Sat4j, un moteur libre de raisonnement en logique propositionnelle

Daniel Le Berre

UNIVERSITÉ D'ARTOIS

Sat4j, un moteur libre de raisonnement en logique propositionnelle

Habilitation à Diriger des Recherches

(Spécialité Informatique)

Université d'Artois

présentée et soutenue publiquement le 3 décembre 2010

par

Daniel Le Berre

Composition du jury

Rapporteurs :	Armin Biere	Professeur à l'Université Johannes Kepler
	Roberto Di Cosmo	Professeur à l'Université Paris Diderot
	Enrico Giunchiglia	Professeur à l'Université de Gênes
Examinateurs :	Éric Grégoire	Professeur à l'Université d'Artois
	Christophe Lecoutre	Professeur à l'Université d'Artois
	Joao Marques-Silva	Professeur à l'Université de Dublin
	Lakhdar Saïs	Professeur à l'Université d'Artois
Directeur :	Pierre Marquis	Professeur à l'Université d'Artois

Centre de Recherche en Informatique de Lens – CNRS UMR 8188
Université d'Artois, rue Jean Souvraz, S.P. 18 F-62307, Lens Cedex France
Secrétariat : Tél.: +33 (0)3 21 79 17 23 – Fax : +33 (0)3 21 79 17 70
http://www.cril.fr

Table des matières

Table des figures

Liste des tableaux

Remerciements

I would like first to thank the reviewers and external members of the committee for accepting to be part of it. Each of them has influenced in some way my research directions during that decade. Joao Marques-Silva and Armin Biere have both contributed to change the scope of applicability of SAT technology : the former by creating the conflict driven clause learning infrastructure in GRASP, and the latter by both pioneering the notion of bounded model checking and by being one of the most prolific SAT solver designer. I owe them indirectly the opportunity to run the SAT competition. Enrico Giunchiglia is one of the leading figure on the practical resolution of QBF. Our work on that topic, both in theory and in practice, has been greatly influenced by his own work. Roberto Di Cosmo, project leader of the Mancoosi European project, reflects in that committee one of the latest topics I have been interested in, namely dependency management problems. Our work on Eclipse dependency management problems is a direct application of the scientific results of his previous European Project, EDOS, and has a direct application in Mancoosi, materialized by the open source p2cudf solver. Roberto Di Cosmo is also a leading figure in the free and open source software community, to which we contributed Sat4j. Je tiens a remercier chaleureusement Lakhdar, Christophe, Pierre et Eric qui représentent le CRIL et l'Université d'Artois dans ce jury, avec une mention spéciale pour Pierre qui a accepté de diriger ce travail de synthèse de mon activité de recherche.

Se lancer dans une nouvelle piste de recherche est souvent catalysé par une rencontre. Ce fut le cas pour la logique du choix qualitatif avec Gerhard Brewka, rencontré lors de mon PostDoc en Australie. Ce fut aussi le cas pour la compétition SAT, fruit d'une rencontre avec Laurent Simon venu présenter SatEx à SAT 2001, et dont la co-organisation me fut proposée par John Franco et Hans van Maaren, rencontrés aussi à SAT 2001. Les problèmes d'optimisation pseudo-booléens me furent présentés par Joao-Marques Silva lors d'un de notre premier projet de coopération autour d'OpenSAT. Denis Caromel de l'Université de Nice, m'a présenté le consortium ObjectWeb qui offre par sa forge les outils collaboratifs nécessaires au développement ouvert de Sat4j. Les problèmes de gestion de dépendances Eclipse me furent présentés par Pascal Rapicault. D'un point de vue plus général, Mary-Anne Williams m'a ouvert sur le côté mondial et collaboratif de la recherche, et m'a permis de comprendre une vision anglo-saxonne de cette activité.

Ces rencontres se font aussi localement, au CRIL. J'ai eu la joie de travailler avec divers membres du CRIL depuis 10 ans : Anne, Florian, Gilles, Karima, Lakhdar, Meltem, Olivier, Pierre, Salem, Souhila, Sylvie. Ces collaborations continueront et je l'espère s'élargiront à d'autres membres du CRIL. Sans oublier mes anciens collègues de l'IRIT, Jérôme et Hélène.

La plupart des travaux que j'ai réalisés ont été possibles parce que le CRIL nous offre une certaine liberté dans notre méthode de travail et nos sujets de recherche, et nous fournit les moyens de les mener à bien. Je suis redevable à ce titre à notre directeur, Eric

Grégoire, qui a fondé ce laboratoire et se donne corps et âme pour le développer. Je suis aussi redevable à tous les collègues du CRIL qui m'ont aidé durant cette période d'un point de vue enseignement ou administratif.

Je voudrais enfin remercier spécialement Anne Parrain et Olivier Roussel, qui sont depuis mon arrivée au CRIL les personnes qui subissent mes caprices, mes mauvaises humeurs, mes idées folles, mes doutes. La première, avec qui je partage un bureau depuis septembre 2001, a accepté de me suivre dans le développement de Sat4j. Le second car je le considère comme une référence technique, scientifique et morale, et parce qu'il est l'auteur de la plateforme *évaluation* que nous utilisons pour la compétition SAT depuis 2007 et qui me permet d'exploiter le cluster du CRIL pour évaluer mes prouveurs sans peine.

A Thierry Castell,
disparu le 12 août 1997 à Antelope Canyon, USA,
qui m'a initié à la résolution pratique des problèmes autour de SAT ...

Guide de lecture

Nous avons fait le choix de focaliser cette synthèse de nos activités de recherche autour des fonctionnalités du logiciel libre Sat4j, en utilisant le problème de la gestion des dépendances entre logiciels comme fil conducteur. Nos travaux de recherche plus théoriques sont disponibles dans la partie III sous la forme d'articles de revues. Nous illustrons simplement leur utilité dans le cadre de la gestion des dépendances dans cette synthèse.

Première partie

Synthèse autour de Sat4j

Chapitre 1

Introduction

La résolution pratique du problème SAT a connu un changement d'échelle il y a dix ans, ce qui a complètement changé l'impact de ce domaine de recherche dans notre « vie informatique » de tous les jours : les processeurs sont vérifiés à l'aide de méthodes formelles dans lesquelles les prouveurs SAT jouent un rôle important [32], un système d'exploitation comme Windows 7 vérifie ses pilotes de périphériques à l'aide d'outils utilisant Z3, un prouveur SMT, lui même basé sur un moteur SAT [66], etc. On retrouve aujourd'hui des moteurs SAT dans divers domaines, de la vérification de matériel ou de logiciel, en bioinformatique, en génie logiciel, en bases de données, etc. [36].

Le problème SAT intéressait une poignée de chercheurs seulement lorsque j'ai commencé mon DEA en 1994 : SAT était l'archétype du problème NP-complet [52], et on découvrait une transition de phase en faisant varier le ratio du nombre de clauses sur le nombre de variables sur des instances SAT aléatoires [47]. L'excitation due à la deuxième compétition SAT, le second challenge Dimacs [99], était encore palpable, et les meilleurs prouveurs SAT du moment s'appelaient C-SAT [72], TABLEAU [62], SATO [174] ou POSIT [77] pour les prouveurs complets, tous basés sur l'algorithme de Davis-Putnam-Logeman-Loveland (noté DPLL dans la suite) [64, 63], et GSAT [154], WalkSAT [153] ou Tabu-SAT [134] pour les algorithmes basés sur la recherche locale. La résolution de problèmes issus d'applications, comme la planification basée sur SAT, semblait à l'époque réservée à des prouveurs basés sur la recherche locale. Si une nouvelle génération de prouveurs SAT complets utilisant des techniques de retour arrière intelligent (*backjumping*) et d'apprentissage de clauses (*learning*) comme RelSAT [24], GRASP [132] et SATO 3 [174] sont apparus à cette époque, leurs performances n'étaient pas globalement très différentes d'autres prouveurs SAT complets utilisant des heuristiques sophistiquées comme SATZ [122] : le planificateur Blackbox [102] par exemple utilisait en 1999 des versions aléatoires de SATZ et de RELSAT avec redémarrages [87] pour résoudre des instances SAT issues de problèmes de planification.

Une application, la vérification de modèle bornée (*Bounded Model Checking*) [34, 33] et un prouveur SAT, Chaff [137], vont changer très rapidement le paysage de la résolution pratique du problème SAT. Cette révolution a commencé en montrant que certains problèmes de vérification de modèles qui ne pouvaient pas être résolus par des approches basées sur les diagrammes de décision binaire (*Binary Decision Diagram*) pouvaient être résolus par un prouveur SAT en utilisant une approche similaire à la planification basée sur SAT. Ce résultat a intéressé une nouvelle communauté à la résolution pratique du problème SAT : celle de la conception de circuits (*Electronic Design Automation*). Les instances SAT résultant de ces problèmes de vérification contenant des dizaines de milliers de

variables et jusqu'à des centaines de milliers de clauses, le code source des prouveurs SAT existants ont été étudiés pour trouver le meilleur compromis entre rapidité d'exécution et réduction de l'arbre de recherche développé. La plupart des prouveurs complets développés dans les années 90 étaient conçus pour réduire l'espace de recherche exploré, parfois au prix d'analyses coûteuses. Ces approches étaient souvent évaluées sur des instances aléatoires ou académiques, contenant quelques centaines de variables et quelques milliers de clauses. SATO était un peu différent car il était conçu pour résoudre des problèmes mathématiques par traduction en SAT (*Quasigroups*) [175], contenant quelques milliers de variables au maximum, mais pouvant contenir plusieurs centaines de milliers de clauses. Le prouveur Chaff contient des idées héritées des différents prouveurs complets : l'architecture de GRASP, que Lawrence Ryan appellera dans son mémoire de Master CDCL (*Conflict Driven Clause Learning*) [151], a été conservée, le principe des structures de données paresseuses de SATO a été adopté et amélioré, l'analyse de conflits de GRASP a été simplifiée, le principe d'apprentissage pertinent (*Relevance Learning*) de RELSAT a été utilisé pour planifier l'effacement des clauses apprises, l'utilisation de redémarrages [87] a aussi été conservé, etc. Pour autant, ces différents composants ont été intégrés de façon à obtenir un prouveur SAT complet capable de rapidement essayer des affectations, comme le ferait un algorithme de recherche locale (approche écartée car elle ne permet pas de prouver l'incohérence d'une instance SAT, un point primordial pour ce genre d'application). Chaff a été le premier prouveur SAT à combiner diverses techniques non pas pour leur puissance intrinsèque (adopter pour chaque composant le meilleur disponible) mais pour leur rapport performance/coût d'implémentation. L'illustration flagrante de ce principe était la simplicité et le faible coût de l'heuristique de branchement, qui se retrouve jouer un rôle secondaire car très simple, mais non négligeable car adaptative, dans le prouveur. De plus, les trois composants que sont l'analyse de conflits, l'heuristique et l'apprentissage de clauses interagissent dans Chaff d'une façon innovante.

> Conclusion du premier papier présentant Chaff [137]
> *This paper describes a new SAT solver, Chaff, which has been shown to be at least an order of magnitude (and in several cases, two orders of magnitude) faster than existing public domain SAT solvers on difficult problems from the EDA domain. This speedup is not the result of sophisticated learning strategies for pruning the search space, but rather, of efficient engineering of the key steps involved in the basic search algorithm. Specifically, this speedup is derived from :*
> – *a highly optimized BCP algorithm, and*
> – *a decision strategy highly optimized for speed, as well as focused on recently added clauses.*

Il y a donc eu un avant Chaff et un après. Les travaux de recherche présentés dans ce document se situent entre 1 BC (*Before Chaff*) et 9 AC.

Depuis l'avénement des prouveurs de type Chaff, dénommés prouveurs dirigés par les conflits avec apprentissage de clauses (*Conflict Driven Clause Learning*, CDCL) dans la suite du document, SAT est devenu un domaine de recherche dont les applications se retrouvent dans de multiples domaines. Bien que la logique propositionnelle soit peu expressive, les prouveurs SAT CDCL sont si rapides et capables de traiter des instances si conséquentes que des traductions vers SAT sont viables pour de nombreuses applications [36]. Citons en particulier :

– la vérification de circuits (*Bounded Model Checking, Equivalency Checking*, etc.) ;
– la vérification de programmes (vérification de spécifications, vérification de code em-

barqué, etc) ;
– la bio-informatique ;
– les bases de données.

Notre première contribution lors de cette période d'engouement pour SAT et ses prouveurs fut de mettre en place la compétition internationale de prouveurs SAT en 2002 avec Laurent Simon et Edward Hirsch [158], et de l'organiser régulièrement depuis avec Laurent Simon et Olivier Roussel [1]. La compétition SAT, annuelle de 2002 à 2005, en alternance avec la SAT Race depuis 2006, a permis de promouvoir les travaux sur la résolution pratique d'instances SAT, en constituant un rendez-vous régulier pour les concepteurs de prouveurs SAT et les utilisateurs de ces outils. Elle a surtout permis de faire partager au travers du code source des prouveurs ou de leurs descriptions ces petits détails qui ont rarement leur place dans un article scientifique mais qui font la différence lorsqu'il faut implémenter un prouveur. Ce fut aussi un bon moyen de promouvoir ce domaine de recherche dans d'autres communautés. Nous terminerons ce résumé scientifique, chapitre 5, par un bilan critique des six compétitions SAT et des autres évaluations de prouveurs auxquelles nous avons eu l'occasion de contribuer comme organisateur ou comme simple participant.

Une autre contribution décrite dans ce mémoire est la conception d'une bibliothèque libre de prouveurs propositionnels pour le langage Java, appelée Sat4j. Basée sur une implémentation en Java de la spécification du prouveur Minisat [74], cette bibliothèque intègre une architecture de prouveur CDCL capable d'opérer sur divers types de contraintes ainsi qu'une extension de cette architecture dont l'inférence est basée sur les plans-coupe [94] et non plus sur la résolution [148]. A l'aide de ces deux bases de prouveurs, la bibliothèque permet de résoudre des problèmes de décision comme SAT ou des problèmes d'optimisation comme MaxSat ou l'optimisation de contraintes pseudo-booléennes. Le chapitre 4 décrit en détail les raisons qui nous ont poussé à développer cette bibliothèque et ses fonctionnalités.

Parmi les applications récentes de SAT, on trouve celle de la gestion des dépendances de logiciels [127]. Les utilisateurs de Linux et ceux de la plate-forme Eclipse par exemple sont confrontés à ce problème à chaque fois qu'ils décident d'installer un nouveau logiciel ou greffon sur leur système d'exploitation ou plate-forme préférée. Nous avons eu l'occasion de nous intéresser de près au cas de la gestion des dépendances pour Eclipse depuis trois ans ; nous avons conçu avec Anne Parrain et Pascal Rapicault le nouveau système de gestion de dépendances d'Eclipse basé sur une réduction à un problème d'optimisation pseudo-booléen, appelé p2, en production depuis juin 2008 [114]. Le projet européen Mancoosi [8] s'intéresse entre autres à la gestion des dépendances (plus particulièrement dans le cadre des mises à jour) des distributions Linux. Nous avons eu l'occasion à diverses reprises de collaborer avec les chercheurs impliqués dans ce projet dirigé par Roberto Di Cosmo. Par exemple, nous avons récemment développé un outil de gestion de dépendances de type Linux basé sur p2 afin de participer aux deux compétitions d'outils de gestion des dépendances Linux organisées par le projet Mancoosi, en janvier 2010 pour la compétition « interne » (*Mancoosi internal Solver Competition, MiSC*) et en juin 2010 pour la compétition officielle (*Mancoosi International Solver Competition, MISC*) organisée dans le cadre du workshop FLoC *Logic for Component Configuration*. Ce document présente les différents travaux que nous avons effectués en utilisant le problème de gestion des dépendances comme fil conducteur. Le chapitre suivant présente ce problème dans le

1. http://www.satcompetition.org/

cadre des distributions Linux et dans le cadre d'Eclipse.

La résolution pratique des problèmes de gestion des dépendances nécessite la prise en compte des préférences des utilisateurs. On souhaite par exemple minimiser le nombre ou la taille des paquetages à installer, installer les versions les plus récentes des paquetages, etc. Dans le premier cas, ces préférences sont quantitatives. Il est donc possible d'exprimer ces préférences sous la forme de problèmes d'optimisation (pseudo-booléens, MaxSat). Dans le dernier cas, les préférences sont qualitatives. Nous avons proposé une nouvelle logique, la logique du choix qualitatif, pour représenter de manière simple et complètement intégrée à la logique propositionnelle, des préférences entre alternatives [28]. Nous avons montré que l'inférence dans cette logique correspond exactement à l'inférence lexicographique sur des bases de croyances stratifiées. Nous avons montré de plus qu'il est possible de déterminer si une formule est conséquence logique d'une bases de croyances stratifiées à l'aide d'un simple prouveur SAT [28]. Le chapitre 3 présente la modélisation de ces diverses préférences.

Quelques pistes de recherche concluent le document.

Chapitre 2

De la gestion des dépendances logicielles à SAT

2.1 Problématique de la gestion des dépendances

L'industrialisation de la production de logiciels a entrainé comme pour la manufacture de biens industriels (l'automobile, les ordinateurs, les vélos, etc.) la création de logiciels configurables à base de composants réutilisables. En génie logiciel, la conception d'applications par composants est un domaine de recherche à part entière, qui concerne à la fois la décomposition d'une application en composants réutilisables et la conception d'une application à partir de composants existants. On retrouve cette seconde problématique dans le cadre de la conception de ligne de produits logiciels (*software product line*) [50].

La configuration de produits manufacturés est un axe de recherche pour lequel il existe notamment de nombreux travaux en recherche opérationnelle et en intelligence artificielle. Ainsi, un workshop annuel associé aux grandes conférences internationales d'intelligence artificielle (IJCAI, AAAI et ECAI) dédié à la configuration de produits existe depuis 1999. La configuration de produits logiciels est un cas particulier de la configuration de produits : s'il existe des préoccupations communes (connaître le nombre de configurations existantes, connaître toutes les configurations ayant certaines propriétés, etc.), la configuration de produits logiciels revient souvent à composer des briques de base à l'aide de relations de dépendance simples. Les relations de dépendances peuvent être formalisées par un modèle de caractéristiques (*feature model*) et dans de nombreux cas, ce modèle peut être traduit en logique propositionnelle [23] ou en terme de gestion de dépendances [53].

La plate-forme ouverte Eclipse est un exemple d'architecture permettant la création de lignes de produits logiciels. Elle offre le service de composabilité nécessaire à ce type d'architecture. Chaque éditeur de logiciel peut ensuite proposer une gamme de produits basés sur cette plate-forme. IBM propose de nombreux produits basés sur Eclipse, dont *IBM Rational Application Developer for WebSphere Software* et *IBM Rational Software Architect*, qui sont sans doute les plus connus.

Une distribution Linux peut aussi être considérée comme une ligne de produits logiciels configurable à tout moment par l'utilisateur, à un niveau de granularité supérieur, les composants pouvant être des logiciels à part entière. Les distributions Linux étant composées de logiciels libres destinés à être largement diffusés, elles ont sans doute été les premières à implanter le concept de ligne de produits logiciels à grande échelle, dès le milieu des

années 90.

Dans la suite du document, nous allons nous intéresser à un problème spécifique des lignes de produits logiciels : déterminer une configuration valide, c'est-à-dire une configuration qui respecte les dépendances de ses différents composants. Nous utiliserons le terme générique « gestion des dépendances » pour dénoter ce problème. Nous verrons qu'il existe le plus souvent de nombreuses configurations valides, mais qu'elles n'ont pas toutes le même intérêt pour l'utilisateur. Nous montrerons dans les chapitres suivants comment peuvent être prises en compte les préférences de l'utilisateur pour calculer des configurations préférées.

2.2 Le cas des distributions Linux

L'une des particularités des distributions basées sur le système d'exploitation GNU Linux est de proposer à ses utilisateurs un grand nombre d'applications (de quelques milliers à quelques dizaines de milliers selon les distributions). Contrairement au cas du système d'exploitation Windows de Microsoft et du système d'exploitation Mac OS X d'Apple, pour lesquels les auteurs des logiciels garantissent le fonctionnement pour une version spécifique du système d'exploitation, les logiciels disponibles pour Linux sont souvent liés à des bibliothèques partagées ou des logiciels qui ne sont pas nécessairement présents sur le système de l'utilisateur car il n'y a pas de version standard de Linux [1]. C'est donc à l'utilisateur du logiciel de s'assurer que son système contient bien les bibliothèques ou les logiciels dont le logiciel est dépendant avant de l'installer. En pratique, ce rôle est le plus souvent délégué aux diverses distributions Linux, qui organisent l'écosystème logiciel du système Linux dans leur produit.

Les logiciels étant le plus souvent liés dynamiquement à des versions données de bibliothèques partagées (par exemple GNU glibc), la première difficulté des distributions a été de fournir un ensemble de bibliothèques compatibles avec les logiciels à installer. Devant le nombre croissant de logiciels disponibles, il s'est ensuite avéré que différents composants logiciels pouvaient être utilisés pour réaliser la même tâche. Ainsi, des méta-données explicitant les relations entre les différents composants (dépendances et conflits) et leur rôle dans le système (serveur web, client de courrier électronique, etc) ont été rapidement ajoutées aux composants logiciels eux-mêmes, et des outils capables de prendre en compte ces informations pour installer ou mettre à jour une distribution Linux existent depuis une quinzaine d'années. Dans la suite du document, on appellera paquetage (*package*) un composant logiciel accompagné de ses méta-données.

Dès 1993, la distribution Debian concevait son système de gestion des dépendances `dpkg` [1] basé sur un système de méta-données textuelles (voir exemple 2.1) et un format de paquetage de type archive unix. A la même époque, la distribution Red Hat concevait son système de gestion des dépendances `rpm`, basé sur un format d'archive binaire indépendant de la plate-forme. La plupart des distributions Linux actuelles utilisent l'un ou l'autre format. Les différences entre ces deux approches, et les raisons pour lesquelles ces deux systèmes co-existent sortent du cadre de ce document. Nous nous intéressons ici aux concepts communs, mis en évidence dans le cadre des projets européens EDOS [73] et Mancoosi [8] et nous utiliserons dans nos exemples le format *Common Update Description Format* (CUDF) [169] défini par le projet Mancoosi.

1. Un effort de standardisation existe avec le projet Linux Standard Base, `http://www.linuxbase.org/`

```
Package: firefox-3.0
Priority: optional
Section: web
Installed-Size: 3456
Maintainer: Alexander Sack <asac@ubuntu.com>
Architecture: i386
Version: 3.0.16+nobinonly-0ubuntu0.9.04.1
Replaces: firefox (<< 3), firefox-granparadiso, firefox-libthai, firefox-trunk
Provides: firefox-libthai, www-browser
Depends: fontconfig, psmisc, debianutils (>= 1.16), xulrunner-1.9 (>= 1.9.0.1),
     libatk1.0-0 (>= 1.20.0), libc6 (>= 2.4), libcairo2 (>= 1.2.4),
     libfontconfig1 (>= 2.4.0), libfreetype6 (>= 2.2.1), libgcc1 (>= 1:4.1.1),
     libglib2.0-0 (>= 2.16.0), libgtk2.0-0 (>= 2.16.0),
     libnspr4-0d (>= 4.7.3-0ubuntu1~), libpango1.0-0 (>= 1.14.0),
     libstdc++6 (>= 4.1.1),
     firefox-3.0-branding (>= 3.0.3+nobinonly-0ubuntu1~) |
          abrowser-3.0-branding (>= 3.0.3+nobinonly-0ubuntu1~)
Suggests: ubufox,
     firefox-3.0-gnome-support (= 3.0.16+nobinonly-0ubuntu0.9.04.1),
     latex-xft-fonts, libthai0
Conflicts: firefox (<< 3), firefox-granparadiso (<< 3.0~alpha8-0),
     firefox-libthai, firefox-trunk (<< 3.0~a8~cvs20070914t1713-0)
Size: 887822
Description: safe and easy web browser from Mozilla
 Firefox delivers safe, easy web browsing. A familiar user interface,
 enhanced security features including protection from online identity theft,
 and integrated search let you get the most out of the web.
 .
 Install this firefox package too, if you want to be automatically upgraded to
 new major firefox versions in the future.
```

FIGURE 2.1 – Meta-données pour le paquetage Firefox 3.0, format debian

Un paquetage est représenté de manière unique par un identifiant p et une version v. On notera p_v ce paquetage. On supposera ici qu'une version est représentée par un entier, c'est-à-dire qu'une version permet d'ordonner totalement des paquetages ayant le même identifiant : en réalité, il s'agit le plus souvent d'une chaîne de caractères dont le format dépend de la distribution Linux (par exemple, une version de firefox 3.0 disponible dans la distribution Ubuntu 9.04 est `3.0.16+nobinonly-0ubuntu0.9.04.1`).

Soit P l'ensemble des paquetages. On dénotera par p l'ensemble des paquetages ayant pour identifiant p, i.e. $p = \{p_v | p_v \in P\}$. On utilisera aussi les opérateurs de comparaison habituels ($\leq, <, >, \geq, \neq$) sur les entiers pour représenter en compréhension un ensemble de paquetages ayant le même identifiant : par exemple, $p_{\leq n} = \{p_v | p_v \in P \wedge v \leq n\}$ représente l'ensemble des paquetages ayant pour identifiant p et une version inférieure ou égale à n. Supposons que P contienne les paquetages suivants $\{libnss_1, libnss_2, libnss_3, libnss_4, libnss_5\}$. Alors $libnss_{\geq 3}$ correspond à l'ensemble $\{libnss_3, libnss_4, libnss_5\}$ et $libnss$ correspond à l'ensemble $\{libnss_1, libnss_2, libnss_3, libnss_4, libnss_5\}$.

Chaque paquetage est associé à diverses contraintes concernant les autres paquetages. Les contraintes sont exprimées à l'aide de disjonction de paquetages, en compréhension ou en extension. Comme cela se fait souvent lorsque l'on travaille avec des formules sous forme normale conjonctive, on utilisera indifféremment une notation ensembliste ou une nota-

tion logique pour exprimer des contraintes. Chaque ensemble de paquetage sera interprété comme la disjonction de ses paquetages, par exemple $\{libnss_1, libnss_2, libnss_3, libnss_4, libnss_5\}$ représentera la clause $libnss_1 \vee libnss_2 \vee libnss_3 \vee libnss_4 \vee libnss_5$. Un ensemble d'ensemble de paquetages correspondra à une conjonction de disjonction de paquetages : $\{\{libnss_1, libnss_2\}, \{libnss_3\}, \{libnss_4, libnss_5\}\}$ représentera la formule $(libnss_1 \vee libnss_2) \wedge libnss_3 \wedge (libnss_4 \vee libnss_5)$

Un problème de satisfaction de dépendances pour Linux peut donc se définir à partir du triplet $(P, depends, conflicts)$ où :

P est l'ensemble des paquetages.

depends est une fonction qui associe à chaque paquetage l'ensemble des contraintes sur les paquetages dont dépend ce paquetage, c'est-à-dire une représentation sous forme normale conjonctive des paquetages qui doivent être installés pour que le paquetage puisse être installé.

$$depends : P \rightarrow 2^{2^P}$$

conflicts est une fonction qui associe à chaque paquetage l'ensemble des paquetages qui ne doivent pas être installés pour pouvoir installer ce paquetage.

$$conflicts : P \rightarrow 2^P$$

L'exemple 2.2 montre un problème de gestion des dépendances Linux en utilisant le format CUDF qui sera repris durant ce chapitre. Cet exemple peut être exprimé par le triplet $(P, depends, conflicts)$ suivant :

$$\begin{aligned} P = \{&firefox_{36}, firefox_{25}, evolution_{230}, thunderbird_{30}, libnss_3, chromium_5, opera_9, \\ &safari_5, kmail_4, kmail_3, kde_3, kde_4, gnome_{230}, web-browser_1, mail-client_1\} \\ depends = \{&(firefox_{36}, \{\{libnss_3\}\}), (firefox_{25}, \{\{libnss_3\}\}), (evolution_{230}, \{\{gnome_{230}\}, \\ &\{libnss_3\}\}), (thunderbird_{30}, \{\{libnss_3\}\}), (kmail_4, \{\{kde_4\}\}), (kmail_3, \{\{kde_3\}\}), \\ &(internet-client_1, \{\{firefox_{36}, firefox_{25}, chromium_5, safari_5, opera_9\}, \\ &\{kmail_3, kmail_4, evolution_{230}, thunderbird_{30}\}\} \\ conflicts = \{&(firefox_{36}, \{firefox_{25}\}), (kde_4, \{kde_3\})\} \end{aligned}$$

On notera que la contrainte disjonctive de l'exemple 2.2 $firefox|chromium|safari|opera$ en compréhension correspond en fait à l'expression $firefox_{36} \mid firefox_{25} \mid chromium_5 \mid safari_5 \mid opera_9$ en extension, une fois les identifiants remplacés par la disjonction des paquetages qu'ils représentent.

D'un point de vue logique, cela revient à conditionner l'installation de p_v à la satisfaction de $\phi(p_v) = \bigwedge_{dep \in depends(p_v)} dep \wedge \bigwedge_{conf \in conflicts(p_v)} \neg conf$. On note que $\phi(p_v)$ est sous forme normale conjonctive.

On peut définir la sémantique de la satisfaction des dépendances à l'aide d'une notation ensembliste :

Définition 1 (Cohérence des dépendances Linux) *Un ensemble non vide*[2] *de paquetages $Q \subseteq P$ est cohérent avec $(P, depends, conflicts)$ ssi $\forall q \in Q, (\forall dep \in depends(q), dep \cap Q \neq \emptyset) \wedge (conflicts(q) \cap Q = \emptyset)$.*

2. L'ensemble vide est trivialement cohérent avec $(P, depends, conflicts)$, mais de peu d'intérêt dans le problème qui nous préoccupe.

La cohérence des dépendances Linux peut facilement être exprimée par la satisfaction de la formule $\phi \wedge Q$ en logique propositionnelle, pour laquelle Q dénote la conjonction de ses littéraux et ϕ est définie par :

$$\phi \equiv (\bigwedge_{p_v \in P} p_v \rightarrow (\bigwedge_{dep \in depends(p_v)} dep \wedge \bigwedge_{conf \in conflicts(p_v)} \neg conf)) \wedge \bigvee_{p_v \in P} p_v \quad (2.1)$$

L'ensemble $Q = \{thunderbird_{30}, libnss_3, opera_9, gnome_{230}, internet - client_1\}$ satisfait les dépendances de l'exemple 2.2.

On trouve parfois une liste de paquetages « recommandés », c'est-à-dire un ensemble de paquetages qu'il est souhaitable d'installer quand ce paquetage est installé. Dans l'exemple 2.2, on notera que l'on recommande d'installer `firefox` si l'on installe `thunderbird`, et vice versa. On verra plus tard qu'il s'agit en fait de contraintes souples (*soft constraints*).

On trouve aussi souvent une notion de *paquetages virtuels*, c'est-à-dire des identifiants de paquetages qui ne correspondent pas à quelque chose qui doit être installé sur la machine mais à une façon de structurer l'information. C'est aussi un moyen d'exprimer des contraintes disjonctives dans les dépendances quand les disjonctions ne sont pas autorisées dans les contraintes de dépendance (c'est le cas dans le format rpm par exemple). On considère ici qu'il s'agit uniquement d'une écriture spécifique des dépendances car toute spécification des dépendances avec des paquetages virtuels peut être réécrite uniquement à partir de dépendances sur des paquetages réels.

Si l'on définit $provides : P \rightarrow 2^P \cup V$ avec V l'ensemble des paquetages virtuels, alors on peut utiliser l'équivalence suivante pour réécrire la relation $provides$ en terme de relation $depends$:

$$q \in depends(r) \wedge \psi(q) = \{p | q \in provides(p)\} \equiv \psi(q) \in depends(r)$$

Dans l'exemple 2.2, on note que l'on a explicitement déclaré deux contraintes disjonctives dans le paquetage `internet-client`. En utilisant des paquetages virtuels, on peut « cacher » ces contraintes disjonctives. C'est ce que propose l'exemple 2.3. On note que deux paquetages virtuels, `web-browser` et `mail-client`, sont apparus, et que les disjonctions n'apparaissent plus dans les dépendances. On supposera dans la suite du document que ces deux problèmes de satisfaction de dépendances sont équivalents, et sont représentés sous forme normale sans paquetages virtuels (comme dans l'exemple 2.2).

La combinatoire du problème de gestion des dépendances est liée à la présence de contraintes disjonctives. Sans elles, on obtiendrait dans l'équation 2.1 une CNF dont toutes les clauses sauf la dernière sont de Horn, donc une CNF dont la cohérence est décidable en temps polynomial.

2.3 Le cas de la plate-forme ouverte Eclipse

Eclipse est une plate-forme ouverte libre principalement écrite en Java, et initialement développée par IBM pour remplacer son environnement de développement *Visual Age for Java*. Elle a été rendue libre lors de sa sortie en novembre 2001. Si l'objectif initial du projet était de concevoir un environnement de développement intégré ouvert pour Java, le succès rencontré l'a transformé en plate-forme ouverte pour applications clientes riches (*Eclipse Rich Client Platform*) [2] depuis la version 3.0, en juin 2004. Cette ouverture est

```
package: firefox
version: 36
depends: libnss = 3
conflicts: firefox < 30
recommends: thunderbird

package: firefox
version: 25
depends: libnss = 3
recommends: thunderbird

package: evolution
version: 230
depends: gnome = 230 , libnss = 3

package: thunderbird
version: 30
depends: libnss = 3
recommends: firefox

package: libnss          version: 3

package: chromium        version: 5

package: opera  version: 9

package: safari version: 5

package: kmail  version: 4      depends: kde = 4

package: kmail  version: 3      depends: kde = 3

package: kde    version: 4      conflicts: kde < 4

package: kde    version: 3      conflicts: kde < 3

package: gnome  version: 230    installed: true

package: internet-client
version: 1
depends: firefox | chromium | safari | opera ,
         kmail | evolution | thunderbird
```

FIGURE 2.2 – Exemple de déclaration de dépendances, format CUDF

```
package: firefox
version: 36
provides: web-browser
depends: libnss = 3
conflicts: firefox < 30
recommends: thunderbird

package: firefox
version: 25
provides: web-browser
depends: libnss = 3
recommends: thunderbird

package: chromium
version: 5
provides: web-browser

package: opera
version: 9
provides: web-browser

package: safari
version: 5
provides: web-browser

package: kmail
version: 4
provides: mail-client
depends: kde = 4

package: kmail
version: 3
provides: mail-client
depends: kde = 3

package: evolution
version: 230
provides: mail-client
depends: gnome = 230 , libnss = 3

package: thunderbird
version: 30
provides: mail-client
depends: libnss = 3
recommends: firefox

package: libnss          version: 3

package: internet-client
version: 1
depends: web-browser, mail-client
```

FIGURE 2.3 – Même exemple, en utilisant la propriété “provides” de CUDF

basée sur un modèle de composants ouvert appelé OSGi (*Open Services Gateway initiative*) [4], initialement conçu pour déployer des services à distance sur des passerelles de réseau local domestique (comme des « box » par exemple) puis intégré au domaine des consoles de jeux [3], des téléphones portables ou à l'industrie automobile. Schématiquement, la plate-forme Eclipse est composée d'un conteneur de composants OSGi appelé Equinox et d'un ensemble de composants (*bundle*) OSGi qui fournissent les différents services offerts par la plate-forme. L'ajout de fonctionnalités se fait par ajout de composants dans le conteneur.

Cependant, comme dans le cadre des distributions Linux, il existe des relations de dépendances et de conflits entre les composants. Le système de gestion de dépendances d'Eclipse, p2, agit en amont du conteneur OSGi pour sélectionner les composants OSGi à installer dans le conteneur, et en assurer l'installation.

On notera U l'ensemble des composants installables, appelés unités installables (*Installable Units*, IUs). Chaque IU est identifiée de manière unique par un identifiant iu et une version v. On notera iu_v cette IU et iu l'ensemble des IUs ayant pour identifiant iu. On notera O (*others*) l'ensemble des couples (identifiant,version) qui ne représentent pas des composants OSGi (des paquetages Java par exemple). Chaque IU offre des services (*capabilities*) et a des besoins (*requirements*). Contrairement au cas des distributions Linux, il n'y avait pas, à l'origine, de notion de conflits entre IU, mais simplement une contrainte de singleton qui empêche d'installer plus d'une IU ayant un identifiant donné. Le problème d'installabilité d'une IU pour Eclipse peut donc se définir de la façon suivante, par un quintuplet (U,O,provides,requires,singleton) tel que :

provides est une fonction qui associe à chaque IU un ensemble d'identifiants et de versions, qui dénotent des éléments de $U \cup O$ fournis par cette IU. On trouvera par exemple des services correspondant à la fourniture de composants OSGi ou des services correspondant à des paquetages java. Dans l'exemple 2.4, on note que l'IU fournit un composant OSGi et trois paquetages Java.

$$provides : U \rightarrow 2^{U \cup O}$$

requires est une fonction qui associe à chaque IU un ensemble de disjonction d'identifiants et de versions, qui dénotent les éléments de $U \cup P$ nécessaires à l'exécution de l'IU.

$$requires : U \rightarrow 2^{2^{U \cup O}}$$

singleton est une propriété d'une IU qui, quand elle est vérifiée, stipule qu'une seule IU ayant cet identifiant peut être installée dans le système.

$$singleton : U \rightarrow \{true, false\}$$

Définition 2 (Cohérence des dépendances Eclipse) *Un ensemble non vide d'IU $Q \subseteq U$ est cohérent avec $(U, O, provides, requires, singleton)$ ssi $(\forall q \in Q\ \forall req \in requires(q)\ \exists q' \in Q\ req \cap provides(q') \neq \emptyset) \wedge (\forall iu_v \in Q\ singleton(iu_v) \rightarrow |Q \cap iu| \leq 1)$.*

On peut facilement traduire tout problème de cohérence de dépendances Eclipse en un problème de cohérence de dépendances Linux :

$$ECLIPSE - TO - LINUX : (U, O, provides, requires, singleton) \mapsto (P, depends, conflicts)$$

$$P = U$$

$$depends = \{(p, \{\{r | q \in provides(r) | q \in Q \wedge q \in O\} \cup \{q | q \in Q \wedge q \in U\}\}) | (iu_v, Q) \in requires\}$$

$$conflits = \{(iu_v, \{iu_w | iu_w \in iu \wedge v \neq w\}) | singleton(iu_v) \wedge iu_v \in U\}$$

3. La wiimote de Nintendo est un exemple de périphérique OSGi dans ce contexte.

```
<unit id='org.eclipse.equinox.p2.director' version='1.0.100.v20090520-1905'>
  <update id='org.eclipse.equinox.p2.director' range='[0.0.0,1.0.100.v20090520-1905)'
        severity='0'/>
          [...]
 <provides size='7'>
    <provided namespace='org.eclipse.equinox.p2.iu' name='org.eclipse.equinox.p2.director'
         version='1.0.100.v20090520-1905'/>
    <provided namespace='osgi.bundle' name='org.eclipse.equinox.p2.director'
         version='1.0.100.v20090520-1905'/>
    <provided namespace='java.package' name='org.eclipse.equinox.internal.p2.director'
          version='0.0.0'/>
    <provided namespace='java.package' name='org.eclipse.equinox.internal.p2.rollback'
          version='0.0.0'/>
    <provided namespace='java.package'
          name='org.eclipse.equinox.internal.provisional.p2.director' version='0.0.0'/>
    <provided namespace='org.eclipse.equinox.p2.eclipse.type' name='bundle'
         version='1.0.0'/>
    <provided namespace='org.eclipse.equinox.p2.localization' name='df_LT'
         version='1.0.0'/>
  </provides>
  <requires size='15'>
    <required namespace='osgi.bundle'
         name='org.eclipse.equinox.common' range='[3.5.0,4.0.0)'/>
    <required namespace='osgi.bundle'
          name='org.sat4j.core' range='2.1.0'/>
    <required namespace='osgi.bundle'
          name='org.sat4j.pb' range='2.1.0'/>
    <required namespace='osgi.bundle'
          name='org.eclipse.core.jobs' range='3.4.100'/>
    <required namespace='java.package'
          name='org.eclipse.equinox.internal.p2.core.helpers' range='0.0.0'/>
    <required namespace='java.package'
          name='org.eclipse.equinox.internal.provisional.configurator' range='0.0.0'/>
    <required namespace='java.package'
          name='org.eclipse.equinox.internal.provisional.p2.core' range='0.0.0'/>
    <required namespace='java.package'
          name='org.eclipse.equinox.internal.provisional.p2.engine' range='0.0.0'/>
    <required namespace='java.package'
          name='org.eclipse.equinox.internal.provisional.p2.metadata' range='0.0.0'/>
    <required namespace='java.package'
          name='org.eclipse.equinox.internal.provisional.p2.metadata.query' range='0.0.0'/>
    <required namespace='java.package'
          name='org.eclipse.equinox.internal.provisional.p2.metadata.repository' range='0.0.0'/>
    <required namespace='java.package'
          name='org.eclipse.equinox.internal.provisional.p2.query' range='0.0.0'/>
    <required namespace='java.package'
          name='org.eclipse.equinox.internal.provisional.p2.repository' range='0.0.0'/>
    <required namespace='java.package'
          name='org.eclipse.osgi.util' range='1.0.0'/>
    <required namespace='java.package'
          name='org.osgi.framework' range='1.3.0'/>
  </requires>
  <artifacts size='1'>
    <artifact classifier='osgi.bundle' id='org.eclipse.equinox.p2.director'
          version='1.0.100.v20090520-1905'/>
  </artifacts>
   [...]
</unit>
```

FIGURE 2.4 – Méta-données pour le composant OSGi Eclipse p2 director

L'inverse est aussi possible, mais moins immédiat, car il faut créer deux nouveaux paquetages ayant le même identifiant et deux versions différentes pour chaque couple de paquetages en conflit de façon à exprimer ce conflit en terme de contrainte singleton sur les deux nouveaux paquetages.

Par exemple, la déclaration suivante au format Linux :

```
package: a
version: v
conflicts: b = u
```

se traduira de la façon suivante dans le format Eclipse (la notion de singleton est exprimée par des auto-conflits) :

```
package: a
version: v
depends: p = 1

package: b
version: u
depends: p = 2

package: p
version: 1
conflicts: p    # singleton

package: p
version: 2
conflicts: p    # singleton
```

Dans le cadre d'Eclipse, une dépendance peut être gourmande (*greedy*) ou non, c'est-à-dire qu'elle provoque ou pas l'installation des paquetages dont dépend une IU. Si une dépendance est non gourmande, alors les éléments nécessaires à la satisfaction des dépendances devront être apportés par des dépendances gourmandes ou installés directement par l'utilisateur.

Supposons que nous disposions de quatre IUs A,B,C,D telles que A dépend de B et de C, et D dépend de B. Chacune de ces dépendances peut être gourmande ou non. Le tableau 2.1 montre l'effet du type de dépendance entre A et B et D et B lorsque l'on souhaite installer A et lorsque l'on souhaite installer A et D.

A dépend de B	A dépend de C	D dépend de B	Installer A	Installer A,D
gourmande	gourmande	gourmande	A,B,C	A,B,C,D
non gourmande	gourmande	gourmande	impossible	A,B,C,D
non gourmande	gourmande	non gourmande	impossible	impossible

TABLE 2.1 – Les dépendances gourmandes et non gourmandes d'Eclipse

Si les dépendances sont gourmandes, on retrouve le fonctionnement classique des dépendances. Si la dépendance entre A et B n'est pas gourmande, alors il n'est pas possible de satisfaire les dépendances de A, puisque B n'est apporté par aucune IU. En revanche, si l'on

installe aussi D, qui a une dépendance gourmande sur B, alors il est possible de satisfaire à la fois les dépendances de A et celle de D. A l'opposé, si ces deux dépendances sont non gourmandes, il est impossible de les satisfaire.

Cette notion de dépendance gourmande est utilisée par exemple pour concevoir des applications qui seront soit déployées sur Eclipse *Rich Client Platform*, et fonctionneront donc comme des applications Java classiques, soit déployées sur Eclipse *Rich Ajax Platform*, et fonctionneront donc comme des applications web. Cependant, ces applications ne doivent pas être déployées sur une version d'Eclipse de développeur (*Software Development Kit*). La dépendance doit donc vérifier que soit RCP soit RAP sont présents, mais ne doit pas installer l'une ou l'autre de ces plates-formes si aucune d'elles n'est présente.

Pour prendre en compte ces nouvelles dépendances, on ajoute une nouvelle fonction $ngrequires$ qui associe à chaque IU un ensemble de disjonction d'identifiants et de versions, qui dénotent les éléments de $U \cup O$ nécessaires de manière non gourmande à l'exécution de l'IU.

$$ngrequires : U \rightarrow 2^{2^{U \cup O}}$$

Définition 3 (Satisfaction des dépen. Eclipse avec dépendances non gourmandes) *Un ensemble non vide d'IU* $Q \subseteq U$ ***satisfait*** $(U, O, provides, requires, ngrequires, singleton)$ ***ssi*** $(\forall q \in Q\ \forall req \in requires(q)\ \exists q' \in Q\ req \cap provides(q') \neq \emptyset) \wedge (\forall iu_v \in Q\ singleton(iu_v) \rightarrow |Q \cap iu| \leq 1) \wedge (\forall q \in Q\ \forall req \in ngrequires(q)\ \exists q' \in Q\ \exists req'\ \in requires(q')\ req \cap req' \neq \emptyset)$.

Les dépendances non gourmandes peuvent aussi être encodées à l'aide d'une formule propositionnelle. Soit $NG = \{ng_e | u \in U \wedge e \in ngrequires(u)\}$ l'ensemble des IUs sur lesquelles portent une dépendance non gourmande. Soit $ng : U \cup O \mapsto NG$ la fonction qui associe à tout élément e de $U \cup O$ l'élément ng_e de NG. Tout modèle de la formule suivante permet de construire un ensemble Q satisfaisant les dépendances non gourmandes d'Eclipse :

$$\begin{aligned}
\phi \equiv \bigwedge_{iu_v \in U} (iu_v \rightarrow (& \bigwedge_{req \in requires(iu_v)} \bigvee_{req \in provides(prov_w)} prov_w) \\
& \wedge (iu_v \rightarrow (\bigwedge_{req \in requires(iu_v)} \bigvee_{req \in provides(prov_w)} ng(prov_w)) \\
& \wedge \bigwedge_{singleton(iu_v)} (\sum_{iu_w \in iu} iu_w \leq 1) \\
& \wedge (\bigwedge_{ng_e \in NG} ng_e \rightarrow \bigvee_{e \in provides(p_v) \vee (\exists req \in requires(p_v) e \in req)} p_v)
\end{aligned}$$

La spécification OSGi release 4 sortie en août 2005 définit d'autres relations de dépendances assez subtiles liées au fonctionnement du langage Java, comme par exemple des contraintes sur les paquetages Java qui se trouvent dans les services : la cohérence des classes Java ne peut être garantie que si chaque paquetage Java n'est fourni que par un seul composant. Ce genre de contraintes dépasse notre cadre, qui se limite aux contraintes entre les composants eux-mêmes.

On note que la gestion des dépendances dans le cadre d'Eclipse est très similaire à celle du monde Linux. Il existe cependant une petite différence sémantique concernant les méta-données.

Dans le cadre des distributions Linux, les méta-données représentent des configurations connues pour être compatibles avec les logiciels associés : violer les contraintes des méta-données peut aboutir à un système parfaitement opérationnel, mais qui sort des configurations testées par la distribution (la plupart des outils de gestion des dépendances sous Linux permettent à l'utilisateur de « forcer » une installation, c'est-à-dire d'installer un paquetage dont les dépendances ne sont pas satisfaites). Les contraintes exprimées dans les méta-données représentent donc un sous-ensemble de l'espace des configurations valides à l'exécution.

Dans le cadre d'Eclipse, les méta-données sont fournies par les auteurs des composants, car elles sont utilisées pour charger les classes Java nécessaires à l'exécution du service. Ne pas satisfaire les dépendances déclarées implique que l'on est certain que le service ne pourra pas être déployé. Il y a donc une bijection entre l'espace des configurations opérationnelles et l'ensemble des configurations exprimées par les méta-données.

2.4 En terme de complexité

La notion d'installabilité d'un paquetage est importante car elle permet de déterminer la complexité théorique de la gestion des dépendances. Nous verrons qu'en pratique, les problèmes de décision de gestion des dépendances qui nous intéressent ne sont pas difficiles compte tenu des performances des prouveurs SAT et de la puissance de calcul disponible sur les ordinateurs actuels. Ce sont les problèmes d'optimisation associés qui sont difficiles.

Définition 4 (Installabilité d'un paquetage Linux, INST-LINUX) *Un paquetage* $p_v \in P$ *est installable dans* $(P, depends, conflicts)$ *ssi il existe un ensemble de paquetages* $Q \subseteq P$ *tel que* $p_v \in Q$ *qui satisfait* $(P, depends, conflicts)$.

On retrouve dès 1999 dans [166] une étude du problème d'installabilité de paquets Linux dans le cadre de la distribution Debian et sa formalisation en ASP (en fait, via un langage formel à base de règles basé sur la sémantique des modèles stables). Le problème d'installabilité d'un paquetage Linux dans le format Debian y est prouvé NP-complet par équivalence à la cohérence d'un programme logique ASP. Cette étude propose deux encodages du problème : un encodage pour rechercher une solution et un encodage pour expliquer pourquoi un paquetage ne peut pas être installé. Cependant, ce travail considéré par la communauté ASP comme l'une des premières applications pratiques réussies du formalisme ASP, reste confidentiel en dehors de cette communauté.

En 2005, ce résultat a été retrouvé et généralisé à la gestion des dépendances dans le format RPM dans le cadre du projet européen EDOS [127]. La NP-complétude est prouvée par réduction polynomiale à et depuis 3-SAT [168]. Une formalisation en programmation par contraintes est aussi fournie. La notion d'installabilité d'un paquetage était importante dans le contexte du projet EDOS car l'un des buts du projet était de déterminer un moyen efficace pour une distribution Linux de garantir que tous les paquetages inclus dans une version particulière de la distribution soient installables (ce qui ne signifie pas qu'ils puissent tous être installés en même temps).

Nous allons fournir notre propre preuve de la NP-complétude, sous une forme qui nous semble plus simple que celle définie dans [127] et que nous pourrons réutiliser pour montrer que le problème de l'installabilité d'un composant Eclipse est lui aussi NP-complet.

La première étape de la preuve est de proposer une réduction polynomiale de SAT à INST-LINUX. Les principes de cette réduction sont les suivants :

- Chaque littéral et chaque clause sont représentés par un paquetage.
- Les paquetages identifiés par une variable propositionnelle en version 1 représentent les littéraux positifs et les paquetages identifiés par une variable propositionnelle en version 2 représentent les littéraux négatifs. Les paquetages représentant des littéraux complémentaires sont déclarés être en conflits l'un avec l'autre. Ils n'ont pas de dépendance.
- Les clauses sont représentées par un paquetage d'identifiant $clause$ et ayant pour version l'index de la clause dans la formule. La seule dépendance d'un paquetage représentant une clause est la disjonction des paquetages représentant les littéraux de cette clause.
- Un paquetage $formule$ représentant la formule, qui dépend de toutes les versions des paquetages représentant une clause, complète l'encodage.

Soit A un ensemble de n variables propositionnelles et soit $\phi = \{c_1, c_2, ..., c_m\}$ un ensemble de clauses construites à partir des variables de A.

$CUDF - LINUX(\phi) = (P, depends, conflicts)$ est définie par :

$$P = \{a_1, a_2 | a \in A\} \cup \{clause_i | c_i \in \phi\} \cup \{formule_1\}$$
$$depends = \{(clause_i, \{\{a_1 | a \in c_i\} \cup \{a_2 | \neg a \in c_i\}\}) | c_i \in \phi\} \cup \{(formule_1, \{\{clause_i\} | c_i \in \phi\})\}$$
$$conflicts = \{(a_1, \{a_2\}), (a_2, \{a_1\}) | a \in A\}$$

Un exemple de traduction d'une CNF par $CUDF - LINUX$ est proposée dans la figure 2.5.

Lemme 1 *ϕ est satisfiable ssi $formule_1$ est installable dans $CUDF - LINUX(\phi)$.*

Preuve 1 *1) si ϕ est satisfiable alors $formule_1$ est installable dans $CUDF - LINUX(\phi)$.*
Soit M un modèle de ϕ. Soit $G = \{a_1 | a \in M\} \cup \{a_2 | \neg a \in M\}$. M est un modèle de ϕ donc $\forall c_i \in \phi, c_i \cap M \neq \emptyset$, donc $\forall clause_i \in P, \{Q\} = depends(clause_i), Q \cap G \neq \emptyset$. Donc tous les paquetages $clause_i$ sont installables. Toutes les dépendances de $formule_1$ sont installables, donc $formule_1$ est aussi installable.
2) si $formule_1$ est installable dans $CUDF - LINUX(\phi)$ alors ϕ est satisfiable.
Soit G un ensemble de paquets installables tels que $formule_1 \in G$. Soit $M = \{a | a \in A \wedge a_1 \in G\} \cup \{\neg a | a \in A \wedge a_2 \in G\}$. Puisque $formule_1$ est installable, alors tous les paquetages $clause_i$ sont aussi installables, i.e. $\forall i = 1, .., m, clause_i \in G$, donc $\forall i = 1, .., m, \{Q\} = depends(clause_i) \wedge Q \cap G \neq \emptyset$ donc M satisfait c_i. ■

Une fois ce résultat obtenu, il devient simple de retrouver le résultat suivant :

Propriété 1 *Le problème d'installabilité d'un paquetage Linux est NP-complet [166, 127].*

Preuve 2 Réduction de SAT à INST-LINUX : *en utilisant l'encodage $CUDF - LINUX$ et le Lemme 1. Donc $INST - LINUX$ est NP-difficile.*
Réduction de INST-LINUX à SAT : *donnée par la formule 2.1 en remplaçant la dernière clause par la clause unitaire p_v. Donc $INST - LINUX$ est NP-complet.* ■

Une approche similaire peut être utilisée pour montrer que le problème de l'installabilité de composants Eclipse est lui aussi NP-complet.

```
package: a
version: 1
conflicts: a = 2

package: a
version: 2
conflicts: a = 1

package: b
version: 1
conflicts: b = 2

package: b
version: 2
conflicts: b = 1

package: c
version: 1
conflicts: c = 2

package: c
version: 2
conflicts: c = 1

package: clause
version: 1
depends: a = 2 | b = 1 | c = 1

package: clause
version: 2
depends: a = 2 | b = 2 | c = 1

package: clause
version: 3
depends: a = 1

package: clause
version: 4
depends: c = 2

package: formule
version: 1
depends: clause = 1 , clause = 2 , clause = 3 , clause = 4
```

FIGURE 2.5 – Encodage $CUDF - LINUX((\neg a \vee b \vee c) \wedge (\neg a \vee \neg b \vee c) \wedge a \wedge \neg c)$

Définition 5 (Installabilité d'un composant Eclipse, INST-ECLIPSE) *Un composant* $iu_v \in U$ ***est installable dans*** $(U, O, provides, requires, singleton)$ ***ssi il existe un ensemble de composants*** $Q \subseteq U$ ***tel que*** $iu_v \in Q$ ***qui satisfait*** $(U, O, provides, requires, singleton)$.

$CUDF - ECLIPSE(\phi) = (U, O, provides, requires, singleton)$ est définie par :

$$
\begin{aligned}
U &= \{a_1, a_2 | a \in A\} \cup \{formule_1\} \\
O &= \{clause_i | c_i \in \phi\} \\
provides &= \{(a_1, clause_i) | c_i \in \phi \wedge a \in c_i\} \cup \{(a_2, clause_i) | c_i \in \phi \wedge \neg a \in c_i\} \\
requires &= \{(formule_1, \{\{clause_i\} | c_i \in \phi\})\} \\
singleton &= \{(a_1, true), (a_2, true) | a \in A\} \cup \{(formule_1, false)\}
\end{aligned}
$$

Un exemple de traduction d'une CNF par $CUDF - ECLIPSE$ est proposée dans la figure 2.6. La notion de singleton en CUDF est exprimée par un conflit sur l'identifiant du paquetage ($conflicts(a_1) = a$).

2.5 De la satisfaction à l'optimisation

Nous avons déjà indiqué que la logique propositionnelle permet de représenter assez simplement des problèmes d'installabilité correspondant à des préoccupations réelles. Cependant, le problème d'installabilité n'a que peu d'intérêt en pratique lorsqu'il s'agit d'installer ou de mettre à jour un système. Les contraintes de dépendances admettent souvent de nombreuses solutions, qui ne sont pas toutes d'égales valeurs pour l'application visée. Un outil de gestion des dépendances doit donc prendre en compte d'autres informations afin de calculer la meilleure solution possible.

On notera tout d'abord un principe de parcimonie, qui stipule que l'on préfère les solutions n'installant que des paquetages nécessaires, c'est-à-dire que l'on souhaite éviter d'installer des paquetages qui ne sont pas indispensables. Dans [166], des solutions respectant ce principe sont obtenues automatiquement en utilisant la sémantique des modèles stables.

Comme les paquetages ou composants doivent typiquement être obtenus depuis Internet, on peut facilement imaginer que des informations comme la taille du paquetage puissent être prises en compte dans le calcul d'une solution optimale. On entre alors dans le cadre des problèmes d'optimisation, comme les problèmes de couverture binaire (*binate covering problem*) [60] ou plus généralement les problèmes d'optimisation pseudo-booléens [150]. Dès 2007, un outil de gestion des dépendances basé sur un prouveur pseudo-booléen a été proposé : l'outil Opium pour la distribution Linux *Linspire* [170].

Enfin, comme souvent les paquetages peuvent être disponibles en diverses versions, on peut imaginer préférer installer les versions les plus récentes de ces paquetages. Le prochain chapitre présentera une nouvelle logique, la logique du choix qualitatif, tout à fait adaptée à l'expression de ce type de préférences.

```
package: a
version: 1
conflicts: a
provides: clause = 3

package: a
version: 2
conflicts: a
provides: clause = 1 , clause = 2

package: b
version: 1
conflicts: b
provides: clause = 1

package: b
version: 2
conflicts: b
provides: clause = 2

package: c
version: 1
conflicts: c
provides: clause = 1 , clause = 2

package: c
version: 2
conflicts: c
provides: clause = 4

package: formule
version: 1
depends: clause = 1 , clause = 2 , clause = 3, clause = 4
```

FIGURE 2.6 – Encodage $CUDF - ECLIPSE((\neg a \vee b \vee c) \wedge (\neg a \vee \neg b \vee c) \wedge a \wedge \neg c)$

Chapitre 3

Modélisation de préférences en logique propositionnelle

Il existe actuellement plusieurs cadres permettant de modéliser des problèmes d'optimisation en logique propositionnelle, et disposant d'outils performants pour les résoudre : les contraintes pseudo-booléennes, qui permettent de modéliser de manière compacte certaines fonctions booléennes [150], et les différentes variantes du problème MaxSat [123]. Nous montrerons qu'il existe des relations entre ces deux cadres, et qu'ils sont complémentaires en terme de modélisation.

Les deux cadres précédents permettent d'exprimer des préférences quantitatives, à base de fonctions d'optimisation. Il existe cependant dans de nombreux problèmes des préférences pour lesquelles une modélisation qualitative est préférable. Nous montrerons que la sémantique des préférences sur les versions de paquetage inhérente aux problèmes de gestion des dépendances correspond exactement à celle d'une nouvelle logique conçue pour exprimer des préférences entre alternatives : la logique du choix qualitatif [43].

3.1 Contraintes pseudo-booléennes linéaires

3.1.1 Définitions

Une contrainte pseudo-booléenne linéaire est définie sur un ensemble de variables booléennes x_i à valeurs dans $\{0,1\}$. 1 représente la valeur `vrai` et 0 la valeur `faux`. La forme générale d'une contrainte pseudo-booléenne linéaire est $\sum_i a_i.x_i \triangleright k$ où a_i et k sont des constantes entières et $\triangleright$ est l'un des opérateurs classiques de comparaison ($=, >, \geq, <, \leq$). Les opérateurs d'addition et de multiplication ont leur sémantique mathématique habituelle. La partie droite de la contrainte (k) est appelée le *degré* de la contrainte. Un problème pseudo-booléen est soit une conjonction de contraintes pseudo-booléennes (problèmes de décision), soit, le plus souvent, une conjonction de contraintes pseudo-booléennes plus une fonction d'objectif qui consiste à minimiser ou maximiser une expression $\sum_i a_i.x_i$ (problèmes d'optimisation). Résoudre une conjonction de contraintes pseudo-booléennes correspond au 2e des 21 problèmes NP-complets de Karp [101].

Ces contraintes peuvent être transformées afin de n'utiliser que l'opérateur $\geq$: les contraintes d'égalité sont d'abord codées en deux inégalités ($\geq$ et $\leq$) ; les contraintes $<$ et $\leq$ sont multipliées par -1 pour obtenir des contraintes $>$ et $\geq$ respectivement et enfin $\sum_i a_i.x_i > k$ peut être transformée en $\sum_i a_i.x_i \geq k+1$. À cette étape, les contraintes pseudo-booléennes sont de la forme :

$$\sum_i a_i.x_i \geq k, \qquad a_i, k \in \mathbb{Z} \tag{3.1}$$

En introduisant des littéraux l_i et $\overline{l_i}$, on peut encore transformer les contraintes afin de n'obtenir plus que des coefficients entiers positifs : on pose $\overline{x_i} = 1 - x_i$. Toute conjonction de contraintes pseudo-booléennes linéaires peut ainsi être transformée en une conjonction de contraintes de type $\geq$ sur des littéraux avec des coefficients strictement positifs :

$$\sum_i a_i.l_i \geq k, \qquad a_i, k \in \mathbb{N} \tag{3.2}$$

Par exemple, $3x_1 - 7x_2 < -2$ est normalisé en $3\overline{x_1} + 7x_2 \geq 6$.

3.1.2 Des contraintes pseudo-booléennes à la gestion des dépendances

Les clauses propositionnelles et les contraintes de cardinalités (normalisées) peuvent être considérées comme un cas particulier de contraintes pseudo-booléennes.

- La clause $libnss_1 \vee libnss_2 \vee libnss_3 \vee libnss_4 \vee libnss_5$ se traduit en $libnss_1 + libnss_2 + libnss_3 + libnss_4 + libnss_5 \geq 1$;
- la contrainte de cardinalité $atleast(k, \{libnss_1, libnss_2, libnss_3, libnss_4, libnss_5\})$ en $libnss_1 + libnss_2 + libnss_3 + libnss_4 + libnss_5 \geq k$;
- $atmost(k, \{libnss_1, libnss_2, libnss_3, libnss_4, libnss_5\})$ en $\overline{libnss_1} + \overline{libnss_2} + \overline{libnss_3} + \overline{libnss_4} + \overline{l_5} \geq 5 - k$.

A contrario, toute formule pseudo-booléenne peut être exprimée par un ensemble de clauses : il suffit par exemple de construire la table de vérité de cette formule pour en déduire une forme normale conjonctive. En pratique, il existe des façons beaucoup plus efficaces de procéder (cf. minisat+ [160] par exemple). Une contrainte pseudo-booléenne est donc une représentation compacte de certaines fonctions booléennes.

Nous avons noté dans le cadre du problème de gestion de dépendance Eclipse qu'une contrainte de singleton est exactement une contrainte de cardinalité : $libnss_1 + libnss_2 + libnss_3 + libnss_4 + libnss_5 \leq 1$. Représenter cette contrainte de cardinalité sans ajout de variables nécessite 10 clauses binaires : $\{\neg libnss_i \vee \neg libnss_j | 1 \leq j < i \leq 5\}$. Sa représentation sous forme de contrainte de cardinalité nécessite 5 littéraux et un degré. La représentation sous forme clausale nécessite 20 littéraux.

On retrouve certains patrons simples pour transformer un problème de gestion des dépendances sous forme clausale en un problème pseudo-booléen :

1. $a \rightarrow b_1 \vee b_2 \vee \ldots \vee b_m \equiv -a + \sum_i b_i \geq 0$
2. $\neg a \vee \neg b \equiv \overline{a} + \overline{b} \geq 1$

Une représentation pseudo-booléenne du problème d'installation d'un paquetage q satisfaisant les règles de dépendances Linux pourrait être :

$$(\bigwedge_{p_v \in P} (-p_v + \sum_{dep \in depends(p_v)} dep \geq 0) \wedge \bigwedge_{conf \in conflicts(p_v)} \overline{p_v} + \overline{conf} \geq 1)) \wedge q \qquad (3.3)$$

En ajoutant la fonction objectif $min : \sum_{p_v \in P} p_v$, on peut maintenant exprimer que l'on souhaite une solution qui minimise le nombre de paquetages à installer. Si l'on note $taille(p_v)$ la taille (en Kio par exemple) de chaque paquetage, utiliser la fonction objectif $min : \sum_{p_v \in P} taille(p_v) * p_v$ permet d'exprimer que l'on cherche une solution qui minimise la taille totale des paquetages à installer.

Ces problèmes d'optimisation pseudo-booléens sont très particuliers car leurs contraintes sont des clauses. Ils correspondent au problème de couverture binaire (*binate covering problem*) [60] étudié depuis longtemps dans la communauté de la vérification de circuits [131].

Le cadre de l'optimisation pseudo-booléen est suffisant pour exprimer diverses préférences entre les solutions attendues d'un problème de gestion de dépendances si celles-ci sont quantitatives : compter le nombre de paquets ayant une certaine propriété (installé, récent), cumuler des propriétés numériques (taille), etc.

3.2 MaxSat

Une autre forme de problème d'optimisation très proche du problème SAT est le problème MaxSat [123]. Dans sa forme généralisée, on dispose d'un ensemble de contraintes pondérées par des entiers et construites à partir d'un ensemble de variables propositionnelles PS. On note $\phi = \{(w_1, c_1), (w_2, c_2, \ldots (w_n, c_n)\}$ cet ensemble.

Toute affectation I des variables propositionnelles de PS est évaluée de la façon suivante :
- $poids(I, (w_i, c_i)) = 0$ si I satisfait c_i, w_i sinon.
- $poids(I, \phi) = \sum_{wc \in \phi} poids(I, wc)$

On cherche une affectation I des éléments de PS qui minimise $poids(I)$, i.e. qui minimise le poids des contraintes violées par cette affectation, ou de manière duale maximise le poids des contraintes satisfaites. On utilise en pratique la définition en terme de contraintes

violées car cela permet de raisonner à partir des seules contraintes violées.

Lorsque tous les poids sont égaux, on retrouve le cadre original du problème MaxSat, où toutes les clauses ont la même valeur.

On considère parfois un poids particulier, noté ∞, qui signifie que la clause associée ne peut pas être violée. On parle alors de clause *dure*. Les autres clauses sont considérées comme *souples*. Lorsque toutes les clauses sont dures, on retrouve le cadre original du problème SAT. Dans les autres cas, on parle de problème MaxSat partiel, qui contient un mélange de clauses dures et de clauses souples.

La table 3.1 résume les différentes dénominations utilisées lors de la compétition MaxSat en fonction des poids utilisés sur les contraintes.

poids	∞	dénomination
k	non	MaxSat
k	oui	MaxSat partiel
$\mathbb{N}$	non	MaxSat pondéré
$\mathbb{N}$	oui	MaxSat partiel pondéré

TABLE 3.1 – Dénominations utilisées pour caractériser les problèmes MaxSat

Une représentation MaxSat partiel du problème d'installation d'un paquetage q satisfaisant les règles de dépendances Linux minimisant le nombre de paquetages installés est :

$$(\bigwedge_{p_v \in P} (\infty, p_v \rightarrow (\bigwedge_{dep \in depends(p_v)} dep) \wedge \bigwedge_{conf \in conflicts(p_v)} (\infty, p_v \rightarrow \neg conf)) \wedge (\infty, q) \wedge (\bigwedge_{p_v \in P, p_v \neq q} (k, \neg p_v)) \tag{3.4}$$

Pour minimiser la taille des paquetages installés, il faut utiliser une représentation sous forme de problème MaxSat partiel pondéré :

$$(\bigwedge_{p_v \in P} (\infty, p_v \rightarrow (\bigwedge_{dep \in depends(p_v)} dep) \wedge \bigwedge_{conf \in conflicts(p_v)} (\infty, p_v \rightarrow \neg conf)) \wedge (\infty, q) \wedge (\bigwedge_{p_v \in P, p_v \neq q} (taille(p_v), \neg p_v)) \tag{3.5}$$

Cette traduction est immédiate car on se trouve dans le cadre d'un problème de couverture binaire. Son principal intérêt se situe dans la résolution pratique du problème : elle permet d'utiliser des prouveurs MaxSat, qui reçoivent beaucoup d'attention ces dernières années.

L'utilisation du formalisme MaxSat est intéressante pour exprimer la notion de dépendance optionnelle dans Eclipse ou de dépendance recommandée sous Linux. En effet, il s'agit de dépendances qui doivent être satisfaites autant que possible. Ce qui correspond exactement à la sémantique d'une clause souple. Si l'on suppose que ces dépendances optionnelles sont représentées par l'ensemble $optDepends$ de manière similaire à $depends$, on peut modéliser la gestion des dépendances optionnelles de la façon suivante :

$$(\bigwedge_{p_v \in P} \ldots \wedge (\infty, q) \wedge (k', p_v \rightarrow (\bigwedge_{dep \in optDepends(p_v)} dep)) \tag{3.6}$$

Si maintenant on souhaite ajouter une fonction d'optimisation pour minimiser le nombre de paquetages à installer, ou leur taille, on se retrouve avec le problème classique de la compensation des poids. Il faut choisir des valeurs pour k et k' : $k' > |P| * k$ par exemple si la satisfaction des dépendances optionnelles est plus importante que la minimisation de la fonction objectif.

Une manière d'éviter ces phénomènes de compensation est d'utiliser des préférences qualitatives.

3.3 La logique du choix qualitatif

La logique du choix qualitatif [43] (article joint chapitre 10) est une nouvelle logique qui étend la logique propositionnelle classique d'un nouvel opérateur, la disjonction ordonnée ($\times$). Ce nouvel opérateur permet d'exprimer une préférence entre plusieurs alternatives. $firefox_{36} \times firefox_{25}$ par exemple signifie dans ce contexte que l'on souhaite installer une version de firefox, si possible la dernière version, 36, sinon la version 25.

