kurshan Editors, DIMACS Series, pp. 75–84, juin 1990.

- [47] O. Coudert, J. C. Madre, "A Unified Framework for the Formal Verification of Sequential Circuits", in Proc. of *ICCAD'90*, Santa Clara CA, USA, novembre 1990.
- [48] O. Coudert, J. C. Madre, "Symbolic Computation of the Valid States of a Sequential Machine: Algorithms and Discussion", in Proc. of the *International Workshop on Formal Methods in VLSI Design*, Miami, USA, janvier 1991.
- [49] O. Coudert, J. C. Madre, "A New Method to Compute Prime and Essential Prime Implicants of Boolean Functions", in Proc. of the *MIT VLSI Conference*, Cambridge MA, USA, mars 1992.
- [50] O. Coudert, J. C. Madre, "Implicit and Incremental Computation of Primes and Essential Primes of Boolean functions", in Proc. of the *29th DAC*, Anaheim CA, USA, Juin 1992.
- [51] O. Coudert, J. C. Madre, "A New Implicit DAG Based Prime and Essential Prime Computation Technique", in Proc. of the *International Symposium on Information Sciences*, Fukuoka, Japon, juillet 1992.
- [52] J. A. Darringer, "The Application of Program Verification Techniques to Hardware Verification", in Proc. of the *16th DAC*, 1979.
- [53] A. Debreil, C. Berthet, A. Jerraya, "Symbolic Computation of VHDL Hierarchical Descriptions", in Proc. of the *First European Conference on VHDL Methods*, Marseille, France, septembre 1990.
- [54] J. P. Delahaye, *Outils Logiques pour l'Intelligence Artificielle*, Editions Eyrolles, 1986.
- [55] S. Devadas, H. K. Ma, and R. Newton, "On the Verification of Sequential Machines at Differing Levels of Abstraction", *IEEE Transactions on CAD*, Vol. 7, No. 6, juin 1988.
- [56] D. L. Dill, *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*, PhD Thesis, Carnegie Mellon University, 1988.
- [57] P. E. Dunne, *The Complexity of Boolean Networks*, APIC Series N°29, Academic Press, 1988.
- [58] D. Dietmeyer, *Logic Design of Digital Systems*, Allyn and Bacon, 2nd edition, 1978.
- [59] E. A. Emerson, "Temporal and Modal Logic", *Formal Models and Semantics*, Handbook of Theoretical Computer Science, Jan van Leeuwen Editor, Elsevier, pp. 995–1072, 1990.
- [60] R. Enders, T. Filkorn, D. Taubner, "Generating BDDs for Symbolic Model Checking in CCS", in Proc. of *Computer-Aided Verification'92*, Montreal, 1992.
- [61] P. Flajolet, J. M. Steyaert, "A Complexity Calculus for Recursive Tree Algorithms", *Mathematical Systems Theory*, Vol 19, pp. 301–331, Springer-Verlag, 1987.

