Formules booléennes quantifiées : transformations formelles et calculs parallèles

Benoit da Mota

université angers

Année : 2010
N°d'ordre : 1079

FORMULES BOOLÉENNES QUANTIFIÉES :
TRANSFORMATIONS FORMELLES ET CALCULS PARALLÈLES

THÈSE DE DOCTORAT

Spécialité : Informatique

ÉCOLE DOCTORALE STIM
« SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET MATHÉMATIQUES »

Présentée et soutenue publiquement

Le 3 décembre 2010

À Angers

Par **Benoit Da Mota**

Devant le jury ci-dessous :

Président :	Jin-Kao HAO,	Professeur à l'Université d'Angers
Les rapporteurs :	Arnaud LALLOUET,	Professeur à l'Université de Caen
	Bertrand MAZURE,	Maître de conférences (HDR) à l'Université d'Artois
Les examinateurs :	Jin-Kao HAO,	Professeur à l'Université d'Angers
	Laurent SIMON,	Maître de conférences à l'Université de Paris-Sud 11
Directeur de thèse :	Pascal NICOLAS,	Professeur à l'Université d'Angers
Co-directeur :	Igor STÉPHAN,	Maître de conférences (HDR) à l'Université d'Angers

Laboratoire d'Étude et de Recherche en Informatique d'Angers
2, Bd Lavoisier, 49045 Angers Cedex 01

ED 503

À Élise,

À la mémoire de Pascal Nicolas,
sa grande valeur scientifique,
son implication,
sa passion,
son humour et
sa grande humanité.

Merci pour tout.

Remerciements

En premier lieu, je remercie les membres du jury : Arnaud LALLOUET, professeur à l'Université de Caen et Bertrand MAZURE, maître de conférences à l'Université d'Artois, les rapporteurs ; ainsi que Jin-Kao HAO, professeur à l'Université d'Angers et Laurent SIMON, maître de conférences à l'Université de Paris-Sud 11, les examinateurs.

Je souhaite exprimer toute ma reconnaissance à Pascal NICOLAS, professeur à L'université d'Angers, tant pour ses qualités professionnelles que personnelles. Je remercie également Igor STÉPHAN, maître de conférences à l'Université d'Angers, en particulier pour ces riches discussions que nous avons tenues à propos des formules booléennes quantifiées. Je leur suis très reconnaissant pour leur confiance, leur disponibilité et leur écoute durant toute cette thèse.

Je remercie également mes collègues et amis doctorants ou docteurs fraîchement promus : Wassim AYADI, Una BENLIC, Amaria Adila BOUABDALLAH, Marc-Olivier BUOB, Julien CHAUVEAU, Lionel CHAUVIN, Amandine DUFFOUX, Sylvain LAMPRIER, Sami LAROUM, Jorge MATURANA ORTIZ, Stéphane NGOMA, Jean-Luc PAILLAT, Mathilde PERRAIN, Daniel PORUMBEL, Thomas RAIMBAULT et Nadarajen VEERAPEN ; ainsi qu'une pensée toute particulière pour Julien ROBET.

Je remercie aussi l'ensemble des enseignants-chercheurs du LERIA, les techniciens et les secrétaires qui ont accompagné mes études et rendu possible ce doctorat.

Je remercie chaleureusement ma belle-famille, en particulier Brigitte et Bruno DENÉCHÈRE, pour leur confiance, leur soutien, ainsi que pour tout l'intérêt qu'ils portent à mon travail.

J'embrasse ma fille Capucine, née pendant la rédaction de cette thèse. Ses découvertes, sa bonne humeur et ses fous rires ont illuminé ces derniers mois.

Je dédie cette thèse à ma femme Élise, sans qui rien n'aurait été possible. Elle est la plus belle chose qui me soit arrivée et je lui renouvelle tout mon amour.

Sommaire

Introduction générale

D'après le dictionnaire [1], l'*informatique* est la « Science du traitement rationnel, notamment par machines automatiques, de l'information considérée comme le support des connaissances humaines et des communications dans les domaines technique, économique et social ». Dans cette thèse, intitulée « Formules booléennes quantifiées : transformations formelles et calculs parallèles », les connaissances auxquelles nous souhaitons appliquer des traitements sont exprimées dans un langage logique, celui des formules booléennes quantifiées (QBF). L'aspect traitement, présent dans la définition, est aisément capturé par les termes *transformation* et *calcul* du titre de ce manuscrit.

Pourquoi étudier les formules booléennes quantifiées ? Un intérêt constant est porté à l'amélioration des algorithmes pour SAT, le problème de satisfiabilité de la logique propositionnelle classique. En effet, de nombreuses tâches en représentation de connaissances peuvent être efficacement (en temps polynomial) réduites à SAT, qui est le problème **NP**-complet canonique. Cette technique a été appliquée avec succès pour de nombreuses applications comme par exemple la conception de circuits intégrés [Larrabee, 1992; Kim *et al.*, 2000], la vérification de modèle [Biere *et al.*, 1999a], la planification classique [Kautz et Selman, 1992; Kautz et Selman, 1998], l'ordonnancement [Crawford et Baker, 1994], la cryptographie [Massacci et Marraro, 2000; Potlapally *et al.*, 2007] et bien d'autres. La maturité des procédures de calcul permet aujourd'hui de résoudre certains problèmes industriels de grande taille. Mais ce n'est pas suffisant pour une multitude d'importants problèmes de décision parmi des champs très divers et dont l'étude théorique démontre qu'ils possèdent une complexité supérieure à celle de SAT [Egly et Tompits, 2001; Demaine, 2001]. Les formules booléennes quantifiées ont été introduites dans les années 70 par Meyer et Stockmeyer pour illustrer ce qu'ils ont appelé *hiérarchie polynomiale* en théorie de la complexité [Meyer et Stockmeyer, 1972; Stockmeyer et Meyer, 1973; Stockmeyer, 1977]. Le problème de validité pour les formules booléennes quantifiées (QSAT) est une généralisation du problème SAT et est le problème **PSPACE**-complet canonique. La syntaxe et la sémantique de la logique propositionnelle sont augmentées par l'addition de deux symboles, appelés *quantificateurs*. Ce bond dans les classes de complexité est le prix à payer pour représenter un nombre plus important de problèmes et permettre un codage plus concis. Par exemple, en intelligence artificielle, de nombreuses tâches de décision se réduisent efficacement à QSAT [Egly *et al.*, 2000; Egly *et al.*, 2001; Besnard *et al.*, 2009]. Il est possible à l'aide des QBF de modéliser des jeux finis à deux

1. Le Trésor de la Langue Française Informatisé, à l'adresse `http://atilf.atilf.fr/tlf.htm`

joueurs [Papadimitriou, 1994] ou encore de représenter certaines tâches en vérification formelle [Benedetti et Mangassarian, 2008]. Ce problème a donc fait l'objet d'études, d'une part pour mettre au point des procédures performantes et d'autre part pour exprimer différents problèmes dans ce langage. Bien que le problème de décision pour QBF soit **PSPACE**-complet [Stockmeyer et Meyer, 1973], les progrès en puissance de calcul lors des dernières décennies ont permis de s'y attaquer. La première procédure de décision pour QBF a été présentée en 1995 [Büning *et al.*, 1995] et les premiers résultats pratiques ont été publiés en 1997 [Cadoli *et al.*, 1997; Cadoli *et al.*, 1998].

L'héritage de SAT est encore très présent aujourd'hui dans le domaine des QBF. Il se retrouve dans le format d'entrée, qui est la forme normale conjonctive, efficace pour des procédures SAT, mais qui pour son extension aux QBF montre ses limites. Les problèmes s'expriment rarement sous cette forme et pour cause : elle capture mal la symétrie présente dans la sémantique des QBF [Ansótegui *et al.*, 2005; Sabharwal *et al.*, 2006]. Il est plus naturel d'utiliser un fragment du langage moins restrictif et plus expressif pour représenter un problème, puis de transformer la formule obtenue dans la forme requise. Les QBF ont emprunté les quantificateurs à la logique du premier ordre et ont hérité, par la même occasion, de la mise sous forme prénexe, une transformation permettant de placer tous les quantificateurs et les variables sur lesquelles ils portent, en tête de la formule. Ce traitement ne propose qu'une traduction inefficace pour certains opérateurs logiques, fait occulté dans la littérature sur le sujet. Dans la formule ainsi transformée, il est possible d'isoler une formule propositionnelle et de lui appliquer une deuxième transformation afin de produire une formule dans le format hérité de SAT. Chacune des ces transformations cause une perte de la structure de la formule et empêche de capturer la symétrie dont nous parlions précédemment. Nous nous focalisons sur la première transformation, moins étudiée, la seconde, nécessaire à SAT, étant le sujet de nombreux travaux.

Depuis quelques années, face aux défis technologiques et industriels, les microprocesseurs ont vu leurs fréquences de fonctionnement stagner, voire reculer, au profit du nombre d'unités de calcul. Dans le domaine du calcul haute performance, la barre du pétaflop (10^{15} opérations à virgule flottante par seconde) a été franchie en 2008. La puissance de calcul des ordinateurs augmente toujours, mais pour l'exploiter il faut intégrer la programmation parallèle même dans les procédures destinées à être exécutées sur une machine personnelle. C'est dans l'optique du calcul haute performance ou distribué que nous nous intéressons à la conception d'une procédure capable d'appliquer des traitements en parallèle à une formule booléenne quantifiée.

Suite à cette introduction, cette thèse s'articule en deux parties, précédées d'un chapitre d'état de l'art pour les formules booléennes quantifiées.

Dans le premier chapitre, nous présentons les aspects syntaxiques et sémantiques de la logique propositionnelle et des formules booléennes quantifiées. Nous faisons un bref passage par la théorie de la complexité afin d'y situer les problèmes auxquels nous portons notre attention. Enfin, nous dépeignons un paysage des applications théoriques et pratiques avec des QBF et détaillons deux méthodes de modélisation : d'une part avec les jeux finis à deux joueurs et d'autre part avec la vérification de modèle.

La première partie, intitulée *Algorithmes séquentiels pour QBF*, regroupe les chapitres 2

et 3. Le chapitre 2 est un état de l'art des procédures séquentielles pour le problème de décision des QBF. Dans un premier temps, nous donnons une chronologie de la recherche dans ce domaine, puis nous proposons un dénombrement des procédures disponibles aujourd'hui, en fonction du format accepté en entrée. Notre présentation se poursuit par la description des algorithmes classiques. Nous terminons ce tour d'horizon par des approches plus originales. Le chapitre 3 constitue notre contribution dans cette partie sur les algorithmes séquentiels. Notre classification des procédures en fonction du format accepté en entrée lors du chapitre précédent n'est pas anodine. Puisque la plupart des procédures acceptent une entrée sous une forme restreinte, nous commençons par détailler les transformations nécessaires et mettons en exergue les cas qui ont motivé notre travail. Nous présentons ensuite un motif particulier, que nous nommons *résultat intermédiaire* et qui possède des propriétés intéressantes. Celles-ci nous permettent de proposer des transformations pour des QBF quelconques par renommage de sous-formule. Ce chapitre se termine par une étude expérimentale, visant à valider nos propositions et à mesurer leurs impacts avec quelques procédures de l'état de l'art.

La deuxième partie, intitulée *Architecture parallèle et QBF*, regroupe les chapitres 4 et 5. Le chapitre 4 est un état de l'art pour l'architecture parallèle matérielle et logicielle, ainsi que pour l'application au problème de décision des QBF. Tout d'abord, nous faisons un brin d'histoire des microprocesseurs de 1971 à nos jours. Puis nous introduisons quelques notions sur le parallélisme et son efficacité. Nous examinons ensuite les différentes architectures matérielles parallèles, les modèles de mémoires et de communication. Nous réalisons un tour d'horizon des différents paradigmes de programmation parallèle avant de décrire celui par passage de messages. Enfin, nous présentons et analysons les procédures existantes pour la résolution des QBF en parallèle. Le chapitre 5 présente notre contribution dans ce domaine. Nous nous appuyons sur les constats effectués au chapitre 3 quant à la forme en entrée vis-à-vis de l'expressivité et de l'efficacité pour proposer les bases d'un outil de traitement des QBF quelconques. Nous proposons un nouveau format pour écrire des QBF et nous présentons notre modèle de données expressif et extensible, ainsi que la manière dont les traitements lui sont appliqués. Ensuite, nous exposons la philosophie de notre procédure parallèle de type client/serveur, qui mise sur un découpage syntaxique contrairement aux procédures parallèles de l'état de l'art. Puis, nous détaillons l'implantation de notre premier type de client, capable de traiter ces éléments et nous explicitons nos choix techniques. Nous déroulons dans la suite, un exemple complet et illustré de notre architecture parallèle afin d'en faciliter la compréhension. Nous terminons par une courte étude expérimentale visant essentiellement à évaluer les performances relatives de notre procédure.

Une conclusion générale où nous discutons des perspectives de nos travaux, vient clore ce manuscrit.

Chapitre 1

Les formules booléennes quantifiées

Sommaire

Un langage logique est un langage formel permettant de représenter des connaissances et de raisonner avec celles-ci. Ce langage possède un alphabet et des règles de syntaxe. Il peut être utilisé dans un système syntaxique sans se soucier du sens des symboles manipulés, il s'agit de *théorie de la preuve*. Une autre utilisation consiste à donner un sens aux symboles par l'intermédiaire d'une sémantique, il s'agit de *théorie des modèles*. Nous commençons par décrire le langage de la logique propositionnelle, puis nous donnons sa sémantique. Cette description s'inspire de la logique propositionnelle et des prédicats présentées dans *Logic for Computer Science* [Gallier, 1985]. Une fois terminés ces préliminaires, nous présentons la logique booléenne quantifiée comme étant une extension de la logique propositionnelle. Nous présentons ensuite quelques notions de complexité, puis des exemples d'applications avec les formules booléennes quantifiées.

1.1 La logique propositionnelle

1.1.1 Syntaxe

Nous nous intéressons d'abord à l'aspect syntaxique de la logique propositionnelle. Nous allons définir ce qu'est une formule propositionnelle (ou proposition). Tout d'abord, l'alphabet est défini ci-dessous.

Définition 1.1.1. *(Alphabet de la logique propositionnelle) L'alphabet de la logique propositionnelle, noté α_{prop}, est constitué :*

- *du symbole $\top$ (resp. $\bot$), constante propositionnelle représentant ce qui est toujours vrai (resp. faux),*
- *d'un ensemble dénombrable de symboles propositionnels, noté $\mathcal{SP}$;*
- *de l'ensemble des connecteurs logiques usuels : le symbole $\wedge$ est utilisé pour la conjonction, $\vee$ pour la disjonction, $\neg$ pour la négation, $\rightarrow$ pour l'implication, $\leftrightarrow$ pour la bi-implication et $\oplus$ pour le ou exclusif ;*
- *des symboles auxiliaires "(" et ")".*

L'ensemble des opérateurs logiques binaires est noté $\mathcal{O} = \{\wedge, \vee, \rightarrow, \leftrightarrow, \oplus\}$. Le langage **PROP** est généré à l'aide des règles présentées dans la définition suivante.

Définition 1.1.2. *(***PROP***) L'ensemble des propositions* **PROP** *est le plus petit ensemble vérifiant les conditions suivantes :*

- *tout élément de $\mathcal{SP} \cup \{\top, \bot\}$ est élément de* **PROP** *;*
- *si F est élément de* **PROP** *alors $(\neg F)$ est élément de* **PROP** *;*
- *si F et G sont éléments de* **PROP** *alors $(F \wedge G)$, $(F \vee G)$, $(F \rightarrow G)$, $(F \leftrightarrow G)$ et $(F \oplus G)$ sont éléments de* **PROP***.*

Soit $\{a, b, c, d\}$ un ensemble de symboles propositionnels, alors $(((a \vee (\neg c)) \vee d) \oplus (((\neg a) \wedge b) \wedge \bot))$ appartient à **PROP**, contrairement à $((\vee a)$ et $(a \wedge b) \rightarrow)$. Par souci de clarté, il est courant d'omettre certaines parenthèses à condition de ne pas rendre la proposition ambiguë. Il arrive aussi que certaines parenthèses soient remplacées par les symboles "["

et "]" afin de rendre plus visibles certaines chaînes ou sous-chaînes. Nous pouvons écrire la première proposition ainsi : $[(a \vee \neg c \vee d) \oplus (\neg a \wedge b \wedge \bot)]$.

Nous définissons maintenant quelques notions de vocabulaire nécessaires par la suite :

- un *littéral* est un symbole propositionnel ou bien sa négation. Soit a un symbole propositionnel, alors a est appelé *littéral positif*, $\neg a$ est appelé *littéral négatif*, a et $\neg a$ sont des *littéraux complémentaires* ;
- une *clause* est une disjonction de littéraux ;
- un *terme* est une conjonction de littéraux.

Nous distinguerons trois formes particulières pour les propositions.
Une proposition est sous *forme normale négative (NNF)* si elle est exclusivement constituée de conjonctions, de disjonctions et de littéraux. La proposition suivante est sous NNF :

$$((a \wedge b) \vee (\neg b \vee \neg a))$$

Une proposition est sous *forme normale conjonctive (CNF)* si c'est une conjonction de clauses. La proposition suivante est sous CNF :

$$((a \vee b) \wedge (\neg b \vee \neg a))$$

Une proposition est sous *forme normale disjonctive (DNF)* si c'est une disjonction de termes. La proposition suivante est sous DNF :

$$((a \wedge b) \vee (\neg b \wedge \neg a))$$

Nous pouvons remarquer dans les propositions précédentes que la forme normale conjonctive et la forme normale disjonctive sont des cas spécifiques de la forme normale négative.

1.1.2 Sémantique

La sémantique de la logique propositionnelle est régie par deux postulats. Premièrement, la valeur de vérité d'un symbole propositionnel est soit $vrai$, soit $faux$ (postulat de bivalence). Deuxièmement, la valeur de vérité d'une formule propositionnelle est exclusivement fonction de la valeur de vérité des propositions et des symboles propositionnels qui la composent (postulat de vérifonctionnalité). Nous commençons par définir le domaine **BOOL** des valeurs de vérité.

Définition 1.1.3. *(**BOOL**)*
L'ensemble des valeurs de vérité est l'ensemble $\mathbf{BOOL} = \{vrai, faux\}$.

À chaque constante et opérateur (resp. $\top$, $\bot$, $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, $\oplus$) est associée une fonction booléenne (resp. $i_\top, i_\bot :\rightarrow \mathbf{BOOL}$, $i_\neg : \mathbf{BOOL} \rightarrow \mathbf{BOOL}$, $i_\wedge, i_\vee, i_\rightarrow, i_\leftrightarrow, i_\oplus : \mathbf{BOOL} \times \mathbf{BOOL} \rightarrow \mathbf{BOOL}$) qui en définit sa sémantique. Nous rappelons la sémantique usuelle des constantes propositionnelles et des opérateurs logiques :

- $i_\top = vrai$;

x	y	$i_\neg(x)$	$i_\wedge(x,y)$	$i_\vee(x,y)$	$i_\rightarrow(x,y)$	$i_\leftrightarrow(x,y)$	$i_\oplus(x,y)$
$vrai$	$vrai$	$faux$	$vrai$	$vrai$	$vrai$	$vrai$	$faux$
$vrai$	$faux$	$faux$	$faux$	$vrai$	$faux$	$faux$	$vrai$
$faux$	$vrai$	$vrai$	$faux$	$vrai$	$vrai$	$faux$	$vrai$
$faux$	$faux$	$vrai$	$faux$	$faux$	$vrai$	$vrai$	$faux$

TABLE 1 – Sémantique usuelle des opérateurs logiques.

- $i_\bot = faux$;
- Pour les opérateurs logiques cf. TAB.1.

Définition 1.1.4. *(Valuation et sémantique des propositions) Une valuation v est une fonction de $\mathcal{SP}$ dans* **BOOL**, *elle attribue une valeur de vérité à chaque symbole propositionnel. L'ensemble des valuations est noté* **VALUATION**. *Nous notons*

$$v[x := b](y) = \begin{cases} b & \text{si } x = y \text{ et } b \in \mathbf{BOOL} \\ v(y) & \text{sinon.} \end{cases}$$

Soit une valuation v, la sémantique des propositions selon cette valuation v est définie inductivement :

- $[[\bot]](v) = i_\bot$;
- $[[\top]](v) = i_\top$;
- $[[x]](v) = v(x)$ *si* $x \in \mathcal{SP}$;
- $[[(A \circ B)]](v) = i_\circ([[A]](v), [[B]](v))$ *si* $A, B \in$ **PROP** *et* $\circ \in \mathcal{O}$;
- $[[\neg A]](v) = i_\neg([[A]](v))$ *si* $A \in$ **PROP**.

Définition 1.1.5. *(Modèle, satisfiabilité et équivalence logique)*

- *Un* modèle *d'une proposition A, noté $v \models A$, est une valuation v telle que l'application de la sémantique vérifie la proposition, ie. $[[A]](v) = vrai$.*
- *Une valuation v* falsifie une proposition *A, noté $v \not\models A$, si l'application de la sémantique ne vérifie pas la proposition, ie. $[[A]](v) = vrai$.*
- *Une proposition est* satisfiable *si elle admet au moins un modèle, sinon elle est* insatisfiable.
- *Si tout modèle d'une proposition A est un modèle pour une proposition B, noté $A \models B$, alors B est* conséquence logique *de A.*
- *Deux propositions A et B sont* logiquement équivalentes, *noté $A \equiv B$, si $A \models B$ et $B \models A$.*

La proposition $(a \wedge b)$ est satisfiable car elle admet la valuation $v[a := \mathbf{vrai}][b := \mathbf{vrai}]$ comme modèle. La proposition $(a \wedge \neg a)$ est insatisfiable car elle n'admet aucun modèle.

Pour toute proposition, il est possible d'écrire une proposition logiquement équivalente sous une des formes normales (NNF, CNF ou DNF), présentées dans la section précédente.

1.2 Les formules booléennes quantifiées

1.2.1 Syntaxe des QBF

Nous choisissons de présenter la syntaxe des QBF comme étant une extension du langage de la logique propositionnelle.

Définition 1.2.1. *(Alphabet des QBF) L'alphabet des formules booléennes quantifiées, noté α_{qbf}, est constitué :*

- *de l'alphabet de la logique propositionnelle, α_{prop} ;*
- *du symbole $\exists$ (resp. $\forall$) pour la quantification existentielle (resp. universelle).*

Le langage **QBF** est généré à l'aide des règles présentées dans la définition suivante.

Définition 1.2.2. *(**QBF**) L'ensemble des formules booléennes quantifiées* **QBF** *est le plus petit ensemble vérifiant les conditions suivantes :*

- *tout élément de* **PROP** *est élément de* **QBF** *;*
- *si F est élément de* **QBF** *alors $(\neg F)$ est élément de* **QBF** *;*
- *si F et G sont éléments de* **QBF** *alors $(F \wedge G)$, $(F \vee G)$, $(F \rightarrow G)$, $(F \leftrightarrow G)$ et $(F \oplus G)$ sont éléments de* **QBF** *;*
- *si F est élément de* **QBF** *et x un élément de $\mathcal{SP}$ alors $(\exists x\ F)$ et $(\forall x\ F)$ sont des éléments de* **QBF**.

Les quantificateurs ont été empruntés aux logiques d'ordre 1 et plus. Le passage de la logique propositionnelle aux formules booléennes quantifiées s'accompagne de quelques changements de dénomination. Comme en logique d'ordre 1, un quantificateur fait référence à une variable, pour les QBF nous parlerons de quantificateurs sur les *variables booléennes* de la formule, ou plus simplement *variables*. L'ensemble des symboles propositionnelles $\mathcal{SP}$ est aussi appelé *ensemble des variables booléennes*. Par convention, des quantificateurs différents lient des variables différentes, la QBF est dite *polie*. Nous reprenons les facilités de notation introduites pour la logique propositionnelle et présentons quelques exemples.

Soit $\{a, b, c, d\} \subset \mathcal{SP}$, alors les symboles propositionnels a, b, c et d sont aussi appelés *variables* et $(\forall b(((a \vee (\neg c)) \vee d) \oplus (\exists a(((\neg a) \wedge b) \wedge \bot))))$ appartient à **QBF**. Il est possible de l'écrire aussi ainsi : $(\forall b[(a \vee \neg c \vee d) \oplus (\exists a(\neg a \wedge b \wedge \bot))])$. Les chaînes de caractères $((\wedge \exists b)$ et $(b \rightarrow a) \rightarrow \forall)$ n'appartiennent pas à **QBF**.

Dans la QBF $(\forall b(((a \vee (\neg c)) \vee d) \oplus (\exists a(((\neg a) \wedge b) \wedge \bot))))$, nous pouvons remarquer que certaines variables n'ont pas de quantificateur qui y font référence ; c'est le cas de la variable d par exemple . Une variable non quantifiée est appelée *variable libre*. Une QBF qui ne possède pas de variable libre est une *QBF close*, dans le cas inverse c'est une *QBF ouverte*. Par la suite, nous utilisons q pour faire référence à un quantificateur quelconque et $\mathcal{VL}(F)$ pour représenter l'ensemble des variables libres dans une QBF F. Maintenant, nous définissons la notion de *portée* d'un quantificateur, ainsi que le vocabulaire qui l'accompagne.

Définition 1.2.3. *(Portée des quantificateurs) Soit $(q\ x\ F)$ une formule booléenne quantifiée, avec $q \in \{\forall, \exists\}$ et F une formule booléenne quantifiée où la variable booléenne x est libre, alors :*

- *F est la portée du quantificateur q ;*
- *toute variable de F est sous la portée du quantificateur q ;*
- *q porte sur x.*

La QBF $F = (\forall b(((a \vee (\neg c)) \vee d) \oplus (\exists a(((\neg a) \wedge b) \wedge \bot))))$, nous permet de faire quelques remarques :

- le quantificateur $\forall$, portant sur b, a comme portée $(((a \vee (\neg c)) \vee d) \oplus (\exists a(((\neg a) \wedge b) \wedge \bot)))$;
- le quantificateur $\exists$, portant sur a, a comme portée limitée $(((\neg a) \wedge b) \wedge \bot)$;
- la QBF F n'est pas close. La variable a dans $((a \vee (\neg c)) \vee d)$ est libre et différente de la variable a quantifiée existentiellement ci-dessus. Par convention, nous préférons renommer la variable quantifiée afin d'ôter toute ambiguïté. Nous écrivons par exemple, $F = (\forall b(((a \vee (\neg c)) \vee d) \oplus (\exists a'(((\neg a') \wedge b) \wedge \bot))))$. Le renommage d'une ou plusieurs variables s'effectue par remplacement syntaxique.

Définition 1.2.4. *(Remplacement syntaxique) Une fonction de remplacement σ est une fonction des variables booléennes dans les formules booléennes quantifiées. L'ensemble support d'une fonction de remplacement σ est l'ensemble des variables booléennes x telles que $\sigma(x) \neq x$. Soit $\{x_1, ..., x_n\}$ l'ensemble support de la fonction de remplacement σ, alors le remplacement syntaxique étendue aux QBF sera noté $\overline{\sigma} = [x_1 \leftarrow \sigma(x_1), \ldots, x_n \leftarrow \sigma(x_n)]$. L'application de $\overline{\sigma}$ est définie inductivement comme suit :*

$\overline{\sigma}(\bot)$	$=$	$\bot$	
$\overline{\sigma}(\top)$	$=$	$\top$	
$\overline{\sigma}(x)$	$=$	x	*si* $\overline{\sigma} = [x_1 \leftarrow \sigma(x_1), ..., x_n \leftarrow \sigma(x_n)]$ *et x une variable booléenne telle que* $x \notin \{x_1, ..., x_n\}$
$\overline{\sigma}(x_i)$	$=$	F_i	*si* $\overline{\sigma} = [x_1 \leftarrow \sigma(x_1), ..., x_n \leftarrow \sigma(x_n)]$, $\sigma(x_i) = F_i$, $F_i \in$ **QBF** *et* $x_i \in \{x_1, ..., x_n\}$
$\overline{\sigma}(\neg G)$	$=$	$\neg\overline{\sigma}(G)$	*si* $G \in$ **QBF**
$\overline{\sigma}((G\ o\ H))$	$=$	$(\overline{\sigma}(G)\ o\ \overline{\sigma}(H))$	*si* $G, H \in$ **QBF** *et* $o\ \in \{\vee, \wedge, \rightarrow, \leftrightarrow, \oplus\}$
$\overline{\sigma}((qx\ G))$	$=$	$(qx\ \overline{\sigma}(G))$	*si* $G \in$ **QBF**, $q \in \{\exists, \forall\}$ *et x une variable booléenne telle que* $x \notin \{x_1, ..., x_n\}$
$\overline{\sigma}((qx_i\ G))$	$=$	$(qx_i\ \overline{\sigma'}(G))$	*si* $G \in$ **QBF**, $q \in \{\exists, \forall\}$ *et* $\overline{\sigma'} = [x_1 \leftarrow \sigma(x_1), \ldots, x_{i-1} \leftarrow \sigma(x_{i-1}), x_{i+1} \leftarrow \sigma(x_{i+1}), \ldots, x_n \leftarrow \sigma(x_n)]$

Une formule booléenne quantifiée est *prénexe* ou *sous forme prénexe*, si tous ses quantificateurs apparaissent au début de celle-ci.

Définition 1.2.5. *(Forme prénexe, lieur et matrice) Un* lieur *Q est une chaîne de caractères $q_1\ x_1, ..., q_n\ x_n$, avec $x_1, ..., x_n$ des variables booléennes distinctes et $q_1, ..., q_n$ des quantificateurs. Une QBF $(Q\ F)$ est sous* forme prénexe *si Q est un lieur et F une formule booléenne sans quantificateurs, c'est à dire si F appartient au langage* **PROP**. *F est alors appelée* matrice *de la QBF prénexe.*

Dans le lieur d'une QBF prénexe, le passage d'un quantificateur existentiel à un quantificateur universel ou inversement est appelé *une alternance de quantificateurs*. De plus, il est usuel de dire que deux quantificateurs consécutifs de même type ont la même portée.

La matrice d'une QBF prénexe étant une proposition, il est possible d'utiliser aussi la distinction des trois formes normales de la logique propositionnelle. Soit la QBF quelconque $F = (\forall x(\exists z(x \to (\neg(\exists y(y \vee z))))))$ alors :

- $F \equiv \forall x \exists z \exists y(x \to (\neg(y \vee z)))$ est sous *forme prénexe ;*
- $F \equiv \forall x \exists z \exists y(\neg x \vee (\neg y \wedge \neg z)))$ est sous *forme normale négative* et plus particulièrement sous *forme normale disjonctive ;*
- $F \equiv \forall x \exists z \exists y((\neg x \vee \neg y) \wedge (\neg x \vee \neg z))$ est sous *forme normale négative* et plus particulièrement sous *forme normale conjonctive*.

1.2.2 Sémantique des QBF

La sémantique des formules booléennes quantifiées étend la sémantique de la logique propositionnelle et est régie par les deux mêmes prédicats de bivalence et vérifonctionnalité. Elle fait appel à la sémantique des constantes et opérateurs booléens définie de manière habituelle et rappelée dans la section 1.1.2. Nous devons donc introduire la sémantique des quantificateurs afin de traiter le langage **QBF**.

Définition 1.2.6. *(Sémantique des quantificateurs)*
Soit une valuation v, la sémantique des quantificateurs selon cette valuation est définie par :

$$i^x_\exists, i^x_\forall : (\mathbf{VALUATION} \to \mathbf{BOOL}) \times (\mathbf{VALUATION}) \to \mathbf{BOOL}$$

$$i^x_\exists(f)(v) = i_\vee(f(v[x := \mathbf{vrai}]), f(v[x := \mathbf{faux}]))$$

$$i^x_\forall(f)(v) = i_\wedge(f(v[x := \mathbf{vrai}]), f(v[x := \mathbf{faux}]))$$

Définition 1.2.7. *(Sémantique des QBF)*
Soit une valuation v, la sémantique des QBF selon cette valuation v est définie inductivement :

- $[[\bot]](v) = i_\bot$;
- $[[\top]](v) = i_\top$;
- $[[x]](v) = v(x)$ *si* $x \in \mathcal{SP}$;
- $[[(F \circ G)]](v) = i_\circ([[F]](v), [[G]](v))$ *si* $F, G \in$ **QBF** *et* $\circ \in \mathcal{O}$;
- $[[\neg F]](v) = i_\neg([[F]](v))$ *si* $F \in$ **QBF** ;
- $[[(\exists x\ F)]](v) = i^x_\exists([[F]])(v)$ *si* $F \in$ **QBF** ;
- $[[(\forall x\ F)]](v) = i^x_\forall([[F]])(v)$ *si* $F \in$ **QBF**.

Définition 1.2.8. *(Validité) Une QBF F, close ou non, est* valide, *noté* $\models F$, *si pour toute valuation v,*

$$[[F]](v) = \mathbf{vrai}$$

Dans l'exemple suivant, nous appliquons la sémantique sur une QBF close. Le résultat est **vrai** (resp. **faux**) si la QBF est valide (resp. non valide). Soit v une valuation, x, y et z des variables booléennes. Soit une formule propositionnelle $M = ((x \vee y) \leftrightarrow z)$, matrice pour la QBF $F = \exists x \exists y \forall z M$. Nous appliquons la sémantique des QBF sur F avec une valuation v qui n'a aucune influence puisque la QBF est close.

$$
\begin{aligned}
&[[\exists x \exists y \forall z M]](v) \\
&= i^x_\exists([[\exists y \forall z M]])(v) \\
&= i_\vee([[\exists y \forall z M]](v[x := \mathbf{vrai}]), [[\exists y \forall z M]](v[x := \mathbf{faux}])) \\
&= i_\vee(i^y_\exists([[\forall z M]])(v[x := \mathbf{vrai}]), i^y_\exists([[\forall z M]])(v[x := \mathbf{faux}])) \\
&= i_\vee(\\
&\quad i_\vee(\quad [[\forall z M]](v[x := \mathbf{vrai}][y := \mathbf{vrai}]), [[\forall z M]](v[x := \mathbf{vrai}][y := \mathbf{faux}]) \quad), \\
&\quad i_\vee(\quad [[\forall z M]](v[x := \mathbf{faux}][y := \mathbf{vrai}]), [[\forall z M]](v[x := \mathbf{faux}][y := \mathbf{faux}]) \quad) \quad) \\
&= i_\vee(\\
&\quad i_\vee(\quad i^z_\forall([[M]])(v[x := \mathbf{vrai}][y := \mathbf{vrai}]), i^z_\forall([[M]])(v[x := \mathbf{vrai}][y := \mathbf{faux}]) \quad), \\
&\quad i_\vee(\quad i^z_\forall([[M]])(v[x := \mathbf{faux}][y := \mathbf{vrai}]), i^z_\forall([[M]])(v[x := \mathbf{faux}][y := \mathbf{faux}]) \quad) \quad) \\
&= i_\vee(\\
&\qquad i_\vee(\\
&\qquad\quad i_\wedge(\quad [[M]](v[x := \mathbf{vrai}][y := \mathbf{vrai}][z := \mathbf{vrai}]), \\
&\qquad\qquad\quad [[M]](v[x := \mathbf{vrai}][y := \mathbf{vrai}][z := \mathbf{faux}]) \quad), \\
&\qquad\quad i_\wedge(\quad [[M]](v[x := \mathbf{vrai}][y := \mathbf{faux}][z := \mathbf{vrai}]), \\
&\qquad\qquad\quad [[M]](v[x := \mathbf{vrai}][y := \mathbf{faux}][z := \mathbf{faux}]) \quad) \quad), \\
&\qquad i_\vee(\\
&\qquad\quad i_\wedge(\quad [[M]](v[x := \mathbf{faux}][y := \mathbf{vrai}][z := \mathbf{vrai}]), \\
&\qquad\qquad\quad [[M]](v[x := \mathbf{faux}][y := \mathbf{vrai}][z := \mathbf{faux}]) \quad), \\
&\qquad\quad i_\wedge(\quad [[M]](v[x := \mathbf{faux}][y := \mathbf{faux}][z := \mathbf{vrai}]), \\
&\qquad\qquad\quad [[M]](v[x := \mathbf{faux}][y := \mathbf{faux}][z := \mathbf{faux}]) \quad) \quad) \quad)
\end{aligned}
$$

À cet endroit de l'exemple, les quantificateurs ont disparu, la sémantique des QBF coïncide avec la sémantique de la logique propositionnelle. Il est intéressant de remarquer que la matrice M apparaît 8 fois à cette étape et qu'il y a autant de valuations différentes, ce qui correspond à l'ensemble des lignes d'une table de vérité pour la matrice. Afin de rendre la suite de l'exemple plus lisible nous posons :

$$
\begin{aligned}
&v_{111} = v[x := \mathbf{vrai}][y := \mathbf{vrai}][z := \mathbf{vrai}], \quad && v_{110} = v[x := \mathbf{vrai}][y := \mathbf{vrai}][z := \mathbf{faux}] \\
&v_{101} = v[x := \mathbf{vrai}][y := \mathbf{faux}][z := \mathbf{vrai}], \quad && v_{100} = v[x := \mathbf{vrai}][y := \mathbf{faux}][z := \mathbf{faux}] \\
&v_{011} = v[x := \mathbf{faux}][y := \mathbf{vrai}][z := \mathbf{vrai}], \quad && v_{010} = v[x := \mathbf{faux}][y := \mathbf{vrai}][z := \mathbf{faux}] \\
&v_{001} = v[x := \mathbf{faux}][y := \mathbf{faux}][z := \mathbf{vrai}], \quad && v_{000} = v[x := \mathbf{faux}][y := \mathbf{faux}][z := \mathbf{faux}]
\end{aligned}
$$

$$
\begin{aligned}
&= i_\vee(\\
&\qquad i_\vee(\\
&\qquad\quad i_\wedge(\quad i_\leftrightarrow([[(x \vee y)]](v_{111}), [[z]](v_{111})), i_\leftrightarrow([[(x \vee y)]](v_{110}), [[z]](v_{110})) \quad), \\
&\qquad\quad i_\wedge(\quad i_\leftrightarrow([[(x \vee y)]](v_{101}), [[z]](v_{101})), i_\leftrightarrow([[(x \vee y)]](v_{100}), [[z]](v_{100})) \quad) \quad), \\
&\qquad i_\vee(\\
&\qquad\quad i_\wedge(\quad i_\leftrightarrow([[(x \vee y)]](v_{011}), [[z]](v_{011})), i_\leftrightarrow([[(x \vee y)]](v_{010}), [[z]](v_{010})) \quad), \\
&\qquad\quad i_\wedge(\quad i_\leftrightarrow([[(x \vee y)]](v_{001}), [[z]](v_{001})), i_\leftrightarrow([[(x \vee y)]](v_{000}), [[z]](v_{000})) \quad) \quad) \quad)
\end{aligned}
$$

$$
\begin{array}{l}
= i_\vee(\\
\quad i_\vee(\\
\quad\quad i_\wedge(\quad i_\leftrightarrow(i_\vee([[x]](v_{111}),[[y]](v_{111})),[[z]](v_{111})), \\
\quad\quad\quad\quad\quad i_\leftrightarrow(i_\vee([[x]](v_{110}),[[y]](v_{110})),[[z]](v_{110})) \quad), \\
\quad\quad i_\wedge(\quad i_\leftrightarrow(i_\vee([[x]](v_{101}),[[y]](v_{101})),[[z]](v_{101})), \\
\quad\quad\quad\quad\quad i_\leftrightarrow(i_\vee([[x]](v_{100}),[[y]](v_{100})),[[z]](v_{100})) \quad) \quad), \\
\quad i_\vee(\\
\quad\quad i_\wedge(\quad i_\leftrightarrow(i_\vee([[x]](v_{011}),[[y]](v_{011})),[[z]](v_{011})), \\
\quad\quad\quad\quad\quad i_\leftrightarrow(i_\vee([[x]](v_{010}),[[y]](v_{010})),[[z]](v_{010})) \quad), \\
\quad\quad i_\wedge(\quad i_\leftrightarrow(i_\vee([[x]](v_{001}),[[y]](v_{001})),[[z]](v_{001})), \\
\quad\quad\quad\quad\quad i_\leftrightarrow(i_\vee([[x]](v_{000}),[[y]](v_{000})),[[z]](v_{000})); \quad) \quad) \quad)
\end{array}
$$

$$
\begin{array}{l}
= i_\vee(\\
\quad i_\vee(\\
\quad\quad i_\wedge(\quad i_\leftrightarrow(i_\vee(\mathbf{vrai},\mathbf{vrai}),\mathbf{vrai}), i_\leftrightarrow(i_\vee(\mathbf{vrai},\mathbf{vrai}),\mathbf{faux}) \quad), \\
\quad\quad i_\wedge(\quad i_\leftrightarrow(i_\vee(\mathbf{vrai},\mathbf{faux}),\mathbf{vrai}), i_\leftrightarrow(i_\vee(\mathbf{vrai},\mathbf{faux}),\mathbf{faux}) \quad) \quad), \\
\quad i_\vee(\\
\quad\quad i_\wedge(\quad i_\leftrightarrow(i_\vee(\mathbf{faux},\mathbf{vrai}),\mathbf{vrai}), i_\leftrightarrow(i_\vee(\mathbf{faux},\mathbf{vrai}),\mathbf{faux}) \quad), \\
\quad\quad i_\wedge(\quad i_\leftrightarrow(i_\vee(\mathbf{faux},\mathbf{faux}),\mathbf{vrai}), i_\leftrightarrow(i_\vee(\mathbf{faux},\mathbf{faux}),\mathbf{faux}) \quad) \quad) \quad)
\end{array}
$$

La sémantique des QBF a été appliquée, l'étape suivante consiste à faire appel à la sémantique des opérateurs, $i_\wedge, i_\vee, i_\leftrightarrow$, afin de finir l'évaluation de la QBF :

$$
\begin{array}{l}
= i_\vee(\\
\quad\quad i_\vee(\quad i_\wedge(i_\leftrightarrow(\mathbf{vrai},\mathbf{vrai}),i_\leftrightarrow(\mathbf{vrai},\mathbf{faux})), i_\wedge(i_\leftrightarrow(\mathbf{vrai},\mathbf{vrai}),i_\leftrightarrow(\mathbf{vrai},\mathbf{faux})) \quad), \\
\quad\quad i_\vee(\quad i_\wedge(i_\leftrightarrow(\mathbf{vrai},\mathbf{vrai}),i_\leftrightarrow(\mathbf{vrai},\mathbf{faux})), i_\wedge(i_\leftrightarrow(\mathbf{faux},\mathbf{vrai}),i_\leftrightarrow(\mathbf{faux},\mathbf{faux})) \quad)) \\
= i_\vee(i_\vee(i_\wedge(\mathbf{vrai},\mathbf{faux}),i_\wedge(\mathbf{vrai},\mathbf{faux})),i_\vee(i_\wedge(\mathbf{vrai},\mathbf{faux}),i_\wedge(\mathbf{faux},\mathbf{vrai}))) \\
= i_\vee(i_\vee(\mathbf{faux},\mathbf{faux}),i_\vee(\mathbf{faux},\mathbf{faux})) \\
= i_\vee(\mathbf{faux},\mathbf{faux}) \\
= \mathbf{faux}
\end{array}
$$

Dans l'exemple précédent, la QBF $\exists x \exists y \forall z((x \vee y) \leftrightarrow z)$ est *non valide*. Si nous avions appliqué la sémantique des QBF sur $\forall z \exists x \exists y((x \vee y) \leftrightarrow z)$, pourtant très proche, nous aurions pu prouver la *validité* de cette formule. Les notions de *satisfiabilité* et de *modèle* issues de la logique propositionnelle s'appliquent aussi pour les QBF quelconques pas forcément closes.

Définition 1.2.9. *(Modèle, satisfiabilité et équivalence logique)*

- *Un* modèle *d'une QBF F, noté $v \models F$, est une valuation v telle que l'application de la sémantique vérifie la QBF, ie.* $[[F]](v) = \mathbf{vrai}$.
- *Une valuation v* falsifie une QBF *F, noté $v \not\models F$, si l'application de la sémantique ne vérifie pas la QBF, ie.* $[[F]](v) = \mathbf{faux}$.
- *Une QBF est* satisfiable *si elle admet au moins un modèle, sinon elle est* insatisfiable.
- *Si tout modèle d'une QBF F est un modèle pour une QBF G, noté $F \models G$, alors G est* conséquence logique *de F.*
- *Deux QBF F et G sont* logiquement équivalentes, *noté $F \equiv G$, si $F \models G$ et $G \models F$.*

Dans le cas d'une QBF avec des variables libres, la validité de la formule est un résultat plus fort que la satisfiabilité. En effet, cela signifie que la valuation n'a aucune influence sur la satisfiabilité de la QBF. Dans le cas d'une QBF close, le résultat de l'application de la sémantique est le même pour toute valuation. Par conséquence, validité et satisifiablité se confondent pour les formules closes.

Théorème 1.2.10. *Une QBF close est satisfiable si et seulement si elle est valide.*

Par la suite nous préférerons parler du problème de validité des QBF closes, plutôt que du problème de satisfiabilité, bien que pour ce cas spécifique il s'agisse du même problème. A contrario, ces notions de vocabulaire, et particulièrement la différence entre validité et satisfiabilité, prennent tout leur intérêt lorsque les QBF ont des variables libres.

Autre conséquence directe de la définition de la validité : toutes les QBF valides sont logiquemement équivalentes entre-elles et a fortiori, équivalentes à $\top$.

Théorème 1.2.11. *Une QBF F, close ou non, est valide si et seulement si $F \equiv \top$.*

Par exemple, pour deux QBF, $F = \forall z \exists x \exists y((x \vee y) \leftrightarrow z)$ et $\top$. Quelle que soit v une valuation, nous pouvons démontrer :
(1) $[[\forall z \exists x \exists y((x \vee y) \leftrightarrow z)]](v) = \mathbf{vrai}$;
(2) $[[\top]](v) = \mathbf{vrai}$.
Donc $\models \top$ et $\models (\forall z \exists x \exists y((x \vee y) \leftrightarrow z))$ et tous les modèles de F sont des modèles pour $\top$ (ie. $F \models \top$) et inversement (ie. $\top \models F$). Nous avons alors $\forall z \exists x \exists y((x \vee y) \leftrightarrow z) \equiv \top$.

La validité d'une QBF est préservée par le remplacement d'une sous-formule par une QBF qui lui est équivalente. Il existe un certain nombre d'équivalences logiques, reposant sur la sémantique, qui permettent de transformer syntaxiquement les QBF et qui préservent la validité. Le théorème qui suit est l'expression syntaxique de la sémantique des quantificateurs. Ces équivalences permettent la suppression des quantificateurs et par la même occasion transforment une expression de **QBF** vers **PROP** après élimination de tous les quantificateurs. Toutefois, cette opération ne peut pas être réalisée en espace polynomial.

Théorème 1.2.12. *Soit F une QBF et x une variable booléenne, alors les deux équivalences logiques suivantes sont vraies :*

$$(\exists x\ F) \equiv ([x \leftarrow \bot](F) \vee [x \leftarrow \top](F))$$
$$(\forall x\ F) \equiv ([x \leftarrow \bot](F) \wedge [x \leftarrow \top](F))$$

De la même manière, tout un ensemble d'équivalences permet de déplacer les quantificateurs vers l'intérieur ou l'extérieur d'une QBF. Ainsi, toute QBF est logiquement équivalente à une QBF prénexe. Cela dit, il n'est pas toujours possible de préserver le nombre de variables et/ou la taille de la formule. Nous y reviendrons très en détail au cœur du chapitre 3. Enfin, si une QBF est sous forme prénexe, alors elle est logiquement équivalente à une QBF prénexe dont la matrice est sous une des formes normales (NNF, CNF ou DNF).

1.3 Notions de complexité

Il est courant de lire et d'écrire, que le problème de décision de la satisfiabilité des propositions booléennes est le problème **NP**-complet canonique. De la même manière, le problème de décision de la validité des formules booléennes quantifiées est décrit comme le problème **PSPACE**-complet canonique. Il serait difficile de continuer notre propos sans s'attarder, ne serait-ce que très peu, sur ces aspects qui ont trait à la complexité des calculs. Dans cette section, nous allons nous efforcer de donner quelques rudiments de complexité inspirés de deux ouvrages de référence dans le domaine : « *Computational Complexity* » [Papadimitriou, 1994] et « *Computers and Intractability : A Guide to the Theory of* **NP**-*Completeness* » [Garey et Johnson, 1979]. Pour ce faire, nous décrivons succinctement la *machine de Turing* et le lien étroit qui la lie aux *classes de complexité*. Enfin, nous présentons la machine de Turing à oracles qui permet de décrire la hiérarchie polynomiale.

1.3.1 Machines de Turing et complexité

La machine de Turing, du nom de son inventeur Alan Turing, sert de modèle formel pour les algorithmes dans la théorie de la complexité. Dans son livre « *Computational Complexity* » [Papadimitriou, 1994], Christos H. Papadimitriou parle de la machine de Turing en ces termes : « *C'est amusant le peu dont nous avons besoin pour tout avoir !* ». La machine de Turing basique est capable d'exprimer n'importe quel algorithme et de simuler n'importe quel langage de programmation.

Une machine de Turing est un dispositif qui accepte les *mots* d'un *langage* $\mathcal{L}$ définis sur un *alphabet* Θ. De façon simplifiée, elle est composée de :

- Un *ruban* d'entrée où est placé le mot dont nous souhaitons vérifier l'appartenance au langage $\mathcal{L}$. Ce ruban est de longueur infinie afin de pouvoir écrire suffisamment de symboles si nécessaire durant l'exécution. L'alphabet Θ contient toujours un symbole de *début de ruban* et le symbole spécial « *blanc* » ;
- Une *tête de lecture* qui indique le prochain caractère à lire sur le ruban. Elle peut se déplacer vers la droite ou vers la gauche, elle permet également de lire et d'écrire un caractère ;
- Un *ensemble fini d'états*, dont un *état de départ* et un ou plusieurs *états accepteurs*. Quand un état accepteur est atteint, le mot en entrée a été accepté par la machine de Turing comme mot du langage $\mathcal{L}$;
- Une *fonction de transition* qui permet à partir d'un état et d'un caractère lu de déterminer l'état suivant, le symbole à écrire et le déplacement de la tête de lecture. Cette fonction représente le programme de la machine.

Avec cette définition, la fonction de transition permet d'associer à un état courant et un caractère lu, un unique état successeur. C'est pour cela qu'une telle machine de Turing est qualifiée de *déterministe*. Il existe deux extensions de cette *machine de Turing déterministe*. Premièrement, il est possible de décrire une machine avec plusieurs rubans. Deuxièmement, il est possible de considérer une machine de Turing à accès aléatoire ou RAM pour

« *Random Access Machine* ». Ces deux machines peuvent être simulées par une machine de Turing déterministe basique avec une perte d'efficacité polynomiale. Une extension qui va plus nous intéresser d'un point de vue théorique est la *machine de Turing non déterministe* où la fonction de transition est remplacée par une *relation*. Alors, à partir d'un état courant et d'un caractère lu, la relation de transition peut désigner plusieurs états successeurs. Un mot du langage est accepté s'il existe au moins une exécution qui mène à un état accepteur. Le nombre de transitions de cette exécution particulière détermine le temps d'exécution du programme sur une machine non déterministe. En effet, il faut imaginer qu'une telle machine explore toutes les possibilités permises par la relation de transition simultanément et s'arrête dès qu'un état accepteur est atteint. Mais, nous ne savons pas construire de machine non déterministe et pour exécuter un programme représentable à l'aide d'une machine de Turing non déterministe, il faut que nous fassions un choix stratégique ou aléatoire lors des transitions. Si ce choix est mauvais, il faudra revenir en arrière et faire un autre choix. Il est possible aussi de faire le bon choix à l'aide d'un *oracle*, c'est ce que nous verrons dans la section suivante.

Comme nous l'avons dit précédemment, le temps d'exécution d'une machine de Turing déterministe est mesuré en nombre de transitions menant à un état accepteur. Une autre mesure nous intéresse : l'espace, c'est-à-dire la taille de ruban, nécessaire à l'exécution. Nous souhaitons classer les langages à l'aide de ces mesures en fonction de la taille du mot en entrée, notée n. Nous nous intéressons uniquement aux langages *décidables*, c'est à dire si une machine de Turing accepte le langage et ne permet pas d'exécution infinie. On parle de *procédure de décision* pour le langage. $\mathbf{TIME}(f(n))$ est la classe de langages décidés par une machine de Turing déterministe dans un temps borné par $f(n)$. $\mathbf{NTIME}(f(n))$ est la classe de langages décidés par une machine de Turing non déterministe dans un temps borné par $f(n)$. $\mathbf{SPACE}(f(n))$ est la classe de langages décidés par une machine de Turing déterministe avec un ruban dont la longueur est bornée par $f(n)$. Pour des raisons pratiques évidentes, il est préférable que ces fonctions $f(n)$ aient une croissance la plus lente possible. Il est alors possible de définir les trois classes de langages qui nous seront utiles par la suite :

- $\mathbf{P} = \bigcup_{k\geq 1}\mathbf{TIME}(n^k)$ est la classe des langages acceptés polynomialement en temps sur une machine de Turing déterministe ;
- $\mathbf{NP} = \bigcup_{k\geq 1}\mathbf{NTIME}(n^k)$ est la classe des langages acceptés polynomialement en temps sur une machine de Turing non déterministe ;
- $\mathbf{PSPACE} = \bigcup_{k\geq 1}\mathbf{SPACE}(n^k)$ est la classe des langages acceptés polynomialement en espace sur une machine de Turing déterministe.