D'un point de vue logique, cela signifie que :

$$\begin{aligned} firefox_{36} \times firefox_{25} &\models firefox_{36} \\ firefox_{36} \times firefox_{25} \wedge \neg firefox_{36} &\models firefox_{25} \\ firefox_{36} \times firefox_{25} \wedge \neg firefox_{36} \wedge \neg firefox_{25} &\models \bot \end{aligned}$$

L'intérêt de ce nouvel opérateur est d'être complètement intégré à la logique, il est donc possible de construire n'importe quelle formule logique combinant les opérateurs classiques et la disjonction ordonnée. La formule $chromium \times safari \times firefox \times opera$ qui indique que l'on préfère utiliser de préférence le navigateur chromium, si ce n'est pas possible un autre navigateur basé sur webkit (safari), sinon firefox et en dernier recours, le navigateur opera. Nous avons indiqué que cette formule doit en fait être comprise comme représentant la formule $chromium_5 \times safari_5 \times (firefox_{36} \vee firefox_{25}) \times opera_9$ (car il faut remplacer chaque identifiant par la disjonction des versions disponibles), qui est une formule bien formée dans la logique QCL. De même, si l'on souhaite intégrer les préférences entre les versions de firefox, nous obtenons la formule QCL $chromium_5 \times safari_5 \times firefox_{36} \times firefox_{25} \times opera_9$ qui contient uniquement des disjonctions ordonnées puisque la disjonction ordonnée est un opérateur associatif [43].

Il est aussi possible d'exprimer des préférences conditionnelles, comme par exemple :
$gnome_{230} \rightarrow (evolution_{230} \times thunderbird_{30}) \wedge$
$\neg gnome_{230} \rightarrow (thunderbird_{30} \times (kmail_3 \vee kmail_4) \times evolution_{230})$
qui signifie que l'on préfère installer $evolution$ à $thunderbird$ comme lecteur de mail si l'environnement de bureau $gnome$ est installé. Dans le cas contraire, on préfère installer $thunderbird$ à $kmail$, $evolution$ étant le dernier choix.

Cette logique est donc parfaitement adaptée à la gestion des dépendances, car elle permet d'exprimer des préférences « locales », c'est à dire des préférences qui s'appliquent à un sous-ensemble des variables propositionnelles. La sémantique de la disjonction ordonnée de QCL est exactement celle qui existe généralement entre les différentes versions d'un même paquetage. Il est donc possible de modéliser la notion de préférences entre paquetages à l'aide de contraintes disjonctives ordonnées (aussi appelée contraintes de choix

basiques).

En pratique, nous avons maintenant besoin de calculer un modèle qui satisfasse une formule QCL, par exemple la formule $\alpha \wedge (kmail_4 \times kmail_3) \wedge (firefox_{36} \times firefox_{25})$ où α est une CNF modélisant le problème de gestion des dépendances.

Ce calcul se fait par transformation d'une formule QCL générale à une forme normale, ne contenant que des disjonctions ordonnées dans des formules de base.

Définition 6 (Formule QCL de base, optionalité) *On appelle formule QCL de base toute formule QCL de la forme $\phi = \alpha_1 \times \alpha_2 \times ... \times \alpha_n$ où α_i sont des formules propositionnelles classiques. On appelera n l'optionalité de ϕ.*

Il est toujours possible de traduire une formule QCL en une formule QCL de base en appliquant les règles de réécriture suivantes (on suppose que $n \leq m$ sans perte de généralité puisque les connecteurs logiques $\wedge$ et $\vee$ sont commutatifs) :

$$(A_1 \times ... \times A_n) \vee (B_1 \times ... \times B_m) = (A_1 \vee B_1) \times ... \times (A_n \vee B_n) \times B_{n+1} \times .... \times B_m \quad (3.7)$$

$$(A_1 \times ... \times A_n) \wedge (B_1 \times ... \times B_m) = C_1 \times C_2 \times ... \times C_m \text{ avec} \quad (3.8)$$

$$C_i = ((A_1 \vee ... A_i) \wedge B_i) \vee (A_i \wedge (B_1 \vee B_2 \vee ... B_i) \forall i \in 1..n \quad (3.9)$$

$$C_j = ((A_1 \vee ... A_n) \wedge B_j) \forall j \; n < j \leq m \quad (3.10)$$

$$\neg(A_1 \times ... \times A_n) = \neg(A_1 \vee ... \vee A_n) \quad (3.11)$$

Dans le cas de préférences simples, comme $(kmail_4 \times kmail_3) \wedge (firefox_{36} \times firefox_{25})$, on obtient la formule QCL de base $(kmail_4 \wedge firefox_{36}) \times (((kmail_3 \vee kmail_4) \wedge firefox_{25}) \vee (kmail_3 \wedge (firefox_{36} \vee firefox_{25})))$.

Dans le cas de préférences imbriquées, par exemple la formule
$gnome_{230} \rightarrow (evolution_{230} \times thunderbird_{30}) \wedge$
$\neg gnome_{230} \rightarrow (thunderbird_{30} \times (kmail_3 \vee kmail_4) \times evolution_{230})$,
on obtient une formule QCL de base plus complexe.

On normalise d'abord toutes les clauses avec la première règle de réécriture :
$gnome_{230} \rightarrow (evolution_{230} \times thunderbird_{30}) \equiv (gnome_{230} \rightarrow evolution_{230}) \times thunderbird_{30}$ [1]
$\neg gnome_{230} \rightarrow (thunderbird_{30} \times (kmail_3 \vee kmail_4) \times evolution_{230}) \equiv$
$(\neg gnome_{230} \rightarrow thunderbird_{30}) \times kmail_3 \vee kmail_4 \times evolution_{230}$

On applique ensuite la seconde règle sur la conjonction de ces deux contraintes pour obtenir la formule QCL de base : $(gnome_{230} \rightarrow evolution_{230}) \wedge (\neg gnome_{230} \rightarrow thunderbird_{30}) \times ((gnome_{230} \rightarrow (evolution_{230} \vee thunderbird_{30}) \wedge (kmail_3 \vee kmail_4)) \vee ((\neg gnome_{230} \rightarrow (thunderbird_{30} \vee kmail_3 \vee kmail_4)) \wedge thunderbird_{30} \times (gnome_{230} \rightarrow evolution_{230} \vee thunderbird_{30}) \wedge evolution_{230}$

Le même exemple peut être visualisé par des informations stratifiées comme dans la figure 3.1. On dispose de deux préférences locales, représentées par des informations stratifiées dans un contexte partculier sur la gauche. Ces informations sont normalisées pour être rassemblées dans un même référentiel (informations stratifiées sur la droite).

1. On peut trouver cette transformation étrange, et s'attendre plutôt à $(gnome_{230} \rightarrow evolution_{230}) \times (gnome_{230} \rightarrow thunderbird_{30})$. Cependant, cette traduction a du sens car la seconde option, $thunderbird_{30}$, ne sera étudiée que lorsque la première option sera falsifiée, c'est à dire $gnome_{30}$ installé et $evolution_{230}$ non installé.

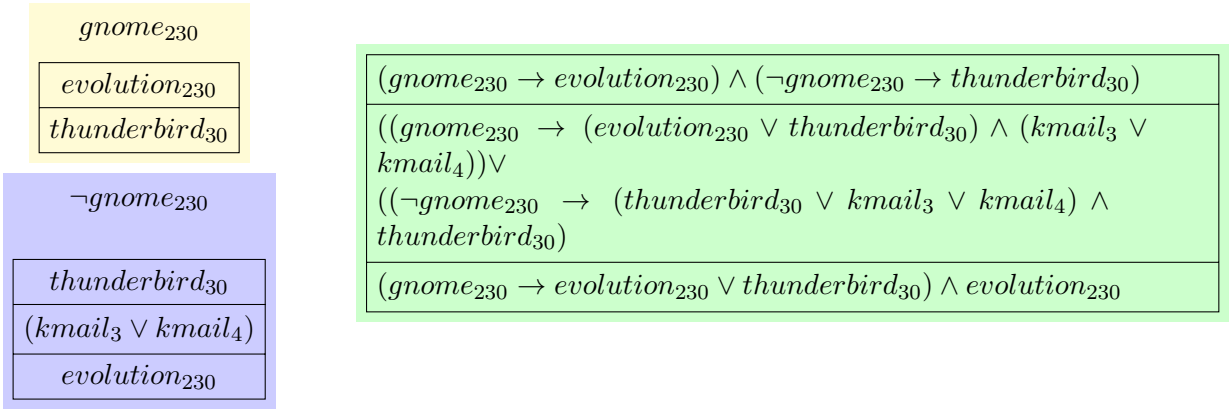


FIGURE 3.1 – Formules QCL : préférences imbriquées (gauche) vs forme basique (droite).

On peut remarquer sur les exemples précédents que les propriétés suivantes résultent de nos règles de transformation :

- Les règles de transformations ne changent pas l'optionalité de la formule, i.e. le nombre d'imbrications de disjonctions ordonnées dans une formule QCL de base est égal au nombre d'imbrications de disjonctions ordonnées dans la formule initiale.
- La seconde règle de réécriture ne préserve pas le format CNF et peut produire une formule de taille exponentielle par rapport à l'optionalité de la formule.

Dans le cadre de préférences de l'utilisateur, ou de préférences entre les versions de différents logiciels ou bibliothèques, les formules seront toujours des conjonctions de formules QCL de base ne contenant que des littéraux, ayant une optionalité maximum m très inférieure au nombre de formules à considérer.

Une fois la formule QCL normalisée, il est possible de calculer un modèle de cette formule en calculant les modèles lexico-préférés de la formule stratifiée correspondante. Ce calcul peut se faire à l'aide d'un simple prouveur SAT [28] (article joint chapitre 9).

3.4 Mélanger le tout

Nous venons de présenter sommairement divers formalismes pour exprimer les préférences rencontrées dans les problèmes de gestion des dépendances. Nous avons illustré pour chaque formalisme le traitement d'une préférence unique. Il est nécessaire en réalité de prendre en compte l'ensemble de ces préférences. Il en résulte deux problèmes principaux :

- comment fusionner les différents formalismes ?
- comment agréger les différentes préférences ?

Il est possible de traduire un problème MaxSat en problème d'optimisation pseudo-booléen en ajoutant une nouvelle variable par clause souple. C'est ce que nous faisons dans Sat4j MaxSat (cf . section 4.2.4). Il est aussi possible de traduire l'inférence lexicographique sur une base stratifiée en un problème MaxSat, pour lequel le poids des clauses des strates les plus hautes est supérieur au poids de toutes les clauses des strates inférieures. Donc la modélisation grâce à la logique QCL peut aussi se modéliser *in fine* en problème d'optimisation pseudo-booléen, et donc être résolu en pratique par un prouveur pseudo-booléen.

Le problème que nous rencontrons est alors un problème classique d'optimisation multi-critère : nous devons optimiser plusieurs critères (la fonction objectif initiale du problème pseudo-booléen, la fonction objectif de la traduction du problème MaxSat pour les dépendances optionnelles et enfin la fonction objectif de la traduction de l'inférence lexicographique).

Chapitre 4

La bibliothèque SAT4J

4.1 De JSat à Sat4j

Durant ma thèse, l'ensemble des algorithmes proposés avaient été implémentés dans un outil en Java appelé « Autour De SAT » (ADS) [108]. Le moteur de cet outil était un algorithme DPLL [64, 63] de base pour calculer des P-impliquants M-premiers [45], un ensemble de modèles minimaux par rapport à un sous-ensemble cohérent de littéraux donné de la formule d'entrée. Le principal défaut de cet outil était que le moteur SAT était loin de l'état de l'art à cette époque. D'où l'idée d'en créer un nouveau, plus efficace.

Dès la fin de ma thèse, lors de mon stage post-doctoral en Australie, j'ai entrepris de développer un nouveau moteur SAT pour ADS, appelé JSat. A la différence du moteur SAT d'ADS, il s'agissait avant tout d'un prouveur SAT flexible en Java. Il m'a permis d'étudier diverses utilisations de la propagation unitaire, pour détecter des littéraux impliqués ou équivalents dans des instances SAT, notamment pour créer des pré-processeurs [109] (article joint chapitre 11). Ce moteur SAT était diffusé sous la forme d'une bibliothèque libre sous licence GNU LGPL, afin de favoriser son utilisation par d'autres groupes de recherche, voire dans des logiciels propriétaires. A ma connaissance, la seule utilisation extérieure de cette bibliothèque concerne le web sémantique, plus exactement la correspondance d'ontologies dans l'outil S-Match [84].

Si les prouveurs de JSat étaient plus efficaces que le moteur d'ADS, l'architecture des prouveurs SAT utilisée dans JSat était toujours DPLL. Avec l'apparition de prouveurs CDCL plus efficaces [137, 74], l'architecture DPLL ne convenait plus à nos besoins, le but étant de pouvoir résoudre des instances SAT issues de problèmes réels. Dans le cadre d'un partenariat luso-français de la conférence des présidents d'université, nous avons eu l'occasion de mettre en place, avec Gilles Audemard et Olivier Roussel au CRIL, et Joao Marques Silva et Ines Lynce à l'INESC-ID, la bibliothèque libre de prouveur SAT OpenSAT [18]. L'objectif était de fournir une API en Java permettant de facilement intégrer les services d'un prouveur SAT dans une application Java, avec en plus deux implémentations de cette API : une implémentation de référence flexible et libre intégrée au projet OpenSAT développée au CRIL, et une autre implémentation, JQuest2, développée à l'INESC, représentant l'état de l'art des prouveurs CDCL. L'implémentation de référence fut conçue en utilisant des structures de données génériques, ce qui produisit une bibliothèque de prouveurs SAT flexible, capable de gérer à la fois des algorithmes de type DPLL et des algorithmes de type CDCL, mais inefficace (19ème durant la compétition SAT 2003 [115], juste derrière SATO). De même, les résultats du prouveur JQuest2, contenant toutes les techniques de l'état de l'art, et des implémentations non triviales de ces techniques, ne

parvinrent pas au niveau espéré par INESC (15ème durant la compétition SAT 2003, juste devant zchaff) : la perte d'efficacité due au langage Java s'avérait trop importante. Le projet OpenSAT s'arrêta l'année suivante, après avoir permis au CRIL de développer son premier prouveur QBF, OpenQBF, basé sur une version étendue de DPLL pour les formules booléennes quantifiées.

L'année 2003 marque pour la communauté SAT une année importante : l'apparition de Minisat [74], dont la conception résulte de l'expérience de deux auteurs de prouveurs SAT, Niklas Eén pour SATZoo et Niklas Sörensson pour SATNik, et de la volonté de concevoir un prouveur SAT CDCL simple à comprendre. Niklas Eén avait été invité à soumettre une description de SATZoo dans les actes de SAT 2003 (entre 2002 et 2004 seule une sélection des papiers présentés à la conférence SAT figurait dans ses actes) car SATZoo avait remporté les deux catégories (SAT et SAT+UNSAT) sur les instances fabriquées durant la compétition SAT 2003 [115]. Minisat était la réponse à cette invitation. C'est la lecture de cet article qui m'a donnée l'idée de créer une implémentation en Java de la spécification de Minisat. L'accès au code source de l'application (900 lignes de code C++ avec les structures de données) permettait de compléter la description du prouveur avec tous les détails de l'implémentation. Le but de ce prouveur étant d'être efficace, j'ai utilisé un moyen similaire à celui de Sharad Malik lorsque Chaff a été conçu : j'ai demandé à chacun de mes étudiants de première année de master (maîtrise à l'époque) de réaliser une implémentation de Minisat en Java. Toutes ces implémentations ont été évaluées contre ma propre implémentation de la spécification. Le prouveur de Frédéric Laihem se révéla deux fois plus rapide que le mien, malgré un espace de recherche parcouru plus grand : l'efficacité de son prouveur se situait au niveau de la gestion des littéraux et des variables, où la structure de données utilisée violait les principes élémentaires de la programmation orientée objet pour fournir une efficacité redoutable. La première version de Sat4j [113] soumise à la compétition SAT 2004, résulte de la fusion de mon implémentation de Minisat en Java et de la structure de gestion des littéraux de Frédéric Laihem. La particularité du prouveur soumis à la compétition SAT 2004 était de n'apprendre qu'une partie des clauses dérivées de l'analyse de conflits, d'où le nom *MiniLearning*. Ce prouveur se classa 8ème pour l'ensemble des instances issues d'applications, devant des prouveurs comme Quantor ou SATZoo [116]. Notre objectif était atteint.

Dès l'année suivante, nous avons intégré dans Sat4j un prouveur pseudo-booléen [150] dont l'inférence est basée sur les plans-coupe [88, 94], comme proposé dans les prouveurs Galena [46] et PBChaff [70]. Ce prouveur participa à la première évaluation de prouveurs pseudo-booléens organisée par Vasco Manquinho et Olivier Roussel [129]. Nous avons aussi intégré un prouveur CSP basé sur une traduction simple de CSP en extension via SAT pour la première compétition de prouveurs CSP co-organisée par des membres du CRIL [3]. Cette traduction sera étendue aux contraintes en compréhension l'année suivante, en utilisant un moteur JavaScript pour évaluer ces contraintes. 2005 a aussi été l'année de l'adoption du projet Sat4j au sein de la forge logicielle du consortium ObjectWeb, (devenu OW2 depuis Janvier 2008) [6] : l'utilisation d'une forge publique était une volonté personnelle afin de favoriser l'intervention de divers acteurs autour de ce projet. En 2006, nous avons introduit la résolution de problèmes MaxSat par traduction en un problème d'optimisation pseudo-booléen. Ces fonctionnalités ont été améliorées au fil des ans, après analyse des résultats des différentes évaluations et compétitions, et aussi grâce au retour de quelques utilisateurs sur des applications précises, comme par exemple le prouveur MaxSat et de l'application de *British Telecom* [121]. La version 2.0 de Sat4j a apporté la modularité, avec la découpe en composants des diverses fonctionnalités de la bibliothèque

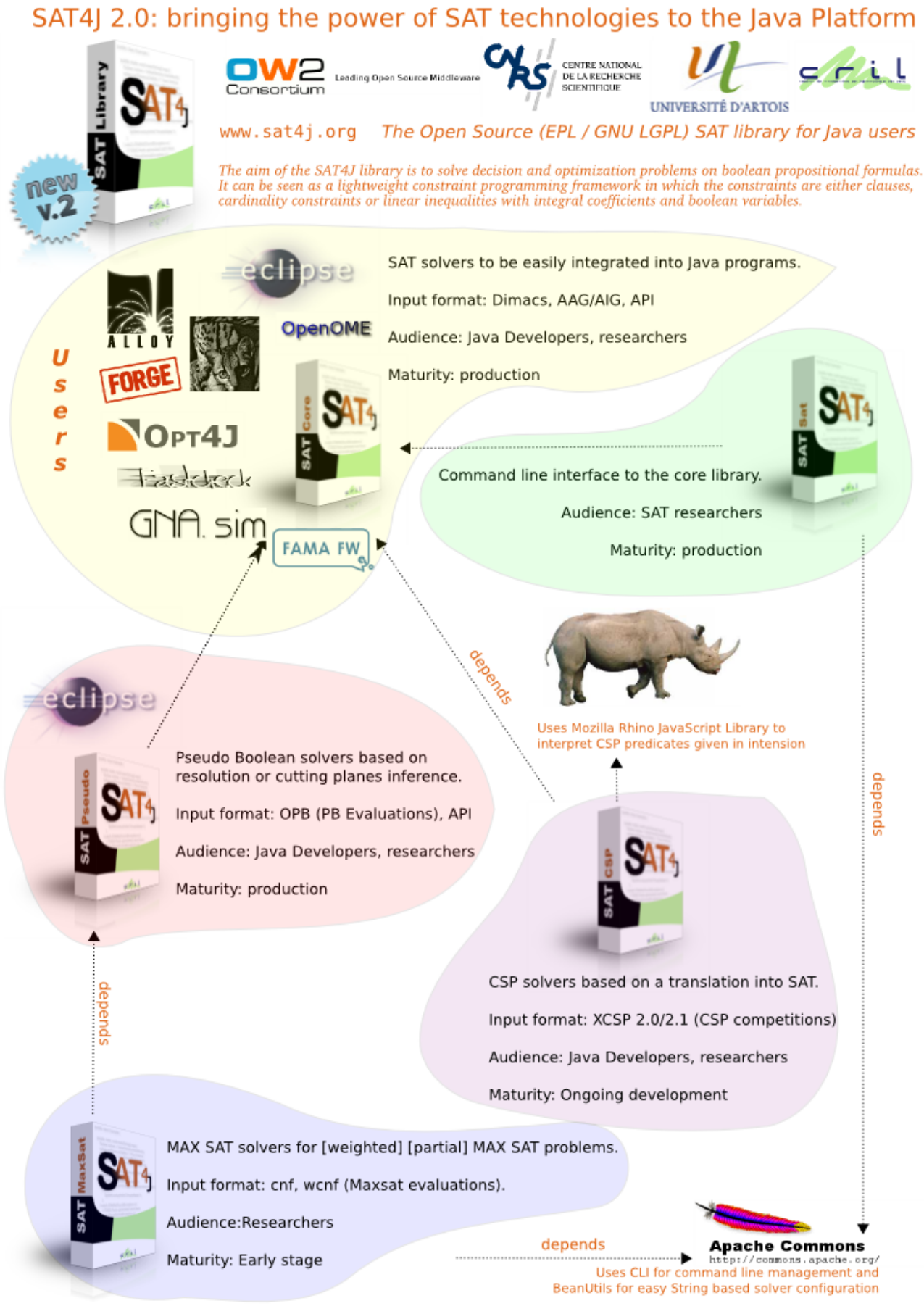


FIGURE 4.1 – Architecture de Sat4j depuis la version 2.0

conçue au départ de manière monolithique : la figure 4.1 montre les différents composants de Sat4j 2.0 qui n'ont pas changé depuis. En juin 2008, Eclipse 3.4 sortait en incluant un outil de gestion de dépendances basé sur Sat4j 2.0, utilisant un couplage faible avec l'application. De nombreuses améliorations permettant une meilleure intégration des diverses fonctionnalités de l'outil dans les applications Java ont été ajoutées pour faciliter l'intégration de Sat4j dans les nouvelles versions d'Eclipse. D'autres améliorations et fonctionnalités ont été initiées par le retour des utilisateurs.

On peut se poser la question de l'intérêt de développer soi-même un prouveur SAT, alors qu'il en existe de nombreux très efficaces disponibles, dont certains, comme Picosat [31] ou Minisat [74] par exemple, sont libres, très utilisés dans la communauté et particulièrement bien mis à jour par leurs auteurs. La première raison pour laquelle nous avons entrepris ce travail était de disposer d'un prouveur SAT en Java. Un seul autre prouveur SAT est actuellement disponible en Java à notre connaissance (JAT [59]), et il est utilisé comme base d'un prouveur SMT (SVTK). Ses fonctionnalités sont transverses à celles de Sat4j. En terme de performances, JAT semble moins performant que Sat4j : 39 instances résolues sur le jeu des 100 instances de la SAT Race 2006 [159] pour JAT révision 1045, contre 58 pour Sat4j 2.2 [1]. Les prouveurs JQuest et JQuest2, développés entre 2002 et 2003 à INESC, n'ont jamais été libres, et ne sont plus développés. Si Sat4j se trouve livré en standard avec de nombreux logiciels Java (Alloy, Eclipse, GNA.sim), c'est avant tout parce qu'il est développé en Java. De plus, nous souhaitions disposer un prouveur SAT capable de prendre en compte différents types de contraintes. Seule la version initiale de Minisat (jusqu'à la version 1.13) permettait cela. Assurer cette fonctionnalité nous aurait obligé à maintenir une version alternative de Minisat. Enfin, développer un prouveur SAT est le meilleur moyen de comprendre son fonctionnement interne. Développer un prouveur CDCL m'a personnellement convaincu en 2004 que les prouveurs CDCL et DPLL fonctionnaient de façon complètement différente [25, 146], après avoir pensé et défendu le contraire pendant plusieurs années.

A ce jour, nous disposons d'une bibliothèque de prouveurs SAT, MaxSat et pseudo-booléen raisonnablement efficaces en Java, flexible, utilisée dans de nombreuses applications dans le monde entier. Le fait de l'avoir conçue en Java et diffusée sous forme de logiciel libre a permis son adoption dans la plate-forme ouverte Eclipse, ce qui en fait le prouveur pseudo-booléen le plus utilisé dans le monde. Sat4j est passé en quelques années du statut de simple prototype de recherche à un composant clé d'une application très diffusée dans le monde Java.

4.2 Aperçu général des fonctionnalités

Sat4j est avant tout un moteur SAT de type CDCL capable de prendre en compte divers types de contraintes. Par défaut, la bibliothèque offre la possibilité de travailler avec des clauses, des contraintes de cardinalité et des contraintes pseudo-booléennes. Il est assez aisé pour un utilisateur averti de la bibliothèque de mettre en oeuvre ses propres contraintes. Deux méthodes d'inférence sont proposées : la résolution [148] et les plans-coupe vus comme une généralisation de la résolution [94]. De plus, la bibliothèque permet de résoudre à la fois des problèmes de décision (SAT, *Pseudo-Boolean Satisfaction*) ou des

1. sur un quad-core 8400 à 2,66GHz et 4Go de RAM, avec un temps limite d'exécution de 900 secondes CPU par instance.

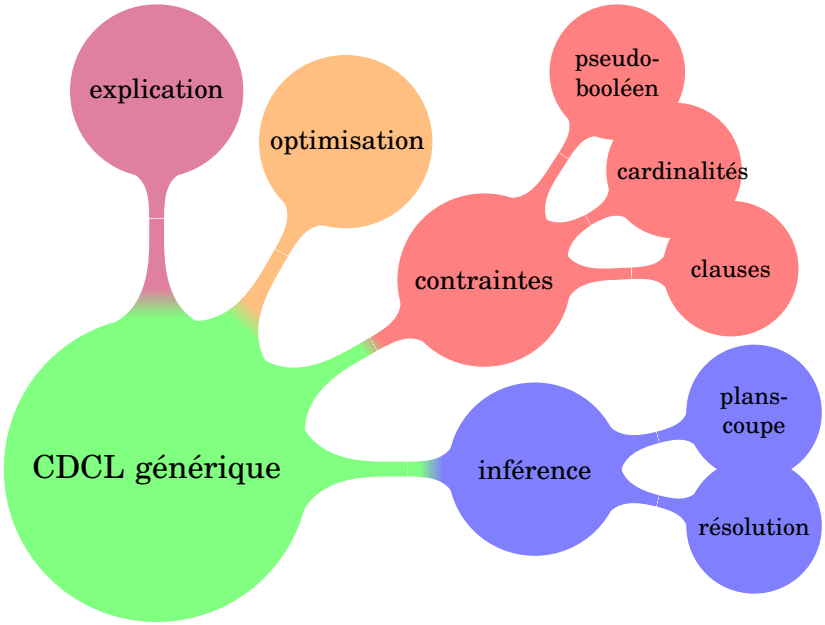


FIGURE 4.2 – Concepts utilisés dans Sat4j

problèmes d'optimisation (MaxSat, PBO, WBO). Enfin, nous avons ajouté récemment un support non intrusif de détection de noyau incohérent pour apporter à la bibliothèque la possibilité de justifier une absence de solution. Ces fonctionnalités sont résumées sur la figure 4.2. La figure 4.3 récapitule les liens entre ces fonctionnalités et les problèmes propositionnels à portée de Sat4j.

4.2.1 A la base, un moteur SAT CDCL

Le prouveur *Sat-Core* qui sous-tend toute l'architecture est basé sur l'implémentation originelle de Minisat 1.x [74] : le moteur, dirigé par les conflits avec apprentissage de clauses (CDCL), et la gestion de l'activité des variables n'ont pas été modifiés. La plupart des composants-clés du prouveur sont configurables : heuristique de choix de variables, structures de données, techniques de simplification des clauses apprises, gestion

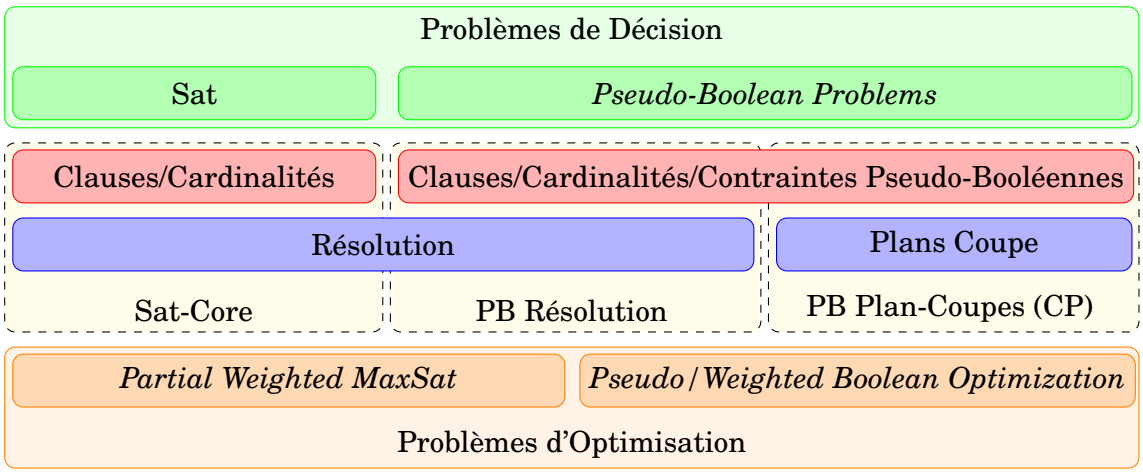


FIGURE 4.3 – Portée de la bibliothèque SAT4J et de ses prouveurs

des clauses apprises, politique de redémarrages, etc. Le tableau 4.1 récapitule les principales options offertes par le prouveur SAT de Sat4j, ainsi que la **configuration par défaut Sat4j 2.2**. En 2007, Frank Hutter a utilisé ParamILS pour rechercher la configuration optimale de Sat4j [97]. Il s'est avéré qu'à l'époque, la configuration par défaut était optimale. Dans Sat4j 2.2, la configuration par défaut est la suivante : les *redémarrages rapides* suivent la stratégie in/out présentée par Armin Biere dans Picosat [31] ; la *minimisation des clauses apprises* de Minisat 1.14 est utilisée à la fin de l'analyse du conflit ; la sélection de variables suit celle de RSAT [145] ; enfin, le prouveur conserve les clauses dérivées construites avec des littéraux provenant de peu de niveaux de décisions différents comme proposé par Glucose [19].

Fonction	Alternatives
Structures de données	**Watched Literals** [137], Head/Tail [174]
Redémarrages	Minisat [74], **Picosat (In/Out)** [31], Luby [124]
Simplification de clauses	Sans, Simple, **Récursive** [167]
Sauvegarde de phase	Sans,**RSAT** [145], ***dernière clause apprise***
Apprentissage	Sans, ***Limité***, **Total**
Gestion des clauses apprises	Sans, basé sur la mémoire, **glucose** [19]

TABLE 4.1 – Principales options de configuration du prouveur SAT de Sat4j.

Structure de données paresseuse La bibliothèque fournit deux implémentations d'une structure de données paresseuse pour SAT : les littéraux observés (*watched literals*) comme proposé dans Minisat, et une implémentation de la structure tête/queue (*Head/Tail*) où les littéraux observés sont placés en début et en fin de la clause. A la différence de la solution utilisée dans SATO [174], dans laquelle les marqueurs de début et fin de clause sont déplacés lors de la falsification des littéraux, notre implémentation échange les littéraux à l'intérieur de la clause pour maintenir si possible des littéraux non affectés à chaque extrémité de la clause, comme dans les versions récentes de Picosat. L'ordre des littéraux n'est plus préservé, comme c'est le cas dans l'implémentation des littéraux observés de Minisat. Cependant, les deux structures de données ont la même propriété de ne provoquer aucun coût durant un retour arrière. La première structure est plus simple et plus élégante à programmer, et, de notre expérience, elle est légèrement plus efficace. Néanmoins, la deuxième structure de données présente l'avantage de ne pas avoir été brevetée. Une analyse fine des performances des deux structures de données est difficile car l'espace de recherche parcouru est dépendant de ces structures, donc différent dans les deux cas.

Minimisation de clause La simplification des clauses dérivées de l'analyse de conflit présentée dans [167] a été rendue générique, afin de pouvoir prendre en compte des contraintes ou des structures de données de natures différentes. Cela a été rendu possible en relâchant deux hypothèses : i) le premier littéral de la raison est satisfait et ii) tous les autres littéraux de la raison sont falsifiés. Ces hypothèses ne sont pas vérifiées dans le cas des contraintes pseudo-booléennes car plusieurs littéraux peuvent être propagés dans ces contraintes, en laissant le reste des littéraux soit falsifiés soit indéfinis. La première hypothèse n'est pas non plus vérifiée par la structure de données tête/queue. D'un point de vue algorithmique, la procédure de simplification a été rendue générique en appliquant la procédure récursive uniquement sur les littéraux falsifiés de la contrainte, en utilisant un test spécifique pour appliquer la valeur de vérité d'un littéral à tous les littéraux d'une

contrainte. Cette procédure est donc strictement plus lente que la procédure originale dans le cas des clauses utilisant la structure de données des littéraux observés.

Contrôle de l'apprentissage des clauses dérivées par l'analyse des conflits Dès le début, Sat4j fut conçu de telle sorte que l'apprentissage des clauses puisse être réalisé de manière indépendante du reste du prouveur. Dans un prouveur CDCL, il est nécessaire de garder une trace des clauses dérivées par l'analyse de conflit uniquement dans le but d'expliquer plus tard la raison de la satisfaction d'un littéral lors de l'analyse d'un autre conflit. Cependant, il n'est pas nécessaire de prendre en compte cette clause lors de la propagation des littéraux : il n'est pas nécessaire d'observer ses littéraux. Sat4j conserve donc toujours les clauses dérivées lors de l'analyse de conflit comme raison d'une affectation, mais offre plusieurs stratégies pour déterminer si une clause doit être apprise : ne rien apprendre, filtrer les clauses selon leur taille absolue ou relative, tout apprendre. On pourrait très facilement appliquer une stratégie basée sur le nombre de niveaux de décision qu'utilise Glucose [19] pour effacer ces clauses *a posteriori* afin d'éviter de les apprendre *a priori*. Le problème principal est de déterminer le nombre maximal de niveaux des clauses à apprendre. Cette flexibilité offerte par Sat4j est venue au départ d'une mauvaise compréhension des prouveurs CDCL. En effet, les années passées à utiliser et concevoir des prouveurs DPLL me poussaient à croire que l'heuristique était le composant le plus important d'un prouveur SAT, et que la force de Chaff résidait essentiellement dans son heuristique capable de s'adapter à chaque instance. L'apprentissage de clause ne devenait qu'un moyen d'améliorer l'heuristique, pas un composant essentiel en lui-même. Sat4j m'a permis de vérifier que cette hypothèse était fausse. Une autre hypothèse de travail est que toutes les clauses dérivées ne sont pas importantes. L'utilisation de techniques d'effacement agressives des clauses apprises comme dans Glucose semble aller dans cette direction. Nous avons limité pendant longtemps l'apprentissage de clauses à celles d'une taille inférieure à 10% du nombre de variables (critère déterminé empiriquement à partir d'expérimentations faites sur les instances de la compétition SAT en 2003). En pratique, cette approche limitait peu l'apprentissage de clauses dans les instances de type Applications, car le nombre de variables y est très important. Nous n'avons pas encore pris le temps d'étudier sérieusement des stratégies de contrôle d'apprentissage de clauses. L'implantation de telles stratégies reste néanmoins très facile à effectuer avec Sat4j.

Utilisation de la polarité des variables apparaissant dans les dernières clauses apprises L'heuristique de Berkmin [85] me paraissait intéressante, mais difficile à mettre en oeuvre efficacement. L'idée principale de cette heuristique est de s'assurer que les dernières clauses apprises sont satisfaites, tout en focalisant la recherche sur les variables non affectées de ces clauses. En pratique, on ne sait pas immédiatement qu'une clause est satisfaite quand on utilise une structure de données paresseuse. D'où l'idée d'utiliser une technique facile à mettre en oeuvre qui ne garantit pas que toutes les dernières clauses sont satisfaites mais qui va dans ce sens. L'idée est de garder pour chaque variable la polarité dans laquelle elle apparaît dans la dernière clause apprise la contenant, et de brancher d'abord sur cette phase lors d'une décision. De cette manière, les clauses récemment apprises doivent être satisfaites. Cependant, cette approche ne permet pas de focaliser la recherche sur les variables apparaissant dans les dernières clauses apprises, comme l'heuristique originale le demande.

4.2.2 Un pivot, le prouveur pseudo-booléen

Notre objectif a rapidement été de construire un prouveur pseudo-booléen disposant d'un système de preuve plus fort que la résolution. Il s'agissait notamment d'utiliser les plans-coupe pour dériver des contraintes pseudo-booléennes et non des clauses lors de la phase d'analyse des conflits, à la lumière de travaux sur Galena [46] et PBChaff [71]. Notre prouveur basé sur les plans-coupe fonctionne de la façon attendue sur des exemples jouets comme les pigeons [90], et il se révèle très efficace sur certains problèmes, comme ceux issus d'une traduction de problèmes d'agrégation d'ordres d'intervalle [110] par exemple. Cependant, il est incapable de résoudre de nombreux problèmes qui sont triviaux pour d'autres prouveurs pseudo-booléens. Nous avons donc décidé de maintenir en parallèle un prouveur pseudo-booléen dont le système de preuve reste limité à la résolution. En pratique, ce dernier est le plus efficace sur de nombreux benchmarks (cf. section 4.3.2).

Règles d'inférence sur les contraintes pseudo-booléennes

En calcul propositionnel, il y a essentiellement deux règles d'inférence qui peuvent être appliquées sur une formule en forme normale conjonctive : la *résolution* [148] et la *fusion*. Cette dernière règle est souvent oubliée car habituellement les clauses sont vues comme des ensembles de littéraux, et la règle de fusion est obtenue par l'union ensembliste. Néanmoins, la fusion est nécessaire pour la complétude de la procédure de réfutation. La fusion joue également un rôle central dans le calcul des prédicats où des littéraux peuvent être fusionnés par calcul du plus grand unificateur. Dans ce cadre également, la résolution et la fusion sont toutes deux nécessaires pour obtenir une procédure de réfutation complète. En revanche la fusion n'est plus triviale comme elle l'est en calcul propositionnel.

Avec les contraintes pseudo-booléennes, nous avons basiquement besoin des deux mêmes règles pour obtenir une procédure de réfutation complète, mais nous pouvons également utiliser quelques autres règles d'inférence qui n'ont pas leur équivalent dans le calcul propositionnel. La règle correspondant à la résolution est appelée *plan-coupe* et elle calcule une combinaison linéaire positive de deux contraintes pseudo-booléennes.

Pour une contrainte pseudo-booléennes en forme (3.1) :

$$\textit{plan-coupe:} \quad \frac{\begin{array}{c}\sum_i a_i.x_i \geq k \\ \sum_i a'_i.x_i \geq k'\end{array}}{\begin{array}{c}\sum_i(\alpha.a_i + \alpha'.a'_i).x_i \geq \alpha.k + \alpha'.k' \\ \text{avec } \alpha > 0 \text{ et } \alpha' > 0\end{array}}$$

En général, les coefficients pour la combinaison linéaire sont choisis de manière à éliminer au moins une variable, i.e. tels que $\exists i, \alpha.a_i + \alpha'.a'_i = 0$. Contrairement à ce qui se passe avec la résolution en calcul propositionnel, on peut former une combinaison qui n'élimine aucune variable. On peut également forcer une combinaison qui élimine plus d'une variable (ce qui est impossible en calcul propositionnel).

La seconde règle essentielle correspond à la règle de fusion qui est appelée la *saturation*. Si un coefficient a_j est plus grand que le degré k, alors il peut être remplacé par k (ce qui revient à faire une fusion de certaines occurrences de l_j). Cette règle ne peut s'appliquer que sur des contraintes PB de la forme (3.2) :

$$\textit{saturation:} \quad \frac{a_j.l_j + \sum_{i \neq j} a_i.l_i \geq k \text{ avec } a_j > k}{k.l_j + \sum_{i \neq j} a_i.l_i \geq k}$$

Une manière simple de comprendre cette règle est de considérer que, si l_j est vrai, la contrainte sera satisfaite même si a_j est réduit à k. Ainsi, réduire les coefficients supérieurs au degré ne change pas les solutions pour la contrainte.

En appliquant la règle de saturation sur les contraintes en forme (3.2), nous obtenons maintenant une forme normale pour les contraintes PB :

$$\sum_i a_i.l_i \geq k, \qquad a_i, k \in \mathbb{N}, \forall i a_i \leq k \tag{4.1}$$

La règle d'*arrondi*↑ permet de diviser le degré et chaque coefficient par un entier strictement positif et d'arrondir à l'entier supérieur chaque nombre obtenu :

$$\textit{arrondi}\uparrow: \quad \frac{\sum_i a_i.l_i \geq k}{\sum_i \lceil a_i/\alpha \rceil .l_i \geq \lceil k/\alpha \rceil \text{ avec } \alpha > 0}$$

Avec cette règle, on peut donc inférer une simple clause à partir d'une contrainte pseudo-booléenne en appliquant la règle de l'arrondi↑ avec $\alpha = k$ suivie de la règle de saturation.

La dernière régle d'inférence utilisée dans notre cadre est la *réduction*. Elle consiste à supprimer un des littéraux de la contrainte et à ôter du degré son coefficient :

$$\textit{réduction:} \quad \frac{\sum_i a_i.l_i \geq k}{\sum_{i \neq j} a_i.l_i \geq k - a_j}$$

Les plans-coupe ont été introduits dans le cadre de la programmation linéaire en nombres entiers par R.Gomory en 1958 [88]. La relation entre les plans-coupe et la résolution a été étudiée par J.N. Hooker [94] qui a montré que les plans-coupe sont une généralisation de la résolution. Belaïd Benhamou, Lakhdar Saïs et Pierre Siegel [29] ont proposé un système de preuve complet pour un type particulier de contraintes de cardinalités. Ils considèrent une contrainte de cardinalité comme un couple (liste de littéraux, degré). Cela signifie qu'un littéral peut apparaître plusieurs fois dans une contrainte : ces contraintes sont donc équivalentes aux contraintes pseudo-booléennes.

P. Barth [21] a proposé la première extension de l'algorithme DPLL au cas des contraintes pseudo-booléennes. Depuis, plusieurs autres solveurs ont été proposés dans ce même cadre. [130] propose le premier prouveur d'optimisation pseudo-booléen basé sur l'architecture CDCL (Grasp). Nous situons nos travaux dans la continuité de ce travail, en nous basant sur les travaux qui cherchent à conjuguer l'efficacité des prouveurs SAT comme Chaff [137] avec la puissance des règles d'inférence spécifiques aux contraintes pseudo-booléennes. La première extension de Chaff aux contraintes PB est apparue dès 2002 avec PBS [9], mais le moteur d'inférence était encore basé sur la résolution. L'utilisation des plans-coupe lors de l'analyse des conflits a été proposée indépendamment dans les solveurs PBChaff [70] et Galena [46]. Une approche hybride a ensuite été proposée dans Pueblo [155]. Plus récemment, une approche SMT a été proposée pour résoudre ces problèmes [49].

Pour plus de détails concernant les plans-coupe et le calcul propositionnel, nous invitons le lecteur à se référer, par exemple, à [38]. Un chapitre complet du *Handbook of Satisfiability* est dédié aux contraintes pseudo-booléennes [150].

Prouveur pseudo-booléen basé sur la résolution

Sat4j-PB-Res est un prouveur basé sur le moteur *Sat-Core* mais qui permet de prendre en compte des contraintes de cardinalité et des contraintes pseudo-booléennes. La résolution est conservée durant l'analyse de conflit, donc seules des clauses peuvent être apprises. Un point positif de l'utilisation de la résolution est que cela nous permet d'utiliser la procédure générique de minimisation des clauses décrite précédemment. Un point négatif est qu'un

tel prouveur ne tire pas complètement parti du pouvoir d'expression des contraintes pseudo-booléennes, et qu'ainsi, il ne peut pas résoudre efficacement des instances comme le problème des pigeons dont la preuve de non satisfiabilité est exponentielle pour la résolution mais linéaire pour les plans-coupe [90]. Ce prouveur est similaire dans le principe à des prouveurs comme PBS [9] ou SATZoo, mais, d'un point de vue pratique, il utilise les techniques les plus récentes dans la résolution SAT ; il prend en compte des coefficients de taille arbitraire ; et il est compatible avec les formats d'entrée des compétitions de prouveurs pseudo-booléens.

Prouveur pseudo-booléen basé sur les plans-coupe

Sat4j-PB-CP a les mêmes caractéristiques que Sat4j-PB-Res, sauf qu'il utilise les plans-coupe à la place de la résolution pendant l'analyse de conflit, comme décrit dans les articles [46, 71]. Le système de preuve utilisé dans ce prouveur est donc plus puissant que la résolution. Néanmoins, la procédure d'analyse de conflit avec les plans-coupe est beaucoup plus coûteuse que la résolution, et nécessite de travailler avec des entiers en précision arbitraire pour éviter les problèmes de dépassement de capacité. Le prouveur est ainsi plus lent de deux ordres de grandeur en terme de nombres d'affectations de valeurs de vérité par seconde sur certains problèmes. Mais il peut résoudre certaines instances académiques (comme les pigeons) ou déterminer une valeur optimale pour des instances qui sont hors de portée du prouveur basé sur la résolution, grâce à son système de preuve plus performant.

Les prouveurs pseudo-booléens sont plus lents lorsqu'ils doivent traiter des contraintes pseudo-booléennes car nous n'avons pas trouvé de structure paresseuse efficace équivalente aux littéraux observés pour ces contraintes. [46] montrent comment généraliser la structure des littéraux observés au cadre des contraintes pseudo-booléennes. Cependant, cette approche se révèle coûteuse à mettre en oeuvre car le nombre de littéraux à observer varie durant la recherche, et nécessite de prendre en compte la valeur des coefficients, ce qui est d'autant plus pénalisant lorsque l'on travaille en précision arbitraire. C'est particulièrement le cas pour le prouveur basé sur les plans-coupe car le nombre de contraintes pseudo-booléennes augmente durant la recherche avec l'analyse de conflits. C'est la raison pour laquelle, au sein du prouveur, chaque contrainte est représentée sous sa forme la plus simple (comme une clause ou une contrainte de cardinalité si cela est possible), indépendamment de la manière avec laquelle elle est représentée à l'origine.

4.2.3 D'un algorithme de décision à un algorithme d'optimisation

Il existe de nombreuses façons de résoudre des problèmes d'optimisation en calcul propositionnel : la recherche locale, le « *branch & bound* », etc. L'une des solutions les plus simples est de transformer la résolution d'un problème d'optimisation en une résolution successive de problèmes de satisfaction. Ainsi, soit un problème d'optimisation donné par un ensemble de contraintes (clauses, cardinalités, pseudo-booléennes) muni d'une fonction d'objectif de la forme $objFct = \sum a_i l_i$ que l'on cherche à minimiser. La première étape est de chercher une solution satisfaisant les contraintes. Lorsqu'une solution M est trouvée, la valeur de la fonction objectif pour cette solution est calculée ($y = objFct(M)$) et une nouvelle contrainte pseudo-booléenne $objFct < y$ est ajoutée à l'ensemble des contraintes pour tenter de trouver une solution strictement meilleure. Comme toutes les contraintes obtenues par ce renforcement sont de la forme $objFct < y'$ avec $y' < y$, seule la dernière contrainte pseudo-booléenne ainsi créée est conservée pour le problème. En revanche, toutes les autres contraintes apprises par analyse de conflits durant la recherche sont conservées. Lorsque

le prouveur ne trouve plus de solution, la dernière solution obtenue est prouvée optimale. Cette approche constitue l'algorithme 1.

Une telle approche est souvent appelée *recherche linéaire*, par contraste avec l'approche binaire plus classique qui utilise à la fois un minorant et un majorant et qui procède par recherche dichotomique pour localiser la valeur optimale. Dans notre cas, le temps mis par Sat4j pour prouver l'incohérence (et donc trouver un minorant) dans les problèmes pseudo-booléens est souvent beaucoup plus long que pour trouver une solution. C'est pourquoi nous utilisons plutôt une recherche linéaire.

entrée : Un ensemble de clauses, de contraintes de cardinalités et de contraintes pseudo-booléennes ensDeContraintes, et une fonction d'objectif à minimiser objFct
sortie : un modèle de ensDeContraintes, ou UNSAT si le problème est insatisfiable.

réponse ← `estSatisfiable` (ensDeContraintes);
if réponse *est* UNSAT **then**
  return UNSAT
end
repeat
  modèle ← réponse ;
  réponse ← `estSatisfiable` (ensDeContraintes ∪ {objFct < objFct (modèle)});
until *(*réponse *est* UNSAT*)*;
return modèle ;

Algorithme 1: Optimisation par renforcement (recherche linéaire)

Année après année, nous avons réduit les différences entre les options actives dans les prouveurs SAT et pseudo-booléens. Les modifications les plus remarquables pour le cas des problèmes d'optimisation sont :

- l'heuristique de sélection des variables prend en compte les littéraux de la fonction d'objectif. Les littéraux qui ont un poids négatif seront d'abord satisfaits, tandis que les littéraux qui apparaissent avec un poids positif seront d'abord falsifiés. La valeur absolue des poids est utilisée pour initialiser le poids des variables dans l'heuristique.
- la version 2.2 fournit également une meilleure intégration de la stratégie des redémarrages à l'intérieur de la procédure d'optimisation. Le contexte de la recherche est alors pris en compte, i.e., il n'est pas réinitialisé à chaque appel au prouveur SAT dans l'algorithme 1.

4.2.4 Résolution par traduction : les prouveurs MaxSat

Sat4j traduit les problèmes MaxSat partiellement pondérés (acronyme PWMS, *Partial Weighted MaxSat* en anglais) dans des problèmes d'optimisation pseudo-booléens. Comme toutes les autres variantes (MaxSat, MaxSat partiel et MaxSat pondéré) peuvent être considérées comme des cas particuliers de PWMS, une telle aproche peut être utilisée pour toutes les catégories des évaluations MaxSat. L'idée est d'ajouter une nouvelle variable pour chaque clause souple pondérée qui représente le fait que cette clause a été violée, et de traduire le problème de maximisation sur les contraintes souples pondérées en un problème de minimisation d'une fonction linéaire sur ces variables. Formellement, supposons que $T = \{(w_1, C_1), (w_2, C_2), \ldots, (w_n, C_n)\}$ est l'ensemble original de clauses pondérées du problème PWMS. Nous traduisons alors ce problème T en $T' = \{s_1 \vee C_1, s_2 \vee C_2, \ldots, s_n \vee$

$C_n\}$ avec la fonction objectif $\Sigma_{i=1}^{n} w_i s_i$. Cette approche pourrait paraître inapplicable en pratique car elle requiert l'ajout d'autant de nouvelles variables de sélection que de clauses dans le problème original. Néanmoins, il y a un certain nombre de cas qui peuvent être traités différemment.

clauses dures ($w_i = \infty$) Il n'y a pas besoin de nouvelles variables pour les clauses dures, puisqu'elles doivent être satisfaites. Elles peuvent être traitées directement par le prouveur SAT ;

clauses souples unitaires Ces contraintes sont violées lorsque leur littéral est falsifié. Ainsi, il suffit de considérer directement ce littéral dans la fonction d'optimisation, et aucune nouvelle variable de sélection n'est nécessaire. Dans ce cas, la fonction d'optimisation doit minimiser $\Sigma_{i=1}^{n} w_i \overline{l_i}$ où l_i est le littéral de la clause unitaire C_i.

Ainsi, sur MaxSat partiel ou partiellement pondéré, en fonction de la proportion du nombre de clauses souples par rapport au nombre de clauses dures, le nombre de nouvelles variables peut être négligeable par rapport au nombre de variables initiales. Cela est particulièrement vrai pour les instances du problème de couverture binaire [60], qui correspond à un cas particulier du problème PWMS, où toutes les clauses souples sont unitaires. On notera de plus que la traduction de MaxSat à PBO produit toujours une instance du problème de couverture binaire, puisque toutes les contraintes sont des clauses. Si le problème original encodé en MaxSat est une couverture binaire, notre transformation reconstruit exactement le problème initial. Le prouveur pseudo-booléen utilisé dans Sat4j-MaxSat est Sat4j-PB-Res. Puisque ce prouveur est de type CDCL adapté à la résolution d'instances issues d'applications, notre approche se révèle très peu performante sur des instances PWMS aléatoires, mais donne de bons résultats sur plusieurs classes de d'instances MaxSat partiel ou partiellement pondéré industrielles de l'évaluation MaxSat 2009 [13].

4.2.5 Explication de l'incohérence

Lorsque Sat4j a été intégré à Eclipse, l'une des plus fortes demandes a été de fournir une explication en cas d'échec d'une installation. Dans la communauté SAT, on appelle cela la recherche d'un noyau incohérent minimal, ou d'une sous-formule incohérente minimale (acronyme anglais MUS, *Minimal Unsatisfiable Subformula*). De nombreuses approches existent pour calculer ces noyaux incohérents, et de nombreuses techniques ont vu le jour récemment. Eric Grégoire, Bertrand Mazure et Cédric Piette du CRIL ont par exemple proposé une approche basée sur l'utilisation de la recherche locale [178, 177]. Dans le cadre des prouveurs CDCL, il est souvent nécessaire de conserver une trace des étapes de résolution effectuées par le prouveur [176, 86]. Nous ne voulions pas implémenter une solution intrusive dans notre prouveur, afin d'éviter des problèmes de performance (en vitesse et en mémoire). De plus, nous avions besoin d'une solution qui fonctionne avec des contraintes génériques (clauses et contraintes de cardinalité pour le cas particulier d'Eclipse) car cela permet d'avoir une bijection entre un MUS sur les contraintes et une explication dans le cadre de l'application. Si une contrainte de cardinalité est représentée par un ensemble de clauses, alors il faut reconnaitre l'ensemble des clauses qui proviennent de cette traduction dans un MUS pour introduire cette contrainte de cardinalité dans l'explication au niveau application.

Notre première approche fut d'utiliser l'algorithme intégré pour ce même but dans Ilog Solver : QuickXplain [100]. Cet algorithme avait pour principal avantage d'être assez simple à mettre en oeuvre, et de fonctionner pour tout type de contraintes. Les résultats préliminaires semblaient satisfaisants et cette solution fut adoptée pour Eclipse 3.5. Cependant, des problèmes d'efficacité sont apparus lors du développement de Eclipse 3.6.

Nous avons donc décidé d'adopter une technique plus « intégrée » au prouveur SAT : nous utilisons la satisfaction sous hypothèse proposée dans Satzoo, la version initiale de Minisat [76], afin de dériver une clause, lors de l'analyse du dernier conflit, constituée uniquement de littéraux de l'hypothèse (de manière similaire à la procédure mise en place dans Minisat 1.14) comme décrit dans [16]. Comme pour MaxSat, nous ajoutons une nouvelle variable de sélection par contrainte. L'hypothèse est la négation des variables de sélection. Si la formule est incohérente, l'analyse de conflit retourne un sous-ensemble des littéraux de l'hypothèse qui correspond à un noyau incohérent. Ce noyau est alors minimisé en utilisant l'algorithme dédié QuickXplain, et en exploitant également les variables de sélection pour activer/désactiver les clauses.

Cette approche n'est sans doute pas la plus efficace pour calculer des explications (c'est la plus mauvaise d'après [16]), mais elle a l'avantage d'être cohérente avec le fonctionnement du reste de la bibliothèque qui utilise beaucoup l'ajout de nouvelles variables et la satisfaction sous hypothèse ; elle a aussi l'avantage d'être simple à mettre en oeuvre ; et plus important, ses performances sont suffisantes pour nos besoins.

4.3 Evolution de la bibliothèque

Depuis 2004, nous avons rendu publiques sept versions de Sat4j, approximativement une nouvelle version par an. Jusqu'en 2008, c'était le calendrier des différentes compétitions qui dictait la sortie d'une version publique : lorsque les compétitions de l'année étaient passées, nous sortions une nouvelle version de Sat4j comprenant les prouveurs ayant participé à ces compétitions. Depuis l'inclusion de Sat4j dans Eclipse, début 2008, une nouvelle version publique de Sat4j est publiée au second trimestre de l'année, pour qu'elle puisse être intégrée à la nouvelle version publique d'Eclipse qui sort fin juin.

L'évolution de Sat4j a été guidée jusqu'en 2007 par l'amélioration des performances du prouveur SAT sur les instances du type Applications. Depuis fin 2007, date du début de nos travaux sur la gestion des dépendances dans Eclipse, et de l'utilisation de Sat4j dans Eclipse, nous avons focalisé notre attention sur l'intégration de la bibliothèque dans les applications Java, la séparation des différents composants de la bibliothèque pour rendre plus robuste la bibliothèque et faciliter sa maintenance, et sur la résolution de problèmes pseudo-booléens.

Nous allons maintenant décrire l'évolution des performances des différents composants de Sat4j depuis sa création, et comparer les performances des versions récentes contre l'état de l'art pour chaque type de problème, en nous basant sur des résultats publics récents des compétitions SAT, pseudo-booléenne et MaxSat.

4.3.1 SAT

La figure 4.4 montre les performances des différentes versions de Sat4j dans les conditions de la première phase de la compétition SAT 2009 : 292 instances à résoudre dans la catégorie Applications, 281 dans la catégorie Fabriquées et 570 dans la catégorie Aléatoires. On note que Sat4j est bien plus performant que OpenSAT dans toutes les catégories. Si l'on considère la catégorie Applications, des progrès significatifs sont apparus avec la version 1.7 : il s'agit de la version qui marque l'introduction des redémarrages rapides, de la sauvegarde de la phase à la RSAT et de la minimisation récursive des clauses de Minisat 1.14. Depuis, les performances sont un peu moins bonnes, principalement pour des raisons de conception : la bibliothèque a été découpée en composants, la procédure de minimisation de

clauses a été généralisée. On notera que jusqu'à la version 1.7, le prouveur par défaut de Sat4j n'apprend que des clauses dont la taille est inférieure à 10% du nombre de variables (*MiniLearning*), alors que le comportement par défaut depuis la version 2.0 est d'apprendre toutes les clauses, comme les autres prouveurs SAT.

Voici une description des différents changements intervenus pour chaque version publique de Sat4j :

1.0 MiniLearning : implémentation de Minisat en Java avec apprentissage de clauses dont la taille est inférieure à 10 % du nombre de variables.

Compet 2005 MiniLearning2 : utilisation d'une structure de données spécifique pour les contraintes binaires proposée par Lawrence Ryan [151].

1.5 Version de Sat4j basée sur Java 5.

1.6 Simplification basique des clauses apprises [167]. Effacement des clauses apprises quand la mémoire disponible se fait rare.

1.7 simplification récursive des clauses apprises [167]. Redémarrages rapides de Picosat [31] et sauvegarde des phases de RSAT [145]. Retour d'un contributeur, Dieter von Holten, sur le code Java pour améliorer les performances (*speedup* d'environ 10%).

2.0 Restructuration du code pour une décomposition en composants. Généralisation de la procédure de minimisation. Apprentissage de toutes les clauses. Bytecode compatible Java 1.4.

2.1 Pas de changement significatif relatif au prouveur SAT.

2.2 Gestion de l'effacement des clauses à la Glucose et ordonnancement des littéraux par niveau de décision décroissant comme dans Picosat. Tous les paramètres fournis par Armin Biere.

Concernant la catégorie Fabriquées, le meilleur prouveur est l'implémentation initiale suivie de près par la toute dernière version : il s'agit des implémentations les plus rapides car pour la première ses fonctionnalités sont minimales et pour la seconde l'effacement agressif de clauses apprises à la Glucose [19] permet de maintenir la vélocité du prouveur.

Concernant la catégorie Aléatoires, les prouveurs Comp 05 et 1.5 sont les meilleurs, car ils n'utilisent pas de fonctionnalités spécifiques aux instances Applications, comme la mémoire des phases ou la minimisation des clauses apprises. On peut noter que les courbes Applications et Aléatoires présentent des variations opposées, alors que les courbes Aléatoires et Fabriquées sont plutôt semblables.

Sat4j n'est pas un prouveur SAT efficace en terme de performances pures, il s'agit en fait d'un des pire prouveur SAT de type CDCL évalués lors des dernières compétitions SAT ou SAT Race dans la catégorie Applications. Les résultats présentés sur le tableau 4.2 montrent que Sat4j s'est classé 29ème sur 50 lors de la première phase de la compétition SAT 2009 [118] dans la catégorie Applications, et plus précisément 29ème sur 31 prouveurs spécifiquement conçus pour résoudre ce type de problèmes. Il faut avouer que la concurrence est rude : les gagnants de la compétition SAT 2007 se retrouvent en 17ème et 19ème place. On note aussi que le CRIL a su développer des prouveurs efficaces dans cette catégorie : Glucose [19], Lysat [17] ou ManySAT [91] se retrouvent dans les 10 premiers prouveurs, et ils ont obtenu respectivement une médaille d'or et une médaille d'argent, deux médailles de bronze, et un prix spécial du jury à l'issue de la compétition.

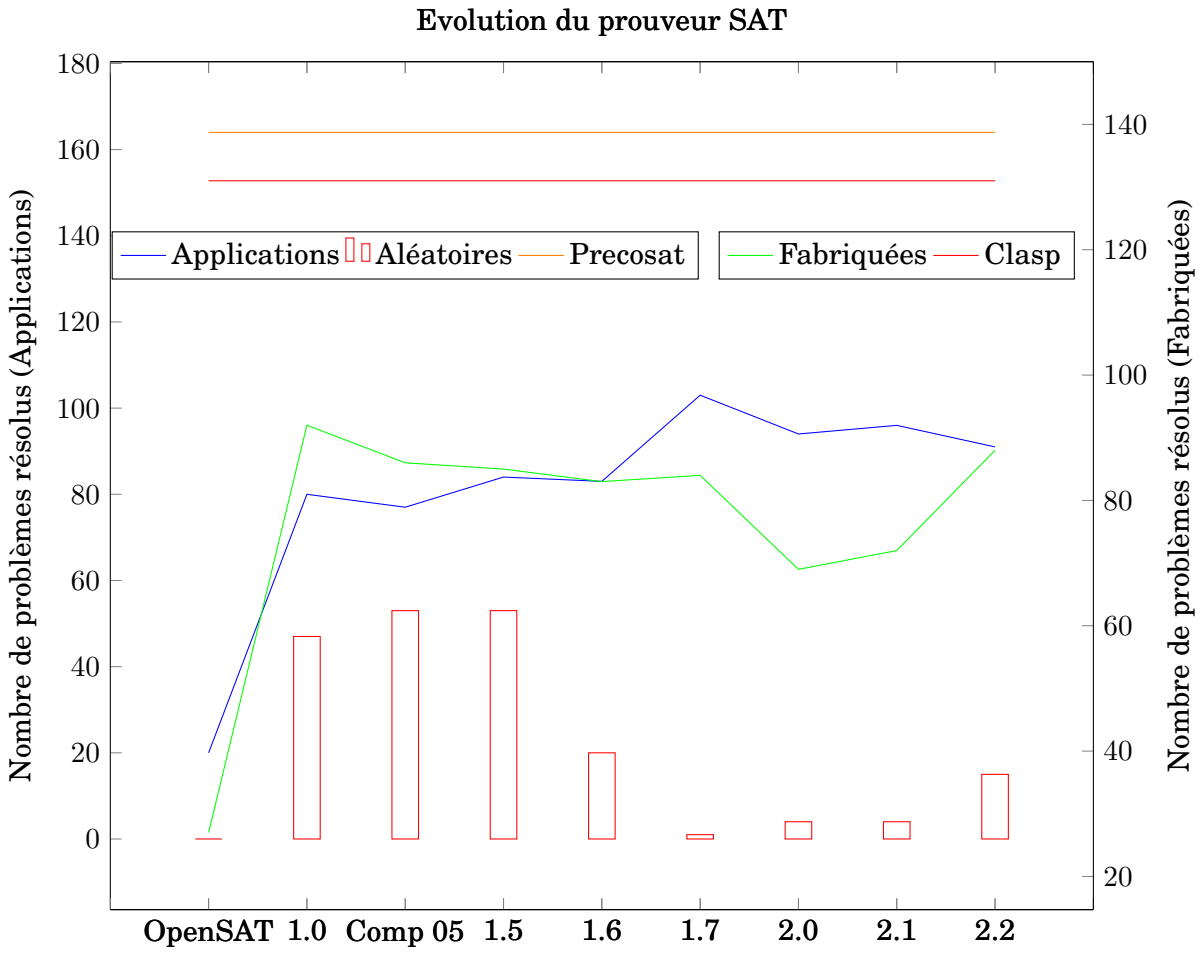


FIGURE 4.4 – Evolution des performances des différentes versions de Sat4j dans les conditions de SAT 2009 phase 1 : nombre de benchmarks résolus avec un temps limite de 20mn dans chaque catégorie parmi respectivement 292, 281 et 570 benchmarks.

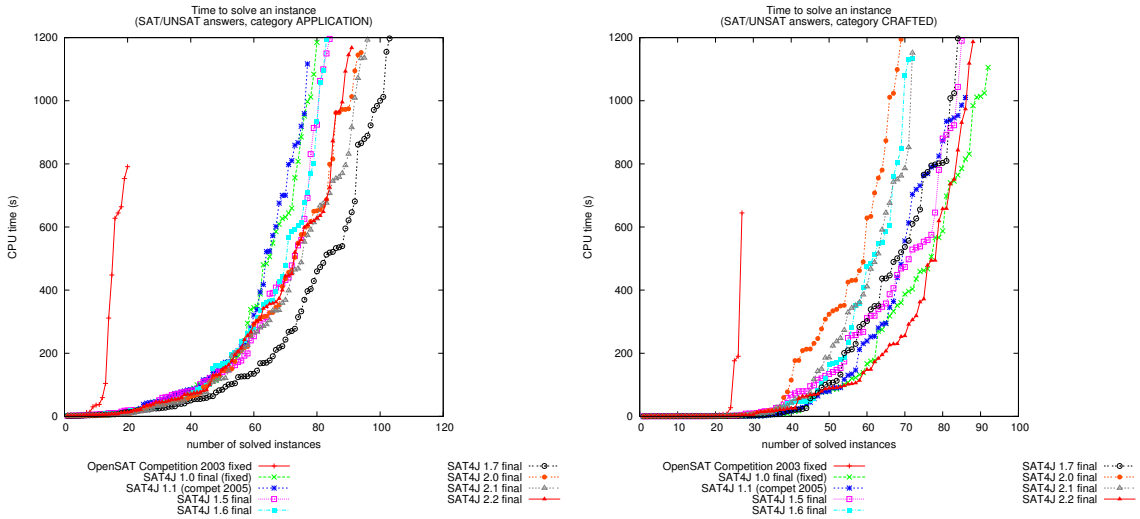


FIGURE 4.5 – Performances des différentes versions de Sat4j lors de la première phase de SAT09 catégories Applications et Fabriquées

Il y a plusieurs raisons qui expliquent les faibles performances de Sat4j pour la résolution d'instances SAT :

Java Malgré les progrès réalisés sur les machines virtuelles Java, le code de Sat4j ne peut pas s'exécuter aussi rapidement que du code natif. Il est difficile de donner des chiffres précis concernant le temps perdu par la JVM, mais un facteur 2 ou 3 est généralement accepté. Néanmoins, dans le cadre d'un prouveur SAT, le même code est tout le temps utilisé donc les techniques de compilations intelligentes à la volée (*hotspot compiler*) en code natif doivent être efficaces. En utilisant un temps limite de 2000s au lieu de 1200s, la version 2.2 résout 106 instances et la version 1.7 127 instances. En utilisant un temps limite supérieur (5000s), la version 2.2 résout 130 instances et la version 1.7 143 instances. Sat4j 1.7 se situerait donc au niveau du gagnant de 2007, Picosat 535, qui n'utilise pas non plus de pré-processeur, si l'on considère que le langage seul ralentit le prouveur d'un facteur légèrement inférieur à 2. Il faut noter aussi que les limitations du langage rendent difficiles, sinon impossibles, la réalisation de certaines structures de données (comme les littéraux bloqués par exemple).

Programmation Objet La bibliothèque utilise largement le polymorphisme, ce qui implique aussi des temps d'exécution plus élevés. La version initiale de Minisat [74], utilisant le polymorphisme pour les contraintes, a été remplacée par une version gérant uniquement des clauses en 2005 afin d'optimiser Minisat pour obtenir de très bons résultats à la compétition SAT 2005 (3 médailles d'argent obtenues). Une partie des différences en termes de performance entre la version 1.7 et la version 2.0 sont liées à un redécoupage du code pour séparer les différents modules de Sat4j, afin d'en simplifier la maintenance et de rendre la bibliothèque plus robuste aux changements (donc plus de polymorphisme). Une autre explication pour la perte de performance est l'utilisation de la bibliothèque Java 1.4 (contre 1.6 auparavant) pour respecter les contraintes d'Eclipse.

Pré-processeur La plupart des prouveurs SAT intègrent un pré-processeur depuis les excellents résultats obtenus par le couple SatELite+Minisat [75] lors de la compétition SAT 2005 (4 médailles d'or), dont l'intégration donnera Minisat 2 l'année suivante. Nous n'avons pas implémenté notre propre pré-processeur en Java car l'utilisation du pré-processeur est délicate dans le cadre de la résolution de problèmes d'optimisation. En utilisant SatELite devant Sat4j 1.7, le prouveur résultant peut résoudre153 instances avec un temps limite de 5000s, dont 112 instances au bout de 1200s.

Le choix du langage Java est un choix personnel, assumé, à contre-courant de celui de la communauté. La plupart des auteurs de prouveurs SAT cherchent la performance, donc ils développent avec des langages proches de la machine, en C ou C++. L'idée de Sat4j était de fournir aux développeurs Java un prouveur entièrement en Java « efficace ». Sat4j n'est sans doute pas aussi performant que je l'aurais souhaité, mais son adoption dans la communauté Java est certainement plus importante que je ne l'ai jamais espéré (inclusion dans Eclipse, Alloy/Kodkod, AHEAD, FAMA, etc). La majorité des utilisateurs avancés de Sat4j sont des étudiants de master pour leurs travaux pratiques ou des chercheurs de la communauté génie logiciel. Sat4j est aussi utilisé comme prouveur SAT pour Windows : la plupart des prouveurs SAT étant disponibles sous forme binaire pour Linux, Sat4j est un moyen simple pour obtenir un prouveur SAT sous Windows (voir SATlotyper). Sa diffusion sous la licence libre GNU LGPL a permis son intégration à la suite logicielle de la société Genostar, dans l'outil GNA.sim [65].

Rang	Prouveur	# Résolu	# Sat	# Unsat	Temps CPU
	Virtual Best Solver (VBS)	196	79	117	33863.84
1	precosat 236	164	65	99	37379.67
2	MiniSat 2.1 (Sat-race'08 Edition)	155	65	90	27011.56
3	LySAT i	153	57	96	35271.11
4	glucose 1.0	152	54	98	34784.84
5	MiniSAT 09z	152	59	93	37872.87
6	kw	150	58	92	35080.23
7	ManySAT 1.1 aimd 1	149	54	95	34834.19
8	ManySAT 1.1 aimd 0	149	54	95	38639.59
9	MXC	147	62	85	27968.90
10	ManySAT 1.1 aimd 2	145	51	94	34242.50
	...				
17	SAT07 Rsat	133	56	77	28975.23
18	SApperloT base	129	55	74	31762.78
19	SAT07 picosat 535	126	59	67	33871.13
	...				
27	SATzilla2009_C	106	45	61	25974.72
28	VARSAT-crafted	99	44	55	23553.01
29	Sat4j CORE 2.1 RC1	95	46	49	25380.84
30	satake	92	40	52	18309.62
31	CSat	91	40	51	20461.14
32	SATzilla2009_R	59	36	23	6260.03
	...				
50	gnovelty+2	4	4		91.85

TABLE 4.2 – Positionnement de Sat4j et d'autres prouveurs du CRIL lors de la première phase de la compétition SAT 2009, catégorie Applications (292 benchmarks à résoudre).