- [62] P. Flajolet, P. Sipala, J. M. Steyaert, “Analytic Variations on the Common Subexpression Problem”, *Rapport de Recherche INRIA N°1210*, avril 1990.
- [63] S. J. Friedman, K. J. Supowit, “Finding the Optimal Variable Ordering for Binary Decision Diagrams”, *IEEE Transactions on Computer*, Vol C-39, N° 5, pp. 710–713, mai 1990.
- [64] M. Fujita, H. Fujisawa, N. Kawato, “Evaluation and Improvements of a Boolean Comparison Method Based on Binary Decision Diagrams”, in Proc. of *ICCAD’88*, Santa Clara, USA, pp. 2–5, novembre 1988.
- [65] J. H. Gallier, *Logic for Computer Science – Foundations of Automated Theorem Proving*, Harper and Row Publishers, New York, 1986.
- [66] A. Ghosh, S. Devadas, A. R. Newton, “Test Generation for Highly Sequential Circuits”, in Proc. of *ICCAD’89*, Santa-Clara CA, USA, novembre 1989.
- [67] M. J. C. Gordon, “LCF-LSM”, *Technical Report N°41*, Computer Laboratory, University of Cambridge, England, 1983.
- [68] M. J. C. Gordon, “Proving a Computer Correct”, *Technical Report N°42*, Computer Laboratory, University of Cambridge, England, 1984.
- [69] I. P. Goulden, D. M. Jackson, *Combinatorial Enumeration*, Wiley-Interscience Series in Discrete Mathematics, J. Wiley and Sons, 1983.
- [70] A. Greiner, F. Gourdy, R. Marbot, J. Y. Murzin, M. Guillemet, “Noisy : An Electrical Noise Checker for VLSI Circuits”, in Proc. of *ICCAD’88*, Santa Clara, USA, novembre 1988.
- [71] N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud, “The Synchronous Data-Flow Programming Language LUSTRE”, in *IEEE Special Issue on Synchronous Languages*, pp. 1305–1320, septembre 1992.
- [72] G. Huet, “Résolution d’Equations dans les Langages d’Ordre 1, 2, ..., ω ”, *Thèse d’Etat*, Université Paris 7, 1976.
- [73] G. Huet, D. C. Oppen, “Equations and Rewrite Rules: A Survey”, *Academic Press Inc.*, 1980.
- [74] G. J. Holtzman, “Algorithms for Automated Protocol Validation”, in Proc. of the *Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, juin 1989.
- [75] J. E. Hopcroft, J. D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, Reading, Mass., 1979.
- [76] J. Hsiang, *Topics in Automated Theorem Proving and Program Generation*, PhD Thesis, University of Urbana Champaign, Illinois, 1983.
- [77] J. Hsiang, “Refutational Theorem Proving Using Term Rewriting Systems”, *Artificial Intelligence N°25*, pp. 255–300, 1985.

- [78] J. Hsiang, "Rewrite Methods for Theorem Proving in First Order Theory with Equality", *Journal of Symbolic Computation* N°3, pp. 133–151, 1987.
- [79] W. A. Hunt, "FM8501: A Verified Microprocessor", in Proc. of the *IFIP WG 10.2 Workshop: From HDL Descriptions to Guaranteed Correct Circuit Designs*, North Holland Publishing, 1986.
- [80] S. H. Hwang, A. R. Newton, "An efficient Design Correctness Checker of Finite State Machines", in Proc. of *ICCAD'87*, Santa Clara, USA, novembre 1987.
- [81] S. W. Jeong, B. Plessier, G. D. Hachtel, F. Somenzi, "Variable Ordering for FSM Traversal", in Proc. of the *MCNC Workshop*, North Carolina, USA, mai 1991.
- [82] S. C. Kleene, *Mathematical Logic*, John Wiley and Sons, NY, 1967
- [83] S. Kohavi, *Switching and Finite Automata Theory*, McGraw-Hill, New York, 1978.
- [84] R. P. Kurshan, "Analysis of discrete Event Coordination", in *Lecture Notes in Computer Science*, Springer-Verlag, pp. 414–453, 1990.
- [85] B. Lin, H. J. Touati, A. R. Newton, "Don't Care Minimization of Multi-Level Sequential Logic Networks", in Proc. of *ICCAD'90*, Santa-Clara CA, USA, novembre 1990.
- [86] B. Lin, "Efficient Symbolic Manipulation of Equivalence Relations and Classes", in Proc. of the *International Workshop on Logic Synthesis*, MCNC North Carolina, USA, mai 1991.
- [87] R. Lipsett, C. Schaefer, C. Ussery, *VHDL: Hardware Description and Design*, Kluwer Academic Publishers, 1989.
- [88] K. L. McMillan, J. Schwalbe, "Formal Verification of the Encore Gigamax Cache Consistency Protocol", in *International Symposium on Shared Memory Multiprocessors*, 1991.
- [89] J. C. Madre, J. P. Billon, "Proving Circuit Correctness using Formal Comparison Between Expected and Extracted Behaviour", in Proc. of the *25th DAC*, Anaheim CA, USA, juillet 1988.
- [90] J. C. Madre, J. P. Billon, O. Coudert, M. Currat, "Applying an Automated Theorem Proving Technique to Hardware Verification", in *Progress in Computer Aided Design*, Volume 5, Ablex Publishing, USA, 1990.
- [91] J. C. Madre, O. Coudert, M. Currat, A. Debreil, C. Berthet, "The Formal Verification Chain at BULL", in Proc. of the *EUROASIC Conference*, Paris, France, juin 1990.
- [92] J. C. Madre, O. Coudert, "Automating the Diagnosis and the Rectification of Design Errors with PRIAM", in Proc. of *ICCAD'89*, Santa Clara, USA, novembre 1989.