Nous pouvons maintenant exposer quelques relations connues entre ces classes. Puisque une machine de Turing déterministe peut être vue comme un cas particulier d'une machine de Turing non déterministe, il est immédiat que $\mathbf{P} \subseteq \mathbf{NP}$. La réciproque est un problème ouvert qui permettrait de répondre à l'un des problème les plus fondamentaux en théorie de la complexité, $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$. On conjecture habituellement que $\mathbf{P} \neq \mathbf{NP}$. G. J. Woeginger dresse une liste des travaux [1] essayant de statuer sur cette question. C'est à V. Également, il est assez facile de montrer que $\mathbf{NP} \subseteq \mathbf{PSPACE}$, mais la réciproque

1. `http://www.win.tue.nl/~gwoegi/P-versus-NP.htm`

est encore une fois un problème ouvert. Il est possible de simuler en temps exponentiel une machine de Turing non déterministe N avec une machine déterministe. Alors, pour une constante c dépendante de N, $\mathbf{NTIME}(f(n)) \subseteq \bigcup_{c>1} \mathbf{TIME}(c^{f(n)})$. Pour la classe de langages **NP**, $\mathbf{NP} = \bigcup_{k\geq 1}\mathbf{TIME}(2^{n^k})$. Comme nos ordinateurs peuvent exécuter uniquement des programmes déterministes, ce résultat traduit la difficulté en pratique pour résoudre des problèmes de décision de la classe **NP** ou **PSPACE**.

Pour finir ce bref survol de la théorie de la complexité, nous devons encore parler de *réduction*, de *complétude* et du *complémentaire d'un langage*. Toute classe de complexité contient une infinité de langages, or parmi ces langages certains sont plus *difficiles* à décider que d'autres. Nous pouvons alors introduire la *réduction* qui permet de conclure qu'un langage $\mathcal{L}_1$ est au moins aussi *difficile* qu'un langage $\mathcal{L}_2$, si $\mathcal{L}_2$ peut être réduit à $\mathcal{L}_1$. Une réduction *efficace* de $\mathcal{L}_2$ à $\mathcal{L}_1$ est une fonction R des chaînes de caractères dans les chaînes de caractères, calculable par une machine de Turing déterministe en espace $\mathcal{O}(log(n))$, telle que pour chaque mot $x \in \mathcal{L}_2$, $R(x) \in \mathcal{L}_1$. Il est aussi important de préciser qu'une réduction est un algorithme en temps polynomial. La réductibilité est transitive et il est donc légitime de s'intéresser au langage le plus grand ou *difficile* d'une classe de complexité. Un langage $\mathcal{L}$ est *complet* pour une classe de complexité si tous les autres langages appartenant à cette classe peuvent être réduits à celui-ci. Pour qu'un problème complet existe pour une classe de complexité, il faut que cette classe soit close par réductions. C'est le cas entre autre de **P**, **NP** et **PSPACE**. Si un problème est complet pour deux classes de complexité, alors celles-ci sont égales. Le théorème de Cook [Cook, 1971] établit que le problème SAT est **NP**-complet et avec une variante, Stockmeyer et Meyer prouvent que la problème de validité des QBF (QSAT) est **PSPACE**-complet [Stockmeyer et Meyer, 1973]. Les langages complets pour une classe de complexité sont particulièrement intéressants à étudier. D'une part, tous les autres langages de cette classe peuvent s'y réduire. D'autre part, il s'agit du langage le moins enclin à appartenir à une classe de complexité plus faible. Enfin, pour un alphabet Θ et un langage $\mathcal{L} \in \Theta^*$, le complémentaire de $\mathcal{L}$, noté $\overline{\mathcal{L}}$, est la langage $\Theta^* - \mathcal{L}$; c'est-à-dire l'ensemble des chaînes de caractères qui ne sont pas dans le langage $\mathcal{L}$, mais qui sont formées correctement à l'aide de l'alphabet Θ. Pour toute classe de complexité $\mathcal{C}$, **co**$\mathcal{C}$ représente la classe $\{\overline{\mathcal{L}} : \mathcal{L} \in \mathcal{C}\}$.

1.3.2 Machine de Turing à oracles et hiérarchie polynomiale

Tout comme les machines de Turing non déterministes, les machines à oracles n'existent pas à proprement parler. Il s'agit essentiellement d'un outil théorique pour résoudre des situations dans la théorie de la complexité. Un oracle est un dispositif algorithmique qui répondrait correctement et immédiatement à une requête. La réponse est utilisée pour continuer la calcul, voire même pour formuler la requête suivante et ainsi de suite. Plus formellement, une machine de Turing avec oracle $M^?$ est une machine de Turing déterministe avec plusieurs rubans, dont un spécifique pour les requêtes et trois états spécifiques ; le « ? » représente un oracle quelconque. La complexité en temps d'une telle machine est définie comme pour une machine ordinaire, à la différence près que chaque question à l'oracle compte pour un pas. Il est donc possible de définir de nouvelles classes de lan-

gages. Pour $\mathcal{C}$ une quelconque classe de complexité de temps déterministe ou non, $\mathcal{C}^A$ est la classe des langages décidés dans un temps borné comme dans $\mathcal{C}$, mais avec l'appel à un oracle A. A peut être une classe de complexité ; ainsi $\mathbf{P}^{\mathbf{NP}}$ est l'ensemble de langages décidés en temps polynomial par une machine déterministe de Turing en faisant appel à un oracle capable de répondre en un pas de calcul aux questions pour tous les langages décidés en temps polynomial par une machine de Turing non déterministe (ie. la classe **NP**).

La hiérarchie polynomiale (**HP**) [Stockmeyer, 1977], est une classe de complexité définie récursivement à partir de ses sous-classes. Sa définition à l'aide des machines de Turing avec oracle est particulièrement adaptée à la philosophie de notre proposition d'architecture parallèle au chapitre 5. Les cas de base de la définition récursive sont :

$$\Delta_0^{\mathbf{P}} = \Sigma_0^{\mathbf{P}} = \Pi_0^{\mathbf{P}} = \mathbf{P}$$

Puis pour tout entier $k \geq 0$:

$$\begin{aligned}\Delta_{k+1}^{\mathbf{P}} &= \mathbf{P}^{\Sigma_k^{\mathbf{P}}} \\ \Sigma_{k+1}^{\mathbf{P}} &= \mathbf{NP}^{\Sigma_k^{\mathbf{P}}} \\ \Pi_{k+1}^{\mathbf{P}} &= \mathbf{co}\Sigma_{k+1}^{\mathbf{P}}\end{aligned}$$

Dans cette hiérarchie nous retrouvons 3 cas particuliers :

$$\begin{aligned}\mathbf{P} &= \Delta_1^{\mathbf{P}} = \Delta_0^{\mathbf{P}} = \Pi_0^{\mathbf{P}} = \Sigma_0^{\mathbf{P}} \\ \mathbf{NP} &= \Sigma_1^{\mathbf{P}} \\ \mathbf{coNP} &= \Pi_1^{\mathbf{P}}\end{aligned}$$

Ces définitions impliquent les inclusions suivantes :

$$\begin{aligned}\Sigma_k^{\mathbf{P}} \subseteq \Delta_{k+1}^{\mathbf{P}} \subseteq \Sigma_{k+1}^{\mathbf{P}} \\ \Pi_k^{\mathbf{P}} \subseteq \Delta_{k+1}^{\mathbf{P}} \subseteq \Pi_{k+1}^{\mathbf{P}}\end{aligned}$$

Ces relations d'inclusion à l'intérieur de la hiérarchie polynomiale sont illustrées par la figure 1. Pour ces inclusions, le fait d'être strictes ou non est un ensemble de problèmes ouverts, parmi lesquels nous retrouvons $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$. De la même manière, il est habituel de considérer la conjecture où chaque *étage* k est inclus strictement dans un plus grand $k+1$. S'il est possible de prouver l'égalité de ces 2 étages, la hiérarchie polynomiale *s'effondre* au niveau k. Et donc, si $\mathbf{P} = \mathbf{NP}$, elle s'effondre complètement. Il nous reste à définir la hiérarchie polynomiale elle-même, comme étant l'union de tous ses niveaux :

$$\mathbf{HP} = \bigcup_{k \geq 0} \Sigma_k^{\mathbf{P}}$$

Il est possible de montrer que $\mathbf{HP} \subseteq \mathbf{PSPACE}$ mais encore une fois, $\mathbf{HP} \stackrel{?}{=} \mathbf{PSPACE}$ est un problème ouvert. Cependant, **PSPACE** possède des problèmes complets et si $\mathbf{HP} = \mathbf{PSPACE}$, alors **HP** posséderait des problèmes complets ; par conséquent la hiérarchie polynomiale aurait un nombre fini de niveaux différents et s'effondrerait. La figure 2 illustre **PSPACE**, ainsi que **HP** et son contenu. Le théorème théorème 3.1 dans

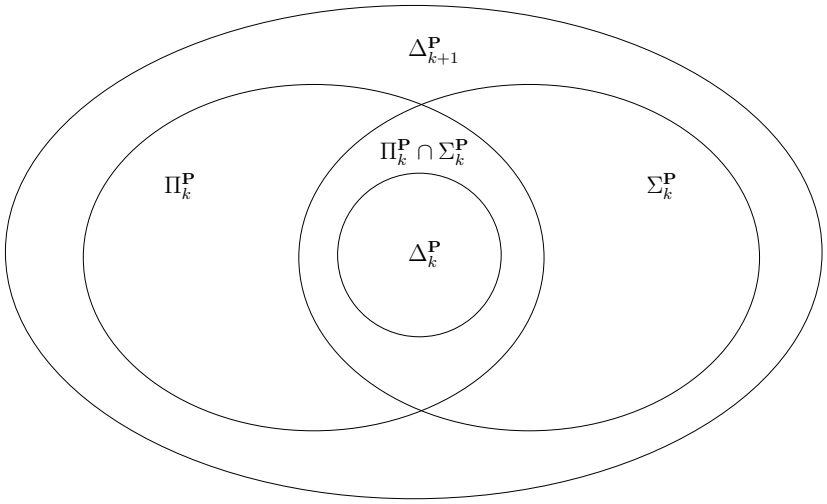


FIGURE 1 – Relations d'inclusion à l'intérieur de la hiérarchie polynomiale.

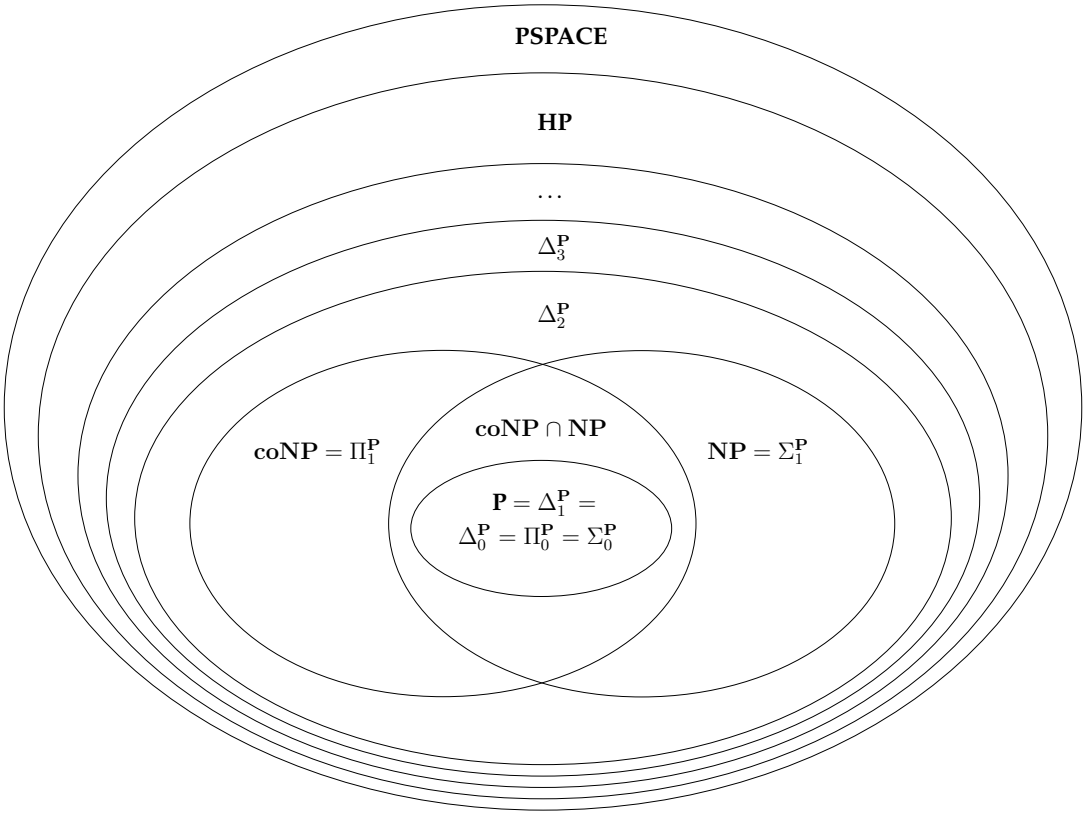


FIGURE 2 – Vue d'esprit de la hiérarchie polynomiale et de **PSPACE**.

[Stockmeyer, 1977] permet de raccrocher la notion de quantificateurs à la hiérarchie polynomiale. Une QBF prénexe dont le lieur est de la forme $\exists x_1 \forall x_2 \exists x_3 \dots q_k x_k$, où les quantificateurs s'alternent (c'est-à-dire que $q_k = \exists$ (resp. $\forall$) si k est impair (resp. pair)), appartient à la classe $\Sigma_k^{\mathbf{P}}$. Inversement, une QBF prénexe dont le lieur est de la forme $\forall x_1 \exists x_2 \forall x_3 \dots q_k x_k$, où les quantificateurs s'alternent, appartient à la classe $\Pi_k^{\mathbf{P}}$. Par exemple, la validité d'une QBF prénexe avec une seule $\exists\forall$-alternance de quantificateurs est un problème de la classe $\Sigma_2^{\mathbf{P}}$, alors que le problème de la validité d'une QBF prénexe avec une seule $\forall\exists$-alternance de quantificateurs est un problème de la classe $\Pi_2^{\mathbf{P}}$. D'un point de vue de la complexité théorique, les QBF prénexes peuvent ainsi être classer en fonction de leur lieur. Il est possible aussi de vouloir réaliser une procédure de décision spécifique à un niveau de la hiérarchie polynomiale. Des personnes se sont notamment intéressées aux QBF des classes $\Sigma_2^{\mathbf{P}}$ et $\Pi_2^{\mathbf{P}}$ car y sont représentables des problèmes pratiques, en vérification formelle notamment, sans oublier $\Sigma_1^{\mathbf{P}}$ pour SAT.

1.4 Applications et codage avec des QBF

De nombreuses tâches en représentation de connaissances peuvent être efficacement réduites à SAT. Cette technique a été appliquée avec succès pour de nombreuses applications comme par exemple la conception de circuits intégrés, la planification classique, l'ordonnancement, la cryptographie et bien d'autres. La maturité des procédures de calcul permet aujourd'hui de résoudre certains problèmes industriels de grande taille. Mais ce n'est pas suffisant pour une multitude d'importants problèmes de décision parmi des champs très divers et dont l'étude théorique démontre qu'ils possèdent une complexité supérieure à celle de SAT [Egly et Tompits, 2001; Demaine, 2001]. C'est donc naturellement que le problème de décision des QBF a fait l'objet d'études, d'une part pour mettre au point des procédures de résolution efficaces et d'autre part pour exprimer différents problèmes dans ce langage.

En intelligence artificielle, de nombreuses tâches se réduisent à QSAT [Egly *et al.*, 2000; Egly *et al.*, 2001; Besnard *et al.*, 2009] : l'abduction, l'existence d'extension stable en logique auto-épistémique, l'existence d'extension en logique des défauts, l'existence de modèle stable en programmation logique disjonctive, la circonscription propositionnelle et l'argumentation déductive. Les QBF permettent de représenter des jeux finis à deux joueurs [Papadimitriou, 1994; Demaine, 2001] comme le puissance 4 [Gent et Rowley, 2003] pour n'en citer qu'un ou encore la planification conditionnelle [Rintanen, 1999a]. La vérification formelle dont est issue la plupart des benchmarks pour QBF consiste à prouver ou réfuter la correction d'un système devant respecter certaines propriétés en utilisant des méthodes formelles. L'utilisation de QBF comme langage pour certaines tâches de raisonnement permet d'obtenir des formules plus compactes que si elles avaient été décrites en logique propositionnelle [Benedetti et Mangassarian, 2008]. Comme application nous pouvons citer, la vérification de circuits logiques et de descriptions de protocoles [Ayari et Basin, 2000; Pan et Vardi, 2004b; Ling *et al.*, 2005], la vérification de modèles (model checking et bounded model checking) [Dershowitz *et al.*, 2005; Biere, 2004], la vérification de convergence [Bryant *et al.*, 2003], le calcul de diamètre d'un système [Mneimneh et Sa-

kallah, 2003] et le diagnostic de pannes [Scholl et Becker, 2001; Mangassarian *et al.*, 2007; Staber et Bloem, 2007].

Parmi ces applications, deux d'entre elles nous intéressent plus particulièrement. En premier lieu, les jeux finis à deux joueurs permettent de donner l'intuition du sens des QBF. Enfin, la vérification de modèle est une bonne illustration du pouvoir de représentation des QBF comparativement à la logique propositionnelle.

1.4.1 Jeux finis à deux joueurs

Une manière intuitive de comprendre la sémantique d'une QBF prénexe $Q\ F$ est de regarder le problème comme un jeu où deux joueurs s'affrontent [Schaefer, 1976]. Un des joueurs est appelé joueur-$\exists$ et est associé au quantificateur existentiel ; l'autre joueur, appelé joueur-$\forall$, est associé au quantificateur universel. Les joueurs vont s'affronter pendant un nombre de tours fini et fixé à l'avance par le nombre d'alternances de quantificateurs dans lieur Q, d'où le qualificatif de jeu *fini*. Les « coups » à jouer sont déterminés par une lecture de la gauche vers la droite du lieur. L'objectif du joueur-$\exists$ est de satisfaire la formule F à l'aide des variables existentiellement quantifiées ; a contrario le joueur-$\forall$ essaye de faire perdre le joueur-$\exists$ en falsifiant F à l'aide des variables universellement quantifiées. Ainsi une QBF dans la classe de complexité $\Sigma_k^{\mathbf{P}}$ (resp. $\Pi_k^{\mathbf{P}}$) est un jeu fini booléen à deux joueurs en k coups, dans lequel le joueur-$\exists$ (resp. joueur-$\forall$) joue le premier. Aussi, il n'est pas étonnant de trouver parmi les problèmes dans **PSPACE** des généralisations de jeux finis à deux joueurs relativement connus comme le Go [Lichtenstein et Sipser, 1980], l'HEX [Even et Tarjan, 1976; Reisch, 1981], l'Othello [Iwata et Kasai, 1994], le Puissance 4 [Gent et Rowley, 2003], l'Amazons [Hearn, 2005] ou encore le Phutbal [Dereniowski, 2008].

Imaginons un jeu en 2 tours où le joueur-$\exists$ commence. Nous différencions les règles du jeu des conditions de victoire. Le joueur-$\exists$ va jouer le premier coup, représenté par un ensemble X_1 de variables, en respectant les règles codées par la formule $R_1^{\exists}(X_1)$. Il doit essayer de garantir la satisfaction des conditions de victoire $V(X_1, X_2)$ quel que soit le coup du joueur-$\forall$, représenté par un ensemble de variables X_2. Le joueur-$\forall$ va ensuite jouer le second et dernier coup en respectant les règles codées par $R_2^{\forall}(X_1, X_2)$ dépendantes du coup précédent (comme ne pas jouer sur une case occupée pour certains jeux de plateau), tout en essayant de falsifier les conditions de victoires pour le joueur-$\exists$. Ce jeu peut alors être représenté par la QBF

$$\exists X_1 \forall X_2 (R_1^{\exists}(X_1) \wedge (R_2^{\forall}(X_1, X_2) \rightarrow V(X_1, X_2))).$$

Il existe 3 façons pour le joueur-$\exists$ de gagner :

1. le joueur-$\exists$ joue les variables de X_1, respecte les règles $R_1^{\exists}(X_1)$ et remplit les conditions de victoires $V(X_1, X_2)$ sans avoir besoin de faire jouer le joueur-$\forall$;
2. le joueur-$\exists$ joue les variables de X_1, respecte les règles $R_1^{\exists}(X_1)$ et le joueur-$\forall$ joue les variables de X_2 mais enfreint les règles $R_2^{\forall}(X_1, X_2)$;
3. le joueur-$\exists$ joue les variables de X_1, respecte les règles $R_1^{\exists}(X_1)$, puis le joueur-$\forall$ joue les variables de X_2, respecte lui aussi les règles $R_2^{\forall}(X_1, X_2)$, et finalement les coups joués remplissent les conditions de victoires $V(X_1, X_2)$ pour le joueur-$\exists$.

Il est possible de généraliser à un jeu en k coups, en le représentant par la QBF

$$\exists X_1 \forall X_2 \dots q X_k (R_1^{\exists}(X_1) \wedge (R_2^{\forall}(X_1, X_2) \rightarrow (\dots (R_k^q(X_1, \dots, X_k)\, o\, V(X_1, \dots, X_k))))) \quad (1.1)$$

où $q = \exists$ et $o = \wedge$ (resp. $q = \forall$ et $o = \rightarrow$) si k est impair (resp. pair). De la même manière, un jeu en k coups où le joueur-$\forall$ commence peut être représenté par la QBF

$$\forall X_1 \exists X_2 \dots q X_k (R_1^{\forall}(X_1) \rightarrow (R_2^{\exists}(X_1, X_2) \wedge (\dots (R_k^q(X_1, \dots, X_k)\, o\, V(X_1, \dots, X_k))))) \quad (1.2)$$

où $q = \exists$ et $o = \wedge$ (resp. $q = \forall$ et $o = \rightarrow$) si k est pair (resp. impair). et le lien avec les classes de complexité dans la hiérarchie polynomiale. Les formules (1.1) et (1.2) sont construites de telles manières que le jeu se déroule en parcourant le lieur et la formule de la gauche vers la droite. Dès que le joueur-$\exists$ ne respecte pas les règles il perd ; dès que le joueur-$\forall$ ne respecte pas les règles, le joueur-$\exists$ gagne. Si personne n'enfreint les règles, nous savons si les conditions de victoires sont obtenues, seulement une fois que les k coups sont joués. Or, il est possible pour certains jeux que sans qu'aucun joueur n'enfreigne les règles, le joueur-$\exists$ gagne ou perde avant la fin de la partie. Il suffit d'inclure dans les règles une condition sous-entendue : un coup est légal si le jeu n'est pas déjà gagné ou perdu. Dans l'optique d'un parcours de la gauche vers la droite du lieur et de la formule, cette condition évite de jouer des coups inutiles. Malheureusement, aucune règle n'impose a priori de parcourir le lieur ou la formule d'une QBF dans un certain sens et la mise sous CNF d'un tel jeu détruit complètement la structure de la QBF le représentant. Intuitivement, les coups du joueur-$\forall$ ne respectant pas les règles sont répartis dans différentes clauses, rendant leurs détections difficiles. Une des méthodes pour résoudre un problème à l'aide d'une QBF est de le représenter assez intuitivement comme un jeu fini à deux joueurs. L'adversaire, le joueur-$\forall$, peut représenter aussi bien une personne, que l'environnement, l'inconnu ou encore le hasard. Il traduit l'approche *pessimiste* du problème : *face à tout ce que je ne maîtrise pas, je dois apporter une solution*. Le problème de représentation du joueur-$\forall$ sous CNF, aussi appelé *talon d'Achile des QBF* dans [Ansótegui *et al.*, 2005], ne touche donc pas seulement les applications ludiques, mais la plupart, pour ne pas dire la totalité des problèmes modélisés à l'aide de QBF mis ensuite sous CNF. Une solution envisagée est d'améliorer le codage afin de mieux guider les procédures QBF , voire d'adapter les procédures pour prendre en compte ces nouvelles informations [Ansótegui *et al.*, 2005]. Il est possible de rendre la détection des coups du joueur-$\forall$ ne respectant pas les règles plus facile, en codant le problème avec une forme alliant CNF et DNF, qui prend mieux en compte l'aspect symétrique des deux joueurs [Sabharwal *et al.*, 2006]. Une solution simple et assez radicale est de rajouter le concept de *quantification restreinte* aux QBF [Benedetti *et al.*, 2007; Benedetti et Mangassarian, 2008] dans un nouveau langage **QBF**$^+$. Les règles pour le joueur-$\exists$ sont forcément en conjonction avec le reste de la formule, alors que les règles pour le joueur-$\forall$ « impliquent » la suite de la formule. Dans ce langage ainsi restreint, la formule (1.1) s'écrit simplement

$$\exists X_1 [R_1^{\exists}(X_1)]\ \forall X_2 [R_2^{\forall}(X_1, X_2)]\ \dots\ q X_k [R_k^q(X_1, \dots, X_k)]\ V(X_1, \dots, X_k)$$

Pour k pair, cette formule se lit ainsi : *Il existe une affectation de X_1 telle que $R_1^{\exists}(X_1)$ et quelle que soit l'affectation de X_2 telle que $R_2^{\exists}(X_1, X_2)$, ... il existe une affectation de X_k telle*

que $R_k^{\exists}(X_1, \ldots, X_k)$ *et* $V(X_1, \ldots, X_k)$. La dernière solution, qui nous semble la plus élégante, est de travailler sur la formulation d'origine qui contient toute l'information et peut toujours être transformée en interne pour des besoins calculatoires.

1.4.2 Vérification de modèle

La vérification de modèle (*model checking* en anglais) désigne un ensemble de techniques pour la vérification de systèmes dynamiques. L'objectif est de vérifier automatiquement qu'un modèle, un système ou sa représentation, vérifie une spécification ou une propriété, souvent exprimée à l'aide d'une logique temporelle. Ces logiques sont souvent interprétées dans des structures de Kripke [Browne *et al.*, 1988; Biere *et al.*, 1999b], d'où vient le terme « modèle ». Une structure de Kripke est un automate fini non déterministe décrit par un 4-uplet $\{S, I, T, \mathcal{L}\}$: un ensemble fini de n états S, un ensemble d'états initiaux $I \subseteq S$, une relation de transition $T \subseteq S \times S$ et une fonction d'étiquetage des états $\mathcal{L}$ par des propositions. La relation de transition possède une condition propre aux systèmes dynamiques : chaque état doit avoir un successeur dans T. Par conséquent, il est toujours possible de construire un chemin infini dans une structure de Kripke. Par la suite, nous considérons que chaque état s_i, pour $0 \leq i \leq n$ peut être représenté par un vecteur de variables booléennes. Un état $s \in I$ est reconnu comme un état initial et est noté $I(s)$. La transition d'un état s_i à un état s_{i+1} est valide si et seulement si $(s_i, s_{i+1}) \in T$, noté plus simplement $T(s_i, s_{i+1})$. Une séquence d'états $s_0, s_1, \ldots, s_k$ est un *chemin valide*, si toutes les transitions d'un état i à $i+1$ sont valides (ie. $(s_i, s_{i+1}) \in T$), avec $0 \leq i < k$.

La vérification de modèle pour la propriété de fiabilité/sécurité (*safety property* en anglais) est un problème classique en vérification formelle, nommé aussi problème d'accessibilité. À partir d'un ensemble d'états mauvais, à ne pas atteindre, le problème s'exprime intuitivement ainsi : *est-il vrai qu'aucun mauvais état n'est accessible à partir des états initiaux ?*. Il n'y a pas de borne a priori sur la longueur des chemins prouvant l'accès à un état proscrit. C'est pourquoi, le terme *unbounded model checking* est parfois employé pour faire référence à ce problème. Nous nous intéressons à la variante de ce problème, appelé par opposition, *bounded model checking* (BMC), qui consiste à vérifier si un état mauvais est accessible en k transitions ou moins, avec k une borne fixée à l'avance. Un état s interdit, est reconnu par $B(s)$. Il est alors possible de réduire ce problème à SAT :

$$I(s_0) \wedge T(s_0, s_1) \wedge \cdots \wedge T(s_{k-1}, s_k) \wedge B(s_k) \tag{1.3}$$

L'inconvénient majeur de cette formulation est la duplication de la relation de transition k fois. Cette relation est la partie la plus complexe et peut représenter des dizaines de milliers de clauses et le rayon d'analyse d'un système peut être en milliers de pas. Pourtant, la vérification bornée de modèle est en pratique très utilisée pour détecter les mauvais comportements des systèmes. Avec des problèmes de grande taille, cette formulation nécessite une quantité trop importante de ressources mémoires. C'est donc assez logiquement, que la communauté de la vérification formelle s'est penchée sur une approche utilisant les QBF afin de ne pas avoir à dupliquer la relation de transition [Rintanen, 2001; Mneimneh et Sakallah, 2003; Dershowitz *et al.*, 2005; Jussila et Biere, 2006]. L'objectif du

codage avec le langage **QBF** est de représenter l'idée suivante : *Pour tous les états x et x', si ceux-ci sont adjacents dans la séquence testée, alors ils sont consistants avec la relation de transition*. Pour notre précédente séquence d'états $s_0, \ldots, s_k$, il est possible d'écrire :

$$\forall x \forall x'([((x \leftrightarrow s_0) \wedge (x' \leftrightarrow s_1)) \vee \cdots \vee ((x \leftrightarrow s_{k-1}) \wedge (x' \leftrightarrow s_k))] \rightarrow T(x, x'))$$

Pour la suite, nous nommons cette formulation $T_\forall(s_0, \ldots, s_k)$ et présentons la formulation QBF naïve du problème BMC :

$$\exists s_0 \ldots \exists s_k (I(s_0) \wedge T_\forall(s_0, \ldots, s_k) \wedge B(s_k)) \tag{1.4}$$

En général, la relation de transition est le plus grand élément de la formule. Ce codage est efficace en espace : il permet pratiquement de diviser par k l'espace nécessaire comparativement à la formule (1.3). D'un point de vue performance, ce codage n'est pas avantageux et peut être grandement amélioré pour mieux guider les procédures. Comme ce n'est pas l'aspect qui nous intéresse ici, nous guidons le lecteur vers l'article *Experience and Perspectives in QBF-based Formal Verification* [Benedetti et Mangassarian, 2008] pour plus de détails sur le codage et la vérification en pratique.

La capture d'un motif dans le but de ne pas le répéter est une illustration de l'expressivité offerte par les QBF. Il est possible de faire le lien entre les jeux finis à deux joueurs, qui donnent l'intuition de la sémantique des QBF et le codage de BMC. Si nous regardons attentivement, la formulation (1.1) ou (1.2) des jeux finis à deux joueurs, nous remarquons qu'à chaque quantificateur correspond une formule vérifiant si les règles ont été respectées. Puisque pour de très nombreux jeux, les règles restent fixes au cours d'une partie, n'est-il pas possible de les exprimer une seule fois et d'y faire ensuite référence ? Cette question, que nous laissons volontairement sans réponse, témoigne du travail encore nécessaire pour *bien « programmer »* avec le langage **QBF**, en parallèle de la recherche de procédures performantes.

Première partie

Algorithmes séquentiels pour QBF

Chapitre 2

État de l'art pour le problème de validité des QBF

Sommaire

2.1 Seize années de procédures de décision pour la validité des QBF

Alors que les formules booléennes quantifiées ont été introduites dans les années 70 par Meyer et Stockmeyer [Meyer et Stockmeyer, 1972; Stockmeyer et Meyer, 1973; Stockmeyer, 1977], la première procédure de décision a été présentée une vingtaine d'année plus tard par Büning en 1995 [Büning *et al.*, 1995] et les premiers résultats pratiques ont été présentés en 1997 par Cadoli, Giovanardi et Schaerf [Cadoli *et al.*, 1997; Cadoli *et al.*, 1998]. Bien que le problème de décision pour QBF soit **PSPACE**-complet, les progrès en puissance de calcul ont permis d'envisager de s'attaquer à ce dernier.

Afin de faire une « photographie » de 16 années de recherche dans le domaine des procédures pour QBF, nous listons dans les tables 2 et 3 (p. 39 et 40), les procédures, les articles les décrivant et leurs grandes caractéristiques. Cette liste, bien que volumineuse n'est probablement pas exhaustive à en voir les participants de la compétition *QBFEVAL* [Giunchiglia *et al.*, 2001a; Berre *et al.*, 2004; Narizzano *et al.*, 2006; Peschiera *et al.*, 2010]. Nous avons mentionné uniquement 35 procédures décrites suffisamment, c'est-à-dire dont le travail a fait l'objet d'un ou plusieurs articles. Nous n'avons par exemple pas mentionné `qmaiga` qui est un assemblage de `AIGsolve` et `QMiraXT`. La figure 3 présente par année le volume de nouvelles procédures de décision pour le problème de validité des QBF.

Avant 2001, les formules booléennes étaient un langage « jouet », ie. une bonne illustration de la hiérarchie polynomiale. Peu de personnes s'étaient intéressées à la résolution en pratique de problèmes exprimés dans ce langage.

Entre 2001 et 2006, des problèmes réels issus de la vérification formelle ont fait leur apparition, ainsi que les premiers résultats : très décevants [Scholl et Becker, 2001; Mneimneh et Sakallah, 2003; Dershowitz *et al.*, 2005; Jussila et Biere, 2006]. Les QBF permettent de coder des problèmes de vérification formelle de façon plus succincte comparativement au codage SAT. Les conclusions sont sans appel : « *We found that modern state-of-the-art general-purpose QBF solvers are still unable to handle the real-life instances of BMC problems in an efficient manner*» [Dershowitz *et al.*, 2005] ; ou encore : « *Our results clearly show that much more research in QBF is needed to be able to use QBF as alternative to SAT based model checking, even in the bounded case*» [Jussila et Biere, 2006]. Cette période a été particulièrement prolifique en nombre de nouvelles procédures et par conséquent en nouvelles techniques (*retour arrière intelligent, apprentissage de lemmes, arbre de quantificateurs, encodage CNF/DNF, propagation « don't care », etc.*). Entre 2007 et 2010, un nombre moins important de nouvelles procédures est apparu (la compétition *QBFEVAL* se fera désormais tous les 2 ans) mais de nouvelles techniques continuent de voir le jour et viennent améliorer les procédures existantes. La procédure `QuBE`, en version 7 en 2010, est un bon exemple de procédure qui n'a cessé d'évoluer depuis 2001 : nous pouvons recenser des articles traitant du *retour arrière intelligent* ou *backjumping* [Giunchiglia *et al.*, 2001b; Giunchiglia *et al.*, 2003], de *l'apprentissage de clauses et de cubes* [Giunchiglia *et al.*, 2002; Giunchiglia *et al.*, 2004a; Giunchiglia *et al.*, 2006a], de *littéraux monotones* [Giunchiglia *et al.*, 2004a], d'une structure de données spécifique, les « *Watched literals* » [Giunchiglia *et*

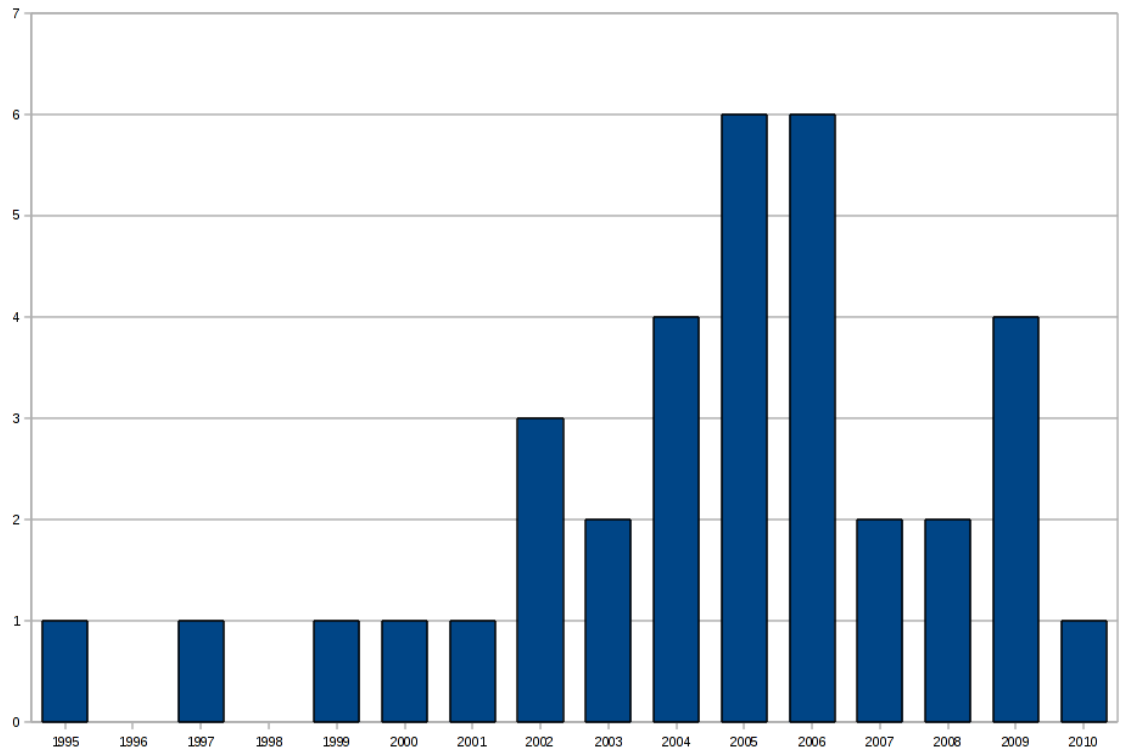


FIGURE 3 – Présentations de nouvelles procédures de décision pour le problème de validité des QBF au cours des 16 dernières années.

al., 2004b], d'une *structure de quantificateurs* [Giunchiglia *et al.*, 2006b; Giunchiglia *et al.*, 2007], d'une *procédure parallèle* `PaQube`, extension de `QuBE` [Lewis *et al.*, 2009a] et finalement d'un *processus de pré-traitement* `sQueezeBF` [Giunchiglia *et al.*, 2010].

Aujourd'hui, la plupart des problèmes réels à notre disposition sont issus de la vérification formelle. De nombreux progrès ont été réalisés, tant au niveau des procédures que du codage des problèmes [Benedetti et Mangassarian, 2008; Mangassarian *et al.*, 2010]. Les QBF sont aujourd'hui capables d'être un langage cible pour certains problèmes tant les procédures actuelles sont capables de surpasser les procédures SAT. Malgré tout, les QBF ne connaissent toujours pas le succès qu'à connu SAT il y a quelques années, peut être parce que les procédures ne sont pas à leur pleine maturité et sûrement parce qu'entre SAT et QBF il existe de réelles difficultés lors du codage d'un problème [Benedetti et Mangassarian, 2008; Mangassarian *et al.*, 2010] puis lors de sa transformation dans la forme acceptée majoritairement par les procédures : CNF prénexe [Zhang et Malik, 2002b; Egly *et al.*, 2003; Ansótegui *et al.*, 2005; Zhang, 2006; Sabharwal *et al.*, 2006]. La figure 4 présente pour chaque année le nombre de procédures disponibles en fonction du format accepté en entrée. En 2010, sur les 35 procédures que nous avons analysées :

– 91% des procédures acceptent une entrée prénexe ;
– 74% des procédures acceptent une entrée prénexe CNF ;

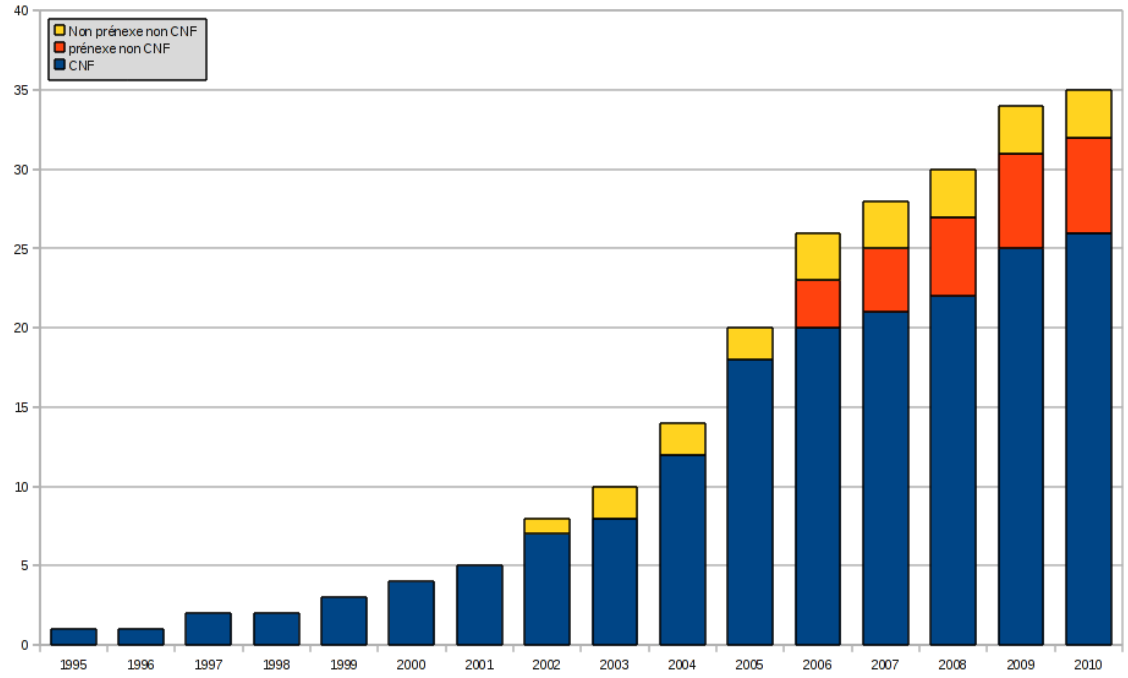


FIGURE 4 – Évolution du nombre de procédures de décision disponibles pour le problème de validité des QBF en fonction du format accepté en entrée au cours des 16 dernières années.

- 9% des procédures acceptent une entrée non prénexe ;
- seule une des procédures accepte en entrée des QBF quelconques avec potentiellement des bi-implications : `QSAT`. Mais il n'existe aucune implantation connue pour QBF ;
- 2 procédures acceptent des QBF prénexes quelconques avec potentiellement des bi-implications : `pQBF` et `Qbf2nlp`.

La conception de procédures acceptant des QBF quelconques est aussi limitée par le nombre de formats standards proposés. La majorité des problèmes sont disponibles sous CNF prénexe dans le format *QDIMACS*. Deux formats permettent d'écrire des QBF non prénexe *QBF1.0* et *qpro*. Le format de la procédure `qpro` est devenu le nouveau standard pour la compétition *QBFEVAL 2010, section non CNF non prénexe*, il s'agit d'un format NNF non prénexe. Aucun format n'a été proposé pour des QBF quelconques contenant des bi-implications et des ou-exclusifs, il n'y a donc aucun problème disponible dans ce fragment du langage des QBF. L'apparition de problèmes issus du monde réel au format *QDIMACS* avait fait passer les QBF de l'état de « jouet » à l'état d'alternative prometteuse, puis à l'état d'alternative compétitive. Peut être que le même phénomène, pour un fragment plus riche du langage QBF, se produira d'ici quelques années pour les procédures travaillant sur ce fragment.

Algorithme 1 : $QDLL$

Données :
Un lieur $Q = q_1 x_1 q_2 x_2 ... q_n x_n$.
Une matrice F' d'une QBF prénexe telle que $F = Q\ F'$.
Résultat : Vrai (resp. Faux) si F est valide (resp. non valide).
début
  $Q\ F' = simplifier(Q, F')$
  si $F' \equiv \top$ **alors**
    retourner Vrai ; `/* succès */`
  sinon
    retourner Faux ; `/* échec */`
  fin
  $Q = q_1 x_1 Q'$; `/* q1x1 choisis à l'aide d'une heuristique */`
  si $q_1 = \exists$ **alors**
    retourner $QDLL(Q', [x_1 \leftarrow \top](F'))$ **||** $QDLL(Q', [x_1 \leftarrow \bot](F'))$
  sinon
    retourner $QDLL(Q', [x_1 \leftarrow \top](F'))$ **&&** $QDLL(Q', [x_1 \leftarrow \bot](F'))$
  fin
fin

2.2 Extension de la procédure de Davis, Logemann et Loveland

Nous avons présenté les formules booléennes quantifiées (cf. section 1.2) comme étant une extension des formules propositionnelles. C'est assez naturellement, que les premièrs résultats pratiques [Cadoli *et al.*, 1997; Cadoli *et al.*, 1998] ont été produits par une extension de la procédure de Davis, Logemann et Loveland pour SAT [Davis *et al.*, 1962], appelée *DLL*. Cette procédure et toutes les optimisations qui lui ont été apportées, ont fait le succès des procédures SAT. Son extension aux QBF est appelée de manière générale *QDLL*. Comme pour DLL, il s'agit d'un algorithme de recherche, les améliorations apportées avant (les pré-traitements) et pendant la recherche ont pour objectif de réduire l'espace de recherche parcouru.

L'algorithme 1 décrit en pseudo-code l'extension de la procédure de Davis, Logemann et Loveland pour les QBF. Pour des raisons pédagogiques, nous ne considérons que les QBF prénexes. L'algorithme parcourt le lieur de l'extérieur vers l'intérieur ; de manière générale cette façon de parcourir le lieur est appelée « *top-down* » dans la littérature. Le processus inverse est appelé « *bottom-up* ». Le point central de la procédure réside dans la sémantique des quantificateurs exprimée par les deux équivalences du théorème 1.2.12, rappelées ci-dessous, avec Q un lieur et F une matrice :

$$(\exists x Q\ F) \equiv ([x \leftarrow \bot](Q\ F) \vee [x \leftarrow \top](Q\ F))$$
$$(\forall x Q\ F) \equiv ([x \leftarrow \bot](Q\ F) \wedge [x \leftarrow \top](Q\ F))$$

Dans le pseudo code que nous présentons, nous avons sciemment remplacé le $\vee$ (resp. $\wedge$) par la version du langage de programmation C, le `||` (resp. le **&&**). Ces tests booléens séquentiels traduisent mieux l'idée de l'algorithme : dans le cas du $\vee$ (resp. $\wedge$), si le premier test retourne **vrai** pour un succès (resp. **faux** pour un échec), il est inutile de faire le second test. Le fait de rendre les tests séquentiels ramène à la notion de *retour arrière* ou *backtrack* nécessaire avec une machine déterministe. L'efficacité de la procédure réside essentiellement dans l'adjonction de deux fonctions heuristiques qui choisissent la variable pour continuer la recherche et la première valeur à tester. De même, la fonction $simplifier$ qui applique un ensemble de règles ayant comme objectif d'obtenir plus vite un échec ou un succès est également essentielle. Pour `Evaluate`, la fonction $simplifier$ teste les clauses unitaires, les littéraux monotones, la vérité et la fausseté triviale et la falsification de clauses par paires. Contrairement à la procédure DLL pour SAT, le choix de la variable est limité par le lieur. En effet l'équivalence et la non-équivalence suivante tiennent dans le cas général pour Q un lieur, F une matrice, q et q' deux quantificateurs tels que $q \neq q'$, x et y deux variables booléennes :

$$qxqyQ\ F \equiv qyqxQ\ F$$
$$qxq'yQ\ F \not\equiv q'yqxQ\ F$$

Dit autrement, le choix de la variable est limitée aux variables consécutives quantifiées de la même façon. Ceci est un sérieux handicap. Soit la formule booléenne quantifiée sous forme prénexe $F = \forall x_1 ... \forall x_n \exists y\ (y \vee G)$, avec G une formule booléenne. F est trivialement valide dès lors que y prend la valeur **vrai** quelles que soient les valeurs des autres variables. Or, il n'est pas possible de choisir la variable y tant qu'elle est sous la portée d'un quantificateur universel. Quel que soit l'ordre dans lequel seront choisies $x_1, .., x_n$, les 2^{n-1} appels récursifs auront lieu et tous se termineront positivement, au pire grâce à l'occurrence de la variable y dans la disjonction. Nous voyons bien que les contraintes pesant sur le choix de la variable ne permettent pas d'espérer autant de gain que dans le cas de SAT.

L'algorithme 1, qui assure la complétude de la recherche et son caractère polynomial en espace, n'est visiblement pas suffisant pour décider efficacement de la validité des QBF. Mais de nombreuses améliorations ont été apportées. Cet algorithme est la base de 19 procédures sur 35, soit 54% du total : `Evaluate`, `decide`, `QSOLVE`, `QuBE`, `Semprop`, `Quaffle`, `WalkQSAT`, `ZQSAT`, `CLearn`, `S-QBF`, `Q-PREZ`, `pQBF`, `2clsQ`, `IQTest`, `Duaffle`, `AB`, `CirQit`, `QMiraXT(1T)`, `DepQBF`. La procédure `StruQS` alterne entre cet algorithme et celui présenté dans la section 2.3. Le système `QBDD(DLL)` utilise cet algorithme comme générateur de modèles. La procédure `qpro` est issue du domaine de la théorie de la preuve, plus précisément du *calcul des séquents* et du *système G de Gentzen* [Gallier, 1985]. Toutefois, la procédure est très proche d'un QDLL du fait de l'application des règles d'élimination de quantificateurs de type « top-down ».

2.3 Extension(s) de la résolution et de la procédure de Davis et Putnam

La *résolution propositionnelle*, issue de la *théorie de la preuve*, est un algorithme pour la décision du problème de satisfiabilité des formules propositionnelles sous *CNF*, dont l'objectif est de prouver l'insatisfiabilité par l'obtention de la clause vide. L'inconvénient majeur de cette méthode, est la possibilité non négligeable d'avoir besoin d'une quantité exponentielle de ressources [Galil, 1977; Haken, 1985]. La procédure `QKN` [Büning *et al.*, 1995] ou *Q-résolution* est l'extension de la *résolution* aux QBF par adjonction de la *réduction universelle*. La résolution propositionnelle décide de l'insatisifiablité d'une proposition par l'obtention d'une clause vide. À partir d'un ensemble de clauses, deux sont choisies, contenant des littéraux complémentaires, et une nouvelle clause est produite par application de l'équivalence

$$((\neg A \vee B) \wedge (A \vee C)) \equiv ((\neg A \vee B) \wedge (A \vee C) \wedge (B \vee C)).$$

La réduction universelle consiste à ôter d'une clause toute variable universellement quantifiée pour laquelle son quantificateur est plus interne que celui des variables existentiellement quantifiées de cette clause. La Q-résolution décide de la non validité d'une QBF par l'obtention d'une clause vide.

La procédure de Davis et Putnam [Davis et Putnam, 1960], appelée *DP*, est aussi un algorithme pour SAT. La multi-résolution consiste à appliquer toutes les résolutions possibles sur un symbole propositionnel afin d'éliminer toutes ses occurrences. Cet algorithme effectue la *multi-résolution* jusqu'à obtenir $\top$ ou $\bot$. Bien qu'inefficace sur des problèmes aléatoires, la procédure DP peut s'avérer intéressante pour des problèmes structurés [Dechter et Rish, 1994; Rish et Dechter, 2000]. Pour le problème de besoin mémoire exponentiel, des progrès ont été réalisés pour le problème SAT. La procédure `ZRES` [Chatalic et Simon, 2000a; Chatalic et Simon, 2000b; Chatalic et Simon, 2001] propose de représenter l'ensemble des clauses par des *diagrammes de décision binaires zéro-réduits* ou *ZBDD* [ichi Minato, 1993]. Cette structure, aux propriétés intéressantes pour compresser des ensembles de clauses, vient de la volonté de représenter des formules booléennes à l'aide de *graphes orientés acycliques* (DAG) et plus précisément à l'aide de *diagrammes de décision binaires* ou *BDD* [Lee, 1959; Akers, 1978]. De nouveaux opérateurs ensemblistes pour manipuler des ensembles de clauses sont introduits et la procédure DP est capable ainsi de travailler sur un nombre très important de clauses compressées (les auteurs parlent de 10^{60} clauses). Elle surpasse même les procédures de type DLL sur certains problèmes structurés. Dans la même veine, Pan et Vardi proposent de travailler avec un autre type de BDD, les *diagrammes de décision binaires ordonnés* ou *OBDD* [Pan et Vardi, 2004a]. L'extension de DP aux QBF, appelée QDP, souffre du même inconvénient que la version propositionnelle, à savoir un besoin potentiellement exponentiel en mémoire. Les procédures `QMRES` et `QBDD` [Pan et Vardi, 2004b] sont des adaptations aux QBF du travail sur DP et la résolution à l'aide de BDD (ZBDD et OBDD). La procédure `QMRES` s'est avérée être compétitive avec les solveurs QDLL au moment de la publication de l'article.