Rang	Prouveur	# Résolu	# Sat	# Unsat	Temps CPU
	Virtual Best Solver (VBS)	194	124	70	19204.67
1	clasp 1.2.0-SAT09-32	131	78	53	22257.76
2	SATzilla2009_I	128	86	42	21700.11
3	SATzilla2009_C	125	73	52	16701.85
4	MXC	124	80	44	22256.57
5	precosat 236	122	81	41	22844.50
6	IUT_BMB_SAT 1.0	120	76	44	22395.97
7	SAT07 minisat	119	76	43	22930.58
8	SAT07 SATzilla CRAFTED	114	82	32	18066.80
9	MiniSat 2.1 (Sat-race'08 Edition)	114	74	40	18107.02
10	glucose 1.0	114	75	39	20823.96
	...				
30	VARSAT-random	84	47	37	14023.19
31	satake	75	55	20	16261.12
32	iPAWS	71	71		7352.89
33	Sat4j CORE 2.1 RC1	71	50	21	15136.95
34	adaptg2wsat2009	70	68	2	9425.51
35	adaptg2wsat2009++	66	64	2	5796.69
36	Hybrid2	66	66		10425.56
	...				
51	slstc 1.0	33	33		4228.67

TABLE 4.3 – Positionnement de Sat4j et d'autres prouveurs du CRIL lors de la première phase de la compétition SAT 2009, catégorie fabriquées (281 benchmarks à résoudre).

4.3.2 Pseudo-booléen

La figure 4.6 montre l'évolution des deux prouveurs pseudo-booléens disponibles dans Sat4j. Nous ne présentons pas les résultats de notre premier prouveur de 2005 car le format de la compétition PB a changé en 2006. Le prouveur PB basé sur la résolution n'est disponible que depuis Sat4j 1.7. On note que de légères améliorations sont intervenues au fil des ans sur le prouveur basé sur les plans-coupe, mais rien d'exceptionnel. Ces améliorations sont principalement dues à une représentation spécifique des contraintes pseudo-booléennes lorsqu'elles correspondent à des clauses ou à des contraintes de cardinalité, et à une généralisation du prouveur SAT (qui permet par exemple la simplification de clauses dérivées de divers types de contraintes lors de l'analyse de conflits). On note que la version du prouveur PB CP la moins efficace est la version 1.7 (qui contient pourtant le meilleur prouveur SAT). La dernière version de Sat4j, incluant une meilleure intégration des redémarrages lors de l'optimisation, améliore sensiblement les performances du prouveur PB basé sur la résolution.

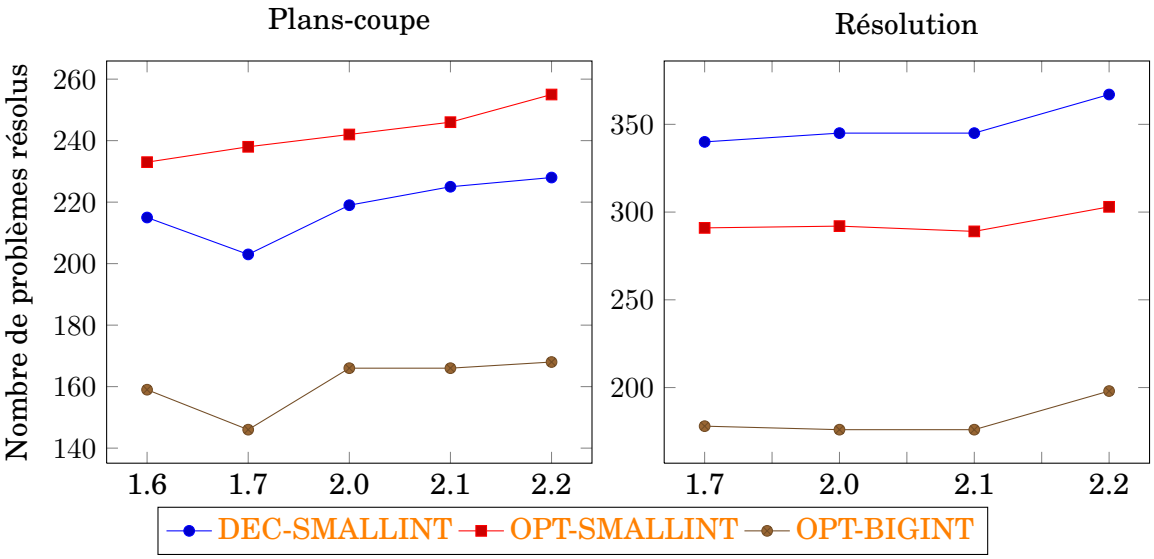


FIGURE 4.6 – Performance des différentes versions de Sat4j dans les conditions de PB 2010 : nombre de benchmarks résolus dans chaque catégorie parmi respectivement 452, 699 et 532 benchmarks.

Coopération de prouveurs sur architecture multi-coeur

Les résultats des deux types de prouveur sont souvent très différents : certains problèmes de la compétition peuvent facilement être prouvés incohérents par un prouveur plan-coupe alors que ces problèmes sont très difficiles pour le prouveur à base de résolution. Cela peut s'expliquer par le fait que le système de preuve du premier permet de trouver des preuves d'incohérence courtes, qui n'existent pas dans le cadre de la résolution. C'est aussi le cas lorsque le prouveur doit prouver l'optimalité d'une solution : bien souvent, le prouveur basé sur la résolution est capable de trouver une solution optimale, mais sans pouvoir prouver qu'elle est bien optimale. Le prouveur basé sur les plans-coupe est lui souvent si lent qu'il n'arrive pas à atteindre la solution optimale.

Nous avons donc cherché, Anne Parrain et moi-même, un moyen de faire collaborer les deux types de prouveurs, afin de profiter des avantages des deux : chercher une preuve d'incohérence rapidement via les plans-coupe, utiliser la résolution jusqu'à trouver une solution optimale, puis construire une preuve de l'optimalité avec les plans-coupe. S'il est facile d'utiliser un prouveur basé sur les plans-coupe au début de la recherche pour prouver rapidement l'incohérence, il est plus délicat de détecter le moment où le prouveur doit prouver l'optimalité d'une solution. Olivier Roussel nous a proposé une solution simple à mettre en oeuvre dans Sat4j : utiliser les deux prouveurs en parallèle dans notre procédure d'optimisation. L'idée est d'utiliser en parallèle ces prouveurs lors du test de cohérence (estSatisfiableParallele dans l'algorithme 2). Le premier prouveur qui répond stoppe l'autre. Chaque prouveur reçoit ensuite la nouvelle contrainte sur la valeur de la fonction objectif. Il y a donc une coopération des prouveurs, puisque chaque prouveur reçoit, dès sa découverte, un nouveau majorant (inférieur au précédent) de la valeur que prendra la fonction objectif à l'optimum.

entrée : Un ensemble de clauses, de contraintes de cardinalités et de contraintes pseudo-booléennes ensDeContraintes, et une fonction d'objectif à minimiser objFct
sortie : un modèle de ensDeContraintes, ou UNSAT si le problème est insatisfiable.

réponse ← `estSatisfiableParallèle` (ensDeContraintes);
if réponse *est* UNSAT **then**
  return UNSAT
end
repeat
  modèle ← réponse ;
  réponse ← `estSatisfiableParallèle` (ensDeContraintes ∪ {objFct < objFct (modèle)});
until *(*réponse *est* UNSAT*)*;
return modèle ;

Algorithme 2: Optimisation par renforcement (recherche linéaire) avec prouveurs pseudo-booléens en parallèle.

Afin d'illustrer le fonctionnement de ce prouveur, la figure 4.7 présente les traces des deux prouveurs séquentiels soumis à l'évaluation PB 2010, dans la catégorie optimisation, petits entiers et contraintes linéaires, sur l'instance `logic-synthesis/normalized-jac3`. On note qu'aucun des deux prouveurs n'est capable de calculer une solution optimale. Le prouveur à base de plans-coupe trouve une meilleure solution que le prouveur à base de résolution, en très peu de temps (25 secondes), mais n'est pas capable de prouver que cette solution est optimale.

La figure 4.8 présente la trace du prouveur utilisant les deux prouveurs en parallèle pour chaque test de cohérence. On note que la première solution est trouvée par le prouveur basé sur les plans-coupe, ensuite plusieurs solutions sont déterminées par le prouveur basé sur la résolution, puis le prouveur à base de plans-coupe trouve les dernières solutions et prouve l'optimalité. On peut noter qu'il y a bien coopération des prouveurs, comme escompté. Le prouveur à base de plans-coupe met plus de temps à trouver la solution optimale (118 secondes contre 25 secondes) mais il peut prouver l'optimalité de cette solution en 46 secondes, alors qu'il lui était impossible de le faire en y consacrant 1775 secondes en séquentiel. Ce résultat peut sembler surprenant. Il est lié à la « chance » du prouveur basé sur les plans-coupe de dériver dans cet exemple les « bonnes » contraintes

```
% Plans Coupe
1.17/0.78 c #vars     1731
1.17/0.78 c #constraints  1254
1.76/1.03 c SATISFIABLE
1.76/1.03 c OPTIMIZING...
1.76/1.03 o 26
3.40/1.91 o 25
5.93/3.41 o 24
6.97/4.33 o 23
7.49/4.88 o 22
8.44/5.72 o 21
9.00/6.27 o 20
9.62/6.87 o 19
10.44/7.61 o 18
11.54/8.79 o 17
13.03/10.13 o 16
25.34/22.07 o 15
1800.11/1773.42 s SATISFIABLE
```

```
% Résolution
1.17/0.75 c #vars     1731
1.17/0.75 c #constraints  1254
1.57/0.91 c SATISFIABLE
1.57/0.91 c OPTIMIZING...
1.57/0.91 o 26
2.55/1.42 o 23
2.96/1.60 o 22
3.35/1.80 o 21
16.34/14.32 o 20
55.04/52.91 o 19
766.33/763.00 o 18
1800.04/1795.76 s SATISFIABLE
```

FIGURE 4.7 – Trace des prouveurs pseudo-booléens lancés séparément. Le temps CPU et le temps montre en main précèdent l'affichage des prouveurs.

pour prouver l'optimalité. Faire coopérer ces prouveurs, c'est-à-dire fournir à chaque prouveur des solutions trouvées par l'autre prouveur, change pour chacun d'entre eux sa façon d'explorer l'espace de recherche, et l'on peut avoir, comme dans ce cas, de bonnes surprises. En analysant les résultats du prouveur lors de la compétition PB dans la catégorie OPT-SMALLINT-LIN, on trouve de nombreux cas pour lesquels le prouveur hybride est plus lent que chacun des prouveurs : il est dans ce cas malchanceux. Les diagrammes de dispersion de la figure 4.9 montrent bien, sur la droite, que prendre en compte le temps CPU et pas le temps montre en main rend le prouveur hybride deux fois plus lent que le meilleur prouveur. Les résultats de la compétitions 2010 de la figure 4.10 montrent cependant que cette approche est intéressante en pratique.

Comparaison avec l'état de l'art en 2010

Par rapport à l'état de l'art, Sat4j est comparable aux autres prouveurs PB. La table 4.4 présente les résultats de la catégorie satisfaction de contraintes pseudo-booléennes. Notre prouveur hybride se classe deuxième, derrière un prouveur pseudo-booléen de type portfolio (incluant les prouveurs pseudo-booléens de Sat4j de l'année dernière). Compte tenu des mauvais résultats du prouveur SAT utilisé comme base, ce résultat est très satisfaisant.

Dans la catégorie optimisation, c'est un prouveur basé sur l'outil commercial de programmation linéaire CPLEX qui se révèle le meilleur. Notre prouveur hybride n'arrive que 7ème, derrière 4 prouveurs basés sur la programmation linéaire, et deux versions de Bsolo [130], utilisant une approche de type *Branch&Bound*.

Concernant les instances contenant de grands entiers, Sat4j est l'un des seuls prouveurs correct capable de traiter des entiers en précision arbitraire. Il était le seul concurrent dans cette catégorie l'année dernière, seulement deux prouveurs se sont révélés corrects dans cette catégorie cette année [128]. Ce choix nous permet de ne pas nous limiter lors de la

```
% Résolution et Plans Coupe en parallèle
1.35/0.84 c #vars     1731
1.35/0.84 c #constraints  1254
1.99/1.85 c SATISFIABLE
1.99/1.85 c OPTIMIZING...
1.99/1.85 o 26 (Plans Coupe)
2.61/2.89 o 25 (Résolution)
3.91/3.92 o 24 (Résolution)
4.12/5.00 o 23 (Résolution)
5.92/6.01 o 22 (Résolution)
7.72/7.04 o 21 (Résolution)
9.63/8.07 o 20 (Plans Coupe)
13.04/10.09 o 19 (Plans Coupe)
15.66/12.10 o 18 (Plans Coupe)
20.27/15.14 o 17 (Plans Coupe)
70.03/41.35 o 16 (Plans Coupe)
218.63/118.14 o 15 (Plans Coupe)
305.11/164.68 s OPTIMUM FOUND
```

FIGURE 4.8 – Trace des prouveurs pseudo-booléens lancés en parallèle. Pour chaque solution trouvée, le nom du prouveur ayant répondu le premier est mentionné.

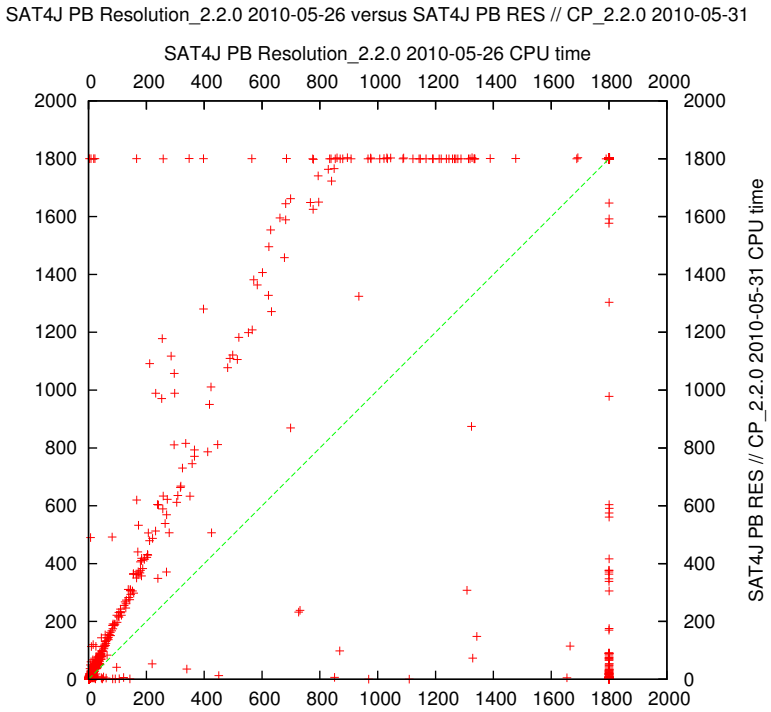


FIGURE 4.9 – Comparaison des performances en temps CPU cumulé entre la version séquentielle du prouveur PB basé sur la résolution et la version hybride en parallèle.

Rang	Prouveur	# Résolu	# Sat	# Unsat	Temps CPU
	Virtual Best Solver (VBS)	432	180	252	18241.79
1	borg-pb	415	179	236	38544.74
2	SAT4J PB RES // CP 2.2.0	382	173	209	31891.63
3	bsolo 3.2 Card	380	172	208	50981.11
4	wbo 1.4a	378	171	207	27704.32
5	PB/CT 0.1 fixed	369	164	205	35145.84
6	SAT4J PB Résolution 2.2.0	367	174	193	46664.10
7	bsolo 3.2 Cl	355	170	185	61733.63
8	SCIP 1.2.1.3 with SoPlex 1.4.2	351	139	212	62141.26
9	SCIP 1.2.1.2 with SoPlex 1.4.2	351	141	210	65260.84
10	SCIPclp SCIP 1.2.1.2	344	144	200	54632.83
11	pb_cplex	337	155	182	39196.42
12	SCIP 1.2.1.2 without any LP solver	288	154	134	53015.70
13	SAT4J PB Plans Coupe 2.2.0	228	106	122	25513.09
14	PB-wave alpha 2	66	66		7400.95

TABLE 4.4 – Positionnement de Sat4j lors de la de la compétition PB 2010, catégorie décision, petits entiers, contraintes linéaires, DEC-SMALLINT-LIN (452 benchmarks à résoudre).

Rang	Prouveur	# Résolu	# Opt.	# Unsat	Temps CPU
	Virtual Best Solver (VBS)	472	439	33	19877.90
1	pb_cplex 2010-06-29	417	384	33	37700.68
2	SCIP 1.2.1.3 with SoPlex 1.4.2	354	321	33	45520.92
3	bsolo 3.2 Card	333	300	33	47182.34
4	bsolo 3.2 Cl	328	295	33	42827.61
5	SCIP 1.2.1.2 with Clp 1.11.1	319	286	33	25689.98
6	SCIP 1.2.1.2 with SoPlex 1.4.2	317	284	33	24202.46
7	SAT4J PB RES // CP 2.2.0	315	282	33	29160.61
8	SAT4J PB Résolution 2.2.0	303	270	33	14709.64
9	PB/CT 0.1 fixed	283	251	32	25446.64
10	SAT4J PB Plans Coupe 2.2.0	255	226	29	29801.79
11	SCIP 1.2.1.2 without any LP solver	187	158	29	26322.18

TABLE 4.5 – Positionnement de Sat4j lors de la de la compétition PB 2010, catégorie optimisation, petits entiers, contraintes linéaires, OPT-SMALLINT-LIN (532 benchmarks à résoudre).

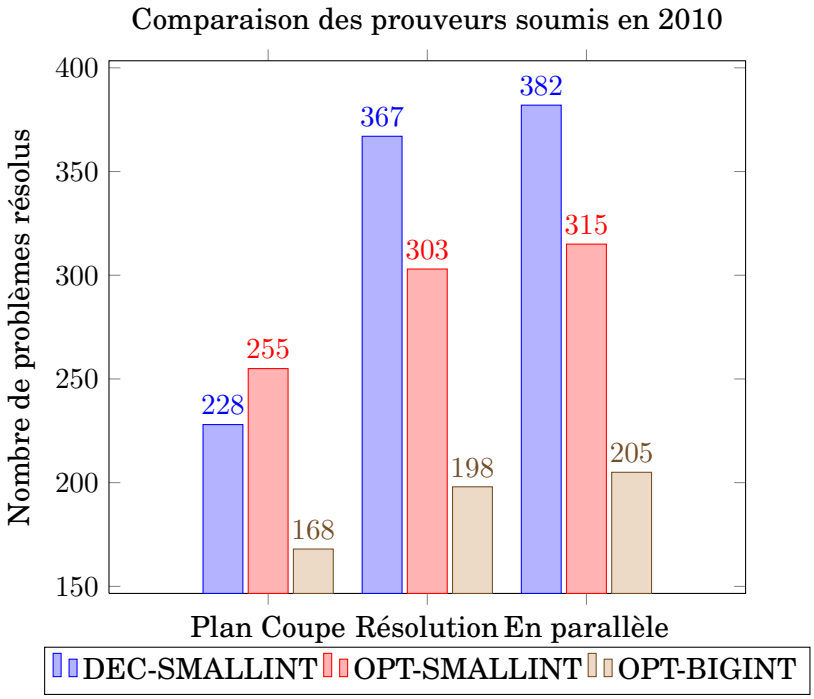


FIGURE 4.10 – Performance des différents prouveurs pseudo-booléens de Sat4j 2.2.1 dans les conditions de PB 2010 : nombre de benchmarks résolus dans chaque catégorie parmi respectivement 452, 699 et 532 benchmarks.

modélisation des problèmes.

4.3.3 MAX-SAT

Le prouveur MaxSat intégré dans Sat4j a été conçu au départ pour se faire une idée des performances d'une simple traduction en problème d'optimisation pseudo-booléen lors de la première évaluation de prouveurs MaxSat en 2006. A cette époque, l'évaluation se faisait quasi exclusivement sur des problèmes aléatoires ou fabriqués de type Max 2-SAT ou Max 3-SAT. Sat4j était de loin le plus mauvais compétiteur. En 2008, nous avons amélioré les performances de notre prouveur MaxSat en utilisant un prouveur pseudo-booléen basé sur la résolution et en détectant les problèmes de couverture binaire. Cette année correspond aussi à l'apparition d'instances MaxSat provenant de problèmes réels (bio-informatique [89], conception de circuits [152], etc.). Sat4j a dominé les autres prouveurs sur les instances hipp d'inférence d'haplotype [89], qui correspondent à des problèmes de couverture binaire. D'une manière générale, les résultats pointent une différence évidente entre les prouveurs conçus spécifiquement pour les problèmes MaxSat (*maxsatz) et les prouveurs basés sur une réutilisation de prouveurs SAT (comme Sat4j, MSUnCore [133], PM2 [10, 11], etc) : les premiers obtiennent de bons résultats sur les instances MaxSat de petite taille, aléatoires ou fabriquées, mais ne passent pas à l'échelle. Les seconds fonctionnent beaucoup mieux sur les instances issues d'applications, souvent de très grande taille (plusieurs millions de variables et de clause, comme en SAT). Depuis 2007, Sat4j MaxSat est utilisé par le laboratoire 4C pour résoudre des problèmes de souscription téléphonique de British Telecom [120]. Malgré les mauvais résultats obtenus dans les évaluations MaxSat à cette époque, Sat4j obtenait de bons résultats sur ces problèmes spécifiques par rapport à des outils comme Choco ou CPLEX. Des échanges avec ces utilisateurs de Sat4j MaxSat

nous ont donné l'occasion d'améliorer sensiblement les performances de notre prouveur fin 2008 et début 2009 [51, 121]. La figure 4.11 présente l'évolution des performances de Sat4j MaxSat sur les instances de l'évaluation 2010 [14]. On peut noter que contrairement au cas des prouveurs SAT et pseudo-booléens, les progrès sont significatifs ces dernières années.

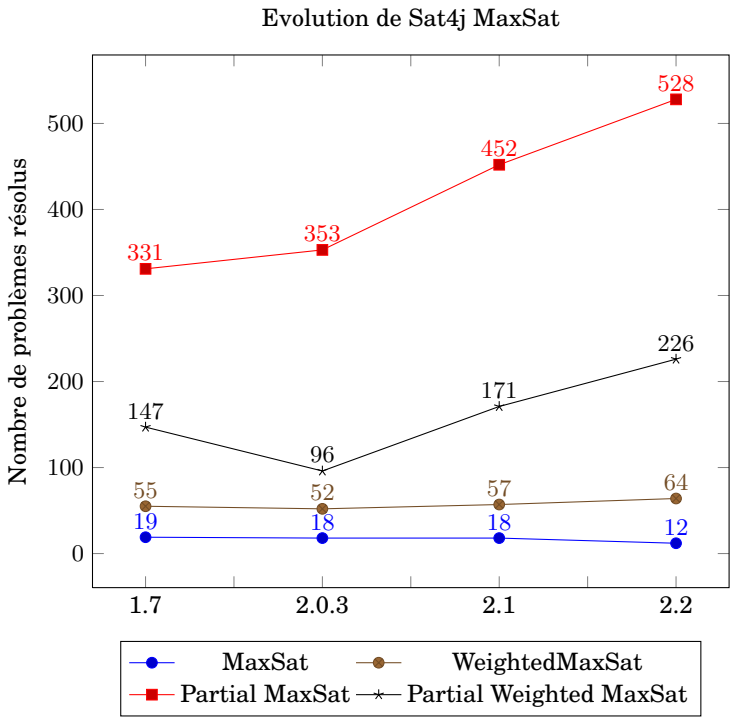


FIGURE 4.11 – Performance des différentes versions de Sat4j dans les conditions de MAX-SAT 2010 : nombre de benchmarks résolus dans chaque catégorie parmi respectivement 544, 625, 349 et 660 benchmarks.

Nous avons choisi quelques résultats représentatifs de l'évaluation MaxSat 2009 pour illustrer le comportement de Sat4j par rapport aux autres prouveurs MaxSat sur les instances issues d'applications. Le tableau 4.6 (issu de [13]) présente les résultats de la catégorie MaxSat sur les instances « industrielles ». La première série de problèmes (CircuitDebuggingProblems) contiennent énormément de clauses et de variables (de 200K à 1,3M de variables, de 168K à 2M de clauses), et ont pour spécificité d'avoir une solution optimale de valeur 1, c'est à dire qu'il est possible de satisfaire toutes les contraintes sauf une. Cela avantage clairement les prouveurs basés sur la détection de noyaux incohérents comme MSUncore [133], WPM1 ou PM2 [10, 11], puisque cette approche doit effectuer $n+1$ tests de cohérence avec n la valeur de la solution optimale. Sat4j ne résout que deux de ces instances et il échoue sur une instance (un bug dans Sat4j lié au fait que ces instances déclarent beaucoup plus de variables qu'ils n'en utilisent en réalité) et est incapable de charger les six autres par manque de mémoire (la mémoire est limitée à 512 Mio durant l'évaluation MaxSat). En faisant tourner Sat4j sur le cluster du CRIL, avec 1,8Gio de mémoire vive, la version soumise à l'évaluation 2010 peut en résoudre cinq et une version corrigeant le bug sur ces instances en résout six sur les neuf (MsUncore les résout toutes). Une instance requiert plus de mémoire (elle contient 800K variables et 2,2M de clauses, nous obtenons donc un problème OPB de plus de 3M de variables !) mais elle est résolue en 12s avec plus de mémoire (2,8Gio). Les deux autres instances nécessitent plus de temps

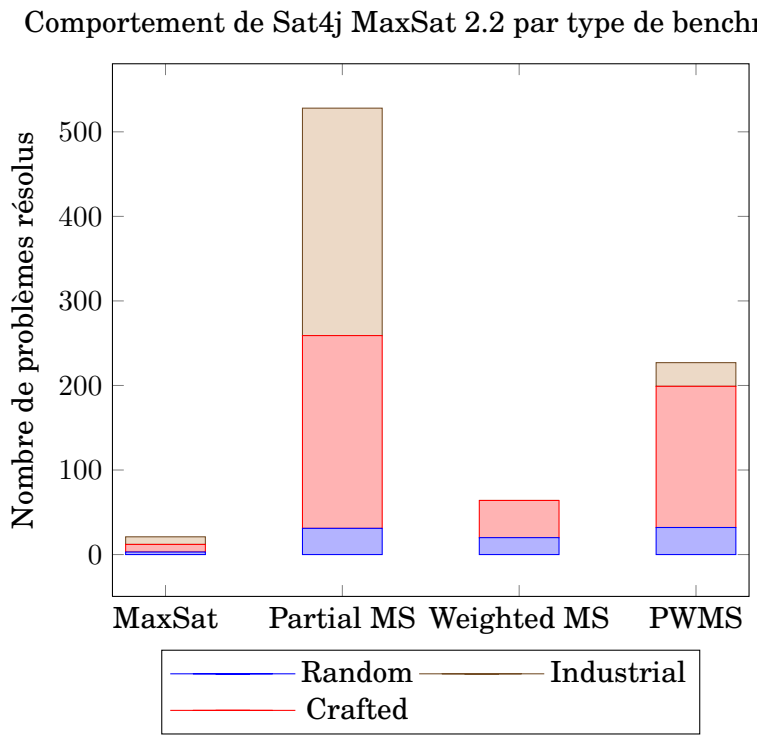


FIGURE 4.12 – Détail du type de benchmarks résolus lors de la compétition MAXSAT 2010.

(3K secondes pour la première, échec après plus de 128K secondes pour la seconde). Pour les instances SeanSafarpour, on peut faire les mêmes remarques : manque de mémoire pour Sat4j, valeur optimale proche de 1 dans de nombreux cas. On retrouve des résultats similaires lors de l'évaluation 2010 [14], avec des résultats qui semblent pires pour Sat4j car les instances les plus « faciles » de SeanSafarpour ont été enlevées.

Instance_set	#I	MSUnCore	wbo	wpm1	pm2
CircuitDebuggingProblems/	9	17.50(8)	178.88(8)	19.52(5)	20.32(5)
SeanSafarpour/	112	37.97(84)	110.29(82)	48.72(73)	30.41(61)
Solved instances		92	90	78	66

Instance_set		SAT4J-Maxsat	IncMaxsatz	WMaxSatz-1.6	WMaxSatz-2.5
CircuitDebuggingProblems/		33.37(2)	0.00(0)	0.00(0)	0.00(0)
SeanSafarpour/		174.42(32)	3.59(2)	84.69(2)	6.06(2)
Solved instances	121	34	2	2	2

TABLE 4.6 – Performances de Sat4j 2.1 lors de MaxSat 2009 : Industrial Max Sat

Le tableau 4.7 représente les résultats de la catégorie *Partial MaxSat* industrielle [13]. Sat4j se retrouve deuxième en terme de nombre total de problèmes résolus. Certains problèmes sont issus de la traduction d'instances de couverture binaire de la compétition pseudo-booléenne, il n'est donc pas étonnant que Sat4j se comporte bien sur ces benchmarks, puisqu'il les « reconnait » et les traite comme tels. Il n'y a pas de problème de

mémoire ici : Sat4j trouve souvent des majorants proches de la solution optimale dans le temps imparti. Sat4j se retrouve quatrième dans cette catégorie en 2010 [14], derrière PM2 et wbo, et un nouveau venu, QmaxSat, qui utilise la même technique que Sat4j mais en transformant les contraintes de cardinalités en clauses. Si ce classement est certainement dû aux améliorations des autres prouveurs, il est aussi en partie la conséquence d'une réduction du nombre d'instances dans les familles de benchmarks (bcp-msp par exemple réduite de 148 à 64 benchmarks, et Sat4j passe de 95 à 12 benchmarks résolus !), et pas d'une baisse des performances de notre prouveur (la figure 4.11 montre bien au contraire qu'il progresse).

Instance_set	#I	pm2	SAT4J-Maxsat	wpm1	wbo	MSUnCore
CircuitTraceCompaction/	4	442.99(3)	431.75(3)	0.00(0)	0.00(0)	0.00(0)
HaplotypeAssembly/	6	34.52(5)	0.00(0)	31.70(5)	9.72(5)	31.52(5)
PROTEIN_INS/	12	56.83(2)	319.63(2)	44.11(1)	11.96(1)	534.08(1)
bcp-fir/	59	13.60(57)	7.99(10)	27.99(56)	112.01(37)	53.11(49)
bcp-hipp-yRa1/SU/	38	101.65(28)	268.15(8)	21.44(11)	109.84(10)	57.08(10)
bcp-hipp-yRa1/simp/	138	8.10(136)	19.95(132)	1.40(130)	3.56(131)	1.35(130)
bcp-msp/	148	91.31(94)	7.54(95)	12.25(42)	45.35(24)	102.03(23)
bcp-mtg/	215	1.36(215)	67.03(196)	6.03(170)	15.67(173)	6.38(166)
bcp-syn/	74	12.64(38)	98.81(20)	5.70(31)	31.76(30)	35.20(32)
pbo-mqc/nencdr/	128	233.02(92)	318.54(114)	82.62(50)	73.63(63)	35.74(48)
pbo-mqc/nlogencdr/	128	148.66(108)	176.42(125)	69.84(75)	76.27(67)	56.44(67)
pbo-routing/	15	1.18(15)	223.31(12)	0.99(15)	0.96(15)	0.63(15)
Solved instances	965	793	717	586	556	546

Instance_set	WMaxSatz-2.5	WMaxSatz-1.6	IUT_BCMB_LSWM	Clone	IUT_BCMB_WM
CircuitTraceCompaction/	0.00(0)	0.00(0)	0.00(0)	0.00(0)	0.00(0)
HaplotypeAssembly/	0.00(0)	0.00(0)	0.00(0)	0.00(0)	0.00(0)
PROTEIN_INS/	0.26(1)	0.22(1)	0.29(1)	26.99(1)	1.49(1)
bcp-fir/	6.99(7)	14.83(7)	7.34(7)	118.32(8)	10.53(7)
bcp-hipp-yRa1/SU/	0.00(0)	0.00(0)	0.00(0)	0.00(0)	0.00(0)
bcp-hipp-yRa1/simp/	218.83(44)	242.21(43)	202.00(44)	105.18(74)	187.72(40)
bcp-msp/	30.18(94)	38.61(94)	30.35(93)	77.38(91)	32.50(90)
bcp-mtg/	118.80(153)	48.11(143)	84.52(119)	45.70(104)	58.06(102)
bcp-syn/	148.71(22)	133.83(22)	139.79(22)	183.55(23)	127.35(24)
pbo-mqc/nencdr/	544.37(59)	597.77(58)	512.29(54)	0.00(0)	0.00(0)
pbo-mqc/nlogencdr/	331.39(95)	383.59(92)	353.89(95)	333.79(86)	0.00(0)
pbo-routing/	5.82(5)	7.27(5)	5.63(2)	10.97(5)	13.64(2)
Solved instances	965	480	465	437	392	266

TABLE 4.7 – Performances de Sat4j 2.1 lors de MaxSat 2009 : Industrial Partial Max Sat

4.4 Basés sur Sat4j : Eclipse p2 et p2cudf

4.4.1 p2

Depuis juin 2008, la version 3.4 d'Eclipse utilise Sat4j pour la gestion de ses dépendances [114]. Le nouveau système de gestion de dépendance d'Eclipse, p2, a été principalement développé par Pascal Rapicault, et par l'ensemble des contributeurs du projet p2. Lors de la version 3.4 d'Eclipse, p2 produisait un fichier opb à destination de Sat4j. Sat4j n'était pas vraiment intégré à Eclipse. Dès l'année suivante, il a été décidé d'intégrer plus finement Sat4j et p2, afin de pouvoir expliquer à l'utilisateur pourquoi une requête ne pouvait pas être satisfaite. Depuis Janvier 2009, je suis devenu « *Eclipse p2 committer* » afin d'assurer une intégration du mécanisme d'explication développé dans Sat4j pour Eclipse financée par Genuitec dans le cadre d'une étude de faisabilité CNRS. Mes contributions se limitent typiquement à une seule classe Java, `Projector`, qui se charge de traduire les dépendances

et la requête en problème d'optimisation pseudo-booléen.

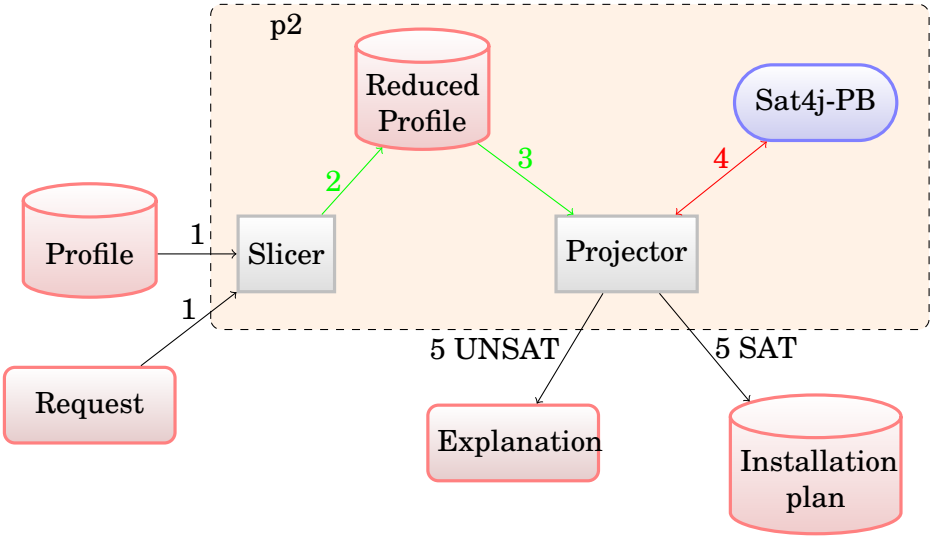


FIGURE 4.13 – Transformation d'une requête p2 en problème d'optimisation pseudo-booléen.

Le principe de fonctionnement de p2 vis à vis de Sat4j est représenté sur la figure 4.13. Un profil contient une description détaillée des dépendances entre les différents plugins disponibles. La première étape consiste à réduire la taille du profil en ne tenant compte que des dépendances (par fermeture transitive) de la requête. Ensuite ce profil et la requête sont transformés en problème d'optimisation pseudo-booléen. Sat4j cherche alors une solution dans un temps limité. Tous les détails concernant l'encodage et le contexte de cette application est disponible dans [114] (article joint chapitre 7).

4.4.2 p2cudf

J'ai suivi depuis mai 2008 le projet européen Mancoosi, et plus précisément la partie gestion des dépendances dans le monde Linux. Pascal Rapicault et moi-même avons saisi l'opportunité de pouvoir participer à la première compétition interne de prouveurs de Mancoosi (MiSC) pour construire un système basé sur p2 capable de résoudre des problèmes de gestion de dépendances dans le format défini par ce projet : *Common Upgrade Description Format* [169].

Le principe de fonctionnement de ce prouveur est présenté dans la figure 4.14. La requête CUDF est traduite en un profil p2 et une requête p2. Cette requête est ensuite prise en charge par une version simplifiée de p2 disponible dans Eclipse 3.5 (juin 2009). La solution retournée par p2 est ensuite décodée en CUDF. Le système en lui-même n'était pas difficile à concevoir. Le travail le plus important était de comprendre les subtilités du format CUDF et les relations entre les concepts utilisés dans le monde Linux et ceux du monde Eclipse. Les détails de l'encodage utilisé dans cet outil sont présentés dans [12]. p2cudf a donné des résultats satisfaisants lors de cette première compétition, et son encodage en problème d'optimisation pseudo-booléen a été adopté par un partenaire du projet (INESC).

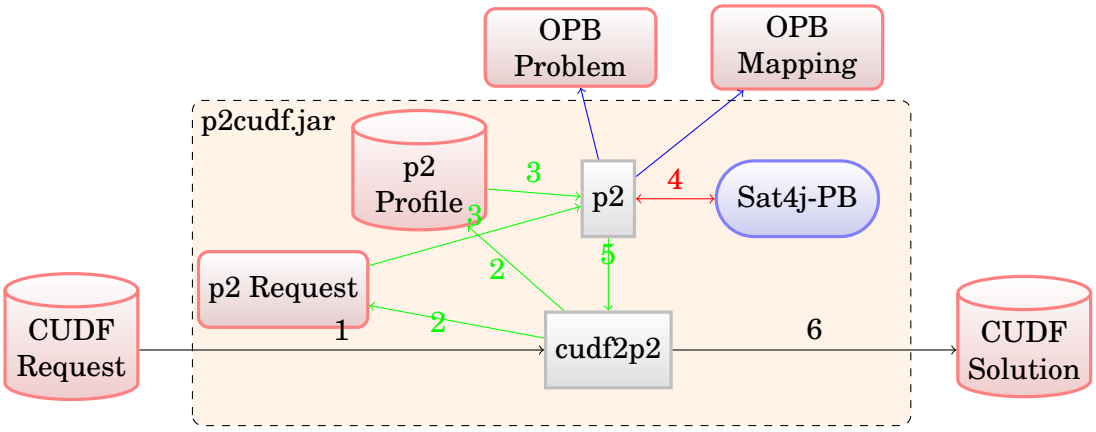


FIGURE 4.14 – Transformation d'une requête CUDF en requête p2 dans p2cudf.

p2cudf a été mis à niveau pour participer à la compétition internationale MISC organisée en juin 2010 durant FLoC. Quelques changements a priori mineurs ont été introduits dans les benchmarks (champs `keep` et `recommends`), et dans une fonction d'optimisation (prise en compte des paquetages recommandés). Ne disposant pas d'instances de test pour ces nouvelles fonctionnalités, p2cudf avait été testé sur quelques instances simples. Malheureusement, à la lecture des résultats de la compétition, on comprend vite que les problèmes CUDF n'étaient pas correctement analysés par p2cudf. Les dysfonctionnements ont été corrigés rapidement, et la gestion des critères d'optimisation a été généralisée pour permettre à l'utilisateur de fournir ses propres critères d'optimisation.

Le travail effectué sur p2cudf a été bénéfique pour p2 par plusieurs aspects :
- Le format CUDF est très pratique pour écrire des scénarios de tests. Nous envisageons d'utiliser ce format pour écrire les tests de p2 dans le futur.
- Participer aux deux compétitions, MiSC et MISC, nous a permis de valider l'approche que nous avons mise en place dans Eclipse (car les réponses obtenues sont celles escomptées) tout en mettant en évidence un problème de passage à l'échelle (sur les instances contenant 50K paquetages).
- Disposer d'une version de p2 qui n'est pas directement utilisée dans Eclipse nous permet d'expérimenter des codages, des fonctions d'optimisation, etc. que nous ne pouvons pas nous permettre de changer sur la version en production sans justification.
- p2cudf est conçu pour fonctionner avec tout prouveur pseudo-booléen, en utilisant le format OPB de l'évaluation pseudo-booléenne.

En résumé, p2cudf nous a permis de prendre du recul sur la problématique de la gestion des dépendances.

p2cudf est un logiciel libre (licence EPL) disponible à l'adresse `http://wiki.eclipse.org/Equinox/p2/CUDFResolver`.

4.5 Sat4j en 2010

Nous avons déjà indiqué que la bibliothèque Sat4j fournit depuis six ans divers prouveurs propositionnels (Sat, MaxSat, pseudo-booléen, CSP, etc). Ces prouveurs ont été éva-

lués années après années lors des diverses évaluations internationales. Nous avons toujours mis un point d'honneur à participer à ces évaluations, même si nos prouveurs ne montraient pas de bonnes performances ou si nous n'avions pas trouvé le temps d'améliorer nos prouveurs, simplement dans un souci de transparence pour les utilisateurs de la bibliothèque, mais aussi pour situer les performances réelles de nos prouveurs par rapport à l'état de l'art.

Nous avons déjà indiqué que les prouveurs Sat disponibles actuellement dans Sat4j ne sont plus au niveau de l'état de l'art. Si nous avons pu fournir jusqu'en 2006 des prouveurs en Java relativement efficaces (Sat4j a terminé devant Chaff lors de la Sat Race 2006 [159]), la concurrence dans ce domaine est rude : comme notre moteur SAT est généraliste pour servir de base à nos prouveurs pseudo-booléens, et ne dispose pas d'un pré-processeur comme la plupart des autres prouveurs, il est illusoire dans l'état actuel de nos connaissances sur la résolution pratique de SAT d'espérer rattraper ce retard. Néanmoins, Sat4j reste utile pour résoudre certains problèmes combinatoires par traduction à SAT lorsque l'on désire travailler en Java ou pour expérimenter rapidement une idée sur une plateforme de prouveurs SAT flexible et ouverte, dans l'esprit de SIMO [82] ou d'OpenSAT [18].

Les prouveurs pseudo-booléens sont certainement plus utiles à la communauté : les performances de nos prouveurs sont comparables à celles des autres prouveurs pseudo-booléens, et ils ont l'avantage de travailler en précision arbitraire, ce qui n'est pas le cas pour la plupart des autres prouveurs. Etant utilisé depuis deux ans par des millions de personnes, le prouveur basé sur la résolution a fait la preuve de sa fiabilité. Cependant, les performances brutes des prouveurs pseudo-booléens restent encore à améliorer. Les problèmes les plus difficiles de gestion de dépendances Linux de la compétition MISC sont actuellement hors de portée de nos prouveurs pseudo-booléens [12].

Le prouveur MaxSat fourni dans Sat4j obtient des résultats tout à fait corrects lors des évaluations MaxSat sur les instances issues d'applications, malgré les faibles performances du prouveur Sat sur lequel il est basé. Ces résultats montrent que le prouveur pseudo-booléen basé sur la résolution de Sat4j est particulièrement bien adapté à la résolution de couverture binaire.

Sat4j a été adopté par de nombreuses équipes de recherche. On retrouve ainsi Sat4j dans divers domaines :

Vérification formelle Alloy 4 [98] ou Forge [68, 67] (Daniel Jackson, MIT), TASM [143] (Kristina Lundqvist, MIT), FastCheck [48] (Radu Rugina, Cornell).

Ligne de produits logiciels AHEAD [22] (Batory, U. Texas) FAMA FW [26] (Benavides, Séville), FeatureIDE [107] (U. Magdeburg), S.P.L.O.T. [135] (U. Waterloo).

Bio-informatique GNA.sim [65](de Jong, INRIA/Genostar), GraMoFoNe [37] (U. Marne la vallée), Satlotyper [140] (Max Planck Institute).

Autre Opt4j [125] (U. Erlangen-Nuremberg), AProVE [81] (J. Giesl, U. Aachen), OpenOME [5] (U. Toronto).

Chapitre 5

Evaluation de prouveurs propositionnels

Une bonne partie de mon activité de recherche de ces dix dernières années a consisté à évaluer des prouveurs propositionnels, soit dans le cadre du développement de mes propres prouveurs, soit dans le cadre d'évaluations internationales [116, 111].

Nous avons déjà mentionné que certains prouveurs sont capables de résoudre des instances du problème SAT qui représentent des problèmes réels, importants pour notre société. En quelques années, l'univers de la vérification de circuits a été chamboulée par les performances des prouveurs CDCL sur les instances SAT issues de la vérification bornée de modèles (*bounded model checking*) [34]. Toutes les sociétés qui travaillent autour de la vérification formelle de circuits ou de programmes ont rapidement intégré un prouveur SAT dans leur boîte à outils (Moshe Vardi dans une conversation personnelle estime que l'adoption de la technologie SAT s'est faite en seulement deux ans, ce qui ne s'était jamais vu auparavant). Le fait que le prix CAV 2009 récompense les auteurs de GRASP et Chaff "*For fundamental contributions to the development of high-performance Boolean satisfiability solvers*" montre à quel point cette technologie est devenue importante aux yeux de cette communauté.

Les résultats des prouveurs SAT actuels sont en effet impressionnants : lors de la compétition SAT 2009, deux prouveurs, SapperloT et picosat 913, ont été capables de résoudre une instance SAT (*post-cbmc-zfcp-2.8-u2-noholes.cnf*) de 10M de variables et 30M de clauses en respectivement 51 et 116 secondes ! Les autres n'ont tout simplement pas été capables de la lire avec les contraintes mémoire de la compétition (1.8 Gio de RAM). Cependant, cela ne doit pas faire oublier qu'aucun prouveur n'a été capable de résoudre en 5000s une instance SAT contenant 121 variables, 252 clauses et 756 littéraux durant la même compétition.

Le rôle de la compétition internationale de prouveurs SAT est à la fois de constituer une vitrine technologique permettant de mettre en évidence les problèmes pour lesquels les prouveurs sont efficaces mais aussi un miroir sans complaisance qui reflète les faiblesses de ces prouveurs. C'est avant tout un bon moyen d'observer en pratique le comportement des différents types de prouveurs disponibles à l'heure actuelle sur un assez large éventail de benchmarks.

5.1 La compétition SAT

La communauté SAT a depuis longtemps la culture des évaluations communautaires. Les premières compétitions ont débuté dans les années 90. Depuis le début des années 2000, des compétitions sont organisées chaque année.

La première compétition SAT fut organisée en 1992 [44] par Hans Kleine-Büning et Michael Buro. La deuxième compétition SAT, la plus connue car elle a introduit le format de représentation des benchmarks qui est toujours utilisé aujourd'hui, a eu lieu l'année suivante dans le cadre du second challenge Dimacs [99]. Une troisième compétition a eu lieu à Beijing en 1996, organisée par James Crawford et DingXin Wang [61].

Lorsque le prouveur Chaff est apparu [137], montrant des performances impressionnantes par rapport aux autres prouveurs existants sur certaines classes de benchmarks, la question d'organiser une nouvelle compétition s'est posée. La création de la conférence internationale SAT en 2002 à Cincinatti (les rencontres précédentes étaient des workshops), semblait une bonne occasion pour organiser une compétition. Laurent Simon, Edward Hirsch et moi-même avons eu l'opportunité d'organiser cette première compétition [158, 147] (article joint chapitre 12). John Franco, organisateur de la conférence, nous a fourni un cluster de machines (des Pentium III avec 512Mo de RAM) pour faire tourner la compétition. Hans van Maaren nous a fourni des prix pour motiver la communauté à y participer. La compétition fut organisée à l'aide de l'infrastructure logicielle développée par Laurent Simon pour le système SAT-Ex [157], avant et durant la conférence. Le prouveur Chaff se révéla le meilleur prouveur de la compétition (2 catégories remportées sur 6, plus 2 deuxièmes places) mais deux autres prouveurs de type CDCL furent aussi primés (Limmat et Berkmin [85]). Ce résultat démontrait que la résolution pratique d'instances SAT évoluait rapidement. Une compétition est organisée tous les ans depuis, sous sa forme originale ou sous une forme de course (*SAT Race*) organisée par Carsten Sinz. La figure 5.1 montre l'évolution de la participation aux différentes compétitions SAT et la place de plus en plus grande occupée par les prouveurs CDCL, qui tend à pousser la communauté SAT vers une monoculture [1].

L'organisation de ces compétitions a avant tout un rôle d'animation de la communauté : elle fixe un rendez-vous annuel aux chercheurs pour tester leurs prouveurs ou proposer de nouveaux benchmarks. C'est aussi un bon moyen de promouvoir les travaux sur la résolution pratique du problèmes SAT au sein de la communauté. Enfin, la compétition SAT a attiré des utilisateurs de technologie SAT issus d'autres communautés, voire non académiques, à la conférence SAT.

Les compétitions fournissent aussi un moyen de comparer de manière indépendante des prouveurs, dans l'esprit du site SAT-Ex. Dans un processus similaire à l'évaluation des publications scientifiques, la compétition SAT a été conçue comme un moyen d'évaluer les prouveurs SAT par les pairs. Elle se base sur quelques idées simples pour cela :

Les organisateurs garantissent l'application des règles de participation à la compétition et l'exécution de tous les prouveurs dans des conditions comparables ;

L'auteur du prouveur valide les expérimentations, en surveillant l'exécution de son prouveur sur les différents benchmarks ;

1. Terme utilisé par Moshe Vardi lors d'une présentation invitée à SAT 2009.

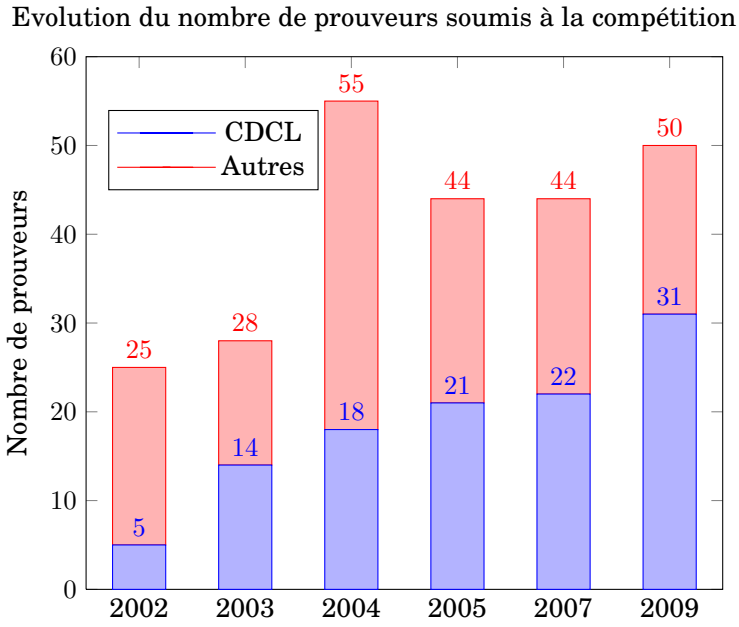


FIGURE 5.1 – Evolution du nombre de participants à la compétition SAT

Les juges analysent les résultats de manière anonyme, sélectionnent les meilleurs prouveurs dans chaque catégorie et déclarent les vainqueurs ;

La communauté peut vérifier a posteriori tous les résultats, et dispose de toutes les données nécessaires pour faire sa propre analyse de la compétition.

Les prouveurs et benchmarks sont disponibles facilement à des fins de recherche, sous forme binaire ou de préférence en source.

La première règle semble triviale mais nous verrons qu'elle devient plus difficile à satisfaire sur les ordinateurs actuels.

Vérifier manuellement tous les résultats est nécessaire car cette tâche n'est pas complètement automatisable. Déléguer cette tâche aux auteurs des prouveurs maximise les chances de détecter un problème qui se serait produit dans la compétition car ils connaissent mieux que quiconque le comportement attendu de leur prouveur. C'est pour cette raison que nous sommes en faveur de la deuxième règle, et que nous aimerions voir ce principe appliqué dans les autres compétitions. Durant l'organisation des 7 compétitions SAT, nous avons pu noter que chaque année, au moins un auteur nous a rapporté un résultat inattendu lors de l'exécution de son prouveur. Il s'agit quelquefois de problèmes de notre part (gestion du temps CPU pour les prouveurs multi-thread vs multi-processus par exemple lors de la compétition SAT 2007) ou des problèmes dont la cause est restée inconnue (erreur de segmentation non reproductible, peut-être liée à un bug dans la gestion mémoire du prouveur ou au système d'exploitation ou au matériel). Si ces dysfonctionnements inexpliqués sont rares sur un cluster, nous avons noté un nombre significatif de problèmes lorsque nous faisions tourner la compétition sur une ferme de calcul en 2003 et 2005.

La troisième règle est une façon de s'assurer que les décisions sont prises sur la seule base du comportement des prouveurs. Dans le système actuel utilisé par les compétitions SAT et PB, au moins un des organisateurs doit connaître les noms des prouveurs soumis (car ceux ci sont recompilés avant d'être intégrés à la compétition, et les auteurs sont

contactés en cas de problème). Cette règle permet aussi de déléguer aux juges la gestion des cas particuliers qui se produisent lors d'une compétition.

La quatrième règle permet à la communauté toute entière de vérifier les résultats et d'en disposer comme bon lui semble. La compétition SAT n'est pas un très bon élève sur le long terme car ses données se sont trouvées éparpillées sur divers serveurs : Cincinnati, le LRI et le CRIL. A l'heure actuelle, les résultats des compétitions (2002–2005) ne sont plus disponibles dans le détail, mais uniquement via les publications résumant ces évènements [158, 147, 115, 116, 117].

La cinquième règle récompense la communauté pour l'organisation de la compétition en lui donnant accès à de nouveaux prouveurs et à de nouveaux benchmarks. Il faut noter que la diffusion de ces logiciels ou données devraient être formellement régulée par une licence. Dans la pratique, peu de prouveurs et quasiment aucun benchmark ne sont diffusés avec une licence.

Lors de la conférence fédérée sur la logique (*Federated Logic Conference*, FLoC) 2010, il y avait 7 compétitions organisées par différentes communautés (CASC, TERM-COMP, SMT-COMP, MISC, SAT Race, Pseudo Booléen, MAXSAT). On peut ajouter à ces compétitions QBF-Eval [144], qui a été présentée lors de la conférence SAT, mais qui fut organisée l'année précédente. Ces compétitions sont représentatives des pratiques qui existent dans les différents domaines de la logique.

Si la première et la quatrième règles sont communes à presque toutes ces compétitions, les règles deux et trois ne sont appliquées que dans le cadre de l'évaluation pseudo-booléenne, organisée par Olivier Roussel et Vasco Manquinho [129]. Ce n'est pas étonnant puisque Olivier Roussel a conçu son outil d'évaluation de prouveurs en se basant sur celui de Laurent Simon, et qu'il en a adopté les principes généraux. Depuis 2007, la compétition SAT utilise aussi ce système. L'évaluation pseudo-booléenne est aussi exemplaire pour la quatrième règle : les résultats de toutes les compétitions, jusqu'à la trace des prouveurs, sont disponibles en ligne. La compétition QBF ne satisfait que partiellement la quatrième règle, car elle ne permet pas d'accéder aux traces des prouveurs. La SAT Race ne satisfait pas non plus cette règle : d'un côté, cela est gênant car les prouveurs soumis à la SAT Race ne sont pas forcément disponibles, mais d'un autre côté, les conditions de la SAT Race sont facilement reproductibles sans utiliser de cluster vu le nombre limité d'instances utilisées (c'est d'ailleurs ce que font de nombreux chercheurs depuis l'existence de la SAT Race).

Certaines compétitions ont de bonnes raisons de ne pas appliquer les règles deux et trois :

- Ce n'est pas envisageable si la compétition se déroule pendant la conférence, comme CASC, TERM-COMP et SMT-COMP.
- Certaines compétitions utilisent un environnement qui est disponible en ligne en dehors des compétitions, qui permet à tout chercheur de tester ses prouveurs à loisir dans cet environnement (SMT-COMP et TERM-COMP). Pour les compétitions SAT ou pseudo booléenne, l'environnement d'exécution n'est pas accessible en dehors de ces compétitions.

Le tableau 5.1 résume pour chaque compétition les règles utilisées par chacune d'elles, si elles disposent ou non d'un point d'accès unique aux benchmarks (colonne LIB), et combien de prouveurs participent à la compétition (pour COMP-TERM, il y a 23 catégories

et les prouveurs sont le plus souvent adaptés pour une catégorie particulière). Certaines compétitions, notées *, ne disposent pas de site dédié pour les benchmarks, ils sont simplement disponibles depuis les sites web des compétitions (cas de PB, MAXSAT et MISC). Concernant la compétition pseudo-booléenne, une PB-LIB existe[2], créée par Andrew J. Parkes. Cependant, elle n'est plus active. Concernant MISC, il est recommandé de diffuser son prouveur sous forme de logiciel libre, mais ce n'est pas obligatoire.

Compétition	Règle 1	Règle 2	Règle 3	Règle 4	Règle 5	LIB ?	# prouveurs
SAT (2009)	x	x	x	x	x	-	50
SAT Race	x	-	x	-	-	-	19
PB	x	x	x	x	-	*	14
MAXSAT	x	-	-	-	x	*	17
QBF	x	-	x	-	-	x	11
SMT	x	NA	NA	x	x	x	10
CASC	x	NA	NA	x	x	x	34
TERM	x	NA	NA	x	-	x	>30
MISC	x	-	x	x	*	*	6

TABLE 5.1 – Résumé des caractéristiques des compétitions organisées lors de FLoC 2010.

5.1.1 La classification des benchmarks

La classification des benchmarks pour la compétition SAT dégage trois catégories principales :

aléatoires des benchmarks k-SAT générés aléatoirement selon une distribution uniforme, qui correspondent à un modèle mathématique connu [62]. Le principal intérêt de cette catégorie est de vérifier en pratique des résultats théoriques (heuristiques, minorant ou majorant de complexité, etc.). Cette catégorie est devenue la catégorie reine des prouveurs basés sur la recherche locale, seuls prouveurs capables de résoudre des instances satisfiables contenant des milliers voire des dizaines de milliers de variables, et des prouveurs DPLL, qui écrasent les prouveurs CDCL dans cette catégorie. Une analyse fine de cette catégorie lors de la compétition SAT 2005 a été publiée par Oliver Kullmann [105].

applications des benchmarks issus de problèmes « réels », réduits au problème SAT. L'objectif de cette catégorie est de montrer comment se comporte un prouveur SAT comme « moteur de résolution générique ». On trouve dans cette catégorie des instances de très grande taille, parfois résolues en quelques dizaines de secondes par certains prouveurs. Cette catégorie s'est longtemps intitulée « industrielle », à cause du nombre important de benchmarks provenant du domaine de la vérification de circuits (ceux d'IBM [173] et de Miroslav Velev [171] par exemple). Cette dénomination a perdu son sens au fur et à mesure des années, quand la diversité des origines des benchmarks s'est accrue (cryptographie [136], bio-informatique [40], etc).

fabriqués (*crafted*) des benchmarks conçus pour être difficiles (*born to be hard*). On retrouve dans cette catégorie des benchmarks connus pour fournir des preuves par résolution de taille exponentielles (pigeons [90], chaînes d'équivalence, etc.) ou qui sont construits pour être très difficiles pour les prouveurs actuels (spence [163]). L'intérêt de cette catégorie est double : déterminer les instances les plus petites

2. http://www.cirl.uoregon.edu/PBLIB/

non solubles par les prouveurs « état de l'art » afin de montrer les limites des approches actuelles et déterminer le prouveur SAT le plus véloce (plus grand nombre de décisions par seconde), qui représente en quelque sorte un prouveur robuste, généraliste. Cette catégorie met aussi souvent en avant les prouveurs utilisant des techniques de raisonnement spécifiques (comme LSAT [142], March [92, 93] ou TTS [162] par exemple).

Les benchmarks sont ensuite classés par source, tout simplement.

La classification des benchmarks est assez basique dans la communauté SAT par rapport à la communauté raisonnement automatique par exemple qui dispose de TPTP (*thousands of problems for theorem provers*), de la communauté SMT qui dispose de SMT-LIB, ou encore de la communauté QBF avec QBFLIB, des entrepôts de problèmes qui servent de source unique de benchmarks pour les compétitions.

Le principal avantage de cette approche est de pouvoir acquérir une connaissance fine de la difficulté des benchmarks disponibles, en lançant les prouveurs d'une compétition a posteriori sur l'ensemble des benchmarks de l'entrepôt. Cette information peut ensuite être prise en compte pour la sélection des benchmarks dans la compétition suivante. C'est de cette manière que fonctionnent TPTP/CASC, SMT-LIB/SMT-COMP et QBFLIB/QBFEVAL.

Les raisons pour lesquelles cette approche n'existe pas pour SAT sont multiples :

- Un entrepôt de benchmarks existe indépendamment de la compétition SAT depuis 1999 : SATLIB [96]. Le rôle de SATLIB fut important pour la récolte des benchmarks des compétitions des années 90, et elle a accueilli les benchmarks des compétitions SAT de 2002 et 2003. Malheureusement, les instances SAT de type application étaient à l'époque de taille conséquente par rapport à l'espace disponible sur SATLIB. Les archives de benchmarks utilisés lors des compétition se sont ensuite éparpillées entre le LRI et le CRIL, mais elles restent accessibles de façon centralisée depuis le site web de la compétition.
- Il y a une volonté pour la compétition SAT d'intégrer des instances SAT nouvelles à chaque compétition, afin d'introduire une part d'incertitude dans les résultats, et d'éviter que les approches de type portfolio comme SATZilla [172] ne soient trop avantagées : sans nouveaux benchmarks, ces approches gagneraient vraisemblablement toujours la compétition, puisque leurs prouveurs sont entraînés sur les instances déjà disponibles.
- L'idée de maintenir à jour les performances de tous les prouveurs sur tous les benchmarks disponibles pour SAT était l'idée de SAT-Ex [157]. Si cela était envisageable à la fin des années 90, le nombre croissant de benchmarks et de prouveurs depuis l'avénement de Chaff a rendu cette tâche très difficile, à moins de disposer d'une infrastructure type cluster de calcul dédiée à cette tâche tout au long de l'année : par exemple, lancer 50 nouveaux prouveurs (moyenne des participants à la compétition SAT) sur 6000 benchmarks (le CRIL dispose en local de près de 10 000 benchmarks SAT, pouvant contenir des doublons), pendant 20 minutes (temps limite de la première phase de la qualification) demanderait 12 machines fonctionnant sans interruption pendant 1 an ...

Cependant, les moyens technologiques changent rapidement : on utilise maintenant des ordinateurs avec plusieurs coeurs et plusieurs Gio de mémoire vive. On pourrait imaginer aujourd'hui effectuer ce travail avec trois machines quadri-coeurs au lieu des 12 machines monoprocesseurs. Mais on se retrouve alors avec un autre problème : comment garantir la

reproductilité des résultats si plusieurs couples (prouveur, benchmark) sont lancés sur la même machine ?

La table 5.2 illustre l'hétérogénéité des performances qui peut être observée lorsque plusieurs instances d'un même prouveur SAT sont lancées en parallèle sur le même benchmark sur un ordinateur de bureau avec un processeur quatre coeurs Intel Q8400 contenant un cache de second niveau de 2 Mio, avec 4 Gio de RAM fonctionnant sous Mandriva Linux 2010.1. Les benchmarks utilisés ne sont pas forcément les mêmes pour chaque prouveur, car nous souhaitions que chaque prouveur prenne au moins une minute pour résoudre son benchmark dans les conditions optimales, afin d'obtenir des résultats de mise à l'échelle comparables. Cependant, comme la taille de ces benchmarks varie, il faut interpréter ces résultats avec précaution. Il faut noter que ces résultats ne sont pas vraiment reproductibles, les chiffres présentés servent uniquement à illustrer la difficulté d'utiliser ces machines pour évaluer des prouveurs. On note que pour tous les prouveurs, le temps CPU nécessaire à la résolution d'une instance augmente quand le nombre d'instances du prouveur augmente. Cela s'explique par un partage des ressources, la mémoire du cache de second niveau et la mémoire vive essentiellement. On note que le comportement de picosat diffère des autres prouveurs, car le partage des ressources ne semble pas homogène entre les différentes instances du prouveur. C'est sans doute dû à une utilisation plus importante du cache de second niveau au sein de ce prouveur.

prouveur	benchmark	Nombre d'instances lancées en parallèle				Ralentissement
		1	2	3	4	
Minisat 2	ibm-2004-1_11-k80 262808v/1045990c	72	81, 81	110, 110, 110	130, 130, 131, 131	1,82
Minisat 2.2	ibm-2002-31_1r3-k30 43575v/194072c	164	193, 193	250, 265, 265	344, 345, 345, 347	2,12
Picosat 936	ibm-2004-1_31_2-k25 31125v/129472c	100	115, 193	157, 162, 239	212, 213, 305, 305	3,05
Lingeling 276	ibm-2002-19r-k100 310152v/1194099c	61	64, 64	72, 73, 74	83, 83, 83, 83	1,36
SAT4J 2.2	ibm-2002-05r-k90 180140v/976191c	141	157, 159	185, 186, 187	216, 217, 217, 217	1,54

TABLE 5.2 – Effet de l'exécution en parallèle de plusieurs prouveurs sur une machine mono-processeur quadri-coeurs de bureau sur le temps d'exécution de ces prouveurs (temps CPU utilisateur retourné par la commande Unix time en secondes).

La table 5.3 présente le même genre d'information pour un noeud de notre cluster contenant deux processeurs avec quatre coeurs (Intel Xeon X5550) contenant un cache de second niveau de 8 Mio, avec 32 Gio de RAM fonctionnant sous CentOS. On remarque que dans ce cas, les performances des prouveurs sont plus homogènes. Nous pensons que le fait de disposer de 4 fois plus de cache de second niveau permet d'expliquer cette différence de comportement. Un prouveur comme Lingeling, spécialement conçu pour limiter sa consom-

mation mémoire sur des machines 64 bits [30], fonctionne quasiment sans perte de performance lorsque 2 instances du prouveur tournent sur chaque processeur. Ce résultat peut être comparé à celui de Minisat 2.2 (car il est testé sur le même benchmark) qui subit un ralentissement de l'ordre de 10%.

prouveur (benchmark)	Nombre d'instances lancées en parallèle 1	2	3	4	5	6	7	8	Ralentissement
Minisat 2 (ibm-2004-23-k80) 165606v/686695c	197	197, 197	197, 222, 226	226, 226, 227, 228	225, 229, 249, 249, 249	246×2, 249, 250, 255, 269	247×2, 267, 268×2, 270, 279	271×4, 272×4	1,38
Minisat 2.2 (ibm-2002-31_1r3-k30) 43575v/194072c	91	91, 91	91, 100, 100	100, 101, 101, 102	100×2, 110×2, 116	110×3, 112×2, 115	110×3, 120×3, 123	122×8	1,34
Picosat 936 (ibm-2004-29-k55) 37714v/168958c	91	91, 93	93, 99, 99	98, 99, 99, 103	98, 106, 108×3	107×4, 110, 118	107×2, 108, 116×3, 119	117×8	1,28
Lingeling (ibm-2002-31_1r3-k30) 43575v/194072c	88	88, 88	88, 89, 89	89, 89, 89, 89	89, 89, 91, 91, 91	90×5, 91	90×3, 92×3, 94	92×8	1,04
SAT4J 2.2 (manol-pipe-g8nidw) 121170v/358426c	87	84, 87	85, 90, 100	90, 96, 99, 106	92, 98, 98, 103, 105	100, 103, 104, 105, 105, 107	98, 104, 105, 113, 113, 117, 119	99, 102, 103, 109, 113, 116, 117, 119	1,36

TABLE 5.3 – Effet de l'exécution en parallèle de plusieurs prouveurs sur une machine bi-processeurs quadri-coeurs type cluster sur le temps d'exécution de ces prouveurs (temps CPU utilisateur retourné par la commande Unix time en secondes).

Ce problème se pose pour la prochaine compétition SAT : le CRIL dispose de deux clusters, un ancien cluster de bi-Xeon avec 2Gio de RAM et un nouveau cluster de bi-quad core avec 32 Gio de RAM. Les compétitions précédentes ont tourné sur l'ancien cluster, avec un seul couple (prouveur, benchmark) lancé par noeud (deux processeurs). Ce choix était obligatoire vu la quantité limitée de mémoire vive disponible sur ces machines : 2Gio de RAM est un minimum pour expérimenter des prouveurs SAT sur des benchmarks de type application.

Cependant, nous disposons maintenant de machines plus puissantes, avec beaucoup plus de mémoire vive. On pourrait aisément lancer 4 couples (prouveur, benchmark) (voire 8) par noeud (2 processeurs, 8 coeurs). Cependant, à regarder les résultats précédents, on imagine bien que cette décision aura une conséquence non négligeable sur le comporte-

ment du prouveur. A l'heure actuelle, nous pensons faire tourner la première phase de la compétition en utilisant 4 prouveurs par noeuds (2 par processeur), et en utilisant uniquement 2 prouveurs par noeud dans la seconde phase, mais en relançant tous les prouveurs sur tous les benchmarks (et non plus les benchmarks non résolus comme dans les précédentes éditions).

5.1.2 Pourquoi demander la disponibilité des sources des prouveurs ?

La compétition SAT, comme CASC [165], demande aux participants de la compétition SAT de mettre à disposition de la communauté scientifique le code source de leur prouveur. Il y a plusieurs raisons pour cela :

- Le code source était nécessaire à l'origine pour compiler tous les prouveurs avec le même compilateur, en utilisant des bibliothèques statiques afin de lancer le prouveur dans un environnement complètement contrôlé [3].
- La compétition SAT est un service communautaire, qui donne aux vainqueurs une certaine publicité dans la communauté et en dehors. En contrepartie, la communauté souhaite avoir connaissance des techniques mises en oeuvre dans le prouveur. Les descriptions des prouveurs, volontairement ou non, ne précisent généralement pas toutes les spécificités du prouveur. A l'extrême, en 2003, Forklift a gagné la compétition SAT sans que quiconque ne sache quelles techniques étaient mises en oeuvre dans ce prouveur. Par conséquent, dès l'année suivante, le code source des prouveurs devait être mis à disposition de la communauté scientifique pour pouvoir concourir.