- [93] J. C. Madre, O. Coudert, “Formal Verification of Digital Circuits Using a Propositional Theorem Prover”, in Proc. of the *IFIP Working Conference on the CAD Systems Using AI Techniques*, Tokyo, juin 1989.
- [94] J. C. Madre, “PRIAM, Un Outil de Preuve Formelle de Circuits Digitaux”, Thèse de troisième cycle, Ecole Nationale Supérieure des Télécommunications, juin 1990.
- [95] J. C. Madre, O. Coudert, “A Logically Complete Reasoning Maintenance System Based on a Logical Constraint Solver”, in Proc. of the *International Joint Conference on Artificial Intelligence (IJCAI)*, Sydney, Australia, août 1991.
- [96] S. Malik, A. R. Wang, R. K. Brayton, A. Sangiovanni-Vincentelli, “Logic Verification using Binary Decision Diagrams in a Logic Synthesis Environment”, in Proc. of *ICCAD’88*, Santa Clara, USA, pp. 6–9, novembre 1988.
- [97] T. F. Melham, “Abstraction Mechanisms for Hardware Verification”, in *VLSI Specification, Verification and Synthesis*, G. Birtwistle and P. A. Subrahmanyam Editors, Kluwer Academic Publishers, 1988.
- [98] E. Mendelson, *Introduction to Mathematical Logic*, Wadsworth and Brooks, 1987.
- [99] G. J. Milne, “The Correctness of a Simple Silicon Compiler”, *Internal Report CSR-127-83*, Computer Science Department, University of Edinburgh, Scotland, UK, 1983.
- [100] S. Minato, N. Ishiura, S. Yajima, “Fast Tautology Checking Using Shared Binary Decision Diagrams - Experimental Results”, in *Formal VLSI Correctness Verification*, L.J.M. Claesen Editor, North-Holland, pp. 107–111, novembre 1989.
- [101] S. Minato, N. Ishiura, S. Yajima, “Shared Binary Decision Diagrams with Attributed Edges for Efficient Boolean Function Manipulation”, in Proc. of the *27th DAC*, Las Vegas NA, USA, pp. 52–57, juin 1990.
- [102] B. M. E. Moret, “Decision Trees and Diagrams”, *Computing Surveys*, Vol 14, N°14, 1982.
- [103] V. Pitchumani, E. P. Stabler, “A Formal Method for Computer Design Verification”, in Proc. of the *19th DAC*, juillet 1982.
- [104] C. Pixley, “A Computational Theory and Implementation of Sequential Hardware Equivalence”, in *Computer-Aided Verification’90*, E. M. Clarke and R. P. Kurshan Editors, DIMACS Series, pp. 293–320, juin 1990.
- [105] J. P. Queille, J. Sifakis, “Fairness and Related Properties in Transition Systems”, *Acta Informatica*, pp. 195–220, 1983.
- [106] C. Ratel, LGI Grenoble, communication privée, 1991.
- [107] R. Reiter, “A Theory of Diagnosis from First Principles”, *Artificial Intelligence N° 32*, Elsevier Science Publishers, 1987.