L'algorithme 2 décrit en pseudo-code QDP. Le point central de la procédure réside

Algorithme 2 : QDP

Données :
Un lieur $Q = q_1x_1q_2x_2...q_nx_n$.
Une matrice F' d'une QBF prénexe CNF telle que $F = Q\ F'$.
Résultat : Valide ou Non Valide.
début
  $Q\ F' = simplifier(Q, F')$
  si $F' \equiv \top$ **alors**
    retourner Valide
  fin
  si $F' \equiv \bot$ **alors**
    retourner Non Valide
  fin
  $q_kx_k = choix\_variable(Q)$; /* avec $1 \leq k \leq n$ */
  $F' = ((x_k \vee F^+) \wedge (\neg x_k \vee F^-) \wedge F'')$
  $Q = Q_1q_kx_kQ_2$
  /* avec x_k absente de F^+, F^- et F''. Si $q_k = \forall$ alors les variables quantifiées existentiellement dans le lieur Q_2 sont absentes des formules F^- et F^+. */
  si $q_k = \exists$ **alors**
    /* multi-résolution sur x_k */
    retourner $QDP(Q_1Q_2, cnf((F^+ \vee F^-) \wedge F''))$
  sinon
    /* élimination (interne) de x_k */
    retourner $QDP(Q_1Q_2, ((F^+ \wedge F^-) \wedge F''))$
  fin
fin

dans la *multi-résolution propositionnelle* et *l'élimination interne* exprimée par les équivalences ci-dessous :

$$Q\exists xQ'\ ((x \vee F^+) \wedge (\neg x \vee F^-) \wedge F'') \equiv QQ'((F^+ \vee F^-) \wedge F'')$$
$$Q\forall xQ''\ ((x \vee F^+) \wedge (\neg x \vee F^-) \wedge F'') \equiv QQ''((F^+ \wedge F^-) \wedge F'')$$

où Q, Q' et Q'' sont des lieurs, F^-, F^+ et F'' sont des formules booléennes (non quantifiées) qui ne contiennent pas la variable x. Les variables quantifiées existentiellement dans le lieur Q'' sont absentes des formules F^- et F^+. Obtenir une formule de la forme $((x \vee F^+) \wedge (\neg x \vee F^-) \wedge F'')$ est trivial sous CNF. Il suffit d'extraire M^+ (resp.M^-) l'ensemble des clauses contenant x (resp. $\neg x$), alors $M^+ \equiv (x \vee F^+)$ (resp. $M^- \equiv (\neg x \vee F^-)$), où F^+ (resp. F^-) est l'ensemble de clauses M^+ (resp.M^-) auxquelles les occurrences de x (resp. $\neg x$) ont été supprimées. Sous cette forme, la *réduction universelle* est immédiate et « gratuite ». Dans le cas existentiel on retrouve le même problème qu'au propositionnel : il faut mettre sous CNF la disjonction de deux ensembles de clauses. Or, il est impossible d'utiliser la méthode par *renommage de sous-formules* [Tseitin, 1970] (détaillée dans la

section 3.2.1) qui rajoute de nouvelles variables, sous peine que la procédure ne termine jamais. La seule solution est d'utiliser la distributivité de $\vee$ et du $\wedge$ (la méthode dite *académique*) qui fait croître exponentiellement le nombre de clauses. Une solution pour limiter cette croissance, est de travailler sur un ensemble de clauses compressées à l'aide de BDD [Pan et Vardi, 2004a].

La *Q-résolution* et l'algorithme QDP sont la base de 5 procédures sur 35, soit 14% du total : `QKN`, `quantor`, `QMRES`, `QuBIS`, `AIGsolve`. Plus généralement, un certain nombre d'améliorations apportées à QDLL avant (pré-traitement) et pendant la recherche font appel à la résolution. Par exemple, le pré-traitement `sQueezeBF` applique la Q-résolution [Giunchiglia *et al.*, 2010], de même, la procédure `sKizzo` [Benedetti, 2005b] applique la Q-résolution dans une limite d'espace avant d'effectuer la *skolémisation symbolique*. La procédure `2clsQ` [Samulowitz et Bacchus, 2006] applique l'*hyper-résolution binaire* [Bacchus et Winter, 2003], une extension du principe de résolution propositionnelle, présentée comme un pré-traitement.

Le *retour arrière intelligent* ou *backjumping* est une forme *d'apprentissage de lemmes* (*learning*), technique issue de l'amélioration des procédures SAT. L'objectif est « d'apprendre » des échecs afin de ne pas parcourir inutilement certaines portions de l'espace de recherche. Étendu aux QBF, ce principe consiste à apprendre des échecs et des succès. Cet apprentissage peut être vu comme une façon de faire coopérer la résolution et la recherche [Pulina et Tacchella, 2009]. Cela s'accompagne de l'inconvénient de la résolution : le nombre de clauses apprises peut être exponentiel [Letz, 2002]. Il faut donc sélectionner et apprendre un nombre raisonnable de clauses bien choisies. En 2002, trois procédures intègrent l'apprentissage de lemmes : `QuBE` [Giunchiglia *et al.*, 2002], `Semprop` [Letz, 2002] et `Quaffle` [Zhang et Malik, 2002a; Zhang et Malik, 2002b]. *Backjumping* ou *learning* sont aujourd'hui au cœur de la plupart des procédures QBF basée sur QDLL.

2.4 Procédures par élimination de quantificateurs

La procédure QDLL que nous avons présentée possède deux restrictions majeures. Premièrement, la formule doit être prénexe et souvent les procédures de l'état de l'art requièrent une forme plus restrictive. Deuxièmement, à chaque itération, la procédure peut choisir une variable uniquement parmi les plus externes. L'avantage de cette procédure est son caractère polynomial en espace. La Q-résolution et la procédure QDP possèdent deux restrictions importantes et un inconvénient de taille. Premièrement, La formule en entrée doit être sous forme normale conjonctive. Deuxièmement, pour éliminer une variable universellement quantifiée, dans les clauses concernées, celle-ci doit être plus interne que toute autre variable quantifiée existentiellement. Ces deux procédures sont dans le pire des cas exponentielles en espace.

La procédure `Élimination` que nous présentons maintenant ne possède pas de restriction sur la formule en entrée et permet de choisir une variable quelconque à chaque itération. Le point central de cette procédure réside dans l'application de la sémantique des quantificateurs afin d'éliminer une variable et son quantificateur associé. La QBF pouvant être non prénexe, la sémantique des quantificateurs peut s'appliquer à une sous-

Algorithme 3 : Élimination

Données : Une QBF F quelconque.
Résultat : Valide ou Non Valide.
début
  $Q\ F' = simplifier(Q, F')$
  si $F \equiv \top$ **alors**
    retourner Valide
  fin
  si $F \equiv \bot$ **alors**
    retourner Non Valide
  fin
  $qx\ G = choix\_sous\_formule(F)$
  $F = [y \leftarrow (qx\ G)](F')$
  /* G est une QBF sous la portée du quantificateur q portant sur la variable x. Au moins une occurence libre de y est présente dans F'. */
  si $q = \exists$ **alors**
    retourner *élimination*$([y \leftarrow ([x \leftarrow \top](G) \vee [x \leftarrow \bot](G))](F'))$
  sinon
    retourner *élimination*$([y \leftarrow ([x \leftarrow \top](G) \wedge [x \leftarrow \bot](G))](F'))$
  fin
fin

formule. Soient F, F' et G des QBF, q un quantificateur, x et y des variables, avec au moins une occurrence libre de y dans F'. Alors, d'une façon générale, $(qx\ G)$ est une sous-formule de F si $F = [y \leftarrow (qx\ G)](F')$. L'application de la sémantique de q est exprimée dans les équivalences suivantes.

$$[y \leftarrow (\exists x\ G)](F') \equiv [y \leftarrow ([x \leftarrow \top](G) \vee [x \leftarrow \bot](G))](F')$$
$$[y \leftarrow (\forall x\ G)](F') \equiv [y \leftarrow ([x \leftarrow \top](G) \wedge [x \leftarrow \bot](G))](F')$$

La QBF G et les possibles quantificateurs présents à l'intérieur sont recopiés deux fois . Le cas échéant, ceux-ci devront être renommés pour lever toute ambiguïté. Dans le pire des cas l'algorithme 3 est exponentiel en espace. Cette procédure peut être efficace avec par exemple l'adjonction de la *minimisation de portée* ou *miniscoping* [Ayari et Basin, 2002] et avec une approche « *bottom-up* ». Cela a donné naissance à la procédure `QUBOS`. Si seul le quantificateur le plus externe est choisi à chaque itération, nous retrouvons la procédure QDLL pour les QBF prénexes et son caractère polynomial en espace. Il est également possible de retrouver l'algorithme QDP pour les QBF prénexes sous CNF, en éliminant uniquement les variables universellement quantifiées les plus internes. L'algorithme `Élimination` est la base de 2 procédures sur 35, `QUBOS` et `Nenofex`, soit 6% du total.

2.5 Des procédures avec des philosophies différentes

Les algorithmes présentés précédemment sont les plus représentés, mais certaines procédures tentent des approches différentes. Nous avons isolé 4 types de mécanismes dont la différence est parfois ténue. Certaines procédures déjà citées se retrouvent de nouveau évoquées ici. Enfin, nous nous intéressons à quelques approches transversales.

Les procédures incomplètes. Le problème de décision pour formules booléennes quantifiées étant **PSPACE**-complet, vouloir une procédure complète et efficace peut paraître déraisonnable, c'est pourquoi une approche stochastique peut être envisagée. Le prix à payer est l'incapacité à conclure en cas d'arrêt sans succès de la procédure. Dans le domaine SAT, la procédure `WalkSAT` [Selman et Kautz, 1993] avait exploré cette voie avec succès. Pour SAT, vérifier si une valuation est un modèle se fait en temps polynomial. Pour QBF, le modèle propositionnel d'une QBF (cf. section 1.2.2) est insuffisant pour certifier de la validité d'une QBF. Il faut lui adjoindre une *politique* [Coste-Marquis *et al.*, 2006b; Büning *et al.*, 2007], qui peut être vu comme un modèle fonctionnel des QBF. Malheureusement, vérifier si un couple *politique/modèle propositionnel* certifie de la validité d'une QBF est **coNP**. Par conséquent, peu de procédures incomplètes ont été proposées, elles sont au nombre de deux : `WalkQSAT`, une extension de `WalkSAT`, et `QBDD(LS)`. La procédure `QBDD(LS)` utilise en fait une procédure incomplète pour générer des modèles SAT, agrégés ensuite pour former un modèle QBF. Cette façon de procéder la catégorise aussi comme *procédure à oracles*.

Les procédures à oracles. L'esprit d'une procédure que nous appelons « *procédure à oracles* » est de faire appel à une procédure de résolution pour un problème de plus faible complexité, comme SAT. La procédure pose plusieurs questions à cet oracle afin d'avancer dans sa quête de validité. Nous avons recensé trois procédures de ce type : `QSAT`, `QBDD(LS)` et `QBDD(DLL)`. Dans ces trois procédures la question posée à l'oracle n'est pas exactement la satisfiabilité d'une proposition. La procédure `QSAT` demande tous les modèles, alors que `QBDD(LS)` et `QBDD(DLL)` demande « juste » un modèle. La procédure `QSAT` fait de l'élimination de quantificateurs « bottom-up », mais son originalité réside dans l'appel à un oracle. Cette procédure sera détaillée dans le chapitre 5.

Les procédures de transformation. Les procédures que nous nommons « *procédures de transformation* » sont très proches des procédures à oracles. L'idée est de faire appel à une procédure de complexité égale ou inférieur après avoir traduit le problème sous un autre formalisme. Par exemple, la procédure `sKizzo` traduit la QBF en proposition (SAT) à l'aide de la *skolémisation symbolique* [Benedetti, 2004]. Les besoins en mémoire de cette traduction sont potentiellement exponentiels. La procédure `sKizzo` est alors prévue dans ce cas pour se comporter comme une procédure à oracles, en posant plusieurs problèmes à la procédure SAT. La procédure `Qbf2nlp` [Stéphan *et al.*, 2009] traduit la QBF d'origine en *programme ASP* (« *Answer Set Programming* », soit programmation par ensembles de réponses) avec variables. Ce programme est ensuite passé à un « *grounder* » , qui parmi

d'autres traitements va transformer le programme avec variables en un programme propositionnel. Cette phase d'instantiation est généralement exponentielle en espace pour les programmes issus de la traduction d'une QBF. Finalement, la QBF sera décidée à l'aide d'une procédure de résolution pour les ASP après avoir subi ces deux transformations. La procédure `QuBIS` peut être utilisée comme pré-traitement. L'élimination de quantificateur « bottom-up » permet d'obtenir une formule avec un seul type de quantificateur, le problème peut alors être résolu par une procédure SAT presque toujours de type DLL. C'est le cas de `QUBOS` et `Nenofex`.

Les procédure(s) portfolio(s). Le terme *portfolio* est emprunté à la terminologie des procédures parallèles. Une procédure (ou plutôt méta-procédure) portfolio dispose d'un certain nombre de procédures pour résoudre son problème. Avec une seule unité de calcul, il faut essayer de prévoir quelle sera la meilleure en fonction du problème en entrée. Seule la procédure `AQME` [Pulina et Tacchella, 2007] procède ainsi. La propriété de *largeur de l'arbre* ou *treewidth* est un bon marqueur de la difficulté d'une QBF [Pulina et Tacchella, 2008b] et sert de critère pour le choix du solveur. La procédure est d'abord entraînée sur un échantillon de QBF afin de réaliser une classification pour déterminer la meilleur procédure à exécuter en fonction des attributs de la QBF. Une fois entraînée la procédure peut être utilisée de façon classique : elle choisira d'elle même la procédure qui lui semble la plus performante. Dans la dernière version d'`AQME`, la phase d'entraînement n'est plus présente : la procédure est dite *auto-adaptative* [Pulina, 2010].

Les approches transversales. Nous avons vu au chapitre 1 que le problème QSAT est **PSPACE**-complet. Or, la classe de complexité **PSPACE** contient un ensemble de classes vérifiant les relations d'inclusion : $\mathbf{P} \subseteq (\mathbf{NP} \cap \mathbf{coNP}) \subseteq (\mathbf{\Sigma_2^P} \cap \mathbf{\Pi_2^P}) \subseteq \cdots \subseteq \mathbf{HP} \subseteq \mathbf{PSPACE}$. Pour certains fragments du langage **QBF**, décider de la validité d'un mot de ce langage est de complexité inférieur à **PSPACE**. Par exemple, le problème de validité d'une QBF prénexe close dont le lieur ne contient que des quantificateurs existentiels est **NP**-complet puisque il s'agit du problème SAT. Certains fragments ont été étudié théoriquement [Büning et Zhao, 2004] ou algorithmiquement. Il existe, par exemple, un algorithme en temps polynomial pour décider d'une QBF sous CNF dont les clauses contiennent au plus deux littéraux [Aspvall *et al.*, 1979]. La procédure `qbfl` [Letombe, 2005] détecte certains fragments [Coste-Marquis *et al.*, 2005; Coste-Marquis *et al.*, 2006a; Bubeck et Büning, 2008], comme les QBF restreintes aux clauses de Horn, afin de les traiter plus efficacement. Toutefois, certaines classes sont peu courantes et pour d'autres leur détection peut être très coûteuse en temps. Une autre approche consiste à travailler sur une QBF quelconque et d'appliquer des algorithmes sur des fragments précis sans chercher à résoudre la QBF. C'est ce que nous avons nommé les *pré-traitements*, parmi lesquels la *Q-résolution* dans `sQueezeBF` ou *l'hyper-résolution binaire* dans `2clsQ`. La suppression des symétries pour SAT est très utilisée pour pré-traiter les problèmes afin de les résoudre plus efficacement. La suppression des symétries a aussi été étudiée pour les QBF et permet d'améliorer les performances des procédures de décision sur de nombreuses instances [Audemard *et al.*, 2007b; Audemard *et al.*, 2007a; Jabbour, 2008].

Procédures	Références
QKN	[Büning *et al.*, 1995]
Evaluate	[Cadoli *et al.*, 1997; Cadoli *et al.*, 1998]
decide	[Rintanen, 1999b]
QSOLVE	[Feldmann *et al.*, 2000]
QuBE	[Giunchiglia *et al.*, 2001c]
Semprop	[Letz, 2002]
Quaffle	[Zhang et Malik, 2002a; Zhang et Malik, 2002b]
QUBOS	[Ayari et Basin, 2002]
QSAT	[Plaisted *et al.*, 2003]
WalkQSAT	[Gent *et al.*, 2003]
quantor	[Biere, 2004]
QMRES, QBDD	[Pan et Vardi, 2004b]
ZQSAT	[GhasemZadeh *et al.*, 2004]
QBDD(LS), QBDD(DLL)	[Audemard et Saïs, 2004; Audemard et Saïs, 2005]
CLearn	[Gent et Rowley, 2005]
sKizzo	[Benedetti, 2004; Benedetti, 2005b]
S-QBF	[Samulowitz et Bacchus, 2005]
Q-PREZ	[Chandrasekar et Hsiao, 2005]
pQBF	[Stéphan, 2006a; Stéphan, 2006b; Stéphan, 2007]
2clsQ	[Samulowitz et Bacchus, 2006]
qpro	[Egly *et al.*, 2009]
IQTest	[Zhang, 2006]
Duaffle	[Sabharwal *et al.*, 2006]
AB	[Benedetti, 2006]
AQME	[Pulina et Tacchella, 2007; Pulina, 2010]
Qbf2nlp	[Da Mota *et al.*, 2007; Stéphan *et al.*, 2009]
QuBIS	[Pulina et Tacchella, 2008a]
Nenofex	[Lonsing et Biere, 2008]
AIGsolve	[Pigorsch et Scholl, 2009]
CirQit	[Goultiaeva *et al.*, 2009a; Goultiaeva *et al.*, 2009b]
StruQS	[Pulina et Tacchella, 2009]
QMiraXT(1T)	[Lewis *et al.*, 2009b]
DepQBF	[Lonsing et Biere, 2010b; Lonsing et Biere, 2010a]

TABLE 2 – Liste des procédures QBF de l'état de l'art et articles les décrivant.

Procédures	Années	Pré.	CNF	Bi-impl.	Algo. 1	Algo. 2	Algo. 3	Philo. Inc.	Philo. Ora.	Philo. Tra.	Philo. Por.
QKN	1995	✓	✓			✓					
Evaluate	1997	✓	✓		✓						
decide	1999	✓	✓		✓						
QSOLVE	2000	✓	✓		✓						
QuBE	2001	✓	✓		✓						
Semprop	2002	✓	✓		✓						
Quaffle	2002	✓	✓		✓						
QUBOS	2002						✓			✓	
QSAT	2003			✓			✓				
WalkQSAT	2003	✓	✓		✓			✓			
quantor	2004	✓	✓			✓					
QMRES	2004	✓	✓			✓					
QBDD	2004	✓	✓			✓					
ZQSAT	2004	✓	✓		✓						
QBDD(LS)	2005	✓	✓					✓	✓		
QBDD(DLL)	2005	✓	✓						✓		
CLearn	2005	✓	✓		✓						
sKizzo	2005	✓	✓							✓	
S-QBF	2005	✓	✓		✓						
Q-PREZ	2005	✓	✓		✓						
pQBF	2006	✓		✓	✓						
2clsQ	2006	✓	✓		✓						
qpro	2006				✓*						
IQTest	2006	✓			✓						
Duaffle	2006	✓			✓						
AB	2006	✓	✓		✓						
AQME	2007	✓	✓								✓
Qbf2nlp	2007	✓		✓						✓	
QuBIS	2008	✓	✓			✓					
Nenofex	2008	✓					✓			✓	
AIGsolve	2009	✓	✓			✓					
CirQit	2009	✓			✓						
StruQS	2009	✓	✓		✓	✓					
QMiraXT(1T)	2009	✓	✓		✓						
DepQBF	2010	✓	✓		✓						
Total sur 35		32	26	3	21	7	2	2	2	4	1

TABLE 3 – Résumé des procédures QBF classées par année : la deuxième colonne donne des détails sur le fragment du langage accepté en entrée (prénexe, CNF et acceptant les bi-implications) ; la troisième colonne précise le ou les algorithmes principaux (* la procédure QSAT est très proche de l'algorithme 1) ; la dernière colonne donne la philosophie de certaines procédures (incomplète, oracles, transformation et portfolio).

Chapitre 3

Équivalences et forme prénexe pour les formules booléennes quantifiées

[Da Mota *et al.*, 2008] Benoit Da Mota, Igor Stéphan, and Pascal Nicolas. Une mise sous forme prénexe préservant les résultats intermédiaires pour les formules booléennes quantifiées. In *Journées Nationales de l'IA Fondamentale*, 2008.

[Da Mota, 2009] Benoit Da Mota. Équivalences et forme prénexe pour les formules booléennes quantifiées. *Rencontre des Jeunes Chercheurs en IA*, 2009.

[Da Mota *et al.*, 2009a] Benoit Da Mota, Igor Stéphan, and Pascal Nicolas. A New Prenexing Strategy for Quantified Boolean Formulae with Bi-Implications. In *Sixth International Workshop on Constraints in Formal Verification*, 2009.

[Da Mota *et al.*, 2009b] Benoit Da Mota, Igor Stéphan, and Pascal Nicolas. Une nouvelle stratégie de mise sous forme prénexe pour des formules booléennes quantifiées avec bi-implications. *Revue I3 (Information - Interaction - Intelligence)*, 9(2), 2009.

[Da Mota, à paraître] Benoit Da Mota. Équivalences et forme prénexe pour les formules booléennes quantifiées. *Revue d'Intelligence Artificielle*, à paraître.

Sommaire

3.1 Introduction

Par un panorama des seize années de recherche sur les procédures de décision du problème QBF, nous avons mis en avant au chapitre 2 que la plupart des procédures actuelles pour décider des QBF nécessitent d'avoir en entrée une formule sous forme normale conjonctive (74%). Celle-ci a été largement étudiée [Plaisted et Greenbaum, 1986; de la Tour, 1992; Egly, 1996], car cette forme normale est utilisée (presque) unanimement pour la satisfiabilité des propositions (formules booléennes non quantifiées). Or, les problèmes sont rarement exprimés directement sous cette forme. Il est plus naturel de les représenter en utilisant la richesse du langage et donc à l'aide d'opérateurs plus expressifs (l'implication, la bi-implication et le ou exclusif) et avec des quantificateurs à l'intérieur des formules. De plus, la forme normale conjonctive semble mal adaptée pour décider de la validité d'une QBF [Zhang et Malik, 2002b; Egly *et al.*, 2003; Ansótegui *et al.*, 2005; Zhang, 2006; Sabharwal *et al.*, 2006]. La mise sous forme normale conjonctive nécessite que la formule soit sous forme prénexe, forme elle-même requise en entrée de 91% des procédures. Cependant, il n'existe pas une unique forme prénexe associée à une QBF. Selon la stratégie choisie pour l'ordre d'extraction des quantificateurs, le résultat varie et par conséquent le temps de résolution peut être fortement influencé [Egly *et al.*, 2003; Giunchiglia *et al.*, 2006c]. En outre, aucune règle n'est fournie pour extraire les quantificateurs de formules booléennes quantifiées contenant des bi-implications et des ou exclusifs. Le seul choix possible est d'exprimer ces opérateurs en fonction des opérateurs pour lesquels nous connaissons des règles. Nous montrons que ce choix a des conséquences sur la taille de la formule et sur le nombre de variables. La procédure `qpro` [Egly *et al.*, 2009] est une alternative intéressante mais elle se limite à la forme normale de négation et par conséquent n'accepte pas les bi-implications, même problème pour `QUBOS` [Ayari et Basin, 2002]. La seule procédure qui puisse répondre à nos attentes est `QSAT` [Plaisted *et al.*, 2003], mais il n'existe aucune implantation connue pour QBF.

Les différentes étapes précédant la décision d'un problème à l'aide du problème de validité des QBF, sont illustrées dans la figure 5. Nous nous intéressons à celle qui consiste à ré-écrire une formule sous une forme logiquement équivalente sans bi-implication ni ou exclusif. Nous proposons dans un premier temps des équivalences permettant de traiter un motif particulier : les variables quantifiées existentiellement qui représentent un *résultat intermédiaire*. Dans un second temps nous exploitons les propriétés de ce motif afin de proposer une solution au cas général de la suppression des bi-implications et ou exclusifs. Nous montrons enfin, comment il est possible de réaliser une mise sous forme normale disjonctive (DNF) polynomiale en espace, ainsi qu'une forme qui mixe CNF et DNF. Pour la partie expérimentale, nous introduisons un problème théorique ad hoc afin de valider notre approche. Puis, nous utilisons des QBF codant la vérification formelle de circuits logiques et plus particulièrement l'additionneur n-bits pour illustrer en pratique l'intérêt de notre mise sous forme prénexe par renommage de sous-formules et dans quelles proportions elle améliore les résultats de quelques procédures de l'état de l'art.

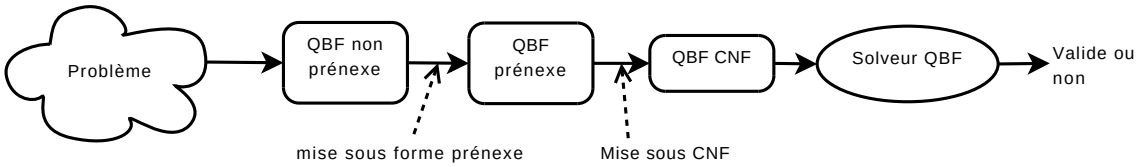


FIGURE 5 – Étapes menant à la décision d'un problème via les QBF.

3.2 Mise sous forme normale conjonctive, forme prénexe et bi-implications

3.2.1 Mise sous forme normale conjonctive et disjonctive

Pour le problème de satisfiabilité en logique propositionnelle, la forme normale conjonctive est particulièrement bien adaptée : la falsification de la proposition est restreinte à la clause. Chaque clause représente en négatif une affectation qui falsifie la proposition. La forme duale est la *forme normale disjonctive*, constituée d'une disjonction de termes, qui est particulièrement bien adaptée à la satisfaction d'une proposition : la satisfiabilité est restreinte au terme. Chaque terme représente une solution pour la proposition. La forme normale conjonctive est facile à obtenir en temps et en espace polynomial en utilisant une méthode par introduction de symboles propositionnels. Nous nommons cette méthode *mise sous forme normale conjonctive par renommage de sous-formule* [Tseitin, 1970]. Dans le cas général d'une proposition quelconque, la forme normale disjonctive ne peut pas s'obtenir en temps polynomial, sauf si $\mathbf{P} = \mathbf{NP}$. La mise sous forme normale conjonctive par renommage de sous-formule est utilisée aussi sur la matrice des formules booléennes quantifiées prénexes et est illustrée ci-dessous. Nous la comparons avec la méthode *académique* utilisant la distributivité du $\wedge$ et du $\vee$.

Soit $\{a, b, c, d, e, f, v_1, v_2, v_3, v_4\}$ un ensemble de variables booléennes et soit $F = \exists a \forall b \exists c \exists d \exists e \exists f(((((a \leftrightarrow b) \leftrightarrow c) \leftrightarrow d) \leftrightarrow e) \leftrightarrow f)$ une QBF, alors nous appliquons la mise sous forme normale conjonctive par renommage de sous-formule :

$$
\begin{aligned}
F &\equiv \exists a \forall b \exists c \exists d \exists e \exists f \exists v_1((v_1 \leftrightarrow (a \leftrightarrow b)) \wedge ((((v_1 \leftrightarrow c) \leftrightarrow d) \leftrightarrow e) \leftrightarrow f)) \\
&\equiv \exists a \forall b \exists c \exists d \exists e \exists f \exists v_1 \exists v_2((v_1 \leftrightarrow (a \leftrightarrow b)) \wedge (v_2 \leftrightarrow (v_1 \leftrightarrow c)) \wedge \\
&\quad (((v_2 \leftrightarrow d) \leftrightarrow e) \leftrightarrow f)) \\
&\equiv \exists a \forall b \exists c \exists d \exists e \exists f \exists v_1 \exists v_2 \exists v_3((v_1 \leftrightarrow (a \leftrightarrow b)) \wedge (v_2 \leftrightarrow (v_1 \leftrightarrow c)) \wedge \\
&\quad (v_3 \leftrightarrow (v_2 \leftrightarrow d)) \wedge ((v_3 \leftrightarrow e) \leftrightarrow f)) \\
&\equiv \exists a \forall b \exists c \exists d \exists e \exists f \exists v_1 \exists v_2 \exists v_3 \exists v_4((v_1 \leftrightarrow (a \leftrightarrow b)) \wedge (v_2 \leftrightarrow (v_1 \leftrightarrow c)) \wedge \\
&\quad (v_3 \leftrightarrow (v_2 \leftrightarrow d)) \wedge (v_4 \leftrightarrow (v_3 \leftrightarrow e)) \wedge (v_4 \leftrightarrow f))
\end{aligned}
$$

$$\begin{aligned}\equiv\ & \exists a \forall b \exists c \exists d \exists e \exists f \exists v_1 \exists v_2 \exists v_3 \exists v_4 (\\ & (\neg v_1 \vee a \vee \neg b) \wedge (\neg v_1 \vee \neg a \vee b) \wedge (v_1 \vee a \vee b) \wedge (v_1 \vee \neg a \vee \neg b) \wedge \\ & (\neg v_2 \vee v_1 \vee \neg c) \wedge (\neg v_2 \vee \neg v_1 \vee c) \wedge (v_2 \vee v_1 \vee c) \wedge (v_2 \vee \neg v_1 \vee \neg c) \wedge \\ & (\neg v_3 \vee v_2 \vee \neg d) \wedge (\neg v_3 \vee \neg v_2 \vee d) \wedge (v_3 \vee v_2 \vee d) \wedge (v_3 \vee \neg v_2 \vee \neg d) \wedge \\ & (\neg v_4 \vee v_3 \vee \neg e) \wedge (\neg v_4 \vee \neg v_3 \vee e) \wedge (v_4 \vee v_3 \vee e) \wedge (v_4 \vee \neg v_3 \vee \neg e) \wedge \\ & (v_4 \vee \neg f) \wedge (\neg v_4 \vee f))\end{aligned}$$

La même formule possédant k bi-implications est mise sous forme normale conjonctive à l'aide de $k-1$ nouvelles variables booléennes quantifiées existentiellement. Cette forme possède $2+4(k-1)$ clauses de longueur 3 maximum.

La méthode qui consiste à utiliser la distributivité du $\vee$ et du $\wedge$ peut potentiellement faire croître exponentiellement la taille de la formule et est plus coûteuse en temps. La formule F possède 6 variables booléennes, soit 2^6 valuations possibles dont 2^5 valuations (non factorisables) qui falsifient la matrice de F. Sa mise sous forme normale conjonctive à l'aide de la distributivité possède 2^5 clauses de longueur 6. Pour une formule semblable possédant n variables booléennes, sa mise sous forme normale conjonctive possède 2^{n-1} clauses de longueur n. Revenons sur la notion de coût en temps. Obtenir l'ensemble des clauses qui falsifient cette formule revient à la résoudre, ce qui est exponentiel en temps en fonction de n le nombre de variables (en admettant que $\mathbf{P} \neq \mathbf{NP}$). Cela s'explique par l'augmentation exponentielle du nombre d'applications de la distributivité en fonction de n pour ce type de formule qui représente un des pires cas de figure.

3.2.2 Mise sous forme prénexe et motivations

Dans [Egly *et al.*, 2003], les auteurs définissent des stratégies pour la mise sous forme prénexe selon l'ordre d'extraction des quantificateurs et montrent expérimentalement que cela peut avoir une forte influence sur le temps de résolution nécessaire aux différentes procédures. Cependant, il n'existe pas de règle pour extraire des quantificateurs de la bi-implication (ou du ou-exclusif). La mise sous forme prénexe se décompose en trois étapes que nous détaillons ici :

1. le remplacement de la bi-implication et du ou-exclusif par leurs définitions selon l'implication, la négation, la conjonction et/ou la disjonction est généralement réalisé par les deux équivalences suivantes :

   1) $(A \leftrightarrow B) \equiv ((A \rightarrow B) \wedge (B \rightarrow A))$ 2) $(A \oplus B) \equiv ((A \vee B) \wedge \neg(A \wedge B))$

   Chacun peut constater que les formules A et B sont dupliquées ;
2. le renommage des variables telles que des quantificateurs distincts lient des variables booléennes distinctes peut être réalisé avant l'étape d'extraction des quantificateurs mais peut aussi être interfolié avec celle-ci ;
3. l'extraction des quantificateurs est généralement basée sur les équivalences suivantes, héritées de la logique classique du premier ordre, elles restent vraies pour

les QBF (F, G et H sont des QBF et x est variable telle que $x \notin \mathcal{VL}(H)$).

3) $(\exists x\ \neg F) \equiv \neg(\forall x\ F)$	4) $(\forall x\ \neg F) \equiv \neg(\exists x\ F)$
5) $(\forall x\ F) \equiv F$, si $x \notin \mathcal{VL}(F)$	6) $(\exists x\ F) \equiv F$, si $x \notin \mathcal{VL}(F)$
7) $(\forall x\ (F \wedge H)) \equiv ((\forall x\ F) \wedge H)$	8) $(\forall x\ (F \vee H)) \equiv ((\forall x\ F) \vee H)$
9) $(\exists x\ (F \wedge H)) \equiv ((\exists x\ F) \wedge H)$	10) $(\exists x\ (F \vee H)) \equiv ((\exists x\ F) \vee H)$
11) $(\forall x\ (F \wedge G)) \equiv ((\forall x\ F) \wedge (\forall x\ G))$	12) $(\exists x\ (F \vee G)) \equiv ((\exists x\ F) \vee (\exists x\ G))$
13) $(\forall x\ (F \rightarrow H)) \equiv ((\exists x\ F) \rightarrow H)$	14) $(\exists x\ (F \rightarrow H)) \equiv ((\forall x\ F) \rightarrow H)$
15) $(\forall x\ (H \rightarrow G)) \equiv (H \rightarrow (\forall x\ G))$	16) $(\exists x\ (H \rightarrow G)) \equiv (H \rightarrow (\exists x\ G))$

Les équivalences 13 à 16 pour l'implication sont aisément déductibles des équivalences 3 à 12. L'extraction des quantificateurs est un processus qui consiste à appliquer ces équivalences de la droite vers la gauche jusqu'à obtenir un point fixe correspondant à une QBF sous forme prénexe. La mise sous forme prénexe conserve la validité mais ne garantit ni de garder la taille de la formule ni l'espace de recherche associé. Alors que l'équivalence représente le « si et seulement si » du langage naturel, le ou-exclusif représente sa négation. Le pouvoir d'expression de la bi-implication s'étend bien au delà de la simple équivalence logique entre $(F \leftrightarrow G)$ et $((G \rightarrow F) \wedge (F \rightarrow G))$: un quantificateur présent dans F ou G, de par les équivalences logiques 13 à 16, sera extrait aussi bien sous sa forme universelle que sous sa forme existentielle, et ce quelle que soit sa forme initiale. Si cela n'a aucun impact vis-à-vis de la validité, il n'en est pas de même vis-à-vis du calcul. Nous montrons dans les exemples suivants qu'à la fin des trois étapes de la mise sous forme prénexe, l'augmentation de la taille de la formule n'est pas le seul problème, même pour une formule très simple.

Soit a une variable booléenne, ϕ, ϕ_1 et ϕ_2 des formules booléennes quantifiées telles que $\phi = (\phi_1 \leftrightarrow \exists a(\phi_2))$ et $a \notin \mathcal{VL}(\phi_1)$. Dans un premier temps il faut exprimer la bi-implication à l'aide de la conjonction et de l'implication :

$$\phi \overset{1}{\equiv} ((\phi_1 \rightarrow \exists a(\phi_2)) \wedge (\exists a(\phi_2) \rightarrow \phi_1))$$

Les formules ϕ_1 et ϕ_2 ont été dupliquées. Ensuite, il faut extraire les quantificateurs des implications :

$$\phi \overset{16,13}{\equiv} (\exists\ a(\phi_1 \rightarrow \phi_2) \wedge \forall\ a(\phi_2 \rightarrow \phi_1))$$

Il faut renommer une des variables a afin d'extraire les quantificateurs de la conjonction. Soit b une nouvelle variable booléenne et $\phi_2' = [a \leftarrow b](\phi_2)$ alors :

$$\phi \overset{9,7}{\equiv} \exists a\ \forall b((\phi_1 \rightarrow \phi_2) \wedge (\phi_2' \rightarrow \phi_1)) \tag{3.1}$$

L'ordre d'extraction des quantificateurs aurait pu être inversé, nous aurions obtenu :

$$\phi \overset{7,9}{\equiv} \forall b\ \exists a((\phi_1 \rightarrow \phi_2) \wedge (\phi_2' \rightarrow \phi_1)) \tag{3.2}$$

Les formules (3.1) et (3.2) sont équivalentes, or dans dans le cas général, si F est une formule booléenne quantifiée, $(\forall y\ (\exists x\ F))$ n'est pas équivalente à $(\exists x\ (\forall y\ F))$. Le fait de savoir que les quantificateurs peuvent s'inverser contrairement au cas général est une

information importante qui pourrait être exploitée par les procédures de décision pour le problème de validité des QBF. Les symboles propositionnels a et b représentent le même symbole propositionnel de la formule non prénexe mais rien dans la formule prénexe ne semble les mettre en relation. Sans traiter directement les bi-implications il est impossible d'éviter cet écueil.

Pour conclure cette partie sur les motivations, nous envisageons le cas où un quantificateur doit traverser plusieurs bi-implications. Nous montrons dans l'exemple suivant, que le choix de la position des parenthèses peut influer sur la taille de la formule mise sous forme prénexe. Soit a une variable booléenne, ϕ_d, ϕ_g, ϕ_1, ϕ_2 et ϕ_3 des QBF telles que $\phi_g = ((\phi_1 \leftrightarrow \phi_2) \leftrightarrow (\exists a\ \phi_3))$, $\phi_d = (\phi_1 \leftrightarrow (\phi_2 \leftrightarrow (\exists a\ \phi_3)))$ avec $a \notin \mathcal{VL}(\phi_1)$ et $a \notin \mathcal{VL}(\phi_2)$ alors $\phi_d \equiv \phi_g$ par associativité de la bi-implication. Nous mettons ϕ_g sous forme prénexe :

$$
\begin{array}{lcl}
\phi_g & \overset{1}{\equiv} & (((\phi_1 \leftrightarrow \phi_2) \rightarrow (\exists a\ \phi_3)) \wedge ((\exists a\ \phi_3) \rightarrow (\phi_1 \leftrightarrow \phi_2))) \\
\phi_g & \equiv & (((\phi_1 \leftrightarrow \phi_2) \rightarrow (\exists a\ \phi_3)) \wedge ((\exists b\ [a \leftarrow b](\phi_3)) \rightarrow (\phi_1 \leftrightarrow \phi_2))) \\
\phi_g & \overset{16,13}{\equiv} & (\exists a\ ((\phi_1 \leftrightarrow \phi_2) \rightarrow \phi_3) \wedge \forall b\ ([a \leftarrow b](\phi_3) \rightarrow (\phi_1 \leftrightarrow \phi_2))) \\
\phi_g & \overset{9,7}{\equiv} & \exists a \forall b\ (((\phi_1 \leftrightarrow \phi_2) \rightarrow \phi_3)\ \wedge ([a \leftarrow b](\phi_3) \rightarrow (\phi_1 \leftrightarrow \phi_2)))
\end{array}
$$

puis ϕ_d :

$$
\begin{array}{lcl}
\phi_d & \overset{1}{\equiv} & (\phi_1 \leftrightarrow ((\phi_2 \rightarrow (\exists a\ \phi_3)) \wedge ((\exists a\ \phi_3) \rightarrow \phi_2))) \\
\phi_d & \overset{1}{\equiv} & ((\phi_1 \rightarrow ((\phi_2 \rightarrow (\exists a\ \phi_3)) \wedge ((\exists a\ \phi_3) \rightarrow \phi_2))) \wedge \\
 & & (((\phi_2 \rightarrow (\exists a\ \phi_3)) \wedge ((\exists a\ \phi_3) \rightarrow \phi_2)) \rightarrow \phi_1)) \\
\phi_d & \equiv & ((\phi_1 \rightarrow ((\phi_2 \rightarrow (\exists a\ \phi_3)) \wedge ((\exists b\ [a \leftarrow b](\phi_3)) \rightarrow \phi_2))) \wedge \\
 & & (((\phi_2 \rightarrow (\exists c\ [a \leftarrow c](\phi_3))) \wedge ((\exists d\ [a \leftarrow d](\phi_3)) \rightarrow \phi_2)) \rightarrow \phi_1)) \\
\phi_d & \overset{13,16}{\equiv} & ((\phi_1 \rightarrow (\exists a\ (\phi_2 \rightarrow \phi_3) \wedge \forall b\ ([a \leftarrow b](\phi_3)\ \rightarrow \phi_2))) \wedge \\
 & & ((\exists c\ (\phi_2 \rightarrow [a \leftarrow c](\phi_3)) \wedge \forall d\ ([a \leftarrow d](\phi_3)\ \rightarrow \phi_2)) \rightarrow \phi_1)) \\
\phi_d & \overset{9,7}{\equiv} & ((\phi_1 \rightarrow \exists a \forall b\ ((\phi_2 \rightarrow \phi_3) \wedge ([a \leftarrow b](\phi_3)\ \rightarrow \phi_2))) \wedge \\
 & & ((\exists c\ (\phi_2 \rightarrow [a \leftarrow c](\phi_3)) \wedge \forall d\ ([a \leftarrow d](\phi_3)\ \rightarrow \phi_2)) \rightarrow \phi_1)) \\
\phi_d & \overset{7,9}{\equiv} & ((\phi_1 \rightarrow \exists a \forall b\ ((\phi_2 \rightarrow \phi_3) \wedge ([a \leftarrow b](\phi_3)\ \rightarrow \phi_2))) \wedge \\
 & & (\forall d \exists c\ ((\phi_2 \rightarrow [a \leftarrow c](\phi_3)) \wedge ([a \leftarrow d](\phi_3)\ \rightarrow \phi_2)) \rightarrow \phi_1)) \\
\phi_d & \overset{16,9}{\equiv} & \exists a((\phi_1 \rightarrow \forall b\ ((\phi_2 \rightarrow \phi_3) \wedge ([a \leftarrow b](\phi_3)\ \rightarrow \phi_2))) \wedge \\
 & & (\forall d \exists c\ ((\phi_2 \rightarrow [a \leftarrow c](\phi_3)) \wedge ([a \leftarrow d](\phi_3)\ \rightarrow \phi_2)) \rightarrow \phi_1)) \\
\phi_d & \overset{14,9}{\equiv} & \exists a \exists d((\phi_1 \rightarrow \forall b\ ((\phi_2 \rightarrow \phi_3) \wedge (\phi_3[a \leftarrow b] \rightarrow \phi_2))) \wedge \\
 & & (\exists c\ ((\phi_2 \rightarrow \phi_3[a \leftarrow c]) \wedge (\phi_3[a \leftarrow d] \rightarrow \phi_2)) \rightarrow \phi_1)) \\
\phi_d & \overset{15,7}{\equiv} & \exists a \exists d \forall b((\phi_1 \rightarrow ((\phi_2 \rightarrow \phi_3) \wedge ([a \leftarrow b](\phi_3)\ \rightarrow \phi_2))) \wedge \\
 & & (\exists c\ ((\phi_2 \rightarrow [a \leftarrow c](\phi_3)) \wedge ([a \leftarrow d](\phi_3)\ \rightarrow \phi_2)) \rightarrow \phi_1)) \\
\phi_d & \overset{15,7}{\equiv} & \exists a \exists d \forall b \forall c\ ((\phi_1 \rightarrow ((\phi_2 \rightarrow \phi_3) \wedge ([a \leftarrow b](\phi_3)\ \rightarrow \phi_2))) \wedge \\
 & & (((\phi_2 \rightarrow [a \leftarrow c](\phi_3)) \wedge ([a \leftarrow d](\phi_3)\ \rightarrow \phi_2)) \rightarrow \phi_1))
\end{array}
$$

Les formules ϕ_g et ϕ_d, bien qu'équivalentes, n'ont pas du tout la même taille ni le même nombre de variables une fois mises sous forme prénexe. Maintenant, si la formule considérée est $(\phi_1 \leftrightarrow (\phi_2 \leftrightarrow \ldots (\phi_n \leftrightarrow (\exists a\ \phi_{n+1}))))$, alors 2^n variables booléennes sont introduites dont la moitié est quantifiée universellement. Les formules ϕ_n et ϕ_{n+1} sont

recopiées 2^n fois, la formule ϕ_{n-1} est recopiée 2^{n-1} fois, et ainsi de suite, jusqu'à ϕ_1 qui est recopiée 2 fois. La taille de la formule et le nombre de variables croissent exponentiellement avec le nombre de bi-implications traversées. Quant à la formule $((\exists a\ \phi_{n+1}) \leftrightarrow (\phi_n \leftrightarrow ...(\phi_2 \leftrightarrow \phi_1)))$, après la mise sous forme prénexe, elle possède 2 variables additionnelles dont une quantifiée universellement et chaque ϕ_k est présente seulement 2 fois. Nous pourrions choisir de placer les parenthèses de cette manière, mais si chacune des ϕ_k possède un quantificateur à extraire, nous n'évitons pas le pire cas. Nous remarquerons que la mise sous forme prénexe classique ne permet pas de résoudre ce cas. Pourtant, il doit exister une mise sous forme prénexe en espace polynomial, sans quoi ce fragment du langage des QBF n'appartiendrait plus à la *hiérarchie polynomiale*.

3.3 Extraction de résultats intermédiaires et renommage de sous-formules

La section précédente a montré l'importance du problème de la mise sous forme prénexe lorsque des quantificateurs apparaissent dans les bi-implications. Puisque $(A \oplus B) \equiv \neg(A \leftrightarrow B) \equiv (\neg A \leftrightarrow B)$, dans ce qui suit, nous ne traiterons que des bi-implications. Autant que nous sachions, ce problème est généralement occulté dans le processus de traduction du problème spécifié en une QBF sous CNF équivalente donnée en entrée à une procédure de décision QBF. Tout d'abord, nous mettons en évidence un motif particulier aux propriétés intéressantes lors de la mise sous forme prénexe. Puis nous montrons que ce motif permet pour le langage QBF, le renommage de sous-formules.

3.3.1 Forme prénexe et extraction de résultats intermédiaires

Dans cette section, nous portons notre attention sur deux cas très fréquents qui apparaissent en programmation ou en spécification : la déclaration de *résultats intermédiaires* et la déclaration de *domaine*. Le premier cas est basé sur l'introduction d'un symbole propositionnel existentiellement quantifié en association avec une conjonction dans le but d'améliorer la lisibilité de la spécification ou de capturer les résultats intermédiaires d'un calcul présent à plusieurs endroits. Pour les QBF, par extension des résultats propositionnels classiques [Tseitin, 1970], nous nous intéressons au motif $(\exists x\ ((x \leftrightarrow F) \wedge G))$, x variable intermédiaire absente de F, qui est équivalent à $[x \leftarrow F](G)$. Le second cas est basé sur l'introduction d'une variable booléenne quantifiée universellement en association avec une implication pour exprimer le domaine de variable (ie : une propriété à vérifier par la variable). Dans les deux cas, nous appelons $(x \leftrightarrow F)$ la définition de x. Le résultat suivant démontre qu'en fait les deux techniques sont équivalentes dans le cas des QBF.

Théorème 3.3.1. *Soient F et G deux QBF et x une variable booléenne qui représente un résultat intermédiaire F, avec $x \notin \mathcal{VL}(F)$, alors*

$$(\exists x\ ((x \leftrightarrow F) \wedge G)) \equiv (\forall x\ ((x \leftrightarrow F) \rightarrow G)).$$

Démonstration.

$$
\begin{array}{ll}
& (\exists x\ ((F \leftrightarrow x) \wedge G)) \\
\equiv & ((F \leftrightarrow \top) \wedge [x \leftarrow \top](G)) \vee ((F \leftrightarrow \bot) \wedge [x \leftarrow \bot](G)) \\
\equiv & (((F \leftrightarrow \top) \vee [x \leftarrow \bot](G)) \wedge ((F \leftrightarrow \bot) \vee [x \leftarrow \top](G)) \wedge ([x \leftarrow \bot](G) \vee [x \leftarrow \top](G))) \\
\overset{*}{\equiv} & ((F \leftrightarrow \top) \vee [x \leftarrow \bot](G)) \wedge ((F \leftrightarrow \bot) \vee [x \leftarrow \top](G)) \\
\equiv & (\neg(F \leftrightarrow \top) \rightarrow [x \leftarrow \bot](G)) \wedge (\neg(F \leftrightarrow \bot) \rightarrow [x \leftarrow \top](G)) \\
\equiv & ((F \leftrightarrow \bot) \rightarrow [x \leftarrow \bot](G)) \wedge ((F \leftrightarrow \top) \rightarrow [x \leftarrow \top](G)) \\
\equiv & ((\forall x\ (F \leftrightarrow x)) \rightarrow G)
\end{array}
$$

car $((\neg A \vee B) \wedge (A \vee C) \wedge (B \vee C)) \overset{*}{\equiv} ((\neg A \vee B) \wedge (A \vee C))$. (résolution) □

Par le théorème précédent, nous nous focalisons uniquement sur le motif existentiel. Puisque les solveurs prennent en entrée généralement des QBF sous CNF, le statut spécial des variables intermédiaires est totalement perdu et elles sont traitées comme les variables du problème initial. Une manière simple de se séparer de ces symboles intermédiaires est d'appliquer l'équivalence déjà introduite $(\exists x\ ((x \leftrightarrow F) \wedge G)) \equiv [x \leftarrow F](G)$ mais cela peut conduire à une croissance exponentielle de la taille de la formule. À la place, nous proposons d'extraire ces symboles propositionnels grâce au théorème suivant.

Théorème 3.3.2. *Soient F, G et H trois QBF et x une variable booléenne qui représente le résultat intermédiaire F, avec $x \notin \mathcal{VL}(H)$, $x \notin \mathcal{VL}(F)$ alors*

$$(H \leftrightarrow (\exists x\ ((x \leftrightarrow F) \wedge G))) \equiv (\exists x\ ((x \leftrightarrow F) \wedge (H \leftrightarrow G)))$$

Démonstration.

$$
\begin{array}{ll}
& H \leftrightarrow (\exists x((F \leftrightarrow x) \wedge G)) \\
\equiv & ((H \rightarrow (\exists x\ ((F \leftrightarrow x) \wedge G))) \wedge ((\exists x\ ((F \leftrightarrow x) \wedge G)) \rightarrow H)) \\
\equiv & \exists x \forall x'((H \rightarrow ((F \leftrightarrow x) \wedge G)) \wedge (((F \leftrightarrow x') \wedge [x \leftarrow x'](G)) \rightarrow H)) \\
\equiv & \exists x \forall x'((\neg H \vee ((F \leftrightarrow x) \wedge G)) \wedge (\neg((F \leftrightarrow x') \wedge [x \leftarrow x'](G)) \vee H)) \\
\equiv & \exists x \forall x'((\neg H \wedge \neg((F \leftrightarrow x') \wedge [x \leftarrow x'](G))) \vee (((F \leftrightarrow x) \wedge G) \wedge H)) \\
\equiv & ((\forall x'\ (\neg H \wedge (\neg(F \leftrightarrow x') \vee \neg[x \leftarrow x'](G)))) \vee \\
& \ (\exists x\ (((F \leftrightarrow x) \wedge G) \wedge H))) \\
\equiv & (((\neg H \wedge (\neg(F \leftrightarrow \bot) \vee \neg[x \leftarrow x'][x' \leftarrow \bot](G))) \wedge \\
& \ (\neg H \wedge (\neg(F \leftrightarrow \top) \vee \neg[x \leftarrow x'][x' \leftarrow \top](G)))) \vee \\
& \ ((((F \leftrightarrow \bot) \wedge [x \leftarrow \bot](G)) \wedge H) \vee (((F \leftrightarrow \top) \wedge [x \leftarrow \top](G)) \wedge H))) \\
\equiv & ((((\neg H \wedge \neg(F \leftrightarrow \bot)) \vee (\neg H \wedge \neg[x \leftarrow \bot](G))) \wedge \\
& \ ((\neg H \wedge \neg(F \leftrightarrow \top)) \vee (\neg H \wedge \neg[x \leftarrow \top](G)))) \vee \\
& \ ((((F \leftrightarrow \bot) \wedge [x \leftarrow \bot](G)) \wedge H) \vee (((F \leftrightarrow \top) \wedge [x \leftarrow \top](G)) \wedge H))) \\
\equiv & (((\neg H \wedge \neg(F \leftrightarrow \bot)) \wedge (\neg H \wedge \neg(F \leftrightarrow \top))) \vee \\
& \ ((\neg H \wedge \neg[x \leftarrow \bot](G)) \wedge (\neg H \wedge \neg(F \leftrightarrow \top))) \vee \\
& \ ((\neg H \wedge \neg(F \leftrightarrow \bot)) \wedge (\neg H \wedge \neg[x \leftarrow \top](G))) \vee \\
& \ ((\neg H \wedge \neg[x \leftarrow \bot](G)) \wedge (\neg H \wedge \neg[x \leftarrow \top](G))) \vee \\
& \ ((((F \leftrightarrow \bot) \wedge [x \leftarrow \bot](G)) \wedge H) \vee (((F \leftrightarrow \top) \wedge [x \leftarrow \top](G)) \wedge H)))
\end{array}
$$

$$
\begin{array}{rl}
\equiv & (((\neg H \wedge \neg[x \leftarrow \bot](G)) \wedge (F \leftrightarrow \bot)) \vee \\
& ((\neg H \wedge \neg[x \leftarrow \top](G)) \wedge (F \leftrightarrow \top)) \vee \\
& ((H \wedge [x \leftarrow \bot](G)) \wedge (F \leftrightarrow \bot)) \vee \\
& ((H \wedge [x \leftarrow \top](G)) \wedge (F \leftrightarrow \top))) \\
\equiv & (\exists x\ (((\neg H \wedge \neg G) \wedge (F \leftrightarrow x)) \vee ((H \wedge G) \wedge (F \leftrightarrow x)))) \\
\equiv & (\exists x\ ((F \leftrightarrow x) \wedge ((\neg H \wedge \neg G) \vee (H \wedge G)))) \\
\equiv & (\exists x\ ((F \leftrightarrow x) \wedge (H \leftrightarrow G)))
\end{array}
$$

□

Les quatre algorithmes suivants appliquent ensemble et récursivement ce théorème à toutes les bi-implications satisfaisant le motif.