Assurer la disponibilité du code source est un moyen majeur pour diffuser la connaissance nécessaire pour concevoir un prouveur SAT efficace. Les auteurs des prouveurs récompensés lors des compétitions SAT, dans toutes les catégories, connaissent très bien le code des prouveurs des autres compétiteurs. On y trouve des détails algorithmiques de trop bas niveau pour apparaître dans un article scientifique, des structures de données optimisées, des constantes « magiques », etc. Il faut bien avouer qu'accéder à ces détails est facilité si l'on a une vision globale assez juste du prouveur. La publication de l'article sur Minisat [74] et la mise à disposition de son code source en 2003 me semble être une contribution importante à la diffusion de la connaissance sur l'implémentation des prouveurs CDCL.

Le problème de la communication de connaissances par code source est que le code source d'un prouveur n'est en général pas une référence valable pour un article scientifique. D'où l'idée de favoriser la publication de descriptions de systèmes très techniques. Nous avons essayé plusieurs approches : inviter les gagnants de la compétition SAT à publier un article dans les actes publiés après la conférence (2003 [74, 69] et 2004 [92, 126]). Nous avons enfin publié un volume entier du journal JSAT [117] dédié aux différentes compétitions liées à SAT 2005. Ce volume contient à la fois des descriptions de prouveurs, des descriptions de benchmarks, ainsi que diverses analyses concernant la compétition SAT, l'évaluation QBF ou l'évaluation pseudo-booléenne. On notera que l'article le plus cité de JSAT à ce jour est celui concernant le prouveur pseudo-booléen basé sur SAT `minisat+` [160], paru dans ce volume. Le second article le plus cité de JSAT est celui présentant picosat, vainqueur de la compétition SAT en 2007 [31], paru dans le volume 4 qui comporte

3. Autant que cela soit possible.

une seconde édition du journal concernant les évaluations de la conférence SAT 2007 [161]

Nous avons toujours demandé aux participants de la compétition SAT de fournir des descriptions de leur soumission (prouveur ou benchmarks). Nous avons édité des *booklets* en 2004 et 2009 rassemblant ces descriptions de benchmarks et de prouveurs. Cependant, ces descriptions n'étaient pas non plus des références valables pour un article scientifique (la technique de minimisation des clauses apprises de Minisat 1.14 est pendant longtemps restée limitée à la description de 2 pages de Minisat 2005). D'où l'idée de mettre en place un nouveau type de contributions dans la revue JSAT, spécifique pour la description de systèmes, recevant un processus de relecture adapté. On retrouve ce type de contributions dans diverses communautés du raisonnement automatique (CADE, TABLEAUX, etc.). Les premières descriptions de systèmes ont été publiées dans le volume 7 de JSAT en août 2010 dans le cadre du workshop *Pragmatics of SAT* organisé durant FLoC 2010.

Demander l'accès au code source des prouveurs a aussi ses inconvénients, le principal étant d'empêcher certains prouveurs de participer à la compétition. De nombreux prouveurs sont développés dans des sociétés (Intel, IBM, OneSpin, etc.), et ne pas autoriser ces prouveurs à participer à la compétition SAT donnerait une image consciemment faussée de l'état de l'art dans le domaine. Nous avons été attentifs à ce problème en instituant la possibilité d'évaluer des prouveurs dans les conditions de la compétition sans qu'ils puissent gagner (prouveurs hors concours), en relâchant la contrainte de diffusion du code source voire même d'un exécutable. C'est dans ces conditions que nous avons évalué le prouveur d'Intel, Eureka, en 2005. Une autre solution a été de mettre en place tous les deux ans une *SAT Race*, spécifiquement ouverte à tous les prouveurs sans contrepartie pour la communauté : Eureka y a participé en 2006 et 2008, mais n'a plus participé à la compétition SAT.

5.1.3 Problèmes rencontrés avec les prouveurs parallèles

Nous avons déjà noté les problèmes inhérents à l'utilisation de matériel multi-coeurs pour évaluer des prouveurs séquentiels. Ces mêmes problèmes, et de nouveaux, apparaîssent lorsqu'il s'agit d'évaluer des prouveurs parallèles. La compétition SAT accueille depuis 2007 les prouveurs parallèles. En 2007, nous comparions les performances de ces prouveurs sur deux processeurs. En 2009, nous les avons comparé sur des machines avec 2 processeurs, 4 et 16 coeurs. Nous avons fait le choix d'utiliser le temps CPU consommé par les prouveurs parallèles (somme du temps CPU de chaque process) afin de pouvoir comparer les prouveurs parallèles aux prouveurs séquentiels. On considère dans ce cas le temps CPU comme une ressource qui mesure l'effort nécessaire à la résolution d'une instance. Résoudre une instance en 6 heures sur une machine mono-processeur ou en 1 heure sur une machine avec 6 processeurs/cores, correspond au même coût CPU. La communauté du parallélisme a une autre vision des choses : seul le temps écoulé compte. C'est cette règle qu'a adopté la Sat Race pour évaluer les prouveurs parallèles en 2008 et 2010 : les prouveurs parallèles se voient offrir le même temps montre en main que les prouveurs séquentiels. A eux de tirer le meilleur parti des ressources disponibles. Cette même approche était adoptée cette année pour CASC par exemple.

Ces deux approches ont des avantages et des inconvénients. Nous pensons que l'approche de la compétition Sat est la plus juste en terme de ressources consommées. Le meilleur prouveur est celui qui résout un problème donné en utilisant le moins de ressources. Un autre point de vue est de penser en terme d'utilisateur final : il est préférable d'obtenir une réponse en 1 heure au lieu de 2 heures, même si l'on a consommé 4 coeurs, 2

fois plus de CPU, pour cela. Les ressources en CPU sont considérées ici comme perdues si elles ne sont pas utilisées.

La première approche pénalise dans une certaine mesure les prouveurs parallèles quand les ressources sont importantes : en 2009, notre approche revenait à diviser le temps limite par 16 pour un prouveur parallèle sur 16 coeurs. La deuxième favorise pleinement les prouveurs parallèles, puisqu'elle ignore complètement la notion de ressources. Il faudrait sans doute trouver un compromis entre ces deux approches [4].

Un autre problème posé par les prouveurs parallèles est la reproductibilité des résultats. En théorie, un prouveur lancé plusieurs fois sur la même instance dans les mêmes conditions doit toujours donner le même résultat. Cette hypothèse est nécessaire si l'on souhaite pouvoir vérifier les résultats de la compétition. Les prouveurs SAT sont généralement déterministes car cela permet à leur développeur de reproduire exactement le comportement du prouveur pour améliorer les structures de données ou corriger d'éventuels dysfonctionnements. Améliorer la reproductibilité des résultats dans un cadre parallèle demande des efforts de synchronisation réguliers et de tri des informations échangées qui vont forcément réduire l'efficacité du prouveur. Nous avons noté précédemment que simplement lancer des prouveurs en parallèle sur la même instance rendait leur comportement non reproductible si un partage des ressources est nécessaire. Ce problème de reproductibilité entraine un problème de robustesse des prouveurs, et d'évaluation lors de la compétition. Lors de la compétition SAT, les prouveurs parallèles sont lancés une seule fois sur chaque benchmark. Dans le cas de la SAT Race 2008, ils étaient lancés 3 fois, et une instance résolue par l'un des lancés était considérée comme résolue. On peut noter que les prouveurs étaient plus ou moins robustes (nombre de benchmarks résolus sur au moins un des trois lancés/nombre de benchmarks résolus à chaque lancé, pour un total de 100 benchmarks) : ManySAT 90/79, pMinisat 85/84 et MiraXT 73/66. pMinisat est bien plus robuste que les deux autres prouveurs, mais ManySAT est le prouveur ayant réussi à résoudre le plus d'instances. Durant la SAT Race 2010, la règle des trois lancés a été conservée, pour mesure (d'une certaine façon) la robustesse des prouveurs, mais seul le premier lancé était comptabilisé. On peut noter que les prouveurs se comportent différemment selon les lancés, sans que cela ne vienne changer l'ordre des prouveurs basé sur le nombre d'instances résolues : pLingeling 78/79/79/80, ManySat 1.5 75/74/74/78, ManySat 1.1 72/73/71/76, SArTagnan 70/70/72/76, Antom 67/65/68/69.

5.1.4 Classement des prouveurs

Le classement des prouveurs peut être basé sur plusieurs critères :

efficacité On se place ici dans un cadre très pragmatique : n benchmarks sont à résoudre. Les prouveurs capables de résoudre le plus grand nombre de benchmarks sont considérés comme les meilleurs.

robustesse On privilégie les prouveurs capables de résoudre des problèmes dans une grande variété de familles à des prouveurs capables de résoudre de nombreux benchmarks issus de familles spécifiques.

rapidité Le temps mis à résoudre les instances doit aussi être pris en compte : le temps limite donné aux prouveurs pour résoudre chaque instance influe sur les deux précédents critères. On peut aussi prendre en compte le temps cumulé nécessaire à la résolution des instances. Comptabiliser le temps limite en cas de non-résolution est

4. Ce problème n'est pas réglé à l'heure actuelle pour la compétition 2011.

une mauvaise idée car les prouveurs se retrouveraient souvent avec le même temps, alors que certains parmi eux pourraient résoudre le benchmark considéré en quelques minutes supplémentaires lorsque d'autres pourraient mettre des mois voire des années à le faire.

Lors des premières éditions de la compétition SAT, ces critères étaient pris en compte de manière lexicographique. Le premier critère était le nombre de familles/séries de benchmarks résolues : une série est résolue si au moins un benchmark de cette famille est résolu. Le second critère était le nombre total de benchmarks résolus. En cas d'égalité, c'est la somme des temps CPU qui était comptabilisé (inclure ou pas le temps limite en cas de non-résolution n'a pas d'importance ici, car les prouveurs ont résolu le même nombre de benchmarks). Cette approche est simple à mettre en oeuvre mais la notion de rapidité se trouvait reléguée en dernière position.

En 2005, Allen Van Gelder, juge de la compétition 2005, a défini un nouvel algorithme pour ordonner les prouveurs [80]. Il s'agit de prendre en compte tous les critères et de les agréger en une valeur numérique. Le principe de cette méthode est de partager des primes entre les prouveurs. Ces primes sont des constantes arbitraires.

- Une prime par série est partagée équitablement entre tous les prouveurs capables de résoudre cette série. Ainsi, une série que peu de prouveurs peuvent résoudre rapportera plus de points qu'une série résolue par tous les prouveurs.
- Une prime par benchmark est partagée équitablement entre tous les prouveurs capables de résoudre ce benchmark. Même remarque que précédemment.
- Une prime de temps par benchmark est partagée de manière non équitable, inversement proportionnelle au temps nécessaire à sa résolution.

La formule est du score est exactement :

$$solutionAward(p,s) = \frac{1000}{nbSolved(p)} \text{ si } s \text{ résout } p \text{ sinon } 0.$$

$$seriesAward(series,s) = \frac{1000 * factor}{nbSolved(series)} \text{ si } s \text{ résout } series \text{ sinon } 0.$$

avec $factor = 3$ si $|series| > 5$, sinon $factor = 1$.

$$speedFactor(p,i) = \frac{10000}{1 + timeUsed(p,i)} \text{ si } i \text{ résout } p \text{ sinon } 0.$$

$$speedAward(p,s) = \frac{1000 * speedFactor(p,s)}{\sum_i speedFactor(s,i)}$$

$$score(s) = \sum_{series} seriesAward(series,s) + \sum_p solutionAward(p,s) + \sum_p speedAward(p,s)$$

Personnellement, cette solution me convenait : elle avait l'avantage de mettre en avant les prouveurs capables de résoudre des benchmarks qu'aucun autre prouveur ne pouvait résoudre. C'était plus spécifiquement le cas dans la catégorie fabriquées (en 2007 par exemple, le prouveur tts [162] s'est retrouvé deuxième du classement). Les participants à la compétition SAT n'ont pas vraiment adopté cette formule, pour diverses raisons :

- Le classement final peut sembler surprenant si l'on ne comprend pas l'objectif du score.

– Il est difficile de calculer les scores à partir des résultats bruts (nombre d'instances résolues, temps de résolution), donc la vérification des scores par la communauté est difficile.
– Une trop grande importance est donnée aux instances résolues par un seul prouveur.

De notre point de vue, le principal problème de ce score est d'être dépendant des prouveurs participants à la compétition. C'est un problème car ce score ne peut pas être comparé a posteriori à celui d'un autre prouveur, comme on peut le faire lorsque l'on compare le nombre de benchmarks résolus : du moment que les conditions d'expérimentations sont les mêmes, rien n'empêche de comparer des résultats publics avec de nouveaux résultats, comme nous l'avons fait dans le chapitre précédent.

Lors de la compétition 2009, nous avons demandé aux compétiteurs de voter pour leur méthode de classement préférée : le résultat était clairement en faveur d'une approche très simple, utilisée dans beaucoup d'évaluations, basée sur un classement des critères suivants (le plus prioritaire d'abord) : nombre de benchs résolus, temps cpu cumulé par instances résolues. Par rapport à la méthode utilisée lors de la compétition SAT de 2002, on perd la notion de série, donc de robustesse des prouveurs.

Récemment, j'ai eu l'occasion de développer avec Pierre Marquis et Meltem Öztürk une traduction de calcul d'agrégation d'ordres d'intervalle en problème de couverture binaire [110]. Ce travail se situe dans le cadre de l'agrégation de votes pour différentes alternatives. Chaque votant doit comparer deux à deux les alternatives et exprimer soit une préférence soit une indifférence entre celles-ci. Il est connu en théorie du choix social qu'il est impossible d'agréger des préférences individuelles en une préférence collective en satisfaisant des règles attendues (universalité, non dictature, unanimité, indifférence aux options non pertinentes, théorème de Arrow [15]). Nous avons choisi d'utiliser une approche qui consiste à minimiser la somme des distances entre les préférences individuelles et la préférence collective. Notre mesure de distance prend en compte qu'une différence entre s_1 est préféré à s_2 et s_2 est préféré à s_1 est plus importante qu'entre s_1 est indifférent à s_2 et s_2 est préféré à s_1, en nous basant sur la mesure de distance définie par [103].

Cette approche peut être utilisée pour ordonner les prouveurs SAT. Les votants sont les benchmarks et les alternatives sont les prouveurs . Soit $temp(s, p)$ le temps mis par le prouveur s pour résoudre le benchmark p. Soit s_1 et s_2 deux prouveurs, s_1 est préféré à s_2 ssi $temp(s_1, p) + 1 < temps(s_2)$. Si aucun des prouveurs n'est préféré à l'autre, alors il sont équivalents (c'est le cas si la différence des temps de résolution est faible, moins d'une seconde, ou si les deux prouveurs n'arrivent pas à résoudre le benchmark dans le temps limite)[5]. Chaque benchmark induit donc un ordre d'intervalle sur les prouveurs. Utiliser 100 benchmarks dans la SAT Race revient à considérer 100 votants. La minimisation des distances cumulées correspond à un problème d'optimisation de type couverture binaire *binate covering problem* [60]. Les détails de cette transformation sont disponibles dans [110] (article joint chapitre 13)

Nous avons expérimenté notre approche sur les instances de la SAT Race 2006. Si Sat4j PB résolution n'arrive pas à prouver l'optimalité de la solution en 30mn, par contre Sat4j PB plans-coupe trouve 10 solutions optimales en quelques secondes. La figure 5.2 montre la première solution trouvée. Les arcs pleins représentent la relation de préférence stricte. Les arcs en pointillés représentent la relation d'indifférence. Les nombres qui étiquettent

5. Nous aurions pu utiliser un pourcentage du temps de résolution, car les intervalles n'ont pas besoin d'être de taille fixe.

les arcs correspondent aux nombres de votants qui ont fait ce choix. Chaque noeud du graphe est associé à 15 arcs. On remarque que Minisat2 est généralement préféré aux autres prouveurs, hormis deux variantes de Minisat, Actin et cadenceminisat. Notre approche ne permet pas de classer les prouveurs au sens d'un ordre ou d'un préordre total. En effet, la relation résultante est encore un ordre d'intervalle. Les meilleurs prouveurs de la SAT Race au sens de cette agrégation, i.e. les éléments non dominés de l'ordre d'intervalle, sont ceux qui sont représentés en haut du graphe. Le classement officiel de la SAT Race 2006 est représenté par des arcs épais non étiquetés.

minisat	0	56	53	49	45	55	62	51	62	67	66	65	67	68	71	70
eureka	36	0	44	42	48	52	54	37	55	63	60	61	60	63	63	62
rsat	27	47	0	31	37	43	52	35	56	66	62	60	67	67	67	67
cadence	27	41	38	0	32	42	47	28	51	56	55	52	57	59	59	56
actin	29	33	40	33	0	44	51	32	55	59	54	58	60	57	60	59
barcelogic	25	22	28	29	27	0	36	26	39	54	49	46	53	56	53	49
picosat	15	23	20	25	19	27	0	18	35	46	43	44	46	50	52	47
qpicosat	25	34	38	37	34	41	45	0	43	48	46	47	50	50	50	47
tinisat	18	23	18	22	15	22	28	24	0	36	38	40	40	43	48	44
sat4j	9	17	7	9	9	14	16	15	24	0	29	30	38	34	38	38
qcompsat	9	11	12	10	10	15	17	7	22	26	0	26	31	28	35	30
zchaff	13	10	14	17	13	13	15	14	14	28	21	0	29	28	33	29
compsat	7	15	7	10	6	9	12	11	16	15	21	21	0	22	27	24
mxc	4	10	5	7	6	9	8	5	12	18	15	16	20	0	21	20
mucsat	4	9	5	5	6	7	5	7	9	15	10	12	17	18	0	19
hypersat	6	8	7	10	10	14	11	9	13	14	16	15	19	18	20	0

TABLE 5.4 – Matrice des votes pour la SAT Race 2006. 100 votants. Les prouveurs sont dans le même ordre en colonne. La valeur d'une cellule (ligne,col) correspond au nombre de votants qui préfèrent ligne à col. L'indifférence est obtenue en faisant 100 - (ligne,col) - (col,ligne).

Cette première solution est celle qui contient le plus d'indifférences. Les 9 autres solutions changent les indifférences entre picosat et mxc, qpicosat et compsat, qpicosat et mxc, qpicosat et mucsat par des préférences :

1. qpicosat>mucsat
2. qpicosat>mxc, qpicosat>mucsat
3. picosat>mxc, qpicosat>mxc
4. picosat>mxc
5. picosat>mxc, qpicosat>mxc, qpicosat>mucsat
6. picosat>mxc, qpicosat>mucsat
7. picosat>mxc, qpicosat>compsat, qpicosat>mxc, qpicosat>mucsat
8. qpicosat>compsat, qpicosat>mxc, qpicosat>mucsat
9. qpicosat>compsat

Si la méthode de classement des prouveurs influe directement sur le classement final, une autre manière de modifier le classement est de faire varier la méthode utilisée pour sélectionner les benchmarks.

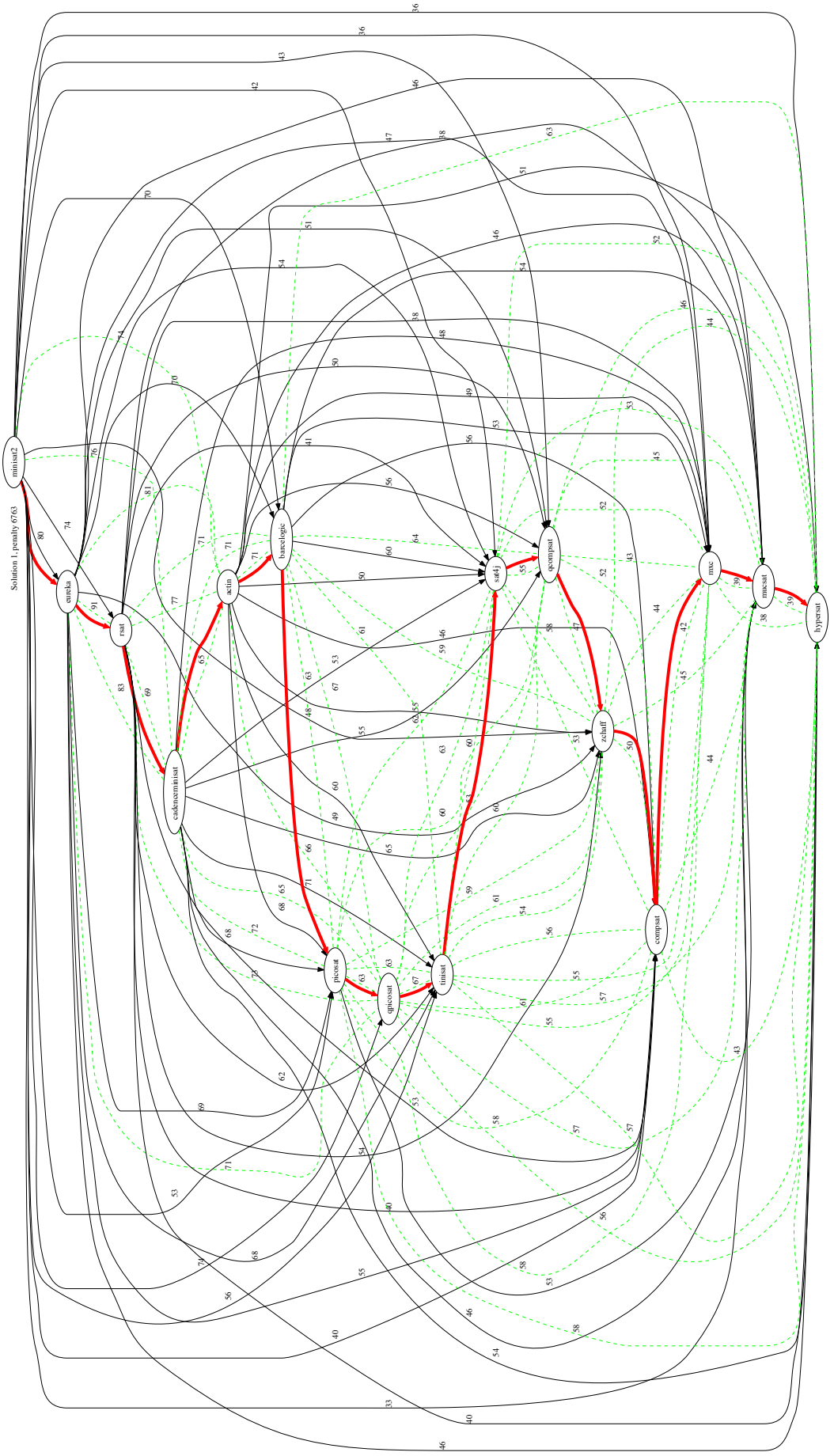


FIGURE 5.2 – Ordonnancement des prouveurs de la SAT Race 2006 en utilisant la technique d'agrégation d'ordre d'intervalle.

.

	3-SAT				5-SAT				7-SAT			
Generated benchmarks parameters												
	ratio	start	stop	by	ratio	start	stop	by	ratio	start	stop	by
Medium	4.26	360	560	20	21.3	90	120	10	89	60	75	5
Large	4.2	2000	18000	2000	20	700	1100	100	81	140	220	20
Number of generated benchmarks												
	SAT		?		SAT		?		SAT		?	
Medium	110		110		40		40		40		40	
Large	90		-		50		-		50		-	

FIGURE 5.3 – Benchmarks aléatoires générés pour la compétition SAT 2009

5.1.5 Processus de sélection des benchmarks

Il existe de nombreux benchmarks disponibles pour la communauté SAT, grâce aux premières compétitions des années 90 et au format d'entrée standard Dimacs. Dans la catégorie aléatoire, les benchmarks sont générés depuis 2005 selon les préconisations de Oliver Kullmann [105]. Les différentes instances générées en 2009 sont résumées à la figure 5.3. Depuis le début, la compétition SAT accueille de nouveaux benchmarks dans les catégories applications et fabriquées, soumis en même temps que les prouveurs. Un compétiteur peut soumettre à la fois un prouveur et des benchmarks. La catégorisation des benchmarks dans l'une ou l'autre catégorie est validée par les juges de la compétition.

D'une année sur l'autre, des benchmarks sont réutilisés. Durant les premières compétitions SAT, l'ensemble des benchmarks non résolus d'une compétition étaient systématiquement ajoutés à ceux utilisés lors de la compétition suivante. La conséquence de cette approche a été d'obtenir une grande partie d'instances non résolues en 2004.

Cependant, utiliser des instances trop faciles (résolues par tous les prouveurs) ou trop difficiles (résolues par aucun prouveur) n'a pas de sens dans une compétition car ils ne permettent pas de discriminer les prouveurs. Il faut donc trouver des instances susceptibles d'être résolues par au moins un prouveur (de difficulté « moyenne »).

Nous essayons d'évaluer les prouveurs sur environ 300 benchmarks différents par catégorie. Dans la catégorie aléatoire, comme les ensembles de benchmarks dédiés aux prouveurs complets et incomplets sont quasi disjoints (les instances aléatoires satisfiables de petite taille sont triviales pour les prouveurs incomplets), nous en générons le double. Chaque compétition évalue donc chaque prouveur sur environ 1200 benchmarks.

Depuis la compétition 2007, les benchmarks anciens ou nouveaux sont classés comme faciles, moyens et difficiles à l'aide d'un ensemble de prouveurs témoins (en 2009, nous avons utilisé les gagnants 2007 de chaque catégorie comme prouveurs témoins). En 2009, les juges décidèrent de composer le jeu de benchmarks de 10% d'instances faciles, 40% d'instances moyennes et 50% d'instances difficiles.

Durant CASC, les prouveurs sont lancés sur des benchmarks de difficulté croissante, ce qui permet d'observer des changements de classement lors de l'affichage en direct des résultats.

5.1.6 A propos de la « robustesse » des prouveurs SAT

Au début de la compétition SAT, tous les benchmarks étaient « mélangés » avant d'être utilisés dans la compétition, c'est à dire que les variables étaient renommées, les littéraux étaient déplacés dans les clauses, et les clauses déplacées dans la formule. C'était une mesure simple de prévention contre la reconnaissance syntaxique des benchmarks, puisque bon nombre d'entre eux sont disponibles avant le début de la compétition.

Rapidement, il est apparu que ces changements modifiaient le comportement des prouveurs :
- La modification syntaxique des benchmarks change le comportement des prouveurs (*lisa syndrom* [115]) : les prouveurs peuvent aller jusqu'à résoudre un benchmark sous une forme et ne pas le résoudre sous une autre.
- La modification des benchmarks dans la catégorie application ne permet pas de rendre compte des performances réelles des prouveurs dans cette catégorie : les prouveurs sont bien moins performants sur les instances mélangées que sur les instances originales [116].

Ces constatations nous ont conduit à comparer les résultats des instances modifiées aux originales en 2005 et à éliminer la modification des benchmarks depuis. Idéalement, il faudrait prendre en compte ce problème de robustesse au niveau de la compétition. [141] propose de lancer chaque prouveur sur 15 variantes syntaxiques de chaque benchmark pour obtenir des résultats statistiquement significatifs.

Nous sommes bien conscients des limites de l'approche actuelle. Cependant, nous pensons qu'elle est adaptée au rôle de la compétition : animer la communauté et fournir une vitrine technologique pour la résolution pratique du problème SAT.

5.2 Les autres compétitions

En dehors de la compétition SAT, j'ai eu l'occasion de participer à d'autres compétitions, comme conseiller technique (SAT Race), comme organisateur aux débuts de la compétition QBF, ou comme simple compétiteur (pseudo-booléen, MaxSat et MISC).

5.2.1 SAT Race

Organisateur : Carsten Sinz

La SAT Race fut organisée par Carsten Sinz en 2006, car nous avions décidé de faire une pause au niveau de la compétition SAT (le nombre de participants venait de baisser pour la première fois après trois années de hausse consécutives, c.f. figure 5.1). Depuis, les compétitions SAT et les SAT Race alternent chaque année pour animer la communauté des développeurs de prouveurs SAT.

La SAT Race a un but un peu différent de la compétition SAT :
- Elle se focalise sur la catégorie applications.
- Elle fonctionne avec un nombre de benchmarks limité (100).
- Le temps limite est plus court que celui de la première phase de la compétition (15 minutes contre 20 minutes).
- La règle de la diffusion du code source des participants est levée, ce qui rend l'accès à la compétition possible à divers prouveurs développés hors du monde académique.

Le classement de la SAT Race en 2006 et 2008 suivait une approche similaire au classement de la compétition SAT 2005 : un bonus de rapidité était distribué aux prouveurs. Suivant l'approche prise par la compétition SAT 2009, le classement de 2010 est simplement basé sur le nombre de benchmarks résolus.

Dès sa première édition, les benchmarks et les conditions d'expérimentation de la SAT Race ont été repris dans les parties expérimentales des publications sur SAT, pour deux raisons : la mise à disposition d'un ensemble de benchmarks limité couvrant divers domaines d'application ; la possibilité de reproduire les résultats sur un seul ordinateur en temps raisonnable (en 25 heures maximum par prouveur).

5.2.2 QBF

Organisateurs : Claudia Peschiera, Luca Pulina, Armando Tacchella

Fort de notre expérience sur la première compétition SAT, nous avons mis en place avec Armando Tacchella et Massimo Narizzano les premières évaluations de prouveurs QBF [119, 111]. L'une des principales difficultés concernant les prouveurs QBF est l'impossibilité de vérifier les résultats des prouveurs. Cela pose un souci à la fois pour les concepteurs des prouveurs, qui peuvent difficilement valider les réponses de leur prouveur, et pour la validation des résultats de la compétition : si deux prouveurs QBF répondent de manière contradictoire, il n'y a pas de moyen aisé d'identifier le prouveur incorrect. Des mécanismes de certification sont donc apparus [138].

Le système de classement de la compétition QBF est YASM (*Yet Another Scoring Method*) [139]. Comme dans le cadre du score *purse* de la compétition SAT, le score obtenu est dépendant des autres prouveurs participant à la compétition.

L'évaluation QBF et QBFLIB fournissent benchmarks et résultats des évaluations QBF depuis 2004.

OpenQBF a été soumis aux évaluation QBF de 2003 à 2006. QBFL [56] a été soumis aux évaluations QBF de 2004 à 2006.

5.2.3 Pseudo-booléenne

Organisateurs : Vasco Manquinho et Olivier Roussel

La principale différence entre la compétition SAT et la compétition pseudo-booléenne est la gestion des problèmes d'optimisation et celle du format d'entrée. On retrouve dans la validation des valeurs optimales le même problème que pour les réponses UNSAT. En effet, même si pour chaque instance pour laquelle une solution optimale est trouvée un problème de décision était construit en remplaçant la fonction objectif par une contrainte sur cette dernière forçant sa valeur à être strictement inférieure à la valeur optimale, seule une preuve de non-optimalité pourrait être vérifiée (réponse SAT). L'évaluation pseudo-booléenne gère aussi les solutions non optimales, et fournit des indications comme le temps nécessaire à déterminer la solution optimale (à comparer avec le temps incluant la preuve d'optimalité).

Les instances de l'évaluation pseudo-booléenne doivent être classées selon la taille des coefficients des littéraux car la plupart de prouveurs sont limités par la représentation machine des entiers (sur 32 ou 64 bits). Cette distinction a permis de fournir des problèmes adaptés à la plupart des prouveurs pseudo-booléens disponibles tout en permettant de comparer les prouveurs ayant fait un autre choix.

L'évaluation pseudo-booléenne est un exemple en terme d'ouverture : les organisateurs fournissent une bibliothèque permettant de lire les instances dans le format de la compétition en C/C++/Java, une linéarisation automatique des contraintes non linéaires est effectuée pour les prouveurs qui ne supportent pas nativement ces contraintes, les prouveurs peuvent ne participer qu'à quelques catégories (contrairement à la compétition SAT), etc. En 2010, les erreurs découvertes sur les prouveurs des catégories WBO ont été gérées de manière exemplaire : chaque participant a eu une chance de régler rapidement les erreurs détectées à la lumière de quelques benchmarks afin de pouvoir publier quelques résultats utiles pour la communauté.

SAT4J a participé à toutes les évaluations pseudo-booléennes depuis 2005, et supporte les différentes variantes du format d'entrée : contraintes non linéaires, *weighted boolean optimization*. Les informations en ligne de ces évaluations ont été un outil précieux pour valider les résultats de nos prouveurs.

5.2.4 MAXSAT

Organisateurs : Josep Argelich, Chu-Min Li, Felip Manya, et Jordi Planes

L'évaluation MAXSAT existe depuis 2006. La première édition contenait deux catégories, MaxSat et Weighted MaxSat. L'évaluation suivante a introduit le concept de clause dure, Partial MaxSat et Weighted Partial MaxSat. Seules les réponses optimales sont comptabilisées. Par conséquent, seuls les prouveurs complets sont évalués : ce choix est discutable, car les prouveurs incomplets pour ce problème sont utilisés depuis longtemps avec succès.

Les résultats détaillés des prouveurs ne sont disponibles que depuis 2009. Avant cela, il fallait rejouer l'évaluation pour étudier ces résultats. Comme l'ensemble des prouveurs n'étaient pas disponibles, cela minimisait l'impact de l'évaluation. Depuis 2009, de nombreuses améliorations ont été apportées à la présentation des résultats :

- les traces des prouveurs sont désormais visibles ;
- les résultats par catégorie sont proposés par famille de benchmarks, en classant les prouveurs par ordre décroissant de nombre de benchmarks résolus, et en mettant en avant les meilleurs prouveurs par famille. Cette approche, très visuelle, donne une bonne vision du comportement des prouveurs, et de leur robustesse. Nous allons adopter cette représentation pour la prochaine compétition SAT, pour la seconde phase.

Dès l'évaluation 2008, les premiers benchmarks MaxSat de type applications sont apparus [152, 89]. Il y en a régulièrement depuis. L'évaluation MaxSat a atteint ses objectifs.

Sat4j a participé a toutes les évaluations MaxSat depuis 2006. En 2009, Sat4j était considéré comme un prouveur MaxSat de l'« état de l'art ».

5.2.5 MiSC/MISC

Organisateurs : Stefano Zacchiroli, Ralph Treinen et Roberto Di Cosmo

Le projet Mancoosi a organisé en janvier 2010 une compétition interne de gestion-

naires de dépendances, MiSC[6], et en juin 2010 une compétition internationale MISC[7]. La première avait pour but de préparer la seconde. MiSC a permis de tester l'infrastructure de compétition, le format de représentation des problèmes, et de fournir un premier jeu de benchmarks CUDF. MISC a permis d'attirer une nouvelle équipe de recherche sur la problématique de la gestion des dépendances.

Ces deux compétitions sont différentes des autres compétitions mentionnées précédemment dans le sens où elles ne sont pas organisées par des personnes proches des prouveurs mais par des personnes proches du problème à résoudre. Ceci a une conséquence sur le format d'entrée des benchmarks, CUDF [169], qui est beaucoup plus complexe que pour les autres compétitions. Il s'agit en fait d'un format de modélisation, pas d'un simple format d'entrée pour prouveur. Si cela est pratique pour l'utilisateur final du produit (j'ai déjà mentionné que nous souhaitons l'utiliser dans Eclipse pour décrire des situations à tester), c'est à mon avis un frein à la participation à la compétition. p2cudf ne respecte pas le format d'entrée CUDF général, mais fait des hypothèses sur les instances qui seront disponibles pour la compétition (en supposant par exemple qu'elles sont bien formées, et en ignorant les informations non pertinentes).

On suppose que m prouveurs participent à la compétition. Le classement de ces prouveurs se fait à partir de points, obtenus à partir des différentes réponses du prouveur :

1. les prouveurs ayant trouvé une solution sont classés selon la qualité de la solution, et obtiennent le nombre de points correspondant à leur classement, les prouveurs étant ex-aequo si ils trouvent des solutions équivalentes. Le temps n'est pas pris en compte ici.
2. les prouveurs ne trouvant pas de solution obtiennent $2 * m$ points.
3. les prouveurs terminant sans donner de réponse (seg. fault, timeout, etc.) obtiennent $3 * m$ points.
4. les prouveurs donnant une solution incorrecte obtiennent $4 * m$ points.

On peut noter que cette façon d'accorder des points correspond vraiment à un point de vue utilisateur. On préfère obtenir une solution, même si elle n'est pas optimale. Sinon, on veut obtenir un message qui indique que ce n'est pas possible. Les cas de prouveurs qui atteignent le temps limite ou qui se plantent est à éviter. Enfin, le pire des cas est celui d'un prouveur qui ne répond pas correctement.

Cependant, cette approche a révélé quelques faiblesses lors de MISC :
- Un prouveur qui atteint le temps limite est plus pénalisé qu'un prouveur qui ne retourne pas de solution. Par conséquent, un prouveur qui ne retourne jamais une seule solution peut être mieux classé qu'un prouveur qui retourne quelques solutions et atteint le temps limite. Notre prouveur p2cudf s'est retrouvé dans ce cas pour la catégorie `cudf_set`, car nous n'avions pas compris le codage des contraintes singleton en CUDF (auto-conflit), ce qui rendait systématiquement contradictoire la traduction de ces problèmes en problème d'optimisation pseudo-booléen.
- Il y a peu de différences de points entre une solution optimale et une solution non optimale. Ainsi, un prouveur qui répond toujours de manière sous-optimale, mais sans atteindre le temps limite, peut se retrouver mieux classé qu'un prouveur qui répond

6. http://www.mancoosi.org/misc-live/20100114/
7. http://www.mancoosi.org/misc-2010/

majoritairement de manière optimale mais atteint le temps limite à quelques occasions. C'est la raison pour laquelle p2cudf-trendy a été classé devant uns-trendy sur la category `debian-dudf`. Notre encodage des paquetages recommandés, nouveauté de MISC, n'étant pas correcte, notre prouveur retournait toujours une solution sous-optimale.

Ces problèmes étaient déjà présents lors de MiSC : p2cudf a été déclaré vainqueur des deux pistes de MiSC simplement parce que le prouveur `unsa` s'interrompait brutalement sur les instances `caixa`, et qu'il a été fortement pénalisé pour cela.

Dans la plupart des compétitions, un prouveur dans le cas 4 serait simplement disqualifié, et le cas 3 est simplement ignoré. Concernant le cas 2, cela dépend de la sémantique : dans p2cudf, répondre qu'il n'y a pas de solution signifie vraiment qu'il n'y a pas de solution (réponse UNSAT sur l'encodage en PBO). Dans MiSC, il s'agit d'un réponse qui signifie plutôt « je ne sais pas », qui ne devrait pas être pénalisée de manière différente que le cas 3. Reste le cas 1. On peut se rapprocher ici de ce qui se fait dans la compétition pseudo-booléenne ou en MaxSat : dans les deux cas, seule les solutions optimales sont comptabilisées.

5.3 Leçons apprises

Voici quelques règles qui nous semblent importantes lorsque l'on souhaite organiser une compétition.

Format Les benchmarks doivent être disponibles dans un format aussi simple que possible à gérer pour l'auteur du prouveur. Le cas du format Dimacs est sans doute un extrême en terme de simplicité, dans le sens où s'il est trivial à lire pour un prouveur, c'est un format assez fruste pour un utilisateur de technologie SAT (l'entête avec la valeur maximale de l'identifiant de variables et le nombre de contraintes est un frein à la construction "à la volée" des instances).

Jeu de test Un ensemble représentatif de benchmarks avec les réponses attendues doivent être disponibles. C'est d'autant plus important quand il est question de solution optimale. On a pu noter une grande différence de stabilité et de correction entre les prouveurs soumis à la première compétition SAT en 2002 et ceux soumis aux compétitions suivantes : la principale raison de cette amélioration était de centraliser dans une archive une grande diversité de benchmarks pour lesquels les résultats de la compétition 2002 étaient disponibles en ligne.

Vérification des résultats Les réponses individuelles des prouveurs doivent être vérifiées quand c'est possible (contre exemple : réponse UNSAT, QBF). La cohérence globale des solutions doit aussi être vérifiée : on ne doit pas avoir un prouveur qui déclare une instance SAT et un autre prouveur qui déclare cette instance UNSAT, on ne doit pas avoir un prouveur qui déclare une solution optimale et avoir un autre prouveur qui trouve une solution meilleure.

Discussion avec l'auteur Si un problème est détecté durant la compétition, ce problème doit être rapporté le plus rapidement possible à l'auteur, afin de lui donner une chance de corriger ce problème, et éventuellement de soumettre une version corrigée du prouveur. De notre expérience, la plupart des problèmes sont facilement corrigés par leurs auteurs : ils résultent souvent de changements de dernière minute, ou d'état très particuliers du prouveur. Rapporter les résultats de prouveurs incorrects n'a pas de

sens : on ne peut rien conclure du comportement du prouveur. Nous disqualifions les prouveurs incorrects dans la compétition SAT. Détecter ce genre de problème sur un prouveur donne l'occasion à l'auteur de soumettre une version corrigée hors concours. Les conséquences d'une incorrection peuvent dépendre du contexte de la compétition : on notera que cette année, de nombreux prouveurs se sont révélés incorrects sur les instances de la nouvelle catégorie WBO dans l'évaluation pseudo-booléenne, probablement car il y avait peu de benchmarks disponibles avant l'évaluation. Les organisateurs ont décidé de laisser une chance aux auteurs de corriger leur prouveur afin d'avoir quelques résultats sur cette nouvelle catégorie de benchmarks. Ce genre de décision a tout son sens sur des problèmes nouveaux.

Accès aux résultats Le point primordial à notre avis est de laisser l'auteur et le reste de la communauté accéder aux détails de l'exécution du couple (prouveur, benchmarks). Cela permet à l'auteur de valider les résultats de son prouveur durant la compétition et à la communauté de vérifier les résultats a posteriori.

Validité des résultats publiés Il arrive quelquefois que certains résultats soient invalidés a posteriori (benchmark déclaré UNSAT par un seul prouveur pendant l'évaluation et prouvé SAT par la suite, problème découvert dans le décodage de la sortie du prouveur, problème découvert dans le prouveur lui-même par son auteur, etc.) Les résultats étant publics, et servant de référence aux autres chercheurs, il nous semble important de mettre en ligne sinon des résultats corrigés, des indications qui indiquent clairement les problèmes découverts après la compétition.

5.4 Progrès ou pas ?

Une question qui revient souvent est de savoir si des progrès significatifs sont apparus depuis l'avènement de Chaff. La figure 5.4 montre les performances des différents gagnants de la compétition SAT *dans la catégorie application* et des vainqueurs de la SAT RAce dans les conditions de la première phase de la compétition SAT 2009. Ces mêmes résultats sont montrés sous forme de « cactus » sur la page suivante. Zchaff [137] et Limmat sont exactement les binaires utilisés lors de la compétition de 2002. La version de Berkmin [85] soumise à la compétition 2002 contenait un bug qui limitait ses performances, nous avons donc utilisé Berkmin 561, la seule version de Berkmin disponible en ligne et soumise à la compétition SAT 2003. Forklift est le gagnant de la compétition 2003, pour lequel nous n'avons aucune information. Satzoo est le gagnant de la catégorie fabriquée en 2003, et ancêtre de Minisat. Le prouveur Siege [151] n'a jamais gagné la compétition SAT, car aucune description de son fonctionnement interne ni son code source n'a été disponible. Il était considéré comme le meilleur prouveur disponible en 2003/2004. Zchaff 2004 [126] représente les gagnants de 2004 (dont Jerusat faisait aussi partie). SatELite [75] est le vainqueur de la compétition 2005. Minisat 2.0 était le gagnant de la SAT Race 2006. La version utilisée ici est celle soumise à la compétition 2007, très similaire. Picosat [31] et Rsat [145] sont les gagnants de la compétition 2007. Minisat 2.1 est la gagnant de la SAT Race 2008. Precosat et glucose [19] représentent les gagnants de la compétition 2009 dans la catégorie application, clasp celui de la catégorie fabriquée. Cryptominisat est le vainqueur de la SAT Race 2010 dans la piste séquentielle, et la version parallèle de lingeling a gagné la SAT Race dans la piste parallèle.

De ces résultats, on peut conclure qu'effectivement, les prouveurs SAT se sont significativement améliorés en 9 ans. On peut remarquer que Siege a été un prouveur terriblement efficace. Il est bien dommage que nous ne connaissions toujours pas les détails de sa

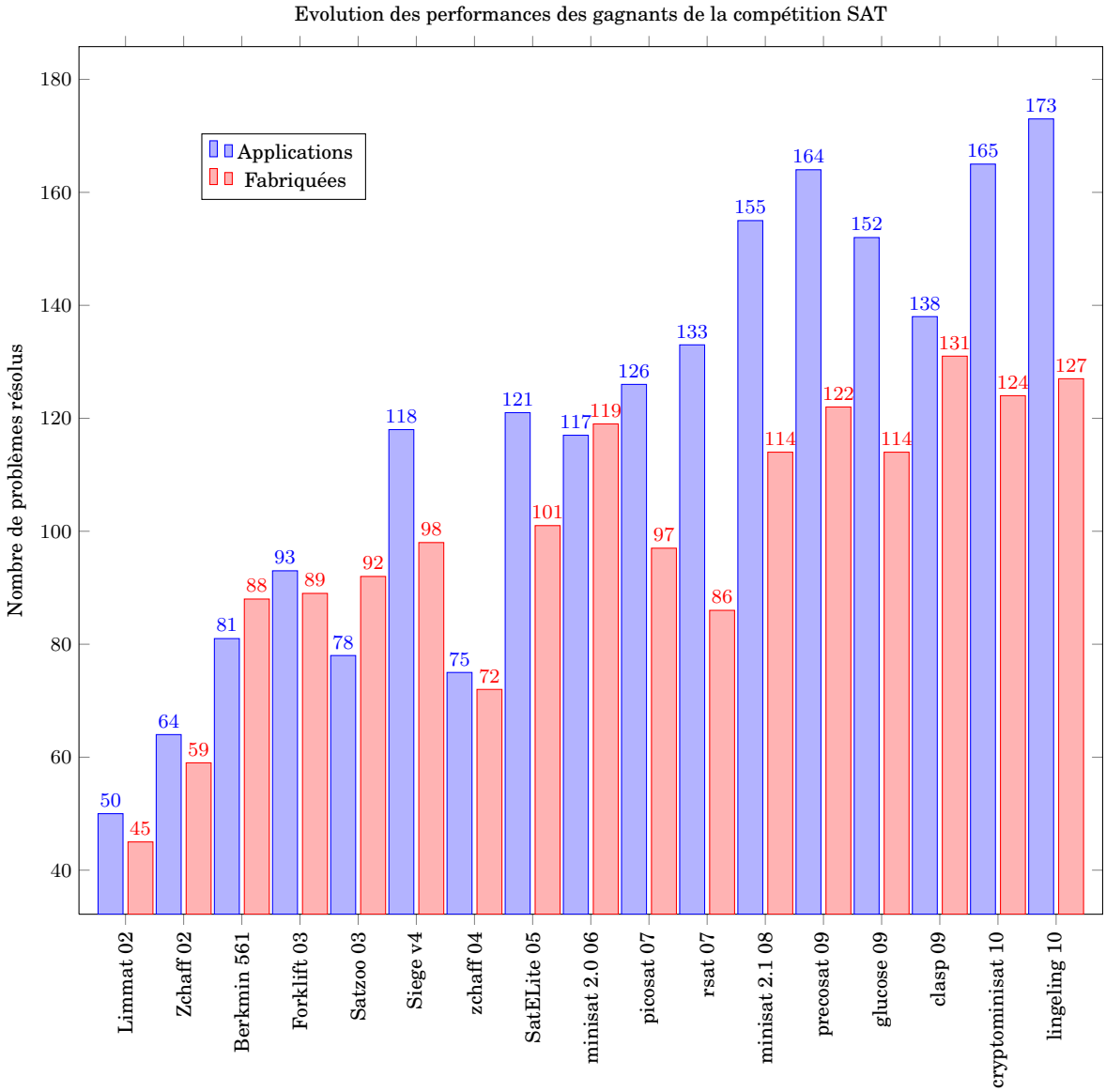
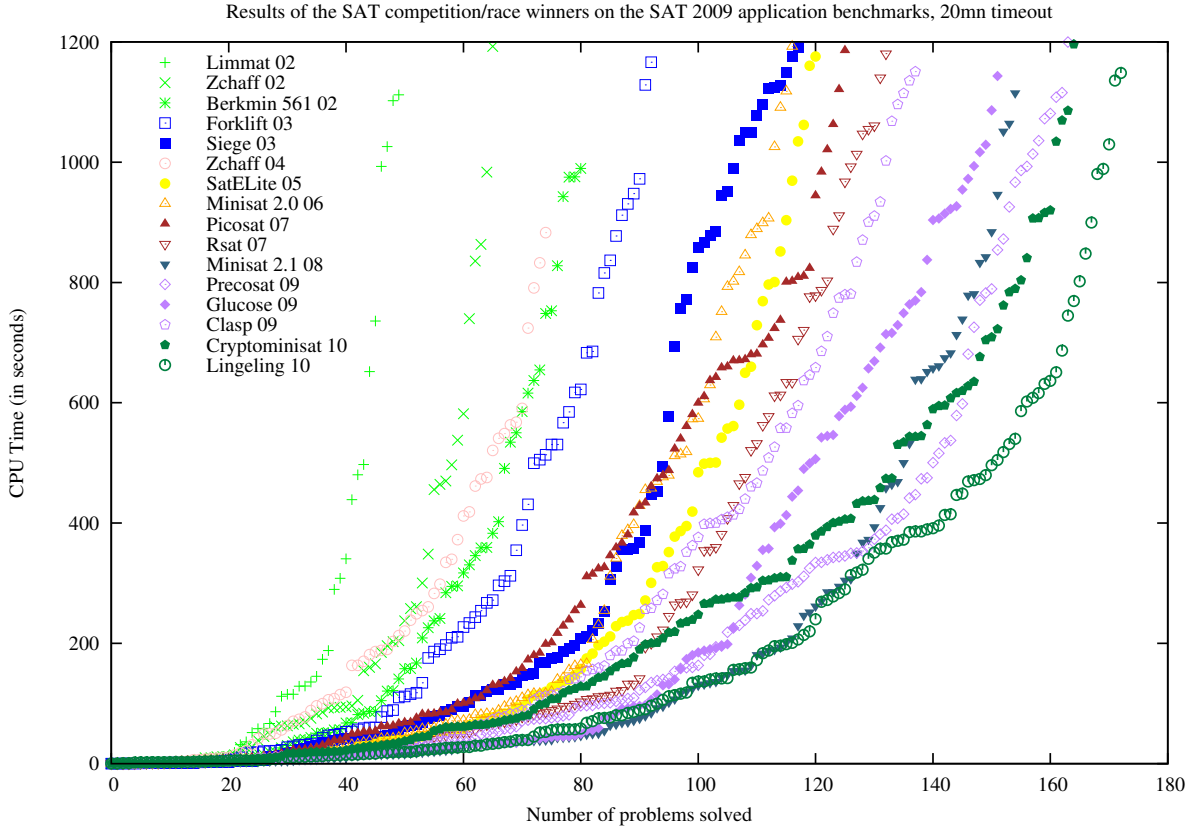
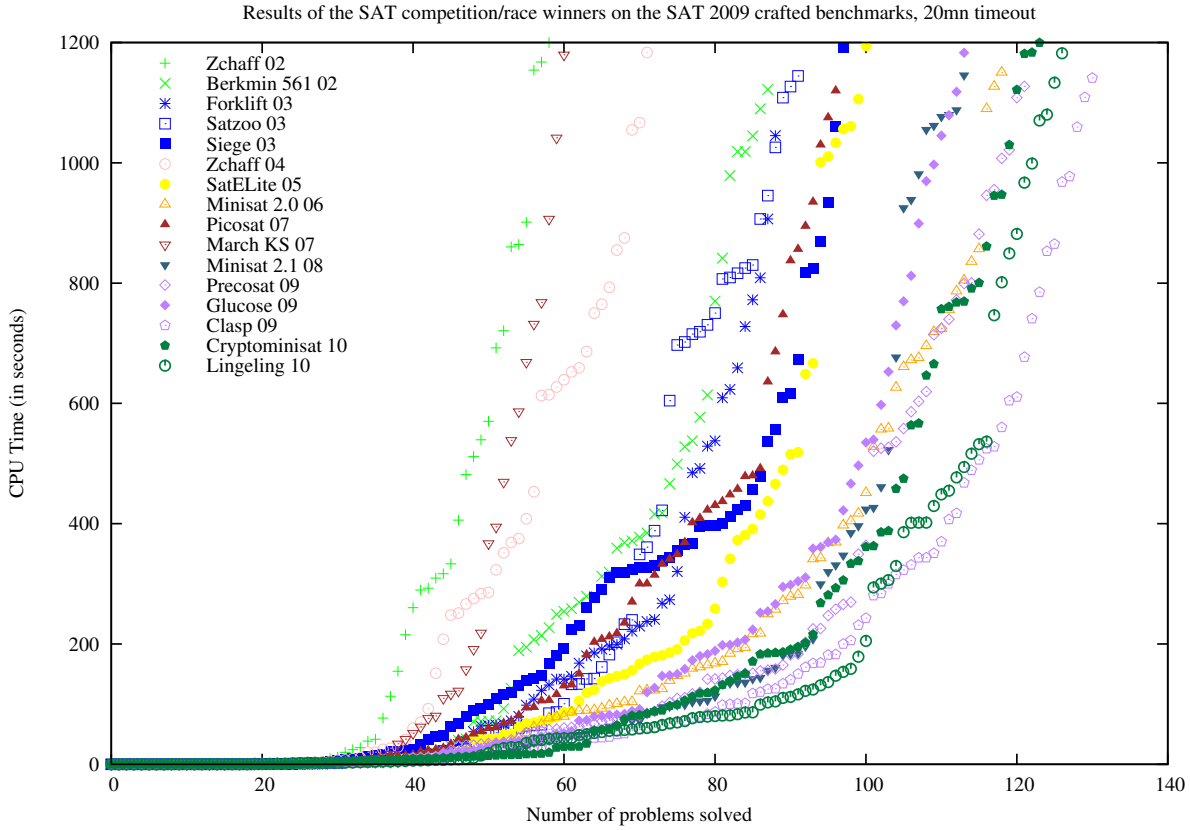


FIGURE 5.4 – Evolution des performances des prouveurs SAT ayant gagné la première phase de la compétition SAT dans la catégorie application, et quelques autres, dans les conditions de la première phase de la compétition SAT 2009.

conception.


Results of the SAT competition/race winners on the SAT 2009 application benchmarks, 20mn timeout
CPU Time (in seconds)
Number of problems solved
Limmat 02
Zchaff 02
Berkmin 561 02
Forklift 03
Siege 03
Zchaff 04
SatELite 05
Minisat 2.0 06
Picosat 07
Rsat 07
Minisat 2.1 08
Precosat 09
Glucose 09
Clasp 09
Cryptominisat 10
Lingeling 10
0
200
400
600
800
1000
1200
0
20
40
60
80
100
120
140
160
180


Results of the SAT competition/race winners on the SAT 2009 crafted benchmarks, 20mn timeout
CPU Time (in seconds)
Number of problems solved
Zchaff 02
Berkmin 561 02
Forklift 03
Satzoo 03
Siege 03
Zchaff 04
SatELite 05
Minisat 2.0 06
Picosat 07
March KS 07
Minisat 2.1 08
Precosat 09
Glucose 09
Clasp 09
Cryptominisat 10
Lingeling 10
0
200
400
600
800
1000
1200
0
20
40
60
80
100
120
140

Chapitre 6

Conclusion, pistes de recherche

Durant ces dix dernières années, nos travaux de recherche se sont focalisés sur deux voies, qui suivent les deux axes de recherche du CRIL.

La première, plus théorique, concerne la représentation et l'inférence en présence de préférences [43, 28], l'agrégation de préférences [149], la recherche de sous-classes traitables pour le formules booléennes quantifiées (QBF) [58], ou plus généralement les problèmes de représentation des certificats (politiques) pour QBF [54, 55]. Ce travaux ont donné lieu à des publications dans les revues ou conférences généralistes sélectives : Artificial Intelligence Journal [43, 28], IJCAI [27], AAAI [57], KR [42, 55].

La seconde, plus pratique, concerne la conception et l'évaluation de prouveurs pour divers formalismes autour de la logique propositionnelle : SAT [109, 158, 147, 115, 116, 117], contraintes pseudo-booléenes [112], MaxSat [113], QBF [119, 111, 56]. Plus récemment, nous nous sommes intéressés au problème de gestion des dépendances entre logiciels [114, 12]. Ce travaux ont donné lieu à des publications dans les revues, conférences ou workshops spécialisés. Le niveau des publications réalisé sur ce sujet peut sembler inférieur à celui des publications sur les travaux précédents, mais il faut prendre en considération le caractère volontairement appliqué de ceux-ci et le fait que l'outil que nous avons développé avec Anne Parrain, Sat4j, est actuellement utilisé par des millions d'utilisateurs d'Eclipse à travers le monde.

Nos travaux vont se poursuivre selon ces deux voies, dans le but de mettre en place un outil de modélisation et de raisonnement « grand public » basé sur la logique propositionnelle, regroupant les divers formalismes traités dans ce document, voire d'autres étudiés au CRIL (la logique propositionnelle épistémique par exemple). L'évolution de Sat4j d'un simple prototype de recherche en une bibliothèque utilisée en production à grande échelle nous montre qu'il est possible, à l'heure actuelle, de valoriser les travaux de recherche effectués au sein du CRIL sous forme de logiciels, notamment des logiciels libres.

Ce document est aussi l'occasion d'ouvrir quelques perspectives de recherche qui nous semblent intéressantes.

6.1 Une approche pragmatique de la résolution des QBF

Nous n'avons pas développé dans ce document nos travaux relatifs au formules booléennes quantifiées. Ils ont cependant un sens dans le contexte de la gestion des dépendances,

notamment ce qui concerne la représentation de politiques [55]. En effet, dans le cadre d'Eclipse par exemple, un ensemble d'unités installables $EclipseIUs$ est fournie par le projet, pour constituer la plate-forme Eclipse. Une société désirant construire un produit $VendorIUs$ au dessus d'Eclipse pourrait se poser la question suivante : est-ce qu'il est possible d'installer mon produit quelle que soit la configuration de la plate-forme Eclipse ?

$$\forall EclipseIUs\ \exists VendorIUs\ dependencyFormula$$

Une autre question pourrait être de déterminer s'il existe des configurations de la plate-forme qui ne permettre pas l'installation de l'application :

$$\exists EclipseIUs\ \forall VendorIUs\ \neg dependencyFormula$$

Dans les deux cas, ces formules sont des $QBF_{2,q}$ selon la terminologie que nous utilisons dans [55].

Nous avons cru après l'avénement de Chaff que les problèmes NP-complets étaient devenus (relativement) faciles en pratique, et que les nouveaux challenges se situaient dans les niveaux supérieurs de la hiérarchie polynomiale. D'où la mise en oeuvre des prouveurs OpenQBF et QBFL et notre intérêt pour les classes traitables pour QBF [58]. Les résultats peu encourageants obtenus en théorie et en pratique dans ce domaine nous conduirent à nous recentrer sur des problèmes de complexité plus proche de SAT : les problèmes d'optimisation de contraintes pseudo-booléennes, MaxSat.

Au vu des résultats encourageants obtenus sur ces problèmes, nous pensons qu'il est possible de concevoir des prouveurs QBF dédiés aux problèmes $QBF_{2,\forall}$ et $QBF_{2,\exists}$ qui pourraient avoir un intérêt pratique tout en offrant des challenges intéressants en théorie pour la représentation des politiques pour $QBF_{2,\forall}$.

6.2 Optimisation multi-critères sur variables booléennes

Nous disposons avec Sat4j d'une plate-forme d'optimisation pseudo-booléenne relativement efficace qui sert de moteur généraliste pour nos applications. Nous avons présenté dans ce document plusieurs problèmes qui pouvaient être traduits individuellement en problème d'optimisation pseudo-booléen. L'agrégation de ces traductions est nécessaire pour mettre en oeuvre un formalisme combinant les divers concepts présentés (fonction objectif, contraintes souples, disjonction ordonnée).

Qu'il s'agisse de classer des prouveurs SAT ou simplement d'agréger divers critères, on se retrouve en face d'un problème d'optimisation multi-critère. Ce domaine de recherche est vaste et amplement étudié en recherche opérationnelle et en théorie de la décision.

Nos problèmes d'optimisation multi-critères sont très particuliers, car il sont limités au cadre des variables booléennes. Les résultats obtenus dans le cadre plus général de la théorie de la décision doivent être étudiés afin de guider nos choix d'agrégation vers des solutions ayant des propriétés souhaitables.

Un premier pas vers le domaine de la théorie de la décision a été réalisé dans le cadre de l'agrégation d'ordres d'intervalle [110].

6.3 Quand l'inférence lexicographique rejoint MaxSat

Nous avions montré dans [28], article joint chapitre 9, qu'il était possible de compiler l'inférence lexicographique en affaiblissant la base de connaissance lorsqu'un conflit était détecté. Trois différentes compilations étaient proposées, toutes basées sur le même principe : remplacer un ensemble de clauses incohérent par la disjonction de ses clauses. Le principal problème de cette approche est l'explosion combinatoire résultant du calcul explicite des disjonctions.

Il est intéressant de noter que nous disposons désormais des moyens pour représenter implicitement ces disjonctions. En effet, en ajoutant une nouvelle variables pour chaque clause de l'ensemble incohérent, il est possible de représenter implicitement la disjonction des clauses à l'aide d'une contrainte de cardinalité.

Soit $S = \{a \vee b, \neg b \vee c, \neg a, \neg c\}$ un ensemble de clauses incohérent. Notre approche consistait à construire l'ensemble $S' = \{a \vee b \vee \neg c, \neg a \neg b \vee c, \neg a \vee \neg c\}$ qui représente sous forme compilée l'ensemble des sous bases cohérentes de S, c'est-à-dire que tout formule qui est conséquence logique de toutes les sous bases lexico-préférées de S sont conséquence logique de S'.

Comme nous disposons maintenant de prouveurs pseudo-booléens, on peut noter qu'il est possible de représenter S' de manière équivalente par $S'' = \{x_1 \vee a \vee b, x_2 \vee \neg b \vee c, x_2 \vee \neg a, x_4 \vee \neg c, x_1 + x_2 + x_3 + x_4 \leq 1\}$.

Si dans cet exemple l'avantage n'est pas flagrant, il l'est lorsque la taille de S augmente.

Il s'avère que cette technique est actuellement largement utilisée dans la résolution pratique de MaxSat [78, 133, 10], et qu'elle est très efficace sur les problèmes issus d'applications. Cela ouvre des perspectives intéressantes pour la réalisation de systèmes à base d'inférence lexicographique.

6.4 A propos des instances SAT « faciles »

Dans l'esprit de nombreuses personnes, les instances SAT issues de problèmes réels sont faciles, ce qui expliquerait pourquoi certains prouveurs SAT sont capables de résoudre des benchmarks contenant des dizaines de millions de variables et de clauses. Lorsque ces instances sont cohérentes, la chance peut toujours être évoquée. Lorsque les instances sont incohérentes, il existe forcément une preuve par résolution courte, car l'algorithme a été capable de résoudre cette instance en un temps très court.

Ces observations sont exactes, mais elles sont incomplètes. Tout observateur extérieur à la communauté peut facilement noter qu'il existe une dichotomie flagrante entre les prouveurs complets pour SAT. Il suffit de prendre deux instances incohérentes pour déterminer leur classe : une instances aléatoire 3-SAT au seuil de 500 variables, et une instance de vérification de modèle bornée contenant plus de 100 000 variables (une de celles proposées par Miroslav Velev par exemple).

Un prouveur de type CDCL est incapable de résoudre la première instance en 20 minutes, alors qu'un bon prouveur DPLL le peut. A contrario, un prouveur DPLL est incapable de résoudre la seconde instance alors qu'elle sera souvent triviale pour un prou-

veur CDCL. Un prouveur non hybride (i.e. pas de type portfolio par exemple) capable de résoudre ces deux instances serait remarquable.

Si tout le monde est d'accord sur ce constat, on ne sait pas à l'heure actuelle déterminer *a priori* si une instance d'un problème SAT peut facilement être résolue par un prouveur de type CDCL. Cette connaissance se construit pour l'instant de manière empirique, au fur et à mesure des expérimentations de chaque chercheur, et durant les évaluations communautaires.

On note cependant que les instances issues de problèmes contenant une structure hiérarchique, comme dans des circuits ou des programmes informatiques, semblent le terrain préféré de ces prouveurs. Le fait que la procédure d'analyse de conflits qui se trouve au coeur des prouveurs de type CDCL basée sur la notion de point unique d'implication (UIP) introduite dans GRASP [132, 156] soit inspirée de la détection d'erreurs dans les circuits électroniques (*Unique Sensitization Points* [79, 104]) n'est certainement pas une coïncidence.

Il a été récemment démontré que le système de preuve des prouveurs CDCL est comparable à la résolution générale [25, 146], alors que le système de preuve d'un prouveur de type DPLL est connu pour être basé sur la résolution en arbre (*tree resolution*) [39], un système de preuve strictement moins performant que la résolution générale. Il faut noter que ces résultats théoriques ont été obtenus après le succès rencontré en pratique par ces prouveurs, afin de justifier leurs performances.

A la lumière de ces résultats, on peut conjecturer que les instances incohérentes qui semblent faciles pour les prouveurs CDCL nécessitent en fait un système de preuve plus puissant que la résolution en arbre. Elles ne sont donc pas faciles de manière inhérentes, mais adaptées au système de preuve des prouveurs CDCL.

De plus, la première implémentation d'un prouveur CDCL, GRASP, n'a pas rencontré le succès de ses successeurs : il faudra attendre une ré-implémentation de ce concept dans un prouveur plus véloce, Chaff, pour que l'efficacité de l'approche soit validée en pratique.

On peut alors se demander pourquoi ces prouveurs sont bien moins performants que les prouveurs DPLL sur des instances k-SAT aléatoires ?

Une explication possible est que l'espace des preuves CDCL étant plus grand que celui des preuves DPLL, l'absence de « structure » empêche les prouveurs CDCL de se focaliser sur la partie pertinente de l'espace des preuves pour l'instance à résoudre.

Le paysage des prouveurs SAT en 2010 est très différent du paysage des prouveurs SAT d'il y a 10 ans : dans les années 90, ce sont les prouveurs SAT à base de recherche locale qui étaient utilisés pour résoudre des instances SAT cohérentes de grande taille. Il semblait impossible à l'époque de pouvoir résoudre ces problèmes avec des prouveurs complets. Actuellement, les prouveurs à base de recherche locale se trouvent cantonnés à la résolution d'instances aléatoires du problème k-SAT. Comment expliquer ce changement ?

Bibliographie

[1] Debian policy manual [online]. Available from : http://www.debian.org/doc/debian-policy/ [cited September 2010].

[2] Eclipse rich client platform [online]. Available from : http://wiki.eclipse.org/index.php/Rich_Client_Platform [cited September 2010].

[3] First international constraint satisfaction solver competition [online]. Available from : http://cpai.ucc.ie/05/CallForSolvers.html.

[4] Open services gateway initiative [online]. Available from : http://www.osgi.org/ [cited September 2010].

[5] Openome [online]. Available from : https://se.cs.toronto.edu/trac/ome/wiki [cited September 2010].

[6] Ow2 consortium [online]. Available from : http://www.ow2.org/.

[7] *2003 Design, Automation and Test in Europe Conference and Exposition (DATE 2003), 3-7 March 2003, Munich, Germany*. IEEE Computer Society, 2003.

[8] Mancoosi : managing software complexity [online]. 2008. Available from : http://www.mancoosi.org [cited September 2010].

[9] F. Aloul, A. Ramani, I. Markov, and K. Sakallah. Generic ILP versus Specialized 0-1 ILP : an update. In *Proceedings of ICCAD'02*, pages 450–457, 2002.

[10] Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. On solving maxsat through sat. In Sandra Sandri, Miquel Sànchez-Marrè, and Ulises Cortés, editors, *CCIA*, volume 202 of *Frontiers in Artificial Intelligence and Applications*, pages 284–292. IOS Press, 2009.

[11] Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. A new algorithm for weighted partial maxsat. In Maria Fox and David Poole, editors, *AAAI*. AAAI Press, 2010.

[12] Josep Argelich, Daniel Le Berre, Inês Lynce, João P. Marques Silva, and Pascal Rapicault. Solving Linux Upgradeability Problems Using Boolean Optimization. *CoRR*, abs/1007.1021, 2010.

[13] Josep Argelich, Chu Min Li, Felip Manya, and Jordi Planes. Fourth max-sat evaluation [online]. 2009. Available from : http://www.maxsat.udl.cat/09/.

[14] Josep Argelich, Chu Min Li, Felip Manya, and Jordi Planes. Fifth max-sat evaluation [online]. 2010. Available from : http://www.maxsat.udl.cat/10/.

[15] K.J. Arrow. *Social choice and individual values*. J. Wiley, New York, 1951. 2nd edition, 1963.

[16] Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Efficient generation of unsatisfiability proofs and cores in sat. In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *LPAR*, volume 5330 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 2008.

[17] Gilles Audemard, Lucas Bordeaux, Youssef Hamadi, Saïd Jabbour, and Lakhdar Saïs. A generalized framework for conflict analysis. In Hans Kleine Büning and Xishun Zhao, editors, *SAT*, volume 4996 of *Lecture Notes in Computer Science*, pages 21–27. Springer, 2008.

[18] Gilles Audemard, Daniel Le Berre, Olivier Roussel, Ines Lynce, and Joao Marques-Silva. Opensat : an open source sat software project. In *Sixth International Conference on Theory and Applications of Satisfiability Testing(SAT'03)*, 2003.

[19] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. In Boutilier [41], pages 399–404.

[20] Fahiem Bacchus and Toby Walsh, editors. *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, volume 3569 of *Lecture Notes in Computer Science*. Springer, 2005.

[21] Peter Barth. A Davis-Putnam based enumeration algorithm for linear pseudo-Boolean optimization. Technical Report MPI–I–95–2–003, Max-Plank-Institut für Informatik, Saarbrücken, 1995.

[22] Don Batory. Algebraic hierarchical equations for application design tool suite [online]. Available from : http://userweb.cs.utexas.edu/users/schwartz/ATS.html [cited September 2010].

[23] Don S. Batory. Feature models, grammars, and propositional formulas. In J. Henk Obbink and Klaus Pohl, editors, *SPLC*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer, 2005.

[24] Roberto J. Jr. Bayardo and Robert C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 203–208, Providence, Rhode Island, 1997.

[25] Paul Beame, Henry A. Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *J. Artif. Intell. Res. (JAIR)*, 22 :319–351, 2004.