- [108] D. E. Ross, K. M. Butler, R. Kapur, M. R. Mercer, “Fast Functional Evaluation of Candidate OBDD Variable Orderings”, in Proc. of *2nd EDAC*, pp. 4–10, février 1991.
- [109] J. S. Savage, *The Complexity of Computing*, Wiley-Interscience Publication, 1976.
- [110] H. Savoj, R. K. Brayton, H. Touati, “Extracting Local Don’t Cares for Network Optimization”, in Proc. of *ICCAD’91*, Santa Clara CA, USA, novembre 1991.
- [111] D. A. Schmitt, *A Methodology for Language Development*, Allyn Bacon Inc., 1986.
- [112] H. Simonis, M. Dincbas, “Using Logic Programming for Fault Diagnosis in Digital Circuits”, *ECRC Technical Report TR-LP-18*, 1986.
- [113] H. Simonis, M. Dincbas, “Using an Extended Prolog for Digital Circuit Design”, *ECRC Technical Report TR-LP-22*, 1987.
- [114] V. Stavridou, H. Barringer and D. A. Edwards, “Formal Specification and Verification of Hardware - A Comparative Case Study”, in Proc. of the *25th DAC*, Anaheim CA, USA, juillet 1988.
- [115] J. M. Steyaert, P. Flajolet, “Patterns and Pattern-Matching in Trees: An Analysis”, *Information and Control, Vol 58, Nos 1–3*, pp 19–58, Academic Press, 1983.
- [116] R. S. Stoll, *Set Theory and Logic*, Dover Publications Inc., 1986.
- [117] M. Stone, “The Theory of Representations for Boolean Algebra”, *Transactions AMS Vol. 40*, pp. 37–111, 1936.
- [118] K. J. Supowit, S. J. Friedman, “A new Method for Verifying Sequential Circuits”, in Proc. of the *23rd DAC*, 1986.
- [119] E. Tiden, “Symbolic Verification of Switch-Level Circuits using a Prolog Enhanced with Unification in Finite Algebra”, in *The Fusion of Hardware Design And Verification*, North Holland, 1988.
- [120] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, A. Sangiovanni-Vincentelli, “Implicit State Enumeration of Finite State Machines using BDD’s”, in Proc. of *ICCAD’90*, Santa Clara CA, USA, novembre 1990.
- [121] F. Van Aelten, J. Allen, S. Devadas, “Verification of Relations between Synchronous Machines”, in Proc. of *ICCAD’91*, Santa Clara CA, USA, novembre 1991.
- [122] J. S. Vitter, P. Flajolet, “Average-Case Analysis of Algorithms and Data Structures”, Algorithms and Complexity, Handbook of Theoretical Computer Science, Vol A, Chapter 9, pp. 433–524, Elsevier Science, 1990.
- [123] F. Vlach, “A Note on the INSTEP Tautology Checker and the IFIP and ISCAS Benchmarks”, in *Formal VLSI Correctness Verification*, L.J.M. Claesen Editor, North-Holland, pp. 89–93, novembre 1989.

- [124] T. J. Wagner, "Verification of Hardware Designs Through Symbolic Manipulation", *International Symposium on Design Automation and Microprocessors*, 1977.
- [125] I. Wegener, *The Complexity of Boolean Functions*, Wiley-Teubner Series in Computer Science, Stuttgart, 1987
- [126] R. Wei, A. L. Sangiovanni-Vincentelli, "PROTEUS: A Logic Verification System for Combinational Circuits", Proc. of the *1986 International Test Conference*, 1986.
- [127] A. Yasuhara, *Recursive Function Theory and Logic*, Academic Press, NY, 1971.
- [128] Zvi Har'El, R. P. Kurshan, "Software for analytical Development of Communications Protocols", in *AT&T Technical Journal*, février 1990.