Algorithme 4 : $recherche$

Données : Une QBF F
Données : Un ensemble de variables booléennes E_{vb}
Résultat : Un ensemble de définitions
début
  suivant F **faire**
    cas où $(\exists x\ G)$
      retourner $recherche(G, E_{vb} \cup \{x\})$
    fin
    cas où $(G \wedge H)$
      retourner $recherche(G, E_{vb}) \cup recherche(H, E_{vb})$
    fin
    cas où $(x \leftrightarrow G)$
      si $x \in E_{vb}$ **alors**
        retourner $\{(x \leftrightarrow G)\}$
      sinon
        retourner $\emptyset$
      fin
    fin
    autres cas
      retourner $\emptyset$
    fin
  fin
fin

Soit $(\phi_H \leftrightarrow \phi)$ une QBF, l'algorithme $recherche$ recherche toute définition satisfaisant le motif $(\exists x\ ((x \leftrightarrow F) \wedge G))$ dans ϕ tout en relâchant la contrainte $x \notin \mathcal{VL}(F)$. En fait, le motif peut être encastré dans une succession de motifs similaires grâce à l'équivalence 9, l'équivalence $(\exists x\ (\exists y\ F)) \equiv (\exists y\ (\exists x\ F))$, ainsi que l'associativité et la commutativité de la conjonction. Par exemple

Algorithme 5 : $extrait$

Données : Un ensemble de définitions E_d
Données : Un ensemble de variables booléennes E_{vb}
Résultat : Une liste de définitions
début
  $L_d = nil$
  tant que *il existe* $(e \leftrightarrow d) \in E_d$ *tel que* $\mathcal{VL}(d) \subseteq E_{vb}$ **faire**
    $E_d = E_d$ privé de l'ensemble des définitions sur e
    $E_{vb} = E_{vb} \cup \{e\}$
    $L_d = \cdot((e \leftrightarrow d), L_d)$
  fin
  retourner L_d
fin

$$recherche((\phi_1 \wedge (\exists a\ (\exists b\ ((a \leftrightarrow \phi_a) \wedge ((b \leftrightarrow \phi_b) \wedge \phi_2))))), \emptyset) = \{(a \leftrightarrow \phi_a), (b \leftrightarrow \phi_b)\}.$$

Dans ce qui suit, nous dénotons $\{(a \leftrightarrow \phi_a), (b \leftrightarrow \phi_b)\}$ par E_d. L'algorithme $extrait$ applique la contrainte $x \notin \mathcal{VL}(F)$ à l'ensemble des définitions obtenu par l'algorithme $recherche$ grâce à un tri topologique qui extrait une liste de définitions. Toutes les définitions de l'ensemble ne sont pas insérées dans la liste : par exemple si $a \in \mathcal{VL}(\phi_b)$, $b \in \mathcal{VL}(\phi_a)$ et E_{vb} est un ensemble de variables booléennes tel que $a \notin E_{vb}$ et $b \notin E_{vb}$ alors $extrait(E_d, E_{vb}) = nil$ sinon si $a \notin \mathcal{VL}(\phi_b)$ mais $b \in \mathcal{VL}(\phi_a)$ alors

$$extrait(E_d, E_{vb}) = \cdot((b \leftrightarrow \phi_b), \cdot((a \leftrightarrow \phi_a), nil))$$

puisque

$$\begin{array}{l}(\phi_1 \wedge (\exists a\ (\exists b\ ((a \leftrightarrow \phi_a) \wedge ((b \leftrightarrow \phi_b) \wedge \phi_2))))) \\ \equiv \quad (\exists b\ ((b \leftrightarrow \phi_b) \wedge (\exists a\ ((a \leftrightarrow \phi_a) \wedge (\phi_1 \wedge \phi_2)))))\end{array}$$

L'algorithme $applique$ applique effectivement le théorème 3.3.2 en deux étapes sur la QBF ϕ pour la liste de définitions extraites par l'algorithme $extrait$. Premièrement, les définitions sont remplacées par $\top$ dans la QBF puisque les définitions sont connectées conjonctivement et sont réintroduites à l'extérieur de la QBF : par exemple

$$(\phi_1 \wedge (\exists a\ (\exists b\ ((a \leftrightarrow \phi_a) \wedge ((b \leftrightarrow \phi_b) \wedge \phi_2)))))$$

est remplacée par

$$\begin{array}{l}((b \leftrightarrow \phi_b) \wedge ((a \leftrightarrow \phi_a) \wedge (\phi_1 \wedge (\exists a\ (\exists b\ (\top \wedge (\top \wedge \phi_2))))))) \\ \equiv \quad ((b \leftrightarrow \phi_b) \wedge ((a \leftrightarrow \phi_a) \wedge (\phi_1 \wedge (\exists a\ (\exists b\ \phi_2))))).\end{array}$$

Deuxièmement, les quantificateurs existentiels pour les symboles propositionnels de la liste de définitions sont éliminés de la formule et réintroduits à l'extérieur de celle-ci : par

Algorithme 6 : *applique*

Données : Une QBF F
Données : Une liste de définitions L_d
Résultat : Une QBF
début
  $L_{temp} = L_d$; `/* Une liste de définitions */`
  tant que *non* $vide(L_d)$ **faire**
    $(e \leftrightarrow d) = tête(L_d)$
    $[v_{temp} \leftarrow (e \leftrightarrow d)](F') = F$
    $F = ((e \leftrightarrow d) \wedge [v_{temp} \leftarrow \top](F'))$
    $L_d = queue(L_d)$
  fin
  tant que *non* $vide(L_{temp})$ **faire**
    $(e \leftrightarrow d) = tête(L_{temp})$
    `/* élimination de` F `du quantificateur existentiel`
       `portant sur` e `et ayant comme portée la QBF` G `*/`
    $[v_{temp} \leftarrow \exists e\, G](F') = F$
    $F = [v_{temp} \leftarrow G](F')$
    `/* réintroduction devant` F `du quantificateur */`
    $F = (\exists e\, F)$
    $L_{temp} = queue(L_{temp})$
  fin
  retourner F
fin

exemple, $((b \leftrightarrow \phi_b) \wedge ((a \leftrightarrow \phi_a) \wedge (\phi_1 \wedge (\exists a\ (\exists b\ \phi_2)))))$ est remplacé par

$$\begin{aligned} & (\exists b\ (\exists a\ ((b \leftrightarrow \phi_b) \wedge ((a \leftrightarrow \phi_a) \wedge (\phi_1 \wedge \phi_2))))) \\ \equiv\ & (\exists b\ ((b \leftrightarrow \phi_b) \wedge (\exists a\ ((a \leftrightarrow \phi_a) \wedge (\phi_1 \wedge \phi_2))))). \end{aligned}$$

L'algorithme $rec\_def\_extract$ applique récursivement l'extraction des motifs sur toutes les définitions possibles dans un processus allant de l'intérieur vers l'extérieur de la formule. Ainsi, des définitions peuvent franchir plusieurs bi-implications. Il n'est pas difficile de prouver grâce au théorème 3.3.2 le théorème de correction suivant.

Théorème 3.3.3. *Soit F une QBF alors $rec\_def\_extract(F, \emptyset) \equiv F$.*

Démonstration. Soit F une QBF alors, par induction sur la structure, $recherche(F, \emptyset)$ est un ensemble de définitions de F qui sont telles qu'elles ne sont séparées de la racine de la formule F (considérée comme un arbre) que par des nœuds étiquetés par des existentielles ou des conjonctions.

Soient F et G deux QBF alors, par point fixe sur l'ordre d'inclusion des ensembles, $extrait(recherche(F, \emptyset) \cup recherche(G, \emptyset), S)$ est une liste $[(e_1 \leftrightarrow d_1), \ldots, (e_n \leftrightarrow d_n)]$ telle que (par la propriété ci-dessus) chaque $(e_i \leftrightarrow d_i)$ est une définition de $(F \leftrightarrow G)$ séparée

Algorithme 7 : $rec\_def\_extract$

Données : Une QBF F
Données : Un ensemble de variables booléennes E_{vb}
Résultat : Une QBF équivalente à F avec ses définitions extraites
début
  suivant F **faire**
    cas où $(qx\ G)$
      retourner $(qx\ rec\_def\_extract(G, \{x\} \cup E_{vb}))$
    fin
    cas où $\neg G$
      retourner $\neg rec\_def\_extract(G, E_{vb})$
    fin
    cas où $(G \circ H)$ *et* $\circ \in \{\wedge, \vee, \rightarrow\}$
      retourner $(rec\_def\_extract(G, E_{vb}) \circ rec\_def\_extract(H, E_{vb}))$
    fin
    cas où $(G \leftrightarrow H)$
      $G' = rec\_def\_extract(G, E_{vb})$; `/* Une QBF */`
      $H' = rec\_def\_extract(H, E_{vb})$; `/* Une QBF */`
      $L_d = recherche(G', \emptyset) \cup recherche(H', \emptyset)$; `/* Une liste de définitions */`
      retourner $applique((G' \leftrightarrow H'), extrait(L_d, E_{vb}))$
    fin
    autres cas
      retourner F
    fin
  fin
fin

de la racine de la formule uniquement par des nœuds étiquetés par des existentielles ou des conjonctions et pour tout e_i, $1 \leq i \leq n$, $\mathcal{VL}(d_i) \subseteq S \cup \{e_1, \ldots, e_n\}$.

Soient F et G deux QBF alors, par récurrence sur le nombre d'éléments de la liste $extrait(recherche(F, \emptyset) \cup recherche(G, \emptyset), S)$, le théorème 3.3.2, l'équivalence 9 et l'équivalence $(\exists x\ (\exists y\ H)) \equiv (\exists y\ (\exists x\ H))$, H une QBF quelconque (avec $E_{vb} = \mathcal{VL}((F \leftrightarrow G))$) : $applique((F \leftrightarrow G), extrait(recherche(F, \emptyset) \cup recherche(G, \emptyset), E_{vb})) \equiv (F \leftrightarrow G)$.

Enfin, par induction sur la structure et par la propriété exprimée ci-dessus, pour une QBF F quelconque, $rec\_def\_extract(F, \emptyset) \equiv F$. □

3.3.2 Mise sous forme prénexe de formules quelconques par renommage de sous-formules

Nous avons maintenant à notre disposition un ensemble d'équivalences permettant d'extraire des résultats intermédiaires à travers une bi-implication ou un ou exclusif. Or

les quantificateurs à extraire lors de la mise sous forme prénexe d'une formule quelconque G ne portent pas uniquement sur des variables intermédiaires. Dans ce cas, il est possible d'introduire un résultat intermédiaire x afin d'extraire F, sous formule de G, avec F qui contient le quantificateur à extraire. Cette intuition est formalisée dans le théorème suivant, démontrée, puis illustrée ensuite.

Théorème 3.3.4. *Soit F et G des QBF, telles que F est une sous formule de G et $\mathcal{VL}(F) \subseteq \mathcal{VL}(G)$. Soit G' une QBF telle que $G = [x \leftarrow F](G')$ [1], alors :*

$$G \equiv \exists x((x \leftrightarrow F) \wedge G')$$

Démonstration.
Cas de base :

- si $G = F$ et $G' = x$, alors
  $G \equiv \exists x((x \leftrightarrow F) \wedge x) \equiv (((\top \leftrightarrow F) \wedge \top) \vee ((\bot \leftrightarrow F) \wedge \bot)) \equiv (\top \leftrightarrow F) \equiv F$;
- si $G = y$ et $G' = y$, avec y une variable, alors
  $G \equiv \exists x((x \leftrightarrow F) \wedge y) \equiv (\exists x(x \leftrightarrow F) \wedge y) \equiv (\top \wedge y) \equiv y$;
- même démonstration pour $G = \top$ et $G = \bot$.

Hypothèse d'induction :

Soit F et J des QBF, telles que F est une sous formule de J et $\mathcal{VL}(F) \subseteq \mathcal{VL}(J)$. Soit J' une QBF telle que $J = [x \leftarrow F](J')$, alors

$$J \equiv \exists x((x \leftrightarrow F) \wedge J').$$

L'induction s'effectue sur la QBF G obtenue en composant J avec : $qy, \neg, \wedge, \vee, \rightarrow, \leftrightarrow, \oplus$ et si besoin une QBF H. y est variable, par hypothèse $y \notin \mathcal{VL}(F)$, $q \in \{\exists, \forall\}$. Pour les opérateurs binaires, $G = H\ o\ J$ et par hypothèse $x \notin \mathcal{VL}(H)$. Donc $G' = H\ o\ J'$ avec $o \in \{\wedge, \vee, \rightarrow, \leftrightarrow, \oplus\}$. Par hypothèse nous pouvons appliquer le théorème 3.3.1 : soient F et J' deux QBF et x une variable qui représente un résultat intermédiaire F, avec $x \notin \mathcal{VL}(F)$, alors

$$(\exists x\ ((x \leftrightarrow F) \wedge J')) \equiv (\forall x\ ((x \leftrightarrow F) \rightarrow J')).$$

Cas d'induction :

- si $G = qy\ J$ et $G' = qy\ J'$, alors :
  $G \equiv \exists x((x \leftrightarrow F) \wedge (qy\ J')) \equiv qy\ \exists x((x \leftrightarrow F) \wedge J') \equiv qy\ J$;
- si $G = \neg J$ et $G' = \neg J'$, alors :
  $G \equiv \exists x((x \leftrightarrow F) \wedge (\neg J')) \equiv \forall x((x \leftrightarrow F) \rightarrow (\neg J')) \equiv \forall x((\neg(x \leftrightarrow F)) \vee (\neg J'))$
  $G \equiv \forall x(\neg((x \leftrightarrow F) \wedge J')) \equiv \neg(\exists x((x \leftrightarrow F) \wedge J')) \equiv \neg J$;
- si $G = (H \wedge J)$ et $G' = (H \wedge J')$, alors :
  $G \equiv \exists x((x \leftrightarrow F) \wedge (H \wedge J')) \equiv (H \wedge (\exists x((x \leftrightarrow F) \wedge J'))) \equiv (H \wedge J)$;
- si $G = (H \vee J)$ et $G' = (H \vee J')$, alors :
  $G \equiv \exists x((x \leftrightarrow F) \wedge (H \vee J')) \equiv \forall x((x \leftrightarrow F) \rightarrow (H \vee J'))$
  $G \equiv \forall x(H \vee ((x \leftrightarrow F) \rightarrow J')) \equiv H \vee (\forall x((x \leftrightarrow F) \rightarrow J')$
  $G \equiv H \vee (\exists x((x \leftrightarrow F) \wedge J')) \equiv (H \vee J)$;

1. *G' est la formule G dans laquelle chaque occurrence de F est remplacée par x.*

- même démonstration pour $G = (H \to J)$ et $G' = (H \to J')$;
- si $G = (H \leftrightarrow J)$ et $G' = (H \leftrightarrow J')$, alors :
  $G \equiv \exists x((x \leftrightarrow F) \wedge (H \leftrightarrow J')) \equiv H \leftrightarrow (\exists x((x \leftrightarrow F) \wedge J')) \equiv (H \leftrightarrow J)$ par utilisation du théorème 3.3.2 ;
- même démonstration pour $G = (H \oplus J)$ et $G' = (H \oplus J')$.

□

Corollaire 3.3.5. *Avec les hypothèses du théorème 3.3.4, les 2 équivalences suivantes sont vraies :*

$$G \equiv \forall x((x \leftrightarrow F) \to G')$$
$$\textit{et}$$
$$G \equiv \forall x((\neg x \leftrightarrow F) \vee G')$$

La démonstration est immédiate en utilisant le théorème 3.3.1 et les équivalences $(A \to B) \equiv (\neg A \vee B)$ *et* $\neg(A \leftrightarrow A) \equiv (\neg A \leftrightarrow A)$.

Reprenons un des exemples de la section 3.2.2. Soient a et x des variables booléennes, ϕ_d, ϕ_g, ϕ_1, ϕ_2 et ϕ_3 des QBF telles que $\phi_g = ((\phi_1 \leftrightarrow \phi_2) \leftrightarrow (\exists a\ \phi_3))$, $\phi_d = (\phi_1 \leftrightarrow (\phi_2 \leftrightarrow (\exists a\ \phi_3)))$ avec $a \notin \mathcal{VL}(\phi_1)$ et $a \notin \mathcal{VL}(\phi_2)$. Pour appliquer le théorème 3.3.4 à la formule ϕ_g, nous prenons $G = \phi_g$ et $F = (\exists a\ \phi_3)$. De même pour la formule ϕ_d, sauf que $G = \phi_d$.

$$\begin{array}{lll}
\phi_g & \overset{th_{3.3.4}}{\equiv} & \exists x\ [(x \leftrightarrow (\exists a\ \phi_3)) \wedge ((\phi_1 \leftrightarrow \phi_2) \leftrightarrow x)] \\
& \overset{1}{\equiv} & \exists x\ [((x \to (\exists a\ \phi_3)) \wedge ((\exists b\ [a \leftarrow b](\phi_3)) \to x)) \wedge ((\phi_1 \leftrightarrow \phi_2) \leftrightarrow x)] \\
& \overset{13,16}{\equiv} & \exists x\ [(\exists a\ (x \to \phi_3) \wedge \forall b\ ([a \leftarrow b](\phi_3) \to x)) \wedge ((\phi_1 \leftrightarrow \phi_2) \leftrightarrow x)] \\
& \overset{9,7}{\equiv} & \exists x \exists a \forall b\ [((x \to \phi_3) \wedge ([a \leftarrow b](\phi_3) \to x)) \wedge ((\phi_1 \leftrightarrow \phi_2) \leftrightarrow x)] \\
\phi_d & \overset{th_{3.3.4}}{\equiv} & \exists x\ [(x \leftrightarrow (\exists a\ \phi_3)) \wedge (\phi_1 \leftrightarrow (\phi_2 \leftrightarrow x))] \\
& \overset{1}{\equiv} & \exists x\ [((x \to (\exists a\ \phi_3)) \wedge ((\exists b\ [a \leftarrow b](\phi_3)) \to x)) \wedge (\phi_1 \leftrightarrow (\phi_2 \leftrightarrow x))] \\
& \overset{13,16}{\equiv} & \exists x\ [(\exists a\ (x \to \phi_3) \wedge \forall b\ ([a \leftarrow b](\phi_3) \to x)) \wedge (\phi_1 \leftrightarrow (\phi_2 \leftrightarrow x))] \\
& \overset{9,7}{\equiv} & \exists x \exists a \forall b\ [((x \to \phi_3) \wedge ([a \leftarrow b](\phi_3) \to x)) \wedge (\phi_1 \leftrightarrow (\phi_2 \leftrightarrow x))]
\end{array}$$

Nous remarquons que les formules ϕ_g et ϕ_d une fois mises sous forme prénexe sont très proches et il est trivial d'affirmer qu'elles sont équivalentes par associativité de la bi-implication. Puisque nous considérons que la variable a n'est pas un résultat intermédiaire, il faut supprimer la bi-implication dans la formule $(x \leftrightarrow (\exists a\ \phi_3))$ afin de terminer la mise sous forme prénexe. La formule ϕ_3 est donc dupliquée, et une variable b quantifiée universellement est introduite. Comme coût supplémentaire, nous avons aussi introduit un résultat intermédiaire x. En contrepartie, le nombre de variables introduites par quantificateur extrait et le nombre de recopies ne dépendent plus du nombre de bi-implications et de ou exclusifs traversés. Pour chaque quantificateur à extraire, nous introduisons 2 nouvelles variables et seule la formule sous la portée de ce quantificateur est recopiée une fois. La définition du résultat intermédiaire introduit est en conjonction avec le reste de la formule. Elle peut être vue comme une condition nécessaire pour satisfaire le cœur de la formule. Il s'agit aussi d'une propriété intéressante pour mettre

une QBF sous forme conjonctive sans passer par la forme prénexe. Le théorème 3.3.4 permet d'étendre la technique de renommage de formules aux QBF. Le corollaire 3.3.5 introduit pour les QBF une technique de renommage de formules par introduction de variables universelles, dont la déclaration du *domaine* est en disjonction avec le reste de la formule. Cette propriété est intéressante car elle permet de mettre au point une technique de mise sous forme disjonctive en temps et en espace polynomial par renommage de sous-formules. Reprenons l'exemple de la section 3.2.1 que nous avions mis sous CNF. Soit $\{a, b, c, d, e, f, v_1, v_2, v_3, v_4\}$ un ensemble de variables booléennes et soit $F = \exists a \forall b \exists c \exists d \exists e \exists f(((((a \leftrightarrow b) \leftrightarrow c) \leftrightarrow d) \leftrightarrow e) \leftrightarrow f)$ une QBF, alors nous appliquons la mise sous forme normale disjonctive par renommage de sous-formule :

$$
\begin{aligned}
F &\equiv \exists a \forall b \exists c \exists d \exists e \exists f \forall v_1((\neg v_1 \leftrightarrow (a \leftrightarrow b)) \vee ((((v_1 \leftrightarrow c) \leftrightarrow d) \leftrightarrow e) \leftrightarrow f)) \\
&\equiv \exists a \forall b \exists c \exists d \exists e \exists f \forall v_1 \forall v_2((\neg v_1 \leftrightarrow (a \leftrightarrow b)) \vee (\neg v_2 \leftrightarrow (v_1 \leftrightarrow c)) \vee \\
&\quad (((v_2 \leftrightarrow d) \leftrightarrow e) \leftrightarrow f)) \\
&\equiv \exists a \forall b \exists c \exists d \exists e \exists f \forall v_1 \forall v_2 \forall v_3((\neg v_1 \leftrightarrow (a \leftrightarrow b)) \vee (\neg v_2 \leftrightarrow (v_1 \leftrightarrow c)) \vee \\
&\quad (\neg v_3 \leftrightarrow (v_2 \leftrightarrow d)) \vee ((v_3 \leftrightarrow e) \leftrightarrow f)) \\
&\equiv \exists a \forall b \exists c \exists d \exists e \exists f \forall v_1 \forall v_2 \forall v_3 \forall v_4((\neg v_1 \leftrightarrow (a \leftrightarrow b)) \vee (\neg v_2 \leftrightarrow (v_1 \leftrightarrow c)) \vee \\
&\quad (\neg v_3 \leftrightarrow (v_2 \leftrightarrow d)) \vee (\neg v_4 \leftrightarrow (v_3 \leftrightarrow e)) \vee (v_4 \leftrightarrow f)) \\
&\equiv \exists a \forall b \exists c \exists d \exists e \exists f \forall v_1 \forall v_2 \forall v_3 \forall v_4(\\
&\quad (v_1 \wedge a \wedge \neg b) \vee (v_1 \wedge \neg a \wedge b) \vee (\neg v_1 \wedge a \wedge b) \vee (\neg v_1 \wedge \neg a \wedge \neg b) \vee \\
&\quad (v_2 \wedge v_1 \wedge \neg c) \vee (v_2 \wedge \neg v_1 \wedge c) \vee (\neg v_2 \wedge v_1 \wedge c) \vee (\neg v_2 \wedge \neg v_1 \wedge \neg c) \vee \\
&\quad (v_3 \wedge v_2 \wedge \neg d) \vee (v_3 \wedge \neg v_2 \wedge d) \vee (\neg v_3 \wedge v_2 \wedge d) \vee (\neg v_3 \wedge \neg v_2 \wedge \neg d) \vee \\
&\quad (v_4 \wedge v_3 \wedge \neg e) \vee (v_4 \wedge \neg v_3 \wedge e) \vee (\neg v_4 \wedge v_3 \wedge e) \vee (\neg v_4 \wedge \neg v_3 \wedge \neg e) \vee \\
&\quad (v_4 \wedge f) \vee (\neg v_4 \wedge \neg f))
\end{aligned}
$$

Nous avons évoqué dans la section 1.4.1, le *talon d'Achile des QBF* [Ansótegui *et al.*, 2005]. Une des solutions proposées est de coder directement le problème avec une forme alliant CNF et DNF, qui prend mieux en compte l'aspect symétrique de la sémantique des quantificateurs[Sabharwal *et al.*, 2006]. Les procédures `IQTest` et `Duaffle` partent de ce constat et proposent de traiter cette forme. Or, il est possible d'interfolier l'utilisation du théorème 3.3.4 et du corollaire 3.3.5 afin d'obtenir une forme CNF/DNF par réintroduction des quantificateurs vers l'intérieur :

$$
\begin{aligned}
F &\equiv \exists a \forall b \exists c \exists d \exists e \exists f \exists v_1((v_1 \leftrightarrow (a \leftrightarrow b)) \wedge ((((v_1 \leftrightarrow c) \leftrightarrow d) \leftrightarrow e) \leftrightarrow f)) \\
&\equiv \exists a \forall b \exists v_1((v_1 \leftrightarrow (a \leftrightarrow b)) \wedge \exists c \exists d \exists e \exists f((((v_1 \leftrightarrow c) \leftrightarrow d) \leftrightarrow e) \leftrightarrow f)) \\
&\equiv \exists a \forall b \forall v_1((\neg v_1 \leftrightarrow (a \leftrightarrow b)) \vee \exists c \exists d \exists e \exists f(((((v_1 \leftrightarrow c) \leftrightarrow d) \leftrightarrow e) \leftrightarrow f))) \\
&\equiv \exists a \forall b \forall v_1((\neg v_1 \leftrightarrow (a \leftrightarrow b)) \vee \\
&\quad \exists c \exists d \exists e \exists f \exists v_1 \exists v_2 \exists v_3 \exists v_4(\\
&\quad\quad (v_2 \leftrightarrow (v_1 \leftrightarrow c)) \wedge (v_3 \leftrightarrow (v_2 \leftrightarrow d)) \wedge (v_4 \leftrightarrow (v_3 \leftrightarrow e)) \wedge (v_4 \leftrightarrow f))) \\
&\equiv \exists a \forall b \forall v_1((v_1 \wedge a \wedge \neg b) \vee (v_1 \wedge \neg a \wedge b) \vee (\neg v_1 \wedge a \wedge b) \vee (\neg v_1 \wedge \neg a \wedge \neg b) \vee \\
&\quad \exists c \exists d \exists e \exists f \exists v_1 \exists v_2 \exists v_3 \exists v_4(\\
&\quad\quad (\neg v_2 \vee v_1 \vee \neg c) \wedge (\neg v_2 \vee \neg v_1 \vee c) \wedge (v_2 \vee v_1 \vee c) \wedge (v_2 \vee \neg v_1 \vee \neg c) \wedge \\
&\quad\quad (\neg v_3 \vee v_2 \vee \neg d) \wedge (\neg v_3 \vee \neg v_2 \vee d) \wedge (v_3 \vee v_2 \vee d) \wedge (v_3 \vee \neg v_2 \vee \neg d) \wedge \\
&\quad\quad (\neg v_4 \vee v_3 \vee \neg e) \wedge (\neg v_4 \vee \neg v_3 \vee e) \wedge (v_4 \vee v_3 \vee e) \wedge (v_4 \vee \neg v_3 \vee \neg e) \wedge \\
&\quad\quad (v_4 \vee \neg f) \wedge (\neg v_4 \vee f)))
\end{aligned}
$$

De la même manière, nous avions présenté (chapitre 1 p. 22) le langage **QBF**$^+$, ajoutant aux QBF le concept de *quantification restreinte*[Benedetti *et al.*, 2007; Benedetti et Mangassarian, 2008]. Il est possible d'interfolier l'utilisation du théorème 3.3.4 et du corollaire 3.3.5 jusqu'à l'obtention d'un jeu fini à deux joueurs :

$$\begin{array}{rcl} F & \equiv & \exists a \forall b \exists c \exists d \exists e \exists f \exists v_1((v_1 \leftrightarrow (a \leftrightarrow b)) \wedge ((((v_1 \leftrightarrow c) \leftrightarrow d) \leftrightarrow e) \leftrightarrow f)) \\ & \equiv & \exists a \forall b \exists v_1((v_1 \leftrightarrow (a \leftrightarrow b)) \wedge (\exists c \exists d \exists e \exists f((((v_1 \leftrightarrow c) \leftrightarrow d) \leftrightarrow e) \leftrightarrow f))) \\ & \equiv & \exists a \forall b \forall v_1((v_1 \leftrightarrow (a \leftrightarrow b)) \rightarrow (\exists c \exists d \exists e \exists f((((v_1 \leftrightarrow c) \leftrightarrow d) \leftrightarrow e) \leftrightarrow f)))) \\ & \equiv & \exists a(\top \wedge (\forall b, v_1((v_1 \leftrightarrow (a \leftrightarrow b)) \rightarrow (\exists c, d, e, f((((v_1 \leftrightarrow c) \leftrightarrow d) \leftrightarrow e) \leftrightarrow f) \wedge \top))))) \end{array}$$

Puis nous passons dans le langage **QBF**$^+$ où F peut s'écrire

$$F^+ = \exists a[R_1^{\exists}(a)]\ \forall b, v_1[R_2^{\forall}(a, b, v_1)]\ \exists c, d, e, f[R_3^{\exists}(a, b, v_1, c, d, e, f)]\ V(a, b, v_1, c, d, e, f)$$

avec

$$\begin{array}{rcl} R_1^{\exists}(a) & = & \top \\ R_2^{\forall}(a, b, v_1) & = & ((v_1 \leftrightarrow (a \leftrightarrow b)) \\ R_3^{\exists}(a, b, v_1, c, d, e, f) & = & ((((v_1 \leftrightarrow c) \leftrightarrow d) \leftrightarrow e) \leftrightarrow f) \\ V(a, b, v_1, c, d, e, f) & = & \top \end{array}$$

3.3.3 Exploitation de la fusion de quantificateurs

Dans [Egly *et al.*, 2003], les auteurs suggèrent d'approfondir leur travail en exploitant la fusion de quantificateurs. La mise sous forme prénexe à l'aide du théorème 3.3.4 permet de faire fusionner les quantificateurs universels dans la chaîne de $\wedge$ ainsi créée par l'application de l'équivalence $(\forall x\ (F \wedge G)) \equiv ((\forall x\ F) \wedge (\forall x\ G))$ qui permet de réduire le nombre de quantificateurs universels. L'exemple suivant illustre cette possibilité.

Soient a, b, c, a', b', c', x, y et z des variables booléennes, ϕ, ϕ_1, ϕ_2 et ϕ_3 des QBF telles que $\phi = (((\forall a\ \phi_1) \leftrightarrow (\exists b\ \phi_2)) \leftrightarrow (\exists c\ \phi_3))$, avec $a \notin \mathcal{VL}(\phi_2)$, $a \notin \mathcal{VL}(\phi_3)$, $b \notin \mathcal{VL}(\phi_1)$, $b \notin \mathcal{VL}(\phi_3)$, $c \notin \mathcal{VL}(\phi_1)$ et $c \notin \mathcal{VL}(\phi_2)$. La présence de $\forall a$ qui porte sur ϕ_1 permet d'illustrer que la fusion de quantificateurs n'est pas limitée par le type des quantificateurs présents dans les formules extraites à l'aide du théorème 3.3.4. Nous appliquons ce théorème en priorité.

$$\begin{array}{rcl} \phi & \overset{th_{3.3.4}}{\equiv} & \exists x \exists y \exists z\ [(x \leftrightarrow (\forall a\ \phi_1)) \wedge (y \leftrightarrow (\exists b\ \phi_2)) \wedge (z \leftrightarrow (\exists c\ \phi_3)) \wedge ((x \leftrightarrow y) \leftrightarrow z)] \\ & \overset{1}{\equiv} & \exists x \exists y \exists z[(((x \rightarrow (\forall a\ \phi_1)) \wedge ((\forall a'\ [a \leftarrow a'](\phi_1)) \rightarrow x))) \\ & & \wedge(((y \rightarrow (\exists b\ \phi_2)) \wedge ((\exists b'\ [b \leftarrow b'](\phi_2)) \rightarrow y))) \\ & & \wedge(((z \rightarrow (\exists c\ \phi_3)) \wedge ((\exists c'\ [c \leftarrow c'](\phi_3)) \rightarrow z))) \wedge ((x \leftrightarrow y) \leftrightarrow z)] \\ & \overset{14,9}{\equiv} & \exists x \exists y \exists z \exists a'[(((x \rightarrow (\forall a\ \phi_1)) \wedge ([a \leftarrow a'](\phi_1) \rightarrow x))) \\ & & \wedge(((y \rightarrow (\exists b\ \phi_2)) \wedge ((\exists b'\ [b \leftarrow b'](\phi_2)) \rightarrow y))) \\ & & \wedge(((z \rightarrow (\exists c\ \phi_3)) \wedge ((\exists c'\ [c \leftarrow c'](\phi_3)) \rightarrow z))) \wedge ((x \leftrightarrow y) \leftrightarrow z)] \\ & \overset{16,9}{\equiv} & \exists x \exists y \exists z \exists a' \exists b \exists c[(((x \rightarrow (\forall a\ \phi_1)) \wedge ([a \leftarrow a'](\phi_1) \rightarrow x))) \\ & & \wedge(((y \rightarrow \phi_2) \wedge ((\exists b'\ [b \leftarrow b'](\phi_2)) \rightarrow y))) \\ & & \wedge(((z \rightarrow \phi_3) \wedge ((\exists c'\ [c \leftarrow c'](\phi_3)) \rightarrow z))) \wedge ((x \leftrightarrow y) \leftrightarrow z)] \end{array}$$

$$
\begin{array}{ll}
\overset{14,15}{\equiv} & \exists x \exists y \exists z \exists a' \exists b \exists c[((\forall a\ (x \to \phi_1) \wedge ([a \leftarrow a'](\phi_1) \to x))) \\
& \wedge(((y \to \phi_2) \wedge \forall b'\ ([b \leftarrow b'](\phi_2) \to y))) \wedge (((z \to \phi_3) \wedge \forall c'\ ([c \leftarrow c'](\phi_3) \to z))) \\
& \wedge((x \leftrightarrow y) \leftrightarrow z)] \\
\overset{9}{\equiv} & \exists x \exists y \exists z \exists a' \exists b \exists c[(\forall a\ ((x \to \phi_1) \wedge ([a \leftarrow a'](\phi_1) \to x))) \\
& \wedge(\forall b'\ ((y \to \phi_2) \wedge ([b \leftarrow b'](\phi_2) \to y))) \wedge (\forall c'\ ((z \to \phi_3) \wedge ([c \leftarrow c'](\phi_3) \to z))) \\
& \wedge((x \leftrightarrow y) \leftrightarrow z)] \\
\overset{11}{\equiv} & \exists x \exists y \exists z \exists a' \exists b \exists c \forall u[((x \to [a \leftarrow u](\phi_1)) \wedge ([a \leftarrow a'](\phi_1) \to x)) \\
& \wedge((y \to \phi_2) \wedge ([b \leftarrow u](\phi_2) \to y)) \wedge ((z \to \phi_3) \wedge ([c \leftarrow u](\phi_3) \to z)) \\
& \wedge((x \leftrightarrow y) \leftrightarrow z)]
\end{array}
$$

La première étape consiste à extraire tous les maillons de la chaîne de bi-implications contenant des quantificateurs à l'aide de l'équivalence du théorème 3.3.4. Puis nous effectuons une mise sous forme prénexe classique sauf que l'équivalence 11 est appliquée, sans quoi nous aurions obtenu :

$$
\begin{array}{lll}
\phi & \overset{7}{\equiv} & \exists x \exists y \exists z \exists a' \exists b \exists c \forall a \forall b' \forall c'[((x \to \phi_1) \wedge ([a \leftarrow a'](\phi_1) \to x)) \\
& & \wedge((y \to \phi_2) \wedge ([b \leftarrow b'](\phi_2) \to y)) \wedge ((z \to \phi_3) \wedge ([c \leftarrow c'](\phi_3) \to z)) \\
& & \wedge((x \leftrightarrow y) \leftrightarrow z)]
\end{array}
$$

De la même manière, la mise sous forme prénexe à l'aide du corollaire 3.3.5, permet de faire fusionner les quantificateurs existentiels dans la chaîne de $\vee$ ainsi créée par l'application de l'équivalence $(\exists x\ (F \vee G)) \equiv ((\exists x\ F) \vee (\exists x\ G))$.

$$
\begin{array}{lll}
\phi & \overset{12}{\equiv} & \forall x \forall y \forall z \forall a \forall b \forall c \exists e[((\neg x \to \phi_1) \vee ([a \leftarrow e](\phi_1) \to \neg x)) \\
& & \vee((\neg y \to [b \leftarrow e](\phi_2)) \vee (\phi_2 \to \neg y)) \vee ((\neg z \to [c \leftarrow e](\phi_3)) \vee (\phi_3 \to \neg z)) \\
& & \vee((x \leftrightarrow y) \leftrightarrow z)]
\end{array}
$$

3.4 Étude expérimentale

Parmi ce que nous avons présenté, il est possible de tester en pratique un certain nombre de propriétés. Nous pouvons vérifier que l'algorithme $rec\_def\_extract$ permet, lorsque des *résultats intermédiaires* sont présents à l'intérieur d'une bi-implication, d'obtenir une forme prénexe CNF plus concise. Nous pouvons alors étudier l'impact de ce nouveau codage sur le temps de calcul pour des procédures de l'état de l'art. Puis de la même façon, nous pouvons étudier l'impact de notre *mise sous forme prénexe par renommage de sous-formule* lorsque des quantificateurs sont présents à l'intérieur d'une bi-implication sur le temps de calcul. Enfin, nous pouvons observer l'effet de la fusion de quantificateurs. Dans un premier temps, nous présentons les deux problèmes qui serviront à notre étude. Un premier problème ad hoc décrit le pire cas pour la mise sous forme prénexe classique. Un deuxième problème, issu de la vérification formelle, permet d'étudier nos propositions sur un exemple pratique. Ensuite, nous revenons sur le choix des procédures de décision retenues pour nos tests avant finalement d'exposer et commenter nos résultats.

3.4.1 Chaînes de bi-implications

Il s'agit maintenant en pratique d'évaluer nos propositions sur le pire cas évoqué à la fin de la section 3.2.2 ($\phi_1 \leftrightarrow (\phi_2 \leftrightarrow ...(\phi_{n-1} \leftrightarrow (\phi_n))))$. Pour cela nous définissons la classe de formules ψ_n, où chaque ϕ_k contient un résultat intermédiaire. Les ϕ_k sont de la forme $(\exists x_k((x_k \leftrightarrow (c \vee b)) \wedge (x_k \wedge a)))$, avec x_k le résultat intermédiaire et a, b et c des variables booléennes présentes dans d'autres maillons de la chaîne d'équivalences. Pour $n \geq 3$, ψ_n est définie inductivement à partir de $\mathcal{L}_{\psi_n}$, un lieur, et $\mathcal{C}_{\psi_n}$, un corps (et non une matrice, car non CNF et contenant des quantificateurs). Alors,

$$\begin{aligned}
\mathcal{L}_{\psi_3} &= \exists e_1 \exists e_0 \forall u_0 \forall u_1 \forall u_2 \\
\mathcal{C}_{\psi_3} &= ((\exists x_3((x_3 \leftrightarrow (u_2 \vee e_0)) \wedge (x_3 \wedge e_1))) \leftrightarrow \\
&\quad ((\exists x_2((x_2 \leftrightarrow (u_1 \vee u_2)) \wedge (x_2 \wedge e_0))) \leftrightarrow \\
&\quad (\exists x_1((x_1 \leftrightarrow (u_0 \vee u_1)) \wedge (x_1 \wedge u_2))))) \\
\psi_3 &= \mathcal{L}_{\psi_3}(\mathcal{C}_{\psi_3})
\end{aligned}$$

est le cas de base et

$$\begin{aligned}
\mathcal{L}_{\psi_n} &= \exists e_{n-2} \mathcal{L}_{\psi_{n-1}} \\
\mathcal{C}_{\psi_n} &= ((\exists x_n((x_n \leftrightarrow (e_{n-4} \vee e_{n-3})) \wedge (x_n \wedge e_{n-2}))) \leftrightarrow \mathcal{C}_{\psi_{n-1}}) \\
\psi_n &= \mathcal{L}_{\psi_n}(\mathcal{C}_{\psi_n})
\end{aligned}$$

est la définition récursive. $\psi_{n,i} = prenex\_cnf(\psi_n)$ (i pour *méthode initiale*) représente la formule ψ_n mise sous CNF avec la méthode classique (notée $prenex\_cnf$). Leur taille étant exponentielle en fonction de n, les $\psi_{n,i}$ ne sont pas décrites en détails ici. La formule $\psi_{n,e}$ représente la QBF ψ_n mise sous CNF après *transformation* à l'aide de l'algorithme $rec\_def\_extract$. $\mathcal{D}_{\psi_n,e}$ représente une liste de définitions extraites (avec $n \geq 3$). Le résultat de l'algorithme d'extraction est :

$$\begin{aligned}
\mathcal{L}_{\psi_3,e} &= \exists e_1 \exists e_0 \forall u_0 \forall u_1 \forall u_2 \exists x_3 \exists x_2 \exists x_1 \\
\mathcal{D}_{\psi_3,e} &= ((x_3 \leftrightarrow (u_2 \vee e_0)) \wedge ((x_2 \leftrightarrow (u_1 \vee u_2)) \wedge (x_1 \leftrightarrow (u_0 \vee u_1)))) \\
\mathcal{C}_{\psi_3,e} &= ((x_3 \wedge e_1) \leftrightarrow ((x_2 \wedge e_0) \leftrightarrow (x_1 \wedge u_2))) \\
\psi_{3,e} &= \mathcal{L}_{\psi_3,e}(prenex_cnf(\mathcal{D}_{\psi_3,e} \wedge \mathcal{C}_{\psi_3,e}))
\end{aligned}$$

$$\begin{aligned}
\mathcal{L}_{\psi_n,e} &= \exists e_{n-2} \mathcal{L}_{\psi_{n-1},e} \exists x_n \\
\mathcal{D}_{\psi_n,e} &= ((x_n \leftrightarrow (e_{n-4} \vee e_{n-3})) \wedge \mathcal{D}_{\psi_{n-1},e}) \\
\mathcal{C}_{\psi_n,e} &= ((x_n \wedge e_{n-2}) \leftrightarrow \mathcal{C}_{\psi_{n-1},e}) \\
\psi_{n,e} &= \mathcal{L}_{\psi_n,e}(prenex_cnf(\mathcal{D}_{\psi_n,e} \wedge \mathcal{C}_{\psi_n,e}))
\end{aligned}$$

La formule $\psi_{n,r}$ représente la QBF ψ_n mise sous forme prénexe par renommage de sous-formules et mise sous CNF, sans utiliser l'algorithme $rec\_def\_extract$. Le but étant d'étudier le cas général, nous considérons les x_k comme n'étant pas des résultats intermédiaires. Aussi nous introduisons des variables intermédiaires v_k à l'aide du théorème 3.3.4, $\mathcal{D}_{\psi_n,r}$ représente une liste de définitions introduites (avec $n \geq 3$), telle que :

$$\begin{aligned}
\mathcal{L}_{\psi_3,r} &= \exists e_1 \exists e_0 \forall u_0 \forall u_1 \forall u_2 \exists v_3 \exists v_2 \exists v_1 \\
\mathcal{D}_{\psi_3,r} &= ((v_3 \leftrightarrow (\exists x_3((x_3 \leftrightarrow (u_2 \vee e_0)) \wedge (x_3 \wedge e_1)))) \wedge \\
&\quad ((v_2 \leftrightarrow (\exists x_2((x_2 \leftrightarrow (u_1 \vee u_2) \wedge (x_2 \wedge e_0))))) \wedge \\
&\quad (v_1 \leftrightarrow (\exists x_1((x_1 \leftrightarrow (u_0 \vee u_1)) \wedge (x_1 \wedge u_2)))))) \\
\mathcal{C}_{\psi_3,r} &= (v_3 \leftrightarrow (v_2 \leftrightarrow v_1))
\end{aligned}$$

n	Lieur pour $\psi_{n,r}(\psi_{n,f})$	$NC_{r/f}$	Lieur pour $\psi_{n,e}$	NC_e
4	∃[3]∀[3]∃[8]∀[4(1)]∃[51]	165	∃[3]∀[3]∃[23]	65
5	∃[4]∀[3]∃[10]∀[5(1)]∃[64]	207	∃[4]∀[3]∃[29]	82
6	∃[5]∀[3]∃[12]∀[6(1)]∃[77]	249	∃[5]∀[3]∃[35]	99
40	∃[39]∀[3]∃[80]∀[40(1)]∃[519]	1677	∃[39]∀[3]∃[239]	677
50	∃[49]∀[3]∃[100]∀[50(1)]∃[649]	2097	∃[49]∀[3]∃[299]	847
60	∃[59]∀[3]∃[120]∀[60(1)]∃[779]	2517	∃[59]∀[3]∃[359]	1017
400	∃[399]∀[3]∃[800]∀[400(1)]∃[5199]	16797	∃[399]∀[3]∃[2399]	6797
500	∃[499]∀[3]∃[1000]∀[500(1)]∃[6499]	20997	∃[499]∀[3]∃[2999]	8497
600	∃[599]∀[3]∃[1200]∀[600(1)]∃[7799]	25197	∃[599]∀[3]∃[3599]	10197
4000	∃[3999]∀[3]∃[8000]∀[4000(1)]∃[51999]	167997	∃[3999]∀[3]∃[23999]	67997
5000	∃[4999]∀[3]∃[10000]∀[5000(1)]∃[64999]	209997	∃[4999]∀[3]∃[29999]	84997
6000	∃[5999]∀[3]∃[12000]∀[6000(1)]∃[77999]	251997	∃[5999]∀[3]∃[35999]	101997

TABLE 4 – Tailles des QBF transformées pour les chaînes de bi-implications (les clauses sont de taille 2 ou 3).

$$
\begin{array}{lcl}
\psi_{3,r} & = & \mathcal{L}_{\psi_{3,r}}(prenex\_cnf(\mathcal{D}_{\psi_{3,r}} \wedge \mathcal{C}_{\psi_{3,r}})) \\
 & & \\
\mathcal{L}_{\psi_n,r} & = & \exists e_{n-2}\mathcal{L}_{\psi_{n-1},r}\exists x_n \\
\mathcal{D}_{\psi_n,r} & = & ((v_n \leftrightarrow (\exists x_n((x_n \leftrightarrow (e_{n-4} \vee e_{n-3})) \wedge (x_n \wedge e_{n-2})))) \wedge \mathcal{D}_{\psi_{n-1},r}) \\
\mathcal{C}_{\psi_n,r} & = & (v_n \leftrightarrow \mathcal{C}_{\psi_{n-1},r}) \\
\psi_{n,r} & = & \mathcal{L}_{\psi_n,r}(prenex\_cnf(\mathcal{D}_{\psi_n,r} \wedge \mathcal{C}_{\psi_n,r}))
\end{array}
$$

La fonction $prenex\_fusion\_cnf$ applique l'équivalence 11 en priorité lors de la mise sous forme prénexe, comme évoqué dans la section 3.3.3, puis effectue la mise sous CNF classique. La version obtenue par fusion de quantificateurs (universels) est nommée $\psi_{n,f}$, telle que :

$$\psi_{n,f} = \mathcal{L}_{\psi_n,r}(prenex\_fusion\_cnf(\mathcal{D}_{\psi_n,r} \wedge \mathcal{C}_{\psi_n,r}))$$

Nous avons développé un algorithme en Prolog, qui permet de générer les instances des chaînes de bi-implications, puis d'effectuer la mise sous CNF. Notons que les instances sont invalides. Le tableau 4 montre les préfixes et le nombre de clauses ($NC_{r/f}$ et NC_e), pour différentes valeurs de n, de la CNF des chaînes de bi-implications ψ_n avec la méthode par renommage de sous-formules ($\psi_{n,r}$), avec la fusion ($\psi_{n,f}$) et avec l'extraction des résultats intermédiaires ($\psi_{n,e}$) avec l'algorithme $rec\_def\_extract$. Pour $\psi_{n,r}$ et $\psi_{n,f}$, seul les lieurs diffèrent : $\forall[4(1)]$ signifie que dans la version avec fusion, les 4 variables universelles ont été renommées en une seule. Nous n'avons pas donné les lieurs des chaînes de bi-implications mises sous CNF de façon classique ($\psi_{n,i}$) car ils sont bien trop grands. Pour avoir un ordre d'idée, avec $n = 4$, il y a 137 variables dont 14 quantifiées universellement, 350 clauses et 12 alternances de quantificateurs. Avec $n = 5$, il y a 282 variables dont 26 universelles, 734 clauses et 22 alternances de quantificateurs. Nous avons évalué la taille de la CNF à un peu moins de 10^{12} variables pour $n = 40$, plus de 10^{120} pour $n = 400$ et plus de 10^{1200} pour $n = 4000$. La première remarque est donc que la méthode de mise sous forme prénexe par renommage de sous-formules per-

met de réduire exponentiellement la taille de la formule mise sous CNF par rapport à la méthode classique. Elle permet ainsi de mettre sous forme prénexe de façon efficace en taille l'ensemble des formules de **QBF** à l'instar de la méthode par extraction de résultats intermédiaires (l'algorithme $rec\_def\_extract$) qui ne traite qu'un cas particulier, mais de façon plus compacte. La deuxième remarque concerne le nombre d'alternances de quantificateurs. Ce nombre est fixe à 4 pour $\psi_{n,r}$ et $\psi_{n,f}$, à 2 pour $\psi_{n,e}$. La combinaison de l'algorithme $rec\_def\_extract$ puis de la mise sous forme prénexe par renommage de sous-formules donne exactement $\psi_{n,e}$, obtenant ainsi la forme la plus compacte. Le nombre d'alternances de quantificateurs est d'environ 2^n pour $\psi_{n,i}$, faisant paraître le problème dans un niveau de complexité très au-dessus de la réalité dans la hiérarchie polynomiale.