[26] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz Cortés. Fama : Tooling a framework for the automated analysis of feature models. In Klaus Pohl, Patrick Heymans, Kyo Chul Kang, and Andreas Metzger, editors, *VaMoS*, volume 2007-01 of *Lero Technical Report*, pages 129–134, 2007.

[27] Salem Benferhat, Souhila Kaci, Daniel Le Berre, and Mary-Anne Williams. Weakening conflicting information for iterated revision and knowledge integration. In Bernhard Nebel, editor, *IJCAI*, pages 109–118. Morgan Kaufmann, 2001.

[28] Salem Benferhat, Souhila Kaci, Daniel Le Berre, and Mary-Anne Williams. Weakening conflicting information for iterated revision and knowledge integration. *Artif. Intell.*, 153(1-2) :339–371, 2004.

[29] B. Benhamou, L. Saïs, and P. Siegel. Two proof procedures for a cardinality based language in propositional calculus. In *11th Annual Symposium on Theoretical Aspects of Computer Science (STACS), volume 775 LNCS*, pages 71–82, Caen, France, 1994.

[30] Armin Biere. Lingeling, plingeling, picosat and precosat at sat race 2010. SAT 2010 solver description. Available from : http://baldur.iti.uka.de/sat-race-2010/descriptions/solver_1+2+3+6.pdf.

[31] Armin Biere. Picosat essentials. *JSAT*, 4(2-4) :75–97, 2008.

[32] Armin Biere. Bounded model checking. In Biere et al. [36], pages 457–481.

[33] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of bdds. In *Proceedings of Design Automation Conference (DAC'99)*, 1999.

[34] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In Rance Cleaveland, editor, *TACAS*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[35] Armin Biere and Carla P. Gomes, editors. *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, volume 4121 of *Lecture Notes in Computer Science*. Springer, 2006.

[36] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[37] Guillaume Blin, Florian Sikora, and Stéphane Vialette. GraMoFoNe : a Cytoscape plugin for querying motifs without topology in Protein-Protein Interactions networks. In Hisham Al-Mubaid, editor, *2nd International Conference on Bioinformatics and Computational Biology (BICoB'10)*, pages 38–43, Honolulu, USA, March 2010. International Society for Computers and their Applications (ISCA).

[38] Alexander Bockmayr and Friedrich Eisenbrand. Combining logic and optimization in cutting plane theory. In Hélène Kirchner and Christophe Ringeissen, editors, *FroCos*, volume 1794 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2000.

[39] Maria Luisa Bonet, Juan Luis Esteban, Nicola Galesi, and Jan Johannsen. On the relative complexity of resolution refinements and cutting planes proof systems. *SIAM J. Comput.*, 30(5) :1462–1484, 2000.

[40] Maria Luisa Bonet and Katherine St. John. Efficiently calculating evolutionary tree measures using sat. In Kullmann [106], pages 4–17.

[41] Craig Boutilier, editor. *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, 2009.

[42] Gerhard Brewka, Salem Benferhat, and Daniel Le Berre. Qualitative choice logic. In Dieter Fensel, Fausto Giunchiglia, Deborah L. McGuinness, and Mary-Anne Williams, editors, *KR*, pages 158–169. Morgan Kaufmann, 2002.

[43] Gerhard Brewka, Salem Benferhat, and Daniel Le Berre. Qualitative choice logic. *Artif. Intell.*, 157(1-2) :203–237, 2004.

[44] M. Buro and H. Kleine Büning. Report on a SAT competition. *Bulletin of the European Association for Theoretical Computer Science*, 49 :143–151, 1993.

[45] Thierry Castell, Claudette Cayrol, Michel Cayrol, and Daniel Le Berre. Using the davis and putnam procedure for an efficient computation of preferred models. In Wolfgang Wahlster, editor, *ECAI*, pages 350–354. John Wiley and Sons, Chichester, 1996.

[46] Donald Chai and Andreas Kuehlmann. A fast pseudo-boolean constraint solver. In *ACM/IEEE Design Automation Conference (DAC'03)*, pages 830–835, Anaheim, CA, 2003.

[47] Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the really hard problems are. In *IJCAI*, pages 331–340, 1991.

[48] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. Practical memory leak detection using guarded value-flow analysis. In Jeanne Ferrante and Kathryn S. McKinley, editors, *PLDI*, pages 480–491. ACM, 2007.

[49] Alessandro Cimatti, Anders Franzén, Alberto Griggio, Roberto Sebastiani, and Cristian Stenico. Satisfiability modulo the theory of costs : Foundations and applications. In Javier Esparza and Rupak Majumdar, editors, *TACAS*, volume 6015 of *Lecture Notes in Computer Science*, pages 99–113. Springer, 2010.

[50] Paul Clements and Linda Northrop. *Software Product Lines : Practices and Patterns*. Addison-Wesley Professional, 3rd edition, August 2001.

[51] Michael Codish, Samir Genaim, and Peter J. Stuckey. A declarative encoding of telecommunications feature subscription in sat. In António Porto and Francisco Javier López-Fraguas, editors, *PPDP*, pages 255–266. ACM, 2009.

[52] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third IEEE Symposium on the Foundations of Computer Science*, pages 151–158, 1971.

[53] Roberto Di Cosmo and Stefano Zacchiroli. Feature diagrams as package dependencies. In *Software Product Lines, 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 20010 Proceedings*, 2010.

[54] Sylvie Coste-Marquis, H. Fargier, J. Lang, Daniel Le Berre, and Pierre Marquis. Résolution de formules booléennes quantifiées : problèmes et algorithmes. In *13 ème congrès francophone AFRIF-AFIA de reconnaissance des Formes et Intelligence Artificielle(RFIA'02)*, pages 289–298, Angers, France, 2002.

[55] Sylvie Coste-Marquis, Hélène Fargier, Jérôme Lang, Daniel Le Berre, and Pierre Marquis. Representing policies for quantified boolean formulae. In Patrick Doherty, John Mylopoulos, and Christopher A. Welty, editors, *KR*, pages 286–297. AAAI Press, 2006.

[56] Sylvie Coste-Marquis, Daniel Le Berre, and Florian Letombe. A branching heuristics for quantified renamable horn formulas. In Bacchus and Walsh [20], pages 393–399.

[57] Sylvie Coste-Marquis, Daniel Le Berre, Florian Letombe, and Pierre Marquis. Propositional fragments for knowledge compilation and quantified boolean formulae. In Manuela M. Veloso and Subbarao Kambhampati, editors, *AAAI*, pages 288–293. AAAI Press / The MIT Press, 2005.

[58] 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.

[59] Scott Cotton. Jat : Java sat solver [online]. 2006. Available from : http://www-verimag.imag.fr/~cotton/jat/.

[60] Olivier Coudert. On Solving Covering Problems. In *Design Automation Conference*, pages 197–202, 1996.

[61] James Crawford and DingXin Wang. International competition on satisfiability testing [online]. 1996. Available from : http://www.cirl.uoregon.edu/crawford/beijing/ [cited September 2010].

[62] James M. Crawford and Larry D. Auton. Experimental results on the crossover point in random 3-sat. *Artif. Intell.*, 81(1-2) :31–57, 1996.

[63] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. In *Communications of the Association for Computing Machinery 5*, pages 394–397, 1962.

[64] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3) :201–215, July 1960.

[65] Hidde de Jong and Michel Page. Search for steady states of piecewise-linear differential equation models of genetic regulatory networks. *IEEE/ACM Trans. Comput. Biology Bioinform.*, 5(2) :208–222, 2008.

[66] Leonardo Mendonça de Moura and Nikolaj Bjørner. Bugs, moles and skeletons : Symbolic reasoning for software development. In Jürgen Giesl and Reiner Hähnle, editors, *IJCAR*, volume 6173 of *Lecture Notes in Computer Science*, pages 400–411. Springer, 2010.

[67] Greg Dennis, Felix Sheng-Ho Chang, and Daniel Jackson. Modular verification of code with sat. In Lori L. Pollock and Mauro Pezzè, editors, *ISSTA*, pages 109–120. ACM, 2006.

[68] Greg Dennis, Kuat Yessenov, and Daniel Jackson. Bounded verification of voting software. In Natarajan Shankar and Jim Woodcock, editors, *VSTTE*, volume 5295 of *Lecture Notes in Computer Science*, pages 130–145. Springer, 2008.

[69] Gilles Dequen and Olivier Dubois. kcnfs : An efficient solver for random k-sat formulae. In Giunchiglia and Tacchella [83], pages 486–501.

[70] Heidi Dixon. *Automated Pseudo-Boolean Inference within the DPLL framework*. PhD thesis, University of Oregon, 2004.

[71] Heidi E. Dixon and Matthew L. Ginsberg. Inference methods for a pseudo-boolean satisfiability solver. In *Proceedings of The Eighteenth National Conference on Artificial Intelligence (AAAI-2002)*, pages 635–640, 2002.

[72] O. Dubois, P. André, Yacine Boufkhad, and Y. Carlier. *Cliques, Coloring and Satisfiability : Second DIMACS Implementation Challenge*, chapter Sat vs. Unsat, pages 415–436. DIMACS Series in Discrete Mathematics and Theorical Computer Science. American Mathematical Society, 1996.

[73] Environment for the development and distribution of open source software (edos) fp6-ist-004312 [online]. 2005. Available from : http://www.edos-project.org.

[74] Niklas Eén Niklas Sörensson. An extensible sat-solver. In *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing, LNCS 2919*, pages 502–518, 2003.

[75] Niklas Eén and Armin Biere. Effective preprocessing in sat through variable and clause elimination. In Bacchus and Walsh [20], pages 61–75.

[76] Niklas Eén and Niklas Sörensson. Temporal induction by incremental sat solving. *Electr. Notes Theor. Comput. Sci.*, 89(4), 2003.

[77] Jon W. Freeman. *Improvements to Propositional Satisfiability Search Algorithms*. PhD thesis, Departement of computer and Information science, University of Pennsylvania, Philadelphia, 1995.

[78] Zhaohui Fu and Sharad Malik. On solving the partial max-sat problem. In Biere and Gomes [35], pages 252–265.

[79] H. Fujiwara and T. Shimono. On the acceleration of test generation algorithms. *IEEE Trans. on Computers*, C-32(12) :1137–1144, Dec. 1983.

[80] Allen Van Gelder, Daniel Le Berre, Armin Biere, Oliver Kullmann, and Laurent Simon. Purse-based scoring for comparison of exponential-time programs [online]. 2005. Available from : http://www.soe.ucsc.edu/~avg/purse-poster.pdf [cited September 2010].

[81] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Automated termination proofs with aprove. In *Proceedings of the 15th International Conference on Rewriting Techniques and Applications (RTA-04), Lecture Notes in Computer Science 3091*, pages 210–220, Aachen, Germany, 2004. Available from : http://aprove.informatik.rwth-aachen.de/.

[82] Enrico Giunchiglia, Massimo Narizzano, Armando Tacchella, and Moshe Y. Vardi. Towards an efficient library for sat : a manifesto. *Electronic Notes in Discrete Mathematics*, 9 :290–310, 2001.

[83] Enrico Giunchiglia and Armando Tacchella, editors. *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*. Springer, 2004.

[84] Fausto Giunchiglia, Pavel Shvaiko, and Mikalai Yatskevich. S-match : an algorithm and an implementation of semantic matching. In Christoph Bussler, John Davies, Dieter Fensel, and Rudi Studer, editors, *ESWS*, volume 3053 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2004.

[85] Evguenii I. Goldberg and Yakov Novikov. Berkmin : A fast and robust sat-solver. In *DATE*, pages 142–149. IEEE Computer Society, 2002.

[86] Evguenii I. Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for cnf formulas. In *DATE* [7], pages 10886–10891.

[87] Carla P. Gomes, Bart Selman, and Henry A. Kautz. Boosting combinatorial search through randomization. In *AAAI/IAAI*, pages 431–437, 1998.

[88] R. Gomory. Outline of an algorithm for integer solutions to linar programs. 64 :275–278, 1958.

[89] Ana Graça, João Marques-Silva, Inês Lynce, and Arlindo L. Oliveira. Efficient haplotype inference with pseudo-boolean optimization. In Hirokazu Anai, Katsuhisa Horimoto, and Temur Kutsia, editors, *AB*, volume 4545 of *Lecture Notes in Computer Science*, pages 125–139. Springer, 2007.

[90] Armin Haken. The intractability of resolution. *Theor. Comput. Sci.*, 39 :297–308, 1985.

[91] Youssef Hamadi, Saïd Jabbour, and Lakhdar Saïs. Control-based clause sharing in parallel sat solving. In Boutilier [41], pages 499–504.

[92] Marijn Heule, Mark Dufour, Joris van Zwieten, and Hans van Maaren. March_eq : Implementing additional reasoning into an efficient look-ahead sat solver. In Hoos and Mitchell [95], pages 345–359.

[93] Marijn Heule and Hans van Maaren. March_dl : Adding adaptive heuristics and a new branching strategy. In *JSAT* [117], pages 47–59.

[94] J. N. Hooker. Generalized resolution and cutting planes. *Ann. Oper. Res.*, 12(1-4) :217–239, 1988.

[95] Holger H. Hoos and David G. Mitchell, editors. *Theory and Applications of Satisfiability Testing, 7th International Conference, SAT 2004, Vancouver, BC, Canada, May 10-13, 2004, Revised Selected Papers*, volume 3542 of *Lecture Notes in Computer Science*. Springer, 2005.

[96] Holger H Hoos and Thomas Stützle. *SAT 2000*, chapter SATLIB : An Online Resource for Research on SAT, pages 283–292. IOS Press. Available from : http://www.satlib.org/.

[97] Frank Hutter, Holger H. Hoos, and Thomas Stützle. Automatic algorithm configuration based on local search. In *AAAI*, pages 1152–1157. AAAI Press, 2007.

[98] Daniel Jackson. Alloy : A new technology for software modelling. In Joost-Pieter Katoen and Perdita Stevens, editors, *TACAS*, volume 2280 of *Lecture Notes in Computer Science*, page 20. Springer, 2002.

[99] D. S. Johnson and M. A. Trick, editors. *Second DIMACS implementation challenge : cliques, coloring and satisfiability*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1996. Available from : http://dimacs.rutgers.edu/Challenges/.

[100] Ulrich Junker. Quickxplain : Preferred explanations and relaxations for over-constrained problems. In Deborah L. McGuinness and George Ferguson, editors, *AAAI*, pages 167–172. AAAI Press / The MIT Press, 2004.

[101] Richard M. Karp. *Complexity of Computer Computations (Symposium Proceedings)*, chapter Reducibility Among Combinatorial Problems. Plenum Press, 1972.

[102] Henry A. Kautz and Bart Selman. Unifying sat-based and graph-based planning. In Thomas Dean, editor, *IJCAI*, pages 318–325. Morgan Kaufmann, 1999.

[103] J.G. Kemeny. Mathematics without numbers. *Daedalus*, 88 :575–591, 1959.

[104] T. Kirkland and M. R. Mercer. A topological search algorithm for atpg. In *DAC '87 : Proceedings of the 24th ACM/IEEE Design Automation Conference*, pages 502–508, New York, NY, USA, 1987. ACM.

[105] Oliver Kullmann. The sat 2005 solver competition on random instances. In *JSAT* [117], pages 61–102.

[106] Oliver Kullmann, editor. *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*. Springer, 2009.

[107] Christian Kästner, Thomas Thüm, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. Featureide : A tool framework for feature-oriented software development. In *ICSE*, pages 611–614. IEEE, 2009.

[108] Daniel Le Berre. Ads : a unified computational framework for some consistency and abductive based propsositional reasoning. In *2nd Australasian Workshop on Computational Logic(AWCL'01)*, Gold Coast (Australia), 2001.

[109] Daniel Le Berre. Exploiting the real power of unit propagation lookahead. *Electronic Notes in Discrete Mathematics*, 9 :59–80, 2001.

[110] Daniel Le Berre, Pierre Marquis, and Meltem Öztürk. Aggregating interval orders by propositional optimization. In Rossi and Tsoukiàs [149], pages 249–260.

[111] Daniel Le Berre, Massimo Narizzano, Laurent Simon, and Armando Tacchella. The second qbf solvers comparative evaluation. In Hoos and Mitchell [95], pages 376–392.

[112] Daniel Le Berre and Anne Parrain. On extending sat solvers for pb problems. In *14th RCRA workshop Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion(RCRA07)*, Rome, jul 2007.

[113] Daniel Le Berre and Anne Parrain. The sat4j library 2.2, system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7 :59–64, 2010.

[114] Daniel Le Berre and Pascal Rapicault. Dependency management for the eclipse ecosystem. In *Proceedings of IWOCE2009 - Open Component Ecosystems International Workshop*, pages 21–30, August 2009.

[115] Daniel Le Berre and Laurent Simon. The essentials of the sat 2003 competition. In Giunchiglia and Tacchella [83], pages 452–467.

[116] Daniel Le Berre and Laurent Simon. Fifty-Five Solvers in Vancouver : The SAT 2004 Competition. In Hoos and Mitchell [95], pages 321–344.

[117] Daniel Le Berre and Laurent Simon, editors. *Journal on Satisfiability, Boolean Modeling and Computation, Special Volume on the SAT 2005 competitions, evaluations*, volume 2, 2006.

[118] Daniel Le Berre, Laurent Simon, and Olivier Roussel. International sat competition 2009 [online]. 2009. Available from : http://www.satcompetition.org/2009/.

[119] Daniel Le Berre, Laurent Simon, and Armando Tacchella. Challenges in the qbf arena : the sat'03 evaluation of qbf solvers. In Giunchiglia and Tacchella [83], pages 468–485.

[120] David Lesaint, Deepak Mehta, Barry O'Sullivan, Luis Quesada, and Nic Wilson. Solving a telecommunications feature subscription configuration problem. In Peter J. Stuckey, editor, *CP*, volume 5202 of *Lecture Notes in Computer Science*, pages 67–81. Springer, 2008.

[121] David Lesaint, Deepak Mehta, Barry O'Sullivan, Luis Quesada, and Nic Wilson. Developing approaches for solving a telecommunications feature subscription problem. *Journal of Artificial Intelligence Research*, 38 :271–305, 2010.

[122] Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *IJCAI (1)*, pages 366–371, 1997.

[123] Chu Min Li and Felip Manyà. Maxsat, hard and soft constraints. In Biere et al. [36], pages 613–631.

[124] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal Speedup of Las Vegas Algorithms. *Inf. Process. Lett.*, 47(4) :173–180, 1993.

[125] Martin Lukasiewycz, Michael Glaß, Felix Reimann, and Sabine Helwig. Opt4j [online]. Available from : http://opt4j.sourceforge.net/ [cited September 2010].

[126] Yogesh S. Mahajan, Zhaohui Fu, and Sharad Malik. Zchaff2004 : An efficient sat solver. In Hoos and Mitchell [95], pages 360–375.

[127] Fabio Mancinelli, Jaap Boender, Roberto di Cosmo, Jérôme Vouillon, Berke Durak, Xavier Leroy, and Ralf Treinen. Managing the complexity of large free and open source package-based software distributions. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE06)*, pages 199–208, Tokyo, JAPAN, september 2006. IEEE Computer Society Press.

[128] Vasco Manquinho and Olivier Roussel. Pseudo-boolean competition 2010 [online]. 2010. Available from : http://www.cril.univ-artois.fr/PB10/.

[129] Vasco M. Manquinho and Olivier Roussel. The first evaluation of pseudo-boolean solvers (pb'05). In *JSAT* [117], pages 103–143.

[130] Vasco M. Manquinho and João P. Marques Silva. On solving boolean optimization with satisfiability-based algorithms. In *AMAI*, 2000.

[131] Vasco M. Manquinho and João P. Marques Silva. On using satisfiability-based pruning techniques in covering algorithms. In *DATE*, pages 356–363. IEEE Computer Society, 2000.

[132] Joao P. Marques-Silva and Karem A. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pages 220–227, November 1996.

[133] João Marques-Silva and Jordi Planes. Algorithms for maximum satisfiability using unsatisfiable cores. In *DATE*, pages 408–413. IEEE, 2008.

[134] Bertrand Mazure, Lakhdar Saïs, and Eric Grégoire. Tabu search for sat. In *Proceedings of the 14th American National Conference on Artificial Intelligence (AAAI'97)*, pages 281–285, Rhode Island, USA, jul 1997.

[135] Marcílio Mendonça, Moises Branco, and Donald D. Cowan. S.p.l.o.t. : software product lines online tools. In Shail Arora and Gary T. Leavens, editors, *OOPSLA Companion*, pages 761–762. ACM, 2009.

[136] Ilya Mironov and Lintao Zhang. Applications of sat solvers to cryptanalysis of hash functions. In Biere and Gomes [35], pages 102–115.

[137] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff : Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 530–535, 2001.

[138] Massimo Narizzano, Claudia Peschiera, Luca Pulina, and Armando Tacchella. Evaluating and certifying qbfs : A comparison of state-of-the-art tools. *AI Commun.*, 22(4) :191–210, 2009.

[139] Massimo Narizzano, Luca Pulina, and Armando Tacchella. Ranking and reputation systems in the qbf competition. In Roberto Basili and Maria Teresa Pazienza, editors, *AI*IA*, volume 4733 of *Lecture Notes in Computer Science*, pages 97–108. Springer, 2007.

[140] Jost Neigenfind, Gabor Gyetvai, Rico Basekow, Svenja Diehl, Ute Achenbach, Christiane Gebhardt, Joachim Selbig, and Birgit Kersten. Haplotype inference from unphased snp data in heterozygous polyploids based on sat. *BMC Genomics*, 9(1) :356, 2008.

[141] Mladen Nikolic. Statistical methodology for comparison of sat solvers. In Strichman and Szeider [164], pages 209–222.

[142] Richard Ostrowski, Éric Grégoire, Bertrand Mazure, and Lakhdar Saïs. Recovering and exploiting structural knowledge from cnf formulas. In Pascal Van Hentenryck, editor, *CP*, volume 2470 of *Lecture Notes in Computer Science*, pages 185–199. Springer, 2002.

[143] Martin Ouimet and Kristina Lundqvist. The tasm toolset : Specification, simulation, and formal verification of real-time systems. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 126–130. Springer, 2007.

[144] Claudia Peschiera, Luca Pulina, Armando Tacchella, Uwe Bubeck, Oliver Kullmann, and Inês Lynce. The seventh qbf solvers evaluation (qbfeval'10). In Strichman and Szeider [164], pages 237–250.

[145] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In João Marques-Silva and Karem A. Sakallah, editors, *SAT*, volume 4501 of *Lecture Notes in Computer Science*, pages 294–299. Springer, 2007.

[146] Knot Pipatsrisawat and Adnan Darwiche. On the power of clause-learning sat solvers with restarts. In Ian P. Gent, editor, *CP*, volume 5732 of *Lecture Notes in Computer Science*, pages 654–668. Springer, 2009.

[147] Paul W. Purdom, Daniel Le Berre, and Laurent Simon. A parsimony tree for the sat2002 competition. *Ann. Math. Artif. Intell.*, 43(1) :343–365, 2005.

[148] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Communications of the ACM*, 5 :23–41, 1965.

[149] Francesca Rossi and Alexis Tsoukiàs, editors. *Algorithmic Decision Theory, First International Conference, ADT 2009, Venice, Italy, October 20-23, 2009. Proceedings*, volume 5783 of *Lecture Notes in Computer Science*. Springer, 2009.

[150] Olivier Roussel and Vasco M. Manquinho. Pseudo-boolean and cardinality constraints. In Biere et al. [36], pages 695–733.

[151] Lawrence Ryan. Efficient algorithms for clause learning sat solvers. Master's thesis, SFU, February 2004. Available from : http://www.cs.sfu.ca/~mitchell/papers/ryan-thesis.ps.

[152] Sean Safarpour, Hratch Mangassarian, Andreas G. Veneris, Mark H. Liffiton, and Karem A. Sakallah. Improved design debugging using maximum satisfiability. In *FMCAD*, pages 13–19. IEEE Computer Society, 2007.

[153] Bart Selman, Henry A. Kautz, and Bram Cohen. Noise strategies for improving local search. In *AAAI*, pages 337–343, 1994.

[154] Bart Selman, Hector J. Levesque, and David G. Mitchell. A new method for solving hard satisfiability problems. In *AAAI*, pages 440–446, 1992.

[155] Hossein M. Sheini and Karem A. Sakallah. Pueblo : A Hybrid Pseudo-Boolean SAT Solver. In *JSAT* [117], pages 165–182.

[156] João P. Marques Silva and Karem A. Sakallah. GRASP : A Search Algorithm for Propositional Satisfiability. *IEEE Trans. Computers*, 48(5) :506–521, 1999.

[157] Laurent Simon and Philippe Chatalic. SatEx : A Web-based Framework for SAT Experimentation. *Electronic Notes in Discrete Mathematics*, 9 :129–149, 2001.

[158] Laurent Simon, Daniel Le Berre, and Edward A. Hirsch. The sat2002 competition. *Ann. Math. Artif. Intell.*, 43(1) :307–342, 2005.

[159] Carsten Sinz. Sat race 2006 [online]. 2006. Available from : http://fmv.jku.at/sat-race-2006/.

[160] Niklas Eén Niklas Sörensson. Translating pseudo-boolean constraints into sat. In *JSAT* [117], pages 1–26.

[161] Ewald Speckenmeyer, Chu Min Li, Vasco Manquinho, and Armando Tacchella, editors. *Journal on Satisfiability, Boolean Modeling and Computation, Special Volume on the SAT 2007 competitions*, volume 4, 2008.

[162] Ivor Spence. tts : A sat-solver for small, difficult instances. *JSAT*, 4(2-4) :173–190, 2008.

[163] Ivor Spence. sgen1 : A generator of small but difficult satisfiability benchmarks. *ACM Journal of Experimental Algorithmics*, 15, 2010.

[164] Ofer Strichman and Stefan Szeider, editors. *Theory and Applications of Satisfiability Testing - SAT 2010, 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6175 of *Lecture Notes in Computer Science*. Springer, 2010.

[165] Geoff Sutcliff and Christian Suttner. Evaluating general purpose automated theorem proving systems. *Artificial Intelligence*, 131 :39–54, 2001.

[166] Tommi Syrjänen. A rule-based formal model for software configuration. Master's thesis, Helsinki University of Technology, 1999.

[167] Niklas Sörensson and Armin Biere. Minimizing learned clauses. In Kullmann [106], pages 237–243.

[168] WP2 Team. Report on formal management of software dependencies. Technical report, Environment for the Development and Distribution of Open Source Software (EDOS) FP6-IST-004312, 2005.

[169] Ralf Treinen and Stefano Zacchiroli. Common Upgradeability Description Format (CUDF) 2.0. Technical report, Mancoosi European Project, 2010.

[170] Chris Tucker, David Shuffelton, Ranjit Jhala, and Sorin Lerner. Opium : Optimal package install/uninstall manager. In *ICSE*, pages 178–188. IEEE Computer Society, 2007.

[171] Miroslav N. Velev and Randal E. Bryant. Effective use of boolean satisfiability procedures in the formal verification of superscalar and vliw microprocessors. In *DAC*, pages 226–231. ACM, 2001.

[172] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Satzilla : Portfolio-based algorithm selection for sat. *J. Artif. Intell. Res. (JAIR)*, 32 :565–606, 2008.

[173] Emmanuel Zarpas. Back to the sat05 competition : an a posteriori analysis of solver performance on industrial benchmarks. In *JSAT* [117], pages 229–237.

[174] Hantao Zhang. SATO : an efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction (CADE'97), volume 1249 of LNAI*, pages 272–275, 1997.

[175] Hantao Zhang. Combinatorial designs by sat solvers. In Biere et al. [36], pages 533–568.

[176] Lintao Zhang and Sharad Malik. Validating sat solvers using an independent resolution-based checker : Practical implementations and other applications. In *DATE* [7], pages 10880–10885.

[177] Éric Grégoire, Bertrand Mazure, and Cédric Piette. Local-search extraction of muses. *Constraints*, 12(3) :325–344, 2007.

[178] Éric Grégoire, Bertrand Mazure, and Cédric Piette. Using local search to find msses and muses. *European Journal of Operational Research*, 199(3) :640–646, 2009.

Deuxième partie

CV détaillé

CRIL – CNRS UMR 8188
Université d'Artois
Faculté des sciences Jean Perrin
Rue Jean Souvraz SP18
62307 Lens Cedex
☎ *+33 (0)3 21 79 17 88*
FAX *+33 (0)3 21 79 17 70*
✉ *leberre@cril.univ-artois.fr*
http://www.cril.fr/~leberre/

Daniel Le Berre

Maître de Conférences

Statut actuel

Maître de Conférences (section CNU 27 informatique) CRIL – Université d'Artois
PEDR

Etat civil

Né le 29/04/1972 à Quimper (29), marié, trois enfants, nationalité française

Domaines de recherche

Résolution pratique de problèmes NP-difficiles, représentation des connaissances et des raisonnements, logiques et intelligence artificielle.

Parcours professionnel

sept01–présent **Maître de Conférences**, *UFR des Sciences*, Université d'Artois, Lens.

mars00–août01 **Chercheur invité**, Université de Newcastle, Australie.

sept99–fév00 **Attaché Temporaire d'Enseignement et de Recherche**, *Université Paul Sabatier*, Toulouse.

Formation

2000 **Doctorat**, *Université Paul Sabatier*, Toulouse.

1995 **DEA "Intelligence Artificielle"**, *Université Paul Sabatier*, Toulouse.

1994 **Maîtrise d'Informatique**, *Université de Bretagne Occidentale*, Brest.

1993 **Licence d'Informatique**, *Université de Bretagne Occidentale*, Brest.

Fonctions électives

2007–2010 **Membre élu du conseil d'UFR**, *Faculté des Sciences Jean Perrin*, Université d'Artois.

2003–2006 **Membre élu de CSE (sections 25-26-27)**, *Université d'Artois*.

2005–2007 **Membre élu du CA, membre du bureau et webmestre**, *Association Française pour la Programmation par Contraintes (AFPC)*.

Responsabilité de filière

2006–présent **Responsable pédagogique Master 2 "Ingéniérie Logicielle pour Internet"**, *Faculté des Sciences Jean-Perrin*, Université d'Artois.

Autres responsabilités

2007–présent **Responsable de la communication**, *CRIL*.

2010–présent **Correspondant information et communication**, *CRIL*, CNRS (DR18).

Enseignements

Enseignements en cours

Licence 1 **Algorithmique en Python**, *24h*, TP, Filière Mathématique/Informatique.
Initiation à la programmation procédurale

Master 1 **Génie Logiciel**, *32h*, CM/TD/TP, Filière Informatique.
Programmation dirigée par les tests, méthodes agiles, reconception, patrons de conception, initiation aux méthodes formelles

Master 2 **Java pour l'internet**, *40h*, CM/TD/TP, Filière Informatique.
Applet, Servlet, Java Server Pages, Java Standard Tag Library, Struts, JDBC

Master 2 **Java avancé**, *80h*, CM/TD/TP, Filière Informatique.
JEE, EJB 3.0, JPA, JPQL, Programmation orientée aspects (AspectJ), industrialisation du développement, intégration continue

Enseignements passés

Licence 1 **Découverte Informatique**, CM/TP, Filière Biologie.
Initiation à la programmation procédurale avec Maple

Licence 1 **Algorithmique en Pascal**, CM/TD/TP, Filière Mathématique/Informatique.
Initiation à la programmation procédurale

Licence 1 **Documents Numériques**, CM/TP, Tronc commun à toutes les filières.
Partie informatique du référentiel du C2I 1

Licence 2 **Programmation Objet en Java**, TP, Filière Informatique.
Initiation à la programmation objet

Licence 3 **Programmation Objet en Java**, CM/TD/TP, Filière Informatique.
Conception orientée objet, polymorphisme

Licence 3 **Analyse de données**, CM/TD/TP, Filière chimie.
Utilisation d'un tableur et d'un SGBD pour manipuler des données

Encadrement de projets de travail d'étude et de recherche

Les étudiants de Master 1 en informatique ont la possibilité d'effectuer au second semestre un stage en entreprise ou un Travail d'Etude et de Recherche (TER) au sein du CRIL. Le sujet de TER est généralement soit un projet de développement logiciel soit un sujet d'initiation à la recherche. Pratiquement chaque année depuis mon arrivée au CRIL, j'ai encadré un ou plusieurs étudiants dans le cadre d'un sujet de TER.

2002 **Réalisation d'un plugin Eclipse pour ADS**, *Carl Sanz et Julien Smagala*.

2002 **Réalisation d'une bibliothèque JSTL pour la plateforme JADE**, *Olivier Fourdrinoy*.

2003 **Réalisation d'une bibliothèque de BDD en Java**, *Gérald Duquesnoy*.

2004 **Mise à jour du site web SAT Live!**, *Marwan Youssef*.

2004 **Distribution de prouveurs SAT sur une grille via Proactive**, *Frédéric Fontaine*.

2005 **Développement de la bibliothèque SAT4J**, *Médéric Baron et Joffrey Bourgeois*.

2006 **Evolution de la bibliothèque SAT4J**, *Nizar Sahaji et Laurent Stemmer*.

2007 **Réalisation d'un plugin Eclipse pour Alloy**, *Antoine Bourré et François Blarel*.

2008 **Evolution du projet Alloy4Eclipse**, *Romuald Druelle et Lionnel Desruelles*.

2010 **Utilisation d'Xtext pour le projet Alloy4Eclipse**, *Mohamed Bouragba et Mohamed Said*.

2010 **Etude de la compilation de formule propositionnelle**, *Emmanuel Lonca*, sujet proposé avec Pierre Marquis.

Responsabilité de projets

2008 **Etude de faisabilité CNRS**, *Mécanismes d'Explication pour Eclipse*, Genuitec, Etats-Unis.
Responsable scientifique de l'étude

2006–2007 **Partenariats Hubert Curien (ex-PAI) PESSOA no11062SC**, *MUSICA*, INESC-ID, Portugal.
Co-porteur du projet avec Ines Lynce (INESC-ID)

2005 **Action Intégrée Luso Française de la Conférence des Présidents d'Université**, *Pseudo-Booléens*, INESC-ID, Portugal.
Co-porteur du projet avec Joao Marques-Silva (INESC-ID)

2003 **Action Intégrée Luso Française de la Conférence des Présidents d'Université**, *OpenSAT*, INESC-ID, Portugal.
Co-porteur du projet avec Joao Marques-Silva (INESC-ID)

2003 **Action Spécifique STIC CNRS no 83**, *ASQBF*, LRI, France.
Co-porteur de projet avec Philippe Chatalic et Laurent Simon (LRI)

Comité de programme, éditorial, etc.

2011 **First International SAT/SMT Summer School 2011**, *Stata center, MIT, Cambridge, USA*, 12–17 juin 2011.
Membre du comité d'organisation

2007–2009,2011 **RCRA workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion**.
Membre du comité de programme

2010 **Pragmatics of SAT (PoS)**, *FLoC Workshop*, Edinburgh, 10 juillet 2010.
Co-organisateur avec Allen Van Gelder

2002–2010 **International Conference on Theory and Applications of Satisfiability Testing (SAT)**.
Membre du comité de programme

2010 **Logics for Component Configuration (Lococo)**, *FLoC Workshop*, Edinburgh, 10 juillet 2010.
Membre du comité de programme

2008, 2010 **Logic and Search (LaSh)**.
Membre du comité de programme

2010 **Practical Aspects of Automated Reasoning (PAAR)**, *FLoC Workshop*, Edinburgh, 14 juillet 2010.
Membre du comité de programme

2010 **Evaluation Methods for Solvers - Quality Metrics for Solutions (EMS-QMS)**, *FLoC Workshop*, Edinburgh, 20 juillet 2010.
Membre du comité de programme

2010 **ASM, Alloy, B and Z conference (ABZ)**, Orford, Québec, Canada, 22–25 février 2010.
Membre du comité de programme

2009 **IWOCE2009 - Open Component Ecosystems**, *ESEC/FSE 2009 Workshop*, Amsterdam, Pays Bas, 24 août 2009.
Membre du comité de programme

2007– 2009 **Journées Francophones de Programmation par contraintes (JFPC)**.
Membre du comité de programme

2007 **EPIA Workshop on Search Techniques for Constraint Satisfaction**, *Portuguese Conference on Artificial Intelligence.*, 7 Décembre 2007.
Membre du comité de programme

2006 **17th European Conference on Artificial Intelligence (ECAI)**, *Riva del Garda, Italy*, 29 Août– 1er Septembre 2006.
Membre du comité de programme

2003 **6ème Rencontres des Jeunes Chercheurs en Intelligence Artificielle (RJCIA'03)**.
Membre du comité de programme

2002 **NonMonotonic Reasoning NMR'2002 Workshop**, *special session Changing and Integrating Information: From Theory to Practice*.
Membre du comité de programme

2004– **Journal on Satisfiability, Boolean Modeling and Computation**.
Membre du comité éditorial de cette revue électronique depuis sa création.

Autres responsabilités scientifiques

2009 **Expert scientifique**, *Mathematics of Information Technology and Complex Systems Network of Centres of Excellence (MITACS-NCE)*, Canada.

2006–2007 **Expert scientifique**, *Netherlands Organisation for Scientific Research (NWO)*, NWO Computer Science Open Competition, Pays-Bas.

2002–2005, 2007, 2009, 2011 **Co-organisateur**, *Compétition internationale de prouveurs SAT*.

2006,2008,2010 **Conseiller technique**, *SAT Race*.

2003,2004 **Co-organisateur**, *Evaluation QBF*.

2009 **Membre du COS (section 27) pour la chaire CNRS**, *Université d'Artois*.

2009 **Membre du COS (section 27)**, *Université d'Angers*.

Invitations scientifiques

Mars 2008 **"SAT, assessing the progress"**, *Département de la défense américain (DoD)*, Baltimore, `http://gauss.ececs.uc.edu/franco_files/dodreport.pdf`.
Invité par John Franco (Université de Cincinnati) à participer à ce workshop pour faire le point sur la maturité des technologies liées à SAT

Octobre 2009 **Summer School Verification Technology, Systems & Applications**, *INRIA Nancy and the Max Planck Insitute for Informatics Saarbrücken*, Nancy, `http://www.mpi-inf.mpg.de/VTSA09/`.
Invité par Stephan Merz (DR, Inria) à donner un cours sur SAT (2x3 heures)

2004, 2006, 2010 **Guangzhou Symposium on Satisfiability in Logic-Based Modeling**, *Institute of Logic and Cognition of Sun Yat-sen University*, Guangzhou, China.
Invité par Hans Kleine Büning et Xishun Zhao a présenter mes travaux pour promouvoir SAT comme domaine de recherche en Chine.

Co-encadrement de thèse et de master recherche

2003 **Karima Sedki**, Co-encadrement à 50% avec Salem Berferhat.
DEA

2004 **Yann Irrilo**, *Résolution de problèmes pseudo-booléen*, Co-encadrement à 25% avec Anne Parrain, Olivier Roussel et Lakdhar Saïs.
DEA

2002–2005 **Florian Letombe**, *De la validité des formules booléennes quantitiées: étude de complexité et exploitation de classes traitables au sein d'un prouveur QBF*, Soutenue le 13 décembre 2005, Co-encadrement à 33% avec Pierre Marquis et Sylvie Coste-Marquis.
Doctorat

Rapporteur de thèse

20 juin 2007 **David Benavides**, *On the Automated Analysis of Using Feature Models*, Seville, Espagne, Rapporteur Européen.

16 mars 2007 **Jordi Planes**, *Design and Implementation of Exact MAX-SAT solvers*, Lleida, Espagne, Rapporteur.

20 juillet 2006 **Vasco Manquinho**, *Algorithms for Linear Pseudo Boolean Optimization*, IST, Lisbonne, Portugal, Rapporteur.

17 août 2002 **Andrew Slater**, *Investigation into Satisfiability Search*, ANU, Canberra, Australie, Rapporteur.

Développement de logiciels et de sites communautaires scientifiques

2000– **SAT Live!**, *`http://www.satlive.org/`*.

2002– **Compétition SAT**, *`http://www.satcompetition.org/`*.

2003 **OpenSAT**, *`http://sat.inesc.pt/OpenSAT/`*.

2004– **SAT4J**, *`http://www.sat4j.org/`*.

2005–2007 **AFPC**, *`http://www.afpc-asso.org`*.

Publications (2000–2010)

Edition de numéro de revue

Daniel Le Berre and Laurent Simon, editors. *Special Volume on the SAT 2005 competitions and evaluations*, volume 2 of *Journal on Satisfiability, Boolean Modeling and Computation*. IOS Press, March 2006.

Revues d'audience internationale avec comité de lecture

Sylvie Coste-Marquis, Daniel Le Berre, Florian Letombe, and Pierre Marquis. Complexity results for quantified boolean formulae based on complete propositional languages. *Journal on Satisfiability, Boolean Modeling and Computation*, 1:61–88, 2006.

Paul W. Purdom, Daniel Le Berre, and Laurent Simon. A parsimony tree for the sat2002 competition. *Ann. Math. Artif. Intell.*, 43(1):343–365, 2005.

Laurent Simon, Daniel Le Berre, and Edward A. Hirsch. The sat2002 competition. *Ann. Math. Artif. Intell.*, 43(1):307–342, 2005.

Gerhard Brewka, Salem Benferhat, and Daniel Le Berre. Qualitative choice logic. *Artif. Intell.*, 157(1-2):203–237, 2004.

Salem Benferhat, Souhila Kaci, Daniel Le Berre, and Mary-Anne Williams. Weakening conflicting information for iterated revision and knowledge integration. *Artif. Intell.*, 153(1-2):339–371, 2004.

Conférences internationales avec comité de lecture et actes

Daniel Le Berre, Pierre Marquis, and Meltem Öztürk. Aggregating interval orders by propositional optimization. In Francesca Rossi and Alexis Tsoukias, editors. *Algorithmic Decision Theory, First International Conference, ADT 2009, Venice, Italy, October 20-23, 2009. Proceedings*, volume 5783 of *Lecture Notes in Computer Science*. pages 249–260. Springer, 2009.

Salem Benferhat, Daniel Le Berre, and Karima Sedki. Handling qualitative preferences using normal form functions. In David Wilson and Geoff Sutcliffe, editors. *Proceedings of the Twentieth International Florida Artificial Intelligence Research Society Conference, May 7-9, 2007, Key West, Florida, USA*. pages 38–43. AAAI Press, 2007.

Salem Benferhat, Daniel Le Berre, and Karima Sedki. An alternative inference for qualitative choice logic. In Gerhard Brewka, Silvia Coradeschi, Anna Perini, and Paolo Traverso, editors. *ECAI 2006, 17th European Conference on Artificial Intelligence, August 29 - September 1, 2006, Riva del Garda, Italy, Including Prestigious Applications of Intelligent Systems (PAIS 2006), Proceedings*, pages 741–742. IOS Press, 2006.

Sylvie Coste-Marquis, Hélène Fargier, Jérôme Lang, Daniel Le Berre, and Pierre Marquis. Representing policies for quantified boolean formulae. In Patrick Doherty, John Mylopoulos, and Christopher A. Welty, editors. *Proceedings, Tenth International Conference on Principles of Knowledge Representation and Reasoning, Lake District of the United Kingdom, June 2-5, 2006*, pages 286–297. AAAI Press, 2006.

Sylvie Coste-Marquis, Daniel Le Berre, and Florian Letombe. A branching heuristics for quantified renamable horn formulas. In Fahiem Bacchus and Toby Walsh, editors. *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, volume 3569 of *Lecture Notes in Computer Science*, pages 393–399. Springer, 2005.

Sylvie Coste-Marquis, Daniel Le Berre, Florian Letombe, and Pierre Marquis. Propositional fragments for knowledge compilation and quantified boolean formulae. In Manuela M. Veloso and Subbarao Kambhampati, editors. *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, pages 288–293. AAAI Press / The MIT Press, 2005.

Daniel Le Berre, Massimo Narizzano, Laurent Simon, and Armando Tacchella. The second qbf solvers comparative evaluation. In Hoos and Mitchell [SAT04], pages 376–392.

Daniel Le Berre and Laurent Simon. Fifty-five solvers in vancouver: The sat 2004 competition. In Hoos and Mitchell [SAT04], pages 321–344.

[SAT04] Holger H. Hoos and David G. Mitchell, editors. *Theory and Applications of Satisfiability Testing, 7th International Conference, SAT 2004, Vancouver, BC, Canada, May 10-13, 2004, Revised Selected Papers*, volume 3542 of *Lecture Notes in Computer Science*. Springer, 2005.

Daniel Le Berre and Laurent Simon. The essentials of the sat 2003 competition. In Giunchiglia and Tacchella [SAT03], pages 452–467.

Daniel Le Berre, Laurent Simon, and Armando Tacchella. Challenges in the qbf arena: the sat'03 evaluation of qbf solvers. In Giunchiglia and Tacchella [SAT03], pages 468–485.

[SAT03] Enrico Giunchiglia and Armando Tacchella, editors. *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*. Springer, 2004.

G. Brewka and S. Benferhat and D. Le Berre Qualitative Choice Logic. *Proceedings of the 8th International Conference on Principles of Knowledge Representation and Reasoning (KR'02)*, Morgan Kaufmann Publishers, Inc., pp 158-169, Toulouse, France, Avril 2002

S. Benferhat and S. Kaci and D. Le Berre and M.-A. Williams. Weakening conflicting information for iterated revision and knowledge integration *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI'01)*, Morgan Kaufmann Publishers, Inc., vol. 1, pp 109-115, Seattle, Washington, USA, Août 2001

Daniel Le Berre. Exploiting the real power of unit propagation lookahead. *Electronic Notes in Discrete Mathematics*, 9:59–80, 2001.

Workshops internationaux avec comité de lecture

Josep Argelich, Daniel Le Berre, Ines Lynce, Joao P. Marques Silva, and Pascal Rapicault. Solving linux upgradeability problems using boolean optimization. *CoRR*, abs/1007.1021, 2010.

Daniel Le Berre and Pascal Rapicault. Dependency management for the eclipse ecosystem. In *Proceedings of IWOCE2009 - Open Component Ecosystems International Workshop*, pages 21–30, August 2009.

Daniel Le Berre and Anne Parrain. On SAT technologies for dependency management and beyond. In Steffen Thiel and Klaus Pohl, editors. *Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings. Second Volume (Workshops)*. Lero Int. Science Centre, University of Limerick, Ireland, pages 197–200, 2008.

Danie Le Berre and Anne Parrain, On extending SAT solvers for PB problems, dans 14th RCRA workshop Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion (RCRA07), juillet 2007

Daniel Le Berre. ADS: a unified computational framework for some consistency and abductive based propsositional reasoning, dans 2nd Australasian Workshop on Computational Logic (AWCL'01), 2001.

Conférence nationale avec comité de lecture et actes

Sylvie Coste-Marquis, Daniel Le Berre, Florian Letombe, and Pierre Marquis. Fragments propositionnels pour la compilation de connaissances et formules booléennes quantifiées. *Actes du 15ème congrés Francophone Reconnaissance des Formes et Intelligence Artificielle (RFIA'06)*, 2006. Actes électroniques.

Sylvie Coste-Marquis, H. Fargier, J. Lang, Daniel Le Berre, and Pierre Marquis. Résolution de formules booléennes quantifiées : problèmes et algorithmes. In *13 ème congrès francophone AFRIF-AFIA de reconnaissance des Formes et Intelligence Artificielle (RFIA'02)*, pages 289–298, Angers, France, 2002.

Troisième partie

Sélection de publications

Les publications présentées dans ce dossier ont été sélectionnées selon les critères suivants :

- Les plus citées d'après Google Scholar en Septembre 2010 ;
- De préférence les versions longues parues dans des revues ;
- Qui couvrent les divers aspects de l'activité de recherche de ces dix dernières années.

1. Daniel Le Berre and Pascal Rapicault. Dependency management for the eclipse ecosystem. In *Proceedings of IWOCE2009 - Open Component Ecosystems International Workshop*, pages 21–30, August 2009.
2. Sylvie Coste-Marquis, Daniel Le Berre, Florian Letombe, and Pierre Marquis. Complexity results for quantified boolean formulae based on complete propositional languages. *Journal on Satisfiability, Boolean Modeling and Computation*, 1 :61–88, 2006.
3. Gerhard Brewka, Salem Benferhat, and Daniel Le Berre. Qualitative choice logic. *Artif. Intell.*, 157(1-2) :203–237, 2004.
4. Salem Benferhat, Souhila Kaci, Daniel Le Berre, and Mary-Anne Williams. Weakening conflicting information for iterated revision and knowledge integration. *Artif. Intell.*, 153(1-2) :339–371, 2004.
5. Daniel Le Berre. Exploiting the real power of unit propagation lookahead. *Electronic Notes in Discrete Mathematics*, 9 :59–80, 2001.
6. Laurent Simon, Daniel Le Berre, and Edward A. Hirsch. The sat2002 competition. *Ann. Math. Artif. Intell.*, 43(1) :307–342, 2005.
7. Daniel Le Berre, Pierre Marquis, and Meltem Öztürk. Aggregating interval orders by propositional optimization. In Francesca Rossi and Alexis Tsoukias, editors. *Algorithmic Decision Theory, First International Conference, ADT 2009, Venice, Italy, October 20-23, 2009. Proceedings*, volume 5783 of *Lecture Notes in Computer Science*. pages 249–260. Springer, 2009.

Dependency Management for the Eclipse Ecosystem

Eclipse p2, metadata and resolution

Daniel Le Berre*
Univ Lille Nord de France, F-59000 Lille, France
UArtois, CRIL, F-62307 Lens, France
CNRS, UMR 8188, F-62307 Lens, France
leberre@cril.univ-artois.fr

Pascal Rapicault
IBM Rational
Ottawa, Ontario, Canada
pascal_rapicault@ca.ibm.com

ABSTRACT

One of the strength of Eclipse, the well-known open platform for software development, is its extensibility made possible by the built-in pluggability mechanisms. However those pluggability mechanisms only reveal their full potential when extensions created by others are made easy to distribute and obtain. The purpose of Eclipse p2 project is to build a platform addressing the challenges of distribution and obtention of Eclipse and its extensions, which poses the same dependency management issues than for component based systems. This paper focuses on the dependency management aspect of p2. It describes the metadata used to express dependencies, the overall functioning of our resolver and a description of our propositional constraints based encoding. To conclude we describe the challenges to address in future releases.

Categories and Subject Descriptors

D.2.7 [**Distribution, Maintenance, and Enhancement**]: Extensibility; D.2.13 [**Reusable Software**]: Reusable libraries; F.4.1 [**Mathematical Logic**]: Logic and constraint programming

General Terms

Algorithms, Design, Experimentation

Keywords

OSGi, dependency management, boolean encoding

1. INTRODUCTION

Eclipse is a very popular open platform mainly written in Java and designed from the ground up as an integration platform for software development tools but also for rich client applications [13]. As the Eclipse ecosystem becomes more and more important, the Eclipse platform itself and the vertical platforms built on Eclipse all rely on the concept of extensibility, and as such the necessity of a mechanism to acquire those extensions is primordial. To that end, almost since its inception, Eclipse featured an extension acquisition mechanism named Update Manager. However over time, as inter plug-in dependencies became more complex and expressed at a finer grain, and more versions of each component was made available, limitations were being discovered in Update Manager which were hindering the adoption and retention of Eclipse. The term "plug-in hell" was coined. It is at that time that we started to work on Eclipse p2 with the goal of building a "right-grained" provisioning platform attempting to address the challenges that Update Manager had been faced with.

One of the first challenge was heterogeneity in the set of things being deployed since over time it had become clear that most OSGi- and Eclipse-based applications needed to have a manageable way to interact with their environment (e.g JRE, Windows registry keys, etc.).

Second was the need to address in one platform the diversity of provisioning scenarios and offer a solution that would work against controlled repositories -similar to the case of linux packages managed by a specific Linux distribution- or uncontrolled repositories, would allow for fully automated solutions or user-driven ones, or would sport the delivery of extensions as well as complete products.

Finally, and the most important challenge, was to solve the "plug-in hell" that was partially rooted in the non modular way of acquiring components since Update Manager forced extensions to be installed by a special abstraction one level above the actual extension itself. The term "right-grained" provisioning is a response to this problem and indicates that p2 is not an obstruction to the granularity of what a user would want to make available or obtain.

In order to achieve this goal of "right-grained" provisioning, the efficiency, reliability and scalability of the dependency resolver was key. Having learnt from our experience of authoring the OSGi runtime resolver for Equinox, it was obvious that we would need to base our dependency analysis mechanism on proven solver techniques. Coincidentally, later that year, the work of OPIUM[16] and EDOS[12] backed up our intuition on the usability and maturity of a SAT-based approach to address the problem. The dependency problem for Eclipse is closer to the problem addressed by the follow-up to EDOS project, the Mancoosi Project[1, 15], that is the problem of updating complex open source en-

*The work on explanations detailed in section 4.4 has been supported in part by Genuitec company.

IWOCE'09, August 24, 2009, Amsterdam, The Netherlands.

vironments. Nevertheless, dependency constraints[3] studied in both cases are significantly different. In this paper we are presenting the p2 metadata and the motivation for some of these constructs, detail the implementation of our resolver and conclude by the challenges we are interested in addressing in future releases.

2. P2 METADATA

Core to the majority of installers that deal with composable systems (e.g RPM, Debian, etc.) lies a concept of metadata. One of the goals of this metadata, and the point of focus of this paper, is to capture the dependencies that exist between the components of the system and thus to find missing dependencies or to validate dependencies of a system before it is being modified. As described previously, p2 is intended to deal with more than just the typical Eclipse constructs of OSGi bundles. As such, despite the presence of dependency information in the OSGi bundles composing most of Eclipse applications, p2 abstracts dependencies from the elements being delivered in an entity called an *Installable Unit* (also referred to as *IU*). We now introduce the three kinds of installable units that p2 defines.

2.1 Anatomy of an installable unit

An installable unit, the simplest construct, has the following attributes:

An identifier A string naming the installable unit.

A version The version of the installable unit. The combination identifier and version is treated like a unique ID. We will refer to versions of an installable unit to mean a set of installable unit sharing the same identifier but a different version attribute.

A set of capabilities A capability is the way for the installable unit to expose to the rest of the world what it has to offer. This is just a namespace, a name and a version. Namespace and name are strings. The namespace is here to prevent name collision and avoid having everyone adhere to name mangling conventions.

A set of requirements A conjunction of requirements. A requirement is the way for the IU to express its needs. Requirements are satisfied by capabilities. A requirement is composed of a namespace, a name and a version range [1]. In addition to these usual concepts, a requirement can have a filter (under the form of an LDAP filter [8]) which allows for its enablement or disablement depending on the environment where the IU will be installed, and it can also be marked optional meaning that failing to satisfy the requirement does not prevent the IU from being installable. Finally there is a concept of greed discussed later in this section.

An enablement filter An enablement filter indicates in which contexts an installable unit can be installed. Here again the filter will pass or fail depending on the environment in which the IU will be installed.

[1]A version range is expressed by two version number separated by a comma, and surrounded by an angle bracket, meaning value included, or a parenthesis, meaning value excluded.

Greed	Optional	Semantics
true	false	this is a "strong" requirement.
true	true	this is a "weak" requirement.
false	true	this is a "weakest" requirement, where the match will not be brought in.
false	false	this indicates a case where the requirement has to be satisfied but the IU with this requirement wants this to be brought in by another one. Such a need will be presented in 2.3.

Table 1: Greed and optional interaction.

A singleton flag This flag, when set to true, will prevent a system to contain another version of the installable unit with the same identifier.

An update descriptor The identifier and a version range identifying predecessors to this IU. Making this relationship explicit allows to deal with IUs being renamed or avoid undesirable update paths.

An example of an Installable unit representing the SWT bundle is given in Figure 1. The few things to notice are the usage of namespace to avoid clashes between the Java packages and the IU identifier; the usage of singleton because no two versions of this bundle can be installed in the same eclipse instance; the "typing" of the IU as being a bundle (see namespace `org.eclipse.equinox.p2.type` valued to `bundle`); and the identification of the IU by providing a capability in the `org.eclipse.equinox.p2.iu` namespace.

Now, let's come back on requirements and detail the semantics of greed and optional. By default, a requirement is "strong"[2] (optional is false, greed is true). This means that the IU can only be installed if the requirement is met. If a "strong" requirement is guarded by a filter that does not pass, the requirement is ignored. When the optional flag is set to true, then a requirement becomes "weak" and it does not have to be satisfied for the IU to be installed. However any IU potentially satisfying this requirement will be considered, and a best effort will be made to satisfy the requirement.

When it comes to greed, this is a rather atypical concept that we have added to control the addition of IUs as part of the potential IUs to install in order to satisfy the user request. When the greed is true (default case, and the case for strong requirements), the IU satisfying the dependencies are added to the pool of potential candidates. However when the greed is set to false, such a requirement relies on other dependencies from its own IU or others to bring in what is necessary for its satisfaction. This is used in Figure 1 to capture the fact that even though we have an optional dependency on `org.mozilla.xpcom` we don't want to try to satisfy it eagerly. As such, this optional and non greedy requirement is weaker than a typical optional dependency. Table 1 reviews the four combinations of greed and optionality.

[2]Strong is weaker in our context than the notion of strong dependency introduced in [3]

```
id=org.eclipse.swt, version=3.5.0, singleton=true
Capabilities:
    {namespace=org.eclipse.equinox.p2.iu, name=org.eclipse.swt, version=3.5.0}
    {namespace=org.eclipse.equinox.p2.eclipse.type name=bundle version=1.0.0}
    {namespace=java.package, name=org.eclipse.swt.graphics, version=1.0.0}
    {namespace=java.package, name=org.eclipse.swt.layout, version=1.2.0}
Requirements:
    {namespace=java.package, name=org.eclipse.swt.accessibility2, range=[1.0.0,2.0.0), optional=true, filter=(&(os=linux))}
    {namespace=java.package, name=org.mozilla.xpcom, range=[1.0.0, 1.1.0), optional=true, greed=false}
Updates:
    {namespace=org.eclipse.equinox.p2.iu, name=org.eclipse.swt, range=[0.0.0, 3.5.0)}
```

Figure 1: An IU representing the SWT bundle

2.2 Installable unit patch

2.2.1 The need for patches

So far, the concept of IU is pretty much on par with what most package managers are offering. However what is interesting is the different usage we have observed of this metadata and the implication it has on the rest of the system. Indeed, most people building on top of Eclipse are delivering "products" or "subsystems" and as such they want to guarantee that their customer is getting what has been tested. Failing to do this could result in a non stable product, maintenance nightmare and unsatisfied customers. However in an ecosystem where products can be mixed and where repositories can not be used as control points [3], guaranteeing a functional system is harder. Consequently, to palliate to these possible problems, product producers are using installable units as a grouping mechanism (also referred to as *group*) serving three goals:

1. Facilitate the reusability of a set of functionality by aggregating under one group a set of installable units.
2. Capture a particular configuration of the system, and thus group under one IU an extensible element and a default implementation.
3. Lock down the dependencies on installable units being used, which limits the variability of what can be installable and thus guarantees reproducibility of an installation independently of the content of the repository.

These three goals are visible in the abridged version of the Platform group shown in Figure 2. It shows the grouping of a set of unrelated functions together to facilitate reuse; how the default setup of the help system is being delivered using the `org.eclipse.help.jetty` IU[4]; and finally the ranges expressed all show the desire to precisely control a particular version of each IU being delivered.

The counter part of the lock down which is used extensively throughout Eclipse, is that it makes the delivery of service (e.g. the replacement of a particular IU by another one) complex for the following reasons:

1. Products are often made of groups, themselves recursively composed of other groups (in the Figure 2, the Platform group includes the RCP group), which can make for a rather vast ripple on effect all throughout the system when a low level component needs to be serviced.
2. Not all groups deployed on the user's machine are in the control of the same organization. For example, someone can be running a composition of IBM and Artois University products (both including the Eclipse Platform group), but the Platform group is controlled by the Eclipse open source community. Therefore when the Platform team needs to deliver a fix to a user, it simply can not require all the referring groups to be updated.
3. Not all the dependencies onto a particular IU are known ahead of time.

2.2.2 Anatomy of an Installation Unit Patch

To palliate to these problems, p2 provides the concept of *installable unit patch* (referred to as *patch*). The simplest way to think of a patch is as a mechanism that can reach into any IU and change any requirement. A patch is a regular installable unit to which three concepts have been added:

Requirement change It represents the changes made to installable units. It is a set of requirement pairs where each pair represents a rewriting rule. In the rewriting rule, the left part captures the original requirement that needs to be replaced and the right part captures the resulting requirement. No constraints are applied to the capturing requirement or the new value, the replacement could widen the range, or narrow it, or completely change the requirement (e.g. remove a filter, change optionality, etc.). In order to provide flexibility, the capturing requirement applies on an original requirement if the ranges expressed on the two requirements intersect.

Applicability scope It identifies the installable units to which the patch should be applied. It has enough flexibility to patch all the IUs of the system where a requirement change is applicable, or target just a few IUs. The IUs to patch are identified by requirements.

Lifecycle It indicates when the installable unit patch can be applied. This can be seen as a precondition that has to be satisfied for the patch to be applied. It represents the "when" to apply the patch whereas the applicability scope represents the "what" to patch. It takes the form of a non greedy requirement.

[3]Controlled repository is the approach taken by a majority of linux distributions.

[4]The Eclipse help system allows different web-servers to be used.

```
id=org.eclipse.platform.feature.group, version=3.5.0.v2009
Capabilities:
    {namespace=org.eclipse.equinox.p2.iu, name=org.eclipse.platform.feature.group, version=3.5.0.v2009}
Requirements:
    {namespace=org.eclipse.equinox.p2.iu, name=org.eclipse.rcp.feature.group, range=[3.1.0.v2009,3.1.0.v2009]}
    {namespace=org.eclipse.equinox.p2.iu, name=org.eclipse.ant.core, range=[3.2.0.v2009,3.2.0.v2009]}
    {namespace=org.eclipse.equinox.p2.iu, name=org.eclipse.ant.ui, range=[1.0.0.v2008,1.0.0.v2008]}
    {namespace=org.eclipse.equinox.p2.iu, name=org.eclipse.help, range=[4.0.0.v2009,4.0.0.v2009]}
    {namespace=org.eclipse.equinox.p2.iu, name=org.eclipse.help.jetty, range=[4.0.0.v2009,4.0.0.v2009]}
Updates:
    {namespace=org.eclipse.equinox.p2.iu, name=org.eclipse.platform.feature.group, range=(3.4.0, 3.5.0.v2009]}
```

Figure 2: An IU representing the Plaform subsystem of Eclipse

The example in Figure 3 shows a patch replacing the IUs `org.eclipse.ant.core` and `org.eclipse.ant.ui`. This patch will only try to replace the references to these two IUs from the Platform group (see applicability scope) and will only be applied if the `my.product` IU is installed.

We have shown how installable unit patches avoid a large ripple-on change across metadata by "rewriting" requirements and also how updates to an IU could be delivered without knowing all the dependencies on it. Given the alteration possibilities, another possible use of patches is to add or remove dependencies in an installable unit you would want to reuse but had unsatisfactory dependencies.

2.3 Installable unit fragment

The third and last concept of p2 metadata is the *installable unit fragment*. This concept inspired from the OSGi fragment concept [2] aims at providing a mechanism to augment an installable unit by adding to it properties or configuration information [5]. The need for this stems from the desire to make installable units as reusable as possible and thus to extract out of an installable unit the configuration information that would otherwise bind it to a particular environment. The canonical example of this is how p2 chose to handle the delivery of OSGi start levels. Some of the OSGi bundles delivered by p2 need to be configured with an information indicating when they can be started. Because the IU for OSGi bundles can be reused in different applications with different configuration needs, they can not carry the start level information so that information is separated out into an installable unit fragment. This fragment is then attached to the IU(s) it is augmenting (referred to as *host*) at resolution time and treated as part of the host during the installation. Fragments also offer the ability to attach to several IUs and thus deliver default configuration information to all of them, thus reducing the configuration burden.

An installable unit fragment augments the concept of installable unit by adding the concept of host requirement. This identifies the host or hosts to which the fragment applies and is expressed by a set of requirements. These requirements have to be satisfied for the fragment to be installable.

What is interesting in the expression of host requirement is the importance of the concept of greed, especially when dealing with a fragment delivering default configuration information. Indeed, in those cases, even though you have a fragment that can apply to multiple IUs does not mean that you want to install all the IUs to which it applies. To indicate this, the host requirement is set to be non greedy. This is the case described in the fourth line of Table 1. It is used in Eclipse by the IU fragment responsible for the delivery of the default start level to every OSGi bundles installed in the system (see Figure 4). To perform its job, the host requirement of this fragment requires "IUs that are bundles" (`namespace=org.eclipse.equinox.p2.eclipse.type, name=bundle, range=[1.0.0], greed=false`) which matches the capabilities of the same namespace and name provided by IUs like the SWT one (see Figure 1). If the greed attribute had been set to true, then the resolver would have brought in all the IUs delivering bundles thus growing unnecessarily the set of IUs that would have been added to the pool of potential IUs for the solution, whereas what is intended is to have this IU only attach to the bundles that will be brought through other dependencies. To be pedantic, in the case of the current Eclipse release, the size of the problem would have been multiplied by a factor 10 since the Eclipse SDK is made of about 380 IUs and the Galileo repository [6] contains about 3800 IUs.

[5]These are not detailed in this paper because their content and representation is irrelevant to the discussion

3. RESOLVER OVERVIEW

This section describes the overall functioning of our resolver. However the discussion on the propositional constraints encoding and explanation support appears in the next section.

Before detailing the overall solver, it is worth mentioning how p2 manages the installed software. p2 has a concept of profile which keeps track of two key information: the list of all the Installable Units installed, and the set of *root installable units*. The root IUs are not a new kind of installable units, they are installable units that are remembered as having been explicitly asked for installation. These roots are essential for installation, uninstallation and update, since they are used as strict constraints that can't be violated, thus for example avoiding the uninstallation of an IU when installing another one.

3.1 Resolution

p2 resolution process is logically organized in 5 phases:

Change request processing Given a *change request* capturing the desire to install or uninstall an installable unit, a future root set representing the application of this request over the initial root set is produced.

Slicing For each element in the future root set, the slicing produces a transitive closure of all the IUs (referred as

[6]http://download.eclipse.org/releases/galileo

```
id=org.eclipse.ant.critical.fix,version=1.0.0
Capabilities:
    {namespace=org.eclipse.equinox.p2.iu, name=org.eclipse.ant.core.critical.fix, version=3.5.0.v2009}
Requirement Changes:
    { from={namespace=org.eclipse.equinox.p2.iu, name=org.eclipse.ant.core, range=[3.1.0, 3.4.0)},
      to={namespace=org.eclipse.equinox.p2.iu, name=org.eclipse.ant.core, range=[3.4.3,3.4.3]}}
    { from={namespace=org.eclipse.equinox.p2.iu, name=org.eclipse.ant.ui, range=[1.0.0.v2008,1.0.0.v2008]},
      to={namespace=org.eclipse.equinox.p2.iu, name=org.eclipse.ant.ui, range=[1.1.0.v2009,1.1.0.v2009]}}
Applicability Scope:
    {namespace=org.eclipse.equinox.p2.iu, name=org.eclipse.platform.feature.group, range=[3.5.0.v2009,3.5.0.v2009]}
Lifecycle:
    {namespace=org.eclipse.equinox.p2.iu, name=my.product, range=[1.0.0,1.0.0], greed=false}
Updates:
    {namespace=org.eclipse.equinox.p2.iu, name=org.eclipse.platform.feature.group, range=[0.0.0, 3.5.0.v2009)}
```

Figure 3: Example of installable unit patch, patching the Platform group.

```
id=osgi.bundle.default.startlevel, version=1.0.0
Hosts:
    { {namespace=org.eclipse.equinox.p2.eclipse.type, name=bundle, range=[1.0.0], greed=false} }
Capabilities:
    {namespace=org.eclipse.equinox.p2.iu, name=osgi.bundle.default.startlevel, version=1.0.0}
```

Figure 4: Installable unit fragment attaching to any bundle.

slice) that could potentially be part of the final solution of the resolution process by consulting all repositories also passed in. This transitive closure is done with only taking into account enough context[7] to evaluate the various filters but without worrying if any IU being added could be colliding with any others. For each IU, it looks at each requirement, queries the repositories for matches, and add the results of the query to a set of IU to process. In the case of fragment IUs, the host requirement is also treated and for patches the replacement value contained in the requirement changes are as well. If the requirement greed is set to false, or if the requirement enablement filter is not evaluating to true, the process is short-circuited and the next requirement is considered. The slicing phase reduces the dependency problem to the only IUs applicable to a given setting (the OS for instance). Another solution would have been to encode everything into constraints. Our approach here is to use the constraints solver only for non trivial matters.

Projection/encoding The goal of the projection phase is to transform all the installable units of the slice and their dependencies into a system of propositional constraints (see section 4 for details). For each IU, each requirement [8] is queried against the slice and each resulting IU is encoded. In contrast to the slicing phase, the greed flag is ignored and every requirement, except when filtered, is processed since at that point the slicing has isolated the elements that will be part of the solution. It is also during this phase that patches are applied. For this, for each IU being processed, applicable patches (matching the applicability scope) are searched for in the slice and applied. Since patches may or may not be part of the final solution but logically "modifies" the original requirements, special care has to be taken to have a variable representing the initial IU with the patch applied on it, and another one with the patch not applied on it. This is detailed in section 4.3. In addition to this transformation, this phase also fills in a data structure keeping track of every potential hosts for each fragment IU. Finally for each root the corresponding variable in the SAT encoding is set to true to capture the intent of the solution being searched. The generation of the propositional constraints completed, the optimization function is being generated.

PBO-solving The result of the projection is passed to the pseudo boolean solver SAT4J[10] which is responsible for finding an assignment.

Solution extraction From the assignment returned by the solver, a solution is extracted and the final attachment of IU fragment to their host(s) is concretized. In case of failure the solver is invoked to produce an explanation (see section 4.4).

3.2 Updating

In p2, the detection and installation of all applicable updates is not a completely automated process and no simple function in the p2 API is provided to perform this operation. Instead, the only mechanism provided is a function which given a root IU and a set of repositories returns a set of candidates IUs that are updates of the given IU[9]. This allows for selectively searching for updates or looking for an update of everything that is installed. Once all the candidates have been gathered, a subset of the matches are picked and a change request is created and passed to the resolver for validation. Because p2 is a platform, the picking responsibility is left to code outside of the scope of the core of p2. For example, the graphical interface available in the Eclipse

[7]The context can be seen as a map of key/value pair

[8]Here again, requirement host of IU fragments, replacement values in requirement changes of patches as well as applicability scope are treated.

[9]Each candidate has its update descriptor match the IU it is an update of.

SDK performs filtering and the user is only presented with the most recent version of each IU to select from. The user input obtained, a corresponding profile change request is created and passed to the resolver for validation.