Index

- \rightarrow (réécriture), 41, 122
- $[\cdot \rightarrow \cdot]$ (substitution), 41, 122
- σ (substitution), 23, 121
- ϱ (substitution), 23, 121
- $x, x_1, \dots, y, y_1, \dots$ (variables), 17
- \vec{x} (vecteur de variables), 20
- ε (fonction identité ou négation), 19, 83
- \neg (négation), 17
- \vee (disjonction), 17
- \wedge (conjonction), 17
- \Rightarrow (implication), 17
- \Leftrightarrow (équivalence), 17
- \oplus (ou exclusif), 17
- \star (opérateur booléen diadique), 29
- \exists (quantificateur existentiel), 17
- \forall (quantificateur universel), 17
- \exists -élimination, 20
- \forall -élimination, 20
- Q (quantificateur), 20
- i (interprétation), 18
- $=_{\mathcal{B}}$ (égalité booléenne), 18
- $O(\cdot)$, 19
- $o(\cdot)$, 40
- $\Theta(\cdot)$, 36
- @ (concaténation vectorielle), 20
- \perp (indéfini), 25
- f_{\perp} (fonction partielle), 131
- (D, f) (fonction partielle), 131
- Δ (noeud), 30
- L (branche gauche), 30
- H (branche droite), 30
- \preceq , 32
- $|\cdot|$ (taille), 32
- \mathcal{A}_n , 36
- Π , 37
- Σ , 33
- $\binom{\cdot}{\cdot}$, 38
- \sim , 38
- \approx , 36
- \ll , 38
- \lim , 41
- $+$, 43
- $-$, 43
- $++$, 123
- $-+$, 123
- $+-$, 123
- $--$, 123
- $+S$, 125
- $-S$, 125
- ifn**, 48
- π (permutation), 57
- \mathcal{M} (machine séquentielle), 70
- ω (fonction de sortie), 70
- δ (fonction de transition), 70
- (Cns, \vec{f}) (fonction de transition), 70
- Cns , 70
- \vec{f} (fonction vectorielle), 70
- Init* (états initiaux), 70
- Δ (relation de transition), 70, 109
- Λ (relation de sortie), 70
- $*$ (Kleene's star), 72
- Img* (image), 74
- Pre* (image réciproque), 74
- $AX(\cdot)$, 75
- $EX(\cdot)$, 75
- $A[U.]$, 75
- $E[U.]$, 75
- $AF(\cdot)$, 75
- $EF(\cdot)$, 75
- $AG(\cdot)$, 75
- $EG(\cdot)$, 75
- PPI* (*plus proche interpretation*), 81, 93, 95, 97
- rr** (resticteur d'image), 87, 90
- srr** (resticteur d'image strict), 91, 97
- d (hyper-distance), 95
- $\leq_{\vec{x}}$, 96
- \downarrow (constrain), 97