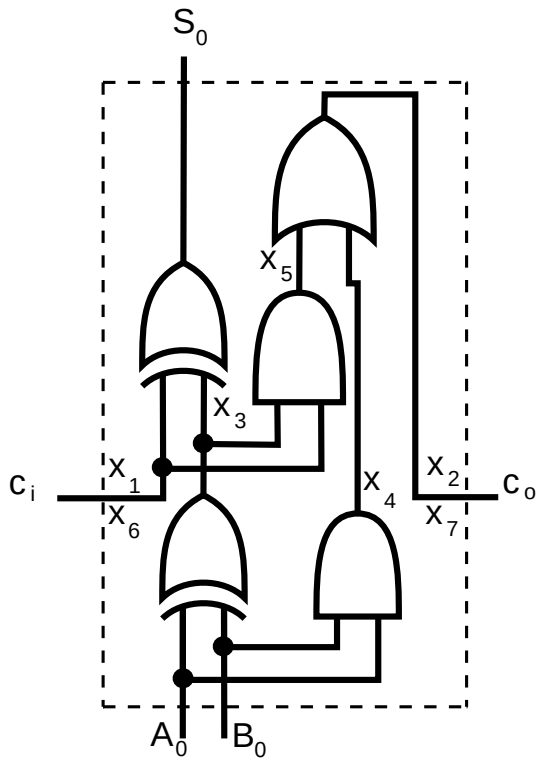
3.4.2 Vérification formelle de circuits : l'additionneur n-bits

Lors de la conception de circuits logiques, le but est de construire un circuit qui corresponde au comportement souhaité. C'est-à-dire que pour chaque jeu d'entrées possible, la sortie attendue est obtenue. Le plus souvent le modèle booléen du circuit diffère grandement du circuit conçu par l'ingénieur pour des raisons pratiques (encombrement, prix, intégration, etc...). La problème de la conception physique correcte de ce circuit se pose donc de manière cruciale. Cette vérification formelle de circuit s'exprime à l'aide de formules de la logique monadique. La construction de modèles bornés (Bounded Model Construction) est une méthode pour résoudre des problèmes de validité de formules de la logique monadique en les transformant en QBF et en cherchant la validité des formules obtenues. Parmi les circuits, la vérification de l'additionneur n-bits est devenu un problème classique dans sa version QBF et sa description complète peut être trouvée dans [Ayari *et al.*, 1999; Ayari et Basin, 2002]. La vérification de l'additionneur consiste à vérifier l'équivalence en logique monadique entre la structure du circuit et le comportement souhaité (n est la taille de l'additionneur, A et B sont les deux n-bits à additionner, S le résultat de l'addition, c_i la retenue en entrée et c_o la retenue en sortie) :
$\phi = \forall n \forall A \forall B \forall S \forall c_i \forall c_o (add_{struc}(n, A, B, S, c_i, c_o) \leftrightarrow add_{comp}(n, A, B, S, c_i, c_o))$

Les formules add_{struc} et add_{comp} contiennent des variables existentielles, qui représentent des résultats intermédiaires, entre autres les retenues (cf. figure 6). Nous donnons un exemple pour l'additionneur n-bits avec $n = 1$. La QBF ϕ_1 code la vérification de l'additionneur pour $n = 1$ avec les fonctions booléennes suivantes :

$$
\begin{array}{ll}
C_1(x) = (c_i \leftrightarrow x) & F_1(x, y) = (S_0 \leftrightarrow (x \oplus y)) \\
C_2(x) = (c_o \leftrightarrow x) & F_2(x, y, z) = (x \leftrightarrow (y \vee z)) \\
C_3(x) = (x \leftrightarrow (A_0 \oplus B_0)) & F_3(x, y) = ((((A_0 \wedge B_0) \vee (A_0 \wedge x)) \vee (B_0 \wedge x)) \leftrightarrow y) \\
C_4(x) = (x \leftrightarrow (A_0 \wedge B_0)) & F_4(x) = (((A_0 \leftrightarrow B_0) \leftrightarrow S_0) \leftrightarrow x) \\
C_5(x, y, z) = (x \leftrightarrow (y \wedge z)) &
\end{array}
$$

Les fonctions C_k, $1 \leq k \leq 5$ représentent les motifs extraits de la QBF initiale ; les fonctions C_3, C_4 et C_5 représentent les définitions des variables intermédiaires x_3, x_4 et x_5 ; les fonctions F_k, $1 \leq k \leq 4$, représentent le cœur de la spécification de l'additionneur



$(x_1 \leftrightarrow c_i)$,
$(x_6 \leftrightarrow c_i)$,
$(x_2 \leftrightarrow c_o)$,
$(x_7 \leftrightarrow c_o)$,
$(x_3 \leftrightarrow (A_0 \oplus B_0))$,
$(x_4 \leftrightarrow (A_0 \wedge B_0))$,
$(x_5 \leftrightarrow (x_1 \wedge x_3))$,
$(S_0 \leftrightarrow (x_1 \oplus x_3))$,
$(x_2 \leftrightarrow (x_4 \vee x_5))$,

FIGURE 6 – Une implantation de l'additionneur complet 1-bit.

n-bits.

$$\begin{array}{l}\phi_1 = \forall c_i \forall c_o \forall A_0 \forall B_0 \forall S_0 \\ [\exists x_1 \exists x_2 (((C_1(x_1) \wedge C_2(x_2)) \wedge \\ \quad (\exists x_3 \exists x_4 \exists x_5 (C_3(x_3) \wedge (F_1(x_3, x_1) \wedge (C_4(x_4) \wedge (C_5(x_5, x_1, x_3) \wedge F_2(x_2, x_5, x_4)))))))))] \\ \leftrightarrow [\exists x_6 \exists x_7 ((C_1(x_6) \wedge C_2(x_7)) \wedge (F_3(x_6, x_7) \wedge F_4(x_6)))]\end{array}$$

La QBF ci-dessous $\phi_{1,i}$ est la QBF initiale ϕ_1 mise sous forme prénexe selon l'algorithme classique (les variables y_k sont issues des variables x_k par recopie des sous-formules de la bi-implication).

$$\begin{array}{l}\phi_{1,i} = \forall c_i \forall c_o \forall A_0 \forall B_0 \forall S_0 \forall y_1 \forall y_2 \forall y_3 \forall y_4 \forall y_5 \forall y_6 \forall y_7 \exists x_1 \exists x_2 \exists x_3 \exists x_4 \exists x_5 \exists x_6 \exists x_7 \\ ([C_1(y_1) \wedge C_2(y_2) \wedge C_3(y_3) \wedge (F_1(y_3, y_1) \wedge C_4(y_4) \wedge C_5(y_5, y_1, y_3) \wedge \\ F_2(y_2, y_5, y_4)] \rightarrow [C_1(x_6) \wedge C_2(x_7) \wedge F_3(x_6, x_7) \wedge F_4(x_6)]) \wedge \\ ([C_1(y_6) \wedge C_2(y_7) \wedge F_3(y_6, y_7) \wedge F_4(y_6)] \rightarrow [C_1(x_1) \wedge C_2(x_2) \wedge C_3(x_3) \wedge \\ F_1(x_3, x_1) \wedge C_4(x_4) \wedge C_5(x_5, x_1, x_3) \wedge F_2(x_2, x_5, x_4)])\end{array}$$

L'ordre des variables influe grandement sur l'efficacité des méthodes de résolution [Egly *et al.*, 2003] ; dans la mise sous forme prénexe de ϕ_1 en $\phi_{1,i}$, l'ordre des variables n'est contraint que par le respect de l'ordre partiel qui nécessite que les quantificateurs universels des variables initiales du problème doivent précéder les quantificateurs exis-

n	4	8	12	16	20
$NQ_i(NQ_i^\forall)$	281(16)	541(68)	801(100)	1061(132)	1321(164)
$NQ_r(NQ_r^\forall)$	289(16)	549(68)	809(100)	1069(132)	1329(164)
$NQ_e(NQ_e^\forall)$	147(14)	283(26)	419(38)	555(50)	691(62)
NC_i	766	1478	2190	2902	3614
NC_r	785	1497	2209	2921	3633
NC_e	383	739	1095	1451	1807

TABLE 5 – Tailles des QBF initiales et transformées mises sous CNF pour différentes tailles d'additionneurs n-bits (les clauses sont de taille 2 ou 3).

tentiels.

$\phi_{1,e} = \forall c_i \forall c_o \forall A_0 \forall B_0 \forall S_0 \exists x_1 \exists x_2 \exists x_3 \exists x_4 \exists x_5 \exists x_6 \exists x_7$
$[C_1(x_1) \wedge C_2(x_2) \wedge C_3(x_3) \wedge C_4(x_4) \wedge C_5(x_5, x_1, x_3) \wedge C_1(x_6) \wedge C_2(x_7)] \wedge$
$[[F_1(x_3, x_1) \wedge F_2(x_2, x_5, x_4)] \leftrightarrow [F_3(x_6, x_7) \wedge F_4(x_6)]]$

La QBF ci-dessus $\phi_{1,e}$ est la QBF initiale ϕ_1 transformée à l'aide de l'algorithme $rec\_def\_extract$, puis mise sous forme prénexe selon l'algorithme classique. La matrice de la QBF obtenue est partagée en deux parties : la définition des variables intermédiaires suivie de l'exploitation de ces variables dans le cœur de la bi-implication.

Afin d'étudier le cas général , nous considérons les x_k comme n'étant pas des résultats intermédiaires. Aussi nous introduisons des variables intermédiaires v_k à l'aide du théorème 3.3.4 La QBF $\phi_{1,r}$ est la QBF initiale ϕ_1 mise sous forme prénexe de cette façon :

$\phi_{1,r} = \forall c_i \forall c_o \forall A_0 \forall B_0 \forall S_0 \exists v_0 \exists v_1$
$(v_0 \leftrightarrow [\exists x_1 \exists x_2 (C_1(x_1) \wedge C_2(x_2)) \wedge$
$(\exists x_3 \exists x_4 \exists x_5 (C_3(x_3) \wedge F_1(x_3, x_1) \wedge C_4(x_4) \wedge C_5(x_5, x_1, x_3) \wedge F_2(x_2, x_5, x_4)))]) \wedge$
$(v_1 \leftrightarrow [\exists x_6 \exists x_7 ((C_1(x_6) \wedge C_2(x_7)) \wedge (F_3(x_6, x_7) \wedge F_4(x_6)))]) \wedge (v_0 \leftrightarrow v_1)$

La variable v_0 représente la définition structurelle, alors que la variable v_1 représente la définition comportementale. La suite de la mise sous forme prénexe est faite à l'aide de la méthode classique, sans prendre en compte les résultats intermédiaires :

$\phi_{1,r} = \forall c_i \forall c_o \forall A_0 \forall B_0 \forall S_0 \exists v_0 \exists v_1 \exists x_1 \exists x_2 \exists x_3 \exists x_4 \exists x_5 \exists x_6 \exists x_7 \forall y_1 \forall y_2 \forall y_3 \forall y_4 \forall y_5 \forall y_6 \forall y_7$
$(([C_1(y_1) \wedge C_2(y_2) \wedge C_3(y_3) \wedge F_1(y_3, y_1) \wedge C_4(y_4) \wedge C_5(y_5, y_1, y_3) \wedge$
$F_2(y_2, y_5, y_4)] \rightarrow v_0) \wedge (v_0 \rightarrow [C_1(x_1) \wedge C_2(x_2) \wedge C_3(x_3) \wedge$
$F_1(x_3, x_1) \wedge C_4(x_4) \wedge C_5(x_5, x_1, x_3) \wedge F_2(x_2, x_5, x_4)])) \wedge$
$((v_1 \rightarrow [C_1(x_6) \wedge C_2(x_7) \wedge F_3(x_6, x_7) \wedge F_4(x_6)]) \wedge$
$([C_1(y_6) \wedge C_2(y_7) \wedge F_3(y_6, y_7) \wedge F_4(y_6)] \rightarrow v_1)) \wedge (v_0 \leftrightarrow v_1)$

Il est intéressant de remarquer qu'une fois mise sous forme prénexe, les formules $\phi_{1,r}$ et $\phi_{1,i}$ ont des tailles comparables, $\phi_{1,i}$ possède deux variables en moins avant la mise sous CNF.

Nous avons développé un algorithme en Prolog, qui permet de générer les instances de l'additionneur et d'appliquer l'algorithme $rec\_def\_extract$, puis d'effectuer la mise sous CNF. La table 5 montre le nombre total de quantificateurs (NQ), le nombre de quantificateurs universels ($NQ^{\forall}$) et le nombre de clauses (NC) de la CNF de l'additionneur n-bits, pour différentes valeurs de n. Un indice i signifie qu'il s'agit de la mise sous forme prénexe classique, un r qu'il s'agit de la mise sous forme prénexe par renommage de sous-formules (sans la fusion) et un e qu'il s'agit de la formule obtenue par application de l'algorithme $rec\_def\_extract$. Afin d'alléger la table 5, nous avons seulement détaillé quelques instances. La combinaison de l'algorithme $rec\_def\_extract$ puis de la mise sous forme prénexe par renommage de sous-formules donne exactement $\phi_{n,e}$, obtenant ainsi la forme la plus compacte. Il est intéressant de noter, que pour un n donné sous CNF, la formule $\phi_{n,r}$ possède plus de variables et de clauses que la formule $\phi_{n,i}$ et cela de façon constante : 19 clauses et 8 variables existentielles en plus quel que soit n. Puisque les quantificateurs à sortir de la formule de l'additionneur n'ont qu'une équivalence à traverser, il est normal que les formules $\phi_{n,i}$ et $\phi_{n,r}$ soient très proches. Comme il a été dit dans la section 3.2.2 : sans traiter directement les bi-implications il est impossible d'éviter l'utilisation de l'équivalence 1. Il est donc inévitable de recopier les sous-formules et de créer de nouvelles variables.

3.4.3 Résultats expérimentaux

3.4.3.1 Choix expérimentaux

Avant de présenter nos résultats expérimentaux, il nous faut détailler dans quelles conditions ceux-ci ont été obtenus. Pour cela nous commençons par expliquer les raisons qui ont motivé notre choix de procédures. Puis nous présentons les 2 machines qui ont servi à réaliser les tests. Enfin, nous présentons et motivons notre protocole de test.

Procédures. Lorsque nous avons commencé à réaliser ces tests en 2008, nous avons dû faire notre choix parmi les procédures disponibles et connues. Nous avons donc regardé quelles procédures obtenaient de bons résultats à la compétition *QBFEVAL* de 2006 et 2007 [Narizzano *et al.*, 2006]. Nous voulions aussi voir l'impact de nos propositions selon l'algorithme implanté dans la procédure de résolution (cf. chapitre 2). Pour représenter les procédures de type QDLL (recherche), nous avons choisi `QuBE` [Giunchiglia *et al.*, 2004b]. Mais celui-ci s'étant beaucoup étoffé, notamment en intégrant un préprocesseur intégrant la Q-résolution dans sa version 6.5, nous avons choisi 2 versions : `QuBE-BJ1.2` plus simple et `QuBE6.5` la plus récente à l'époque. Pour représenter les procédures de type résolution nous avons choisi `quantor` [Biere, 2004] dans sa dernière version 3.0. Enfin, nous avons choisi `sKizzo` (dont la dernière version disponible est la `v0.8.2-beta`) pour représenter une procédure avec une approche différente. Nous l'avons classé comme une procédure de *transformation*. Pour rappel, `sKizzo` transforme une QBF en un ou plusieurs problèmes SAT à l'aide de la skolémisation symbolique. Cette procédure a le gros avantage de pouvoir désactiver les différentes méthodes menant à la décision de la QBF. Il est possible par exemple de désactiver la décision par une

procédure SAT, `sKizzo` utilise alors les autres techniques encore activées pour décider de la validité de la QBF. Toutes ces procédures s'exécutent de façon déterministe : pour une instance et des paramètres donnés, l'exécution est identique.

Machines et protocole pour les tests. Lors de nos premiers tests nous avions un Intel Pentium 4® à 2.80GHz avec 600Mo de mémoire vive disponible et la technologie « Hyper-Threading » activée. L'hyper-threading consiste à émuler deux processeurs logiques sur un processeur physique. Ces deux processeurs virtuels partagent donc le processeur, le cache et le bus système réel. Puis, les tests ont été effectués sur un Intel Xeon® à 2.50GHz avec 4Go de mémoire vive disponible. Sauf mention contraire, tous les résultats présentés sont ceux réalisés par cette machine. Nous avons fixé arbitrairement la limite de temps pour tous nos tests à 3600 secondes. Les exécutions sont séquentielles et les machines sont chargées le moins possible et dédiées aux tests. Pour le problème des chaînes de bi-implications, nous n'avons effectué qu'une seule exécution pour chaque instance et chaque procédure. D'une part, nous avons constaté de faibles différences en répétant les exécutions. D'autre part, nous sommes intéressés par l'ordre de grandeur des résultats et non par une évaluation précise du gain de chaque méthode. Pour l'additionneur n-bits, nous avons vérifié aussi s'il était possible de limiter le nombre d'exécutions. Nous avons donc réalisé l'ensemble des tests une fois sur toutes les instances. Puis nous avons relancé des exécutions sur des instances où les différentes méthodes de mise sous forme prénexe obtenaient des résultats proches. Nous n'avons pas constaté de différences justifiant de nouvelles exécutions vis-à-vis des observations que nous souhaitions faire.

3.4.3.2 Chaîne de bi-implications

Nous mesurons maintenant l'impact de nos propositions sur le temps de résolution des instances des chaînes de bi-implications par différentes procédures. Les résultats sont résumés dans les figures 7, 9 et 8. Pour ce problème, `QuBE6.5` donne toujours de meilleurs résultats que `QuBE-BJ1.2`. Nous avons donc occulté les résultats pour ce dernier. Par souci de clarté, nous n'avons pas consigné sur ces figures les résultats obtenus avec les formules $\psi_{n,r}$. Les résultats pour ces formules sont sensiblement les mêmes que pour les $\psi_{n,f}$ pour toutes les procédures avec les réglages par défaut.

Sur cet exemple, la méthode de mise sous forme prénexe par renommage de sous-formules donne d'aussi bons résultats que la méthode traitant particulièrement les résultats intermédiaires à l'aide de l'algorithme $rec\_def\_extract$. Or ces deux méthodes ne sont pas incompatibles et l'étude expérimentale n'a pas pour but de les opposer. Si une formule possède des résultats intermédiaires, nous sommes capables de les extraire de façon optimale vis-à-vis de la taille de la formule. Si une formule possède des quantificateurs à l'intérieur d'une ou plusieurs bi-implications, nous sommes capables de les extraire en limitant l'impact de la mise sous forme prénexe sur la taille de la formule par une technique de renommage de sous-formules. Nos deux propositions permettent de réduire sensiblement le temps de calcul pour les procédures testées. A priori, nous nous attendons à obtenir les meilleurs résultats avec l'algorithme $rec\_def\_extract$.

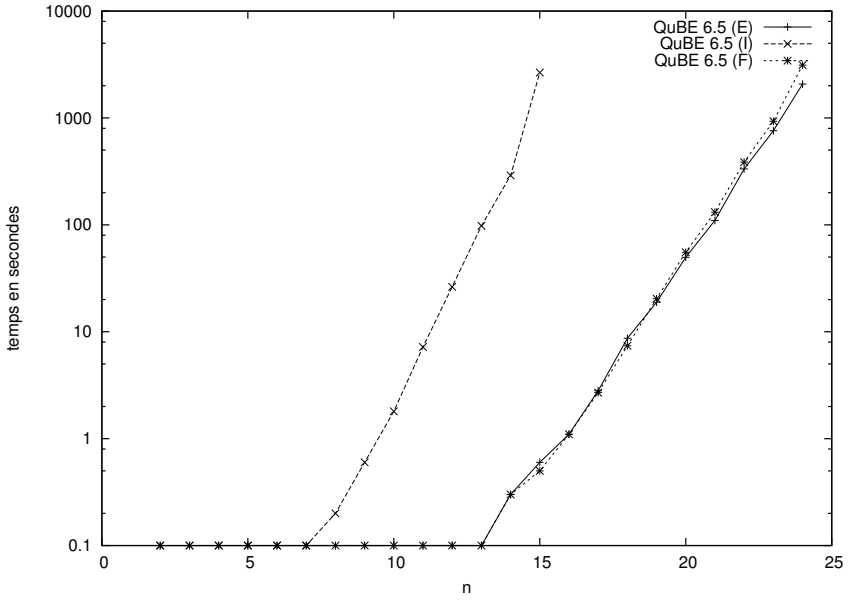


FIGURE 7 – Temps de calcul pour `QuBE6.5` avec les chaînes de bi-implications mises sous forme prénexe de façon classique (I), avec l'algorithme $rec\_def\_extract$ (E) et par renommage de sous-formules et fusion de quantificateurs (F). Le temps en ordonnée est en échelle logarithmique.

La procédure `QuBE`, malgré une nette amélioration, semble être la procédure pour laquelle les formules $\psi_{n,e}$ et $\psi_{n,f}$ profitent le moins. Il est probable que le problème de chaîne de bi-implications soit difficile pour les procédures de recherche. À la vue des résultats des autres procédures, l'invalidité des formules semble plus facile à obtenir par un algorithme d'élimination de quantificateur « *bottom-up* » ou de façon symbolique. La procédure `quantor` est celle qui avec les réglages par défaut obtient les meilleurs temps avec les formules $\psi_{n,e}$ et $\psi_{n,f}$, alors que l'instance $\psi_{n,i}$ pour $n = 5$ dépasse le temps imparti. La formule originale avant mise sous CNF prénexe est pourtant très petite :

$$\begin{aligned}\psi_5 \;=\; & \exists e_3 \exists e_2 \exists e_1 \exists e_0 \forall u_0 \forall u_1 \forall u_2 ((\exists x_5((x_5 \leftrightarrow (e_1 \vee e_2)) \wedge (x_5 \wedge e_3))) \leftrightarrow \\ & ((\exists x_4((x_4 \leftrightarrow (e_0 \vee e_1)) \wedge (x_4 \wedge e_2))) \leftrightarrow ((\exists x_3((x_3 \leftrightarrow (u_2 \vee e_0)) \wedge (x_3 \wedge e_1))) \leftrightarrow \\ & ((\exists x_2((x_2 \leftrightarrow (u_1 \vee u_2)) \wedge (x_2 \wedge e_0))) \leftrightarrow (\exists x_1((x_1 \leftrightarrow (u_0 \vee u_1)) \wedge (x_1 \wedge u_2)))))))\end{aligned}$$

Pour la procédure `sKizzo`, les formules $\psi_{n,e}$ et $\psi_{n,f}$ permettent une amélioration du temps de résolution comparable à `quantor`. Avec la formule $\psi_{n,i}$ pour $n = 7$, `sKizzo` dépasse le temps imparti. Là aussi, $\psi_{7,i}$ a été obtenue à partir d'une formule très petite. Les formules $\psi_{n,f}$ demandent une étude supplémentaire pour évaluer l'impact de la fusion de quantificateurs.

En effet, les trois procédures utilisent une structure de quantificateurs [Giunchiglia *et al.*, 2006c; Biere, 2004; Benedetti, 2005a]. Elles construisent un arbre de quantificateurs ou utilisent une technique de réduction de portée (*miniscoping* par exemple). Dans un arbre de quantificateurs, chaque feuille est étiquetée par un ensemble de clauses, afin de

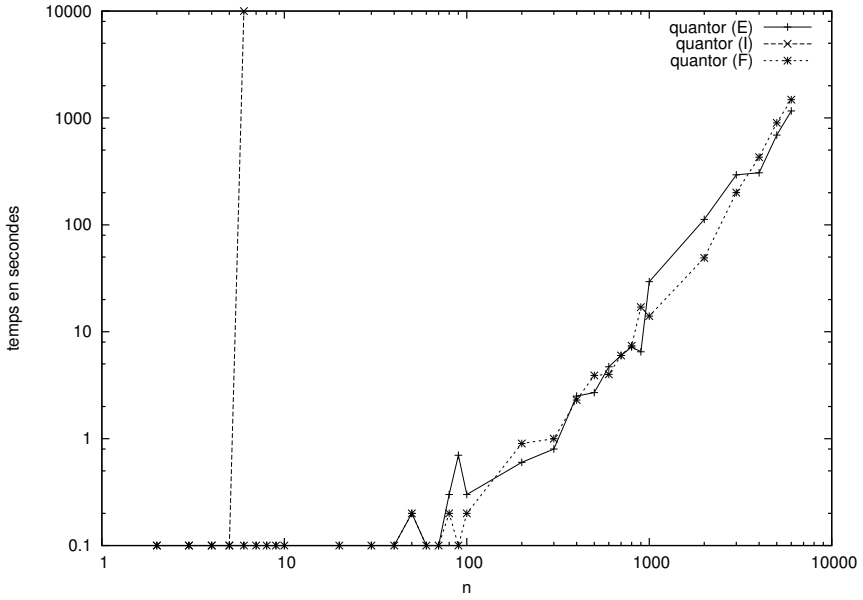


FIGURE 8 – Temps de calcul pour `quantor` avec les chaînes de bi-implications mises sous forme prénexe de façon classique (I), avec l'algorithme $rec\_def\_extract$ (E) et par renommage de sous-formules et fusion de quantificateurs (F). Les deux axes sont en échelle logarithmique.

réduire la portée de chaque quantificateur. Cela revient à appliquer les équivalences de la section 3.2.2 de la gauche vers la droite. Cette technique peut être antagoniste à notre fusion par application de l'équivalence 11 dans le sens inverse. Pour illustrer l'intérêt de la fusion, nous devons désactiver la construction de la structure de quantificateurs. Seul `sKizzo` permet de paramétrer aussi finement son exécution. Pour ce problème, l'utilisation de la fusion de quantificateurs permet un gain important pour une procédure sans structure de quantificateurs. Sans la fusion, nous avons besoin d'introduire n variables quantifiées universellement lors de la mise sous forme prénexe, mais elles peuvent toutes fusionner en une seule. Les résultats sont présentés dans la table 6 (p. 68) et dans la figure 9 ci-contre par la courbe (LF). Dans la table, un 'M' indique un dépassement de la mémoire disponible et un '-' indique que l'instance n'a pas été testée. Les résultats sont meilleurs qu'avec la construction de l'arbre des quantificateurs. L'explication qui nous semble la plus probable vient de la structure des formules : chaque maillon possède trois variables libres, chaque variable libre est présente dans trois maillons et trois variables libres ne sont présentes ensemble que dans un unique maillon. Par conséquent, l'algorithme de construction de l'arbre doit choisir arbitrairement des quantificateurs afin de réduire leur portée. Cet exemple montre que ce n'est pas toujours la meilleure solution et cela rejoint les résultats sur la gestion des dépendances entre quantificateurs dans les QBF [Lonsing et Biere, 2009]. Plus surprenant, ce sont les résultats avec fusion et sans reconstruction de l'arbre de quantificateurs qui sont bien meilleurs que ceux de `sKizzo` avec

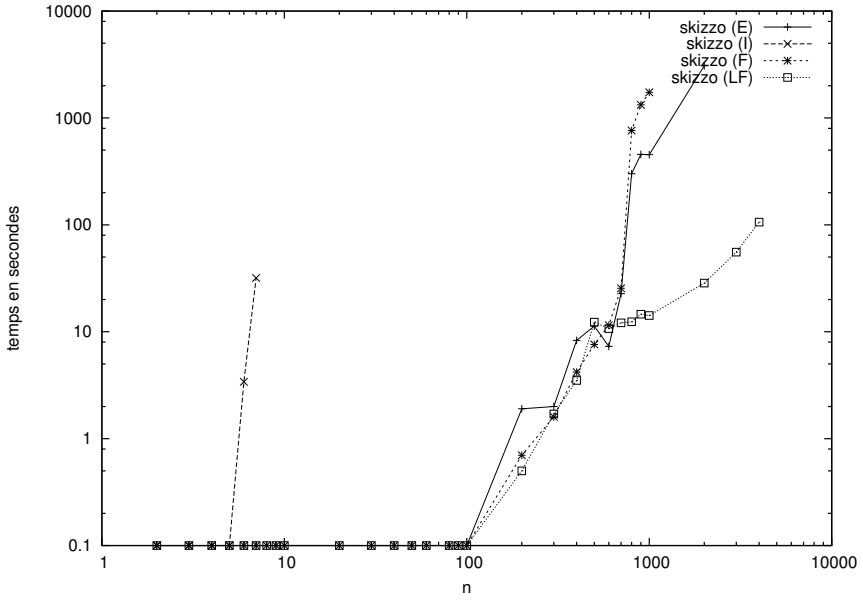


FIGURE 9 – Temps de calcul pour `sKizzo` avec les chaînes de bi-implications mises sous forme prénexe de façon classique (I), avec l'algorithme $rec\_def\_extract$ (E), par renommage de sous-formules et fusion de quantificateurs (F) et par renommage de sous-formules, fusion de quantificateurs et sans construction de l'arbre de quantificateurs (LF). Les deux axes sont en échelle logarithmique.

les autres méthodes ou même `quantor`. Or, nous pensions que les formules $\psi_{n,e}$ seraient la référence. En effet l'application de l'algorithme $rec\_def\_extract$ sur ce problème permet d'obtenir les formules les plus concises et présentant la plus faible complexité dans la hiérarchie polynomiale. La mise sous forme prénexe par renommage de sous-formules extrait chaque maillon entièrement. Il ne reste plus que les variables des définitions des maillons dans la chaîne de bi-implication (cf. fin de la section 3.4.1). Alors qu'avec l'algorithme $rec\_def\_extract$ ce sont toujours des formules qui sont dans les maillons. Autre hypothèse que nous pensons plus vraisemblable : l'utilisation de l'équivalence 1 pour traiter les bi-implication contenant encore des bi-implications dans les formules $\psi_{n,r}$ et $\psi_{n,f}$ permet de générer plus de clauses, faisant ainsi un effet d'apprentissage de lemmes (*Clause learning*). Cela permet de conclure à la falsification de la formule pour certaines affectations par l'obtention plus rapide de la clause vide. Sans la fusion de quantificateurs, cet apprentissage n'est pas efficace, car l'information supplémentaire est représentée par plusieurs variables universelles sans lien. Une fois fusionnées, ces variables sont dans un grand ensemble de clauses contenant chacune la variable issue de la fusion. Toutes ces clauses vont permettent d'élaguer la recherche et plus le problème sera grand plus cette variable aura un poids élevé. Il est même possible que l'ensemble de clauses contenant cette variable soit à lui seul insatisfiable pour ce problème. Nous pouvons constater sur la courbe (LF) de la figure 9, un point d'inflexion vers $n = 500$. Or n représente le nombre

n	20	30	100	200	800	1000	2000	4000
$\psi_{n,r}$	199,4	M	-	-	-	-	-	-
$\psi_{n,f}$	<0,1	<0,1	<0,1	0,5	12,4	14,2	28,5	106,2

TABLE 6 – Temps de calcul pour `sKizzo` sans construction de l'arbre de quantificateurs, avec les chaînes de bi-implications mises sous forme prénexe par renommage de sous-formules ($\psi_{n,r}$) et par renommage de sous-formules et fusion de quantificateurs ($\psi_{n,f}$).

de maillons, donc aussi le nombre de variables universelles qui fusionnent. Nous pensons qu'à partir de $n = 500$, Le nombre d'occurrences de la variable résultant de la fusion dépasse un certain seuil au niveau d'une heuristique de choix de variables dans `sKizzo`.

Il est tout de même un peu gênant d'obtenir de meilleurs résultats avec les formules $\psi_{n,f}$, dans lesquelles sont ignorées de manière délibérée les définitions des résultats intermédiaires, alors que nous motivons notre travail par l'obtention de formules plus compactes et dans une classe de complexité du langage plus faible. Mais il faut tout de même nuancer ces résultats :

- vu sous un autre angle, notre travail contribue à montrer la difficulté liée au codage d'un problème dans le langage **QBF** et les progrès qui peuvent encore être faits dans ce domaine. Or, les procédures disponibles ont comme point de repère des benchmarks dont le codage n'est pas nécessairement optimal. Les procédures ont appris à être plus efficaces sur les codages disponibles et peuvent peut être progresser d'autant plus maintenant sur un codage plus compact ;
- le codage de notre problème de chaîne de bi-implication est mauvais. En effet, dans chaque maillon il y a une définition d'un résultat intermédiaire qui ne sera utilisé qu'une seule fois. Il est peu probable de coder ainsi ce genre de formule ou un simple remplacement syntaxique serait efficace. C'est un problème artificiel créé originalement pour montrer l'efficacité des nos propositions sur la taille de la formule mise sous forme prénexe. De plus, la définition du résultat intermédiaire est très petite. Il faudrait vérifier si des résultats similaires sont observables sur une plus grande variété de formules structurées de la sorte ;
- les formules sont toutes insatisfiables, l'augmentation du nombre de clauses peut potentiellement mener plus vite à la falsification de la formule dans le cas de $\psi_{n,f}$. Il serait intéressant de travailler avec des formules structurées de façon similaire mais valides.

3.4.3.3 L'additionneur n-bits

Nous mesurons maintenant l'impact de nos propositions sur le temps de résolution des instances de l'additionneur n-bits par différentes procédures. Les résultats sont résumés dans les tables 6 et 7, les temps sont en secondes, un 'T' indique que la procédure n'a pas pu résoudre l'instance en moins de 3600 secondes et un 'M' indique un dépassement de la mémoire disponible. `QuBE` quelle que soit sa version n'a pas réussi à résoudre le problème pour $n \geq 3$ avec les QBF mise sous forme prénexe par la méthode classique

	QuBE		quantor			sKizzo sans Q-résolution		
	6.5	BJ1.2						
n	$\phi_{n,e}$	$\phi_{n,e}$	$\phi_{n,i}$	$\phi_{n,r}$	$\phi_{n,e}$	$\phi_{n,i}$	$\phi_{n,r}$	$\phi_{n,e}$
4	0,7	**0,1**	5,3	**0,1**	**0,1**	0,2	**0,1**	**0,1**
6	61,4	9,8	T	5,8	1,8	1,3	0,2	**0,1**
8	T	714,5	M	122,9	106,0	3,3	0,6	**0,1**
10	T	T	M	M	M	3,2	1,0	**0,1**
12	T	T	M	M	M	4,8	1,7	**0,1**
14	T	T	M	M	M	5,3	2,1	**0,3**
16	T	T	M	M	M	6,5	3,8	**0,3**
18	T	T	M	M	M	14,0	6,2	**0,5**
20	T	T	M	M	M	28,9	7,8	**0,4**
22	T	T	M	M	M	19,6	10,3	**1,0**

TABLE 7 – Temps de calcul pour les différentes procédures et instances de l'additionneur n-bits mises sous forme prénexe de façon classique ($\phi_{n,i}$), par renommage de sous-formules ($\phi_{n,r}$) et avec l'algorithme $rec\_def\_extract$ ($\phi_{n,e}$).

($\phi_{n,i}$) et mise sous forme prénexe par renommage de sous-formules ($\phi_{n,r}$). L'amélioration en terme de temps apportée par les codages obtenus par nos deux méthodes est visible et assez comparable pour la procédure `quantor`. Pour la procédure `sKizzo` après avoir désactivé la Q-résolution, les formules $\phi_{n,e}$ permettent d'obtenir de bien meilleurs temps. Avec `sKizzo` comme avec `quantor` il est assez étonnant d'obtenir de meilleurs résultats avec les formules obtenues par renommage de sous-formules plutôt qu'avec la méthode classique. Comme nous l'avons fait remarquer dans la section 3.4.2, les formules $\phi_{n,r}$ possèdent légèrement plus de variables et de clauses que les formules $\phi_{n,i}$. Le problème de l'additionneur n-bits possède une bi-implication centrale qui sera la seule à être traversée lors de la mise sous forme prénexe. Le renommage de sous-formule n'apporte donc rien de plus pour la taille de la formule que la méthode classique. Contrairement aux chaînes d'implications, l'extraction des définitions des résultats intermédiaires est plus efficace, mais cette fois-ci les problèmes sont valides. Face à la méthode classique, il semble que le renommage de sous-formules de manière à extraire les deux côtés de la bi-implication, permette d'obtenir de meilleurs temps. Cela malgré encore une fois, un codage moins compact. Il faut tout de même émettre la même réserve que pour les chaînes de bi-implication concernant les procédures. Celles-ci sont complexes et calibrées pour résoudre efficacement les instances disponibles.

Lors de nos premiers tests (en partie gauche de la table 8 p. 70) avec le Pentium 4® avec 600Mo de mémoire et `sKizzo`, pour $n \leq 10$, l'amélioration avec les formules $\phi_{n,e}$ est significative. Mais ces performances décroissent ensuite pour des tailles supérieures. Puisque `QuBE6.5`, qui intègre aussi la Q-résolution comme préprocesseur, obtient des résultats pires que ceux de `QuBE-BJ1.2` qui est sensé être une version moins performante, nous pensons que la Q-résolution a un grand impact sur ce type de problème.

	sKizzo avec Q-résolution Pentium 4® - 600Mo			sKizzo avec Q-résolution Xeon® - 4Go		
n	$\phi_{n,i}$	$\phi_{n,r}$	$\phi_{n,e}$	$\phi_{n,i}$	$\phi_{n,r}$	$\phi_{n,e}$
4	0,3	0,2	**0,1**	**0,1**	**0,1**	**0,1**
6	1,0	0,9	**0,1**	0,4	0,4	**0,1**
8	5,5	2,9	0,4	2,0	1,2	**0,2**
10	23,7	67,8	6,0	7,5	10,7	**1,6**
12	170,9	507,8	143,7	14,1	23,2	**11,9**
14	915,6	T	T	16,3	74,0	**15,6**
16	T	T	T	33,5	74,4	**29,3**
18	T	T	T	19,3	49,4	**14,4**
20	T	T	T	82,5	78,5	**21,7**
22	T	T	T	35,7	122,9	**14,0**

TABLE 8 – Résultats de sKizzo pour l'additionneur sur différentes machines.

Peut-être que la Q-résolution semble travailler à l'encontre des autres méthodes pour ce problème. Il est possible que la résolution sur les variables intermédiaires (du codage ou de la mise sous CNF) enlève des possibilités aux procédures d'élaguer l'espace de recherche. Les mécanismes d'apprentissage retiennent des règles moins générales, le parcours de l'espace de recherche serait plus long. Nous avons effectué des tests en désactivant la Q-résolution de `sKizzo` et nous avons des résultats différents plus conformes à nos attentes. Sur une machine avec plus de mémoire (4Go) les résultats sont différents : les temps obtenus avec les formules $\phi_{n,e}$ sont inférieurs à ceux obtenus avec le codage original. La table 8 résume les tests effectués sur les 2 machines différentes avec la Q-résolution. Bien que la vitesse des processeurs soient différentes, cela ne suffit pas à expliquer les temps de calcul. En effet, d'une machine à l'autre, les temps ne sont pas proportionnels à la vitesse du processeur. L'élimination de variables à l'aide de la Q-résolution peut potentiellement générer un grand nombre de clauses et l'utilisation intensive de la mémoire a aussi un coût en temps. Il semble que pour ce problème la Q-résolution en nécessite en grande quantité. Mais avec suffisamment de mémoire l'extraction des définitions des résultats intermédiaires obtient les meilleurs résultats malgré la Q-résolution. Quoi qu'il en soit, désactiver la Q-résolution permet d'obtenir un gain de temps significatif pour tous les codages.

Deuxième partie

Architecture parallèle et QBF

Chapitre 4

État de l'art pour l'architecture parallèle et application aux QBF

Sommaire

4.1 Introduction

Les concepts de *programmation parallèle* ou *parallélisme* sont de plus en plus au centre du développement logiciel, sûrement parce que même les ordinateurs personnels aujourd'hui sont majoritairement multi-cœurs (multi-core) et que cette puissance disponible peut être utilisée. Certains téléphones portables possèdent eux aussi plusieurs unités de calcul et les cartes graphiques accélératrices pour la 3D contiennent des centaines de cœurs qui permettent de faire du calcul dans certains domaines autres que la 3D. Dans un futur proche, les ordinateurs personnels pourront avoir 6 voire 8 cœurs et contiennent déjà aujourd'hui un processeur graphique plus ou moins puissant. Dans les laboratoires de la firme Intel®, grand fabriquant de microprocesseurs, des recherches portent sur les architectures dites *many-cores*, laissant sous-entendre que l'avenir des microprocesseurs sera dans la multiplication des unités de calcul. Celles-ci pourront être plus simples et moins performantes que les cœurs actuels, mais elles seront en très grand nombre. En ce qui concerne notre sujet de recherche, les procédures de résolution pour les formules booléennes quantifiées sont majoritairement séquentielles. Si effectivement, dans l'avenir les unités de calcul sont moins performantes, les procédures actuelles seront moins efficaces sur les architectures matérielles futures ou du moins sur celles qui reposeront sur la technologie des semi-conducteurs (en silicium). Malheureusement, il n'existe pas encore d'alternative opérationnelle pour prendre le relais, bien que des recherches sont en cours pour des processeurs optiques et quantiques. Pour continuer de bénéficier des progrès technologiques actuels et futurs, les procédures doivent prendre en compte l'aspect multi-core et many-core. Mais pourquoi la question du calcul en parallèle ne s'est-elle pas posée avant pour les QBF ? En effet, le besoin croissant de puissance n'est pas récent. Ce qui est récent, c'est l'apparition de machine dotée de capacité parallèle pour le grand public et la démocratisation du *calcul haute performance* ou *HPC* (« *High Performance Computing* »). Les super-calculateurs et la programmation parallèle existent depuis déjà plusieurs décennies, par exemple pour les simulations météorologiques. Dans la section 4.2, nous détaillons les conditions qui ont mené à des machines parallèles pour le grand public. Dans la section 4.3 nous introduisons le vocabulaire et les notions nécessaires à un propos sur la programmation parallèle. Ensuite, la section 4.4 présente brièvement les architectures matérielles de calcul, de mémoire et de communication, alors que la section 4.5 présente différents paradigmes de programmation parallèle. Nous détaillons le concept de programmation parallèle *par passage de messages* pour lequel MPI est une spécification. Puis dans la section 4.6, nous expliquons pourquoi décider les QBF en parallèle n'est pas chose facile et ce qui motive un choix d'architecture cible. Enfin, nous faisons un panorama des procédures QBF parallèles de l'état de l'art.

4.2 Transistors, loi de Moore et limites

Le *transistor* est un élément fondamental en électronique et a fortiori en électronique numérique. Dans ce domaine, il est utilisé pour se comporter comme un interrupteur commandé et permet de réaliser des opérations logiques : une porte AND ou OR néces-

Année	Série/modèle	fréquence	gravure	nb. transistors	cœurs	SMT
1971	4004	740 kHz	10 μm	2250	1	-
1973	8080	2 MHz	6 μm	6000	1	-
1976	8085	5 MHz	3 μm	6500	1	-
1978	8086	5 MHz	3 μm	29000	1	-
1982	80286	6 MHz	1.5 μm	134000	1	-
1985	386	16 MHz	1.5 μm	275000	1	-
1989	486	25 MHz	1 μm	1180000	1	-
1993	Pentium	60 MHz	0.8 μm	3100000	1	-
1995	Pentium PRO	200 MHz	0.35 μm	5500000	1	-
1997	Pentium II	233 MHz	0.35 μm	7500000	1	-
1999	Pentium III	450 MHz	0.25 μm	9500000	1	-
2000	Pentium IV	1.4 GHz	180 nm	42000000	1	-
2003	Pentium IV extreme	3.8 GHz	130 nm	175000000	1	oui
2005	Pentium D	3.0 GHz	90 nm	230000000	2	-
2006	Core 2	3 GHz	65 nm	582000000	2-4	-
2008	Core i7	3.2 GHz	45 nm	730000000	4-6	oui
2010	Core i7 980X	3.6 GHz	32 nm	1170000000	6	oui
2010	Xeon X7560	2.26 GHz	45 nm	2300000000	8	oui
2001	Itanium	733 MHz	180 nm	25000000	1	-
2002	Itanium II	1 GHz	130 nm	410000000	1	-
2006	Itanium 2 Duo	1.6 GHz	90 nm	1700000000	2	-
2010	Itanium 9350	1.73 GHz	64 nm	2046000000	4	oui

TABLE 9 – Quelques processeurs Intel de 1971 à 2010. SMT signifie « *Simultaneous Multi Threading* », c'est la capacité d'exécuter plusieurs threads (ici 2) sur le même pipeline d'exécution d'un cœur.

site 6 transistors en pratique. Les premiers transistors ont été inventé aux Bell Laboratories® en 1947 et ceux de la deuxième génération sont fabriqués pour la première fois en silicium en 1954.

Le premier microprocesseurs date de 1971, il s'agit du 4004, d'Intel®, il contient 2250 transistors. Un peu avant, en 1965, Gordon Moore, alors ingénieur chez Fairchild Semiconductor, constate que depuis 1959, le nombre de transistors dans les produits proposés double tous les ans à coût constant. Le circuit le plus dense à l'époque contenait 64 transistors. Il co-fonde la société Intel® en 1968 avec Robert Noyce et Andrew Grove. Il modifie alors sa prédiction d'évolution en 1975 et se place dans le cadre des microprocesseurs. Il prédit alors ce qui est appelé communément la *loi de Moore* : le nombre de transistors des microprocesseurs sur une puce de silicium double tous les deux ans. Ce qui s'est traduit pendant très longtemps par un gain en fréquence de fonctionnement et donc une amélioration des performances.

Malheureusement, le silicium a des limites. Pour intégrer toujours plus de transistors, il faut maîtriser le processus de gravure, la *photolithographie*, pour des dimensions

toujours plus petite. Or, il est quasi impossible de concentrer efficacement les rayons pour les longueurs d'onde nécessaires à des gravures plus fine. Le défi technique est vraiment important. D'un point de vue matière, une piste dans un microprocesseur ne peut pas être plus petite que quelques atomes, dans l'industrie du silicium, cette limite s'appelle le mur ou « *the wall* » et il s'agit d'une limite théorique. Aujourd'hui, il est possible de repousser les dimensions des transistors à environ 20nm, soit environ 80 atomes mis côte à côte, mais vers 2015 nous serons confrontés à de nouveaux effets quantiques si nous arrivons à graver de façon toujours plus fine. S'il s'agissait des seuls problèmes, nous pourrions augmenter la fréquence des processeurs jusqu'en 2015. Pourtant Intel® lançait son premier processeur à 2 cœurs en 2005 pour augmenter les performances des processeurs sans augmenter la fréquence. Le principal point limitant la montée en fréquence n'est autre que la consommation d'énergie et la chaleur générée. Le silicium est un semi-conducteur, sa propriété qui nous intéresse est sa capacité à laisser passer ou non les électrons avec certaines probabilités et dans certaines conditions. A fréquence élevée, de nombreux électrons se perdent en route contribuant à générer des erreurs mais aussi à faire chauffer le processeur. Or, les propriétés du silicium changent avec la chaleur et encore plus d'électrons se perdent dans le substrat. Pour éviter les erreurs il faut augmenter la tension du processeur, augmentant encore le nombre d'électrons perdus et la chaleur. Le coût énergétique devient vite élevé. De plus, il faudrait mettre au point des systèmes de refroidissement plus performants avec une surface de dissipation thermique plus grande. L'encombrement, le poids et le coût des matériaux (cuivre et aluminium par exemple) ne permettent pas d'envisager ce genre de solution, surtout dans le cas d'un matériel portable. La consommation électrique serait également trop importante alors que l'on commence à peine à intégrer les notions de développement durable. Inversement, un processeur gravé plus finement consomme moins d'énergie et prend moins de place. Il est alors possible de mettre plusieurs processeurs là où un seul tenait. Dans le cas des cartes graphiques, il est possible de mettre un très grand nombre (des centaines) d'unités de calcul simples : les processeurs de flux. Sans augmentation de la fréquence des processeurs, le seul moyen de calculer plus vite est d'utiliser plusieurs unités de calculs : il faut donc apprendre à développer des algorithmes efficaces en parallèle.

Nous avons listé dans la table 9 quelques processeurs Intel® de 1971 à 2010. Nous avons choisi de limiter arbitrairement la présentation aux processeurs de cette marque. En 2010, le processeur Core i7 980X, destiné au particulier, possède 6 cœurs et peut exécuter 12 threads simultanément. Le Xeon X7560, plutôt destiné aux solutions moyennes pour le calcul haute performance et ses 2,3 milliards de transistors possède 8 cœurs et peut exécuter 16 threads simultanément. Il est possible de mettre jusqu'à 8 processeurs de ce type dans un serveur. Dans la partie basse du tableau, figurent les processeurs Itanium qui sont un peu spécifiques. En novembre 2004, le calculateur disposant de la seconde plus importante puissance de calcul au monde est basé sur une plate-forme mettant en œuvre 10 240 processeurs Itanium 2. L'Itanium 2 Duo avec ses 1,7 milliard de transistors dépassait largement les prévisions de la loi de Moore pour 2006, qui prévoyait environ 400 millions de transistors. Ce genre d'architecture est capable d'accueillir des dizaines de processeurs dans la même machine. Par exemple, SGI® proposait en 2005 une gamme de serveur Silicon Graphics Prism contenant jusqu'à 256 processeurs Itanium 2 et

16 cartes graphiques dédiées au calcul. Il est aujourd'hui possible d'avoir une puissance de calcul phénoménale, encore faut-il réussir à l'exploiter efficacement en temps et en énergie.

4.3 Parallélisme et efficacité

Comme nous venons de le montrer, la puissance de calcul à notre disposition n'a cessé d'augmenter, dans un premier temps par une augmentation de la fréquence puis maintenant par la multiplication des unités de calcul dans une même machine. Pour exploiter cette puissance de calcul démultipliée, la première solution est d'utiliser les mêmes programmes qu'auparavant, car il est alors possible d'en exécuter un plus grand nombre simultanément. C'est assez efficace avec les programmes en informatique conventionnelle : l'ordinateur peut réaliser de nombreuses tâches usuelles en même temps. La deuxième solution consiste à exécuter un programme sur plusieurs unités de calcul. C'est le cas par exemple dans les jeux qui nécessitent plusieurs unités généralistes et une carte graphique accélératrice afin de proposer un niveau de réalisme toujours plus grand. C'est aussi le cas dans le calcul scientifique, où la quantité de données ne cesse de croître, nécessitant toujours plus de capacités de calcul. C'est une possibilité aussi pour des tâches de raisonnement en intelligence articificielle et dans notre cas pour la décision du problème de validité des QBF. Nous utilisons le mot *parallélisme* pour décrire la capacité d'un programme à exécuter simultanément plusieurs instructions sur plusieurs unités de calcul ou non. Nous verrons dans la section suivante que les architectures matérielles permettant du parallélisme existent depuis longtemps et ne possèdent pas forcément plusieurs processeurs ou même cœurs. Avant cela, nous devons définir quelques notions liées au parallélisme utiles à notre propos.

La plus petite unité d'exécution d'un programme est *l'instruction*, bien qu'elle puisse nécessiter plus d'un cycle d'horloge. Le parallélisme au niveau des instructions existe et est même présent naturellement dans les processeurs même mono-cœur. Ce type de parallélisme nous intéresse peu au niveau d'une application. D'une part il est dépendant de l'architecture utilisée, d'autre part nous avons peu de moyens de le gérer. Pour le parallélisme qui nous intéresse, celui que nous pouvons et devons gérer, la plus petite unité d'exécution est la *tâche* ou le *travail*. C'est à nous de découper nos données ou nos procédures en plus petits grains. La *granularité* d'un programme parallèle est le nombre de travaux réalisés par chaque processeur. Il convient pour chaque problème de trouver un bon découpage pour au moins donner une tâche à chaque unité de calcul et il convient de ne pas trop découper le problème surtout ci cela engendre un surcoût important. Cependant, même avec une granularité suffisante, la dépendance entre les tâches ne permet pas toujours d'occuper toutes les ressources. On parle de *degré de parallélisme* pour faire référence au nombre de tâches pouvant être calculées en même temps. Cette mesure peut varier au cours de l'exécution.

Pour parler de l'exécution d'un programme parallèle, certaines notions sont nécessaires. Le *temps d'exécution* d'un programme séquentiel est assez simple à définir. Pour une exécution parallèle, il est défini comme le temps écoulé entre le lancement d'un pro-

gramme et l'instant où le dernier processeur termine son travail. Il est possible alors de définir *l'accélération relative* ou « *speedup* » comme étant le rapport entre le temps de l'exécution séquentielle et le temps d'exécution en parallèle. Directement liée à cette notion, *l'efficacité* d'un programme parallèle avec n processeurs est l'accélération obtenue avec n processeurs divisée par n. Un algorithme parallèle est *extensible* s'il est capable d'au moins maintenir son efficacité quel que soit le nombre de processeurs. Dans le cas contraire, il existe un nombre de processeurs au-delà duquel il est inutile d'exécuter le programme. Dans le pire des cas, l'accélération décroît au-delà d'un certain nombre de processeurs. D'un point de vue performance et développement durable, il est important d'utiliser ce type d'algorithme dans sa zone d'efficacité, sinon le surcoût en énergie ne sera pas compensé par le gain de vitesse. Certains algorithmes peuvent être particulièrement rentables pour un certain nombre de processeurs. Cela se produit lorsque l'accélération pour n processeurs est supérieure à n, le calcul en parallèle a alors permis de ne pas réaliser toutes les tâches. L'accélération est dite *super linéaire*. Cette propriété a été observée pour des procédures parallèles de décision pour SAT et QBF et est particulièrement intéressante pour des problèmes de complexité supérieure ou égale à **NP**. Cette propriété dépend surtout du type de problème à résoudre. Par exemple l'échange d'informations sur le problème peut permettre de réduire le travail nécessaire. D'une manière générale, pour un problème de décision, il est possible d'arrêter tous les processeurs dès la découverte d'une solution, le gain peut alors être super linéaire.

Pour concevoir un algorithme parallèle efficace, la notion *d'équilibrage de charge* est importante. Quand les tâches possèdent peu de dépendances, la chose est plus aisée. Il est par exemple très facile de paralléliser l'addition de 2 vecteurs ou des traitements pour le rendu graphique en 3 dimensions. De plus, les processeurs travailleront pendant un temps presque identique et prévisible. Lorsque les tâches sont interdépendantes, comme nous verrons pour QBF, la problématique de l'équilibrage de charge est capitale car la durée de chaque tâche est imprévisible. Il faut minimiser les temps où les processeurs sont au repos alors que nous en avons requis l'utilisation.

4.4 Les architectures matérielles

4.4.1 La machine de von Neumann et ses extensions

La machine de von Neumann est composée d'un *processeur* (ou CPU), de *mémoire* et d'entrées/sorties pour interagir avec le monde extérieur. Dans cette architecture, le processeur contient une *unité arithmétique et logique* (ou ALU) et une *unité de contrôle*. Cette dernière est chargée de diriger l'exécution du programme, alors que l'ALU fournit les opérations appelées dans le programme et effectue donc les calculs. La mémoire est divisée en plusieurs catégories. La mémoire volatile, comme les registres et la mémoire centrale, sert aux programmes et données en cours de fonctionnement. La mémoire permanente comme le disque dur de nos jours, sert à stocker les programmes et le système de base de la machine. Quelle soit volatile ou non, la mémoire stocke aussi bien les programmes que les données. Pour être utilisées, les instructions et les données doivent être

chargées dans les registres, des petites zones de mémoire très rapides mais très chères, donc en faible quantité. Le goulot d'étranglement pour la machine de von Neumann se situe au niveau du transfert des instructions et des données depuis les mémoires plus lentes vers les registres. Les machines modernes intègrent à l'intérieur des processeurs plusieurs niveaux de *mémoires caches* qui possèdent des caractéristiques intermédiaires entre les registres et la mémoire vive.

Une extension du modèle de von Neumann très couramment utilisée est le « *pipelining* ». Dans un « *pipeline* », les données avancent les unes derrière les autres, au rythme du signal d'horloge : l'exécution des instructions est découpée en étages afin de produire en théorie un résultat à chaque cycle. Le premier ordinateur à utiliser cette technique est l'IBM Stretch en 1958. Le principal défaut est qu'il est difficile de maintenir rempli un pipeline long avec un programme possédant de nombreux sauts conditionnels. En effet, lors d'un saut conditionnel il faut parfois vider entièrement le pipeline en cas de mauvaise prédiction de branchement et perdre ainsi le temps que l'on pensait économiser par anticipation. En théorie il est possible d'effectuer en parallèle un nombre d'opérations correspondant à la profondeur du pipeline. Pour certaines applications l'amélioration est très sensible. L'Intel Pentium 4® possède un pipeline d'une profondeur de 20 opérations.