The obvious drawback of this approach is that the user may have to go through several iterations of selection to find a suitable alignment of IUs to update to. However, even if p2 could provide some logic to return the most pertinent and up-to-date set of IUs using techniques like MAX-SAT [4], the results would only be relevant in case of a completely automated update mechanism, since giving the opportunity to the user to pick something from the set of updates available could invalidate the solution initially returned and thus potentially lead to similar problems.

4. CONSTRAINTS ENCODING

In the following, we describe the encoding of p2 installation problem into propositional constraints, i.e. clauses or cardinality constraints. We also provide some examples of problems generated with that encoding.

$prov(IU)$ denotes the set of capabilities provided by the installable unit IU and $req(IU)$ denotes the set of capabilities required by the installable unit IU. $alt(cap) = \{IU_k | cap \in prov(IU_k)\}$ denotes the set of IUs providing a given capability. Finally, $optReq(IU)$ denotes the optional requirements of a given IU, and $versions(IU_x)$ denote the ordered set of IUs sharing the same identifier as IU_x but having different version attribute ($IU_x \in versions(IU_x)$), from the latest to the oldest.

4.1 Basic encoding

Each requirement of the form "IU_i requires capability cap_j" is represented by a simple binary (Horn) clause

$$IU_i \rightarrow cap_j$$

So, for each IU_i the requirements are expressed by a conjunction of binary clauses

$$\bigwedge_{cap_j \in req(IU_i)} IU_i \rightarrow cap_j$$

The alternatives for a given capability is given by the clause

$$cap_i \rightarrow IU_{j_1} \vee IU_{j_2} \vee ... \vee IU_{j_n}$$

where $IU_{j_x} \in alt(cap_i)$.

Since we are only interested in the IUs to install, the above two constraints can be aggregated into a conjunction of constraints:

$$f(IU_i) = \bigwedge_{cap_j \in req(IU_i),} (IU_i \rightarrow \bigvee_{IU_x \in alt(cap_j)} IU_x) \quad (1)$$

Note that there is the specific case of $alt(cap_j) = \emptyset$ which means that IU_i cannot be installed due to missing requirements. In that case, the unit clause $\neg IU_i$ is generated.

Some installation units cannot be installed together (e.g. because of the singleton attribute set to true). This can be modeled either with a conjunction of binary negative clauses

$$\bigwedge_{versions(IU_v)=\{IU_v^1,...,IU_v^n\},1 \leq i < j \leq n} (\neg IU_{v_i} \vee \neg IU_{v_j})$$

or equivalently with a single cardinality constraint:

$$(\sum_{IU_v^x \in versions(IU_v)} IU_v^x) \leq 1 \quad (2)$$

We use the second option because our solver manages those constraints natively and because it makes the explanation support easier to implement (see 4.4 for details).

Finally, the user wants to install the installable units identified by the roots. This is modeled with unit clauses:

$$\bigwedge_{UI_i \in rootIUs} UI_i \quad (3)$$

Summing up, the constraints (1), (2) and (3) altogether form an instance of a classical NP- complete SAT problem. That encoding is basically the encoding presented in Edos[12] and Opium [16] and used more recently in OpenSuse 11[10].

4.2 Encoding of optionality

One of the specificity of p2 is the semantic of "weak" dependencies expressed using the `greed` and `optional` attributes. We do not have to worry here about greedy dependencies since there are simply ignored during the slicing stage. An IU IU_i may have optional dependencies to IU IU_j means that IU_j is not mandatory to use IU_i, so IU_i can be installed successfully if IU_j is not available. However, it is expected that p2 should favor the installation of optional packages if possible, i.e. that all optional packages that could be installed are indeed installed. In Figure 1, one can see that SWT has two optional dependencies on SWT accessibility2 and Mozilla XPCOM. The encoding of optional packages is done by creating two specific propositional variables: Abs_{cap} denotes the fact the capability cap is optional, and $Noop_{IU_i}$ is a variable to be satisfied in case none of the optional capabilities of IU_i can be installed. The first set of constraints expresses how to satisfy the required capabilities:

$$\bigwedge_{cap_j \in optReq(IU_i)} (Abs_{cap_j} \rightarrow \bigvee_{IU_x \in alt(cap_j)} IU_x) \quad (4)$$

The second set of constraints expresses that if $Noop_{IU_i}$ is true then all the abstract capability variables must be false, i.e. that $Noop_{UI_i}$ can only be set to true when none of the optional dependencies could be installed.

$$\bigwedge_{cap_j \in optReq(IU_i)} (Noop_{IU_i} \rightarrow \neg Abs_{cap_j}) \quad (5)$$

Note that such set of constraints could also have been expressed equivalently by a single pseudo boolean constraint:

$$(\sum_{cap_j \in optReq(IU_i)} Abs_{cap_j}) + n \times Noop_{IU_i} \leq n$$

where $n = |optReq(IU_i)|$.

We used the first option in our encoding since it looked easier to understand. The second option could be used in order to reduce the number of constraints used in the solver. Contrariwise to the encoding of the singleton attribute, there is no need to use a single constraint here because optionality

[10] `http://en.opensuse.org/Package_Management/Sat_Solver/Basics`

encoding constraints cannot prevent an IU to be installed, thus cannot be part of an explanation.

Finally, we express that IU_i has optional dependencies using a disjunction ending with the $Noop_{IU_i}$ variable. That way, even if none of the optional requirements can be installed, the constraint can still be satisfied by setting $Noop_{IU_i}$ to true.

$$IU_i \rightarrow \bigvee_{cap_j \in optReq(IU_i)} Abs_{cap_j} \vee Noop_{IU_i} \quad (6)$$

EXAMPLE 1. *Let's see how to encode the optional dependencies of SWT on accessibility2 and xpcom shown in Figure 1:*
$Abs_{accessibility2} \rightarrow IU^{1.0}_{accessibility2}$
$\neg Abs_{accessibility2} \vee \neg Noop_{IU_{SWT}}$
$Abs_{xpcom} \rightarrow IU^{1.1}_{xpcom}$
$\neg Abs_{xpcom} \vee \neg Noop_{SWT}$
$IU_{SWT} \rightarrow Abs_{acessibility2} \vee Abs_{xpcom} \vee Noop_{SWT}$
An alternative encoding would be:
$Abs_{accessibility2} \rightarrow IU^{1.0}_{accessibility2}$
$Abs_{xpcom} \rightarrow IU^{1.1}_{xpcom}$
$Abs_{accessibility2} + Abs_{xpcom} + 2 \times Noop_{IU_{SWT}} \leq 2$
$IU_{SWT} \rightarrow Abs_{acessibility2} \vee Abs_{xpcom} \vee Noop_{SWT}$

That encoding of optionality was the original one that shipped with Eclipse 3.4. The encoding has evolved since then. This will be discussed in section 4.5.

4.3 Encoding of patches

Applying a patch from the encoding point of view only applies to requirements changes (see section 2.2), i.e. it means to enable or disable some dependencies according to the application or not of a given patch. We denote by $patchedReqs(IU, p)$ the set of pairs $< old, new >$ of the installable unit IU denoting the rewriting rules of patch p in the requirements of IUs.

We associate to each patch a new propositional variable. We introduce that variable in dependency constraints (1) and (4) the following way:

- Negatively when the patch introduces a new dependency.
- Positively when the patch removes an existing dependency.

It can be summarize that way:

$$\bigwedge_{<old,new> \in patchedReqs(IU,p)} (p \rightarrow encode(new)) \wedge (encode(old) \vee p)$$

where $encode(x)$ denote the encoding of a regular or an optional dependency. The patch encoding changes only the encoding of the requirements affected by a patch.

EXAMPLE 2. *The patch shown in Figure 3 would lead to the following encoding of the dependencies for the IU Platform:*

$IU_{Platform} \rightarrow IU^{3.1.0}_{ant-core} \vee patch$	*(a)*
$patch \wedge IU_{Platform} \rightarrow IU^{3.4.3}_{ant-core}$	*(b)*
$IU_{Platform} \rightarrow IU^{1.0.0}_{ant-ui} \vee patch$	*(a)*
$patch \wedge IU_{Platform} \rightarrow IU^{1.1.0}_{ant-ui}$	*(b)*
$IU_{Platform} \rightarrow IU^{4.0}_{help}$	*(c)*
$IU_{Platform} \rightarrow IU^{4.0}_{jetty}$	*(c)*
$IU_{Platform} \rightarrow IU^{3.1}_{rcp}$	*(c)*

If the patch is enabled (i.e. the propositional variable patch is set to true) then the initial constraints (a) are disabled while the new dependencies (b) are enabled. If the patch is not enabled (i.e. the propositional variable patch is set to false), then the new dependencies (b) are disabled and the original optional dependencies (a) apply. Note that the requirements untouched by the patch (c) do not see their encoding changed.

4.4 When things go wrong: explanation

Explanation is key to help the user understands why a change request cannot be fulfilled. In the above encoding, one can note that there are only two reasons that could prevent a request to succeed:

- At least one of the required IUs is missing.
- The request requires two IUs sharing the same identifier but with different versions that cannot be installed together due to the singleton attribute on at least one of those IUs (see for instance Figure 5).

As a consequence, it is not hard to check why a request cannot be completed. However users expect the explanation to be returned in terms of IUs they know about, the root IUs and the IUs that they are trying to install, and would be confused if provided with just the low level dependency errors. In practice, it means that knowing why a problem occurred is not sufficient. It is important to be able to detail the whole dependencies from the root to the actual cause of the problem. For instance, in Figure 5, the user cannot install the Eclipse Platform IU version 3.5.0.20090528 because the Eclipse SDK version 3.5.0.20090430 is already installed and both of them rely on different versions of Eclipse RCP that is a singleton.

Let S be the set of the constraints encoding presented in the previous sections. From a logical point of view, it is possible to compute one minimal subset S' of the constraints that cannot be satisfied altogether: $S' \subseteq S, S' \models \bot, \nexists S'' \subset S' | S'' \models \bot$. Such set of constraints is often called a MUS (minimal unsatisfiable subformula). S' is an explanation of the impossibility to fulfill the request. If the subset contains a negated literal (specific case of Equation (1), $\neg UI_x \in S'$) then the global explanation is a missing requirement, i.e. the request cannot be completed because IU_x cannot be found in the user's repositories. If the subset contains a cardinality constraints ($\sum IU^x_v \leq 1 \in S'$), then the global explanation is a singleton attribute violation, i.e. the request cannot be completed because it requires several versions of IU_v. Note that if we decided to use a clausal encoding instead of the cardinality constraints encoding, we would have lost the one to one mapping between the original dependencies and the constraints of our encoding.

There are several ways in practice to compute S' from S. The ones based on local search algorithms[14] detect constraints that are likely to be part of S' among the most falsified ones during the search and compute S' in a second step using a complete SAT solver. A more recent and widely used approach is based on the analysis of the last conflict found by a conflict driven SAT solver[17]. Such approach requires some changes in the SAT solver to keep track of all resolutions steps and does not ensure that the computed subset $S'' \subseteq S$ is minimal. A third approach is to rely on a new encoding of the problem into an optimization problem

using selector variables [11]: it is possible to use an optimization function on selector variables to compute a set S' of minimal size. Finally, a generic approach to explanation in constraints solvers was proposed in [9] and implemented in Ilog solver: QuickXplain. The main advantage of such approach is to be independent of the underlying solver, and to work with any kind of constraints.

Our approach inherits some ideas from all those approaches. We decided to implement the QuickXplain algorithm in our framework because it is non intrusive (does not require any change to the solver) and works perfectly with mixed constraints (clauses and cardinality constraints in our case). We use selector variables in our encoding to allow the QuickXplain algorithm to enable/disable the constraints when computing S'. Finally, the constraints given to the QuickXplain algorithm are ordered in decreasing order of their activity in the spirit of the local search approaches.

More precisely, we translate S into S'' by adding a new selector variable sel_i to each constraint in S: $S'' = \{sel_i \vee s_i | s_i \in S\}$. Let SEL denotes the set of all added selector variables. Instead of looking for an assignment satisfying S, we are looking for an assignment satisfying S'' under the assumption that all variables in SEL are set to false, $S'' \bigwedge_{sel_i \in SEL} \neg sel_i$[11]. If such assignment exists, it is an assignment satisfying S, so we are done. If it is not the case, then we use a tailored version of the QuickXplain algorithm that makes use of the selector variables to enable/disable constraints in order to compute S'.

4.5 From decision to optimisation

When all the constraints can be satisfied, there are usually many possible solutions, that are not of equal quality for the end user. Here are a few remarks regarding the quality of the expected solution:

1. An IU should not be installed if there is no dependency to it.
2. If several versions of the same bundle exist, the latest one should preferably be used.
3. When optional requirements exist, the optional requirements should be satisfied as much as possible.
4. User installed patches should be applied independently of the consequences of its application (i.e. the version of the IUs forced, the number of installable optional dependencies, etc.).

We are now looking for the "best" solution, not just any solution, i.e. we moved from providing a certificate for the answer to a decision problem (NP-complete from a complexity theory point of view) to return the solution of an optimization problem (NP-hard). Furthermore, we need to solve a multi-criteria optimization problem since it is likely that several IUs do have optional requirements and that several IUs are available in multiple versions.

To solve our problem, we build a linear optimization function to minimize in which the propositional variables are either given a penalty (positive integer) or a reward (negative integer) to prevent or favor their appearance in the computed assignment.

[11] Assumption based satisfiability testing is available in all Minisat[7] inspired solvers (including SAT4J).

- Each version of an IU gets a penalty as a power of 2 proportional to its age, the older it is the more penalized it is:

$$\sum_{IU_v^i \in versions(IU_v)} 2^i \times IU_v^i \quad (7)$$

That way, each installation of an IU raises at least a penalty equals to one, thus expressing that only required IUs should be installed.

- Each $Noop_x$ variable gets a penalty to favor the installation of optional IU.

$$\sum 2^K \times Noop_x \quad (8)$$

where K is a constant such that 2^K is greater than the maximum penalty for a version (i.e. $K > max(|versions(IU_v)|)$.

- Each Abs_x variable gets a reward to favor the installation of optional dependencies

$$\sum -2^{K+1} \times Abs_x \quad (9)$$

- Each *patch* variable gets a reward of $n \times -2^{K+3}$ if it is applicable (where n denotes the number of applicable patches), else a penalty of 2^{K+2}

$$\sum_{p_i \in applicablePatches()} n \times -2^{K+3} p_i + \sum_{p_i \notin applicablePatches()} 2^{K+2} p_i \quad (10)$$

The objective function of our optimization problem is thus to minimize (7) + (8) + (9) + (10).

The weights in (7) are not satisfactory since they do not provide a total order on the final solution. Suppose that we have two IUs IU_a and IU_b that are available in respectively 3 and 2 versions (namely IU_a^3, IU_a^2, IU_a^1 and IU_b^2, IU_b^1). The objective function for those IUs is thus

$$IU_a^3 + 2 \times IU_a^2 + 4 \times IU_a^1 + IU_b^2 + 2 \times IU_b^1$$

The best solution for such objective function if both IU_a and IU_b must be installed is obviously to install IU_a^3 and IU_b^2. However, if those two IUs cannot be installed together, the solver will answer that the best option is either to install IU_a^3 and IU_b^1 or IU_a^2 and IU_b^2.

The common approach to solve this problem is to rank each IUs in a total order, $IU_1 < IU_2 < ... < IU_m$, meaning that IU_i is more important than IU_j iff $IU_j < IU_i$. Then the coefficients of the optimization function should be generated in such a way that the sum of the coefficients of IU_j should be smaller than the smallest coefficient of IU_i. In our example, it would mean for instance to use the following optimization function:

$$IU_a^3 + 2 \times IU_a^2 + 4 \times IU_a^1 + 8 \times IU_b^2 + 16 \times IU_b^1$$

In that case, the best option is still to install IU_a^3 and IU_b^2, but the second best option is to install IU_a^2 and IU_b^2.

Unfortunately, as noted before, we are in the context of uncontrolled repositories, so there is no obvious/easy way to order the IUs in a total order, so it was decided to keep the initial solution (7) instead of ranking arbitrarily the IUs.

Another drawback of our objective function appeared recently when new IUs got added to the release repository. The part (9) of the optimization function has the undesirable effect to favor the installation of optional requirements

```
Cannot complete the install because of a conflicting dependency.
  Software being installed: org.eclipse.platform.sdk 3.5.0.I20090528
  Software currently installed: org.eclipse.sdk.ide 3.5.0.I20090430
  Only one of the following can be installed at once:
    org.eclipse.rcp.configuration_root.gtk.linux.x86 1.0.0.I20090430
    org.eclipse.rcp.configuration_root.gtk.linux.x86 1.0.0.I20090528
  Cannot satisfy dependency:
    From: org.eclipse.platform.sdk 3.5.0.I20090528-2000
    To: org.eclipse.rcp.configuration.feature.group [1.0.0.I20090528]
  Cannot satisfy dependency:
    From: org.eclipse.rcp.configuration.feature.group 1.0.0.I20090430
    To: org.eclipse.rcp.configuration_root.gtk.linux.x86 [1.0.0.I20090430]
  Cannot satisfy dependency:
    From: org.eclipse.rcp.configuration.feature.group 1.0.0.I20090528
    To: org.eclipse.rcp.configuration_root.gtk.linux.x86 [1.0.0.I20090528]
  Cannot satisfy dependency:
    From: org.eclipse.sdk.ide 3.5.0.I20090430-2300
    To: org.eclipse.rcp.configuration.feature.group [1.0.0.I20090430]
```

Figure 5: Example of explanation

of an IU even if such IU is not installed. A contextualization of the reward is necessary to fix such problem:

In the constraints our first solution to this problem has been to contextualize (4):

$$\bigwedge_{cap_j \in optReq(IU_i)} (Abs_{cap_j} \wedge IU_i \rightarrow \bigvee_{IU_x \in alt(cap_j)} IU_x) \tag{11}$$

That way, the variable Abs_{IU_i} can be forced to true by the objective function but it will not fire the installation of the dependencies if the IU_i is not to be installed. However, that solution has an unexpected behavior in the following test case: suppose that IU_a has an optional dependency on capability b provided by IU_b that has in turn an optional dependency on the capability c provided by IU_c for which we have $alt(IU_c) = \emptyset$. We would generate the following constraints:

$Abs_c \wedge IU_b \rightarrow \bot \equiv \neg Abs_c \vee \neg IU_b$ (a)
$Abs_b \wedge IU_a \rightarrow IU_b$
$IU_a \rightarrow IU_b \vee Noop_{IU_a}$

Since we would like to satisfy as many Abs_x variables as possible, the solver would force Abs_c and Abs_b to be set to *true*. As a consequence, IU_b has to be set to false to satisfy (a). This is possible because the dependency to IU_b is optional, else a strong dependency would force IU_b to be set to *true*. Such unexpected behavior is a direct consequence of changing the constraints because of a wrong behavior of the objective function. The fix should not change the constraints but the objective function itself.

In the objective function The other option is to use a non linear optimization function in our problem, i.e. to make the reward a function of both the abstract variable Abs_{cap_i} and IU_j where $cap_i \in optReq(IU_j)$:

$$\sum -2^{K+1} \times Abs_{cap_i} \times IU_j \tag{12}$$

The problem is that our solver does not propose yet an easy way to work with non linear optimization functions. A solution based on the introduction of new variables $y_k \leftrightarrow Abs_{cap_i} \times IU_j$ where y_k replaces $Abs_{cap_i} \times IU_j$ in the objective function and with the additional constraints $y_k \rightarrow Abs_{cap_i} \vee IU_j$,$Abs_{cap_i} \rightarrow y_k$,$IU_j \rightarrow y_k$ should fix that issue.

That last issue was discovered late in the release cycle of Eclipse 3.5 Galileo. Considering the fact that meeting the scenario of the issue of the first option was less likely than introducing a new issue by integrating a barely untested implementation of the second option in SAT4J, the former has been adopted for Eclipse 3.5.

5. CONCLUSION AND PERSPECTIVE

We presented Eclipse p2, a "right-grained" provisioning platform aimed at solving the diversity of provisioning requirements in a componentized world. We focused on presenting the three metadata constructs which allow the assembly of complex products from components:

- Installable unit, to capture dependencies among component at their lowest level, but also to capture lock downs and ease reusability by grouping;
- Installable unit patch, to tweak dependencies of installable units that can't be changed;
- Installable unit fragment, to "augment" an installable unit and as such allow for installable units to be as reusable as possible by staying context free from their used one deployed.

From there we presented the overall functioning of our resolver and gave a description of our SAT-based encoding that is resolved by the SAT4J Pseudo Boolean solver.
We can report that this approach that has been live for more than one year has proven to be reliable, efficient and scalable even when faced with repositories containing more than 10000 installable units and solution involving about 3000 installable units. Furthermore, a direct consequence of our work is the integration of the very same technology to manage dependencies in other Java related products, namely the upcoming major release of Maven (Maven3) and the repository manager Nexus.

That said, we are looking to further improve the metadata and the resolver to facilitate composition and to address the challenges encountered so far. For the metadata, the main changes we want to investigate are:

1. The ability for the line-up information that is usually expressed in groups to be extracted out into a new kind of installable units. Once extracted, this information would no longer be directly encoded as propositional constraints but would instead be used to drive the optimization function. We hope that such a change would facilitate the serviceability by avoiding the need to create patches, but the biggest challenge with this feature would be composability of several of these new IUs.

2. The ability to express capabilities and requirements on other domains than just version and version ranges. For example we are thinking about adding scalars to describe the amount of memory available.

3. Add negation and disjunction to improve the expressiveness of the requirements.

On the resolver, the main changes planned surround:

1. Stability of resolution to not cause updates of IUs that are not directly related to the change request being performed. The canonical example is where requirements toward an IU tolerate several possibilities and the installation of something new cause the update of the IU because a newer version is available in the repositories. We hope that the concept of line-up previously described will help there.

2. Resolution over a set of profiles, in order to perform the coordinated management of several applications meant to work together (e.g. the client needs to be in sync with the server).

3. Speed up the explanation process. Depending on the size of the set of constraints, it can take several minutes to provide the answer which is not ideal in an interactive tool. This is caused by the $n \times log(n)$ satisfiability tests used by the explanation algorithm used (QuickXplain) where n is the size of the input of the algorithm in number of constraints. n can be as big as the number of constraints in the encoding in the worst case.The idea is to ask more information to the SAT solver instead of using a purely external approach as currently in order to minimize n in the spirit of [6, 5].

6. REFERENCES

[1] Mancoosi, Managing the Complexity of the Open Source Infrastructure. http://www.mancoosi.org.

[2] OSGi Service Platform. http://www.osgi.org/Specifications.

[3] Pietro Abate, Jaap Boender, Roberto Di Cosmo, and Stefano Zacchiroli. Strong dependencies between software components. Technical Report 2, Mancoosi - Seventh Framework Programme, May 2009.

[4] Josep Argelich, Ines Lynce, and Joao Marques-Silva. On solving boolean multilevel optimization problems. In *Twenty-First International Joint Conferences on Artificial Intelligence (IJCAI)*, page to appear, Pasadena, California, USA, 2009.

[5] Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Efficient generation of unsatisfiability proofs and cores in sat. In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *LPAR*, volume 5330 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 2008.

[6] Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. A simple and flexible way of computing small unsatisfiable cores in sat modulo theories. In Joao Marques-Silva and Karem A. Sakallah, editors, *SAT*, volume 4501 of *Lecture Notes in Computer Science*, pages 334–339. Springer, 2007.

[7] Niklas Eén Niklas Sörensson. An extensible sat-solver. In *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing, LNCS 2919*, pages 502–518, 2003.

[8] IETF. http://www.ietf.org/rfc/rfc2254.txt.

[9] Ulrich Junker. Quickxplain: Preferred explanations and relaxations for over-constrained problems. In Deborah L. McGuinness and George Ferguson, editors, *AAAI*, pages 167–172. AAAI Press / The MIT Press, 2004.

[10] Daniel Le Berre and Anne Parrain. SAT4J, a SATisfiability library for java. http://www.sat4j.org.

[11] Ines Lynce and Joao P. Marques Silva. On computing minimum unsatisfiable cores. In *SAT*, 2004.

[12] Fabio Mancinelli, Jaap Boender, Roberto di Cosmo, Jérôme Vouillon, Berke Durak, Xavier Leroy, and Ralf Treinen. Managing the complexity of large free and open source package-based software distributions. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE06)*, pages 199–208, Tokyo, JAPAN, september 2006. IEEE Computer Society Press.

[13] Mark Powell (NASA) Marc Hoffmann, Gilles J. Iachelini (CSC). Eclipse on rails and rockets. http://live.eclipse.org/node/750.

[14] Bertrand MAZURE, Lakhdar SAIS, and Eric GREGOIRE. Detecting logical inconsistencies. In *Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics(AI/Math'96)*, pages 116–121, Fort Lauderdale (FL-USA), jan 1996.

[15] Ralph Treinen and Stefano Zacchiroli. Solving package dependencies : from Edos to Mancoosi. In *DebConf'8*, Argentine, 2008.

[16] Chris Tucker, David Shuffelton, Ranjit Jhala, and Sorin Lerner. Opium: Optimal package install/uninstall manager. In *ICSE*, pages 178–188. IEEE Computer Society, 2007.

[17] Lintao Zhang and Sharad Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formulas. In *Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT03)*, 2003.

Journal on Satisfiability, Boolean Modeling and Computation 1 (2006) 61–88

Complexity Results for Quantified Boolean Formulae Based on Complete Propositional Languages*

Sylvie Coste-Marquis {coste,leberre,letombe,marquis}@cril.univ-artois.fr
Daniel Le Berre
Florian Letombe
Pierre Marquis
CRIL/CNRS, Université d'Artois,
rue de l'Université — S.P. 16,
F-62307 Lens, France

Abstract

Several propositional fragments have been considered so far as target languages for knowledge compilation and used for improving computational tasks from major AI areas (like inference, diagnosis and planning); among them are the ordered binary decision diagrams, prime implicates, prime implicants, "formulae" in decomposable negation normal form. On the other hand, the validity problem VAL($\mathtt{QPROP}_{PS}$) for Quantified Boolean Formulae (QBF) has been acknowledged for the past few years as an important issue for AI, and many solvers have been designed. In this paper, the complexity of restrictions of the validity problem for QBF obtained by imposing the matrix of the input QBF to belong to propositional fragments used as target languages for compilation, is identified. It turns out that this problem remains hard (PSPACE-complete) even under severe restrictions on the matrix of the input. Nevertheless some tractable restrictions are pointed out.

KEYWORDS: *automated reasoning, quantified Boolean formulae*

Submitted September 2005; revised March 2006; published March 2006

1. Introduction

Compiling "knowledge" has been used for the past few years to improve (from the computational point of view) some basic tasks from major AI areas, like inference (both classical and nonmonotonic, see among others [1, 2, 3, 4, 5, 6, 7]), diagnosis (see e.g. [8, 9]) and planning (see e.g. [10, 11, 12]). These approaches typically consist in turning, during an off-line phase, some pieces of information encoded as propositional formulae into formulae from a "more tractable" fragment $\mathcal{C}$. "More tractable" means that the tasks required by the application under consideration become computationally easier, and if possible, feasible in polynomial time when the input belongs to such a fragment [13]. Such tasks usually contain deciding satisfiability (determining whether a given propositional formula has or not a model), the famous SAT problem, which is NP-complete. Among the "tractable" frag-

* A preliminary version of this paper appeared in the proceedings of the 20^{th} National Conference on Artificial Intelligence (AAAI'05), pp. 288-293.

ments considered so far are the (quite influential) ordered binary decision diagrams, prime implicates, prime implicants, and "formulae" in decomposable negation normal form.

On the other hand, VAL($\mathtt{QPROP}_{PS}$), the validity problem for QBF, has a growing importance in AI. This can be explained by the fact that, as the canonical PSPACE-complete problem, many AI problems can be polynomially reduced to VAL($\mathtt{QPROP}_{PS}$) (see e.g., [14, 15, 16, 17, 18]); in particular, VAL($\mathtt{QPROP}_{PS}$) includes SAT as a specific case; furthermore, there is some empirical evidence from various AI fields (including among others planning, nonmonotonic reasoning, paraconsistent inference) that a translation-based approach can prove more "efficient" than domain-dependent algorithms dedicated to such AI tasks. Accordingly, many solvers for VAL($\mathtt{QPROP}_{PS}$) have been designed and evaluated for the past few years (see among others [19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]).

In this paper, we consider several tractable fragments for SAT, used as target languages for knowledge compilation. For each fragment $\mathcal{C}$ under consideration, we focus on the restriction VAL($\mathtt{Q}\mathcal{C}$) of the VAL($\mathtt{QPROP}_{PS}$) problem obtained by imposing the matrix of the input formula to belong to the fragment.

A similar investigation has already been done w.r.t. some *incomplete* propositional fragments [30, 31]. Thus, in his well-known paper where a dichotomy theorem for SAT is presented [30], Schaefer also gave an analogue dichotomy theorem for VAL($\mathtt{QPROP}_{PS}$) (Theorem 6.1); roughly, this theorem shows that the only tractable classes for the restrictions of VAL($\mathtt{QPROP}_{PS}$) among those "characterized locally" (i.e., by the nature of the "clauses" from the matrix) are the Krom one (binary clauses), the Horn one, the reverse Horn one and the affine one (sets of linear equations over the field $\{0, 1\}$, or equivalently, conjunctions of XOR-clauses). Accordingly, several polytime algorithms for the restriction of VAL($\mathtt{QPROP}_{PS}$) to such incomplete fragments can be found in the literature (see [32, 33, 34]).

In the very recent past, VAL(QCSP), the validity problem for quantified constraint networks – a generalization of VAL($\mathtt{QPROP}_{PS}$) when conjunctive constraints are considered – has received much attention; classification theorems giving the complexity of VAL(QCSP) depending on algebraic properties of the constraint language under consideration have been pointed out [35, 36, 37]; such impressive results are deep insights into the study of the complexity of VAL(QCSP); nevertheless, they are concerned in essence with conjunctive constraints, so they cannot be used directly as such to identify the complexity of every restriction of VAL(QCSP) (just like Schaefer's dichotomy theorem for SAT does not characterize every tractable restriction of SAT, like the ones based on ordered binary decision diagrams or on renamable Horn CNF formulae).

In this paper, the complexity of VAL($\mathtt{Q}\mathcal{C}$) is investigated for *complete* propositional fragments $\mathcal{C}$, where a propositional fragment $\mathcal{C}$ is complete if and only if every propositional formula has an equivalent into $\mathcal{C}$. We mainly focus on fragments $\mathcal{C}$ considered in [13]: DNF, d-DNNF, sd-DNNF, DNNF, $\mathtt{OBDD}_<$, FBDD, PI, IP, MODS whose significance for many AI tasks (as well as for problems pertaining to other fields) is acknowledged. We complete the results given in [13] by focusing on an additional query, the VAL($\mathtt{Q}\mathcal{C}$) one. We draw the complexity picture for VAL($\mathtt{Q}\mathcal{C}$) for all those fragments $\mathcal{C}$. We also consider the VAL($\mathtt{Q}\mathcal{C}$) for two additional fragments: $\mathtt{OCNF}_<$and $\mathtt{ODNF}_<$.

Both tractability and intractability results have been derived. Like for the DNF fragment and its supersets including the DNNF fragment and the disjunctions of Horn CNFformulae, the VAL($\mathtt{Q}\mathcal{C}$) problem for the $\mathcal{C} = \mathtt{OBDD}_<$ fragment (and its supersets, the FBDD fragment

and the d-DNNF one) is PSPACE-complete in the general case, while in P whenever the prefix of the instance is compatible with the total, strict ordering $<$ associated with the $\mathtt{OBDD}_<$ "formula". We also consider the complete restrictions $\mathtt{ODNF}_<$ of DNF (and dually, the restrictions $\mathtt{OCNF}_<$ of CNF) and show that for those two sets of fragments $\mathcal{C}$ ($<$ varying), $\text{VAL}(\mathtt{Q}\mathcal{C})$ is in P under the compatibility assumption. Because QMODS is a subset of $\mathtt{QODNF}_<$ for which the compatibility assumption holds (and dually, the fragments QCI of quantified canonical implicates formulae is a subset of $\mathtt{QOCNF}_<$ for which the compatibility assumption holds), we get that the $\text{VAL}(\mathtt{QMODS})$ problem and the $\text{VAL}(\mathtt{QCI})$ problem are in P as well. We finally show that the $\text{VAL}(\mathtt{QPI})$ problem (resp. the $\text{VAL}(\mathtt{QIP})$ problem) is PSPACE-complete, while the complexity falls down to P for the restriction where the prefix of any instance is of the form $\forall X\ \exists Y$ (resp. $\exists X\ \forall Y$).

The rest of the paper is organized as follows: in Section 2, we give some formal preliminaries. In Section 3, the complexity results are presented. In Section 4, some experimental results are provided. In Section 5, the connection between the problem of finding out tractable matrix-based restrictions of $\text{VAL}(\mathtt{QPROP}_{PS})$ and the compilability issue for $\text{VAL}(\mathtt{QPROP}_{PS})$ when matrices are fixed while prefixes may vary is investigated. Finally, Section 6 concludes the paper and gives a few perspectives. We assume the reader familiar with classes of the polynomial hierarchy PH and the complexity class PSPACE, and with standard polynomial many-one reductions (see e.g. [38]).

2. Formal Preliminaries

In this section, we present the syntax and semantics of quantified boolean formulae; we also give a number of easy metatheorems which will prove useful in the following sections. We set the morphology of our language to the following set of connectives: $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$, $\oplus$ (XOR) plus the two boolean constants and the quantifiers $\forall$ and $\exists$; on this ground, a language for quantified boolean formulae can be defined as follows:

Definition 1 (syntax of a quantified boolean formula)**.** *Let PS be a finite set of propositional symbols. The set $\mathtt{QPROP}_{PS}$ of* quantified boolean formulae *(QBFs) over PS is the smallest set of words defined inductively as follows:*[1.]

1. *the boolean constants true, false and every variable from PS belong to $\mathtt{QPROP}_{PS}$.*
2. *if ϕ and ψ belong to $\mathtt{QPROP}_{PS}$ then $(\neg\phi)$, $(\phi \wedge \psi)$, $(\phi \vee \psi)$, $(\phi \Rightarrow \psi)$, $(\phi \Leftrightarrow \psi)$, $(\phi \oplus \psi)$ belong to $\mathtt{QPROP}_{PS}$.*
3. *if ϕ belongs to $\mathtt{QPROP}_{PS}$ and x belongs to PS, then $(\forall x.\phi)$ and $(\exists x.\phi)$ belong to $\mathtt{QPROP}_{PS}$.*
4. *every quantified propositional formula is obtained by applying the three rules above a finite number of times.*

Example 1. *The following formula is a QBF:*

$$\Sigma = \exists a.(((\forall b.(a \wedge b)) \vee ((\neg a) \vee b)) \wedge \forall b.(a \vee ((\neg b) \wedge c)))$$

Figure 1 is a graphical representation of Σ.

1. In order to simplify the syntax, we feel free to omit some parentheses when this does not question equivalence. In addition, we often abbreviate $\exists x.(\exists y.\phi)$ (resp. $\forall x.(\forall y.\phi)$) into $\exists x, y.\phi$ (resp. $\forall x, y.\phi$).

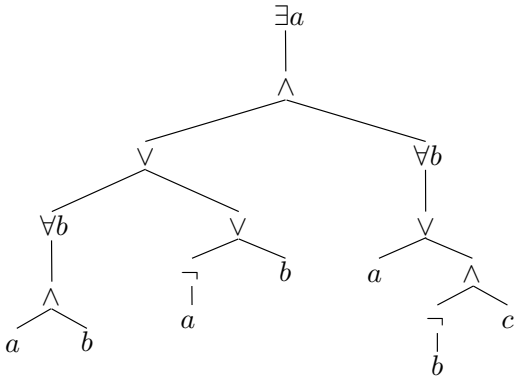


Figure 1. A quantified boolean formula.

■

The occurrences of a propositional variable x in a formula Σ from $\mathtt{QPROP}_{PS}$ can be partitioned into three sets: the *quantified* occurrences of x in Σ are those occurring in a quantification, i.e., just after a quantifier $\forall$ or $\exists$; in every subformula $\forall x.\phi$ (resp. $\exists x.\phi$) of Σ, all the occurrences of x in ϕ are *bound*, the occurrences of x are said to be *in the scope of the quantification* $\forall x$ (resp. $\exists x$). Finally, all the remaining occurrences of x in Σ are *free* ones.

$Var(\Sigma)$ is the set of all variables occurring in Σ. A variable x of Σ is *free* if it has a free occurrence in Σ.

Example 2 (cont'ed)**.** *The first occurrence of a in Σ is quantified, the second occurrence of b in Σ is bound and the third one is free. b and c are the free variables of Σ.* ■

A QBF Σ is said to be *polite* if and only if every bound occurrence of a variable x of Σ is in the scope of an unique quantification, and every free variable has no bound occurrence. A QBF Σ is said to be *prenex* if and only if $\Sigma = Qx_1 \ldots Qx_n.\phi$ where each occurrence of Q stands for either $\forall$ or $\exists$, and ϕ does not contain any quantified occurrence of a variable. ϕ is said to be the *matrix* of Σ and the sequence $Qx_1 \ldots Qx_n$ of quantifications is the *prefix* of Σ. A QBF is said to be *closed* if and only if it has no free variable. A QBF is said to be *quantifier-free* if and only if it does not contain any quantification (obviously enough, such formulae can easily be considered as "standard" propositional formulae). The subset of $\mathtt{QPROP}_{PS}$ containing only quantifier-free formulae is noted $\mathtt{PROP}_{PS}$.

Example 3 (cont'ed)**.** *Σ is neither polite, nor prenex, nor closed.* ■

Let us now consider the semantical aspects of QBF; let us start with the following useful notion of conditioning (also referred to as restriction or cofactor [39]). For every quantified boolean formula Σ and every variable x, $\Sigma_{x \leftarrow 0}$ (resp. $\Sigma_{x \leftarrow 1}$) denotes the QBF obtained by replacing every free occurrence of x in Σ by $false$ (resp. $true$). Formally:

Definition 2 (conditioning). *Let $\Sigma \in \mathtt{QPROP}_{PS}$, $x \in PS$ and $* \in \{1, 0\}$. The* conditioning *of x by $*$ in Σ is the QBF defined inductively as follows:*

$$\Sigma_{x \leftarrow *} = \begin{cases} if\ \Sigma = x\ and\ * = 1\ then\ true \\ if\ \Sigma = x\ and\ * = 0\ then\ false \\ if\ \Sigma \in PS\ and\ \Sigma \neq x\ then\ \Sigma \\ if\ \Sigma = \neg\phi\ then\ \neg(\phi_{x \leftarrow *}) \\ if\ \Sigma = \phi \circ \psi\ then\ \phi_{x \leftarrow *} \circ \psi_{x \leftarrow *}\ with\ \circ\ a\ binary\ connective \\ if\ \Sigma = \forall x.\phi\ or\ \Sigma = \exists x.\phi\ then\ \Sigma \\ if\ \Sigma = \forall y.\phi\ with\ y \neq x\ then\ \forall y.(\phi_{x \leftarrow *}) \\ if\ \Sigma = \exists y.\phi\ with\ y \neq x\ then\ \exists y.(\phi_{x \leftarrow *}) \end{cases}$$

Example 4 (cont'ed). *The conditioning of b by 1 in Σ is the QBF*

$$\Sigma_{b \leftarrow 1} = \exists a.(((\forall b.(a \wedge b)) \vee ((\neg a) \vee true)) \wedge \forall b.(a \vee ((\neg b) \wedge c)))$$

■

We are now ready to define the semantics of a QBF:

Definition 3 (semantics of a quantified boolean formula). *Let I be an interpretation over PS (i.e., a total function from PS to $BOOL = \{0, 1\}$). The* semantics *of a quantified boolean formula Σ in I is the truth value $[\![\Sigma]\!](I)$ from $BOOL$ defined inductively as follows:*

- *if $\Sigma = true$ (resp. $false$), then $[\![\Sigma]\!](I) = 1$ (resp. 0).*
- *if $\Sigma \in PS$, then $[\![\Sigma]\!](I) = I(\Sigma)$.*
- *if $\Sigma = \neg\phi$, then $[\![\Sigma]\!](I) = 1 - [\![\phi]\!](I)$.*
- *if $\Sigma = \phi \wedge \psi$, then $[\![\Sigma]\!](I) = [\![\phi]\!](I) \times [\![\psi]\!](I)$.*
- *if $\Sigma = \phi \vee \psi$, then $[\![\Sigma]\!](I) = max(\{[\![\phi]\!](I), [\![\psi]\!](I)\})$.*
- *if $\Sigma = \phi \Rightarrow \psi$, then $[\![\Sigma]\!](I) = [\![\neg\phi \vee \psi]\!](I)$.*
- *if $\Sigma = \phi \Leftrightarrow \psi$, then $[\![\Sigma]\!](I) = [\![(\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi)]\!](I)$.*
- *if $\Sigma = \phi \oplus \psi$, then $[\![\Sigma]\!](I) = [\![\neg(\phi \Leftrightarrow \psi)]\!](I)$.*
- *if $\Sigma = \forall x.\phi$, then $[\![\Sigma]\!](I) = min(\{[\![\phi_{x \leftarrow 0}]\!](I), [\![\phi_{x \leftarrow 1}]\!](I)\})$.*
- *if $\Sigma = \exists x.\phi$, then $[\![\Sigma]\!](I) = max(\{[\![\phi_{x \leftarrow 0}]\!](I), [\![\phi_{x \leftarrow 1}]\!](I)\})$.*

Clearly enough, every connective used in the morphology of $\mathtt{QPROP}_{PS}$ (including quantifications which can be viewed as unary connectives) is truth-functional.

An interpretation I is said to be a *model* of Σ, noted $I \models \Sigma$, if and only if $[\![\Sigma]\!](I) = 1$. If Σ has a model, it is *satisfiable*; otherwise, it is *unsatisfiable*. If every interpretation I over PS is a model of Σ, Σ is *valid*, noted $\models \Sigma$. If every model of Σ is a model of μ, then μ is a *logical consequence* of Σ, noted $\Sigma \models \mu$. Finally, when both $\Sigma \models \mu$ and $\mu \models \Sigma$ hold, Σ and μ are *equivalent*, noted $\Sigma \equiv \mu$.

It is not difficult to prove (again by structural induction) that the semantics of any quantified boolean formula Σ depends only on its free variables, in the sense that, for any interpretation J over PS which coincides with a given interpretation I on all the free variables of Σ, I is a model of Σ if and only if J is a model of Σ. Especially, the semantics of a closed formula is the same in every interpretation over PS. Stated otherwise, such a formula is equivalent to one of the boolean constants $true$ or $false$, hence it is satisfiable if and only if it is valid.

As in propositional logic, $\Sigma \models \mu$ holds if and only if the formula $(\Sigma \wedge \neg\mu)$ is unsatisfiable if and only if the formula $(\Sigma \Rightarrow \mu)$ is valid. More generally, since the connectives are truth-functional ones, a substitution metatheorem holds for quantified boolean formulae: replacing any subformula by an equivalent one preserves equivalence w.r.t. the overall formula.

It is easy to show that the conditioning of x by $*$ in Σ can be computed in time linear in $|\Sigma|$. Furthermore, it is easy to show by structural induction that successive conditionings commute: for any $\Sigma \in \mathtt{QPROP}_{PS}$, any $x, x' \in PS$ and $*, *' \in \{0,1\}$, we have

$$(\Sigma_{x \leftarrow *})_{x' \leftarrow *'} \equiv (\Sigma_{x' \leftarrow *'})_{x \leftarrow *}.$$

Other interesting metatheorems are given in the following two propositions.

Proposition 1 (folklore)**.** *Let Σ, Φ be formulae from $\mathtt{QPROP}_{PS}$ and x, y be variables from PS.*

1. $\forall x.\Sigma \equiv \Sigma_{x \leftarrow 0} \wedge \Sigma_{x \leftarrow 1}$.
2. $\exists x.\Sigma \equiv \Sigma_{x \leftarrow 0} \vee \Sigma_{x \leftarrow 1}$.
3. $\forall x.\Sigma \equiv \neg(\exists x.(\neg\Sigma))$.
4. *If x is not free in Σ, then* $\forall x.\Sigma \equiv \exists x.\Sigma \equiv \Sigma$.
5. $\forall x.(\Sigma \wedge \Phi) \equiv (\forall x.\Sigma) \wedge (\forall x.\Phi)$.
6. $\exists x.(\Sigma \vee \Phi) \equiv (\exists x.\Sigma) \vee (\exists x.\Phi)$.
7. $\forall x.(\forall y.\Sigma) \equiv \forall y.(\forall x.\Sigma)$.
8. $\exists x.(\exists y.\Sigma) \equiv \exists y.(\exists x.\Sigma)$.
9. *If x is not free in Σ, then* $\forall x.(\Sigma \vee \Phi) \equiv \Sigma \vee (\forall x.\Phi)$.
10. *If x is not free in Σ, then* $\exists x.(\Sigma \wedge \Phi) \equiv \Sigma \wedge (\exists x.\Phi)$.

Points 1 and 2 show that every quantified boolean formula Σ can be turned into a "standard" propositional formula but the transformation suggested (viewing the equivalences as left-to-right rewriting rules) cannot be achieved in polynomial space (and more generally, it is very unlikely that a polysize mapping from $\mathtt{QPROP}_{PS}$ to $\mathtt{PROP}_{PS}$ that preserves equivalence exist, since this would make the polynomial hierarchy PH to collapse). Accordingly, it is assumed that $\mathtt{QPROP}_{PS}$ enables much more compact encodings than $\mathtt{PROP}_{PS}$.

Point 3 shows that universal quantifiers and existential quantifiers are dual ones.

Point 4 gives a sufficient condition for the elimination of quantifications (the condition is not necessary as the example $\Sigma = x \vee \neg x$ shows it).

Points 5, 6, 9 and 10 make precise the interplay between the quantifiers and the connectives $\wedge$ and $\vee$.

Points 7 and 8 show that it is possible to switch two successive quantifications of the same nature in a quantified boolean formula Σ without questioning equivalence. Hence, for every finite, non empty subset $S = \{x_1, \ldots, x_n\}$ of PS, we note $\forall S.\Sigma$ (resp. $\exists S.\Sigma$) as a shortand for $\forall x_1, \ldots, \forall x_n.\Sigma$ (resp. $\exists x_1, \ldots, \exists x_n.\Sigma$).

Contrastingly, it is not possible in general to switch two successive quantifications of different nature while preserving equivalence. Thus, $\forall x.(\exists y.\Sigma)$ is a logical consequence of $\exists y.(\forall x.\Sigma)$ but is not equivalent to it (just consider $\Sigma = x \Leftrightarrow y$). Furthermore, in the general case, we do have neither $\forall x.(\Sigma \vee \Phi) \equiv (\forall x.\Sigma) \vee (\forall x.\Phi)$ nor $\exists x.(\Sigma \wedge \Phi) \equiv (\exists x.\Sigma) \wedge (\exists x.\Phi)$ (as a counterexample, take $\Sigma = x$ and $\Phi = \neg x$).

Based on the metatheorems given in Proposition 1 and the substitution metatheorem, it is easy to prove that every formula from $\texttt{QPROP}_{PS}$ can be turned in polynomial time into a prenex and polite, equivalent formula (bound variables can be renamed without questioning equivalence). Note that several strategies for prenexing a quantified boolean formula exist (depending on the way quantifications are shifted), and that the choice of a prenex formula equivalent to a given quantified boolean formula Σ when several are possible may have a practical impact on the efficiency of deciding the validity of Σ [40].

Example 5 (cont'ed)**.** *Σ is equivalent to the following prenex, polite QBF:*

$$\exists a.(\forall u.(\forall v.(((a \wedge u) \vee ((\neg a) \vee b)) \wedge (a \vee ((\neg v) \wedge c))))).$$

u and v are the fresh variables used to make the formula polite. ■

Finally, there are close connections between (general) quantified boolean formulae and closed ones. Especially we have:

Proposition 2 (folklore)**.** *Let Σ be a formula from $\texttt{QPROP}_{PS}$.*

1. *Σ is satisfiable if and only if the closed formula $\exists Var(\Sigma).\Sigma$ is valid.*
2. *Σ is valid if and only if the closed formula $\forall Var(\Sigma).\Sigma$ is valid.*

Points 1 and 2 above show that the satisfiability problem (resp. the validity problem) of (general) quantified boolean formulae can be reduced in polynomial time to the validity problem of closed quantified boolean formulae. Especially, the satisfiability problem of a "standard" propositional formula Σ can be reduced in polynomial time to the validity problem of the closed quantified boolean formula $\exists Var(\Sigma).\Sigma$.

Example 6 (cont'ed)**.** *Σ is satisfiable since*

$$\exists a, b, c.(\exists a.(((\forall b.(a \wedge b)) \vee ((\neg a) \vee b)) \wedge \forall b.(a \vee ((\neg b) \wedge c))))$$

which can be simplified to

$$\exists b, c.(\exists a.(((\forall b.(a \wedge b)) \vee ((\neg a) \vee b)) \wedge \forall b.(a \vee ((\neg b) \wedge c))))$$

since a is not free in Σ, is a valid, closed QBF. ■

For this reason, the VAL($\mathtt{QPROP}_{PS}$) problem is usually stated as follows:

Definition 4 (VAL($\mathtt{QPROP}_{PS}$))**.** VAL($\mathtt{QPROP}_{PS}$) *is the following decision problem:*

- **Input:** *A prenex, closed, polite formula* Σ *from* $\mathtt{QPROP}_{PS}$*;*
- **Question:** *Is* Σ *valid?*

Example 7 (cont'ed)**.** Σ *is satisfiable if and only if the following instance of* VAL($\mathtt{QPROP}_{PS}$) *is valid:*

$$\exists a, b, c\ \forall u, v.(((a \wedge u) \vee ((\neg a) \vee b)) \wedge (a \vee ((\neg v) \wedge c)))$$

■

More generally, we use the following notations:

Definition 5 (Notations)**.** *Let* $\mathcal{C} \subseteq \mathtt{PROP}_{PS}$*.*
We note:

- $\mathtt{Q}\mathcal{C}$ *is the language of prenex, closed, polite QBF with matrix from* $\mathcal{C}$*.*
- VAL($\mathtt{Q}\mathcal{C}$) *is the following decision problem:*
  - **Input:** *A formula* Σ *from* $\mathtt{Q}\mathcal{C}$*;*
  - **Question:** *Is* Σ *valid?*

3. Tractable vs. Intractable Classes for VAL($\mathtt{QPROP}_{PS}$)

In the following, the complexity of several restrictions of VAL($\mathtt{QPROP}_{PS}$) is investigated. A propositional fragment (i.e., a subset of a propositional language) $\mathcal{C}$ is said to be *tractable* for VAL($\mathtt{QPROP}_{PS}$) (resp. SAT) if and only if the membership to the fragment can be decided in polynomial time, and there also exists a polytime decision algorithm for the validity problem VAL($\mathtt{Q}\mathcal{C}$) (resp. there exists a polytime decision algorithm for the satisfiability problem for formulae from the fragment).

Let us start with intractability results. First, it is well-known that the restriction VAL($\mathtt{QCNF}$) of VAL($\mathtt{QPROP}_{PS}$) is still PSPACE-complete. Indeed, every propositional formula Σ over $\{x_1, \ldots, x_n\}$ can be associated in linear time to a CNF formula Σ' over $\{x_1, \ldots, x_n, y_1, \ldots, y_m\}$ s.t. $\Sigma \equiv \exists\{y_1, \ldots, y_m\}.\Sigma'$. Such a reduction which preserves satisfiability (and much more) is typically used to show that CIRCUIT-SAT can be reduced to SAT restricted to CNF formulae (the basic idea is to introduce a new variable y_i per gate or subformula).

Let us now consider the VAL($\mathtt{Q}\mathcal{C}$) problem for target fragments $\mathcal{C}$ for knowledge compilation; many such fragments have been identified in the literature: DNF, sd-DNNF, d-DNNF, DNNF, FBDD, $\mathtt{OBDD}_<$, MODS, PI, IP, ... As we will see, all such problems are typically intractable.

First, since PSPACE is closed under complementation and the negation of a QBF with a CNF matrix is a QBF with a DNF matrix, it follows directly that the VAL($\mathtt{QDNF}$) problem also is PSPACE-complete. This prevents many tractable fragments $\mathcal{C}$ for SAT from being considered as interesting candidates for VAL($\mathtt{Q}\mathcal{C}$). Among them are all the supersets of DNF

including the DNNF fragment and the disjunctions of Horn CNF formulae which are target classes for knowledge compilation (see [3, 4, 13]).

Let us now turn to complete DAG-based propositional fragments.[2] A "formula"[3] in $\mathtt{NNF}_{PS}$ is a rooted, directed acyclic graph where each leaf node is labeled with *true*, *false*, x or $\neg x$, $x \in PS$; and each internal node is labeled with $\wedge$ or $\vee$ and can have arbitrarily many children. If C is a node in an $\mathtt{NNF}_{PS}$ formula, then $Var(C)$ denotes the set of all variables that label the descendants of node C. Moreover, if ϕ is an $\mathtt{NNF}_{PS}$ formula rooted at C, then $Var(\phi)$ is defined as $Var(C)$. Interesting fragments of $\mathtt{NNF}_{PS}$ are obtained by imposing some of the following requirements [41]:

- **Decomposability**: An and-node C is decomposable if and only if the conjuncts of C do not share variables. That is, if $C_1, \ldots, C_n$ are the children of and-node C, then $Var(C_i) \cap Var(C_j) = \emptyset$ for $i \neq j$. An $\mathtt{NNF}_{PS}$ formula satisfies the decomposability property if and only if every and-node in it is decomposable.

- **Determinism**: An or-node C is deterministic if and only if each pair of disjuncts of C is logically contradictory. That is, if $C_1, \ldots, C_n$ are the children of or-node C, then $C_i \wedge C_j \models \mathit{false}$ for $i \neq j$. An $\mathtt{NNF}_{PS}$ formula satisfies the determinism property if and only if every or-node in it is deterministic.

- **Decision**: A decision node N in an $\mathtt{NNF}_{PS}$ formula is one which is labeled with *true*, *false*, or is an or-node having the form $(x \wedge \alpha) \vee (\neg x \wedge \beta)$, where x is a variable, α and β are decision nodes. In the latter case, $dVar(N)$ denotes the variable x. An $\mathtt{NNF}_{PS}$ formula satisfies the decision property when its root is a decision node.

- **Ordering**: Let $<$ be a total, strict ordering over the variables from PS. An $\mathtt{NNF}_{PS}$ formula satisfying the decision property satisfies the ordering property w.r.t. $<$ if and only if the following condition is satisfied: if N and M are or-nodes, and if N is an ancestor of node M, then $dVar(N) < dVar(M)$.

- **Smoothness**: An or-node C is smooth if and only if each disjunct of C mentions the same variables. That is, if $C_1, \ldots, C_n$ are the children of or-node C, then $Var(C_i) = Var(C_j)$ for $i \neq j$. An $\mathtt{NNF}_{PS}$ formula satisfies the smoothness property if and only if every or-node in it is smooth.

Example 8. *Consider the and-node marked ➴ on the left part of Figure 2. This and-node C has two children C_1 and C_2 such that $Var(C_1) = \{a, b\}$ and $Var(C_2) = \{c, d\}$; Node C is decomposable since the two children do not share variables. Each other and-node in Figure 2 (left) is also decomposable and, hence, the $\mathtt{NNF}_{PS}$ formula in this figure is decomposable. Consider now the or-node marked ➴ on the right part of the figure; it has two children corresponding to subformulae $\neg a \wedge b$ and $\neg b \wedge a$. Those two subformulae are jointly unsatisfiable, hence the or-node is deterministic. Furthermore, the two children*

2. Actually, the DAG-based fragment corresponding to CNF (resp. DNF) can be identified to the fragment CNF (resp. DNF) consisting of tree-like formulae – as defined before – without any significant loss w.r.t. succinctness. See [13].
3. In the following, we will use the term formula to denote the DAG-based representation of that formula.

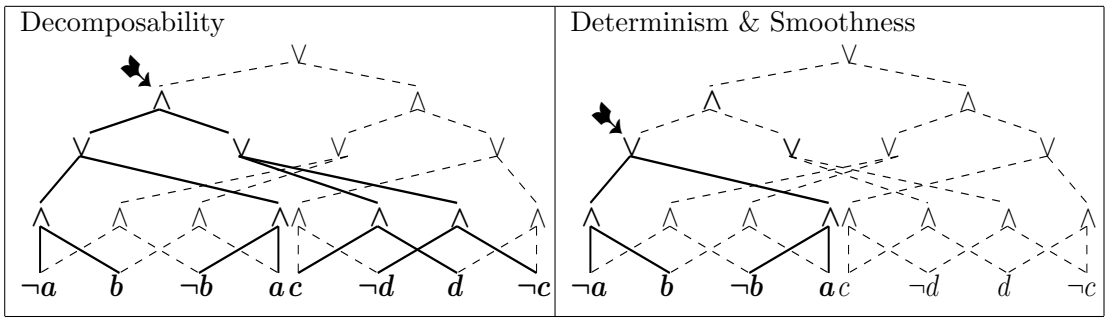


Figure 2. A formula in $\mathtt{NNF}_{PS}$. On the left part, the node marked ⬎ is decomposable while the node marked with the same symbol on the right part denotes a deterministic and smooth node.

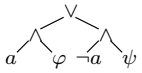
mention the same variables a and b, hence the or-node is smooth. Since the other or-nodes in Figure 2 (right) are also deterministic and smooth, the $\mathtt{NNF}_{PS}$ *formula in this figure is deterministic and smooth.* ■

We consider the following propositional fragments[4.] [13]:

Definition 6 (propositional fragments)**.**

- *The language* DNNF *is the subset of* $\mathtt{NNF}_{PS}$ *of formulae satisfying decomposability.*
- *The language* d-DNNF *is the subset of* $\mathtt{NNF}_{PS}$ *of formulae satisfying decomposability and determinism.*
- *The language* sd-DNNF *is the subset of* $\mathtt{NNF}_{PS}$ *of formulae satisfying decomposability, determinism and smoothness.*
- *The language* FBDD *is the subset of* $\mathtt{NNF}_{PS}$ *of formulae satisfying decomposability and decision.*
- *The language* $\mathtt{OBDD}_{<}$ *is the subset of* $\mathtt{NNF}_{PS}$ *of formulae satisfying decomposability, decision and ordering.*
- *The language* MODS *is the subset of* DNF ∩ d-DNNF *of formulae satisfying smoothness.*

The FBDD language corresponds to *free binary decision diagrams (FBDDs)*, as known in formal verification [42], while its subset obtained by imposing the ordering property w.r.t. a given variable ordering contains the *ordered binary decision diagrams (OBDDs)* [39].

Binary decision diagrams are usually depicted using a more compact notation: labels *true* and *false* are denoted by 1 and 0, respectively; and each decision node 

4. It must be noted that the six languages below are not *stricto sensu* subsets of $\mathtt{PROP}_{PS}$ in the sense that its elements are rooted DAGs, not standard tree-like formulae. Considering DAG-based representations is just a way to enable subformulae sharing; while this is fundamental for the spatial efficiency point of view, this has no impact on the semantical issue, so the definitions and properties reported in the Section 2 can be easily extended to DAG-based formulae.

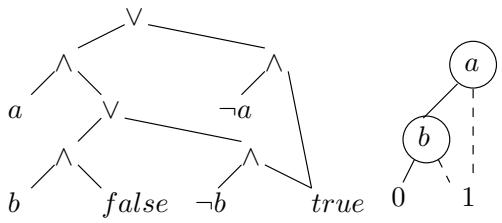


Figure 3. On the left part, a formula in the OBDD$_<$ language. On the right part, a more standard notation for it.

denoted by $\overset{a}{\underset{\varphi}{\diagdown}}\!\!\overset{}{\diagup}_{\psi}$. The OBDD$_<$ formula on the left part of Figure 3 corresponds to the binary decision diagram on the right part of Figure 3.

A MODS encoding of a propositional formula mainly consists of the explicit representation of the set of its models over the set of variables occurring in it. Figure 4 depicts a formula from MODS which is equivalent to the DNFformula $(a \wedge b \wedge c) \vee (\neg a \wedge \neg b \wedge c) \vee (\neg a \wedge b \wedge c) \vee (\neg a \wedge \neg b \wedge \neg c)$.

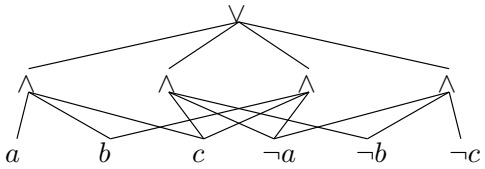


Figure 4. An element of the MODS fragment.

For the sake of completeness, we also consider the dual language of MODS, consisting of CNF formulae given by their *canonical implicates*:

- The language CI is the subset of CNF containing formulae $\Sigma = \gamma_1 \wedge \ldots \wedge \gamma_k$ such that for each clause γ_i $(i \in 1 \ldots k)$ and for each variable $x \in Var(\Sigma)$, γ_i contains x or $\neg x$.

Obviously enough, the negation of any MODS formula can be turned in linear time into a CI formula, and the converse also holds.

Now, it is well-known that eliminating a single quantification within an OBDD$_<$ formula can be achieved in time quadratic in the input size (an OBDD$_<$ formula equivalent to $\exists x.\Sigma$ (resp. $\forall x.\Sigma$) is computed as $\Sigma_{x \leftarrow 0} \vee \Sigma_{x \leftarrow 1}$, (resp. $\Sigma_{x \leftarrow 0} \wedge \Sigma_{x \leftarrow 1}$), see e.g. [39]). Since the size of the resulting formula may be quadratic in the size of the input Σ, there is no guarantee that such an elimination process leads to a formula of size polynomial in the input size when iterated so as to eliminate more than a preset number of variables. Hence, there is no guarantee that the time needed by such an elimination algorithm will remain polynomial in the input size. Actually, the next proposition shows that whatever the approach to solving VAL(QOBDD$_<$), a polytime algorithm is very unlikely:

Proposition 3. VAL(QDNNF), VAL(Qd-DNNF), VAL(QFBDD) *and* VAL($\mathtt{QOBDD}_<$) *are* PSPACE*-complete.*

Proof 1. *The membership directly comes from the fact that* VAL(QCNF) *is in* PSPACE*, the fact that the circuit language associated to* $\mathtt{PROP}_{PS}$ *includes* $\mathtt{NNF}_{PS}$ *as a proper subset, the fact that every circuit (encoding a boolean function over* $\{x_1, \ldots, x_n\}$*) can be mapped in polynomial time to a* CNF *formula over an extended set of variables, whilst equivalent to the circuit whenever the new variables are forgotten (i.e., existentially quantified) (see e.g. [43]), and the fact that* PSPACE *is closed under polynomial reduction.*

As to hardness, since the following inclusions hold (cf. Figure 5 in the appendix)

$$\mathtt{OBDD}_< \subset \mathtt{FBDD} \subset \mathtt{d\text{-}DNNF} \subset \mathtt{DNNF}$$

it is sufficient to prove that the VAL($\mathtt{QOBDD}_<$) *problem is* PSPACE*-hard. The proof is by reduction from* VAL(QDNF)*. The main step is to show that every* DNF *formula* $\phi = \gamma_1 \vee \ldots \vee \gamma_n$ *can be associated in polynomial time to an equivalent QBF of the form* $\exists X.\psi$ *where* $X \cap Var(\phi) = \emptyset$ *and* ψ *is from* $\mathtt{OBDD}_<$ *(whatever* $<$ *over* $Var(\phi)$*). First, let us note* $obdd(\gamma_i)$ *the* $\mathtt{OBDD}_<$ *formula equivalent to the term* γ_i *(*$i \in 1 \ldots n$*); clearly enough, every* $obdd(\gamma_i)$ *can be computed in time polynomial in* $|\gamma_i|$*. Let* $Var(\phi) = \{y_1, \ldots, y_m\}$ *and let* $X = \{x_1, \ldots, x_{n-1}\}$ *be a set of new variables; let* $\psi = \psi^1$*, where the formulae* ψ^i *(*$i \in 1 \ldots n$*) are defined by:*

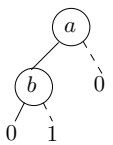
- $\psi^n = obdd(\gamma_n)$*, and*
- $\psi^i = (obdd(\gamma_i) \wedge x_i) \vee (\psi^{i+1} \wedge \neg x_i)$*, for* $i = 1, \ldots, n-1$*.*

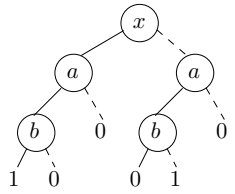
From such definitions, ψ *– which can be read as an* $\mathtt{OBDD}_<$ *formula where the new ordering* $<$ *is the extension of the previous ordering* $y_1 < \ldots < y_m$ *such that* $x_1 < \ldots < x_{n-1} < y_1 < \ldots < y_m$ *– can be computed in time polynomial in the size of* ϕ*. Now, since for every QBF* α*,* β *and every variable* x*, we have that* $\exists x.(\alpha \vee \beta) \equiv (\exists x.\alpha) \vee (\exists x.\beta)$ *(from point 6 in Proposition 1), and* $\exists x.(\alpha \wedge x) \equiv \exists x.(\alpha \wedge \neg x) \equiv \alpha$ *whenever* $x \notin Var(\alpha)$ *(from point 10 in Proposition 1), it immediately follows that* $\phi \equiv \exists X.\psi$*. Finally, the substitution metatheorem for QBF shows that for any prefix* P*, the QBF* $P.\phi$ *is equivalent to the QBF* $P\ \exists X.\psi$*, and this completes the proof.* □

The following example illustrates the proof above:

Example 9. *Let us consider the following* DNF *formula* $\phi = (\gamma_1 \vee \gamma_2)$ *where* $\gamma_1 = (a \wedge b)$ *and* $\gamma_2 = (a \wedge \neg b)$*.*

We have :

$\psi^2 \equiv$ 

So $\psi \equiv \psi^1 \equiv$ 

and $\phi \equiv \exists x.\psi$. ■

Based on the reduction given in the proof above, one can also show that the VAL(QOBDD$_<$) problem spans all the polynomial hierarchy when restrictions are put on the prefix of the input: if no alternations of quantifiers occur, the problem reduces to the satisfiability problem or to the validity problem, and both of them are in P for OBDD$_<$ formulae; if the prefix is of the form $\forall S_1\ \exists S_2$, the problem is Π_1^p-complete (= coNP-complete); if the prefix is of the form $\exists S_1\ \forall S_2\ \exists S_3$, the problem is Σ_2^p-complete, and so on. Since the negation of an OBDD$_<$ formula can be computed as an OBDD$_<$ formula in constant time, we also obtain that if the prefix is of the form $\exists S_1\ \forall S_2$, the problem is Σ_1^p-complete (= NP-complete), if the prefix is of the form $\forall S_1\ \exists S_2\ \forall S_3$, the problem is Π_2^p-complete, and so on.

Adding the smoothness requirement to VAL(d-DNNF) does not help to reduce the complexity; indeed, every d-DNNF formula can be smoothed in polynomial time:

Corollary 1. VAL(Qsd-DNNF) *is* PSPACE*-complete.*

Proof 2. *Direct from the fact that the* VAL(Qd-DNNF) *is* PSPACE*-hard and that a* d-DNNF *can be turned into an equivalent* sd-DNNF *formula in polynomial time (see Lemma A.1 in [13]).* □

Contrastingly, VAL(QOBDD$_<$) is tractable when the corresponding prefixes are compatible with the total, strict ordering $<$ associated with the OBDD$_<$ fragment:

Definition 7 (compatibility)**.** *Let* $\Sigma = QS_1 \ldots QS_n.\phi$ *be a prenex, polite, closed QBF where each* Q *stands for a quantifier and* $\{S_1, \ldots, S_n\}$ *is a partition of* $Var(\phi)$ *which does not contain the empty set. The prefix* $QS_1 \ldots QS_n$ *of* Σ *is said to be* compatible *with a total, strict ordering* $<$ *over* $Var(\phi)$ *if and only if for each* $x, y \in Var(\phi)$ *s.t.* $x < y$*, if* $x \in S_i$ *and* $y \in S_j$ *then* $j \geq i$*.*

Proposition 4. *The restriction of the* VAL(QOBDD$_<$) *problem for instances with a prefix compatible with* $<$ *is in* P*.*

Proof 3. *The proof is constructive and is given by the polytime algorithm* SolveObdd *below. Such an algorithm consists in eliminating the quantifications (from the innermost to the outermost) into the input formula.*

Let x *be the greatest variable w.r.t.* $<$*.*

Let us first assume that x *is existentially quantified. By construction, an interpretation* I *is a model (resp. a counter-model) of an* OBDD$_<$ *formula* ϕ *if and only if the unique path from the root of* ϕ *that is compatible with* I *leads to the sink* 1 *(resp.* 0*); such a path corresponds to an implicant of* ϕ *(resp. its negation).*

Let N *be any decision node labeled by* x *in* ϕ*. Given the ordering assumption about* x*, the only possible children of* N *in* ϕ *are the sink nodes (actually, one of them is the* 1 *sink and the other one is the* 0 *sink if the ordered binary decision diagram* ϕ *is reduced, which can be assumed w.l.o.g. since such a reduction can be achieved in polynomial time).*

Since every model of $\exists x.\phi$ *coincides with a model of* ϕ *except possibly on* x*, it is sufficient to remove every node* N *labeled by* x *and to re-direct its incoming edges to the* 1 *sink to obtain an* OBDD$_<$ *formula equivalent to* $\exists x.\phi$*; the resulting formula is not necessarily reduced,*

but it can be reduced in polynomial time if it is not the case. The overall process can be easily achieved in time polynomial in the size of ϕ.

Now, if the greatest variable x w.r.t. $<$ is universally quantified, we take advantage of the equivalence $\forall x.\phi \equiv \neg(\exists x.\neg\phi)$ (point 3 in Proposition 1) and the fact that an $\mathtt{OBDD}_<$ formula equivalent to the negation of an $\mathtt{OBDD}_<$ formula can be obtained in constant time (just switch the labels of the two sink nodes). This completes the proof. □

Algorithm 1: Polytime algorithm for the restriction of VAL($\mathtt{QOBDD}_<$) to instances with a compatible prefix

procedure SOLVEOBDD
Data: A formula $Q_1x_1 \ldots Q_nx_n.\phi$ from $\mathtt{QOBDD}_<$ where $\phi \in \mathtt{OBDD}_<$ is reduced and $x_1 < \cdots < x_n$
Result: 1 if $Q_1x_1 \ldots Q_nx_n.\phi$ is valid, 0 otherwise
begin
  for i **from** n **to** 1 **do**
    if $Q_i = \exists$ **then**
      foreach *node* N *labeled by* x_i **do**
        $N \leftarrow 1$;
    else
      foreach *node* N *labeled by* x_i **do**
        $N \leftarrow 0$;
    REDUCE(ϕ) ;
  return ϕ ;
end

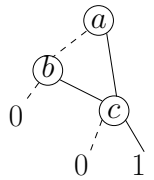
A REDUCE procedure is described in [39], and a more efficient one, running in time linear in the input size, is given in [44]. As a consequence, SOLVEOBDD runs in time $\mathcal{O}(n \times |\phi|)$.