- \Downarrow (restrict), 111, 138
 - \mathcal{V} , 121
 - \mathcal{F}_k , 121
 - \mathcal{T} , 121
 - t (arbre), 121
 - \mathcal{R} , 122
 - \hat{t} , 122
 - \mathcal{F}/\mathcal{R} , 125
 - max, 128
- antilogie, 18
- arbre
 - de Shannon, 30
 - réduit de Shannon, 31
 - typé de Shannon, 43
- BDD (*Binary Decision Diagram*), 28, 39
- between, 100
- choose-index, 99
- composition, 50
 - close, 54
 - complexité, 51, 84, 107
 - évaluation de la composition, 109, 111, 112
- constrain (\Downarrow), 93, 97
- CTL (*Computational Tree Logic*), 74
- couverture, 87
- DAC (*Design Automation Conference*), 142
- DAG (*Directed Acyclic Graph*), 31
- eliminate-and-partition, 88
- élimination
 - existentielle (\exists -élimination), 20
 - des noeuds redondants, 31
 - universelle (\forall -élimination), 20
 - des variables inutiles, 137
 - des variables redondantes, 80
- ensemble 73, 129, 131
- état valide, 71, 73
- EDAC (*European Design Automation Conference*), 143
- exécution symbolique, 70
- expansion de Shannon, 28
- fonction
 - booléenne, 18
 - caractéristique, 73, 129
 - de Skolem, 131
 - de sortie (ω), 70
 - de transition (δ), 70
 - ensemble de fonctions, 23, 131
 - génératrice, 36
 - minimisation, 138
 - négative, 43
 - paramétrée, 132
 - positive, 43
 - symétrique, 33, 57
- forme
 - canonique, 122
 - compatible, 122
 - de Reed-Muller, 61, 64
 - de Shannon, 28, 30, 31, 43
 - normale, 122
 - prénexe, 20
 - sans quantificateur, 19, 54, 61, 63
- formule
 - antilogie, 18
 - close, 18
 - d'état, 75
 - propositionnelle quantifiée, 17
 - satisfiable, 21
 - somme exclusive de produits, 61
 - somme de produits, 59
 - tautologie, 18
- frontier set minimization* (restrict), 111, 138
- graphe
 - 4-Typé, 123
 - construction 47, 48, 49
 - de décision binaire, 28, 39
 - de décision typé, 28
 - d'états, 70
 - symétrisé, 125
 - taille 39, 41
- hyper-distance (d), 95
- ICCAD (*International Conference on Computer Aided Design*), 143
- ICCD (*International Conference on Computer Design*), 141

- identification, 88
 - étendue, 89
- image (*Img*), 73, 82, 130
- image réciproque (*Pre*), 78, 107, 130
- interprétation (i, \bar{x}), 18
- intervalle, 131
- langage
 - accepté, 71
 - généré, 71
- libre (variable), 18
- liée (variable), 19
- logique temporelle, 74
 - CTL, 75, 77, 78
 - propriété de sûreté, 76, 115
- machine (\mathcal{M})
 - de Mealy, 70
 - de Moore, 70
 - machines équivalentes, 72
 - minimisation, 70
 - modèle universel, 76
 - séquentielle, 72
- minimisation
 - de fonctions, 138
 - de machines, 70
 - logique, 79, 110
- model checking*, 74
- modèle universel, 76
- noeud (Δ), 30
 - redondant, 31
- occurrence, 18
- OFF-set, 131
- ON-set, 131
- ordre des variables, 20, 32, 39, 57, 104, 115
- orthogonalité, 29
- paramètre, 23, 132, 134
- partition, 98
- plus proche interpretation (PPI)*, 81, 93, 95, 97
- prénexe, 20
- projecteur, 91
 - projecteur strict, 91
- propriété de sûreté, 76, 115
- relation
 - d'accessibilité, 70
 - de transition (Δ), 70
 - de sortie (Λ), 70
- réécriture (\rightarrow), 122
- résolution d'équations, 22, 56, 138
- restrict (\Downarrow), 111, 138
- restreuteur d'image (**rr**), 87, 90
- restreuteur d'image strict (**srr**), 91, 97
- satisfiable, 21
- séquence
 - acceptée, 72
 - d'entrées, 72
 - générée, 72
- substitution (σ), 50, 121
- support
 - de fonction, 54
 - minimaux, 137
- Skolem (fonction de), 23
- somme
 - exclusive de produits, 61
 - de produits, 59
- Stirling (formule de), 38
- synthèse 79, 81
- taille ($|\cdot|$), 32
 - d'un arbre de Shannon, 30
 - d'un arbre réduit de Shannon, 32, 34, 36, 37
 - d'un circuit, 41
 - d'une formule, 37
 - d'un graphe, 39, 41
- tautologie, 18
- TDG (*Typed Decison Graph*), 28
- terme, 121
- TSI (*Technique et Science de l'Information*), 141
- unification, 23, 122, 138
- variable
 - inutile, 137
 - Q -élimination, 20
 - libre, 18
 - liée, 18
 - ordre des variables, 18, 32, 39, 57, 104, 115

redondante, 80
vectochar, 87, 89, 99