Une deuxième amélioration peut être apportée aux machines, les instructions vectorielles : il s'agit là aussi d'un ancêtre du parallélisme. Sur de tels processeurs, une instruction va s'appliquer à un vecteur de données. Une seule instruction va exécuter la même opération de façon parallèle sur tout le vecteur. Ces processeurs sont efficaces pour du calcul scientifique par exemple, où un même traitement est appliqué à de nombreuses données, on parle d'applications *data-intensive*. Les premiers travaux datent des années 60, le Cray-1 de 1976 est un exemple célèbre de super-calculateur à architecture vectorielle. AMD® en 1998 avec 3DNow ! et Intel® avec MMX en 1997 et SSE en 1999 introduisaient des jeux d'instructions vectorielles à leurs processeurs. Aujourd'hui, certaines cartes graphiques accélératrices 3D sont programmables pour autres choses que la 3D et sont de formidables machines vectorielles. La carte graphique Nvidia Tesla®, moins d'1Kg, consommation 238W, dédiée au calcul haute performance annonce 515 Gflops (515 milliards d'opérations à virgule flottante double précision par seconde). C'est 2000 fois plus que le Cray-1 de 5,5 tonnes et 200kW de consommation (refroidissement compris) et 2 fois plus que le processeur Xeon X7560 à 8 cœurs qui consomme lui 130W et reste plus généraliste.

Une dernière amélioration, datant des années 1950 est le *multi-threading simultané* ou SMT. Cette technique consiste à partager le pipeline d'exécution d'un processeur entre plusieurs threads, pas forcément du même programme. Dans les processeurs Intel®, cette technique s'appelle l'Hyperthreading et permet d'exécuter 2 threads simultanément, on parle de SMT à 2 voies.

4.4.2 Taxonomie de Flynn, architectures mémoires et réseaux

En 1966, Michael Flynn classe les systèmes en fonction du nombre de flux d'instructions et du nombre de flux de données. Il identifie ainsi 4 architectures, schématisées dans

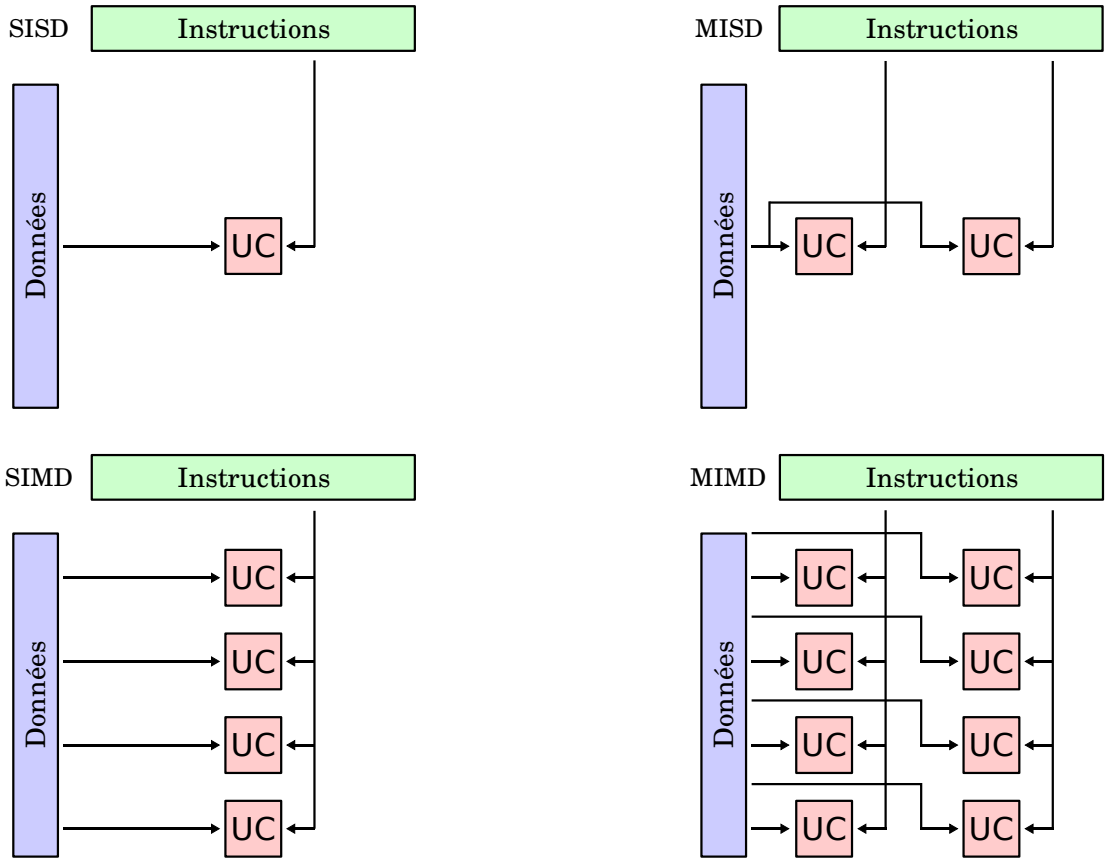


FIGURE 10 – Taxonomie de Flynn de 1966.

la figure 10 [1] :

- **Single-Instruction Single-Data (`SISD`).** C'est un ordinateur séquentiel qui n'exploite le parallélisme ni au niveau des instructions, ni au niveau des flux données. Il s'agit par exemple de la machine classique de von Neumann ou de très anciens mainframes ;
- **Multiple-Instruction Single-Data (`MISD`).** C'est un ordinateur capable d'exploiter un même flux de données à l'aide de plusieurs instructions simultanément. Cette architecture est assez peu commune et est généralement utilisée pour obtenir de la tolérance aux erreurs ou aux pannes, par exemple dans les ordinateurs de vol des navettes spatiales. Les architectures avec pipelines appartiennent à cette catégorie, bien qu'entre deux étapes les données aient changé ;
- **Single-Instruction Multiple-Data (`SIMD`).** C'est un ordinateur capable d'exécuter les mêmes instructions en parallèle, de manière synchrone, sur plusieurs flux de données. Cette architecture convient aux opérations qui se parallélisent assez na-

1. Illustration reprise de `http://en.wikipedia.org/wiki/Flynn's_taxonomy`, puis traduite.

turellement. Comme exemple, il y a bien entendu les processeurs vectoriels et les cartes graphiques accélératrices ;

– **Multiple-Instruction Multiple-Data (`MIMD`).** Ce sont des ordinateurs où de multiples unités de calcul peuvent simultanément, de manière asynchrone, exécuter des instructions différentes sur des données différentes. Cette catégorie regroupe aussi bien les ordinateurs avec mémoire partagée (chaque processeur accède à un espace public de mémoire) ou distribuée (chaque processeur possède son espace privé de mémoire). Nos ordinateurs personnels multi-cœurs sont de type `MIMD` avec mémoire partagée (la mémoire vive). Le super-calculateur du LERIA (décrit en détail dans la section 5.5) est lui de type `MIMD` à mémoire distribuée.

Nous allons nous intéresser un peu plus aux systèmes `MIMD`. Tout d'abord nous revenons sur les architectures mémoires de ces systèmes. En 1988, Eric E. Johnson donne une classification [Johnson, 1988] des systèmes `MIMD` en fonction de leur manière d'accéder à la mémoire. La cohérence de cache est requise dès lors que la mémoire est partagée et comme il est préférable pour le programmeur de ne pas s'en occuper, cette tâche incombe à l'architecture. Cette cohérence a un coût et est un facteur limitant pour des architectures de grande taille. Il ressort essentiellement 3 architectures viables et utilisées :

– **Cache Coherent Uniform Memory Access (`CC-UMA`).** Les processeurs et la mémoire sont interconnectés par un bus, chaque processeur accède à la mémoire à la même vitesse. Si beaucoup de processeurs veulent accéder simultanément à la mémoire, un problème de saturation peut survenir. Ces architectures sont plutôt limitées par la bande passante du bus et ne sont pas très extensibles. C'est le cas par exemple d'un ordinateur avec un ou plusieurs processeurs, multi-cœurs ou non, sur la même carte mère avec un unique bus mémoire. Au maximum quelques dizaines de processeurs peuvent être envisagés ;

– **Cache Coherent Non Uniform Memory Access (`CC-NUMA`).** Il est possible de résoudre le problème de bande passante en disposant de plusieurs modules de mémoires reliés aux processeurs par un réseau de communication de type commutateur (switch, crossbar switch,...). Les processeurs n'ont alors plus accès à la mémoire en temps uniforme. Parfois, les processeurs possèdent de la mémoire locale à laquelle l'accès est plus rapide. Mais toute la mémoire du système est adressable directement à vitesse plus réduite. Ce type d'infrastructure est très onéreuse quand le nombre de commutateurs nécessaires augmente et cette architecture souffre d'autant plus du coût provoqué par le maintien de la cohérence des caches ;

– **No Remote Memory Access (`NoRMA`).** Ici, plus de problème de cohérence de cache, chaque processeur possède son espace privé de mémoire. Les couples processeurs / mémoires sont reliés entre eux par un réseau de communication non nécessairement uniforme ni même statique. Le réseau le plus performant est bien évidement une interconnexion complète, mais c'est aussi la plus onéreuse des solutions. Le super-calculateur du LERIA possède suffisamment peu de nœuds pour être complètement interconnecté à l'aide d'un commutateur Ethernet. Pour des raisons de coût, il est possible d'imaginer de nombreuses topologies réseaux : linéaires, anneaux, hypercube, étoile, etc. Le réseau de calcul distribué BOINC (Berkeley Open Infrastructure for Network Computing) est un architecture `NoRMA` où chaque unité

de calcul communique par le réseau internet. BOINC est une infrastructure appelée grille de calcul ou *grid*, qui héberge plusieurs applications et répond à d'autres exigences. En France, le projet national Grid5000 vise à construire une plate-forme expérimentale de recherche en informatique, constituée d'une grille de calcul de grande taille (5000 processeurs). Le coût est alors partagé entre les différents acteurs.

La plupart de nos ordinateurs sont des systèmes de type `MIMD CC-UMA`, c'est aussi le cas des éléments d'une grille de calcul ou d'un super-calculateur « bon marché » (tout est relatif). Ces éléments sont agencés dans une architecture de type `NoRMA` avec des réseaux plus ou moins rapides qui vont d'internet, potentiellement lent avec de fortes latences et peu connexe, à de l'infiniBand®, complètement interconnecté. Pour de très puissantes machines requérant une grande quantité de mémoire adressable, il est possible de recourir à une architecture de type `CC-NUMA`. Celles-ci peuvent ensuite être interconnectés dans une architecture `NoRMA` avec un réseau de communication très performant. L'architecture `NoRMA` est inévitable pour les très grosses architectures où la cohérence de cache n'est plus envisageable.

4.5 Les solutions de programmation

En programmation parallèle, l'élément fondamental est le *processus*, ce qui inclut les processus légers ou « *threads* ». Grossièrement, un processus est une instance d'un programme ou un sous-programme qui s'exécute de façon plus ou moins autonome sur un processeur. Un programme peut être considéré comme parallèle dès lors qu'il est composé de plusieurs processus. Il existe plusieurs solutions de programmation pour spécifier, créer, coordonner et détruire des processus. Nous faisons un bref tour d'horizon des solutions les plus courantes avant de s'attarder sur le paradigme de la programmation par passage de messages.

4.5.1 Les différentes approches parallèles

La programmation avec mémoire partagée. Intuitivement, chacun aurait tendance à croire que ce paradigme de programmation ne s'applique qu'au système à mémoire partagée. Mais il est tout à fait envisageable *d'émuler* de la mémoire partagée sur un système à mémoire distribuée. Pour cela, il suffit de fournir une façon pour réaliser l'adressage global du système. L'utilisateur peut ensuite faire usage des primitives de programmation pour mémoire partagée sur l'architecture mémoire `NoRMA` et cela de manière transparente. Aussi ce paradigme de programmation ne se limite pas aux architectures `CC-UMA` ou `CC-NUMA`, bien qu'elles soient plus propices à de bonnes performances.
Un processus peut être statiquement créé au début de l'exécution ou dynamiquement durant celle-ci. Une directive très connue de création dynamique de processus est le `fork` ; le processus *parent* peut alors attendre le processus enfant à l'aide de la directive `join`. D'une manière générale, en programmation avec mémoire partagée, les processus sont

gérés par trois primitives. La première permet de spécifier quelles données sont accessibles par tous les processus. La seconde permet d'empêcher les accès impropres à ces données partagées. Certaines séquences d'instructions doivent être arrangées pour être en exclusion mutuelle (à l'aide d'un sémaphore par exemple), dans ce qui est appelé une *section critique* tout en évitant les interblocages ou « *deadlock* ». La troisième et dernière primitive courante est une directive de synchronisation des processus, comme la directive `barrier`.
Les *threads POSIX* sont un bon exemple de technologie pour la programmation avec mémoire partagée. Il s'agit d'un standard supporté sur l'ensemble des systèmes UNIX et dérivés. Les créations, synchronisations et sections critiques sont effectuées explicitement par l'utilisateur. Plus récemment, l'interface *OpenMP*, compatible avec les systèmes UNIX et Windows, offre un ensemble de directives permettant de paralléliser du code séquentiel à l'aide de directives à destination du pré-compilateur. Le code source reste ainsi très proche du code séquentiel et l'utilisateur n'a pas à gérer les processus.

L'appel de procédures distantes (RPC). Cette technologie permet à un programme de demander l'exécution d'une procédure ou d'une fonction dans un autre espace d'adressage. Le plus souvent, il s'agit d'une autre machine d'un réseau privé. Cet appel distant se fait sans que le programmeur ait besoin explicitement de coder les détails de l'interaction à distance : pour le programmeur le code est identique qu'il soit exécuter localement ou à distance. L'idée est que finalement, la communication entre des processus est plus générale que la simple transmission de données : il est possible de communiquer un appel de procédure ou de fonction. Ce protocole utilise un modèle client/serveur et permet de gérer les messages entre les ordinateurs afin de surmonter les problèmes liés à l'aspect synchrone de RPC à l'origine. Cette technologie est particulièrement bien adaptée aux architectures à mémoire distribuée, mais fonctionne aussi sur des systèmes à mémoire partagée. Nous pouvons citer par exemple son utilisation pour la conception de micro-noyaux et le tristement célèbre ver informatique *Blaster* qui utilisait une faille dans le service *DCOM RPC* de Windows 2000 et XP pour se propager.

Les langages pour données parallèles. Il s'agit probablement de l'approche la plus simple et la plus adaptée aux machines avec une architecture `SIMD` et elle est très commune pour les systèmes `MIMD`. C'est d'ailleurs l'idée qui se cache derrière l'interface *OpenMP*. Pour des structures très régulières il est possible pour le programmeur de juste spécifier que le calcul peut être distribué sur plusieurs processus. Le compilateur se chargera alors de remplacer automatiquement le code par des directives permettant d'effectuer les opérations sur les données en parallèle. Ces structures régulières leurs valent le qualificatif de données parallèles, au sens que les traitements à effectuer dessus sont parallélisables naturellement. Ce n'est pas le cas de SAT, QBF et de beaucoup d'autres problèmes. Ce paradigme s'adapte très bien à tout type d'architecture mémoire, cependant pour les architectures `NoRMA` la problématique de mise en cohérence de données ou *data mapping* est fondamentale pour avoir une bonne répartition de charge.

Calcul générique sur un processeur graphique (GPGPU). Les processeurs graphiques ou *GPU* sont devenus de plus en plus programmables à tel point qu'ils peuvent faire les calculs qui autrefois nécessitaient l'utilisation d'un processeur généraliste (*CPU*). Cela est devenu possible par l'ajout d'étages programmables, les *pixel et vertex shader* (programmes qui tournent sur le GPU, originalement pour le rendu d'une image) et par une plus grande précision arithmétique dans les pipelines de rendu. Ces mécanismes ont d'abord été détournés pour effectuer des traitements sur des données non graphiques, puis différent acteurs ont fourni des interfaces de programmation pour GPU : `OpenCL`, `CUDA` et `DirectCompute`. Finalement, les constructeurs proposent des GPU dimensionnés pour le calcul. Ce type de programmation fonctionne bien avec des données parallèles. `OpenCL` est plus généraliste, au sens que l'objectif est de solliciter les capacités à la fois des GPU et des CPU dans un cadre de programmation adapté aux différentes architectures.

La programmation par passage de messages. C'est le paradigme de programmation le plus répandu pour les systèmes `MIMD NoRMA` mais il s'adapte aux autres architectures. De manière basique, les processus se coordonnent en envoyant et recevant explicitement des messages. MPI (« *Message Passing Interface*) [Snir *et al.*, 1995] en est l'interface la plus commune et sera détaillée dans la section suivante, PVM (« *Parallel Virtual Machine* ») est une autre bibliothèque connue fonctionnant par passage de messages. Puisque les éléments d'une architecture de calcul sont souvent de type `MIMD CC-UMA`, il est possible de mélanger la programmation en mémoire partagée avec la programmation par passage de messages. Cependant, la gestion conjointe de MPI et des threads n'est pas toujours très stable et pose parfois des problèmes.

4.5.2 MPI : Message Passing Interface

Cette description s'inspire du livre « *Parallel Programming with MPI* » [Pacheco, 1997] qui nous a servi de base pour découvrir cette approche. MPI [Snir *et al.*, 1995] est un protocole de communication indépendant d'un langage de programmation. Il spécifie un ensemble de directives pour faire communiquer point à point ou de façon collective des processus. Les objectifs de MPI sont les hautes performances, l'évolutivité et la portabilité. Aujourd'hui, c'est le modèle dominant dans le domaine du HPC et plus particulièrement pour les systèmes à mémoire distribuée. Il existe deux versions majeures de MPI. La première version `MPI-1` (en version 1.2), date de 1993 et possède aujourd'hui 128 fonctions. La deuxième version `MPI-2` (en version 2.1) est un surensemble de `MPI-1` auquel sont ajoutés de nouveaux concepts, portant à plus de 500 le nombre de fonctions. Les programmes respectant le standard `MPI-1` sont donc compatibles avec les implantations de `MPI-2`, bien que certaines fonctions soient dépréciées. Un programme MPI travaille avec des processus, mais typiquement, pour maximiser les performances, chaque processus est assigné à une unité de calcul. L'affectation est réalisée au lancement par un agent, la commande `mpirun` sur notre système, qui initie le programme MPI. Cet agent est aussi chargé de proposer une topologie virtuelle de départ afin de rendre plus

Type MPI	Type C
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

TABLE 10 – Les types prédéfinis dans MPI.

efficaces les communications pour l'architecture sur laquelle le programme est exécuté. Il est ensuite possible de créer nos propres topologies. Nous détaillons ci-dessous un peu plus les concepts fondamentaux de MPI-1 :

- **les communicateurs et topologies.** Un *communicateur* est un ensemble de processus pouvant communiquer ensemble et les processus d'un communicateur sont ordonnés dans une *topologie*. Au départ, il existe un unique communicateur contenant tous les processus, MPI_COMM_WORLD. Sa topologie a été définie par l'agent de démarrage (mpirun par exemple) afin d'optimiser les communications collectives pour un programme quelconque. Les communicateurs et les topologies permettent à un programmeur d'adapter l'architecture virtuelle des processus pour prendre en compte les spécificités d'un problème et l'architecture de calcul utilisée ;

- **les communications point à point.** Ce type de communication permet, comme son nom l'indique, d'échanger des messages entre 2 processus. Dans un modèle client/serveur c'est le mode de communication dominant. Cela peut se réaliser de manière bloquante (synchrone) : les 2 processus attendent d'être disponibles ensemble avant de communiquer ; ou non bloquante (asynchrone) : un processus envoie un message et l'autre vérifie s'il en a reçu ;

- **les communications collectives.** Ce genre de communications implique tous les processus d'un communicateur, c'est pour cela que la topologie virtuelle d'un communicateur est importante pour les performances. On retrouve des instructions très simples comme MPI_Bcast, qui envoie un message identique à tous les processus. D'autres sont plus complexes comme MPI_Reduce, qui permet d'effectuer directement des opérations pour un groupe de processus. Par exemple, si MPI_Reduce

prend en paramètre l'opération `MPI_MAX`, alors le processus qui a initié la réduction recevra la valeur maximum du calcul réalisé en parallèle sur ce communicateur. Il est alors simple de chercher le maximum d'un très grand tableau en parallèle. Plus d'une dizaine d'opérations de réduction sont prédéfinies. Il existe aussi des instructions de synchronisation comme `MPI_Barrier` qui ne va pas sans rappeler la primitive `barrier` des threads POSIX. Une faible utilisation des communications collectives rend limité l'intérêt des communicateurs et des topologies ;

– **les types prédéfinis et les type dérivés.** Il existe 13 types prédéfinis dans MPI qui peuvent être communiqués, ils sont listés dans la table 10 tirée de [Pacheco, 1997]. Le type `MPI_PACKED` permet de définir des types dérivés composés des types prédéfinis. Toutes les données de ces types peuvent être envoyées seules ou groupées en tableaux. Il est très fastidieux de créer des `MPI_PACKED` et l'envoi de structure avec pointeurs est un vrai casse-tête. La bibliothèque C++ Boost.MPI est une interface faisant appel à une implantation de MPI et répond entre autre à cette contrainte à l'aide de Boost.Serialization. Un objet d'une classe peut être envoyé et reçu de façon simple si la classe est sérialisable, c'est à dire si elle implante la méthode `serialize` requise par Boost.Serialization. Il est alors possible d'envoyer des objets ou une structure d'objets de façon transparente si le surcoût en temps de calcul de l'opération de sérialisation est négligé.

Brièvement, `MPI-2` apporte trois nouveaux concepts. Premièrement, les fonctions sur les entrées/sorties en parallèle permettent de rendre abstraite la gestion des entrées/sorties sur les systèmes distribués. Deuxièmement, les fonctions pour la gestion dynamique des processus ont pour objectif de pouvoir créer dynamiquement des processus ou de faire participer des processus MPI qui ont été démarrés séparément. Et pour finir, des fonctions permettent de lire, écrire et accumuler des données distantes. Les principales implantations en C pour MPI sont `MPICH` et `MPICH 2`, `LAM/MPI`, `Open MPI` et des versions propriétaires comme celles d'Intel, HP ou Microsoft.

4.6 Décider de la validité des QBF en parallèle

Alors que les premiers résultats pratiques pour QBF étaient présentés en 1997 par Cadoli, Giovanardi et Schaerf [Cadoli *et al.*, 1997; Cadoli *et al.*, 1998], la première procédure parallèle a été présentée trois ans plus tard par Feldmann, Monien et Schamberger [Feldmann *et al.*, 2000]. Il n'existait alors que trois procédures séquentielles : `QKN` [Büning *et al.*, 1995], `Evaluate` [Cadoli *et al.*, 1997] et `decide` [Rintanen, 1999b]. Pour plus de détails le lecteur peut se référer au chapitre 2. Il faudra attendre presque dix années avant de voir de nouveaux travaux sur les procédures de décision QBF en parallèle. Autant que nous le sachions, il existe trois implantations de solveurs parallèles pour le problème de validité des QBF. Par ordre d'apparition : `PQSOLVE` [Feldmann *et al.*, 2000], `QMiraXT` [Lewis *et al.*, 2009b] et `PaQube` [Lewis *et al.*, 2009a]. Elles sont toutes dédiées aux QBF sous CNF prénexes et sont toutes basées sur un algorithme de recherche séquentiel par élimi-

nation du quantificateur le plus externe vers le plus interne, c'est à dire une extension de l'algorithme QDLL. Elles appliquent une *stratégie de partitionnement sémantique* qui choisit une variable booléenne x du bloc de quantificateurs le plus externe et applique la sémantique des quantificateurs pour partitionner le problème et distribuer les tâches : si la QBF dont on désire décider de la validité est équivalente à $qxQM$ alors les deux tâches $Q[x \leftarrow \top](M)$ et $Q[x \leftarrow \bot](M)$ sont distribuées. De par leur modèle, `PQSOLVE` et `PaQube` sont plus extensibles aux clusters et grids que `QMiraXT` mais ce dernier solveur semble tenir plus compte de l'évolution du matériel que les deux premiers. Nous allons maintenant détailler les différentes procédures.

4.6.1 PQSOLVE

Les auteurs de `PQSOLVE` commencent par présenter une procédure de décision séquentielle pour les QBF, `QSOLVE` [Feldmann *et al.*, 2000]. Cette procédure reprend l'algorithme QDLL ainsi que les améliorations présentes à l'époque : *inversion de quantificateurs* et *vérité/fausseté triviale*. Ils améliorent ce dernier concept en limitant le nombre d'applications à l'aide d'une heuristique. La procédure séquentielle présentée surpasse les procédures de l'état de l'art. La procédure `PQSOLVE` en est la version distribuée qui utilise les techniques de la parallélisation des programmes d'échecs. Il s'agit d'un modèle pair-à-pair afin de répartir les communications et la charge : un processus inactif demande du travail à un processus choisi au hasard et en devient l'esclave pour une tâche. Chaque processus a une pile de tâches à réaliser qui est augmentée par le partitionnement sémantique de celles qui sont estimées trop complexes ; le maître envoie une de ces tâches à son esclave momentané qui a sollicité une tâche à réaliser. Un esclave peut devenir lui-même le maître d'un autre processus. La communication est réalisée par MPI. Les tests sont effectués sur deux fermes de calculs différentes, pouvant contenir jusqu'à 128 processeurs. Les résultats sont très encourageants : l'algorithme parallèle effectue moins de travail que la version séquentielle. Avec la ferme de calcul possédant un réseau de communication propriétaire (CluStar network®), l'accélération est même super linéaire pour de nombreux exemples. Cependant, les améliorations apportées aux procédures séquentielles les années qui suivent permettent de surpasser `QSOLVE` et sa version parallèle. Des techniques telles que le retour arrière intelligent et l'apprentissage font faire un grand bond en avant aux procédures séquentielles, mais il n'est pas aisé d'intégrer ces techniques dans des procédures parallèles.

4.6.2 QMiraXT

La procédure `QMiraXT` [Lewis *et al.*, 2009b] est dédiée à la prise en compte du potentiel de performance des architectures modernes multi-cœur et/ou processeurs multithreadés. En utilisant un solveur threadé à mémoire partagée, les clauses apprises par conflit [Zhang et Malik, 2002a] sont partagées en les différents espaces de recherche. Il n'y a pas de processus maître mais à la place un « *Master Control Object* » (MCO) qui permet aux threads de communiquer via des messages asynchrones pour des évènements

globaux (par exemple, si un sous-problème est valide ou non). Le MCO prend en charge aussi la stratégie de partitionnement sémantique pour garantir que tout l'espace de recherche a été parcouru et distribue les tâches. Afin de simplifier la génération de sous-problèmes et facilité la vérification de la complétude de la recherche, l'algorithme « *Single Quantification Level Scheduling* » (ou SQLS) est introduit. L'idée derrière l'algorithme SQLS est de limiter le choix des variables pour le partitionnement à un seul niveau de quantification. La tâche pour le MCO est alors facilitée : plus besoin de mémoriser le processus qui a généré la tâche ni même l'ordre de génération. Le partionnement passe au niveau inférieur quand tous les processus attendent des sous-problèmes. Cette phase parallèle est précédée d'une phase séquentielle de pré-traitement. Dans un premier temps, les variables inutilisées ou présentes sous une seule polarité sont supprimées. Puis, est appliqué un algorithme *d'hyper résolution binaire* afin de trouver les variables équivalentes afin de les éliminer. Les auteurs précisent que la plupart des solveurs font cela car de nombreux problèmes sont codés de façon non efficace. Finalement, `quantor` est utilisé comme pré-traitement pour réduire le nombre de variables quantifiées universellement. Puisque `quantor` peut être exponentiel en espace et en temps, des limites sont fixées : 5 secondes et 128Mo de mémoire. Ce n'est qu'après cette phase que la procédure utilise le parallélisme. De cette manière, l'approche résolution et expansion *bottom-up* de `quantor` est mixée avec un algorithme QDLL *top-down* parallèle.

Les résultats expérimentaux montrent de très bonnes performances pour `QMiraXT` même avec un seul processus. Avec 4 processus, soit le nombre de cœurs dans nos machines, les temps sont divisés par plus de 4 par rapport à `sKizzo`, `quantor` et `Quaffle` et 22 nouveaux problèmes sont résolus. La comparaison avec `quantor` est intéressante puisque certaines familles de problèmes sont résolus en moins de 5 secondes : lorsque `quantor` est efficace, le pré-traitement de `QMiraXT` suffit à résoudre les problèmes, sinon la recherche en parallèle prend le relais. Le solveur `QMiraXT` semble prendre bien en compte l'évolution des machines en proposant une procédure performante bénéficiant de l'apprentissage partagé sur le bus système, très rapide et avec de faibles temps de latence et non sur un réseau de communication tel Ethernet. Contrairement à `PQSOLVE`, `QMiraXT` ne s'oriente pas vers du calcul massivement parallèle.

4.6.3 PaQube

Les auteurs de `QuBE` et de `QMiraXT` ont joint leurs efforts pour réaliser une version parallèle de `QuBE` tout en profitant des avancées de `QMiraXT` sur le parallélisme [Lewis *et al.*, 2009a]. La procédure `PaQube` [Lewis *et al.*, 2009a] est conçue selon le modèle maître/esclave où un processus est dédié au maître (ce qui ne nécessite pas de CPU dédié) et les autres aux esclaves qui réalisent en fait la recherche. `PaQube` est un solveur QBF parallèle basé sur MPI. Au travers de messages, les esclaves partagent certaines des clauses et cubes issus respectivement des conflits et solutions appris. Ainsi, chaque esclave cumule localement toute l'expertise obtenue par l'ensemble des autres esclaves dans une certaine limite d'espace. La phase de recherche en parallèle est précédée d'un pré-traitement séquentiel par `sQueezeBF` Enfin, le travail principal du maître

est de réaliser la complétude de la recherche, comme pour `QMiraXT`, avec l'algorithme SQLS. L'attention est portée sur le partage des connaissances collectées sur chaque esclave par apprentissage de clauses et de terme. Il faut limiter les échanges sous peine de saturer les canaux de communications. Les réglages ont été déterminés expérimentalement afin d'obtenir les meilleures performances possibles pour la machine de test et les problèmes. La machine de test possède 4 processeurs qui contient chacun 4 cœurs, les résultats sont donnée pour `PaQube` jusqu'à 16 processus. La procédure `QuBE` ayant obtenu les meilleurs résultats lors de la dernière compétition *QBFEVAL*, elle sert d'unique point de comparaison avec `PaQube`. Il est dommage de ne pas voir figurer les résultats de `QMiraXT` surtout que `QuBE` souffre d'une contention du bus mémoire au delà de 4 processus. Il est difficile à la vue des résultats d'imaginer `PaQube` s'exécuter sur une ferme de calcul en partageant les connaissances à travers un réseau de communication qui sera forcément plus lent que le bus mémoire et qui aura plus de temps de latence. Il est difficile de juger de l'accélération de `PaQube` pour différents nombres de processus car les temps présentés contiennent les problèmes pour lesquels la limite de temps a été atteinte. Avec 16 processus, `PaQube` résout 47 problèmes de plus que `QuBE` et 13 problèmes non résolus lors de la dernière évaluation. Outre les performances, l'intérêt de `PaQube` est surtout situé au niveau de l'étude expérimentale réalisée autour de l'apprentissage en partageant les connaissances par passage de messages (avec MPI). Il faut pouvoir partager des informations pour accélérer la recherche, encore faut-il recevoir l'information avant de ne plus en avoir besoin. La procédure `PaQube` est plus extensible aux clusters que `QMiraXT` grâce à l'utilisation de MPI.

Chapitre 5

Une nouvelle architecture parallèle pour manipuler les QBF

[Da Mota *et al.*, 2010b] Benoit Da Mota, Pascal Nicolas, and Igor Stéphan. Une nouvelle architecture parallèle pour le problème de validité des QBF. In *Sixièmes Journées Francophones de Programmation par Contraintes, JFPC'10*, 2010.

[Da Mota *et al.*, 2010a] Benoit Da Mota, Pascal Nicolas, and Igor Stéphan. A new Parallel Architecture for QBF Tools. In *Proceedings of the 2010 International Conference on High Performance Computing & Simulation, HPCS 2010*, pages 324–330. IEEE, 2010.

Sommaire

5.1 Introduction

La plupart des procédures récentes et efficaces pour décider de la validité des QBF, nécessitent d'avoir en entrée une formule sous forme normale négative ou dans une forme encore plus restrictive comme la forme normale conjonctive. Mais il est rare que les problèmes s'expriment directement sous ces formes qui détruisent complètement la structure originale des problèmes. Il est plus naturel d'utiliser toute l'expressivité du langage QBF : tous les connecteurs logiques usuels, y compris l'implication, la bi-implication et le ou-exclusif, ainsi que la possibilité d'utiliser des quantificateurs à l'intérieur de la formule (cf. chapitre 3). Aussi, notre premier objectif est de développer un solveur QBF sans restriction sur les formules en entrée.

Dans le chapitre 4 nous avons vu que depuis quelques années, la fréquence du ou des cœurs des processeurs ne progresse plus, voire même diminue. En contrepartie, le nombre de cœurs par processeur augmente et le domaine de la programmation parallèle est en pleine effervescence : multithread, multi-CPU, GPU computing, HPC, grid computing, cloud computing, etc. Certaines procédures de résolution profitent déjà de ce nouveau potentiel et les futurs solveurs devront en faire de même pour rester compétitifs. Le second objectif de notre travail est d'exploiter les opportunités qu'offre la programmation parallèle pour s'attaquer au problème de validité des QBF en réutilisant, adaptant, améliorant ou combinant des algorithmes séquentiels pour QBF à l'intérieur d'une architecture parallèle.

Par ailleurs, au delà du problème de validité, nous nous intéressons aussi à d'autres problèmes en rapport avec QBF, tels celui de leur compilation pour une représentation plus compacte et un accès de complexité moindre aux solutions. Dans ce cas, une procédure de décision ne suffit plus (la réponse du système n'étant plus simplement « oui » ou « non »), mais un ensemble de formules décrivant l'espace des solutions pour les symboles propositionnels existentiellement quantifiés. Aussi dans ce but, notre objectif à long terme est de réaliser une architecture aussi ouverte que possible pour offrir à l'utilisateur final un ensemble d'outils en rapport avec les QBF.

5.2 Représentation des informations

Dans cette section, nous allons expliquer comment nous remplissons notre premier objectif : réaliser une procédure sans restriction syntaxique. Dans un premier temps, nous présentons un nouveau format pour écrire des formules booléennes quantifiées quelconques. Puis dans un second temps, nous exposons la structure de données que nous avons élaborée afin de stocker et de manipuler ces formules. Celle-ci ne doit être en aucun cas liée au format d'entrée et doit permettre de remplir les objectifs que nous lui imposerons. Finalement, nous présentons de quelle façon des traitements sont appliqués sur cette structure.

5.2.1 Un format plus expressif pour les QBF.

Afin de s'acquitter des problèmes liés à la mise sous forme prénexe et à la mise sous forme normale conjonctive, notre procédure devra travailler sur des QBF non prénexes non CNF. De plus nous voulons pouvoir traiter les implications, les bi-implications et les ou-exclusifs. Mais il n'existe pas de format qui permette une telle description des QBF. Par conséquence, il n'existe pas non plus de problème écrit avec ce niveau d'expressivité. Alors que dans le chapitre 3 nous illustrions comment minimiser la perte d'informations lors la suppression des bi-implications, ici nous ne voulons plus avoir à effectuer ces suppressions. La première étape consiste à proposer un « nouveau » format pour que des personnes puissent écrire et partager de telles formules sans avoir à les transformer.

Le format QBF1.0 (décrit à cette adresse : `http://www.qbflib.org/boole.html`) permet d'écrire des formules booléennes quantifiées quelconques, mais limite les opérateurs logiques à la conjonction et à la disjonction. Nous l'avons étendu avec le symbole -> pour l'implication, le symbole <-> pour la bi-implication et le symbole # pour le ou-exclusif. Nous donnons la grammaire correspondante, sous la forme de Backus-Naur :

```
<input>    ::= <exp> EOF
<exp>      ::= <NOT> <exp> | <q_set> <exp> |
               <LP> <exp> <op> <exp> <RP>  |
               <LP> <exp> <RP> | <VAR>
<q_set>    ::= <quant> <LSP> <var_list> <RSP> |
               <quant>  "{"  <var_list>  "}"
<quant>    ::= <EXISTS> | <FORALL>
<var_list> ::= <VAR> <var_list> | <VAR>
<op>       ::= <OR> | <AND> | <IMPL> | <EQU> | <XOR>
<NOT>      ::= "!"
<LP>       ::= "("
<RP>       ::= ")"
<LSP>      ::= "["
<RSP>      ::= "]"
<OR>       ::= "|"
<AND>      ::= "&"
<IMPL>     ::= "->"
<EQU>      ::= "<->"
<XOR>      ::= "#"
<EXISTS>   ::= "exists"
<FORALL>   ::= "forall"
<VAR>      ::= {A sequence of non-special ASCII characters}
```

Soit la QBF $\exists a \forall b (a \leftrightarrow (b \oplus (\exists c (a \wedge \neg c))))$. Elle s'écrit dans notre format :

```
exists{a}(forall{b}( a <-> (b # (exists{c}(a & ! c)))))
```

Ce nouveau format, que nous baptisons QBF1.1 assure une compatibilité ascendante : une QBF dans le format QBF1.0 est compatible avec le format QBF1.1. Ce choix nous

permet de pouvoir tester et résoudre dans notre architecture tous les problèmes décrits dans le format QBF1.0, au même titre que les problèmes que nous proposons au format QBF1.1.

5.2.2 Un modèle inspiré des patrons de conception.

En conception orientée objet, il n'est pas toujours facile de trouver un découpage en objets pertinents, qui par leur organisation et leurs méthodes devront répondre aux objectifs du modèle. Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides, surnommés le « *Gang of Four* », ont isolé des problèmes de conception récurrents. Les *patrons de conception* (ou « *Design Patterns* ») sont des solutions à ces problèmes courants. Leur livre « *Design Patterns : Elements of Reusable Object-Oriented Software* » [Gamma *et al.*, 1994] est un catalogue de patrons de conceptions. Ils sont divisés en trois familles selon leur utilisation :

- les patrons de construction traitent l'instanciation des objets ;
- les patrons structuraux définissent des manières pour organiser les classes d'un programme dans une structure plus générale ;
- les patrons comportementaux définissent comment organiser les objets pour qu'ils fonctionnent et interagissent ensemble.

L'objectif principal de notre structure de données est de représenter le plus fidèlement possible le langage **QBF** décrit dans la section 1.2.1, afin de préserver l'information présente dans la formulation d'origine. Puis il faut pouvoir appliquer des traitements de façon uniforme aux différents objets qui représenteront la formule. Notre structure de donnée n'a pas pour objectif d'être performante ou économe en espace. Nous cherchons un maximum d'expressivité et d'extensibilité. Toutefois, dans l'optique d'une architecture parallèle, l'aspect *taille en mémoire* doit être considéré pour ne pas ralentir l'ensemble si les transferts sont fréquents. Les deux sous-sections suivantes détaillent le choix et l'utilisation de deux patrons de conception : un patron structurel (composite) et un patron comportemental (visiteur).

Une structure composite pour représenter les QBF.

A partir du langage **QBF**, il est assez naturel de représenter les éléments d'une formule sous forme d'un arbre. Certains composants, les nœuds, contiennent d'autres éléments de formule et d'autres sont des feuilles terminales. Dans notre architecture, quand nous devons interagir avec une formule, nous voulons avoir l'impression de nous adresser à un unique objet. Par exemple, pour afficher une formule, nous ne voulons pas nous adresser aux différents composants. La façon dont est représentée la formule n'intéresse pas le client et si la représentation s'enrichit cela doit rester transparent pour l'utilisateur. Pour continuer l'exemple, si nous rajoutons un type de nœud ou de feuille, la formule doit s'afficher sans que le client se rende compte d'un changement structurel. Le patron de conception structurel *composite* répond à cette attente. En effet, il est conseillé de l'utiliser dès lors qu'un client doit être capable d'ignorer la différence entre un objet individuel

et la composition d'objets. Ce patron de conception a comme conséquence directe de faciliter l'ajout de nouveaux types de composants. Ceux-ci fonctionnent directement dans la structure existante sans changer le code du client.

Une QBF est représentée par un ensemble d'éléments abstraits de formule (figure 20 en annexe p. 120), liés les uns aux autres sous forme d'un arbre. Un élément abstrait de formule impose uniquement 2 actions aux éléments concrets : savoir démarrer un visiteur abstrait et savoir se sérialiser/désérialiser. Nous reviendrons dans la section suivante sur la notion de *visiteurs* qui permet d'externaliser les traitements. La *sérialisation* est un processus qui consiste à encoder les informations en mémoire sous forme d'un fichier ou d'un flux. La *désérialisation* est le processus inverse, qui permet de restaurer les informations en mémoire. Cette fonctionnalité nous permet, par exemple, de passer de façon transparente une formule à travers le réseau. Le client a l'impression d'avoir réalisé un transfert mémoire à mémoire. Parmi les éléments concrets d'une formule, il y a les nœuds de l'arbre : les opérateurs logiques (binaires et unaires) et les quantificateurs. Puis il y a les feuilles de l'arbre : les littéraux et les constantes propositionnelles. Notre structure de données est donc un ensemble d'éléments abstraits, composites ou feuilles héritant tous de l'élément abstrait de formule. Une formule concrète est sérialisable afin de la transformer en flux, puis désérialisable afin de la restaurer en mémoire. Afin de stocker plus efficacement les formules possédant une partie sous forme normale conjonctive, nous avons défini un élément concret feuille CNF, stockant directement une matrice de littéraux. Il est alors facile de représenter des formules CNF à l'aide uniquement des nœuds de quantificateurs et d'une feuille CNF.

Des traitements séparés des données

Dans la section précédente nous avions évoqué la notion de *visiteur*. En fait, *visiteur* est un patron de conception comportemental connexe au patron *composite*. Il permet de placer un traitement et donc le comportement dans une classe externe, le visiteur, plutôt que de le distribuer dans la structure de composites et de feuilles. Les visiteurs sont donc des traitements externes qui s'adaptent automatiquement à l'élément concret rencontré. La structure de données risque de peu changer, et nous pouvons ajouter ainsi autant de traitements qu'il sera nécessaire sans polluer les classes de la structure de données. Cependant ce patron possède deux conséquences négatives dont il faut tenir compte. Il peut compromettre l'encapsulation des objets : les champs des éléments concrets visités sont soit publics soit il y a des opérations publiques disponibles pour les lire et les modifier. Il faut admettre que dans une équipe de développement formée d'une personne, cette conséquence est négligeable. Deuxième conséquence, il devient plus difficile de rajouter des classes concrètes à la structure de données. En effet, il faut dans ce cas modifier l'ensemble des visiteurs abstraits et concrets pour prendre en compte le nouvel élément. Même si la structure de données change peu, si cela se produit le travail peut vite devenir fastidieux. Toutefois, il existe un moyen de contourner le problème en C++ à l'aide des patrons ou « *templates* » qui n'ont rien à voir avec les patrons de conception. Ces *patrons C++*, permettent d'écrire du code générique en passant un type en paramètre. Il est alors possible d'écrire une méthode par défaut pour visiter un type inconnu. Cette méthode

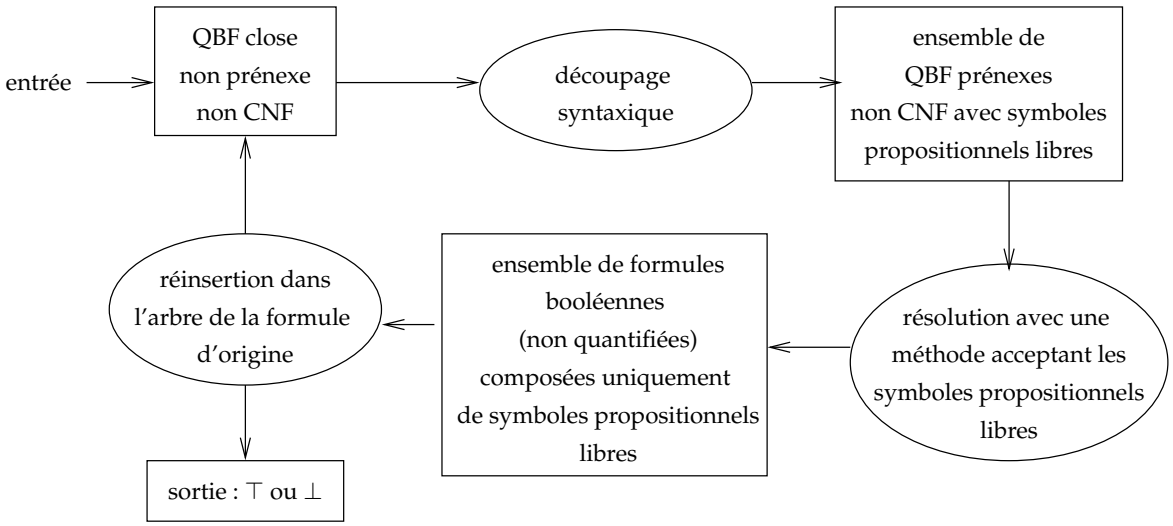


FIGURE 11 – Boucle générale d'exécution pour le problème de validité des QBF.

lève une exception et permet d'en aviser le client. Ainsi, seuls les traitements qui utilisent ce nouveau type doivent être étendus.

Une classe nommée *formula abstract visitor* décrit le visiteur abstrait dont tous les visiteurs concrets héritent. La figure 21 en Annexe (p. 121), illustre sans les *patrons c++*, quelques exemples de traitements :

- le visiteur *replace sibling into formula* effectue un changement dans la formule ;
- le visiteur *formula infix ostringstream* crée un flux représentant la formule en notation infixe ;
- le visiteur *formula simple polarity* calcule la polarité des différentes variables de la formule.

Chacun de ces traitements s'applique sur un élément abstrait de formule (objet de la classe *abstract formula*), rendant transparent pour le client la hiérarchie du modèle de données.

5.3 L'approche parallèle

Notre procédure parallèle consiste à appliquer plusieurs copies de procédures séquentielles sur une petite partie du problème original. Pour cela, il faut pouvoir extraire des sous-problèmes disjoints qui pourront être traités à part. Le découpage peut se faire au niveau syntaxique ou au niveau sémantique. Nous présentons un algorithme simple d'extraction syntaxique de sous-problèmes qui requiert de savoir résoudre des QBF avec variables libres. Puis, nous présentons notre architecture parallèle maître/esclaves dont le fonctionnement général est illustré par la figure 11 et décrit ci-après. Le nœud maître lit la formule en entrée, extrait syntaxiquement des sous-formules indépendantes, les dis-

Algorithme 8 : découpage_syntaxique

Données : Une QBF F
Résultat : Un ensemble de QBF prénexes avec variables libres
début
  si *est_prénexe(F)* **alors**
    retourner *{F}*
  fin
  suivant F **faire**
    cas où $\neg G$
      retourner *découpage_syntaxique(G)*
    fin
    cas où qxG, *avec* $q \in \{\exists, \forall\}$
      retourner *découpage_syntaxique(G)*
    fin
    cas où GoH, *avec* $o \in \{\vee, \wedge, \rightarrow, \leftrightarrow, \oplus\}$
      retourner *découpage_syntaxique*$(G) \cup$ *découpage_syntaxique*(H)
    fin
    autres cas
      retourner $\emptyset$
    fin
  fin
fin

tribue et réinsère les résultats dans la formule d'origine. L'opération est répétée jusqu'à avoir décidé de la validité ou de la non validité de la formule.

5.3.1 Le découpage syntaxique

L'extraction syntaxique de sous-problèmes, consiste à chercher des sous-formules que l'on peut traiter à part. De manière générale, cela consiste à prendre un opérateur logique binaire et travailler récursivement sur les deux sous-formules obtenues. Pour une procédure qui procède par élimination de quantificateurs, il faut au moins qu'une de ces deux sous-formules possède un quantificateur pour que l'extraction de sous-problèmes ait un sens.

Notre objectif est de travailler sur la formulation originale du problème en QBF, d'une part afin de garder toute l'information présente, d'autre part nous pensons que la position des quantificateurs a un sens et une justification pour l'auteur de la QBF. Alors nous avons choisi comme stratégie d'extraire l'ensemble des QBF prénexes non CNF avec potentiellement des variables libres ; cette stratégie est illustrée par l'algorithme 8.

Pour la QBF $\exists a \forall b(((a \leftrightarrow b) \wedge (\forall c(c \vee b))) \wedge (\exists d(a \wedge \neg d)))$, notre stratégie de découpage syntaxique extrait $\{\forall c(c \vee b), \exists d(a \wedge \neg d)\}$ comme ensemble des sous-problèmes qui peuvent être traités séparément. Nous appelons *largeur syntaxique* la taille de l'ensemble obtenu par un découpage. Ce découpage syntaxique ne varie pas d'une exécution

à l'autre, ce qui permet de limiter les hypothèses lors de l'étude des résultats expérimentaux. L'inconvénient de ce découpage est la faible largeur syntaxique obtenue qui souvent ne dépasse pas 2. Il est possible d'appliquer un pré-traitement avant l'algorithme de découpage syntaxique afin d'augmenter la largeur syntaxique, en déplaçant les quantificateurs. Cette opération peut se faire en minimisant la perte d'informations à condition de ne pas déplacer de quantificateurs à travers des bi-implications ou des ou-exclusifs (cf. chapitre 3).

5.3.2 Une architecture Maître/esclaves

Dans notre procédure parallèle le nœud maître est chargé d'assurer la répartition de tâches et la complétude de la recherche. Il lit la formule originale, extrait syntaxiquement des sous-problèmes, les distribue aux nœuds de calcul, puis il attend les réponses et réinsère les résultats dans la formule d'origine. Pour le problème de validité des QBF, il suffit de répéter l'opération jusqu'à obtenir $\top$ ou $\bot$. Le rôle d'un nœud de calcul est alors d'accepter une QBF prénexe non CNF avec des variables libres et de répondre par une formule booléenne non quantifiée équivalente, uniquement composée des variables libres. Pour cela, nos nœuds de calcul peuvent utiliser par exemple la méthode décrite dans la section 5.4.1.

À chaque cycle d'extraction de sous-formule nous pouvons attribuer une *largeur syntaxique*. Nous appelons *largeur syntaxique maximale*, la plus grande valeur de largeur atteinte. Ce nombre traduit le besoin maximal en nœuds de calcul pour une formule et un découpage syntaxique donné. De même, nous appelons *largeur syntaxique minimale*, la plus petite valeur de largeur rencontrée. Selon le découpage syntaxique choisi, l'approche parallèle pourrait vite être limitée. Tout nœud de calcul au delà de la largeur syntaxique maximale serait inutile. Or, de nombreux problèmes possèdent une largeur maximale de 1 ou 2 avec le découpage présenté dans la section 5.3.1.

Avant d'appliquer une méthode de résolution, le client utilise une heuristique de découpage. Il a la responsabilité de créer des sous-problèmes si le problème qu'il reçoit lui parait trop difficile. Le client peut par exemple effectuer un découpage sémantique sur les variables libres et renvoyer un ensemble de tâches au nœud maître. Si tous les clients sont occupés et que le problème est difficile, le client rajoute des tâches en file d'attente, si le problème semble facile il le résout a priori rapidement et attend une nouvelle tâche. Si des clients sont en attente de travail et que le problème est difficile, le nouveau découpage va permettre au maître de redonner des tâches.

5.4 Une implantation basée sur la procédure QSAT

Dans l'architecture présentée dans cette thèse, la tâche des nœuds esclaves est de calculer pour une QBF, une formule propositionnelle équivalente au sens de la préservation des modèles ; la QBF étant une sous-formule d'une QBF plus large, celle là est remplacée dans cette dernière par la proposition générée. Nous avons choisi pour instancier notre architecture parallèle une procédure de l'état de l'art qui applique un partitionnement

Algorithme 9 : simp

Données : Une proposition G
Données : Un ensemble de symboles propositionnels (libres) L
Données : Une valuation partielle v
Résultat : Une proposition G' équivalente à $\exists x\ G$ ne contenant que des symboles propositionnels de L

```
début
    si v ⊭ G;        /* G est insatisfiable pour la valuation v */
    alors
        si v est vide alors
            retourner ⊥
        sinon
            retourner clause(v) /* clause(v) renvoie une clause C sous
                CNF telle que v ⊭ C et v ⊭ G                    */
        fin
    sinon
        si v ⊨ G;    /* G est une tautologie pour la valuation v */
        alors
            retourner ⊤
        sinon
            Soit un symbole propositionnel y ∈ L
            retourner simp(G, L \ {y}, v[y := faux]) ∧ simp(G, L \ {y}, v[y := vrai])
        fin
    fin
fin
```

syntaxique de l'espace de recherche avec un oracle : la procédure `QSAT` [Plaisted *et al.*, 2003]. Nous décrivons tout d'abord la procédure `QSAT`, puis notre client de calcul. Nous faisons un bref tour d'horizon des choix techniques réalisés, puis nous développons un exemple complet pour illustrer le fonctionnement global de notre architecture.