The following example illustrates a run of SOLVEOBDD:

Example 10. *Let $<$ be the ordering over $\{a, b, c\}$ such that $a < b < c$. Let us consider*

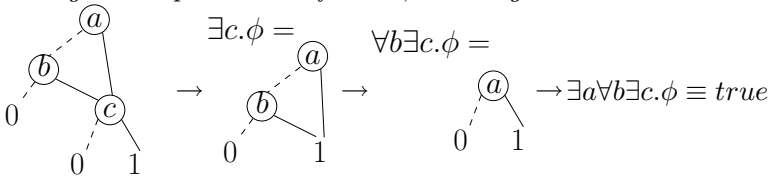
$$\Sigma = \exists a\ \forall b\ \exists c.\phi$$

$$\textit{with } \phi = (a \vee b) \wedge c$$

The $\mathtt{OBDD}_<$ formula associated to ϕ is: 

.

The algorithm proceeds as follows, allowing us to conclude that Σ is valid.



■

Compatibility proves to be a key for deciding the validity problem for QBF when restricted to other fragments. Thus, we have also considered the two propositional fragments $\texttt{ODNF}_<$ and $\texttt{OCNF}_<$ which are respectively a subset of DNF and of CNF. Let $<$ be a total, strict ordering over the variables from $PS = \{x_1, \ldots, x_n\}$ s.t. $x_1 < \ldots < x_n$. We define:

- The language $\texttt{ODNF}_<$ is the set of all DNF formulae $\Sigma = \gamma_1 \vee \ldots \vee \gamma_k$ where each satisfiable term γ_i $(i \in 1 \ldots k)$ of Σ is such that for every $j \in 1 \ldots n$, if x_j or $\neg x_j$ occurs in γ_i then for every l such that $1 \leq l < j$, x_l or $\neg x_l$ occurs in γ_i.
- The language $\texttt{OCNF}_<$ is the set of all CNF formulae $\Sigma = \gamma_1 \wedge \ldots \wedge \gamma_k$ where each non tautologous clause γ_i $(i \in 1 \ldots k)$ of Σ is such that for every $j \in 1 \ldots n$, if x_j or $\neg x_j$ occurs in γ_i then for every l such that $1 \leq l < j$, x_l or $\neg x_l$ occurs in γ_i.

Example 11. *Let $PS = \{a, b, c\}$ and $<$ such that $a < b < c$. $\neg a \vee (a \wedge b)$ is a formula from $\texttt{ODNF}_<$. $(a \vee b) \wedge (\neg a \vee b \vee c)$ is a formula from $\texttt{OCNF}_<$.* ■

Clearly enough, $\texttt{OCNF}_<$ (resp. $\texttt{QOCNF}_<$) is the dual fragment of $\texttt{ODNF}_<$ (resp. $\texttt{QODNF}_<$); especially the negation of any $\texttt{ODNF}_<$ (resp. $\texttt{QODNF}_<$) formula can be computed in linear time as a $\texttt{OCNF}_<$ (resp. $\texttt{QOCNF}_<$) formula, and the converse also holds. Furthermore, both $\texttt{OCNF}_<$ and $\texttt{ODNF}_<$ are complete propositional fragments since they include respectively the complete fragments MODS and CI.

Proposition 5. *The restrictions of* VAL($\texttt{QODNF}_<$) *and* VAL($\texttt{QOCNF}_<$) *to instances with a prefix compatible with $<$ are in* P.

Proof 4. *Since the negation of a $\texttt{QOCNF}_<$ formula can be computed in linear time as a $\texttt{QODNF}_<$ formula, it is sufficient to prove that* VAL($\texttt{QODNF}_<$) *is in* P *when restricted to instances with a prefix compatible with $<$.*

Again, the proof is constructive and is given by the polytime algorithm Solve$\texttt{ODNF}_<$ *below. This algorithm consists in eliminating quantifications as an internal law in the $\texttt{ODNF}_<$ fragment. In this algorithm, every $\texttt{ODNF}_<$ formula ϕ is considered w.l.o.g. as a set of terms and every term as a set of literals. After the first step of the algorithm, every term in ϕ is satisfiable (removing unsatisfiable terms in ϕ can be easily achieved in polynomial time). The resulting $\texttt{ODNF}_<$ formula is either the empty set (i.e., the empty disjunction equivalent to $false$ ($\{\} \equiv false$)) or a set containing the empty conjunction, equivalent to $true$ ($\{\{\}\} \equiv true$).*

Let us explain how quantifications can be eliminated and first consider the case of existential quantifiers; we take advantage of two equivalences: for every pair of QBF α and β and every variable $x \in PS$, we have $\exists x.(\alpha \vee \beta) \equiv (\exists x.\alpha) \vee (\exists x.\beta)$ (point 6 in Proposition 1), and $\exists x.\gamma$ is equivalent to the term $\gamma \setminus \{x, \neg x\}$ obtained by removing x and $\neg x$ from γ whenever γ is a satisfiable term (viewed as the set of its literals) (which is a direct consequence of the definition of conditioning and point 2 in Proposition 1). Let x be the last variable of $Var(\Sigma)$ w.r.t. $<$. Clearly enough, if γ is a canonical term over $X \cup \{x\}$ (with $x \notin X$), then $\gamma \setminus \{x, \neg x\}$ is a canonical term over X. Hence removing every occurrence of x and $\neg x$ in such an $\texttt{ODNF}_<$ formula ϕ leads to an $\texttt{ODNF}_<$ formula equivalent to $\exists x.\phi$.

Let us now focus on the case of universal quantifiers. Let ϕ be an $\texttt{ODNF}_<$ formula, viewed as a set of satisfiable terms over $X \cup \{x\}$, where $x \notin X$ and x is the last variable of $Var(\Sigma)$

w.r.t. $<$. *Each term of* ϕ *containing* x *or* $\neg x$ *is canonical over* $X \cup \{x\}$. *Now, the terms* γ *of* ϕ *can be partitioned into three sets (disjunctively interpreted):* W, S *and* S'. W *is the set of all terms* γ *s.t.* $x \notin Var(\gamma)$. W *can be viewed as an* $\mathtt{ODNF}_<$ *formula over* X. S *is the set of all terms* γ *of* ϕ *containing* x *or* $\neg x$ *s.t.* $switch(\gamma, x)$ *belongs to* ϕ *as well, where* $switch(\gamma, x)$ *is the canonical term over* $X \cup \{x\}$ *which coincides with* γ *for every variable of* X *but contains* $\neg x$ *whenever* γ *contains the literal* x, *while* $switch(\gamma, x)$ *contains* x *whenever* γ *contains* $\neg x$. *It is obvious that* γ *belongs to* S *if and only if* $switch(\gamma, x)$ *belongs to* S. *Finally,* S' *is the set of remaining terms from* ϕ *(i.e. every term* $\gamma \in S'$ *contains* x *or* $\neg x$ *but does not belong to* S*). By construction, we have* $\forall x.\phi \equiv \forall x.(W \vee S \vee S') \equiv W \vee \forall x.(S \vee S')$ *(from point 9 from Proposition 1). Let us now observe that* $S = \{\gamma_1, \ldots, \gamma_k\}$ *(disjunctively interpreted) is independent from* x *[45] in the sense that it is equivalent to the disjunction* $\psi = (\gamma_1 \setminus \{x, \neg x\}) \vee \ldots \vee (\gamma_k \setminus \{x, \neg x\})$ *which is an* $\mathtt{ODNF}_<$ *formula over* X *as well (for every* γ_i *(*$i \in 1 \ldots k$*), we have* $\gamma_i \vee switch(\gamma_i, x) \equiv \gamma_i \setminus \{x, \neg x\}$*). Clearly enough,* ψ *can be computed in time polynomial in the size of* ϕ. *Since* ψ *is independent from* x, *we get* $\forall x.\phi \equiv W \vee \psi \vee (\forall x.S')$. *Finally, it remains to show that* $\forall x.S'$ *is unsatisfiable (so that* $\forall x.\phi \equiv W \vee \psi$*). Let* γ *be any canonical term over* X *viewed as an interpretation over* X*; by definition,* γ *satisfies* $\forall x.S'$ *if and only if both the extension of it assigning* x *to* 0 *and the extension of it assigning* x *to* 1 *are models of* S'*; but this is impossible due to the definition of* S' *(if one of such extensions belongs to* S', *it cannot be the case that the second one belongs to* S' *as well since it is obtained by switching* x *in the first one). Accordingly,* $(\forall x.S')$ *is unsatisfiable, and this concludes the proof.* □

Algorithm 2: Polytime algorithm for the restriction of VAL($\mathtt{QODNF}_<$) to instances with a compatible prefix

procedure SOLVEODNF$_<$
Data: A formula $Q_1 x_1 \ldots Q_n x_n.\phi$ where $\phi \in \mathtt{ODNF}_<$ and $x_1 < \ldots < x_n$
Result: 1 if $Q_1 x_1 \ldots Q_n x_n.\phi$ is valid, 0 otherwise
begin
  Remove unsatisfiable terms from ϕ ;
  for i **from** n **to** *1* **do**
    if $Q_i = \forall$ **then**
      $\phi' \leftarrow \{\gamma \in \phi \mid switch(\gamma, x_i) \in \phi$ or $x \notin Var(\gamma)\}$;
    else
      $\phi' \leftarrow \phi$;
    $\phi \leftarrow \{\gamma \setminus \{x_i, \neg x_i\} \mid \gamma \in \phi'\}$;
  if $\phi = \{\}$ **then**
    return 0 ;
  else
    return 1 ;
end

SOLVEODNF$_<$ runs in time $\mathcal{O}(n \times |\phi|^2)$. This algorithm could be easily refined, returning 0 as soon as the empty set has been obtained.

Let us illustrate how SolveODNF$_<$ works on two simple examples (where $a < b$):

Example 12.

1. $\forall a \exists b.\{\{\neg a\}, \{a, \neg b\}\} \rightarrow \forall a.\{\{\neg a\}, \{a\}\} \rightarrow \{\{\}\} \equiv true.$
2. $\exists a \forall b.\{\{\neg a, b\}, \{a, b\}\} \rightarrow \exists a.\{\} \rightarrow \{\} \equiv false.$

■

As an immediate consequence, one gets the tractability of val(QMODS) and val(QCI):

Corollary 2. val(QMODS) *and* val(QCI) *are in* P.

Proof 5. *Direct from the fact that every formula from* QMODS *(resp.* QCI*) is a formula from* QODNF$_<$ *(resp.* QOCNF$_<$*) with a prefix compatible with* $<$*, whatever the strict, total ordering* $<$*.* □

Let us now turn to two additional, important propositional fragments in AI: the prime implicates one and the (dual) prime implicants one (see e.g., [46] for a survey of their applications in abduction, assumption-based reasoning, closed world reasoning and other AI areas). Formally, a *prime implicate* of $\phi \in PROP_{PS}$ is a clause δ s.t. $\phi \models \delta$ and for every clause δ' s.t. $\phi \models \delta'$ and $\delta' \models \delta$, we have $\delta \equiv \delta'$.

Definition 8 (prime implicates formulae)**.** *A prime implicates formula (or Blake formula) from* PROP$_{PS}$ *is a* CNF *formula* Σ *where every prime implicate of* Σ *(up to logical equivalence) appears as a conjunct.* PI *is the language of all prime implicates formulae (a proper subset of* CNF*).*

Example 13. *The following is a prime implicates formula:*

$$(a \vee b) \wedge (\neg b \vee c \vee d) \wedge (a \vee c \vee d).$$

■

The set of prime implicates formulae is a tractable fragment for SAT since (1) a CNF formula Σ is a prime implicates one if and only if no clause of it is properly entailed by another clause of Σ and every resolvent from two clauses from Σ is entailed by a clause of Σ (this shows that the problem of deciding whether a propositional formula is a prime implicates one can be decided in polynomial time), and (2) a prime implicates formula Σ is satisfiable if and only if it does not reduce to the empty clause.

Unfortunately, PI is not a tractable fragment for val(QPROP$_{PS}$) (under the usual assumptions of complexity theory):

Proposition 6. val(QPI) *is* PSPACE*-complete.*

Proof 6. *Membership comes directly from the fact that* val(QCNF) *is in* PSPACE *and every* PI *formula also is a* CNF *formula.*

As to hardness, let us give a polytime reduction from val(QCNF) *to* val(QPI)*. Let* ϕ *be a* CNF *formula over* $\{x_1, \ldots, x_n\}$*, viewed as the set* S *of its clauses. We take advantage of*

the following property, which results directly from the correctness of resolution-based prime implicates algorithms (like Tison's one [47]): a set S of clauses contains all its prime implicates if and only if whenever two clauses from S have a resolvent δ, there exists a clause $\epsilon \in S$ s.t. $\epsilon \models \delta$.

Let us assume that S is totally ordered w.r.t. an arbitrary (but fixed) ordering $<$. Let δ_i and δ_j be two clauses from S with $i < j$; when δ_i and δ_j have a resolvent, replace δ_i by $\delta_i \vee y_{i,j}$ and δ_j by $\delta_j \vee \neg y_{i,j}$; doing it in a systematic way for every ordered pair of clauses from S leads to generate in polynomial time a CNF *formula ψ over an extended vocabulary $\{x_1, \ldots, x_n\} \cup Y$ where $\mathcal{O}(n^2)$ new variables $y_{i,j}$ are introduced. By construction, every binary resolvent from clauses of ψ is tautologous, hence implied by any clause of ψ. As a consequence, ψ contains all its prime implicates, and a prime implicates formula θ equivalent to ψ can be computed in time polynomial in $|\psi|$, just by removing every clause of ψ which is properly implied by another clause from ψ.*

Now, for every pair of QBFs α and β and every variable $x \in PS$, we have $\forall x.(\alpha \wedge \beta) \equiv (\forall x.\alpha) \wedge (\forall x.\beta)$ (point 5 in Proposition 1); furthermore, for every non tautologous clause δ (viewed as the set of its literals) and every variable $x \in PS$, $\forall x.\delta$ is equivalent to the clause $\delta \setminus \{x, \neg x\}$ (this is a direct consequence of the definition of conditioning and point 1 in Proposition 1). As a consequence, we have $\phi \equiv \forall Y.\theta$. Finally, the substitution metatheorem for QBFs shows that $QS_1 \ldots QS_k.\phi$ is equivalent to $QS_1 \ldots QS_k \forall Y.\theta$, and this concludes the proof. □

The following example illustrates the reduction given in the above proof:

Example 14. *Let $\phi = (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg c \vee d)$ be a* CNF *formula. We associate in polynomial time ϕ to the following* PI *formula $\theta = (a \vee b \vee y_{1,2}) \wedge (\neg b \vee c \vee d \vee \neg y_{1,2} \vee y_{2,3}) \wedge (\neg c \vee d \vee \neg y_{2,3})$. We have $\phi \equiv \forall y_{1,2}, y_{2,3}.\theta$.* ■

Based on the reduction given in the proof above, one can also show that VAL(QPI) hits every level from the polynomial hierarchy when restrictions are put on the prefix: if no alternations of quantifiers occur, the problem is in P, if the prefix is of the form $\exists S_1\ \forall S_2$, the problem is NP-complete, if the prefix is of the form $\forall S_1\ \exists S_2\ \forall S_3$, the problem is Π_2^p-complete, and so on.

Now, what's about the restriction of VAL(QPI) for the instances for which the rightmost quantification of the prefix is existential? Contrariwise to $\text{OBDD}_<$ formulae in the general case, the negation of a PI formula cannot be computed in polynomial time (and even in *polynomial space*) as a PI formula, hence the same argument cannot be used again. Nevertheless, it is easy to show that a rightmost existential quantification does not lead to a complexity shift:

Proposition 7. *The restriction of* VAL(QPI) *to instances with prefixes of the form $\forall S_1\ \exists S_2$ is in* P.

Proof 7. *The proof is based on the fact that it is possible to eliminate in polynomial time existential quantifications as an internal law in the* PI *fragment. To be more precise, let $PI(\phi)$ be the set of prime implicates of a propositional formula ϕ (only one representative per equivalence class is kept); let us consider the following lemma (a direct consequence of Proposition 55 in [46]):*

Lemma 1. *Let ϕ be a formula from $\mathtt{PROP}_{PS}$ and let Y be a set of variables from PS. We have $PI(\exists Y.\phi) = \{\delta \in PI(\phi) \mid Var(\delta) \cap Y = \emptyset\}$.*

This lemma shows how a CNF *formula ψ over X which is equivalent to $\exists Y.\phi$ can be derived in time polynomial in $|\phi|$.*

Then we exploit the two following properties (already at work in the proof of Proposition 6): for every pair of QBF α and β and every variable $x \in PS$, we have $\forall x.(\alpha \wedge \beta) \equiv (\forall x.\alpha) \wedge (\forall x.\beta)$ and for every non tautologous clause δ (viewed as the set of its literals) and every variable $x \in PS$, $\forall x.\delta$ is equivalent to the clause $\delta \setminus \{x, \neg x\}$. A direct consequence of them is that $\forall X.\psi$ is valid if and only if ψ contains only tautologous clauses, which can be easily checked in time polynomial in $|\psi|$. □

The following example illustrates the lemma given in the above proof:

Example 15. *A* PI *formula equivalent to $\exists a.((a \vee b) \wedge (\neg b \vee c \vee d) \wedge (a \vee c \vee d))$ is $\neg b \vee c \vee d$.* ■

As a corollary to the previous proposition, we obtain that if the prefix of the input is of the form $\exists S_1\ \forall S_2\ \exists S_3$, the VAL(QPI) problem is NP-complete, if the prefix of the input is of the form $\forall S_1\ \exists S_2\ \forall S_3\ \exists S_4$, the problem is Π_2^p-complete, and so on.

The following dual class also is interesting. Let $\phi \in \mathtt{PROP}_{PS}$. A *prime implicant* of ϕ is a term γ s.t. $\gamma \models \phi$ and for every term γ' s.t. $\gamma' \models \phi$ and $\gamma \models \gamma'$, we have $\gamma \equiv \gamma'$.

Definition 9 (prime implicants formulae)**.** *A prime implicants formula from $\mathtt{PROP}_{PS}$ is a* DNF *formula Σ where every prime implicant of Σ (up to logical equivalence) appears as a disjunct.* IP *is the language of all prime implicants formulae (a proper subset of* DNF*).*

Prime implicants formulae are duals of prime implicates formulae: every prime implicant of a formula ϕ is (up to logical equivalence) the negation of a prime implicates of $\neg\phi$ (see e.g. Proposition 8 in [46]). As a consequence, QIP and QPI are also dual classes. Taking advantage of duality, we also obtain that:

Proposition 8. VAL(QIP) *is* PSPACE*-complete.*

Proof 8. *Immediate from Proposition 6 given the fact that* PSPACE *is closed under complementation and the fact that the negation of a* QPI *formula can be turned in linear time into a* QIP *formula.* □

Proposition 9. *The restriction of* VAL(QIP) *to instances with prefixes of the form $\exists S_1\ \forall S_2$ is in* P*.*

Proof 9. *This a direct consequence of Proposition 7 (by duality).* □

Exploiting duality, it is easy to show that the classes Σ_i^p and Π_i^p of the polynomial hierarchy that were not "hit" by restrictions of VAL(QPI) are "hit" by restrictions of VAL(QIP): if no alternations of quantifiers occur, the problem VAL(QIP) is in P, if the prefix of the input is of the form $\forall S_1\ \exists S_2$ or $\forall S_1\ \exists S_2\ \forall S_3$, the problem is coNP-complete, if the prefix of the input is of the form $\exists S_1\ \forall S_2\ \exists S_3$ or $\exists S_1\ \forall S_2\ \exists S_3\ \forall S_4$, the problem is Σ_2^p-complete, and so on.

4. Some Experimental Results

From the practical side, an important question is to determine how existing solvers for $\text{VAL}(\texttt{QPROP}_{PS})$ behave on instances from tractable fragments. Especially, when solvers do not solve "easily" instances from tractable fragments, finding out the reason for it can prove a major step in the improvement of existing solvers. In order to answer this question, we performed some experiments on instances from QCI and $\texttt{QOCNF}_{<}$ under the compatibility assumption, which are tractable subsets of QCNF, the class of QBF almost all existing solvers deal with.

4.1 Experimental settings

We generated $\texttt{QOCNF}_{<}$ benchmarks with compatible prefixes in the following way. Given a number of variables n, each integer in $\{1, \ldots, n\}$ is identified with a variable. We took $<$ as the natural ordering on integers. In order to generate a clause of the matrix of a $\texttt{QOCNF}_{<}$ formula, the size s of the clause is first chosen at random in $\{1, \ldots, n\}$ under a uniform distribution, then the sign of each of the s variables (from 1 to s) of the clause is chosen at random under a uniform distribution.

For our experiments, we set n to 100, 200, 300, 400, 500, 1000 variables, which produce huge benchmarks, compared to the benchmarks commonly used for evaluating state-of-the-art solvers. The number of clauses was arbitrarily fixed to four times the number of variables ($m = 4 \times n$). The prefix is generated so as to ensure the compatibility with $<$. The number of quantifier alternations was fixed to 12: 11 sets of the same size (n div 11) and a last one with the remaining (n mod 11) variables. Note that this number of alternations is also quite high compared to the usual standards in QBF benchmarks.

We used three QBF solvers for those experiments. Our own Java-based QBF solver, called OpenQBF, which participated to the three QBF evaluations and reported medium strength results, while being among the few solvers not declared incorrect during the last two QBF evaluations. We also used Semprop release 010604 and Qube-Rel 1.3, two publicly available QBF solvers. The solvers ran on a PIV 3GHz with 1.5GB of RAM under Linux. The Java solver ran on the latest Sun Java VM (1.5.0_06) using default settings.

4.2 Results

Table 1 summarizes the behaviour of the three solvers on QCI and $\texttt{QOCNF}_{<}$ instances. Note that the running times are expressed in seconds, for solving 100 instances with the same parameters. The *size* column reports the total space taken by the generated benchmarks (for a total greater than 3 GB for those experiments!). The *#true* column reports the number of positive instances of QBF out of 100. One can note that all the instances from QCI were positive. This can be explained by the fact that each clause has roughly half of its literals existentially quantified (since by construction roughly half of the variables are existentially quantified and that the clauses contain all variables).

Note that all solvers perform better on $\texttt{QOCNF}_{<}$ benchmarks than on QCI benchmarks. This is probably due to the difference in the size of the benchmarks: QCI benchmarks are twice as big as $\texttt{QOCNF}_{<}$ benchmarks. Note also that it is usually easier to solve negative instances than positive ones: most of the instances in $\texttt{QOCNF}_{<}$ are negative ones.

Table 1. Running times of three solvers on instances from two polynomial classes of QBF. Cumulative CPU time in seconds for solving 100 instances for a given number of variables.

	CI					$\texttt{OCNF}_<$ (compatible prefix)				
#vars	size	Own	Semp.	Qube	#true	size	Own	Semp.	Qube	#true
100	14	36	10	2	100	8	28	2	1	11
200	62	71	71	7	100	30	44	12	4	12
300	144	124	240	16	100	69	73	38	8	13
400	260	209	554	28	100	125	104	94	14	19
500	411	316	1062	43	100	198	145	165	21	10
1000	1680	1217	8638	174	100	824	532	1601	82	13

The three solvers were able to easily solve all benchmarks (a mean of 86 seconds per benchmark in the worst case, even if the running times ranges over one order of magnitude in that case). Note that those benchmarks are really huge compared to usual QBF benchmarks which explains why a usually robust solver such as Semprop does not scale well.

Those experiments show that the current solvers do not have problems for solving instances from QCI and $\texttt{QOCNF}_<$. This is in contrast with our previous experiments on some other tractable (but incomplete) fragments for the validity problem (quantified Horn CNF and quantified renamable Horn CNF) [48] that were confirmed during the latest QBF evaluation [29].

5. Compilability

It is interesting to note the connection between the problem of finding out tractable matrix-based restrictions of VAL($\texttt{QPROP}_{PS}$) and the compilability issue for VAL($\texttt{QPROP}_{PS}$) when matrices are fixed while prefixes may vary. A basic issue is the compilability to P of VAL($\texttt{QPROP}_{PS}$), i.e., the membership to the compilability class compP of the compilation problem COMP-VAL($\texttt{QPROP}_{PS}$) associated to QBF where each instance is divided into two parts – the fixed one is the matrix and the variable one is the prefix [49, 50].

Definition 10 (COMP-VAL($\texttt{QPROP}_{PS}$))**.** COMP-VAL($\texttt{QPROP}_{PS}$) *is the language of pairs* $\{\langle \Sigma, P\rangle \mid P.\Sigma$ *is a closed, polite, prenex formula from* $\texttt{QPROP}_{PS}$ *which is valid*$\}$.

Definition 11 (compP [50])**.** *A language of pairs* L *belongs to* compP *if and only if there exists a polysize function* f *and a language of pairs* $L' \in$ P *such that* $\langle x, y\rangle \in L$ *if and only if* $\langle f(x), y\rangle \in L'$.

Actually, the practical significance of such a compilability issue comes from the fact that, instead of considering the matrix as fixed, one can more generally consider the case when it can be computed in polynomial time from more basic inputs. Because many target classes $\mathcal{C}$ for knowledge compilation enable polytime conditioning, polytime bounded conjunction and polytime bounded disjunction [13], many inference problems can be encoded (and solved) as QBFs whose matrices in $\mathcal{C}$ can be computed in polynomial time from a compiled formula $\Sigma \in \mathcal{C}$ (the knowledge base) and a clausal query γ.

The membership of COMP-VAL($\mathtt{QPROP}_{PS}$) to compP can be expressed by the following question: can we find a complete propositional fragment $\mathcal{C}$ for which there is a polynomial $p(.)$ s.t. every propositional formula α has an equivalent $\beta \in \mathcal{C}$ satisfying $|\beta| \leq p(|\alpha|)$ and there is a polytime algorithm for VAL($\mathtt{Q}\mathcal{C}$)? Clearly, $\mathcal{C} = \mathtt{MODS}$ (or $\mathtt{CI}$) is not a satisfying answer since it is not the case that every propositional formula has polynomially many models (or canonical implicates). Actually, it seems that no such fragment $\mathcal{C}$ exists:

Proposition 10. COMP-VAL($\mathtt{QPROP}_{PS}$) *is not in* compP *unless* NP $\subseteq$ P/poly. *The conclusion holds under the restriction when at most one alternation of quantifiers occurs in the prefix of the input.*[5.]

Proof 10. *Let us first give a brief refresher about non-uniform complexity classes:*

Definition 12 (Advice-taking Turing machine)**.** *An advice-taking Turing machine is a Turing machine that has associated with it a special "advice oracle" A, which can be any function (not necessarily a recursive one). On input s, a special "advice tape" is automatically loaded with $A(|s|)$ and from then on the computation proceeds as normal, based on the two inputs, s and $A(|s|)$.*

Definition 13 (Polynomial advice)**.** *An advice-taking Turing machine uses polynomial advice if its advice oracle A satisfies $|A(n)| \leq p(n)$ for some fixed polynomial p and all non-negative integers n.*

Definition 14 ($\mathcal{C}$/poly)**.** *If $\mathcal{C}$ is a class of languages defined in terms of resource-bounded Turing machines, then $\mathcal{C}$/poly is the class of languages defined by Turing machines with the same resource bounds but augmented by polynomial advice.*

We show that the NP*-complete* 3-CNF-SAT *problem can be solved in (deterministic) polynomial time using a Turing machine with polynomial advice if* COMP-VAL($\mathtt{QPROP}_{PS}$) *belongs to* compP*. For each integer n, let Φ_n be the* CNF *formula (viewed as a set of clauses) containing all the clauses of the form $y_i \vee \delta_i$ where δ_i is a clause with at most* 3 *literals built up from the set X_n of variables $x_1, \ldots, x_n$ and each y_i is a new symbol (one for each clause δ_i). Clearly enough, Φ_n (up to logical equivalence) depends only on n. Furthermore, the number p_n of clauses in Φ_n is s.t. $p_n \in \mathcal{O}(n^3)$, hence the size of Φ_n is polynomial in n.*

Let ϕ_n be any CNF *formula over X_n. By construction, each clause δ_i from ϕ_n corresponds to a unique clause $y_i \vee \delta_i$ from Φ_n. Let ψ_n be the complement of $\{y_i \vee \delta_i \mid \delta_i \in \phi_n\}$ in Φ_n. One can easily compute in polynomial time from ϕ_n the set Y_{ϕ_n} of variables y_i s.t. the corresponding clauses δ_i belong to ϕ_n. Now, we know that for every pair of QBFs α and β and every variable $y \in PS$, we have $\forall y.(\alpha \wedge \beta) \equiv (\forall y.\alpha) \wedge (\forall y.\beta)$ (point 5 in Proposition 1); for every clause δ (viewed as the set of its literals) and every variable $y \in PS$, if δ does not contain both y and $\neg y$, then $\forall y.\delta$ is equivalent to the clause $\delta \setminus \{y, \neg y\}$ (as a direct consequence of the definition of conditioning and point 1 in Proposition 1); and if the variable y does not occur in the formula α, we have $\forall y.\alpha \equiv \alpha$ (an easy consequence of point 4 in Proposition 1); From those three properties, we get immediately that $\forall Y_{\phi_n}.\Phi_n \equiv \phi_n \wedge \psi_n$.*

5. Contrastingly, the restriction of COMP-VAL($\mathtt{QPROP}_{PS}$) to instances where no alternation occurs obviously is in compP, just because there are only two possible prefixes for each matrix in this case (one is composed of existential quantifications only, and the other one of universal quantifications only).

Since every clause $y_i \vee \delta_i$ from ψ_n contains the literal y_i which is pure in $\phi_n \wedge \psi_n$ (i.e., its negation does not occur), it follows that $\phi_n \wedge \psi_n$ is satisfiable if and only if ϕ_n is satisfiable. Hence, ϕ_n is satisfiable if and only if $\forall Y_{\phi_n}.\Phi_n$ is satisfiable if and only if $\exists X_n\ \forall Y_{\phi_n}.\Phi_n$ is valid. Finally, let us assume that there exists a complete propositional fragment $\mathcal{C}$ for which VAL($\mathtt{Q}\mathcal{C}$) *is tractable and for which there is a polynomial $p(.)$ s.t. every propositional formula α has an equivalent $\beta \in \mathcal{C}$ satisfying $\beta \leq p(|\alpha|)$. Then* 3-CNF-SAT *can be solved in polynomial time by a deterministic Turing machine with advice as follows: for each input ϕ_n, the advice for n first gives a polyspace formula $\Phi_{n,\mathcal{C}}$ from $\mathcal{C}$ which is equivalent to Φ_n and then the satisfiability of ϕ_n is decided in polynomial time as the validity of $\exists X_n\ \forall Y_{\phi_n}.\Phi_{n,\mathcal{C}}$.* □

Note that NP $\subseteq$ P/poly would imply the polynomial hierarchy to collapse at the second level [51], which is considered very unlikely.

6. Conclusion

In this paper, we have presented new tractability and new intractability results for the validity problems for QBFs when the matrices of the inputs belong to a target class for knowledge compilation. In the light of our study, the complexity landscape for VAL($\mathtt{QPROP}_{PS}$) can be completed as reported in Table 2.

Fragment $\mathcal{C}$	**Complexity of** VAL($\mathtt{Q}\mathcal{C}$)
$\mathtt{PROP}_{PS}$ (general case)	PSPACE-c
CNF	PSPACE-c
DNF	PSPACE-c
d-DNNF	PSPACE-c
sd-DNNF	PSPACE-c
DNNF	PSPACE-c
FBDD	PSPACE-c
$\mathtt{OBDD}_<$	PSPACE-c
PI	PSPACE-c
IP	PSPACE-c
$\mathtt{OBDD}_<$ (compatible prefix)	$\in$ P
$\mathtt{ODNF}_<$ (compatible prefix)	$\in$ P
$\mathtt{OCNF}_<$ (compatible prefix)	$\in$ P
MODS	$\in$ P
CI	$\in$ P

Table 2. Complexity results for VAL($\mathtt{QPROP}_{PS}$).

In [13], the authors have investigated the spatial efficiency of several complete propositional fragments, including many of those considered in this paper. A given fragment $\mathcal{C}_1$ is considered at least as concise as a second fragment $\mathcal{C}_2$ whenever there exists a polynomial $p(.)$ s.t. for every formula $\alpha \in \mathcal{C}_2$, there exists an equivalent formula $\beta \in \mathcal{C}_1$ s.t. $|\beta| \leq p(|\alpha|)$.

Our results show that VAL($\texttt{QPROP}_{PS}$) is difficult even when limited to instances with matrices from fragments which are not efficient from the spatial point of view (i.e., the $\texttt{OBDD}_<$ one, the PI fragment and the IP fragment). Tractability is achieved without restrictions only for the MODS fragment and the CI fragment which are among the least efficient ones (as to spatial efficiency) (see [13]). Under the compatibility assumption, tractability is also achieved for the more concise $\texttt{OBDD}_<$, $\texttt{ODNF}_<$ and $\texttt{OCNF}_<$ fragments; those fragments appear as the best candidates among the classes considered in this paper which enable tractable QBF queries; however, the choice of the ordering $<$ has a major impact on the size of the formulae (see [39] for the $\texttt{OBDD}_<$ case).

This work calls for several perspectives. One of them consists in further extending the complexity landscape of VAL($\texttt{QPROP}_{PS}$), focusing on other complete or incomplete fragments. In particular, it would be interesting to determine how the notion of K-Boolean model from [52], in the case when K is a class of boolean functions which can be encoded in polynomial space, could be exploited to give rise to new restrictions of VAL($\texttt{QPROP}_{PS}$) which are computationally easier (under the usual assumptions of complexity theory).

While the focus has been laid on matrix-based restrictions of VAL($\texttt{QPROP}_{PS}$) in this paper, it might seem quite natural at a first glance to consider prefix-based restrictions as well. However, from a theoretical point of view, considering restrictions on the prefix alone (i.e., without further requirements on the matrix like we did it before when considering the $\texttt{OBDD}_<$ fragment, the $\texttt{OCNF}_<$ fragment, the $\texttt{ODNF}_<$ fragment, the prime implicates fragment and the prime implicants fragment) does not look so much interesting. While limiting the number of quantifications alternations in the prefix leads immediately the complexity of VAL($\texttt{QPROP}_{PS}$) to decrease from PSPACE to a given level in the polynomial hierarchy, it does not lead to tractability (under the standard assumptions of complexity theory); indeed, in the limit case, no quantification alternations occur in the prefix so that the VAL($\texttt{QPROP}_{PS}$) problem reduces either to SAT or to UNSAT (depending of the nature of the quantification of all variables), and none of these problems is likely to belong to P provided that no assumption on the matrix is made. Of course, this does not mean that the way quantifications are processed has no practical impact on the efficiency of solvers for VAL($\texttt{QPROP}_{PS}$). For instance, QBFs Σ of the form $\forall\{x_1,\ldots,x_{n-1}\}\exists\{x_n\}.\phi$ where ϕ is a CNF formula can be solved in quadratic time (compute a CNF representation of $\exists\{x_n\}.\phi$ using resolution to forget x_n within ϕ, remove tautologous clauses, then shorten every resulting clause by removing from it every literal built up from $\{x_1,\ldots,x_{n-1}\}$; Σ is valid if and only if the resulting CNF formula does not contain the empty clause). However, solvers for VAL($\texttt{QPROP}_{PS}$) where quantifications are processed from the outermost to the innermost may require an exponential amount of time to solve some Σ of this kind. A deeper investigation of the interaction between prefix-based restrictions and matrix-based ones is a perspective for further research.

Acknowledgments

Many thanks to the Région Nord / Pas-de-Calais through the IRCICA Consortium and the COCOA project, and to the IUT de Lens for their support. Special thanks to the anonymous reviewers for their very constructive remarks and suggestions (especially for suggesting the tractability of VAL($\texttt{QODNF}_<$) and VAL($\texttt{QOCNF}_<$) under the compatibility assumption).

Appendix

Figure 5 depicts the inclusion graph of the propositional fragments considered in this paper.

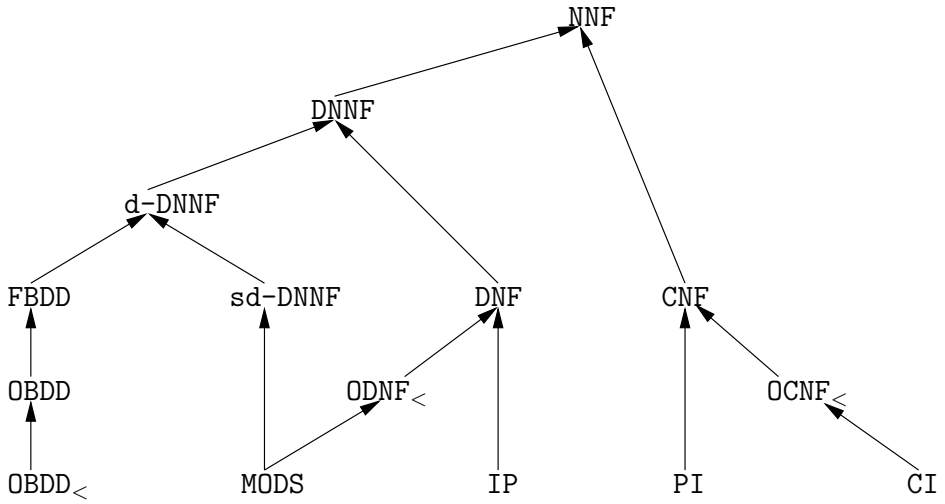


Figure 5. Inclusion graph of propositional fragments. An edge $L_1 \rightarrow L_2$ means that L_1 is a proper subset of L_2.

References

[1] B. Selman and H. Kautz. Knowledge compilation and theory approximation. *Journal of the ACM*, **43**:193–224, 1996.

[2] A. Del Val. Tractable databases: how to make propositional unit resolution complete through compilation. In *KR'94*, pages 551–561, 1994.

[3] R. Schrag. Compilation for critically constrained knowledge bases. In *AAAI'96*, pages 510–515, 1996.

[4] Y. Boufkhad, E. Grégoire, P. Marquis, B. Mazure, and L. Saïs. Tractable cover compilations. In *IJCAI'97*, pages 122–127, 1997.

[5] S. Coste-Marquis and P. Marquis. Knowledge compilation for circumscription and closed world reasoning. *Journal of Logic and Computation*, **11**(4):579–607, 2001.

[6] S. Coste-Marquis and P. Marquis. On stratified belief base compilation. *Annals of Mathematics and Artificial Intelligence*, **42**(4):399–442, 2004.

[7] A. Darwiche and P. Marquis. Compiling propositional weighted bases. *Artificial Intelligence*, **157**(1–2):81–113, 2004.

[8] S. Coste-Marquis and P. Marquis. Characterizing consistency-based diagnoses. In *AI & Math'98*, 1998.
http://rutcor.rutgers.edu/~amai/aimath98.

[9] A. Darwiche. Compiling devices into decomposable negation normal form. In *IJCAI'99*, pages 284–289, 1999.

[10] A. Cimmati, E. Giunchiglia, F. Giunchiglia, and P. Traverso. Planning via model checking: a decision procedure for AR. In *ECP'97*, pages 130–142, 1997.

[11] H. Geffner. Planning graphs and knowledge compilation. In *KR'04*, pages 662–672, 2004.

[12] H. Palacios, B. Bonet, A. Darwiche, and H. Geffner. Pruning conformant plans by counting models on compiled d-DNNF representations. In *ICAPS'05*, pages 141–150, 2005.

[13] A. Darwiche and P. Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, **17**:229–264, 2002.

[14] U. Egly, T. Eiter, H. Tompits, and S. Woltran. Solving advanced reasoning tasks using Quantified Boolean Formulas. In *AAAI'00*, pages 417–422, 2000.

[15] H. Fargier, J. Lang, and P. Marquis. Propositional logic and one-stage decision making. In *KR'00*, pages 445–456, 2000.

[16] J. Rintanen. Constructing conditional plans by a theorem-prover. *Journal of Artificial Intelligence Research*, **10**:323–352, 1999.

[17] G. Pan, U. Sattler, and M.Y. Vardi. BDD-based decision procedures for K. In *CADE'02*, pages 16–30, 2002.

[18] Ph. Besnard, T. Schaub, H. Tompits, and S. Woltran. *Inconsistency tolerance*, volume **3300** of *LNCS State-of-the-Art Survey*, chapter Representing paraconsistent reasoning via quantified propositional logic, pages 84–118. Springer-Verlag, 2005.

[19] M. Cadoli, A. Giovanardi, and M. Schaerf. An algorithm to evaluate Quantified Boolean Formulae. In *AAAI'98*, pages 262–267, 1998.

[20] J. Rintanen. Improvements to the Evaluation of Quantified Boolean Formulae. In *IJCAI'99*, pages 1192–1197, 1999.

[21] R. Feldmann, B. Monien, and S. Schamberger. A distributed algorithm to evaluate quantified boolean formulas. In *AAAI'00*, pages 285–290, 2004.

[22] E. Giunchiglia, M. Narizzano, and A. Tacchella. Backjumping for Quantified Boolean Logic satisfiability. In *IJCAI'01*, pages 275–281, 2001.

[23] R. Letz. Lemma and model caching in decision procedures for Quantified Boolean Formulas. In *Tableaux'02*, pages 160–175, 2002.

[24] L. Zhang and S. Malik. Towards a symmetric treatment of satisfaction and conflicts in quantified boolean formula evaluation. In *CP'02*, pages 200–215, 2002.

[25] G. Pan and M.Y. Vardi. Symbolic decision procedures for QBF. In *CP'04*, pages 453–467, 2004.

[26] G. Audemard and L. Saïs. SAT based BDD solver for Quantified Boolean Formulas. In *ICTAI'04*, pages 82–89, 2004.

[27] D. Le Berre, L. Simon, and A. Tacchella. Challenges in the QBF arena: the SAT'03 evaluation of QBF solvers. In *SAT'03*, volume **2919** of *LNCS*, pages 468–485, 2003.

[28] D. Le Berre, M. Narizzano, L. Simon, and A. Tacchella. The second QBF solvers evaluation. In *SAT'04*, volume **3542** of *LNCS*, pages 376–392, 2004.

[29] M. Narizzano, L. Pulina, and A. Tacchella. The third QBF solvers comparative evaluation. *Journal on Satisfiability, Boolean Modeling and Computation*, **2**:145–164, 2006.

[30] Th. J. Schaefer. The complexity of satisfiability problems. In *STOC'78*, pages 216–226, 1978.

[31] N. Creignou, S. Khanna, and M. Sudan. Complexity classification of boolean constraint satisfaction problems. In *SIAM Monographs on Discrete Mathematics and Applications*, volume **7**. Society for Industrial and Applied Mathematics, 2001.

[32] B. Aspvall, M. Plass, and R. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, **8**:121–123, 1979. Erratum: Information Processing Letters 14(4): 195 (1982).

[33] H. Kleine Büning, M. Karpinski, and A. Flögel. Resolution for quantified boolean formulas. *Information and Computation*, **117**(1):12–18, 1995.

[34] I. P. Gent and A. G. D. Rowley. Solving 2-CNF Quantified Boolean Formulae using variable assignment and propagation. In *QBF workshop at SAT'02*, pages 17–25, 2002.

[35] F. Börner, A. Bulatov, A. Krokhin, and P. Jeavons. Quantified constraints: algorithms and complexity. In *CSL'03*, volume **2803** of *LNCS*, pages 58–70, 2003.

[36] H. Chen. Collapsibility and consistency in quantified constraint satisfaction. In *AAAI'04*, pages 155–160, 2004.

[37] H. Chen. Quantified constraint satisfaction, maximal constraint languages, and symmetric polymorphisms. In *STACS'05*, pages 315–326, 2005.

[38] Ch. H. Papadimitriou. *Computational complexity*. Addison–Wesley, 1994.

[39] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, C-**35**(8):677–692, 1986.

[40] U. Egly, M. Seidl, H. Tompits, S. Woltran, and M. Zolda. Comparing different prenexing strategies for Quantified Boolean Formulas. In *SAT'03*, volume **2919** of *LNCS*, pages 214–228, 2003.

[41] A. Darwiche. Decomposable negation normal form. *Journal of the ACM*, **48**(4):608–647, 2001.

[42] J. Gergov and C. Meinel. Efficient analysis and manipulation of OBDDs can be extended to FBDDs. *IEEE Trans. on Computers*, **43**(10):1197–1209, 1994.

[43] X. Zhao and H. Kleine Büning. Model-equivalent reductions. In *SAT'05*, volume **3569** of *LNCS*, pages 355–370, 2005.

[44] D. Sieling and I. Wegener. Reduction of OBDDs in linear time. *Information Processing Letters*, **48**(3):139–144, 1993.

[45] J. Lang, P. Liberatore, and P. Marquis. Propositional independence - formula-variable independence and forgetting. *Journal of Artificial Intelligence Research*, **18**:391–443, 2003.

[46] P. Marquis. *Consequence finding algorithms*, volume **5** of *Handbook on Defeasible Reasoning and Uncertainty Management Systems*, chapter 2, pages 41–145. Kluwer Academic Publisher, 2000.

[47] P. Tison. Generalization of consensus theory and application to the minimization of boolean functions. *IEEE Trans. on Electronic Computers*, EC–**16**:446–456, 1967.

[48] S. Coste-Marquis, D. Le Berre, and F. Letombe. A branching heuristics for quantified renamable Horn formulas. In *SAT'05*, volume **3569** of *LNCS*, pages 393–399, 2005.

[49] P. Liberatore. Monotonic reductions, representative equivalence, and compilation of intractable problems. *Journal of the ACM*, **48**(6):1091–1125, 2001.

[50] M. Cadoli, F.M. Donini, P. Liberatore, and M. Schaerf. Preprocessing of intractable problems. *Information and Computation*, **176**(2):89–120, 2002.

[51] R. M. Karp and R. J. Lipton. Some connections between non-uniform and uniform complexity classes. In *STOC'80*, pages 302–309, 1980.

[52] H. Kleine Büning, K. Subramani, and X. Zhao. On boolean models for quantified boolean Horn formulas. In *SAT'03*, volume **2919** of *LNCS*, pages 93–104, 2003.

Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT®

Artificial Intelligence 153 (2004) 339–371

Artificial Intelligence

www.elsevier.com/locate/artint

Weakening conflicting information for iterated revision and knowledge integration ☆

Salem Benferhat [a,*], Souhila Kaci [a], Daniel Le Berre [a], Mary-Anne Williams [b]

[a] *Centre de Recherche en Informatique de Lens (C.R.I.L.), Université d'Artois, Rue Jean Souvraz, 62300 Lens, France*

[b] *Business and Technology Research Laboratory, The University of Newcastle, Newcastle NSW 2308, Australia*

Received 15 January 2001; received in revised form 18 August 2003

Abstract

The ability to handle exceptions, to perform iterated belief revision and to integrate information from multiple sources is essential for a commonsense reasoning agent. These important skills are related in the sense that they all rely on resolving inconsistent information. In this paper we develop a novel and useful strategy for conflict resolution, and compare and contrast it with existing strategies. Ideally the process of conflict resolution should conform with the principle of Minimal Change and should result in the *minimal loss* of information. Our approach to minimizing the loss of information is to weaken information involved in conflicts rather than completely discarding it. We implemented and tested the relative performance of our new strategy in three different ways. Surprisingly, we are able to demonstrate that it provides a computationally effective compilation of the lexicographical strategy; a strategy which is known to have desirable theoretical properties.

Keywords: Belief revision; Compilation

☆ This paper is an extended version of [S. Benferhat et al., Weakening conflicting information for iterated revision and knowledge integration, in: Proc. IJCAI-01, Seattle, WA, 2001, pp. 109–115].

* Corresponding author.

E-mail addresses: benferhat@cril.univ-artois.f (S. Benferhat), kaci@cril.univ-artois.fr (S. Kaci), leberre@cril.univ-artois.f (D. Le Berre), maryanne@it.uts.edu.au (M.-A. Williams).

0004-3702/$ – see front matter
doi:10.1016/j.artint.2003.08.003

1. Introduction

Information modeling and management is a fundamental activity in commonsense reasoning. Commonsense reasoning agents require robust and sophisticated information management capabilities for exception handling, iterated revision and the integration of information.

It is well known that in order to make effective commonsense inferences reasoning agents must be capable of making decisions with incomplete information with the aid of things like default rules, and to be able to handle inconsistencies with exceptions when they arise.

Usually this decision involves choices because there is typically more than one way to resolve the conflict.

In this paper we develop a novel and useful strategy for conflict resolution which can be applied to iterated belief revision and information integration.

Belief revision [1,14,17,20] consists in incorporating a new information into a consistent knowledge base. This involves identifying conflicts whenever the new information is inconsistent with the existing information in the knowledge base. Information integration is understood here as the process of amalgaming a set of knowledge bases coming from different sources into a unique knowledge base [2,11]. Even if each source provides a consistent knowledge base, it is unlikely that their fusion results in a consistent knowledge base. This paper more focuses on belief revision. However, the techniques proposed can be applied to information integration as well.

Throughout we assume that the available information is given as ordered knowledge bases, i.e., a ranking of information as logical sentences. Solving conflicts in our context means computing a consistent knowledge base. One well known system that can deal with conflicts in knowledge bases is the so-called *Adjustment* procedure [29]. In essence, Adjustment propagates as many highly ranked formulas as possible, and ignores information at and below the highest rank where an inconsistency is found. The main advantage of this system is its computational efficiency. For example, it only needs at most $\mathrm{Log}_2 n$ calls to a SAT solver [19,28] to build the consistent knowledge base where n is the number of ranks in the knowledge base. The obvious disadvantage of Adjustment, however, is that it can remove more formulas than is necessary to restore the consistency of the knowledge base if the independence of information is not made explicit. In order to overcome this shortcoming another strategy called *Maxi-Adjustment* was introduced [26] and implemented [28]. Maxi-Adjustment has proved to be a useful strategy for real world applications, e.g., software engineering [27], information filtering [10] and intelligent payment systems [31]. The main idea of Maxi-Adjustment is to solve conflicts at each rank of priority in the knowledge base. This is done, incrementally, starting from the information with highest rank. When inconsistency is encountered in the knowledge base, then all formulas in the rank responsible for the conflicts are removed. The other formulas are kept, and the process continues to the next rank.

Clearly Maxi-Adjustment keeps more information than Adjustment, since it does not stop at the first rank where inconsistency is met. Even though Maxi-Adjustment propagates more information than Adjustment, one can still argue that Maxi-Adjustment removes too

much information because it adopts a sceptical approach to the way it removes the conflict sets at each rank.

The purpose of this paper is to describe a significant improvement to Maxi-Adjustment. We call this system *Disjunctive Maxi-Adjustment*, and denote it by DMA. The idea is similar to Maxi-Adjustment, except that conflicting information is weakened instead of being removed. So instead of removing all formulas involved in conflicts, as it is done in Maxi-Adjustment, DMA takes their disjunctions pairwise. If the result is consistent, then we move to the next rank. If the result is still inconsistent, then we replace the formulas in conflicts by all possible disjunctions involving three formulas in the conflict sets and again if the result is consistent we move to the next layer, and if it is inconsistent we consider disjunctions of size 4, 5, etc. The only case where all formulas responsible for conflicts are removed is when the disjunction of all these formulas is inconsistent with the higher priority information.

This paper focuses on the DMA strategy from the theoretical and experimental perspectives. In particular,

- We show that DMA is equivalent to the well known lexicographical strategy [3,7,8, 12,20,21]. More precisely, we show that for an inconsistent base K if $\delta_{DMA}(K)$ is the propositional base obtained using DMA, and $\delta_{Lex}(K)$ is the set of all lexicographically maximal consistent subbases of K, then:

  $$\forall \psi,\ \delta_{DMA}(K) \vdash \psi \quad \text{iff} \quad \forall A \in \delta_{Lex}(K),\ A \vdash \psi.$$

  In other words, we obtain the surprising and computationally useful result that DMA provides a "compilation" of lexicographical systems.
- It is well known that computing conflicts is a hard task [6], and we are able to show that DMA works even if the conflicts are not explicitly computed. For this we propose an alternative, but equivalent, approach to DMA called whole-DMA where disjunctions are built on the whole stratum when we meet inconsistency instead of only on the conflicts.
- We also propose another equivalent alternative to DMA called iterative-DMA where instead of considering disjunctions of size (3, 4, etc.) on the initial set of conflicts, we only compute disjunctions of size 2 but on new sets of conflicts.
- Lastly, we compare these different implementations of DMA experimentally, and contrast their applicability.

All the proofs of technical results are given in Appendix A.

2. Ordered information in Spohn's OCF framework

We consider a finite propositional language denoted by $\mathcal{L}$. Let Ω be the set of interpretations. $\vdash$ denotes the classical consequence relation, Greek letters $\phi, \psi, \ldots$ represent formulas.

We use Spohn's Ordinal Conditional Function [25] framework, which is also known as the Kappa function framework.

At the semantic level, the basic notion of Spohn's ordinal conditional function framework is a function called an OCF, denoted by κ, which is a mapping from Ω to $\mathbb{N} \cup \{+\infty\}$ ($\mathbb{N}$ being the set of natural numbers), such that $\exists \omega \in \Omega, \kappa(\omega) = 0$.[1] $\kappa(\omega)$ can be viewed as the degree of impossibility of ω.

By convention, $\kappa(\omega) = 0$ means that nothing prevents ω from being the real world, and $\kappa(\omega) = +\infty$ means that ω is certainly not the real world.[2] The lower $\kappa(\omega)$ is, the more expected it is, i.e., if $\kappa(\omega) < \kappa(\omega')$ then ω is said to be more plausible than ω'.

In practice, OCF over all possible worlds are not available, however a ranked knowledge base provides a compact representation of an OCF [29].

Since we will be working with ranked knowledge bases throughout, we define a knowledge base to be ranked. In particular, a knowledge base is a set of weighted formulas of the form $K = \{(\phi_i, k_i)\colon i = 1, \ldots, n\}$, called κ-ranked base, where ϕ_i is a propositional formula and k_i is a positive number representing the level of priority of ϕ_i. The higher k_i, the more important the formula ϕ_i. In Section 3.1, we will also represent a κ-ranked base in a stratified way.

Given K, we can generate a unique OCF, denoted by κ_K, such that all the interpretations satisfying all the formulas in K will have the lowest value, namely 0, and the other interpretations will be ranked with respect to the highest formulas that they falsify. Namely:

Definition 1. $\forall \omega \in \Omega$,

$$\kappa_K(\omega) = \begin{cases} 0 & \text{if } \forall (\phi_i, k_i) \in K, \omega \models \phi_i, \\ \max\{k_i\colon (\phi_i, k_i) \in K \text{ and } \omega \not\models \phi_i\} & \text{otherwise.} \end{cases}$$

Then, given κ_K associated with a κ-ranked base K, the models of K are the interpretations ω such that $\kappa_K(\omega) = 0$. Note that if K is inconsistent then κ_K is subnormalized (there is no ω such that $\kappa_K(\omega) = 0$).

Example 1. Let $K = \{(\neg p \vee q, 3), (q, 1)\}$. The set of interpretations is $\{\omega_0 : \neg p \wedge \neg q, \omega_1 : \neg p \wedge q, \omega_2 : p \wedge \neg q, \omega_3 : p \wedge q\}$. Then, $\kappa_K(\omega_1) = \kappa_K(\omega_3) = 0$, $\kappa_K(\omega_0) = 1$ and $\kappa_K(\omega_2) = 3$.

The interpretations ω_1 and ω_3 are the preferred ones since they satisfy all formulas of K. They are called models of K. The interpretation ω_0 is preferred to ω_2 since the highest formula falsified by ω_0 (i.e., $(q, 1)$) is less preferred to the highest formula falsified by ω_2 (i.e., $(\neg p \vee q, 3)$).

The following defines subsumed formulas in a κ-ranked base:

Definition 2. Let K be a κ-ranked base and (ϕ, k) be a formula in K. Then, (ϕ, k) is said to be subsumed if $K_{\geqslant k} - \{(\phi, k)\} \vdash K$, where $K_{\geqslant k} = \{(\psi_i, k_i)\colon (\psi_i, k_i) \in K \text{ and } k_i \geqslant k\}$.

[1] The condition of existence of ω such that $\kappa(\omega) = 0$ is called the normalization condition. An OCF is said to be subnormalized, if there is no such ω. Subnormalized OCF encodes inconsistent beliefs.

[2] Note that the notion of impossible worlds ($+\infty$) does not exist in original works of Spohn.

The following lemma shows that subsumed formulas in K can be removed without changing the OCF associated to K:

Lemma 1. *Let K be a κ-ranked base and (ϕ, k) be a subsumed formula in K. Let $K' = K - \{(\phi, k)\}$. Let κ_K and $\kappa_{K'}$ be the OCF associated to K and K' respectively following Definition* 1. *Then,*

$$\forall \omega, \quad \kappa_K(\omega) = \kappa_{K'}(\omega).$$

Example 1 (*continued*). Let us consider again Example 1. Let K' be a ranked knowledge base obtained from K by adding the formula $(\neg p \vee q, 2)$, namely $K' = \{(\neg p \vee q, 3), (q, 1), (\neg p \vee q, 2)\}$. Let us compute $\kappa_{K'}$. Note that $(\neg p \vee q, 2)$ is a subsumed formula in K'. We have by definition: $\kappa_{K'}(\omega_1) = \kappa_{K'}(\omega_3) = 0$ (ω_1 and ω_3 are models of K'). $\kappa_{K'}(\omega_0) = 1$ (ω_0 only falsifies $(q, 1)$). $\kappa_{K'}(\omega_2) = \max(3, 2, 1) = 3$ (ω_2 falsifies all formulas in K').

Clearly, $\kappa_{K'}(\omega) = \kappa_K(\omega)$ for all ω.

3. Adjustment and maxi-adjustment

3.1. Stratified vs κ-ranked knowledge base

Until now, ranked information is represented by means of κ-ranked bases of the form $K = \{(\phi_i, k_i)\colon i = 1, \ldots, n\}$. We sometimes also represent a κ-ranked base K in a stratified form [2,9,22] as follows: $K = (S_1, \ldots, S_n)$ where S_i $(i = 1, \ldots, n)$ is a stratum containing propositional formulas of K having the same rank and which are more reliable than formulas of the stratum S_j for $j > i$. So the lower the stratum, the higher the rank.

In this paper, we only work on formulas (ϕ_i, k_i) such that $k_i \neq \infty$. The extension to considering such formulas can be done by putting them in a new stratum S_0. As we will see later, S_0 will contain the new information added to the knowledge base in the context of belief revision.

In this representation, subbases are also stratified. That is, if A is a subbase of $K = (S_1, \ldots, S_n)$, then $A = (A_1, \ldots, A_n)$ such that $A_j \subseteq S_j$, $j = 1, \ldots, n$ (A_j may be empty).

Conversely, we can represent a stratified base $K = (S_1, \ldots, S_n)$ by means of a κ-ranked base by associating formulas of each strata S_i to the same rank k_i. These ranks should be such that $k_1 > \cdots > k_n$.

It is clear that a κ-ranked base induces a unique stratified base, while the converse is false.

Given a κ-ranked base K, we define the inconsistency rank of K, denoted by $Inc(K)$, as the maximal rank in K where inconsistency is met. Namely,

$$Inc(K) = \max\bigl\{k_i\colon (\phi_i, k_i) \in K \text{ and } K_{\geqslant k_i} \text{ is inconsistent}\bigr\},$$

where $K_{\geqslant k_i}$ is the set of formulas in K whose weight is at least equal to k_i. If K is consistent, then we put $Inc(K) = 0$.

When K is given in a stratified form, i.e., $K = (S_1, \ldots, S_n)$ then the inconsistency rank gets the rank of the most prioritized stratum where we meet inconsistency. Namely,

$$Inc(K) = i \quad \text{iff} \quad S_1 \cup \cdots \cup S_{i-1} \text{ is consistent and } S_1 \cup \cdots \cup S_i \text{ is inconsistent.}$$

Also, $Inc(K) = 0$ iff $S_1 \cup \cdots \cup S_n$ is consistent.

Let us now introduce the notion of conflicts and kernel which will prove useful in the subsequent discussions:

Definition 3. Let $K = (S_1, \ldots, S_n)$ be a stratified base. A conflict in K, denoted by C, is a set of formulas of K such that:

- $C \vdash \bot$ (inconsistency),
- $\forall \phi, \phi \in C$, $C - \{\phi\} \nvdash \bot$ (minimality).

Definition 4. Let $\mathcal{C}$ be the set of all possible conflicts in K. We define the kernel of K, denoted by $kernel(K)$, as the set of formulas of K which are involved in at least one conflict in $\mathcal{C}$, i.e., $kernel(K)$ is the union of all conflicts in K.

Formulas in K which are not involved in any conflict in K are called *free* formulas.

3.2. The problem

Our aim in this paper is to address the problem of identifying conflicts for the purpose of drawing plausible inferences from inconsistent knowledge bases, iterated revision and information integration. Our technique for resolving conflicts can be used: (i) to build a transmutation for iterated belief revision [29] where the new information can be incorporated into any rank, and (ii) for theory extraction [28] which provides a natural and puissant mechanism for merging conflicting information. Without loss of generality we focus on a particular case of belief revision where some new consistent information φ is added to some consistent κ-ranked knowledge base K. Our approach can be applied to information integration, by letting φ equal to the tautology, and assuming that a κ-ranked base is the result of amalgaming several consistent knowledge bases issued from different sources. One of the main differences between belief revision and knowledge integration is that in belief revision the knowledge is assumed to be consistent and hence all conflicts in K_φ contain φ (the new added formula), while in knowledge integration one may have two independent conflicts. Given a consistent κ-ranked knowledge base K, and a new consistent formula φ we compute $\delta(K \cup \{(\varphi, +\infty)\})$, the consistent set of propositional formulas in $K \cup \{(\varphi, +\infty)\}$. Then, ψ is said to be a plausible consequence of $K \cup \{(\varphi, +\infty)\}$ iff $\delta(K \cup \{(\varphi, +\infty)\}) \vdash \psi$. In the rest of this paper we simply write K_φ instead of $K \cup \{(\varphi, +\infty)\}$. In a stratified form we write $(S_0, S_1, \ldots, S_n)$ where $S_0 = \{\varphi\}$. We briefly recall two important methods to compute $\delta(K \cup \{(\varphi, +\infty)\})$: *Adjustment* and *Maxi-Adjustment*. We will illustrate them using a simple example. We point the reader to [26,29] for more details.

```
Data: a stratified knowledge base K = (S_1, ..., S_n);
      a new sure formula: φ
Result: a consistent propositional base δ_A(K_φ)
begin
    KB ← {φ};
    i ← 1;
    while (i ⩽ n and KB ∪ S_i is consistent) do
        KB ← KB ∪ S_i;
        i ← i + 1
    return KB
end
```

Algorithm 1. ADJUSTMENT(K, φ).

3.3. Adjustment

From a syntactical point of view, the idea of Adjustment is to start with formulas having the highest rank in K_φ and to add as many prioritized formulas as possible while maintaining consistency. We stop at the highest rank (or the lowest stratum) where we meet inconsistency. Formally, we have Algorithm 1.

The selected base will be denoted by $\delta_A(K_\varphi)$.

When the algorithm stops, the last value of the indice i is the inconsistency rank of K_φ. It is defined as the highest rank i such that $\{\varphi\} \cup S_1 \cup \cdots \cup S_i$ is inconsistent. When $i = n+1$ in the above algorithm, this simply means that $K \cup \{(\varphi, +\infty)\}$ is consistent.

Example 2. Let $K = (S_1, S_2, S_3)$ be such that

$$S_1 = \{\neg a \vee \neg b \vee c, \neg d \vee c, \neg e \vee c\},$$

$$S_2 = \{d, e, f, \neg f \vee \neg g \vee c\} \quad \text{and} \quad S_3 = \{a, b, g, h\}.$$

Let $\varphi = \neg c$. Let us apply Algorithm 1. First, we have $\delta_A(K_{\neg c}) = \{\neg c\}$. There is no conflict in $\delta_A(K_{\neg c}) \cup S_1$ then

$$\delta_A(K_{\neg c}) \leftarrow \{\neg c, \neg a \vee \neg b \vee c, \neg d \vee c, \neg e \vee c\}.$$

Now, S_2 contradicts $\delta_A(K_{\neg c})$ due to the conflicts $\{d, \neg d \vee c, \neg c\}$ and $\{e, \neg e \vee c, \neg c\}$. This means that the inconsistency rank of K_φ is equal to 2. Then, we do not add the stratum S_2 and the computation of $\delta_A(K_{\neg c})$ is achieved, and we get $\delta_A(K_{\neg c}) = \{\neg c, \neg a \vee \neg b \vee c, \neg d \vee c, \neg e \vee c\}$.

Note that $\delta_A(K_{\neg c}) \not\vdash h$, even if h is not involved in any conflict in $K_{\neg c}$.

Note that a more efficient binary search based algorithm which only needs $\text{Log}_2\, n$ consistency checks has been developed and implemented[3] [28]. See also [19] for a similar algorithm in the framework of possibilistic logic. Note that the process of selecting the

[3] http://cafe.newcastle.edu.au/systems/saten.html.

consistent base using the Adjustment for new pieces of information placed in the highest rank is identical to that used in possibilistic logic [15].

One can easily see that applying adjustment is not a completely satisfactory way to deal with the inconsistency since formulas with rank lower than $Inc(K_\varphi)$ are ignored even if they are consistent with the selected base.

A formula ψ is said to be an Adjustment consequence of K_φ, denoted by $K_\varphi \vdash_A \psi$, if $\delta_A(K_\varphi) \vdash \psi$. One important property of Adjustment is that it is semantically well defined. More precisely, we have the following soundness and completeness result [30]:

$$K_\varphi \vdash_A \psi \quad \text{iff} \quad \forall \omega \in Pref(\kappa_K, \varphi),\ \omega \models \psi,$$

where $Pref(\kappa_K, \varphi)$ is the set of interpretations which satisfy φ and have minimal rank in the OCF κ_K given by Definition 1.

3.4. Maxi-Adjustment

Maxi-Adjustment [26] was developed to address the problem of discarding too much information for applications like software engineering [27] and information filtering [10].

The idea in Maxi-Adjustment also involves selecting one consistent propositional base from K and φ denoted by $\delta_{MA}(K_\varphi)$. The difference is that it does not stop at the first rank where it meets inconsistency. Moreover, conflicts are solved rank by rank. We start from the first rank and take the formulas of S_1 which do not belong to any conflict in $\{\varphi\} \cup S_1$. Let S_1' be the set of these formulas. Then, we move to the next rank and add all formulas which are not involved in any conflict in $S_1' \cup S_2$, and so on. It is clear that Maxi-Adjustment keeps more formulas than Adjustment. Formally, $\delta_{MA}(K_\varphi)$ is computed following Algorithm 2.

Example 2 (*using Maxi-Adjustment*). Let us consider again the knowledge base given in Example 2. First, we have $\delta_{MA}(K_{\neg c}) = \{\neg c\}$. There is no conflict in $\delta_{MA}(K_{\neg c}) \cup S_1$ then

$$\delta_{MA}(K_{\neg c}) \leftarrow \{\neg c, \neg a \vee \neg b \vee c, \neg d \vee c, \neg e \vee c\}.$$

```
Data: a stratified knowledge base K = (S_1, ..., S_n);
      a new sure formula: φ;
Result: a consistent propositional base δ_MA(K_φ)
begin
    KB ← {φ};
    for i ← 1 to n do;
        if (KB ∪ S_i is consistent) then
            KB ← KB ∪ S_i
        else
            Let C = S_i ∩ kernel (KB ∪ S_i);
            KB ← KB ∪ {φ: φ ∈ S_i and φ ∉ C};
    return KB
end
```

Algorithm 2. MA(K, φ).

Now, S_2 contradicts $\delta_{MA}(K_{\neg c})$ due to the conflicts $\{d, \neg d \vee c, \neg c\}$ and $\{e, \neg e \vee c, \neg c\}$. Then, we do not add the clauses from S_2 involved in these conflicts:

$$\delta_{MA}(K_{\neg c}) \leftarrow \delta_{MA}(K_{\neg c}) \cup \{f, \neg f \vee \neg g \vee c\}.$$

Now, S_3 contradicts $\delta_{MA}(K_{\neg c})$ due to the conflicts $\{a, b, \neg a \vee \neg b \vee c, \neg c\}$ and $\{f, g, \neg f \vee \neg g \vee c, \neg c\}$. Since all the clauses, except h, from the stratum S_3 are involved in one conflict, we only add h to $\delta_{MA}(K_{\neg c})$. Finally, we get:

$$\delta_{MA}(K_{\neg c}) = \{\neg c, \neg a \vee \neg b \vee c, \neg d \vee c, \neg e \vee c, f, \neg f \vee \neg g \vee c, h\}.$$

Note that $\delta_{MA}(K_{\neg c}) \vdash h$.

4. Disjunctive Maxi-Adjustment

Although Maxi-Adjustment retains more information than Adjustment, it can still be argued that it is too cavalier in the way it solves the conflicts.

In this section, we propose a new strategy which is a significant improvement on Maxi-Adjustment. The computation of the consistent base is essentially the same as in Maxi-Adjustment, the only difference is when we meet an inconsistency at some rank, instead of removing all the formulas involved in the conflicts at this rank we weaken them, by replacing them by their pairwise disjunctions. If the result is consistent then we move to the next rank, else we replace these formulas by their possible disjunctions of size 3. If the result is consistent then we move to the next rank, else we add the disjunctions of size 4 of these formulas, and so on. We summarize this process in Algorithm 3.

```
Data: a stratified knowledge base K = (S_1, ..., S_n);
      a new sure formula: φ;
Result: a consistent propositional base δ_DMA(K_φ)
begin
    KB ← {φ};
    for i ← 1 to n do
        if (KB ∪ S_i is consistent) then
            KB ← KB ∪ S_i
        else
            Let C = S_i ∩ kernel(KB ∪ S_i);
            KB ← KB ∪ {φ: φ ∈ S_i and φ ∉ C};
            k ← 2;
            while (k ⩽ |C| and KB ∪ d_k(C) is inconsistent) do
                k ← k + 1;
            if k ⩽ |C| then KB ← KB ∪ d_k(C);
    return KB
end
```

Algorithm 3. DMA(K, φ).

Notation. $d_k(C)$ is the set of all possible nontautological disjunctions of size k between *different* formulas of C. If $k > |C|$ then $d_k(C) = \emptyset$. For example, if $C = \{a, b, c\}$ then $d_2(C) = \{a \vee b, a \vee c, b \vee c\}$.