5.4.1 QSAT

`QSAT` [Plaisted *et al.*, 2003] est une procédure de décision pour les QBF par l'élimination de quantificateurs, des plus internes vers les plus externes. Cette procédure opère itérativement sur une formule $Q(\exists x\ (F \wedge G))$ telle que x n'apparaît pas dans F ; l'équivalence $(\exists x\ (F \wedge G)) \equiv (F \wedge (\exists x\ G))$ permet d'isoler la sous formule $(\exists x\ G)$ qui va être mise en équivalence logique par une procédure $simp$ avec une formule G' ne contenant pas le symbole propositionnel x ; par substitution des équivalents $Q(\exists x\ (F \wedge G)) \equiv Q(F \wedge G')$. Le procédé est similaire avec une quantification universelle. Le processus est itéré jusqu'à élimination de tous les quantificateurs. La procédure $simp$ a besoin d'une procédure de décision pour le problème SAT et d'une procédure de décision pour le pro-

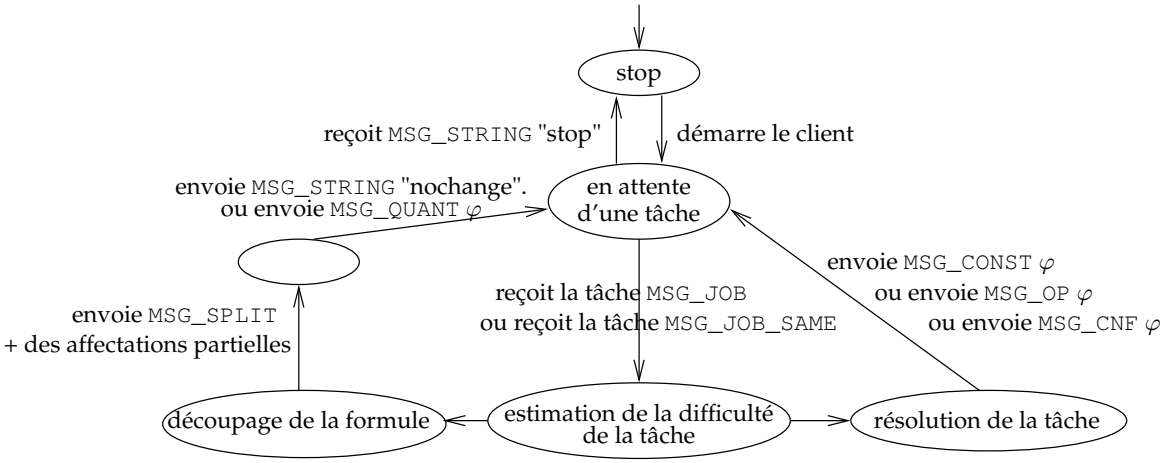


FIGURE 12 – Automate du client.

blème TAUT pour être effective. L'algorithme 9 illustre le fonctionnement de $simp$ en utilisant une valuation partielle qui propage les choix à chaque appel récursif, plutôt que le remplacement syntaxique qui, a contrario, est plus simple pour illustrer notre propos. Cette procédure $simp$ construit une CNF G' sur les symboles propositionnels libres de G telle que $G' \equiv (\exists x\ G)$. Elle opère comme un Davis-Logemann-Loveland par séparation de l'espace de recherche sur un symbole propositionnel libre y de G par appel récursif sur $[y \leftarrow \top](G)$ et $[y \leftarrow \bot](G)$. Pour $C_1 \dots C_n$ des constantes propositionnelles, si $[y_1 \leftarrow C_1] \dots [y_n \leftarrow C_n](G)$ est insatisfiable alors $simp((\exists x\ [y_1 \leftarrow C_1] \dots [y_n \leftarrow C_n](G)))$ retourne la clause $((y_1 \oplus C_1) \vee \dots \vee (y_n \oplus C_n))$; sinon si $[y_1 \leftarrow C_1] \dots [y_n \leftarrow C_n](G)$ est une tautologie alors $simp((\exists x\ [y_1 \leftarrow C_1] \dots [y_n \leftarrow C_n](G)))$ retourne $\top$; sinon si $(\exists x\ [y_1 \leftarrow C_1] \dots [y_n \leftarrow C_n](G))$ ne contient plus de symbole propositionnel libre alors soit cette formule est insatisfiable (et ce cas a déjà été traité) soit elle est satisfiable et $simp((\exists x\ [y_1 \leftarrow C_1] \dots [y_n \leftarrow C_n](G)))$ retourne $\top$; sinon un nouveau symbole propositionnel libre y_{n+1} est considéré et $G_\top = simp((\exists x\ [y_1 \leftarrow C_1] \dots [y_n \leftarrow C_n][y_{n+1} \leftarrow \top](G)))$ et $G_\bot = simp((\exists x\ [y_1 \leftarrow C_1] \dots [y_n \leftarrow C_n][y_{n+1} \leftarrow \bot](G)))$ sont calculés, $(G_\top \wedge G_\bot)$ est retourné. L'algorithme de la procédure $simp$ est basé sur la construction classique de la CNF comme étant la conjonction de la négation des lignes de la table de vérité qui falsifient la formule considérées comme des cubes. Seule l'implantation d'une double restriction de QSAT est décrite dans [Plaisted *et al.*, 2003] : restrictions à SAT et aux formules sous CNF. Dans le cadre de cette dernière restriction, la recherche d'une forme $Q(\exists x\ (F \wedge G))$ telle que x n'apparaît pas dans G est triviale ; seul le choix de x demeure et permet l'édification de stratégies.

5.4.2 Description du client

D'un point vue général, un client reçoit une QBF quelconque avec des variables libres et doit renvoyer une formule booléenne non quantifiée composée uniquement de ces va-

riables libres. La procédure QSAT, décrite dans le paragraphe précédent, s'y prête bien. De plus, il est possible d'éliminer plusieurs quantificateurs consécutifs du même type. Pour une QBF prénexe, il faudra autant d'itérations de la procédure QSAT que d'alternances de quantificateurs. La procédure QSAT possède un inconvénient majeur, elle est efficace sur une certaine catégorie de formules dites longues et fines (long and thin [Plaisted *et al.*, 2003]). De manière générale, notre architecture prévoit qu'un client puisse renoncer à résoudre une tâche qu'il juge trop difficile. La figure 12, décrit le fonctionnement d'un client.

Tout d'abord le client démarre et se met en attente d'une tâche. Puis il reçoit le message `MSG_JOB` suivi d'une tâche : une formule accompagnée d'une affectation partielle ou `MSG_JOB_SAME` suivi uniquement d'une affectation partielle. Le client appelle une fonction heuristique tentant d'évaluer rapidement la difficulté de la tâche reçue :

- la tâche est considérée comme abordable, alors elle est exécutée et un message est retourné :
  - `MSG_CONST` avec $\top$ ou $\bot$;
  - `MSG_OP` avec une formule propositionnelle seulement constituée des symboles propositionnels libres ;
  - `MSG_CNF` avec une formule comme pour `MSG_OP` mais en CNF.
- la tache est considérée trop difficile a résoudre, alors un découpage sémantique est appliqué à la formule et le message `MSG_SPLIT` est envoyé avec un ensemble d'affectations partielles. Puis est envoyé un message :
  - `MSG_STRING` plus « nochange » ;
  - `MSG_QUANT` avec la formule partiellement traitée.

Notre fonction heuristique pour estimer la difficulté d'une tâche est très simple :

- si une tâche a déjà été découpée par cette heuristique, elle est abordable ;
- sinon, si le nombre de variables libres est inférieur à une constante, la tâche est abordable ;
- sinon la tâche n'est pas abordable.

De même notre procédure de découpage sémantique est très simple. À la lecture de la formule, les variables libres sont empilées. Si un découpage est nécessaire, on calcule le nombre de variables libres à affecter sans toutefois dépasser une autre constante définissant le plus fin découpage autorisé. Ensuite, est dépilé le nombre de variables et est généré l'ensemble des affectations partielles possibles. Cet ensemble sera envoyé après `MSG_SPLIT`.

Les plus grandes améliorations que l'on puisse apporter au client sont au niveau de la fonction heuristique qui évalue la difficulté d'une tâche et la façon de choisir les variables du découpage sémantique. L'avantage de notre implantation est le comportement déterministe de ces deux composants, qui permet de limiter le nombre de variations lors de l'exécution. Actuellement, un client découpe toujours une même tâche et cela de la même manière.

Actuellement, un client finit toujours un travail de découpage en retournant le message `MSG_STRING` suivi de « *nochange* », mais dans le futur nous envisageons de pouvoir traiter partiellement une formule.

5.4.3 Choix techniques

MPI. Toute la partie communication est réalisée à l'aide du standard Message passing Interface (MPI)[Snir *et al.*, 1995] détaillé au chapitre 4, section 4.5.2. Nous avons choisi comme implantation Open MPI. Pour notre approche avec une structure de données composite, MPI possède une lacune : il n'est pas possible simplement d'envoyer et recevoir des types de données complexes de façon native. La bibliothèque C++ Boost.MPI est une surcouche répondant à cette contrainte à l'aide de Boost.Serialization. Toutes nos communications sont synchrones et utilisent le send/recv de Boost.MPI, faisant appel à MPI_Send et à MPI_Recv d'Open MPI.

`MiniSat`. Pour utiliser la méthode de la section 5.4.1, il faut pourvoir répondre à deux questions : la formule est-elle une tautologie ? Sinon, la formule est-elle insatisfiable ? Pour répondre, nous avons intégré la procédure de décision de `MiniSat`. Outre le fait que cette procédure est très performante, ses sources sont disponibles et distribuées librement sous licence MIT. Les structures de données dans `MiniSat` sont très différentes des nôtres et sont orientées pour la performance. Nous avons donc une interface (un visiteur) permettant de formuler nos requêtes dans le format de données de `MiniSat`. À l'inverse, nous pouvons interpréter les réponses de `MiniSat` pour les intégrer directement dans notre structure de données. Il est ainsi possible à partir de notre structure de données expressive et extensible, de faire des traitements dans un modèle performant, le tout à l'aide d'un traitement en temps polynomial (linéaire) par rapport à la taille de la formule (nombre de nœuds).

La mise en cache ou caching. Afin de réduire le besoin en bande passante, chaque nœud de calcul garde en cache l'objet solveur `MiniSat` de la dernière sous-formule traitée. Si une nouvelle tâche sur la même sous-formule arrive, le premier gain, est l'économie du transfert de la formule. Mais ce n'est pas tout : `MiniSat` utilise le clause learning. L'instance déjà entraînée de `MiniSat` peut possiblement répondre plus vite à une nouvelle question. L'idéal serait que les différents nœuds de calcul partage l'apprentissage réalisé par chacun, mais ce n'est pas le cas. De plus, il faudrait étudier en contre partie, le surplus de trafic généré par la mutualisation de ces informations sur un système à mémoire distribuée. Il est envisageable de limiter l'apprentissage à un ensemble de processus s'exécutant sur le même serveur et ainsi d'utiliser les avantages de la programmation sur système à mémoire partagée.

5.4.4 Exemple complet

Suite à toutes ses explications, il nous semble intéressant de développer un exemple pédagogique visuel et textuel afin de regrouper tous les aspects de notre architecture et de montrer leur coordination. Pour cela nous avons choisi une formule du problème de chaîne de bi-implication présenté dans le chapitre 3 (section 3.4.1) que nous avons limité

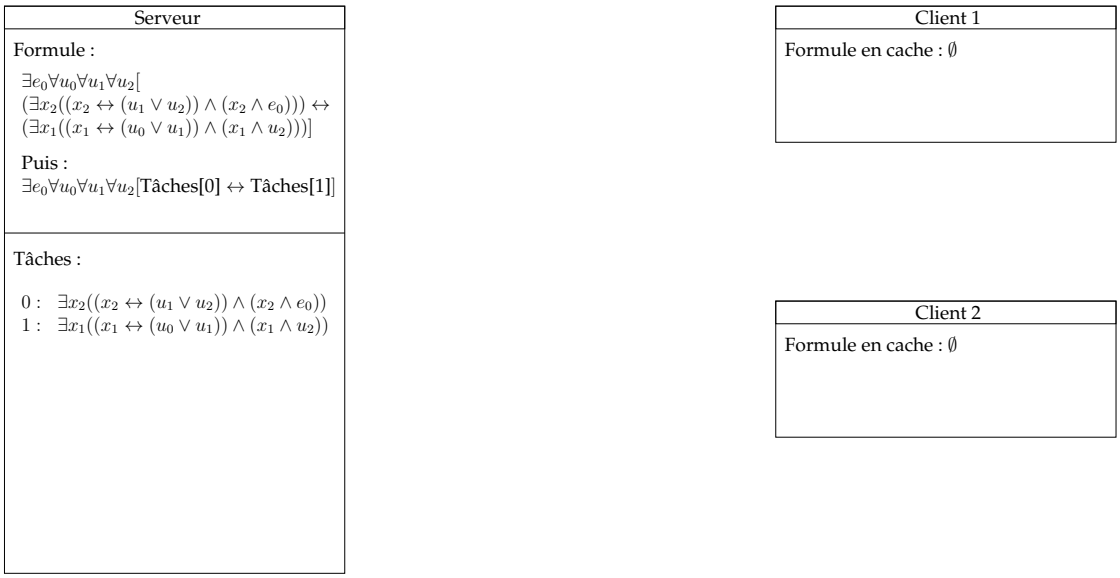


FIGURE 13 – Lecture de la formule, découpage syntaxique et insertion de références aux tâches dans la formule (1/7).

à deux maillons :

$$\exists e_0 \forall u_0 \forall u_1 \forall u_2[(\exists x_2((x_2 \leftrightarrow (u_1 \vee u_2)) \wedge (x_2 \wedge e_0))) \leftrightarrow (\exists x_1((x_1 \leftrightarrow (u_0 \vee u_1)) \wedge (x_1 \wedge u_2)))]$$

La *largeur syntaxique maximale* de cette formule est de 2.

Pas 1 sur 7 (figure 13). Le nœud maître ou serveur lit la formule en entrée, il ne possède aucune tâche à distribuer et inscrit automatiquement tous les nœuds esclaves ou clients en tant que esclaves prêts à travailler. Les client démarrent, ils n'ont pas de formule en cache. Le serveur applique l'algorithme de découpage syntaxique présenté dans la section 5.3.1 et met à jour la formule afin de faire référence aux tâches qui seront accomplies. Les clients sont toujours en attente de travail. Dans notre exemple, l'algorithme de découpage syntaxique trouve 2 sous-formules prénexes avec des variables libres. La formule $\exists x_2((x_2 \leftrightarrow (u_1 \vee u_2)) \wedge (x_2 \wedge e_0))$ possède 3 variables libres u_1, u_2, e_0, ainsi qu'une variable x_2 à éliminer. La formule $\exists x_1((x_1 \leftrightarrow (u_0 \vee u_1)) \wedge (x_1 \wedge u_2))$ possède 3 variables libres u_0, u_1, u_2, ainsi qu'une variable x_1 à éliminer. L'ensemble des formules ainsi obtenues, est stocké dans un tableau de tâches à effectuer

Pas 2 sur 7 (figure 14). Pendant cette étape le serveur s'occupe de répartir les tâches à effectuer parmi les clients. Dans notre exemple, à cette étape, il y a exactement 2 tâches et 2 clients, la répartition est donc très naturelle. Le serveur envoie un message `MSG_JOB` suivi d'une tâche à chaque client. Les tâches ainsi distribuées sont mémorisées dans un tableau qui ne figure pas sur nos figures. Son objectif est double : il permet de lier un

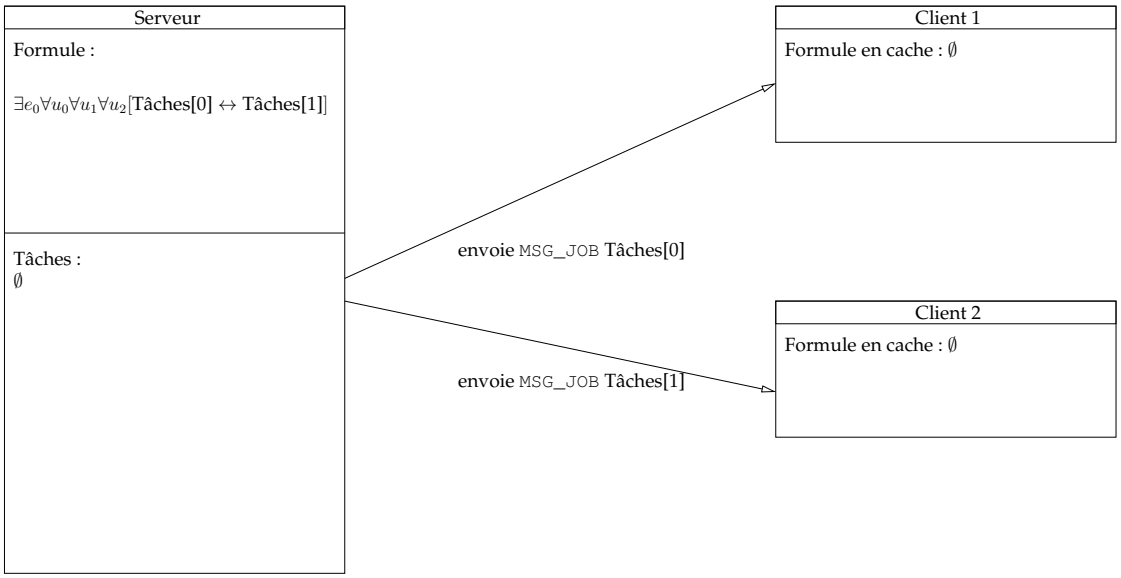


FIGURE 14 – Distribution des tâches aux clients de calcul (2/7).

client à une adresse dans la formule d'origine, afin d'effectuer les futurs remplacements ; et il permet de connaître la dernière formule envoyée afin de faire de la mise en cache pour économiser de la bande passante. La communication étant synchrone et le serveur séquentiel, les clients reçoivent les formules à tour de rôle et appliquent la procédure $simp$ de QSAT sur leurs formules respectives. Pendant ce temps, le serveur est bloqué en réception, il attend la réponse d'un des clients.

Pas 3 sur 7 (figures 15). Cette étape résume un ensemble d'évènements. La figure 15, illustre d'abord la possibilité que les clients répondent de façon presque simultanée. De part le mode de communication, le serveur ne peut traiter qu'une réponse à la fois : Il reçoit le résultat, effectue les modifications nécessaires dans la formule à traiter et redonne immédiatement un travail au client si possible. Ensuite, il réalise exactement les mêmes étapes jusqu'à avoir répondu à toutes les communications initiées par des clients souhaitant donner leurs résultats. La figure 15 illustre ensuite le résultat de toutes ses opérations. Revenons sur l'application de $simp$ sur les formules. Le raisonnement pour les 2 tâches étant le même nous n'expliquons que le résultat obtenu à partir de la formule $\exists x_2((x_2 \leftrightarrow (u_1 \vee u_2)) \wedge (x_2 \wedge e_0))$. La procédure $simp$ énumère toutes les affectations des variables libres qui peuvent falsifier la formule ; nous obtenons pour notre exemple $(\neg e_0 \vee (\neg u_1 \wedge \neg u_2))$. Puisque nous avons exactement toutes les « non » solutions, la négation de cette formule encode l'ensemble des solutions en fonction des variables libres et $\neg(\neg e_0 \vee (\neg u_1 \wedge \neg u_2)) \equiv (e_0 \wedge (u_1 \vee u_2))$. Cette méthode garantit d'obtenir une formule équivalente sous CNF, ce qui est un sérieux avantage dans le cas de `QSAT` et ses restrictions à SAT sous CNF. Dans l'exemple, les clients envoient `MSG_OP` suivi d'une formule,

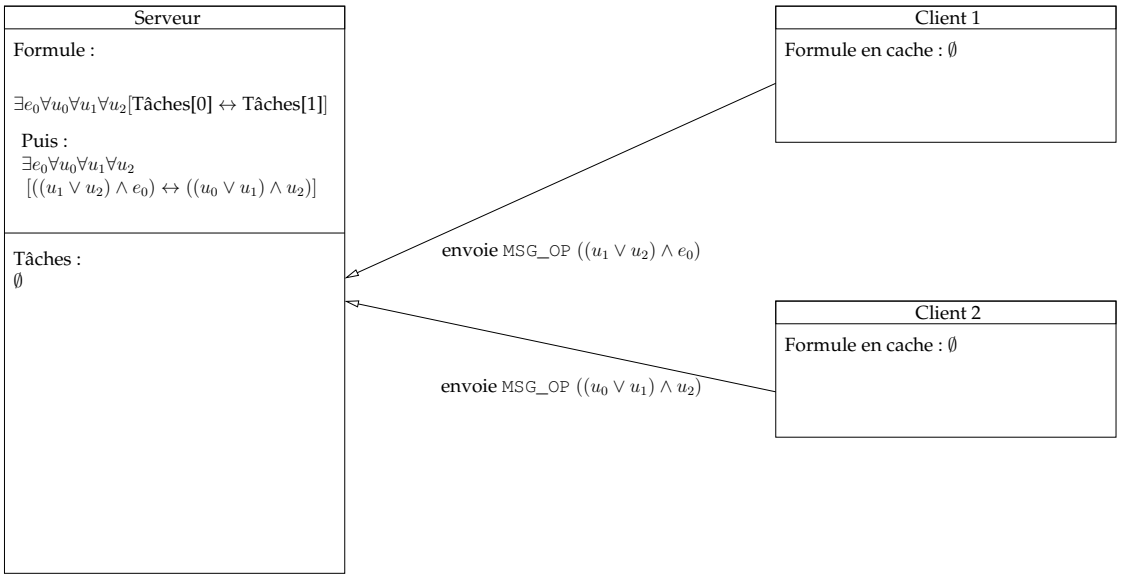


FIGURE 15 – Réponses des clients et réinsertion des résultats dans la formule du serveur (3/7).

en pratique il est possible d'utiliser aussi le message `MSG_CNF`.

Pas 4 sur 7 (figure 16). Le serveur applique l'algorithme de découpage syntaxique. Dans notre exemple, l'algorithme trouve une seule formule prénexe : $\exists e_0 \forall u_0 \forall u_1 \forall u_2 [((u_1 \vee u_2) \wedge e_0) \leftrightarrow ((u_0 \vee u_1) \wedge u_2)]$. Puis le serveur met à jour la formule afin de faire référence à cette unique tâche à effectuer. A cette étape, il y a 1 tâche pour 2 clients, le serveur choisit le client qui n'a rien fait depuis le plus longtemps. Ce choix volontairement très simple peut être amélioré dans le cas où le nombre de clients est supérieur au nombre de tâches. Si le serveur possède des informations sur les clients (puissance/mémoire/réseau/etc...), il peut alors tenter de choisir celui qui sera le plus efficace. Ensuite, le serveur envoie un message `MSG_JOB` suivi de notre unique tâche au client choisi. Afin d'économiser de la bande passante, le serveur utilise la mémorisation de la dernière formule envoyée à chaque client.

Pas 5 sur 7 (figure 17). Pour des raisons d'illustrations, nous supposons que le client 1 juge que la tâche reçue est trop difficile, ce qui n'est évidement pas le cas. Il applique l'algorithme de découpage sémantique et pour cela il sélectionne les variables u_0 et u_1. En pratique, notre algorithme sélectionne les variables en fonction de leur ordre de lecture. Ce premier point est grandement améliorable. Puis, le découpage génère exhaustivement l'ensemble des affectations possibles, alors que de simples propagations permettraient d'en limiter le nombre. Pour notre exemple l'ensemble des affectations est

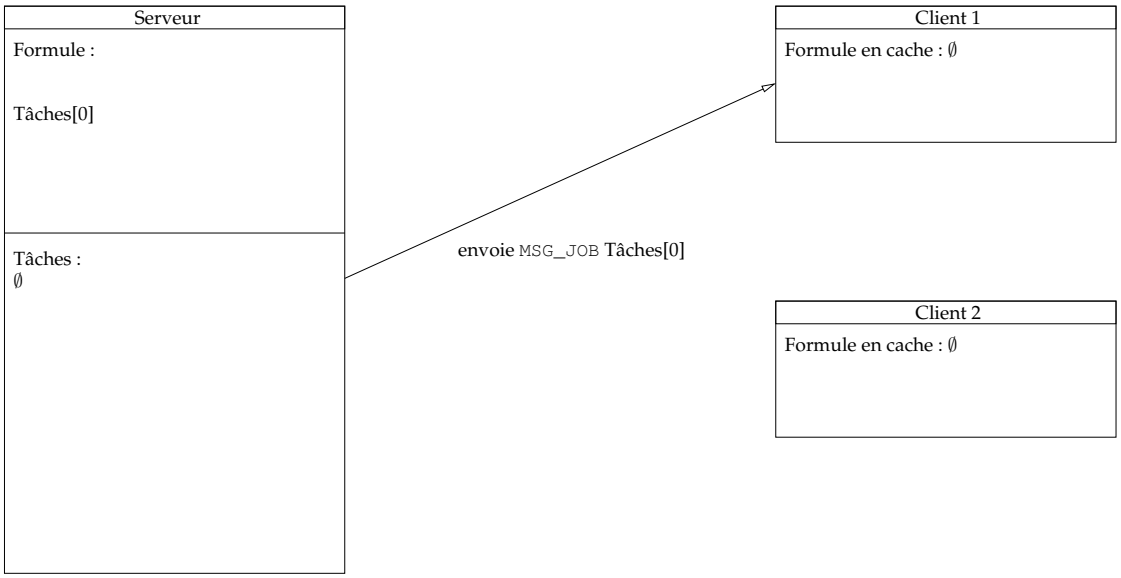


FIGURE 16 – Le serveur extrait syntaxiquement une seule tâche et l'envoie à un client (4/7).

$$\begin{array}{ll}\{ & [u_0 \leftarrow \top, u_1 \leftarrow \top](F), \\ & [u_0 \leftarrow \top, u_1 \leftarrow \bot](F), \\ & [u_0 \leftarrow \bot, u_1 \leftarrow \top](F), \\ & [u_0 \leftarrow \bot, u_1 \leftarrow \bot](F) \quad \}.\end{array}$$

Le client mémorise la formule $\exists e_0 \forall u_0 \forall u_1 \forall u_2 [((u_1 \vee u_2) \wedge e_0) \leftrightarrow ((u_0 \vee u_1) \wedge u_2)]$, car il est possible qu'il résolve une des tâches qu'il a générées. Le client envoie le message `MSG_SPLIT` suivi des affectations, puis `MSG_STRING` avec le message « nochange » pour signifier qu'aucun changement n'a été apporté sur la formule originale.

Pas 6 sur 7 (figure 18). Le serveur doit prendre en compte le découpage sémantique alors que ce n'est pas sa philosophie de fonctionnement. Pour générer les tâches lors d'un découpage de ce type, le serveur fait confiance aux clients. Il doit cependant mettre à jour syntaxiquement la formule. Pour le client 1, les variables u_0 et u_1 sont éliminées de la formule traitée indépendamment des quantificateurs qui les lient puisque la formule a été jugée trop difficile. Pour le serveur ces variables existent toujours potentiellement. Pour tout quantificateur $q \in \{\exists, \forall\}$ et pour x une variable, l'équivalence suivante, orientée satisfaction de la formule, est vraie :

$$qx(F) \equiv qx((x \wedge [x \leftarrow \top](F)) \vee (\neg x \wedge [x \leftarrow \bot](F)))$$

Démonstration. Pour un quantificateur existentiel, le résultat est immédiat. Pour $q = \forall$, il suffit d'appliquer la sémantique du quantificateur universel et de simplifier :

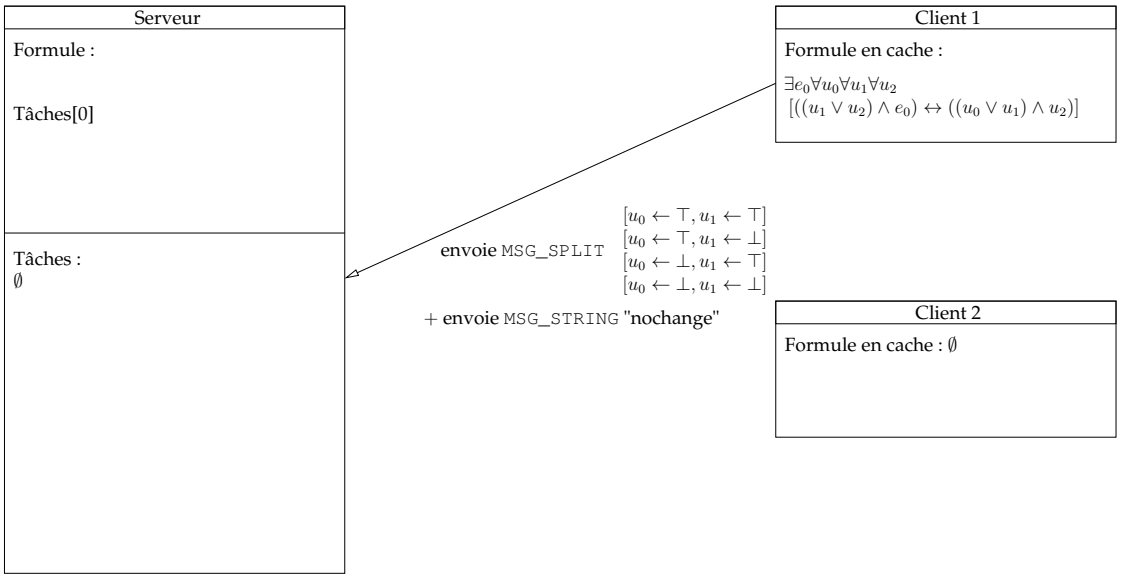


FIGURE 17 – Le client juge la tâche trop difficile, effectue un découpage sémantique et l'envoie au serveur. Le client met en cache cette formule pour économiser son transfert depuis le serveur si besoin. (5/7)

$$\begin{array}{ll} & \forall x((x \wedge [x \leftarrow \top](F)) \vee (\neg x \wedge [x \leftarrow \bot](F))) \\ \equiv & [x \leftarrow \top]((x \wedge [x \leftarrow \top](F)) \vee (\neg x \wedge [x \leftarrow \bot](F)))\wedge \\ & [x \leftarrow \bot]((x \wedge [x \leftarrow \top](F)) \vee (\neg x \wedge [x \leftarrow \bot](F))) \\ \equiv & [x \leftarrow \top](F) \wedge [x \leftarrow \bot](F) \equiv \forall x(F) \end{array}$$

□

De la même manière, l'équivalence suivante, duale de la précédente et orientée falsification de la formule, est vraie :

$$qx(F) \equiv qx((\neg x \vee [x \leftarrow \top](F)) \wedge (x \vee [x \leftarrow \bot](F)))$$

Pour illustrer le fonctionnement de notre architecture, nous avons choisi d'utiliser la première équivalence, orientée satisfaction. Dans notre exemple, le serveur obtient la formule

$$\begin{array}{ll} \exists e_0 q u_0 q' u_1 \forall u_2 & [(u_0 \wedge u_1 \wedge \text{Tâches[0]}) \vee \\ & (u_0 \wedge \neg u_1 \wedge \text{Tâches[1]}) \vee \\ & (\neg u_0 \wedge u_1 \wedge \text{Tâches[2]}) \vee \\ & (\neg u_0 \wedge \neg u_1 \wedge \text{Tâches[3]})] \end{array}$$

valable pour tous q et q', des quantificateurs quelconques. En tenant compte de la valeur de q et q', il est possible d'appliquer la sémantique des quantificateurs et d'obtenir la formule

$$\exists e_0 \forall u_2 (\text{Tâches[0]} \wedge \text{Tâches[1]} \wedge \text{Tâches[2]} \wedge \text{Tâches[3]}).$$

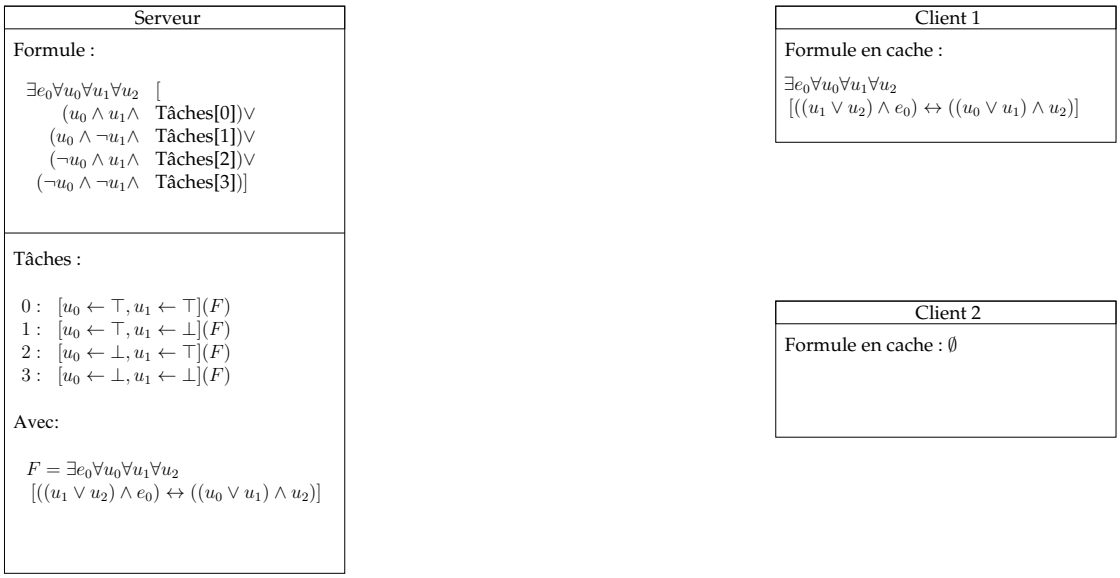


FIGURE 18 – Le serveur ajoute les tâches correspondant au découpage sémantique du client et intègre celui-ci dans la formule (6/7).

Notre serveur ne réalise pas cette opération sémantique, simple pour cet exemple, qui nous permettrait de voir qu'il suffit qu'une tâche soit équivalente à $\bot$ pour que la formule soit insatisfiable. Comme amélioration, il est envisageable de raisonner sur une formule partiellement résolue et d'effectuer des propagations pour conclure plus rapidement. Cette amélioration est directement liée à l'amélioration du découpage sémantique côté client : une première phase de propagation lors du découpage sémantique, puis une deuxième phase de propagation lors de la réception des résultats des tâches par le serveur.

Pas 7 sur 7 (figure 19). La fin de l'exécution est relativement simple. Le serveur distribue les tâches restantes en utilisant, quand c'est possible, *la mise en cache* de formules. Il est alors possible, quand un client possède la formule à traiter, d'envoyer le message `MSG_JOB_SAME` suivi d'une affectation ou le cas échéant, `MSG_JOB` suivi de la formule et d'une affectation. Toutes les tâches restantes sont équivalentes à $\bot$, le serveur attend d'avoir le résultat des 4 tâches et propage les résultats :

$$\begin{aligned}\exists e_0 \forall u_0 \forall u_1 \forall u_2 [\quad & (u_0 \wedge u_1 \wedge \quad \bot) \vee \\ & (u_0 \wedge \neg u_1 \wedge \quad \bot) \vee \\ & (\neg u_0 \wedge u_1 \wedge \quad \bot) \vee \\ & (\neg u_0 \wedge \neg u_1 \wedge \quad \bot)] \equiv \bot\end{aligned}$$

Le serveur conclut à l'insatisfiabilité de la formule, affiche le résultat et demande l'arrêt des clients par l'envoi du message `MSG_STRING` suivi de « stop » ; il peut ensuite rendre la main.

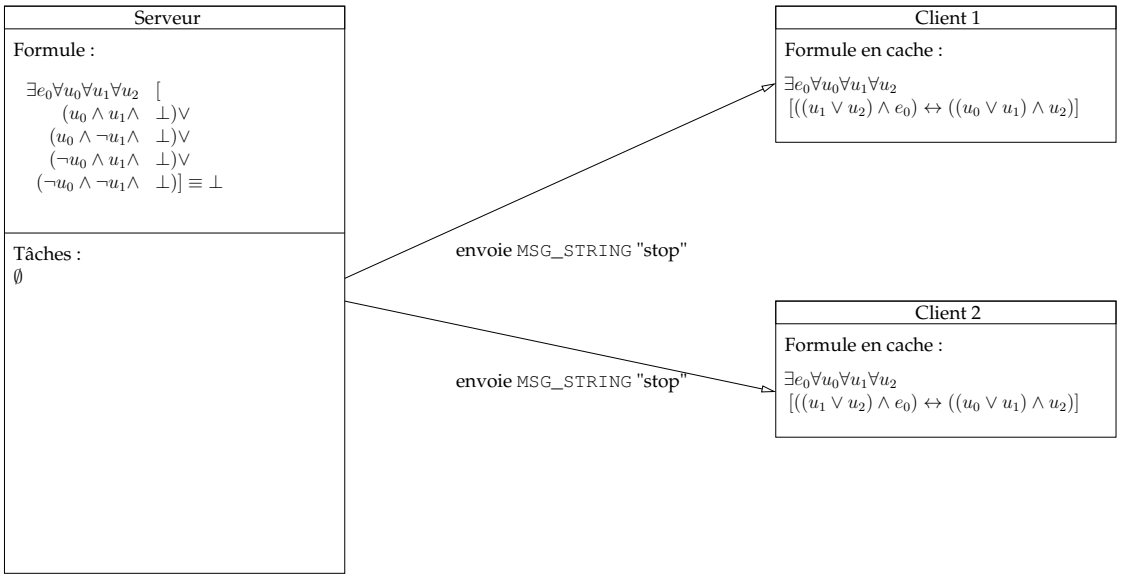


FIGURE 19 – L'exécution se déroule en répétant les étapes 2 et 3 afin d'effectuer toutes les tâches et de les réintégrer dans la formule. Le serveur conclut à l'invalidité de la formule et demande l'arrêt des clients (7/7).

5.5 Résultats expérimentaux

Afin d'évaluer notre architecture, nous avons effectué quelques tests préliminaires sur quelques instances de *qbflib* [Giunchiglia *et al.*, 2001a] et sur quelques problèmes que nous avons générés. Le découpage sémantique est strictement identique quel que soit le nombre de clients de calcul. Les tests ont été réalisés sur une machine de calculs haute performance (HPC) composée de 12 serveurs de calcul Bull Novascale® R422, connectés entre eux par 2 réseaux Ethernet gigabit. Chaque serveur possède 2 nœuds de calcul de type 2x Intel Xeon® E5440 à 2.83GHz et possède 16GB de mémoire locale. Les machines exécutent une version 2.6.18 du noyau Linux en version 64 bits.

Nous avons choisi quelques résultats pour illustrer différents cas. La première colonne de nos tables représente le nombre de clients de calcul. En plus, il faut ajouter le processus maître tournant sur un processeur dédié. Les résultats sont donnés sous la forme d'un couple *temps en secondes et accélération relative*.

5.5.1 Formules prénexes

La table 11 présente les résultats de 4 instances : $counter8\_8$, $counter8\_16$, $ring4\_5$ et $semaphore5\_4$. Elles font partie de la suite QBF1.0 disponible sur *qbflib*. Toutes ces formules sont prénexes, leur largeur syntaxique maximale est donc 1. Le seul choix que nous ayons pour traiter ces formules en parallèle est l'utilisation de l'extraction séman-

Nb. clients	counter8_8		counter8_16		ring4_5		semaphore5_4	
1	105	1,0	7264	1,0	18483	1,0	58812	1,0
2	52	2,0	6627	1,1	11215	1,6	27741	2,1
4	28	3,8	4560	1,6	3601	5,1	19938	2,9
8	15	7,0	2437	3,0	4940	3,7	5537	10,6
16	11	9,5	3247	2,2	2268	8,1	4320	13,6
32	7	15,0	967	7,5	857	21,6	1027	57,2
64	6	17,5	3950	1,8	247	74,8	582	101,1
128	7	15,0	3548	2,0	128	144,4	780	75,4

TABLE 11 – Temps d'exécution en secondes et accélération relative pour différentes instances prénexes, selon le nombre de clients et sans propagation simple sur la formule en entrée.

tique de sous-problèmes décrite dans la section 5.4.2. Le premier problème, counter8_8, est simple ; l'augmentation du nombre de clients est vraiment efficace jusqu'à 8. L'explication est simple : plus nous utilisons de processus de calcul, plus le temps d'initialisation lié à MPI est long. Par exemple, l'initialisation pour 128 processus prend presque 5 secondes.

Pour évaluer notre approche avec beaucoup de ressources de calcul, il nous faut des problèmes plus conséquents. Le second problème, $counter8\_16$, satisfait cette contrainte, malgré cela les temps sont irréguliers et l'accélération relative est assez mauvaise. Deux phénomènes surviennent ici. Tout d'abord, certains sous-problèmes sont très difficiles à résoudre. Par exemple, certaines tâches requièrent plusieurs centaines de secondes. Une tâche qui monopolise une ressource pendant 10% du temps, va limiter à 10 l'accélération relative maximale dans le meilleur des cas, c'est à dire en commençant la recherche par cette tâche. Le second phénomène, est lié à l'utilisation de `MiniSat` comme oracle. Chacun des 4 problèmes ne possède qu'une alternance de quantificateurs. C'est pourquoi, chaque nœud de calcul reçoit la formule une seule fois et crée une unique instance de la procédure `MiniSat`. Toutes les autres tâches utiliseront cette instance en cache et tireront des bénéfices de l'apprentissage de clauses déjà réalisé localement lors des tâches précédentes. La grande part d'indéterminisme lors de notre exécution en parallèle est dans la distribution des tâches, complètement imprévisible. Par extension, chaque sous-problème sémantique entraîne une instance de `MiniSat` : cet apprentissage est imprévisible et différent à chaque exécution. Plus le nombre de nœuds de calcul augmente, plus la probabilité d'apprendre localement une information intéressante avant un sous-problème difficile diminue. Avoir plus de nœuds de calcul, n'implique pas nécessairement une résolution plus rapide s'il n'est pas possible de partager des informations.

Pour le problème $ring4\_5$, nous observons une accélération super-linéaire avec 64 et 128 processus de calcul. À l'inverse du problème précédent, nous pensons que l'apprentissage de `MiniSat` a un effet, positif cette fois-ci. Il est possible que certains sous-problèmes entraînent efficacement la plupart des instances de `MiniSat`. Avec l'accroissement du nombre de nœuds de calcul, chaque solveur reçoit moins de sous-problèmes

Nb. clients	counter8_8		counter8_16		ring4_5		semaphore5_4	
1	52	1,0	6069	1,0	8718	1,0	25647	1,0
2	27	1,9	2959	2,1	3536	2,5	6530	3,9
4	14	3,7	1855	3,3	1597	5,4	1855	13,8
8	10	5,2	3338	1,8	914	9,5	2226	11,5
16	6	8,7	1114	5,4	349	25,0	1993	12,9
32	5	10,4	1527	4,0	202	43,2	778	33,0
64	5	10,4	1249	4,9	89	98,0	240	106,9
128	5	10,4	1750	3,5	38	229,4	588	43,6

TABLE 12 – Temps d'exécution en secondes et accélération relative pour différentes instances prénexes, selon le nombre de clients et avec propagation simple sur la formule en entrée.

à résoudre. Peut-être cela leur permet-il de garder plus longtemps des informations plus pertinentes. Nous remarquons aussi qu'aucune tâche ne monopolise beaucoup de ressources. De part la nature aléatoire des exécutions parallèles, il faudrait multiplier les exécutions pour consolider ces hypothèses. Or, notre procédure est loin d'intégrer toutes les améliorations nécessaires pour avoir un temps d'exécution séquentiel comparable aux procédures de l'état de l'art. Ces tests préliminaires ont essentiellement comme objectifs de valider l'approche et de faire le point sur le niveau des performances actuelles.

Nous observons une accélération super-linéaire pour le problème $semaphore5\_4$ avec 32 et 64 clients. par contre, pour 128 le gain est inférieur à celui pour 64. Contrairement à $ring5\_4$, certaines tâches sont très longues et peuvent représenter plus de 50% du temps total d'exécution pour 64 client ou plus. Comme pour counter8_16, plus l'apprentissage est réparti sur différents clients plus les tâches longues sont imprévisibles et potentiellement pénalisantes.

La table 12 présente les résultats pour les mêmes problèmes mais avec l'application de la propagation booléenne [Stéphan, 2006a] (qui consiste en des déductions à partir des propriétés locales des quantificateurs et des connecteurs) et la recherche des littéraux monotones (n'apparaissant que dans une seule polarité) uniquement sur la formule en entrée. Ces améliorations simples montrent qu'il sera possible d'améliorer les performances de notre procédure de résolution en appliquant les techniques de l'état de l'art. Ces améliorations pourront être appliquées aussi sur les sous-problèmes et lors du découpage sémantique. Nous relevons que pour $ring4\_5$ les temps obtenus correspondent à une accélération super-linéaire et pour 128 clients le gain est de 229,4.

5.5.2 Formules quelconques

Contrairement aux problèmes précédents, $adder\_6$ et $chaine\_30$ ne sont pas prénexes et possèdent des bi-implications et/ou des ou exclusifs. Le problème $adder\_6$ est la vérification de l'additionneur 6-bits, présentée au chapitre 3, section 3.4.2. Il possède une largeur syntaxique de 2, puis de 1 lors de la dernière itération. Le problème $chaine\_30$ est

Nb. clients	adder_6		chaine_30	
1	3479	1,0	68343	1,0
2	1377	2,5	33068	2,1
4	995	3,5	16181	4,2
8	1788	1,9	7504	9,1
16	708	4,9	3780	18,1
32	1242	2,8	2043	33,5
64	1238	2,8	1023	66,8
128	1007	3,5	438	156,0

TABLE 13 – Temps d'exécution en secondes et accélération relative pour différentes instances quelconques, selon le nombre de clients et avec propagation simple sur la formule en entrée.

le problème théorique introduit dans le chapitre 3, section 3.4.1. Il possède une largeur syntaxique de 30 lors de la première itération, puis de 1 ensuite. La table 13 résume les différents résultats.

Pour $adder\_6$, le gain est bon jusqu'à 4 clients. Pour ce problème la dernière itération est la tâche la plus longue, or cette tâche ne possède qu'une alternance de quantificateurs et aucune variable libre : la formule est passée à une procédure SAT, donc seul un client est actif. Dans ce genre de situation, il existe deux possibilités. Premièrement, nous pourrions faire appel à une procédure SAT en parallèle comme `ManySAT` [Hamadi *et al.*, 2009], une des meilleures procédures parallèles à la compétition SAT 2010 (voir [Jabbour, 2008] pour un panorama des solveurs SAT parallèles). Deuxièmement, nous pourrions utiliser notre algorithme de découpage sémantique et répartir les tâches sur les différents clients. Une autre remarque s'impose : le problème d'équivalence de circuits est **coNP**-complet ($\Pi_1^{\mathbf{P}}$) [Papadimitriou, 1994]. Le codage pour QBF proposé dans la littérature [Ayari *et al.*, 1999; Ayari et Basin, 2002] fait paraître le problème dans la classe de complexité $\Pi_2^{\mathbf{P}}$. D'après la définition de la hiérarchie polynomiale **HP**, $\Pi_2^{\mathbf{P}} = \mathbf{co}(\mathbf{NP}^{\mathbf{NP}})$. Cela traduit ce que fait notre procédure en utilisant `MiniSat` comme oracle de la classe **NP**. La conséquence d'un codage dans une mauvaise classe de complexité se fait immédiatement ressentir avec l'algorithme de `QSAT`. L'appel supplémentaire à l'oracle génère des clauses inutilement et possiblement en très grande quantité, pénalisant les étapes suivantes. Et lorsque notre procédure arrive à la dernière étape et demande si la formule est une tautologie, la dite formule est bien plus grande que nécessaire.

Pour $chaine\_30$, nous observons une accélération relative super-linéaire. Dans un premier temps, le nœud maître distribue 30 sous-problèmes qui sont simplifiés très rapidement, puis la largeur syntaxique passe à 1. L'extraction sémantique de sous-problèmes prend le relais. Comme constaté pour $ring5\_4$, les tâches sont de longueurs régulières. Nous pouvons faire les mêmes remarques que pour l'additionneur au niveau de la complexité. Le problème de chaîne de bi-implication peut s'exprimer avec un QBF dans la classe de complexité $\Sigma_2^{\mathbf{P}}$. Le codage proposé utilisant des résultats intermédiaires existentiellement quantifiés fait paraître la formule dans la classe de complexité $\Sigma_3^{\mathbf{P}} = \mathbf{NP}^{\mathbf{\Sigma_2^P}}$.

La différence avec le problème précédent est qu'ici l'invalidité de la formule est prouvée à l'avant dernier étage et donc avant que la formule entière ne soit passée à un unique client. De plus, dans cet exemple, les résultats intermédiaires sont remplacés par un ensemble succinct de clauses contrairement à $adder\_6$. Les deux découpages ont ainsi permis de maintenir une charge conséquente pour tous les clients grâce à une grande granularité obtenue surtout lors du découpage sémantique.

Conclusion générale

Dans cette thèse, intitulée « Formules booléennes quantifiées : transformations formelles et calculs parallèles », nous avons présenté nos contributions dans le domaine des formules booléennes quantifiées. D'une part, ce travail constitue un apport quant aux transformations formelles des QBF. D'autre part, il décrit les bases d'une architecture ouverte de calcul en parallèle pour celles-ci.

Décider de la validité d'une QBF est un problème calculatoire difficile, considéré inabordable dans l'absolu puisque **PSPACE**-complet, mais envisageable en pratique. Cette complexité de calcul permet en contrepartie de résoudre des problèmes aux enjeux théoriques et industriels importants. Force est de constater que la plupart des procédures de l'état de l'art requièrent une entrée sous forme prénexe, voire sous une forme plus restrictive encore, alors que l'encodage des problèmes nécessite un langage plus riche. Pourtant, peu de travaux étudient la mise sous forme prénexe des QBF et tous arrivent à la conclusion que la stratégie utilisée durant cette phase affecte fortement le temps de résolution en pratique.La suppression des bi-implications et des ou-exclusifs, qui participent à l'expressivité du langage, est un point occulté. Notre première contribution est un ensemble d'équivalences et d'algorithmes qui permettent de traiter un motif particulier, *les résultats intermédiaires*. Ce motif apporte une alternative efficace en espace et en temps de résolution, à la suppression naïve des bi-implications et des ou-exclusifs. Il offre également de nouvelles possibilités de transformations dans différents fragments du langage **QBF**.

Notre deuxième contribution concerne la résolution en parallèle des QBF dans leur expressivité maximale. Nous présentons une nouvelle architecture pour la parallélisation du problème de validité des QBF dite « architecture de parallélisation syntaxique » par opposition aux architectures de parallélisation basée sur la sémantique des quantificateurs. Nous avons choisi d'implanter notre architecture selon un modèle maître/esclave avec pour oracle la procédure de décision `QSAT`. Cette implantation est déjà opérationnelle et la seule à notre connaissance à traiter en parallèle des QBF non CNF non prénexes. Le cadre proposé est suffisamment souple et général pour intégrer d'autres procédures du moment même où elles réalisent une élimination des quantificateurs d'une QBF à variables libres.

Parfois, la seule réponse quant à la validité d'une QBF ne suffit pas. Par exemple, dans le cas d'un jeu à deux joueurs, on souhaite connaître la *stratégie* qui permet de toujours gagner. Il s'agit d'une solution pour la QBF, présentée sous forme d'un *certificat* [Benedetti, 2005b]. Une QBF peut avoir plusieurs solutions, cet ensemble peut être *compilé* à partir

de la QBF dans une *base littérale* [Stéphan et Da Mota, 2008; Stéphan et Da Mota, 2009]. De manière annexe à nos travaux de thèse, nous avons réalisé une bibliothèque C++ permettant la représentation des certificats et des bases littérales et leur manipulation à l'aide de diagrammes BDD. Celle-ci a été intégrée à l'intérieur d'une procédure existante, `pQBF`.

Nos travaux ouvrent de nombreuses perspectives :

Certaines procédures classiques dédiées aux QBF utilisent en interne une structure de quantificateurs afin de minimiser leur portée. Ils sont extraits lors de la transformation dans le format d'entrée, puis sont réintroduits à l'intérieur de la formule lors de l'évaluation. Il est possible d'étendre notre travail sur les transformations formelles, en proposant une mise sous forme clausale sans passer par la forme prénexe. Ainsi, les quantificateurs n'ont pas à subir cet aller-retour, au mieux inutile, au pire destructeur d'informations.

Nous n'avons pas détaillé l'intégration de la *propagation booléenne* dans la description de notre architecture de calcul parallèle pour QBF. Actuellement elle n'est appliquée qu'en entrée de la procédure car elle ne permet pas de traiter les variables libres. Dans un premier temps, il faut ajouter le support des variables libres. Dans un second temps, il faut augmenter la décomposition de la formule par l'introduction de variables universelles. De cette manière, la propagation booléenne prendrait en compte la symétrie de la sémantique des quantificateurs.

La procédure `QSAT` comme oracle de notre architecture parallèle se révèle peu performante. L'implantation de nouveaux clients est une perspective prometteuse. Nous pensons notamment à un algorithme à la `quantor` qui, à l'opposé de `QSAT`, permettrait d'éliminer les variables liées d'une formule possédant beaucoup de variables libres. La mise à disposition du code source de la procédure `DepQBF` est aussi une opportunité à saisir, afin d'intégrer un algorithme de recherche performant dans un nouveau client de calcul. Plus nos disposerons de clients différents, plus il sera possible de choisir le traitement adapté à une sous-partie du problème. Puis, à la manière des procédures SAT avec un portfolio, il serait intéressant de faire travailler nos clients en concurrence par moment.

Comme nous l'avons fait ressentir lors de sa description, notre architecture parallèle nécessite encore beaucoup d'optimisations. L'élaboration de nouvelles stratégies de découpage syntaxique est un aspect prometteur. En effet, notre stratégie actuelle ne permet pas d'obtenir un degré de parallélisme suffisant et nous oblige trop souvent à avoir recours au découpage sémantique. Ce dernier, peut aussi être grandement amélioré sur plusieurs points. D'une part, le choix des variables doit être affiné, d'autre part quelques propagations permettraient d'élaguer l'espace de recherche, nous pensons notamment à la propagation booléenne. Il existe des heuristiques guidant le choix des variables pour les formules sous CNF. Sous forme quelconque, la seule solution est de calculer la projection sous CNF afin de réaliser les dénombrements nécessaires. Or, avec une QBF dans un fragment du langage plus expressif, il y a probablement des informations présentes permettant de mieux guider la recherche.

Selon l'architecture matérielle de calcul, la communication peut avoir un coût très important. Les deux procédures parallèles pour QBF les plus récentes intègrent l'apprentissage afin d'élaguer l'espace de recherche. Aucune des deux procédures ne maintient

son efficacité au delà de quelques threads à cause du surcoût en communications. Notre approche par découpage syntaxique, nous semble intéressante dans l'optique du calcul massivement distribué et c'est pour cela que nous souhaitons limiter les découpages sémantiques. Nous partons de l'hypothèse que les fragments syntaxiques que nous traitons en parallèle n'ont pas besoin de communiquer ce qu'ils apprennent sur leur propre structure. L'implantation du *backjumping* et l'apprentissage local est de fait, une perspective prometteuse. Il faut ensuite envisager un apprentissage global, chargé de communiquer les informations apprises sur les variables libres communes aux différents sous-problèmes. Cet apprentissage global, pourra être couplé à une politique de redémarrage, technique qui a fait le succès des solveurs SAT. Une dernière perspective concerne l'implantation d'autres modèles parallèles, comme le modèle pair-à-pair, potentiellement plus souple pour les grandes architectures de calcul.

Même dans une procédure séquentielle, le calcul d'un certificat pour une QBF n'est pas quelque chose d'intuitif. Comment rassembler toute la connaissance nécessaire à ce calcul dans une architecture parallèle ? Pourtant, dans certains cas ce certificat a plus de valeur que le résultat seul de la validité. Dans d'autre cas, il permet de vérifier la correction du résultat. Ce problème de vérification de certificat est un problème combinatoire complexe, puisque **coNP**-complet. L'ajout de clients dédiés au calcul et à la vérification des certificats, permettrait à notre architecture de traiter ce problème en parallèle. Au-delà des certificats, il serait intéressant d'intégrer la compilation de bases littérales à notre architecture car une base littérale regroupe toutes les solutions d'une QBF et permet leur extraction de façon efficace.

Il est admis que les problèmes disponibles sous CNF sont mal codés et qu'une procédure doit intégrer des pré-traitements comme `sQueezeBF` si elles veulent être compétitives. De plus, le fragment CNF ne capture pas la symétrie présente dans la sémantique des quantificateurs. Nous avons présenté un nouveau format pour écrire des QBF dans leur expressivité maximale. Par conséquent, nous ne possédons que très peu de problèmes pour réaliser des tests. Malheureusement, il ne s'agit pas d'un simple problème de traduction, des informations ne sont pas présentes dans les autres formats. Un grand travail est nécessaire pour proposer des problèmes dans une forme la plus expressive possible afin que chacun puisse se ramener au fragment qu'il souhaite, l'inverse n'étant pas possible.

La problématique de représentation de connaissances avec des QBF n'est pas encore très abordée mais ouvre des perspectives intéressantes. Les *résultats intermédiaires* ne servent pas uniquement à effectuer des transformations formelles, il s'agit aussi d'un puissant outil pour programmer avec les QBF et c'est pour ça que nous l'avons remarqué. Il serait intéressant d'étudier ce concept avec des formalismes plus riches comme les problèmes de satisfaction de contraintes quantifiées par exemple.

Annexes

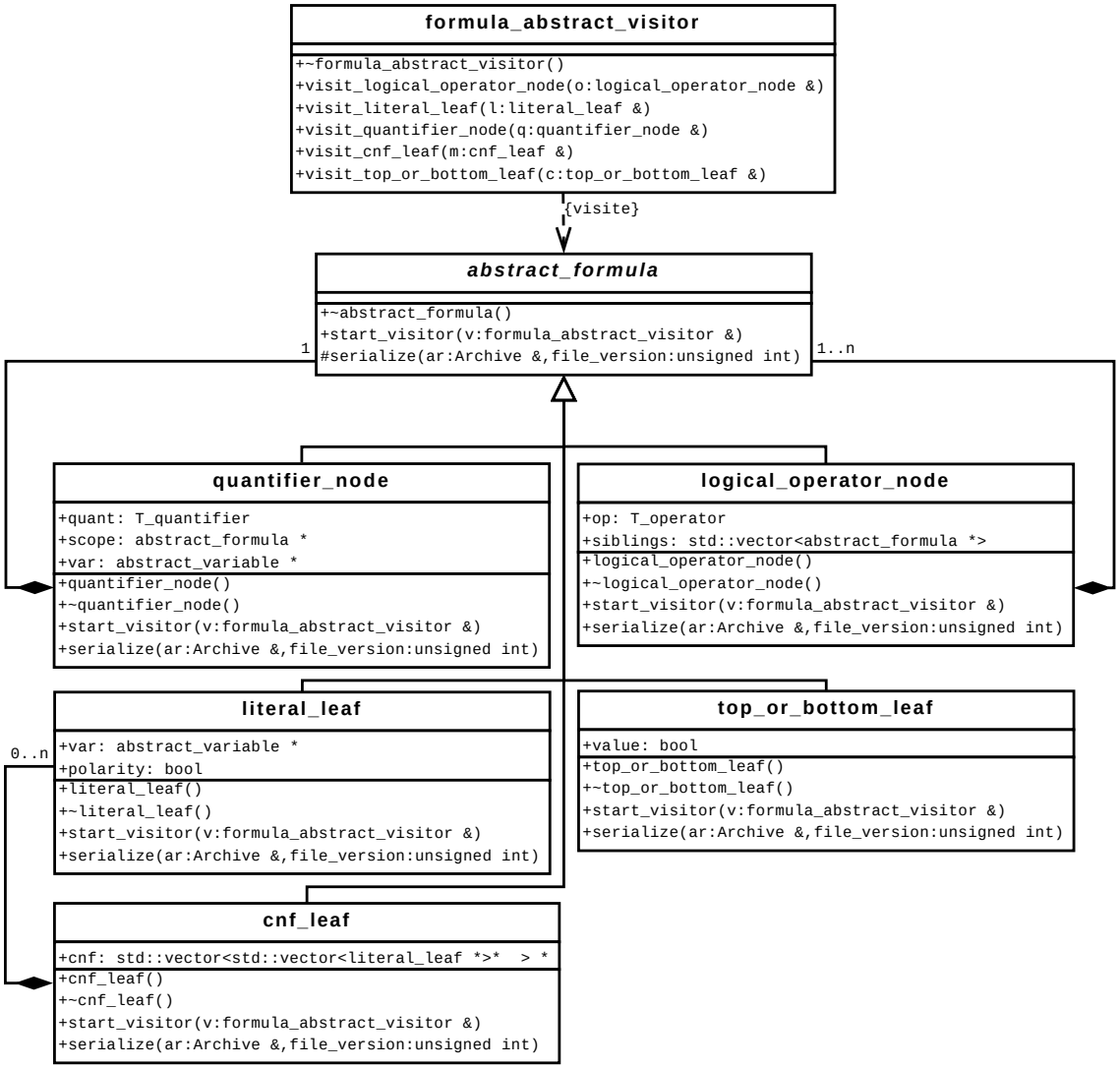


FIGURE 20 – Diagramme UML des formules.

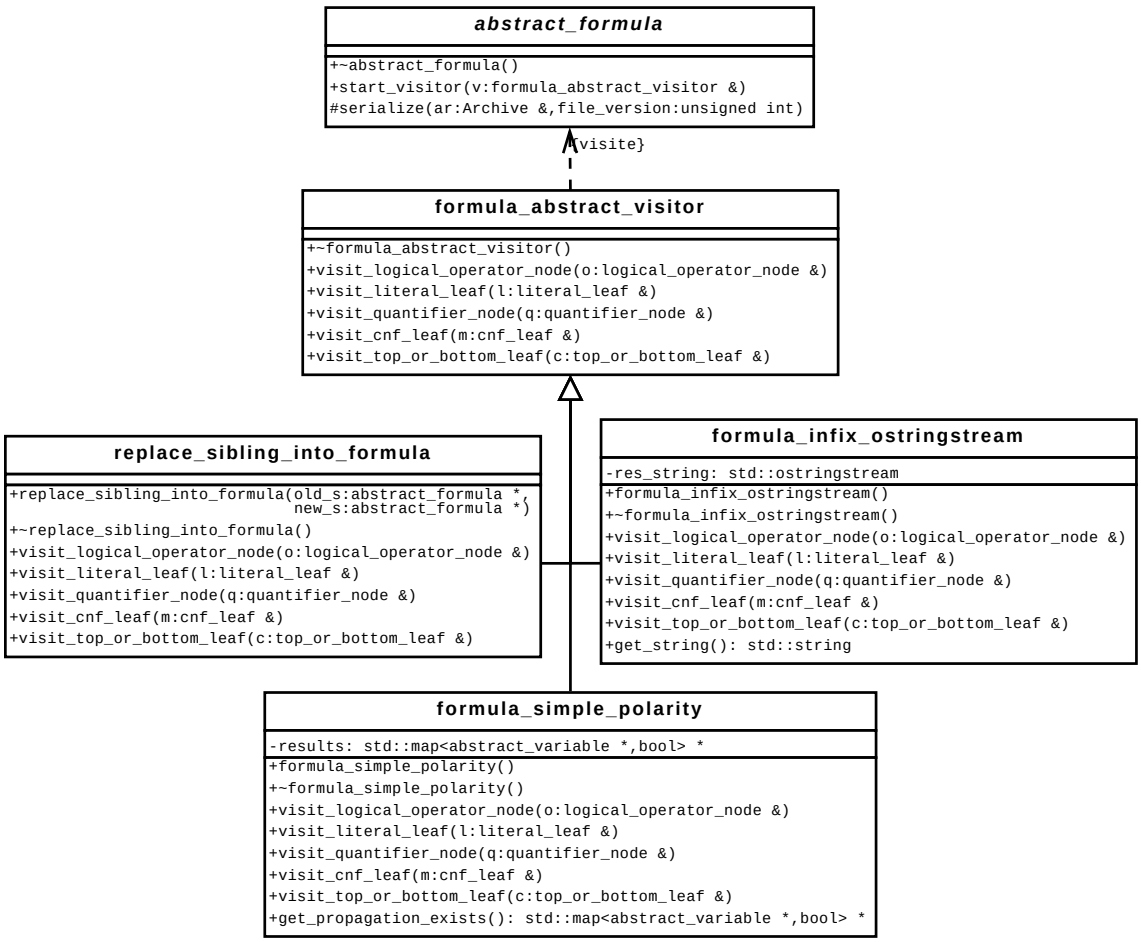


FIGURE 21 – Diagramme UML des visiteurs.

Liste des figures

Liste des tableaux

Références bibliographiques

[Akers, 1978] cité p. 33
Sheldon B. Akers. Binary Decision Diagrams. *IEEE Trans. Computers*, 27(6) :509–516, 1978.

[Ansótegui *et al.*, 2005] cité p. 2, 22, 22, 29, 42, 55
Carlos Ansótegui, Carla P. Gomes, and Bart Selman. The Achilles' Heel of QBF. In *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, AAAI 2005*, pages 275–281, 2005.

[Aspvall *et al.*, 1979] cité p. 38
Bengt Aspvall, Michael F. Plass, and Robert Endre Tarjan. A Linear-Time Algorithm for Testing the Truth of Certain Quantified Boolean Formulas. *Inf. Process. Lett.*, 8(3) :121–123, 1979.

[Audemard *et al.*, 2007a] cité p. 38
Gilles Audemard, Saïd Jabbour, and Lakhdar Saïs. Efficient Symmetry Breaking Predicates for Quantified Boolean Formulae. In *Workshop on Symmetry and Constraint Satisfaction Problems*, pages 14–21, 2007.

[Audemard *et al.*, 2007b] cité p. 38
Gilles Audemard, Saïd Jabbour, and Lakhdar Saïs. Symmetry Breaking in Quantified Boolean Formulae. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence, IJCAI 2007*, pages 2262–2267, 2007.

[Audemard et Saïs, 2004] cité p. 38
Gilles Audemard and Lakhdar Saïs. SAT Based BDD Solver for Quantified Boolean Formulas. In *16th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2004*, pages 82–89, 2004.

[Audemard et Saïs, 2005] cité p. 38
Gilles Audemard and Lakhdar Saïs. A Symbolic Search Based Approach for Quantified Boolean formulas. In *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005*, pages 16–30, 2005.

[Ayari *et al.*, 1999] cité p. 60, 112
Abdelwaheb Ayari, David A. Basin, and Stefan Friedrich. Structural and Behavioral Modeling with Monadic Logics. In *ISMVL*, pages 142–151, 1999.

[Ayari et Basin, 2000] cité p. 20
Abdelwaheb Ayari and David A. Basin. Bounded Model Construction for Monadic Second-Order Logics. In *Computer Aided Verification, 12th International Conference, CAV 2000*, pages 99–112, 2000.

[Ayari et Basin, 2002] cité p. 36, 38, 42, 60, 112
A. Ayari and David A. Basin. Qubos : Deciding Quantified Boolean Logic using Propositional Satisfiability Solvers. In *Formal Methods in Computer-Aided Design, Fourth International Conference, FMCAD 2002*. Springer-Verlag, 2002.

[Bacchus et Winter, 2003] cité p. 35
Fahiem Bacchus and Jonathan Winter. Effective Preprocessing with Hyper-Resolution and Equality Reduction. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003*, pages 341–355, 2003.

[Benedetti *et al.*, 2007] cité p. 22, 55
Marco Benedetti, Arnaud Lallouet, and Jérémie Vautard. QCSP Made Practical by Virtue of Restricted Quantification. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence, IJCAI 2007*, pages 38–43, 2007.

[Benedetti et Mangassarian, 2008] cité p. 1, 20, 22, 24, 29, 29, 55
Marco Benedetti and Hratch Mangassarian. QBF-Based Formal Verification : Experience and Perspectives. *Journal on Satisfiability, Boolean Modeling and Computation*, 5 :133–191, 2008.

[Benedetti, 2004] cité p. 37, 38
Marco Benedetti. Evaluating QBFs via Symbolic Skolemization. In *Logic for Programming, Artificial Intelligence, and Reasoning, 11th International Conference, LPAR 2004*, pages 285–300, 2004.

[Benedetti, 2005a] cité p. 65
Marco Benedetti. Quantifier Trees for QBFs. In *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005*, pages 378–385, 2005.

[Benedetti, 2005b] cité p. 35, 38, 115
Marco Benedetti. sKizzo : A Suite to Evaluate and Certify QBFs. In *Automated Deduction - CADE-20, 20th International Conference on Automated Deduction*, pages 369–376, 2005.

[Benedetti, 2006] cité p. 38
Marco Benedetti. Abstract Branching for Quantified Formulas. In *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, AAAI 2006*, 2006.

[Berre *et al.*, 2004] cité p. 28
Daniel Le Berre, Massimo Narizzano, Laurent Simon, and Armando Tacchella. The Second QBF Solvers Comparative Evaluation. In *Theory and Applications of Satisfiability Testing, 7th International Conference, SAT 2004*, pages 376–392, 2004.

[Besnard *et al.*, 2009] cité p. 1, 20
Philippe Besnard, Anthony Hunter, and Stefan Woltran. Encoding Deductive Argumentation in Quantified Boolean Formulae. *Artificial Intelligence*, 173(15) :1406 – 1423, 2009.

[Biere *et al.*, 1999a] cité p. 1
Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Masahiro Fujita, and Yunshan Zhu. Symbolic Model Checking Using SAT Procedures instead of BDDs. In *DAC*, pages 317–320, 1999.

[Biere *et al.*, 1999b] cité p. 23
Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs. In *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS 1999*, pages 193–207, 1999.

[Biere, 2004] cité p. 20, 38, 63, 65
Armin Biere. Resolve and Expand. In *Theory and Applications of Satisfiability Testing, 7th International Conference, SAT 2004*, pages 59–70, 2004.

[Browne *et al.*, 1988] cité p. 23
Michael C. Browne, Edmund M. Clarke, and Orna Grumberg. Characterizing Finite Kripke Structures in Propositional Temporal Logic. *Theoretical Computer Science*, 59 :115–131, 1988.

[Bryant *et al.*, 2003] cité p. 20
Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. Convergence Testing in Term-Level Bounded Model Checking. In *Correct Hardware Design and Verification Methods, 12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003*, pages 348–362, 2003.

[Bubeck et Büning, 2008] cité p. 38
Uwe Bubeck and Hans Kleine Büning. Models and quantifier elimination for quantified Horn formulas. *Discrete Applied Mathematics*, 156(10) :1606–1622, 2008.

[Büning *et al.*, 1995] cité p. 1, 28, 32, 38, 86
Hans Kleine Büning, Marek Karpinski, and Andreas Flögel. Resolution for Quantified Boolean Formulas. *Inf. Comput.*, 117(1) :12–18, 1995.

[Büning *et al.*, 2007] cité p. 37
Hans Kleine Büning, K. Subramani, and Xishun Zhao. Boolean Functions as Models for Quantified Boolean Formulas. *Journal of Automated Reasoning*, 39(1) :49–75, 2007.

[Büning et Zhao, 2004] cité p. 38
Hans Kleine Büning and Xishun Zhao. On Models for Quantified Boolean Formulas. In *Logic versus Approximation, Essays Dedicated to Michael M. Richter on the Occasion of his 65th Birthday*, pages 18–32, 2004.

[Cadoli *et al.*, 1997] cité p. 1, 28, 30, 38, 86, 86
Marco Cadoli, Andrea Giovanardi, and Marco Schaerf. Experimental Analysis of the Computational Cost of Evaluating Quantified Boolean Formulae. In *Advances in Artificial Intelligence, 5th Congress of the Italian Association for Artificial Intelligence, AI*IA 1997*, pages 207–218, 1997.

[Cadoli *et al.*, 1998] cité p. 1, 28, 30, 38, 86
Marco Cadoli, Andrea Giovanardi, and Marco Schaerf. An Algorithm to Evaluate Quantified Boolean Formulae. In *AAAI/IAAI*, pages 262–267, 1998.

[Chandrasekar et Hsiao, 2005] cité p. 38
Kameshwar Chandrasekar and Michael S. Hsiao. Q-PREZ : QBF Evaluation Using Partition, Resolution and Elimination with ZBDDs. In *18th International Conference on VLSI Design (VLSI Design 2005), with the 4th International Conference on Embedded Systems Design*, pages 189–194, 2005.

[Chatalic et Simon, 2000a] cité p. 33
Philippe Chatalic and Laurent Simon. Multi-resolution on compressed sets of clauses. In *12th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2000*, pages 2–10, 2000.

[Chatalic et Simon, 2000b] cité p. 33
Philippe Chatalic and Laurent Simon. ZRES : The Old Davis-Putman Procedure Meets ZBDD. In *17th International Conference on Automated Deduction, CADE 2000*, pages 449–454, 2000.

[Chatalic et Simon, 2001] cité p. 33
Philippe Chatalic and Laurent Simon. Multiresolution for SAT Checking. *International Journal on Artificial Intelligence Tools*, 10(4) :451–481, 2001.

[Cook, 1971] cité p. 17
Stephen A. Cook. The Complexity of Theorem-Proving Procedures. In *Conference Record of Third Annual ACM Symposium on Theory of Computing, STOC*, pages 151–158, 1971.

[Coste-Marquis *et al.*, 2005] cité p. 38
Sylvie Coste-Marquis, Daniel Le Berre, Florian Letombe, and Pierre Marquis. Propositional Fragments for Knowledge Compilation and Quantified Boolean Formulae. In *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, AAAI 2005*, pages 288–293, 2005.

[Coste-Marquis *et al.*, 2006a] cité p. 38
Sylvie Coste-Marquis, Daniel Le Berre, Florian Letombe, and Pierre Marquis. Complexity Results for Quantified Boolean Formulae Based on Complete Propositional Languages. *JSAT*, 1(1) :61–88, 2006.

[Coste-Marquis *et al.*, 2006b] cité p. 37
Sylvie Coste-Marquis, Hélène Fargier, Jérôme Lang, Daniel Le Berre, and Pierre Marquis. Representing Policies for Quantified Boolean Formulae. In *Proceedings, Tenth International Conference on Principles of Knowledge Representation and Reasoning, KR 2006*, pages 286–297, 2006.

[Crawford et Baker, 1994] cité p. 1
James M. Crawford and Andrew B. Baker. Experimental Results on the Application of Satisfiability Algorithms to Scheduling Problems. In *AAAI*, pages 1092–1097, 1994.

[Da Mota *et al.*, 2007] cité p. 38
Benoit Da Mota, Igor Stéphan, and Pascal Nicolas. From (quantified) Boolean Formulae to Answer Set Programming. In *Proceedings of the 7th International Answer Set Programming Workshop, ASP'07*, 2007.

[Da Mota *et al.*, 2008] cité p. 41
Benoit Da Mota, Igor Stéphan, and Pascal Nicolas. Une mise sous forme prénexe préservant les résultats intermédiaires pour les formules booléennes quantifiées. In *Journées Nationales de l'IA Fondamentale*, 2008.

[Da Mota *et al.*, 2009a] cité p. 41
Benoit Da Mota, Igor Stéphan, and Pascal Nicolas. A New Prenexing Strategy for Quantified Boolean Formulae with Bi-Implications. In *Sixth International Workshop on Constraints in Formal Verification*, 2009.

[Da Mota *et al.*, 2009b] cité p. 41
Benoit Da Mota, Igor Stéphan, and Pascal Nicolas. Une nouvelle stratégie de mise sous forme prénexe pour des formules booléennes quantifiées avec bi-implications. *Revue I3 (Information - Interaction - Intelligence)*, 9(2), 2009.

[Da Mota *et al.*, 2010a] cité p. 91
Benoit Da Mota, Pascal Nicolas, and Igor Stéphan. A new Parallel Architecture for QBF Tools. In *Proceedings of the 2010 International Conference on High Performance Computing & Simulation, HPCS 2010*, pages 324–330. IEEE, 2010.

[Da Mota *et al.*, 2010b] cité p. 91
Benoit Da Mota, Pascal Nicolas, and Igor Stéphan. Une nouvelle architecture parallèle pour le problème de validité des QBF. In *Sixièmes Journées Francophones de Programmation par Contraintes, JFPC'10*, 2010.

[Da Mota, 2009] cité p. 41
Benoit Da Mota. Équivalences et forme prénexe pour les formules booléennes quantifiées. In *Rencontre des Jeunes Chercheurs en Intelligence Artificielle*, 2009.

[Da Mota, à paraître] cité p. 41
Benoit Da Mota. Équivalences et forme prénexe pour les formules booléennes quantifiées. *Revue d'Intelligence Artificielle*, à paraître.

[Davis *et al.*, 1962] cité p. 30
Martin Davis, George Logemann, and Donald W. Loveland. A Machine Program for Theorem-Proving. *Commun. ACM*, 5(7) :394–397, 1962.

[Davis et Putnam, 1960] cité p. 33
Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *Journal ACM*, 7(3) :201–215, 1960.

[de la Tour, 1992] cité p. 42
Thierry Boy de la Tour. An Optimality Result for Clause Form Translation. *J. Symb. Comput.*, 14(4) :283–302, 1992.

[Dechter et Rish, 1994] cité p. 33
Rina Dechter and Irina Rish. Directional Resolution : The Davis-Putnam Procedure, Revisited. In *KR*, pages 134–145, 1994.

[Demaine, 2001] cité p. 1, 20, 20
Erik D. Demaine. Playing Games with Algorithms : Algorithmic Combinatorial Game Theory. In *Mathematical Foundations of Computer Science 2001, 26th International Symposium, MFCS 2001*, pages 18–32, 2001.

[Dereniowski, 2008] cité p. 21
Dariusz Dereniowski. Phutball is PSPACE-hard. *CoRR*, abs/0804.1777, 2008.

[Dershowitz *et al.*, 2005] cité p. 20, 23, 28, 28
Nachum Dershowitz, Ziyad Hanna, and Jacob Katz. Bounded Model Checking with QBF. In *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005*, pages 408–414, 2005.

[Egly *et al.*, 2000] cité p. 1, 20
Uwe Egly, Thomas Eiter, Hans Tompits, and Stefan Woltran. Solving Advanced Reasoning Tasks Using Quantified Boolean Formulas. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence, AAAI/IAAI 2000*, pages 417–422, 2000.

[Egly *et al.*, 2001] cité p. 1, 20
Uwe Egly, Thomas Eiter, Volker Klotz, Hans Tompits, and Stefan Woltran. Computing Stable Models with Quantified Boolean Formulas : Some Experimental Results. In *Answer Set Programming, Towards Efficient and Scalable Knowledge Representation and Reasoning, Proceedings of the 1st Intl. ASP'01 Workshop*, 2001.

[Egly *et al.*, 2003] cité p. 29, 42, 42, 44, 56, 61
Uwe Egly, Martina Seidl, Hans Tompits, Stefan Woltran, and Michael Zolda. Comparing Different Prenexing Strategies for Quantified Boolean Formulas. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003*, pages 214–228, 2003.

[Egly *et al.*, 2009] cité p. 38, 42
Uwe Egly, Martina Seidl, and Stefan Woltran. A Solver for QBFs in Negation Normal Form. *Constraints*, 14(1) :38–79, 2009.

[Egly et Tompits, 2001] cité p. 1, 20
Uwe Egly and Hans Tompits. Proof-Complexity Results for Nonmonotonic Reasoning. *ACM Trans. Comput. Log.*, 2(3) :340–387, 2001.

[Egly, 1996] cité p. 42
Uwe Egly. On Different Structure-Preserving Translations to Normal Form. *Journal of Symbolic Computation*, 22(2) :121–142, 1996.

[Even et Tarjan, 1976] cité p. 21
Shimon Even and Robert Endre Tarjan. A Combinatorial Problem Which Is Complete in Polynomial Space. *J. ACM*, 23(4) :710–719, 1976.

[Feldmann *et al.*, 2000] cité p. 38, 86, 86, 87
Rainer Feldmann, Burkhard Monien, and Stefan Schamberger. A Distributed Algorithm to Evaluate Quantified Boolean Formulae. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence, AAAI/IAAI 2000*, pages 285–290, 2000.

[Galil, 1977] cité p. 32
Zvi Galil. On the Complexity of Regular Resolution and the Davis-Putnam Procedure. *Theoretical Computer Science*, 4(1) :23–46, 1977.

[Gallier, 1985] cité p. 6, 32
J.H. Gallier. *Logic for computer science : foundations of automatic theorem proving*. Harper & Row Publishers, Inc., 1985.

[Gamma *et al.*, 1994] cité p. 94
E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Assison-Wesley Professional Computing Series, 1994.

[Garey et Johnson, 1979] cité p. 14
Michael R. Garey and David S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, 1979.

[Gent *et al.*, 2003] cité p. 38
Ian P. Gent, Holger H. Hoos, Andrew G. D. Rowley, and Kevin Smyth. Using Stochastic Local Search to Solve Quantified Boolean Formulae. In *Principles and Practice of Constraint Programming, 9th International Conference, CP 2003*, pages 348–362, 2003.

[Gent et Rowley, 2003] cité p. 20, 21
Ian Gent and Andrew Rowley. Encoding Connect-4 using Quantified Boolean Formulae. Technical Report APES-68-2003, APES Research Group, July 2003.

[Gent et Rowley, 2005] cité p. 38
Ian P. Gent and Andrew G. D. Rowley. Local and Global Complete Solution Learning Methods for QBF. In *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005*, pages 91–106, 2005.

[GhasemZadeh *et al.*, 2004] cité p. 38
Mohammad GhasemZadeh, Volker Klotz, and Christoph Meinel. Embedding Memoization to the Semantic Tree Search for Deciding QBFs. In *Advances in Artificial Intelligence, 17th Australian Joint Conference on Artificial Intelligence, AI 2004*, pages 681–693, 2004.

[Giunchiglia *et al.*, 2001a] cité p. 28, 108
E. Giunchiglia, M. Narizzano, and A. Tacchella. Quantified Boolean Formulas satisfiability library (QBFLIB), 2001. `www.qbflib.org`.

[Giunchiglia *et al.*, 2001b] cité p. 28
Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. Backjumping for Quantified Boolean Logic Satisfiability. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001*, pages 275–281, 2001.

[Giunchiglia *et al.*, 2001c] cité p. 38
Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. QUBE : A System for Deciding Quantified Boolean Formulas Satisfiability. In *Automated Reasoning, First International Joint Conference, IJCAR 2001*, pages 364–369, 2001.

[Giunchiglia *et al.*, 2002] cité p. 28, 35
Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. Learning for Quantified Boolean Logic Satisfiability. In *AAAI/IAAI*, pages 649–654, 2002.

[Giunchiglia *et al.*, 2003] cité p. 28
Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. Backjumping for Quantified Boolean Logic satisfiability. *Artificial Intelligence*, 145(1-2) :99–120, 2003.

[Giunchiglia *et al.*, 2004a] cité p. 28, 28
Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. Monotone Literals and Learning in QBF Reasoning. In *Principles and Practice of Constraint Programming, 10th International Conference, CP 2004*, pages 260–273, 2004.

[Giunchiglia *et al.*, 2004b] cité p. 28, 63
Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. QuBE++ : An Efficient QBF Solver. In *Formal Methods in Computer-Aided Design, 5th International Confrence, FMCAD 2004*, pages 201–213, 2004.

[Giunchiglia *et al.*, 2006a] cité p. 28
Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. Clause/Term Resolution and Learning in the Evaluation of Quantified Boolean Formulas. *Journal of Artificial Intelligence Research, JAIR*, 26 :371–416, 2006.

[Giunchiglia *et al.*, 2006b] cité p. 28
Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. Quantifier structure in search based procedures for QBFs. In *Proceedings of the Conference on Design, Automation and Test in Europe, DATE 2006*, pages 812–817, 2006.

[Giunchiglia *et al.*, 2006c] cité p. 42, 65
Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. Quantifier Structure in Search Based Procedures for QBFs. In *Proceedings of the conference on Design, automation and test in Europe, DATE 2006*, pages 812–817, 2006.

[Giunchiglia *et al.*, 2007] cité p. 28
Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. Quantifier Structure in Search-Based Procedures for QBFs. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 26(3) :497–507, 2007.

[Giunchiglia *et al.*, 2010] cité p. 28, 35
Enrico Giunchiglia, Paolo Marin, and Massimo Narizzano. sQueezeBF : An Effective Preprocessor for QBFs Based on Equivalence Reasoning. In *Theory and Applications of Satisfiability Testing, 13th International Conference, SAT 2010*, pages 85–98, 2010.

[Goultiaeva *et al.*, 2009a] cité p. 38
Alexandra Goultiaeva, Vicki Iverson, and Fahiem Bacchus. A Compact Representation for Syntactic Dependencies in QBFs. In *Theory and Applications of Satisfiability Testing, 12th International Conference, SAT 2009*, pages 398–411, 2009.

[Goultiaeva *et al.*, 2009b] cité p. 38
Alexandra Goultiaeva, Vicki Iverson, and Fahiem Bacchus. Beyond CNF : A Circuit-Based QBF Solver. In *Theory and Applications of Satisfiability Testing, 12th International Conference, SAT 2009*, pages 412–426, 2009.

[Haken, 1985] cité p. 32
Armin Haken. The Intractability of Resolution. *Theoretical Computer Science*, 39 :297–308, 1985.

[Hamadi *et al.*, 2009] cité p. 112
Youssef Hamadi, Saïd Jabbour, and Lakhdar Saïs. ManySAT : a new Parallel SAT solver. *Journal of Satisfiability Boolean Modeling and Computation*, 2009.

[Hearn, 2005] cité p. 21
Robert A. Hearn. Amazons is PSPACE-complete. *CoRR*, abs/cs/0502013, 2005.

[ichi Minato, 1993] cité p. 33
Shin ichi Minato. Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems. In *DAC*, pages 272–277, 1993.

[Iwata et Kasai, 1994] cité p. 21
Shigeki Iwata and Takumi Kasai. The Othello game on an n*n board is PSPACE-complete. *Theoretical Computer Science*, 123(2) :329–340, 1994.

[Jabbour, 2008] cité p. 38, 112
Saïd Jabbour. *De la Satisfiabilité Propositionnelle aux Formules Booléennes Quantifiées*. PhD thesis, Université d'Artois, CRIL, Lens, France, décembre 2008.

[Johnson, 1988] cité p. 81
Eric E. Johnson. Completing an MIMD multiprocessor taxonomy. *SIGARCH Comput. Archit. News*, 16(3) :44–47, 1988.

[Jussila et Biere, 2006] cité p. 23, 28, 28
Toni Jussila and Armin Biere. Compressing BMC Encodings with QBF. In *Proceedings of the 4th International Workshop on Bounded Model Checking (BMC'06)*, 2006.

[Kautz et Selman, 1992] cité p. 1
Henry A. Kautz and Bart Selman. Planning as Satisfiability. In *ECAI*, pages 359–363, 1992.

[Kautz et Selman, 1998] cité p. 1
Henry A. Kautz and Bart Selman. The Role of Domain-Specific Knowledge in the Planning as Satisfiability Framework. In *AIPS*, pages 181–189, 1998.

[Kim *et al.*, 2000] cité p. 1
Joonyoung Kim, Jesse Whittemore, Karem A. Sakallah, and João P. Marques Silva. On Applying Incremental Satisfiability to Delay Fault Testing. In *Design, Automation and Test in Europe, DATE 2000*, pages 380–384, 2000.

[Larrabee, 1992] cité p. 1
Tracy Larrabee. Test pattern generation using Boolean satisfiability. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 11(1) :4–15, 1992.

[Lee, 1959] cité p. 33
C.Y. Lee. Representation of Switching Circuits by Binary Decision Programs. *Bell Systems Technical Journal*, 38 :985–999, 1959.

[Letombe, 2005] cité p. 38
Florian Letombe. *De la validité des formules booléennes quantifiées : étude de complexité et exploitation de classes traitables au sein d'un prouveur QBF*. PhD thesis, Université d'Artois, CRIL, Lens, France, décembre 2005.

[Letz, 2002] cité p. 35, 35, 38
Reinhold Letz. Lemma and Model Caching in Decision Procedures for Quantified Boolean Formulas. In *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference, TABLEAUX 2002*, pages 160–175, 2002.

[Lewis *et al.*, 2009a] cité p. 28, 86, 88, 88
Matthew D. T. Lewis, Paolo Marin, Tobias Schubert, Massimo Narizzano, Bernd Becker, and Enrico Giunchiglia. PaQuBE : Distributed QBF Solving with Advanced Knowledge Sharing. In *SAT*, pages 509–523, 2009.

[Lewis *et al.*, 2009b] cité p. 38, 86, 87
Matthew D.T. Lewis, Tobias Schubert, and Bernd Becker. QMiraXT - A Multithreaded QBF Solver. In *GI/ITG/GMM Workshop on Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, 2009.

[Lichtenstein et Sipser, 1980] cité p. 21
David Lichtenstein and Michael Sipser. GO Is Polynomial-Space Hard. *J. ACM*, 27(2) :393–401, 1980.

[Ling *et al.*, 2005] cité p. 20
Andrew C. Ling, Deshanand P. Singh, and Stephen Dean Brown. FPGA Logic Synthesis Using Quantified Boolean Satisfiability. In *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005*, pages 444–450, 2005.

[Lonsing et Biere, 2008] cité p. 38
Florian Lonsing and Armin Biere. Nenofex : Expanding NNF for QBF Solving. In *Theory and Applications of Satisfiability Testing, 11th International Conference, SAT 2008*, pages 196–210, 2008.

[Lonsing et Biere, 2009] cité p. 65
Florian Lonsing and Armin Biere. Efficiently Representing Existential Dependency Sets for Expansion-based QBF Solvers. *Electronic Notes in Theoretical Computer Science*, 251 :83–95, 2009.

[Lonsing et Biere, 2010a] cité p. 38
Florian Lonsing and Armin Biere. DepQBF : A Dependency-Aware QBF Solver (System Description). In *Pragmatics of SAT Workshop, POS'10*, 2010.

[Lonsing et Biere, 2010b] cité p. 38
Florian Lonsing and Armin Biere. Integrating Dependency Schemes in Search-Based QBF Solvers. In *Theory and Applications of Satisfiability Testing, 13th International Conference, SAT 2010*, pages 158–171, 2010.

[Mangassarian *et al.*, 2007] cité p. 20
Hratch Mangassarian, Andreas G. Veneris, Sean Safarpour, Marco Benedetti, and Duncan Smith. A Performance-Driven QBF-based Iterative Logic Array Representation with Applications to Verification, Debug and Test. In *International Conference on Computer-Aided Design, ICCAD 2007*, pages 240–245, 2007.

[Mangassarian *et al.*, 2010] cité p. 29, 29
Hratch Mangassarian, Andreas G. Veneris, and Marco Benedetti. Robust QBF Encodings for Sequential Circuits with Applications to Verification, Debug, and Test. *IEEE Trans. Computers*, 59(7) :981–994, 2010.

[Massacci et Marraro, 2000] cité p. 1
Fabio Massacci and Laura Marraro. Logical Cryptanalysis as a SAT Problem. *J. Autom. Reasoning*, 24(1/2) :165–203, 2000.

[Meyer et Stockmeyer, 1972] cité p. 1, 28
Albert R. Meyer and Larry J. Stockmeyer. The Equivalence Problem for Regular Expressions with Squaring Requires Exponential Space. In *13th Annual Symposium on Switching and Automata Theory, FOCS*, pages 125–129, 1972.

[Mneimneh et Sakallah, 2003] cité p. 20, 23, 28
Maher N. Mneimneh and Karem A. Sakallah. Computing Vertex Eccentricity in Exponentially Large Graphs : QBF Formulation and Solution. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003*, pages 411–425, 2003.

[Narizzano *et al.*, 2006] cité p. 28, 63
Massimo Narizzano, Luca Pulina, and Armando Tacchella. Report of the Third QBF Solvers Evaluation. *JSAT*, 2(1-4) :145–164, 2006.

[Pacheco, 1997] cité p. 84, 86
Peter S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publishers, Inc., 1997.

[Pan et Vardi, 2004a] cité p. 33, 34
Guoqiang Pan and Moshe Y. Vardi. Search vs. Symbolic Techniques in Satisfiability Solving. In *The Seventh International Conference on Theory and Applications of Satisfiability Testing, SAT 2004*, 2004.

[Pan et Vardi, 2004b] cité p. 20, 33, 38
Guoqiang Pan and Moshe Y. Vardi. Symbolic Decision Procedures for QBF. In *Principles and Practice of Constraint Programming, 10th International Conference, CP 2004*, pages 453–467, 2004.

[Papadimitriou, 1994] cité p. 1, 14, 15, 20, 112
Christos M. Papadimitriou. *Computational complexity*. Addison-Wesley, Reading, Massachusetts, 1994.

[Peschiera *et al.*, 2010] cité p. 28
Claudia Peschiera, Luca Pulina, Armando Tacchella, Uwe Bubeck, Oliver Kullmann, and Inês Lynce. The Seventh QBF Solvers Evaluation (QBFEVAL'10). In *Theory and Applications of Satisfiability Testing, 13th International Conference, SAT 2010*, pages 237–250, 2010.

[Pigorsch et Scholl, 2009] cité p. 38
Florian Pigorsch and Christoph Scholl. Exploiting structure in an AIG based QBF solver. In *Design, Automation and Test in Europe, DATE 2009*, pages 1596–1601, 2009.

[Plaisted *et al.*, 2003] cité p. 38, 42, 98, 99, 99, 100
David A. Plaisted, Armin Biere, and Yunshan Zhu. A satisfiability procedure for quantified Boolean formulae. *Discrete Applied Mathematics*, 130(2) :291–328, 2003.

[Plaisted et Greenbaum, 1986] cité p. 42
D. A. Plaisted and S. Greenbaum. A Structure-Preserving Clause Form Translation. *Journal of Symbolic Computation*, 2(3) :293–304, 1986.

[Potlapally *et al.*, 2007] cité p. 1
Nachiketh R. Potlapally, Anand Raghunathan, Srivaths Ravi, Niraj K. Jha, and Ruby B. Lee. Aiding Side-Channel Attacks on Cryptographic Software With Satisfiability-Based Analysis. *IEEE Trans. VLSI Syst.*, 15(4) :465–470, 2007.

[Pulina et Tacchella, 2007] cité p. 38, 38
Luca Pulina and Armando Tacchella. A Multi-engine Solver for Quantified Boolean Formulas. In *Principles and Practice of Constraint Programming, 13th International Conference, CP 2007*, pages 574–589, 2007.

[Pulina et Tacchella, 2008a] cité p. 38
Luca Pulina and Armando Tacchella. QuBIS : An (In)complete Solver for Quantified Boolean Formulas. In *Advances in Artificial Intelligence, 7th Mexican International Conference on Artificial Intelligence, MICAI 2008*, pages 34–43, 2008.

[Pulina et Tacchella, 2008b] cité p. 38
Luca Pulina and Armando Tacchella. Treewidth : A Useful Marker of Empirical Hardness in Quantified Boolean Logic Encodings. In *Logic for Programming, Artificial Intelligence, and Reasoning, 15th International Conference, LPAR 2008*, pages 528–542, 2008.

[Pulina et Tacchella, 2009] cité p. 35, 38
Luca Pulina and Armando Tacchella. A Structural Approach to Reasoning with Quantified Boolean Formulas. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence, IJCAI 2009*, pages 596–602, 2009.

[Pulina, 2010] cité p. 38, 38
Luca Pulina. Engineering portfolios of Machine Learning algorithms to solve complex tasks in Robotics and Automated Reasoning. *AI Commun.*, 23(1) :61–63, 2010.

[Reisch, 1981] cité p. 21
Stefan Reisch. Hex ist PSPACE-vollständig. *Acta Inf.*, 15 :167–191, 1981.

[Rintanen, 1999a] cité p. 20
Jussi Rintanen. Constructing Conditional Plans by a Theorem-Prover. *J. Artif. Intell. Res. (JAIR)*, 10 :323–352, 1999.

[Rintanen, 1999b] cité p. 38, 86
Jussi Rintanen. Improvements to the Evaluation of Quantified Boolean Formulae. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 99*, pages 1192–1197, 1999.

[Rintanen, 2001] cité p. 23
Jussi Rintanen. Partial Implicit Unfolding in the Davis-Putnam Procedure for Quantified Boolean Formulae. In *Logic for Programming, Artificial Intelligence, and Reasoning, 8th International Conference, LPAR 2001*, pages 362–376, 2001.

[Rish et Dechter, 2000] cité p. 33
Irina Rish and Rina Dechter. Resolution versus Search : Two Strategies for SAT. *Journal of Automated Reasoning*, 24(1/2) :225–275, 2000.

[Sabharwal *et al.*, 2006] cité p. 2, 22, 29, 38, 42, 55
Ashish Sabharwal, Carlos Ansótegui, Carla P. Gomes, Justin W. Hart, and Bart Selman. QBF Modeling : Exploiting Player Symmetry for Simplicity and Efficiency. In *Theory and Applications of Satisfiability Testing, 9th International Conference, SAT 2006*, pages 382–395, 2006.

[Samulowitz et Bacchus, 2005] cité p. 38
Horst Samulowitz and Fahiem Bacchus. Using SAT in QBF. In *Principles and Practice of Constraint Programming, 11th International Conference, CP 2005*, pages 578–592, 2005.

[Samulowitz et Bacchus, 2006] cité p. 35, 38
Horst Samulowitz and Fahiem Bacchus. Binary Clause Reasoning in QBF. In *Theory and Applications of Satisfiability Testing, 9th International Conference, SAT 2006*, pages 353–367, 2006.

[Schaefer, 1976] cité p. 21
Thomas J. Schaefer. Complexity of Decision Problems Based on Finite Two-Person Perfect-Information Games. In *Conference Record of the Eighth Annual ACM Symposium on Theory of Computing, STOC*, pages 41–49, 1976.

[Scholl et Becker, 2001] cité p. 20, 28
Christoph Scholl and Bernd Becker. Checking Equivalence for Partial Implementations. In *Proceedings of the 38th Design Automation Conference, DAC 2001*, pages 238–243, 2001.

[Selman et Kautz, 1993] cité p. 37
Bart Selman and Henry A. Kautz. Domain-Independent Extensions to GSAT : Solving Large Structured Satisfiability Problems. In *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI 1993*, pages 290–295, 1993.

[Snir *et al.*, 1995] cité p. 84, 84, 101
M. Snir, S. Otto, D. Walker, J. Dongarra, and S. Huss-Lederman. *MPI : The Complete Reference*. MIT Press, Cambridge, 1995.

[Staber et Bloem, 2007] cité p. 20
Stefan Staber and Roderick Bloem. Fault Localization and Correction with QBF. In *Theory and Applications of Satisfiability Testing, 10th International Conference, SAT 2007*, pages 355–368, 2007.

[Stéphan *et al.*, 2009] cité p. 37, 38
Igor Stéphan, Benoit Da Mota, and Pascal Nicolas. From (quantified) Boolean Formulae to Answer Set Programming. *J. Log. Comput.*, 19(4) :565–590, 2009.

[Stéphan et Da Mota, 2008] cité p. 115
Igor Stéphan and Benoit Da Mota. Base littérale et certificat pour les formules booléennes quantifiées. In *Quatrièmes Journées Francophones de Programmation par Contraintes, JFPC'08*, 2008.

[Stéphan et Da Mota, 2009] cité p. 115
Igor Stéphan and Benoit Da Mota. A Unified Framework for Certificate and Compilation for QBF. In *Logic and Its Applications, Third Indian Conference, ICLA 2009*, pages 210–223, 2009.

[Stéphan, 2006a] cité p. 38, 111
Igor Stéphan. Boolean Propagation Based on Literals for Quantified Boolean Formulae. In *17th European Conference on Artificial Intelligence, ECAI 2006*, pages 452–456, 2006.

[Stéphan, 2006b] cité p. 38
Igor Stéphan. Propagation logique pour les formules booléennes quantifiées. In *Actes des Deuxièmes Journées Francophones de Programmation par Contraintes, JFPC'06*, 2006.

[Stéphan, 2007] cité p. 38
Igor Stéphan. Une (presque) génération automatique d'un compilateur de tables de vérité vers un solveur pour formules booléennes quantifiées prénexes. In *Actes des Troisièmes Journées Francophones de Programmation par Contraintes, JFPC'07*, 2007.

[Stockmeyer et Meyer, 1973] cité p. 1, 1, 17, 28
Larry J. Stockmeyer and Albert R. Meyer. Word Problems Requiring Exponential Time : Preliminary Report. In *Conference Record of Fifth Annual ACM Symposium on Theory of Computing, STOC*, pages 1–9, 1973.

[Stockmeyer, 1977] cité p. 1, 18, 18, 28
L.J. Stockmeyer. The Polynomial-Time Hierarchy. *Theoretical Computer Science*, 3 :1–22, 1977.

[Tseitin, 1970] cité p. 34, 42, 47
G. S. Tseitin. On the Complexity of Derivation in Propositional Calculus. In A. O. Slisenko, editor, *Studies in Constructive Mathematics and Mathematical Logic, Part 2*, pages 115–125. Consultants Bureau, New York, 1970.

[Zhang et Malik, 2002a] cité p. 35, 38, 87
Lintao Zhang and Sharad Malik. Conflict driven learning in a quantified Boolean Satisfiability solver. In *Proceedings of the 2002 IEEE/ACM International Conference on Computer-aided Design, ICCAD 2002*, pages 442–449, 2002.

[Zhang et Malik, 2002b] cité p. 29, 35, 38, 42
Lintao Zhang and Sharad Malik. Towards a Symmetric Treatment of Satisfaction and Conflicts in Quantified Boolean Formula Evaluation. In *Principles and Practice of Constraint Programming, 8th International Conference, CP 2002*, pages 200–215, 2002.

[Zhang, 2006] cité p. 29, 38, 42
Lintao Zhang. Solving QBF by Combining Conjunctive and Disjunctive Normal Forms. In *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, AAAI*, pages 143–149, 2006.

Liste des publications personnelles

Revue internationale avec comité de sélection

[1] Igor Stéphan, Benoit Da Mota, and Pascal Nicolas. From (quantified) Boolean Formulae to Answer Set Programming. *J. Log. Comput.*, 19(4) :565–590, 2009.

Revues francophones avec comité de sélection

[2] Benoit Da Mota. Équivalences et forme prénexe pour les formules booléennes quantifiées. *Revue d'Intelligence Artificielle*, à paraître.

[3] Benoit Da Mota, Igor Stéphan, and Pascal Nicolas. Une nouvelle stratégie de mise sous forme prénexe pour des formules booléennes quantifiées avec bi-implications. *Revue I3 (Information - Interaction - Intelligence)*, 9(2), 2009.

Conférence internationale avec comité de sélection

[4] Igor Stéphan and Benoit Da Mota. A Unified Framework for Certificate and Compilation for QBF. In *Logic and Its Applications, Third Indian Conference, ICLA 2009*, pages 210–223, 2009.

Workshops internationaux avec comité de sélection

[5] Benoit Da Mota, Pascal Nicolas, and Igor Stéphan. A new Parallel Architecture for QBF Tools. In *Proceedings of the 2010 International Conference on High Performance Computing & Simulation, HPCS 2010*, pages 324–330. IEEE, 2010.

[6] Benoit Da Mota, Igor Stéphan, and Pascal Nicolas. A New Prenexing Strategy for Quantified Boolean Formulae with Bi-Implications. In *Sixth International Workshop on Constraints in Formal Verification*, 2009.

[7] Benoit Da Mota, Igor Stéphan, and Pascal Nicolas. From (quantified) Boolean Formulae to Answer Set Programming. In *Proceedings of the 7th International Answer Set Programming Workshop, ASP'07*, 2007.

Conférences francophones avec comité de sélection

[8] Benoit Da Mota, Pascal Nicolas, and Igor Stéphan. Une nouvelle architecture parallèle pour le problème de validité des QBF. In *Sixièmes Journées Francophones de Programmation par Contraintes, JFPC'10*, 2010.

[9] Benoit Da Mota. Équivalences et forme prénexe pour les formules booléennes quantifiées. In *Rencontre des Jeunes Chercheurs en Intelligence Artificielle*, 2009.

[10] Benoit Da Mota, Igor Stéphan, and Pascal Nicolas. Des formules booléennes (quantifiées) à la programmation par ensembles réponses. In *16e congrès francophone AFRIF-AFIA, Reconnaissance des Formes et Intelligence Artificielle*, 2008.

[11] Igor Stéphan and Benoit Da Mota. Base littérale et certificat pour les formules booléennes quantifiées. In *Quatrièmes Journées Francophones de Programmation par Contraintes*, 2008.

[12] Benoit Da Mota, Igor Stéphan, and Pascal Nicolas. Une mise sous forme prénexe préservant les résultats intermédiaires pour les formules booléennes quantifiées. In *Journées Nationales de l'IA Fondamentale*, 2008.

FORMULES BOOLÉENNES QUANTIFIÉES :
TRANSFORMATIONS FORMELLES ET CALCULS PARALLÈLES.

Résumé

De nombreux problèmes d'intelligence artificielle et de vérification formelle se ramènent à un test de validité d'une formule booléenne quantifiée (QBF). Mais, pour effectuer ce test les solveurs QBF actuels ont besoin d'une formule sous une forme syntaxique restrictive, comme la forme normale conjonctive ou la forme normale de négation. L'objectif de notre travail est donc de s'affranchir de ces contraintes syntaxiques fortes de manière à utiliser le langage des QBF dans toute son expressivité et nous traitons ce sujet de manière formelle et calculatoire.

Notre première contribution est un ensemble d'équivalences et d'algorithmes qui permettent de traiter un motif particulier, *les résultats intermédiaires*. Ce motif apporte une alternative efficace en espace et en temps de résolution, à la suppression naïve des bi-implications et des ou-exclusifs lors de la mise sous forme prénexe. Il offre également de nouvelles possibilités de transformations dans différents fragments du langage QBF.

Notre deuxième contribution est d'ordre calculatoire et a pour but d'exploiter la puissance des architectures de calcul parallèles afin de traiter des QBF sans restriction syntaxique. Nous élaborons donc une architecture innovante pour la parallélisation du problème de validité des QBF. Son originalité réside dans son architecture dite de « parallélisation syntaxique » par opposition aux architectures de parallélisation basée sur la sémantique des quantificateurs.

Mots-clés : Formules booléennes quantifiées, validité, transformations formelles, calculs parallèles, forme prénexe, découpage syntaxique.

QUANTIFIED BOOLEAN FORMULAE
FORMAL PROCESSINGS AND PARALLEL COMPUTATIONS.

Abstract

Many problems of artificial intelligence and formal verification can be reduced to a validity test of a quantified boolean formula (QBF). But, to perform this test, current QBF solvers need a formula in a restrictive syntactic form, as conjunctive normal form or negation normal form. The goal of our work is to get rid of these strong syntactic constraints in order to use the QBF language in its whole expressivity and we treat this subject in formal and computational manner.

Our first contribution is a set of equivalences and algorithms that can process a particular pattern, *the intermediate results*. This pattern provides an effective alternative in space and in time resolution, at the naive suppression of bi-implications and exclusive-or during the conversion in prenex form. It also offers new opportunities of transformations in different fragments of the QBF language.

Our second contribution is computational and its goal is to use the power of parallel computing architectures to deal with QBF without syntactic restriction. So we are developing an innovative architecture for parallelizing the QBF validity problem. Its originality lies in its architecture of "syntactic parallelization" versus parallelization based on the semantics of quantifiers.

Keywords : Quantified boolean formulae, validity, formal processings, parallel computations, prenex form, syntactic splitting.