Example 2 (*using DMA*). First, we have $KB = \{\neg c\}$. There is no conflict in $KB \cup S_1$ then $KB \leftarrow \{\neg c, \neg a \vee \neg b \vee c, \neg d \vee c, \neg e \vee c\}$. Now, S_2 contradicts KB due to the conflicts $\{d, \neg d \vee c, \neg c\}$ and $\{e, \neg e \vee c, \neg c\}$. We do not add the clauses from S_2 involved in these conflicts:

$$KB \leftarrow KB \cup \{f, \neg f \vee \neg g \vee c\}.$$

Now we create all the possible disjunctions of size 2 with $C = \{d, e\}$: $d_2(C) = \{d \vee e\}$. Since $KB \cup d_2(C)$ is inconsistent, and we cannot create larger disjunctions, we do not add anything from S_2 to KB.

Please note at this rank, we do not add more information than Maxi-Adjustment. Now, S_3 contradicts KB due to the conflicts $\{a, b, \neg a \vee \neg b \vee c, \neg c\}$ and $\{f, g, \neg f \vee \neg g \vee c, \neg c\}$. h is not involved in any conflict. Then,

$$KB \leftarrow KB \cup \{h\}.$$

We now create all the possible pairwise disjunctions with $C = \{a, b, g\}$: $d_2(C) = \{a \vee b, a \vee g, b \vee g\}$. Since $KB \cup d_2(C)$ is inconsistent, we create $d_3(C) = \{a \vee b \vee g\}$. Since $KB \cup d_3(C)$ is consistent, we add $d_3(C)$ to KB and the algorithm stops.

Then

$$\delta_{DMA}(K_\varphi) = \{\neg c, \neg a \vee \neg b \vee c, \neg d \vee c, \neg e \vee c, f, \neg f \vee \neg g \vee c, h, a \vee b \vee g\}$$

which is equivalent to $\{\neg c, \neg a \vee \neg b, \neg d, \neg e, f, \neg g, h, a \vee b\}$.

DMA keeps more information from the last stratum than Maxi-Adjustment does.

Definition 5. A formula ψ is said to be a DMA consequence of K and φ, denoted by $K_\varphi \vdash_{DMA} \psi$, if it is inferred from $\delta_{DMA}(K_\varphi)$. Namely, $K_\varphi \vdash_{DMA} \psi$ iff $\delta_{DMA}(K_\varphi) \vdash \psi$.

It is important to note that DMA consequence relations and MA consequence relations are incomparable, as it is illustrated by the following example:

Example 3. Let $K = \{S_1, S_2\}$ and $\varphi = (a \vee b) \wedge c$ where $S_1 = \{\neg a, \neg b\}$ and $S_2 = \{a \vee \neg c, b \vee \neg c\}$.

Then, $\delta_{MA}(K_\varphi) = \{(a \vee b) \wedge c, a \vee \neg c, b \vee \neg c\}$ which is equivalent to $\{a, b, c\}$, and $\delta_{DMA}(K_\varphi) = \{(a \vee b) \wedge c, \neg a \vee \neg b, a \vee b \neg c\}$ which is equivalent to $\{a \vee b, \neg a \vee \neg b, c\}$.

In fact, DMA coincides with MA when there is no way to weaken conflicts.

5. Two other implementations of DMA

In this section, we propose two alternative ways to compute $\delta_{DMA}(K_\varphi)$ but presented in different syntactical ways.

The first approach, called whole-DMA(K, φ), does not compute the kernel. The main motivation for this alternative is that computing the kernel is in general a hard problem: Determining if a given formula ϕ is in the kernel of KB is Σ_2^p*-complete* ([6, p. 378], consequence of Theorem 8.2 of [16]). This decision problem is in the second level of the polynomial hierarchy, while SAT is "only" on the first level. So determining if one formula belongs to the kernel of KB is one level harder than SAT (to solve that problem, one needs in the worst case an exponential number of calls to SAT).

For the second approach, called iterative-DMA(K, φ), when inconsistency is (again) met after weakening the kernel, then rather than weakening the original kernel by considering its disjunctions of size 3, we only weaken the newly computed kernel obtained by considering disjunctions of size 2. Since DMA focus on weakening conflicting formula, it looked interesting to push that idea ahead and to apply the same principle when inconsistency is detected when adding pairwise disjunctions. The objective of the approach is to keep the revised base syntactically close to the original one (DMA) and to its real conclusions: some disjunctions produced by DMA are in reality subsumed by some consequences of the revised base. It is not the case with IDMA.

In the two following subsections, we only show that whole and iterative DMA are equivalent to DMA. We give in Section 7 a comparison between the three approaches based on experimental results.

5.1. Whole Disjunctive Maxi-Adjustment

We propose a slightly modified version of the DMA algorithm. The idea is that when $KB \cup S_i$ is inconsistent, instead of considering all possible disjunctions of size j of elements of S_i which are in $kernel(KB \cup S_i)$, we consider all possible disjunctions of size j of S_i without computing a kernel.

Note that it looks odd to consider such a solution provided the huge number of disjunctions that can be generated. However, we will see later in the experimental results that is can be efficiently computed. This is mainly justified by the fact that WDMA does not need to explicitly compute the kernel.

First we need the following lemma:

Lemma 2. *Let $KB \cup S$ be inconsistent. Let C be the subset of S in $Kernel(KB \cup S)$, and F be the set of remaining free formulas in S. If for some j $KB \cup d_j(C) \cup F$ is inconsistent then,*

$$\forall \phi, \phi \in F, \text{ we have } \phi \text{ is also free in } KB \cup d_j(C) \cup F.$$

With the help of this lemma, the idea of the Whole Disjunctive Maxi-Adjustment is justified by the following proposition:

Proposition 1. *Let KB be consistent but inconsistent with S. Let C be the subset of S in $kernel(KB \cup S)$, and $F = S - C$ be the set of remaining free formulas in S. Let $d_j(C)$*

```
Data: a stratified knowledge base K = (S_1, ..., S_n);
      a new sure formula: φ;
Result: a consistent propositional base δ_WDMA(K_φ)
begin
    KB ← {φ};
    for i ← 1 to n do
        if (KB ∪ S_i is consistent) then
            KB ← KB ∪ S_i
        else
            k ← 2;
            while (KB ∪ d_k(S_i) is inconsistent and k ⩽ |S_i|) do
                k ← k + 1;
            if k ⩽ |S_i| then KB ← KB ∪ d_k(S_i);
    return KB
end
```

Algorithm 4. WDMA(K, φ).

(*respectively* $d_j(S)$) *be the set of all possible disjunctions of size* j *from* C (*respectively* S). *Then, if* $KB \cup d_{j-1}(S)$ *is inconsistent then*

$$KB \cup d_j(C) \cup F \equiv KB \cup d_j(S).$$

With the help of Proposition 1, the "else" block in the DMA algorithm is replaced by

else
$k \leftarrow 2$
while ($KB \cup d_k(S_i)$ is inconsistent and $k \leqslant |S_i|$) **do** $k \leftarrow k + 1$
if $k \leqslant |S_i|$ **then** $KB \leftarrow KB \cup d_k(S_i)$

Indeed, the whole-DMA algorithm is given as follows (see Algorithm 4).

Example 2 (*using whole-DMA*). First, we have $KB = \{\neg c\}$. S_1 is consistent with KB. Then, $KB \leftarrow KB \cup S_1$.

Now, S_2 contradicts KB. We compute all possible pairwise disjunctions with S_2.

$$d_2(S_2) = \{d \vee e, d \vee f, d \vee \neg f \vee \neg g \vee c, e \vee f, e \vee \neg f \vee \neg g \vee c\}.$$

Since, $KB \cup S_2$ is inconsistent, we compute all possible disjunctions of size 3 between formulas of S_2. We get $d_3(S_2) = \{d \vee e \vee f, d \vee e \vee \neg f \vee \neg g \vee c\}$ which is consistent with KB. Then, $KB \leftarrow KB \cup d_3(S_2)$.

Now, S_3 is inconsistent with KB. We compute all possible pairwise disjunctions with S_3. $d_2(S_3) = \{a \vee b, a \vee g, a \vee h, b \vee g, b \vee h, g \vee h\}$ which is still inconsistent with KB. We have $d_3(S_3) = \{a \vee b \vee g, a \vee b \vee h, b \vee g \vee h, a \vee g \vee h\}$ which is consistent with KB, then $KB \leftarrow KB \cup d_3(S_3)$.

Hence,

$$\delta_{WDMA}(K_{\neg c}) = \{\neg c, \neg a \vee \neg b \vee c, \neg d \vee c, \neg e \vee c, d \vee e \vee f, d \vee e \vee \neg f \vee \neg g \vee c, a \vee b \vee g, a \vee b \vee h, b \vee g \vee h, a \vee g \vee h\}$$

which is equivalent to $\{\neg c, \neg a \vee \neg b, \neg d, \neg e, f, \neg g, a \vee b, h\}$. Then, it is equivalent to $\delta_{DMA}(K_{\neg c})$.

5.2. Iterative Disjunctive Maxi-Adjustment

The idea of this alternative implementation of DMA is as follows: let S_i be inconsistent with KB. Let C and F be the kernel and the remaining formulas of S_i.

Now assume that $KB \cup F \cup d_2(C)$ is still inconsistent. Then rather than weakening C again by considering disjunctions of size 3, we only weaken those formulas in $d_2(C)$ which are still responsible for conflicts. Namely, we split $d_2(C)$ into C' and F' which respectively represent the kernel and remaining formulas of $d_2(C)$. Then instead of taking $KB \cup F \cup d_3(C)$ as in DMA, we take $KB \cup F \cup F' \cup d_2(C')$.

We only add in the stratum formulas not subsumed by one formula already present in S_i because if that formula is free, then the subsumed formula is also free.

Proposition 2. *Let $KB \cup F \cup d_i(C)$ be inconsistent. Let C' be the subset of $d_i(C)$ in kernel$(KB \cup F \cup d_i(C))$, and $F' = d_i(C) - C'$ be the set of remaining formulas. Then,*

$$KB \cup F \cup d_{i+1}(C) \equiv KB \cup F \cup F' \cup d_2(C').$$

```
Data: a stratified knowledge base K = (S_1, ..., S_n);
      a new sure formula: φ
Result: a consistent propositional base δ_IDMA(K_φ)
begin
    KB ← {φ}; i ← 1;
    while i ⩽ n do
        if KB ∪ S_i is consistent then
            KB ← KB ∪ S_i;
            i ← i + 1;
        else
            Let C = S_i ∩ kernel(KB ∪ S_i);
            % Since KB ∪ S_i is inconsistent, C is not empty;
            S_i ← S_i − C;
            if |C| = 1 then
                % Only one conflicting formula (cannot weaken it);
                % Proceed to the next stratum;
                i ← i + 1
            else
                % Conflicting clauses can be weakened;
                S_i ← S_i ∪ {φ: φ ∈ d_2(C) and ∄φ' ∈ S_i subsuming φ};
    return KB
end
```

Algorithm 5. IDMA(K, φ).

The proof is a corollary of Proposition 1 and the following lemma which shows that taking all disjunctions of size i, then reconsidering all disjunctions of size 2 again on the result is the same as considering all disjunctions of size $i+1$:

Lemma 3. *Let A be a set of formulas. Let $B = d_i(A)$ and $C = d_{i+1}(A)$ be the set of all possible disjunctions of A of size i and $i+1$ respectively. Then, $C \equiv d_2(B)$.*

Example 2 (*using iterative-DMA*). First, we have $KB = \{\neg c\}$. There is no conflict in $KB \cup S_1$. Then, $KB \leftarrow KB \cup S_1$.

S_2 is inconsistent with KB due to the conflicts $\{\neg c, \neg d \vee c, d\}$ and $\{\neg c, \neg e \vee c, e\}$. We add $\{f, \neg f \vee \neg g \vee c\}$ to KB. The disjunction $d \vee e$ is still inconsistent with KB, then we move to S_3.

S_3 contradicts KB due to the conflicts $\{a, b, \neg a \vee \neg b \vee c, \neg c\}$ and $\{f, g, \neg f \vee \neg g \vee c, \neg c\}$. h is not involved in any conflict. Then, $KB \leftarrow KB \cup \{h\}$. We now create all the possible pairwise disjunctions with $C = \{a, b, g\}$: $d_2(C) = \{a \vee b, a \vee g, b \vee g\}$.

$KB \cup d_2(C)$ is inconsistent due to the conflict $\{\neg c, \neg a \vee \neg b \vee c, f, \neg f \vee \neg g \vee c, a \vee g, b \vee g\}$. $a \vee b$ in $d_2(C)$ is not involved in the conflict, then $KB \leftarrow KB \cup \{a \vee b\}$.

Now, we take the pairwise disjunctions with $C = \{a \vee g, b \vee g\}$. $d_2(C) = \{a \vee b \vee g\}$. $KB \cup d_2(C)$ is consistent. However, there is no need to add $a \vee b \vee g$ to KB since $a \vee b$ already belongs to KB. Hence, $\delta_{IDMA}(K_{\neg c}) = \{\neg c, \neg a \vee \neg b \vee c, \neg d \vee c, \neg e \vee c, f, \neg f \vee \neg g \vee c, a \vee b, h\}$ which is equivalent to $\{\neg c, \neg a \vee \neg b, \neg d, \neg e, f, \neg g, a \vee b, h\}$. Hence, it is equivalent to $\delta_{DMA}(K_{\neg c})$.

Note that in the last stratum, $a \vee b$ is directly produced by IDMA while DMA produced $a \vee b \vee g$.

6. DMA: compilation of lexicographical inferences

The aim of this section is to show that DMA is a compilation of the lexicographical system (see Fig. 1). An immediate consequence of this fact is that DMA satisfies the rational postulates AGM [1] since it is shown in [4,23] that the lexicographic system satisfies all AGM postulates.The proof in [4] was based on expliciting a total preorder between interpretations. This total preorder will be recalled in Definition 8.

First let us recall the lexicographical inference.

6.1. Syntactic and semantic definitions of lexicographical inference

The lexicographical system [3,20] is a coherence-based approach where an inconsistent knowledge base is replaced by a set of maximally preferred consistent subbases. The preference relation between subbases is defined as follows:

Definition 6. Let $A = (A_1, \ldots, A_n)$ and $B = (B_1, \ldots, B_n)$ be two consistent subbases of K.

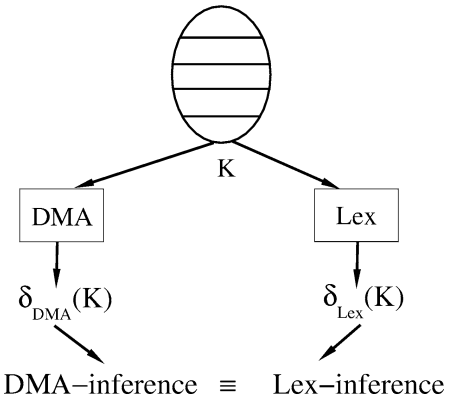


Fig. 1. Equivalence between DMA inference and lexicographical inference.

A is said to be lexicographically preferred to B, denoted by $A >_{Lex} B$, iff

$\exists k$ s.t. $|A_k| > |B_k|$ and $\forall 1 \leqslant j < k, |A_j| = |B_j|$.

Let $\delta_{Lex}(K_\varphi)$ denotes the set of all lexicographically preferred subbases of K_φ, those which are maximal with respect to $>_{Lex}$. Then, the lexicographical inference is defined by:

Definition 7. A formula ψ is said to be a lexicographical consequence of K_φ, denoted by $K_\varphi \vdash_{Lex} \psi$, if it is a classical consequence of all the elements of $\delta_{Lex}(K_\varphi)$, namely

$$\forall A \in \delta_{Lex}(K_\varphi), \quad A \vdash \psi.$$

Example 2 (*continued*). We have $\delta_{Lex}(K_{\neg c}) = (A, B)$ where

$$A = \{\neg c, \neg a \vee \neg b \vee c, \neg d \vee c, \neg e \vee c, f, \neg f \vee \neg g \vee c, a, h\} \quad \text{and}$$
$$B = \{\neg c, \neg a \vee \neg b \vee c, \neg d \vee c, \neg e \vee c, f, \neg f \vee \neg g \vee c, b, h\}.$$

For example, we have

$$K_{\neg c} \vdash_{Lex} a \vee b \text{ since } A \vdash a \vee b \text{ and } B \vdash a \vee b.$$

At the semantic level, the lexicographical inference can be defined in a preferential way a la Shoham [24]. First, we need to rewrite the lexicographical system at the semantic level, which is immediate:

Definition 8. Let $K = (S_1, \ldots, S_n)$. Let ω and ω' be two interpretations, and A_ω, $A_{\omega'}$ be the consistent subbases composed of all formulas of K satisfied by ω and ω' respectively.

Then, ω is said to be lexicographically preferred to ω' with respect to K, denoted by $\omega >_{Lex,K} \omega'$, iff $A_\omega >_{Lex} A_{\omega'}$ (using Definition 6).

Then, we have the following proposition [4].

Proposition 3. *$K_\varphi \vdash_{Lex} \psi$ if and only if $\forall \omega \in Pref(\varphi, >_{Lex}), \omega \models \psi$.*

Namely, ψ is a lexicographical consequence of K_φ if it is true in all interpretations satisfying φ and which are maximal with respect to $>_{Lex,K}$.

6.2. Basic steps of the compilation

The aim of this section is to show that DMA is equivalent to the lexicographical system. DMA offers a clear advantage over the lexicographical system because it obviates the need to explicitly compute $\delta_{Lex}(K_\varphi)$ which may be exponential in size. Formally, we will show the following equivalence:

$$K_\varphi \vdash_{Lex} \psi \quad \text{iff} \quad K_\varphi \vdash_{DMA} \psi. \tag{1}$$

Note that $\delta_{DMA}(K_\varphi)$ is a propositional consistent base.

Example 2 (*continued*). Let us first show on an example that applying the lexicographical system on $K_{\neg c}$ gives the same results as applying *DMA* on $K_{\neg c}$. Indeed,

$$\begin{aligned}
K_{\neg c} \vdash_{Lex} \psi \quad &\text{iff} \quad A \vdash \psi \text{and } B \vdash \psi \\
&\text{iff} \quad A \vee B \vdash \psi, \text{ where } A \vee B = \{\phi_i \vee \psi_j : \phi_i \in A \text{ and } \psi_j \in B\} \\
&\text{iff} \quad \{\neg c, \neg a \vee \neg b, \neg d, \neg e, f, \neg g, a \vee b, h\} \vdash \psi \\
&\quad\ \text{(after removing subsumed formulas in } A \vee B) \\
&\text{iff} \quad \delta_{DMA}(K_{\neg c}) \vdash \psi \\
&\text{iff} \quad K_{\neg c} \vdash_{DMA} \psi.
\end{aligned}$$

Fig. 2 gives an outline of the way to prove equivalence (1). It is composed of two main steps:

Step 1. we construct a new base K' from K s.t.

$$K_\varphi \vdash_{Lex} \psi \quad \text{iff} \quad K'_\varphi \vdash_A \psi. \tag{2}$$

Namely, applying lexicographical system on K_φ is equivalent to applying Adjustment to K'_φ (see Fig. 2).

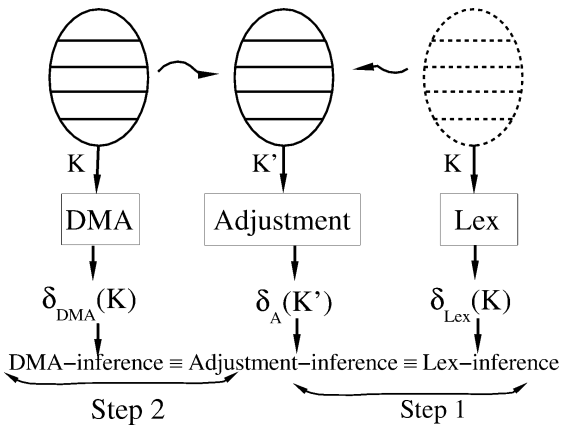


Fig. 2. Basic steps of the compilation of lexicographical inference.

Step 2. in the second step we show that

$$K'_\varphi \vdash_A \psi \quad \text{iff} \quad K_\varphi \vdash_{DMA} \psi. \tag{3}$$

Namely, applying Adjustment to K'_φ is equivalent to applying DMA to K_φ (see Fig. 2). The proof is obtained by successive simplifications of K'. This is formally described by Lemmas 4–6.

6.2.1. Step 1: Constructing K'

At the semantic level, (2) is equivalent to show:

$$\kappa_{K'}(\omega) < \kappa_{K'}(\omega') \quad \text{iff} \quad \omega >_{Lex,K} \omega' \tag{4}$$

where $\kappa_{K'}$ is the OCF associated to K' obtained from Definition 1. This equivalence means that K' and K generate the same ordering on Ω using respectively ordinal conditional function and lexicographical ordering. Indeed, the preferred models with respect to $<_{\kappa,K'}$ and $>_{Lex,K}$ satisfying φ are the same.

Let us now show how to construct K' from K such that it satisfies (4). For this, we use two intuitive ideas.

The first idea is that Adjustment is insensitive to the number of equally reliable formulas falsified while lexicographical system is not (i.e., cardinality of conflict sets). Assume that we have a base $K = (S_1)$ where $S_1 = \{\phi, \psi, \xi\}$, i.e., K contains three formulas with a same rank. We write K in a κ-ranked form as follows: $K = \{(\phi, k_1), (\psi, k_1), (\xi, k_1)\}$, namely all formulas in S_1 have the same rank (equals to some k_1). Then, the rank (using Definition 1) associated with an interpretation ω falsifying one formula is the same as an interpretation falsifying two formulas, and also as an interpretation falsifying the three formulas. However, if we use the lexicographical system, an interpretation falsifying one formula is preferred to an interpretation falsifying two formulas, and this latter is preferred to an interpretation falsifying the three formulas. Now, one can check that if we construct a κ-ranked base

$$K' = \big\{(\phi, k_1), (\psi, k_1), (\xi, k_1), (\phi \vee \psi, 2 * k_1), (\phi \vee \xi, 2 * k_1), (\psi \vee \xi, 2 * k_1), (\phi \vee \psi \vee \xi, 3 * k_1)\big\}$$

from K by adding the disjunctions $\phi \vee \psi$, $\phi \vee \xi$, $\psi \vee \xi$ and $\phi \vee \psi \vee \xi$ with higher ranks, then Eq. (4) is satisfied. So, the first idea is to *add disjunctions with the rank equal to the sum of ranks of formulas composing the disjunctions*.

The second idea is related to the notion of compensation (or *reinforcement*). To illustrate this idea, let us now consider $K = (S_1, S_2)$ such that $S_1 = \{\phi_1\}$ and $S_2 = \{\phi_2, \phi_3, \phi_4\}$. The stratification indicates that ϕ_1 is strictly preferred to ϕ_2 (respectively ϕ_3 and ϕ_4). Now, assume that K is inconsistent and in order to restore its consistency, we either get rid of ϕ_1, or get rid of $\{\phi_2, \phi_3, \phi_4\}$ together. If there are compensation (or reinforcement), then it may be reasonable to only ignore ϕ_1. The lexicographical system does not adapt the idea of compensation. The lexicographical system, if needed, prefers to maintain one prioritary formula, and get rid of all less prioritary formulas. The fact that the lexicographical system does not adapt reinforcement implies that ranks associated with the formulas should satisfy some constraints in order to recover the lexicographical inference. Indeed, let us for instance associate the rank 2 with ϕ_1, and the rank 1 with ϕ_2, ϕ_3, ϕ_4.

Let ω and ω' be two interpretations such that $A_\omega = \{\{\phi_1\}, \{\}\}$ and $A_{\omega'} = \{\{\}, \{\phi_2, \phi_3, \phi_4\}\}$. A_ω means that ω satisfies all formulas of S_1 but falsifies all formulas of S_2. $A_{\omega'}$ means that ω' satisfies all the formulas of S_2 but falsifies all the formulas of S_1. Following the suggestion of the first idea, let us add all possible disjunctions. We obtain:

$$\begin{aligned} K' = \big\{&(\phi_1 \vee \phi_2 \vee \phi_3 \vee \phi_4, 5),\\ &(\phi_1 \vee \phi_3 \vee \phi_4, 4), (\phi_1 \vee \phi_2 \vee \phi_4, 4), (\phi_1 \vee \phi_2 \vee \phi_3, 4),\\ &(\phi_1 \vee \phi_2, 3), (\phi_1 \vee \phi_3, 3), (\phi_1 \vee \phi_4, 3), (\phi_2 \vee \phi_3 \vee \phi_4, 3),\\ &(\phi_1, 2), (\phi_2 \vee \phi_3, 2), (\phi_2 \vee \phi_4, 2), (\phi_3 \vee \phi_4, 2),\\ &(\phi_2, 1), (\phi_3, 1), (\phi_4, 1)\big\}. \end{aligned}$$

We can easily check that $\kappa_{K'}(\omega) = 3$ and $\kappa_{K'}(\omega') = 2$ (namely, ω' is preferred to ω) while $A_\omega >_{Lex,K} A_{\omega'}$. This is due to the fact that the disjunction $\phi_2 \vee \phi_3 \vee \phi_4$ has a rank higher than ϕ_1. Hence, there is a compensation effect. So, in order to recover the lexicographical order, ϕ_1 should have a rank strictly greater than the rank of $\phi_2 \vee \phi_3 \vee \phi_4$. A way to do this is to significantly differentiate the different ranks associated with strata. For this we associate to each formula ϕ_{ij} of S_i a rank k_i such that $k_i > \sum_{l=i+1}^{n} k_l * |S_l|$, where $|S_l|$ is the number of formulas in S_l. It means that *the rank given to a stratum must be greater than the sum of all the ranks given to formulas of less reliable strata*.

Following these two ideas, K' is formally constructed as follows:

Definition 9. Let $K = (S_1, \ldots, S_n)$. We construct from K a κ-ranked base K' in the following way:

1. We first define a new base B:

$$B = \big\{(\phi_{ij}, k_i)\colon\ i = 1, \ldots, n \text{ and } \phi_{ij} \in S_i\big\},$$

   where $k_i > \sum_{l=i+1}^{n} k_l * |S_l|$.
2. Then,

$$K' = \bigcup_{i=1}^{M} \big\{(\phi, b_i) \mid \phi \in D_i(B)\big\}$$

   where $M = \sum_{i=1}^{n} |S_i|$ (i.e., $M = |K|$) and $D_i(B)$ are all possible nontautological disjunctions of size i built from B, and b_i is the sum of ranks of the formulas in $D_i(B)$.

The first item of Definition 9 associates to each formula $\phi_{ij} \in S_i$ a rank k_i, such that k_i is greater than the sum of all ranks of all formulas which are in strata $j > i$. Such assignment always exists. For instance, we can define $k_i's$ as: $k_n = 1$, and $k_i = \sum_{l=i+1}^{n} k_l * |S_l| + 1$.

The second item of Definition 9 means that K' is composed of all possible nontautological disjunctions of B, and the rank associated to each disjunction is simply the sum of the ranks of the formulas composing that disjunction.

Example 4. Let $K = (S_1, S_2)$ where $S_1 = \{\neg a \vee \neg b \vee c\}$ and $S_2 = \{a, b, g\}$. We have $B = \{(\neg a \vee \neg b \vee c, k_1), (a, k_2), (b, k_2), (g, k_2)\}$ where $k_1 > k_2$. Then, the base K' obtained from step 2 is:

$$K' = \big\{(\neg a \vee \neg b \vee c \vee g, k_1 + k_2), (\neg a \vee \neg b \vee c, k_1), (a \vee b \vee g, 3 * k_2), (a \vee b, 2 * k_2), (a \vee g, 2 * k_2), (b \vee g, 2 * k_2), (a, k_2), (b, k_2), (g, k_2)\big\}.$$

Then, we have the following proposition:

Proposition 4. *Let $K = (S_1, \ldots, S_n)$ be a stratified base and φ be a new formula. Let K' be the base constructed from K using Definition* 9. *Then,*

$$K'_\varphi \vdash_A \psi \quad \textit{iff} \quad K_\varphi \vdash_{Lex} \psi.$$

Example 4 (*continued*). It can be checked that the inconsistency degree of K'_φ is equal to k_2. Then, $\delta_A(K'_{\neg c}) = \{\neg c, \neg a \vee \neg b \vee c \vee g, \neg a \vee \neg b \vee c, a \vee b \vee g, a \vee b, a \vee g, b \vee g\}$ (composed of formulas of $K'_{\neg c}$ having a rank greater than K_c) which is equivalent to $\{\neg c, \neg a \vee \neg b, a \vee b, g\}$.

Moreover we have $\delta_{Lex}(K_{\neg c}) = (A_1, A_2)$ where

$$A_1 = \{\neg c, \neg a \vee \neg b \vee c, a, g\} \quad \text{and} \quad A_2 = \{\neg c, \neg a \vee \neg b \vee c, b, g\}.$$

Then,

$$\begin{aligned}
K_{\neg c} \vdash_{Lex} \psi \quad &\text{iff} \quad \forall A_i, A_i \in \delta_{Lex}(K_\varphi), A_i \vdash \psi \\
&\text{iff} \quad A_1 \vdash \psi \text{ and } A_2 \vdash \psi \\
&\text{iff} \quad A_1 \vee A_2 \vdash \psi \text{ (with } A_1 \vee A_2 = \{\phi_i \vee \delta_j \colon \phi_i \in A_1 \text{ and } \delta_j \in A_2\}) \\
&\text{iff} \quad \{\neg c, \neg a \vee \neg b \vee c, a \vee b, g\} \vdash \psi \\
&\text{iff} \quad \{\neg c, \neg a \vee \neg b, a \vee b, g\} \vdash \psi \\
&\text{iff} \quad \delta_A(K'_{\neg c}).
\end{aligned}$$

6.2.2. Step 2: Adjustment on K'_φ is equivalent to DMA on K_φ

The aim of this subsection is to show that the base K' constructed in step 1 allows us to recover the lexicographical system. Namely, we will gradually show that:

Proposition 5. *Let $K = (S_1, \ldots, S_n)$ be a stratified base, and φ be a new formula. Let K' be the base constructed from K using Definition* 9. *Then,*

$$K'_\varphi \vdash_A \psi \quad \textit{iff} \quad K_\varphi \vdash_{DMA} \psi. \tag{5}$$

Recall that $K'_\varphi \vdash_A \psi$ and $K_\varphi \vdash_{DMA} \psi$ are defined by $\delta_A(K'_\varphi) \vdash \psi$ and $\delta_{DMA}(K_\varphi) \vdash \psi$ respectively. Then to show the equivalence (5), we will show that $\delta_A(K'_\varphi)$ is equivalent to $\delta_{DMA}(K_\varphi)$. The idea is to simplify the computation of $\delta_A(K'_\varphi)$ until recovering $\delta_{DMA}(K_\varphi)$.

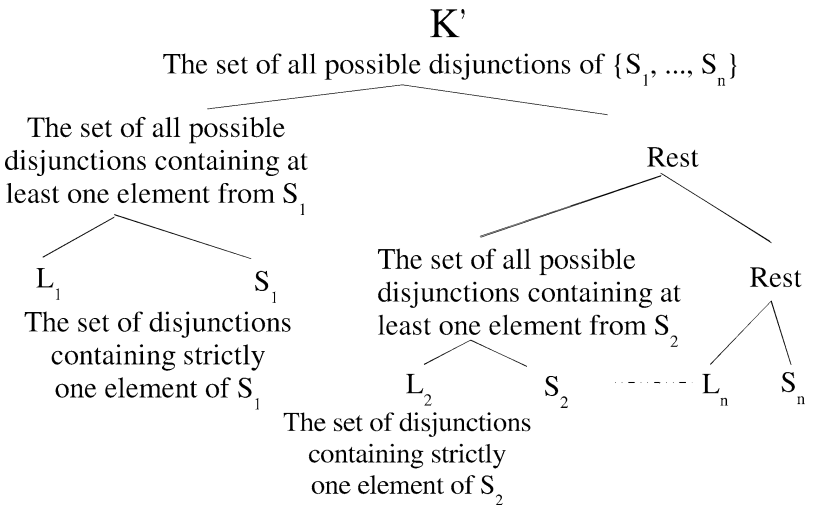


Fig. 3. Stratification process of K'.

First, we need to put K'_φ in a stratified form. The stratification will be constructed progressively. Recall that K' contains all possible, nontautological, disjunctions of K.

Fig. 3 illustrates how the stratification is obtained.We first split K' into two sets: one, denoted by A, containing at least one formula from S_1, and the rest, denoted by B. Note that the rank associated to any element of A is greater than the rank to any element of B. This is due to item (1) in Definition 9. Indeed, formulas of A have a weight at least equals to k_1, which is greater than the sum of the weights associated with formulas in B.

Now A can be split again into two sets S_1 (the first stratum of K), and the rest, denoted by L_1. Formulas of L_1 have a rank greater than any formula of S_1 (since L_1 strictly contain S_1).

The same process can be recursively applied on B, as it is illustrated in Fig. 3. Therefore K'_φ can be rewritten in the form

$$K'_\varphi = (S_0, L_1, S_1, \ldots, L_n, S_n),$$

where $S_0 = \{\varphi\}$ and S_i $(i = 1, \ldots, n)$ are the strata of K, and L_i are disjunctions which strictly contains at least one element of S_i.

Beware, here L_j's $(j = 1, \ldots, n)$ do not necessarily only contain formulas with the same rank. The only requirement is that they contain disjunctions, with a size strictly greater than 1, between different formulas of $S_i \cup \cdots \cup S_n$ including at least one formula from S_i.

Example 4 (*continued*). Recall that

$$K' = \big\{(\neg a \vee \neg b \vee c \vee g, k_1 + k_2), (\neg a \vee \neg b \vee c, k_1), (a \vee b \vee g, 3 * k_2),$$
$$(a \vee b, 2 * k_2), (a \vee g, 2 * k_2), (b \vee g, 2 * k_2), (a, k_2), (b, k_2), (g, k_2)\big\}$$

and $\varphi = \neg c$. Then, in a stratified form we have $K'_\varphi = (S_0, L_1, S_1, L_2, S_2)$ where $S_0 = \{\neg c\}$, $L_1 = \{\neg a \vee \neg b \vee c \vee g\}$, $S_1 = \{\neg a \vee \neg b \vee c\}$, $L_2 = \{a \vee b \vee g, a \vee b, a \vee g, b \vee g\}$ and $S_2 = \{a, b, g\}$.

Our aim is to simplify K'_φ. We start with the two following simplifications. Since we apply Adjustment on K'_φ to compute $\delta_A(K'_\varphi)$, the first simplification consists in ignoring formulas in K'_φ under the inconsistency rank (see Section 3.3).

The second simplification concerns subsumed disjunctions which are not added. This is justified by the fact that $\delta_A(K'_\varphi)$ is a propositional base. Indeed when some formula ϕ belongs to $\delta_A(K'_\varphi)$, there is no need to add disjunctions including this formula to $\delta_A(K'_\varphi)$ since then they will be subsumed.

These two simplifications are formalized in the following lemma:

Lemma 4. *Let $K_\varphi = (S_0, S_1, \ldots, S_n)$ be a stratified base, and $K'_\varphi = (S_0, L_1, S_1, \ldots, L_n, S_n)$ be the base associated with K_φ using Definition 9. Assume that $S_0 \cup \cdots \cup S_{i-1}$ is consistent and $S_0 \cup \cdots \cup S_i$ is inconsistent. Then, applying Adjustment on K'_φ is equivalent to applying Adjustment on*

$$(S_0, S_1, \ldots, S_{i-1}, L_i).$$

Indeed, we say that K'_φ is equivalent to $(S_0, S_1, \ldots, S_{i-1}, L_i)$. Equivalence is to be understood as "results in the same set of conclusions", more formally, $\delta_A(K)$ is equivalent to $\delta_A(K')$ iff $\forall\psi, \delta_A(K) \models \psi$ iff $\delta_A(K') \models \psi$.

Intuitively, Lemma 4 says first remove $(L_{i+1}, S_{i+1}, \ldots, S_n)$, since they are below the inconsistency rank (recall that adjustment stops at the first rank when inconsistency is met). Then remove $(L_1, \ldots, L_{i-1})$. This is justified by the fact that $S_1 \cup \cdots \cup S_{i-1}$ is consistent, and belongs to $\delta_A(K'_\varphi)$. Then there is no need to add formulas from L_j $(j = 1, \ldots, i-1)$ since they are subsumed by the ones from S_j $(j = 1, \ldots, i-1)$.

Example 4 (*continued*). First, we can check in Example 4 that $Inc(K'_\varphi) = k_2$. Then, Adjustment on K'_φ is equivalent to Adjustment on (S_0, L_1, S_1, L_2).

Since $S_0 \cup L_1 \cup S_1$ is consistent and it is above the inconsistency level then it will belong to $\delta_A(K'_\varphi)$. Note that L_1 is a disjunction composed of a formula of S_1. Then, L_1 is subsumed by S_1 in $\delta_A(K'_\varphi)$.

The last simplification concerns now L_i. We will show that disjunctions in L_i can be reduced. Recall that L_i's are sets of all possible nontautological disjunctions between formulas of $S_i \cup \cdots \cup S_n$ containing at least one formula from S_i. Let us first develop the expression of L_i.

Let $d_k(S_i)$ be the set of all possible disjunctions of size k between formulas of S_i. Let $\mathbb{R}_{i+1} = (L_{i+1}, S_{i+1}, \ldots, L_n, S_n)$. Let $m = |S_i|$. Then,

$$L_i = \big((d_m(S_i) \vee \mathbb{R}_{i+1}), d_m(S_i), (d_{m-1}(S_i) \vee \mathbb{R}_{i+1}), d_{m-1}(S_i), \ldots, (d_1(S_i) \vee \mathbb{R}_{i+1})\big),$$

where $(d_j(S_i) \vee \mathbb{R}_{i+1})$ is the set of all possible disjunctions between formulas of $d_j(S_i)$ and formulas of $\mathbb{R}_{i+1}$.

Example 4 (*continued*). We have $K'_\varphi = (S_0, L_1, S_1, L_2, S_2)$. $R_3 = \emptyset$. Then, $L_2 = (d_3(S_2), d_2(S_2)) = \{a \vee b \vee g, a \vee b, b \vee g, a \vee g\}$.

Also, $R_2 = (L_2, S_2)$ and $|S_1| = 1$. Then,

$$L_1 = d_1(S_1) \vee R_2 = \big\{(\neg a \vee \neg b \vee c) \vee (a \vee b \vee g), (\neg a \vee \neg b \vee c) \vee (a \vee b), (\neg a \vee \neg b \vee c) \vee (b \vee g), (\neg a \vee \neg b \vee c) \vee (a \vee g), (\neg a \vee \neg b \vee c) \vee a, (\neg a \vee \neg b \vee c) \vee b, (\neg a \vee \neg b \vee c) \vee g\big\}$$

which is equivalent to $\{\neg a \vee \neg b \vee c \vee g\}$.

As a corollary of Lemma 4 and this expression we get:

Corollary 1. *Let $K_\varphi = (S_0, \ldots, S_n)$ be such that $S_0 \cup \cdots \cup S_{i-1}$ is consistent, and $S_0 \cup \cdots \cup S_i$ is inconsistent. Let $m = |S_i|$. Let K'_φ be the base associated with K_φ. Then, applying Adjustment on K'_φ is equivalent to applying Adjustment on*

$$\big(S_0, \ldots, S_{i-1}, (d_m(S_i) \vee \mathbb{R}_{i+1}), d_m(S_i), \ldots, (d_2(S_i) \vee \mathbb{R}_{i+1}), d_2(S_i), (d_1(S_i) \vee \mathbb{R}_{i+1})\big).$$

The proof is immediate since we have simply replaced L_i by its expression.

We continue the simplification of $\delta_A(K'_\varphi)$. The following lemma is similar to Lemma 4 however it considers the strata $d_k(S_i)$:

Lemma 5. *Let $m = |S_i| \geqslant k > 1$. Let $K_\varphi = (S_0, \ldots, S_n)$ be such that $S_0 \cup \cdots \cup S_{i-1}$ is consistent but $S_0 \cup \cdots \cup S_i$ is inconsistent. Let*

$$K'_\varphi = \big(S_0, \ldots, S_{i-1}, d_m(S_i) \vee \mathbb{R}_{i+1}, d_m(S_i), \ldots, d_2(S_i) \vee \mathbb{R}_{i+1}, d_2(S_i), d_1(S_i) \vee \mathbb{R}_{i+1}\big).$$

Assume that $S_0 \cup S_1 \cup \cdots \cup S_{i-1} \cup d_k(S_i)$ is consistent and $S_0 \cup S_1 \cup \cdots \cup S_{i-1} \cup d_{k-1}(S_i)$ is inconsistent. Then, applying Adjustment on K'_φ is equivalent to applying Adjustment on:

$$\big(S_0, S_1, \ldots, S_{i-1}, d_k(S_i), (d_{k-1}(S_i) \vee \mathbb{R}_{i+1})\big).$$

Example 4 (*continued*). We have $K'_\varphi = (S_0, S_1, L_2)$ where $L_2 = (d_3(S_2), d_2(S_2))$ ($R_3 = \emptyset$). $S_0 \cup S_1 \cup d_2(S_2)$ is consistent. Then, $K'_\varphi \equiv (S_0, S_1, d_2(S_2))$.

The following lemma gives more precisions than the above one:

Lemma 6. *Let $K_\varphi = S_0 \cup \cdots \cup S_n$ be such that $S_0 \cup \cdots \cup S_{i-1}$ is consistent but $S_0 \cup \cdots \cup S_i$ is inconsistent. Let $m = |S_i| \geqslant k > 1$.*

Moreover, assume that $S_0 \cup \cdots \cup S_{i-1} \cup d_k(S_i)$ is consistent but $S_0 \cup \cdots \cup S_{i-1} \cup d_{k-1}(S_i)$ is inconsistent. Then, applying Adjustment on K'_φ is equivalent to applying it on

$$\big(S_0, S_1, \ldots, S_{i-1}, d_k(S_i), \mathbb{R}_{i+1}\big).$$

Now after replacing $\mathbb{R}_{i+1}$ by its expression, we get one of the main results of this paper:

Theorem 1. *Let $K_\varphi = (S_0, \ldots, S_n)$ be such that $S_0 \cup \cdots \cup S_{i-1}$ is consistent but $S_0 \cup \cdots \cup S_i$ is inconsistent. Let $m = |S_i|$. Let $k \leqslant m$ and $k > 1$. Let K'_φ be the base associated with K_φ following step* 1.

If $S_0 \cup \cdots \cup S_{i-1}$ is consistent with $d_k(S_i)$ but inconsistent with $d_{k-1}(S_i)$ then applying Adjustment on K'_φ is equivalent to applying Adjustment on

$$\big(S_0, S_1, \ldots, S_{i-1}, d_k(S_i), L_{i+1}, S_{i+1}, \ldots, L_n, S_n\big).$$

Namely, when we meet inconsistency at the level of S_i we replace this stratum by the set of its disjunctions having the minimal size and consistent with $S_0 \cup \cdots \cup S_{i-1}$.

When Theorem 1 is applied repeatedly it shows that the consistent base computed from K'_φ using Adjustment is computed level by level. We start with the first level. Formulas of some level are added if they are consistent with the selected base. Otherwise, we consider their disjunctions if they are consistent with the selected base. The whole stratum is ignored when the disjunction of all its formulas is still inconsistent with the selected base.

Indeed, this process of computing $\delta_A(K'_\varphi)$ is the same as computing $\delta_{WDMA}(K_\varphi)$. We then showed that $\delta_{WDMA}(K_\varphi)$ is equivalent to $\delta_{DMA}(K_\varphi)$.

Example 4 (*continued*). We have

$$\delta_A(K'_\varphi) = S_0 \cup S_1 \cup d_2(S_2) = \{\neg c, \neg a \vee \neg b \vee c, a \vee b, a \vee g, b \vee g\}.$$

Since $C = \{\neg c, \neg a \vee \neg b \vee c, a, b\}$ is inconsistent, then all disjunctions constructed from g and this conflict C are reduced to g. Namely, the formulas $a \vee g$ and $b \vee g$ are reduced to g since $\{\neg c, \neg a \vee \neg b \vee c\} \vdash \neg a \vee \neg b$ and $\{a \vee g, b \vee g, \neg a \vee \neg b\}$ is equivalent to $\{g\}$.

Therefore, we have $\delta_A(K'_{\neg c}) \equiv \{\neg c, \neg a \vee \neg b \vee c, a \vee b, g\}$ which is equivalent to $\delta_{DMA}(K_{\neg c})$.

7. Experimental results

We now present some experimental results which illustrate the different behaviour of each strategy. We used a propositional logic implementation of the strategies.[4] We chose 8 inconsistent bases at random from the DIMACS challenge [18] (aim-50-no), denoted t1 to t8 in the following tables, containing 50 variables each and 80 clauses for the first 4, 100 clauses for the others. Then we stratified the bases with 20 clauses per strata, keeping the clauses in their original order. It appeared that each time the conflicts were discovered and weakened in the second strata, no more appeared in the remaining strata. Table 1 gives the number of clauses in the second strata after applying a given strategy. WDMA (respectively IDMA) stands for whole-DMA (respectively iterative-DMA). The zeros appearing in the first row for the adjustment policy simply reflect the fact the adjustment approach does not keep any information in and below the inconsistent stratum.

There are no differences between DMA and IDMA because on these examples consistency was either restored using $d_2(C)$ (t2, t3, t5, t6, t7) or all the clauses involved in

[4] ADS: http://cafe.newcastle.edu.au/daniel/ADS/.

Table 1
Number of clauses added for all strategies in the second stratum on eight examples

#clauses	t1	t2	t3	t4	t5	t6	t7	t8
Adj.	0	0	0	0	0	0	0	0
MA	17	7	8	18	13	7	10	17
DMA	17	54	49	18	21	60	35	18
WDMA	168	149	153	161	161	155	152	160
IDMA	17	54	49	18	21	60	35	18

Table 2
Time spent to apply the various strategies on eight examples

time (s)	t1	t2	t3	t4	t5	t6	t7	t8
Adj.	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
MA	137	0.6	2.0	332	6.1	0.3	1.2	304
DMA	136	0.6	2.1	329	6.2	0.3	1.2	302
WDMA	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
IDMA	139	0.6	2.1	329	6.0	0.3	1.2	306

a conflict had to be removed. Whole-DMA clearly hides the information contained in the knowledge base by generating a large number of clauses but timewise its fast.

Table 2 provides the time spent computing each strategy. The zeros in row one results from two calls to a SAT solver on a very small knowledge base with few variables. It is almost the same for row 4 (whole-DMA), with a little overhead due to the computation of the disjunctions to add in the stratum.

These results can be interpreted as follows: computing the set of clauses involved in conflicts (kernel) is costly, so all methods relying on this information will require small KBs to revise. This can be achieved for instance using modular KBs, a common practice in knowledge engineering.

Interestingly, since the three DMA approaches we introduced are logically equivalent, we can propose one way to efficiently compute the DMA policy: whole-DMA, which is only based on satisfiability testing (provided that the produced knowledge base is not very large), current SAT solvers being able to solve some problems with tens of thousand of variables and hundreds of thousand of clauses. This method can be used for instance if the initial formulas in the knowledge base are not important, and that only the queries are important. Namely, with WDMA it is basically impossible to provide a reason (or justification) of some answers to queries, since initial formulas likely no longer belong (syntactically) to the compiled knowledge base. On the other hand, if the knowledge base itself is important for the user, such that the revised base must be as “close” as possible to the original one, an IDMA approach should be used (only necessary information will be weakened), but a computational cost must be paid.

8. Complexity issues

One of the major issues concerning the WDMA approach is that adding disjunctions to the database may end up with a database whose size is exponentially larger than the original one. So our compilation of stratified bases under the lexicographic preference requires exponential space in the worst case.

A recent work from Coste-Marquis and Marquis [13] extending [12] proves (Proposition 7) that if there is no way to compile any stratified knowledge base under inclusion preference[5] in polynomial space, such a translation is possible under lexicographic preference by adding new propositional variables (the compiled base is "query equivalent" to the original one). The idea is first to determine for each stratum S_i the number p_i of formulas from S_i that belongs to every preferred subbases of K w.r.t. lexicographic preference. Then for each formula $\phi_{i,j}$ from S_i a new variable $holds_{i,j}$ is created. The compiled knowledge base K'' consists of formulas $holds_{i,j} \rightarrow \phi_{i,j}$ $\forall \phi_{i,j} \in S_i$ and cardinality formulas encoding that exactly p_i variables in $\phi_{i,1}, \ldots, \phi_{i,|S_i|}$ are true.

In the light of this result, our approach is questionable: if a polynomial size compilation exists, what is the point of using an exponential one? In our case, we do use classical entailment on the compiled base. If the original stratified base contains n variables, then this operation has a time complexity in $O(2^n)$. Now if one wants to use classical entailment on a database where $k = |K|$ variables have been added, the time complexity of that operation becomes $O(2^{n+k})$. This is exponentially worse than with our approach.

So both approaches have their advantage: ours is suitable when the compiled base will be used heavily for entailment purposes, the Coste–Marquis one when the size of the compiled base matters.

9. Conclusion

We introduced a new family of computationally effective strategies for conflict resolution which can be used for iterated belief revision and merging information from multiple sources.

The most important feature of our strategy is that it relies on weakening conflicting information rather than removing conflicts completely, and hence it retains at least as much, and in most cases more, information than all other known strategies.

We compared and contrasted three implementations of our new strategy with existing ones from a theoretical standpoint and by measuring their relative performance.

We were also able to show the surprising result that the DMA policy provides a compilation of the lexicographical system which is known to have desirable theoretical properties. DMA offers the clear advantage of obviating the need to explicitly compute the set of all preferred subbases which can be hard. Another pleasing result is that the DMA strategy can be implemented as whole-DMA where the need to explicitly compute the

[5] Inclusion preference corresponds to replace in Definition 6 the cardinality criteria by the inclusion set criteria. The selected subbases are called preferred subbases [9].

culprits responsible for the conflicts is not required. However, on the other hand, whole-DMA can produce a large number of disjunctions. Whole-DMA is interesting when the number of formulas in a stratum is small and when the number of conflict in a given stratum is low.

Acknowledgements

The authors thank anonymous referees for insightful comments that led to a substantial improvement of the paper. Salem Benferhat and Daniel Le Berre have been supported in part by the IUT de Lens, the CNRS and the Region Nord/Pas-de-Calais under the "TACT Programme".

Appendix A

Lemma 1. *Let K be a κ-ranked base and (ϕ, k) be a subsumed formula in K. Let $K' = K - \{(\phi, k)\}$. Let κ_K and $\kappa_{K'}$ be the OCF associated to K and K' respectively following Definition* 1. *Then,*

$$\forall \omega, \quad \kappa_K(\omega) = \kappa_{K'}(\omega).$$

Proof. Assume that (ϕ, k) is subsumed in K, and let $K' = K - \{(\phi, k)\}$. The equivalence is obtained by showing that (ϕ, k) is not involved in computing $k_\kappa(\omega)$, for any interpretation ω.

Let ω be a given interpretation. We distinguish two cases:

- $\omega \models \phi$.
  If $\forall (\phi_i, k_i) \in K$, we have $\omega \models \phi_i$ then $\kappa_K(\omega) = \kappa_{K'}(\omega) = 0$. Now assume that ω is not a model of K. Then by definition:

$$\begin{aligned}\kappa_K(\omega) &= \max\{k_i\colon\ (\phi_i, k_i) \in K, \omega \not\models \phi_i\} \\ &= \max\{k_i\colon\ (\phi_i, k_i) \in K - \{(\phi, k)\}, \omega \not\models \phi_i\} \quad \text{(since } \omega \models \phi_i\text{)} \\ &= \kappa_{K'}(\omega).\end{aligned}$$

- $\omega \not\models \phi$. Then by definition:

$$\begin{aligned}\kappa_K(\omega) &= \max\{k_i\colon\ (\phi_i, k_i) \in K, \omega \not\models \phi_i\} \\ &= \mathrm{Max}\big(\max\{k_i\colon\ (\phi_i, k_i)\colon\ (\phi_i, k_i) \in K - \{(\phi, k)\}, \omega \not\models \phi_i\}, k\big) \\ &= \max\{k_i\colon\ (\phi_i, k_i)\colon\ (\phi_i, k_i) \in \{(\phi_i, k_i) \in K - \{(\phi, k)\}, \omega \not\models \phi_i\} \\ &\quad \text{(since } K_{\geqslant k} - \{(\phi, k)\} \models \phi \text{ and } \omega \not\models \phi \text{ implies that there exists at} \\ &\quad \text{least a formula } (\phi_j, k_j) \text{ with } k_j > k \text{ such that } \omega \not\models \phi_j\text{)} \\ &= \kappa_{K'}(\omega). \qquad \square\end{aligned}$$

Lemma 2. *Let $KB \cup S$ be inconsistent. Let C be the maximal subset of S in Kernel$(KB \cup S)$, and F be the set of remaining free formulas in S. Namely $C = Kernel(KB \cup S) \cap S$ and $F = S - C$. If $KB \cup d_j(C) \cup F$ is inconsistent then,*

$$\forall \phi, \phi \in F, \text{ we have } \phi \text{ is also free in } KB \cup d_j(C) \cup F.$$

Proof. Let $\phi \in F$. Suppose that ϕ is not free in $KB \cup d_j(C) \cup F$. This means that there exists a conflict *Conf* in $KB \cup d_j(C) \cup F$ which involves ϕ. Let A be the subset of $d_j(C) \cup F$ in *Conf*, i.e., $A = \{\phi\} \cup F' \cup d'_j(C)$ where $d'_j(C) \subseteq d_j(C)$ and $F' \subset F$.

Note that $d'_j(C)$ is a conjunction of formulas each one is the disjunction of size j between some formulas of C. $d'_j(C)$ can be equivalently rewritten as $\Psi_1 \vee \cdots \vee \Psi_m$, where Ψ_l's ($l = 1, \ldots, m$) are conjunctions of formulas of C. This rewriting is simply obtained by distributing the conjunct symbol inside the disjunction of formulas. For instance, if $d'_j(C) = \{\phi_1 \vee \phi_2, \phi_3 \vee \phi_4\}$, this can be equivalently written as $\{\phi_1\phi_3 \vee \phi_1\phi_4 \vee \phi_2\phi_3 \vee \phi_2\phi_4\}$.

The fact that $KB \cup \{\phi\} \cup F' \cup d'_j(C)$ is a conflict implies that $KB \cup \{\Psi_1 \vee \cdots \vee \Psi_m\} \cup F'$ is consistent (minimality condition), which means that $KB \cup F'$ is consistent with at least one Ψ_k.

Now $KB \cup \{\phi\} \cup F' \cup \{\Psi_1 \vee \cdots \vee \Psi_m\}$ is inconsistent means that for all $l = 1, \ldots, m$ we have $KB \cup \{\phi\} \cup F' \cup \{\Psi_l\}$ is inconsistent. Then $KB \cup \{\phi\} \cup F' \cup \{\Psi_k\}$ is inconsistent which means that there is a conflict in $KB \cup \{\phi\} \cup F' \cup \{\Psi_k\}$ involving KB, F' and ϕ and some formulas of Ψ_k (since $KB \cup \{\phi\} \cup F'$ is consistent and $KB \cup \{\Psi_k\} \cup \{F'\}$ is consistent). However this contradicts the hypothesis that ϕ is free in S. □

Proposition 1. *Let KB be consistent but inconsistent with S. Let C be the subset of S in kernel$(KB \cup S)$, and $F = S - C$ be the set of remaining free formulas in S. Let $d_j(C)$ (respectively $d_j(S)$) be the set of all possible disjunctions of size j from C (respectively S). Then, if $KB \cup d_{j-1}(S)$ is inconsistent then*

$$KB \cup d_j(C) \cup F \equiv KB \cup d_j(S).$$

Proof. We give a recursive proof. First we show the proposition for $j = 2$. Namely we suppose that $KB \cup d_1(S)$ is inconsistent and show that $KB \cup d_2(C) \cup F \equiv KB \cup d_2(S)$. Note that $d_1(S) = S$.

Let $C = \{\psi_1, \ldots, \psi_m\}$. Since C is the subset of S in *Kernel*$(KB \cup S)$ then $KB \vdash \neg\psi_1 \vee \cdots \vee \neg\psi_m$.

Let $\phi \in F$. Then $\{\phi \vee \psi_1, \ldots, \phi \vee \psi_m\}$ belongs to $d_2(S)$.

Now applying successive resolutions between $\neg\psi_1 \vee \cdots \vee \neg\psi_m$ and $\{\phi \vee \psi_1, \ldots, \phi \vee \psi_m\}$ leads to ϕ. Hence formulas of F are entailed from $KB \cup d_2(S)$. So we can add explicitly F to $KB \cup d_2(S)$. We get $KB \cup d_2(S) \equiv KB \cup d_2(S) \cup F$.

Note that $d_2(S) = d_2(C) \cup d_2(F) \cup d_2(\{F, C\})$ where $d_2(\{F, C\})$ is the set of disjunctions of size 2 involving both formulas of F and C. Then,

$$KB \cup d_2(S) \cup F \equiv KB \cup d_2(C) \cup d_2(F) \cup d_2\big(\{F, C\}\big) \cup F$$

which is equivalent to $KB \cup d_2(C) \cup F$ since formulas of $d_2(F)$ and $d_2(\{F, C\})$ are subsumed by formulas of F.

Suppose now that the proposition is true for j and show that it is also the case for $j+1$.

Let $KB \cup d_j(S) \equiv KB \cup d_j(C) \cup F$ be inconsistent and let us show that $KB \cup d_{j+1}(C) \cup F \equiv KB \cup d_{j+1}(S)$.

From Lemma 2, formulas of F are free in $KB \cup d_j(C) \cup F$. Let $A = \{\psi_1, \ldots, \psi_m\}$ be the subset of $d_j(C)$ in a conflict in $KB \cup d_j(C) \cup F$. A only involves formulas from $d_j(C)$. Then $KB \vdash \neg\psi_1 \vee \cdots \vee \neg\psi_m$.

Let $\phi \in F$. Then $\{\phi \vee \psi_1, \ldots, \phi \vee \psi_m\}$ belongs to $d_{j+1}(S)$. Now applying successive resolutions between $\neg\psi_1 \vee \cdots \vee \neg\psi_m$ and $\{\phi \vee \psi_1, \ldots, \phi \vee \psi_m\}$ we get ϕ. Then all formulas of F are entailed from $KB \cup d_{j+1}(S)$. So we can add explicitly F to $KB \cup d_{j+1}(S)$.

We have $KB \cup d_{j+1}(S) \equiv KB \cup d_{j+1}(S) \cup F$. Note that $S = C \cup F$. Then,

$$d_{j+1}(S) = d_{j+1}(C) \cup d_{j+1}(F) \cup d_{j+1}\big(\{C, F\}\big)$$

where $d_{j+1}(\{C, F\})$ is the set of disjunctions of size $(j+1)$ involving formulas of both C and F. Then,

$$\begin{aligned} KB \cup d_{j+1}(S) &\equiv KB \cup d_{j+1}(S) \cup F \\ &\equiv KB \cup d_{j+1}(C) \cup d_{j+1}(F) \cup d_{j+1}\big(\{C, F\}\big) \cup F \end{aligned}$$

which is equivalent to $KB \cup d_{j+1}(C) \cup F$ since formulas of $d_{j+1}(F)$ and $d_{j+1}(\{C, F\})$ are subsumed by formulas of F. □

Lemma 3. *Let A be a set of formulas. Let $B = d_i(A)$ and $C = d_{i+1}(A)$ be the set of all possible disjunctions of A of size i and $i+1$ respectively. Then, $C \equiv d_2(B)$.*

Proof.

- Let $\phi \in d_{i+1}(A)$. Then, ϕ is of the form $\phi_1 \vee \cdots \vee \phi_{i+1}$ such that $\phi_j \in A$ for $j = 1, \ldots, i+1$. Let us show that $\phi \in d_2(B)$. For this, it is enough to show that there exist $\psi \in d_i(A)$ and $\psi' \in d_i(A)$ such that $\phi \equiv \psi \vee \psi'$.
  Indeed, let $\psi = \phi_1 \vee \cdots \vee \phi_{j-1} \vee \phi_{j+1} \vee \cdots \vee \phi_{i+1} \in d_i(A)$ and $\psi' = \phi_1 \vee \cdots \vee \phi_j \vee \phi_{j+2} \vee \cdots \vee \phi_{i+1} \in d_i(A)$ since both formulas are disjunctions of i formulas from A. Note that ψ and ψ' contain the same formulas except ϕ_j and ϕ_{j+1}. ϕ_{j+1} only belongs to the first disjunction and ϕ_j only belongs to the second one. Then, the disjunction of these two formulas belongs to $d_2(d_i(A))$. It is equivalent to $\phi_1 \vee \cdots \vee \phi_{j-1} \vee \phi_j \vee \phi_{j+1} \vee \cdots \vee \phi_{i+1}$ which is simply the formula ϕ.
- Let $\phi \in d_2(B)$. Namely, ϕ is of the form $\psi \vee \psi'$ where ψ and ψ' belongs to B. We recall that ψ and ψ' are in $d_i(A)$, namely each of them is a disjunction of i formulas of A.
  We distinguish two cases:
  - $\psi = \psi_1 \vee \cdots \vee \psi_{i-1} \vee \psi_i$ and $\psi' = \psi_1 \vee \cdots \vee \psi_{i-1} \vee \psi'_i$. Namely, ψ and ψ' contain $(i-1)$ same formulas from A, and only differ on one formula of A. Then $\psi \vee \psi' \equiv \phi_1 \vee \cdots \vee \phi_{i-1} \vee \phi_i \vee \phi'_i$, hence there exists a formula equivalent to $\psi \vee \psi'$, in $d_{i+1}(A)$.
  - $\psi = \psi_1 \vee \cdots \vee \psi_{i-1} \vee \psi_i$ and $\psi' = \psi'_1 \vee \cdots \vee \psi'_{i-1} \vee \psi'_i$ differ on strictly more than one formula. Namely $\psi \vee \psi'$ is of size greater than $i+1$, then using the first

part of the proof $\psi \vee \psi'$ is necessary subsumed by some formulas of size $(i+1)$ in $d_2(B)$ since all formulas of size $(i+1)$ are in $d_2(B)$. □

Proposition 2. *Let $KB \cup F \cup d_i(C)$ be inconsistent. Let C' be the subset of $d_i(C)$ in kernel$(KB \cup F \cup d_i(C))$, and $F' = d_i(C) - C'$ be the set of remaining formulas. Then,*

$$KB \cup F \cup d_{i+1}(C) \equiv KB \cup F \cup F' \cup d_2(C').$$

Proof. Suppose that $KB \cup K \cup d_i(C)$ is inconsistent. Let C' be the subset of $d_i(C)$ in *kernel*$(KB \cup F \cup d_i(C))$, and $F' = d_i(C) - C'$ be the set of remaining formulas. From Lemma 3, we have $KB \cup F \cup d_{i+1}(C) \equiv KB \cup F \cup d_2(d_i(C))$.

Now, using Proposition 1 we get: $KB \cup F \cup d_2(d_i(C)) \equiv KB \cup F \cup F' \cup d_2(C')$. □

The proof of Proposition 4 is based on the following lemma:

Lemma A.1. *Let $K = \{S_1, \ldots, S_n)$ be a stratified base and φ be a new formula. Let K' be the base constructed from K using Definition* 9. *Then,*

$$\kappa_{K'_\varphi}(\omega) = \sum_{i=0}^{n} f_i * k_i,$$

where $k_0 = +\infty$ and f_i is the number of formulas in S_i falsified by ω.

Proof. By definition, $\kappa_{K'_\varphi}(\omega)$ corresponds to the highest weight in K'_φ whose associated formula is falsified by ω.

If ω falsifies φ then it is considered as not satisfactory at all. So, we associate the degree $+\infty$ to $\kappa_{K'_\varphi}(\omega)$.

Suppose now that ω satisfies φ. Recall that by construction, K' is composed of all possible disjunctions between formulas of K. Then, the highest weight in K'_φ whose corresponding formula is falsified by ω is the disjunction of all formulas in K falsified by ω.

Recall that we associate the weights k_i to formulas of S_i in K, and the weight of a constructed disjunction in K' is equal to the sum of weights of formulas composing this disjunction. Hence, $\omega \not\models \psi$ where ψ is the disjunction of all formulas of K falsified by ω. The weight of ψ is equal to $\sum_{i=1}^{n} f_i * k_i$ where f_i is the number of formulas in the stratum S_i falsified by ω.

For a generalization, we have $\kappa_{K'_\varphi}(\omega) = \sum_{i=0}^{n} f_i * k_i$ with $k_0 = +\infty$. □

Proposition 4. *Let $K = (S_1, \ldots, S_n)$ be a stratified base and φ be a new formula. Let K' be the base constructed from K using Definition* 9. *Then,*

$$K'_\varphi \vdash_A \psi \quad \textit{iff} \quad K_\varphi \vdash_{Lex} \psi.$$

Proof. To show this equivalence, we will show that

$$\forall \omega, \omega', \kappa_{K'}(\omega) < \kappa_{K'}(\omega') \quad \text{iff} \quad \omega >_{Lex,K} \omega'.$$

(1) Suppose that $\kappa_{K'}(\omega) < \kappa_{K'}(\omega')$ and $\omega \leqslant_{Lex,K} \omega'$. Let A_ω and $A_{\omega'}$ be two maximal consistent subbases of K satisfied by ω and ω' respectively. Then, from Definition 8 we have $\omega \leqslant_{Lex,K} \omega'$ iff $A_\omega \leqslant_{Lex} A_{\omega'}$. $A_\omega \leqslant_{Lex} A_{\omega'}$ means that either $A_\omega =_{Lex} A_{\omega'}$ or $A_\omega <_{Lex} A_{\omega'}$. Let $A_\omega = (A_1, \ldots, A_n)$ and $A_{\omega'} = (A'_1, \ldots, A'_n)$.

- Suppose that $A_\omega =_{Lex} A_{\omega'}$. Then, by definition of lexicographical ordering we have $A_\omega =_{Lex} A_{\omega'}$ iff $\forall j, j = 1, \ldots, n, |A_j| = |A'_j|$. This means that ω and ω' satisfy the same number of formulas in each stratum of K. Then, they also falsify the same number of formulas in each stratum of K. Hence, $\kappa_{K'}(\omega) = \kappa_{K'}(\omega') = \sum_{i=1}^{n}(|S_i| - |A_i|) * k_i$. However, this contradicts the hypothesis $\kappa_{K'}(\omega) < \kappa_{K'}(\omega')$.
- Suppose now that $A_\omega <_{Lex} A_{\omega'}$. Then, by definition of lexicographical ordering we have $A_\omega <_{Lex} A_{\omega'}$ iff $\exists k, |A_k| < |A'_k|$ and $\forall j, j < k$ we have $|A_j| = |A'_j|$.
  Let us now compute $\kappa_{K'}(\omega)$ and $\kappa_{K'}(\omega')$. We have

$$\kappa_{K'}(\omega) = \sum_{i=1}^{n}\big(|S_i| - |A_i|\big) * k_i \quad \text{and}$$

$$\kappa_{K'}(\omega') = \sum_{i=1}^{n}\big(|S_i| - |A'_i|\big) * k_i.$$

  We have $|A_k| < |A'_k|$, then $|S_k| - |A_k| > |S_k| - |A'_k|$.
  Also, $|S_j| - |A_j| = |S_j| - |A'_j|$ for $j < k$. Then,

$$\sum_{i=1}^{k}\big(|S_i| - |A_i|\big) * k_i > \sum_{i=1}^{k}\big(|S_i| - |A'_i|\big) * k_i.$$

  Hence, we have

$$\sum_{i=1}^{k}\big(|S_i| - |A_i|\big) * k_i > \sum_{i=1}^{n}\big(|S_i| - |A'_i|\big) * k_i$$

  since $k_i > \sum_{j=i+1}^{n}(|S_j| * k_j$ then $k_i > \sum_{j=i+1}^{n}(|S_j| - |A'_j|) * k_j$. Then,

$$\sum_{i=1}^{n}\big(|S_i| - |A_i|\big) * k_i > \sum_{i=1}^{n}\big(|S_i| - |A'_i|\big) * k_i.$$

  Hence, $\kappa_{K'}(\omega) > \kappa_{K'}(\omega')$ which contradicts the hypothesis $\kappa_{K'}(\omega) < \kappa_{K'}(\omega')$.

(2) Suppose that $\omega >_{Lex,K} \omega'$ and $\kappa_{K'}(\omega) \geqslant \kappa_{K'}(\omega')$. Let A_ω and $A_{\omega'}$ be the maximal subbases of K satisfied by ω and ω' respectively. Let $A_\omega = (A_1, \ldots, A_n)$ and $A_{\omega'} = (A'_1, \ldots, A'_n)$. Then, from Lemma A.1 we have:

$$\begin{aligned}\kappa_{K'}(\omega) &= f_1 * k_1 + \cdots + f_n * k_n \quad (f_0 = 0 \text{ since } \varphi \text{ is not considered})\\ &= \big(|S_1| - |A_1|\big) * k_1 + \cdots + \big(|S_n| - |A_n|\big) * k_n\\ &= \big(|S_1| * k_1 + \cdots + |S_n| * k_n\big) - \big(|A_1| * k_1 + \cdots + |A_n| * k_n\big)\end{aligned}$$

and

$$\begin{aligned}\kappa_{K'}(\omega') &= f'_1 * k_1 + \cdots + f'_n * k_n \quad (f'_0 = 0 \text{ since } \varphi \text{ is not considered})\\ &= \big(|S_1| - |A'_1|\big) * k_1 + \cdots + \big(|S_n| - |A'_n|\big) * k_n\\ &= \big(|S_1| * k_1 + \cdots + |S_n| * k_n\big) - \big(|A'_1| * k_1 + \cdots + |A'_n| * k_n\big).\end{aligned}$$

We have $\kappa_{K'}(\omega) \geqslant \kappa_{K'}(\omega')$ iff

$$-\big(|A_1| * k_1 + \cdots + |A_n| * k_n\big) \geqslant -\big(|A'_1| * k_1 + \cdots + |A'_n| * k_n\big) \quad \text{iff}$$
$$|A'_1| * k_1 + \cdots + |A'_n| * k_n \geqslant |A_1| * k_1 + \cdots + |A_n| * k_n.$$

Then, we distinguish two cases:

(1) $$|A'_1| * k_1 + \cdots + |A'_n| * k_n = |A_1| * k_1 + \cdots + |A_n| * k_n. \qquad (*)$$

Since the weights k_i are such that there is no compensation. Then, $(*)$ means that $|A'_1| = |A_1|, \ldots, |A'_n| = |A_n|$. Hence, $A_\omega =_{Lex,K} A_{\omega'}$ which means that $\omega =_{Lex,K} \omega'$ (using Definition 8). However, this contradicts the hypothesis $\omega >_{Lex,K} \omega'$.

(2) $$|A'_1| * k_1 + \cdots + |A'_n| * k_n > |A_1| * k_1 + \cdots + |A_n| * k_n. \qquad (**)$$

Also, since there is no compensation between the weights k_i then $(**)$ means that there exists l such that $|A'_j| = |A_j|$ for $j = 1, \ldots, l-1$ and $|A'_l| > |A_l|$. This means that $A_{\omega'} >_{Lex,K} A_\omega$ which is equivalent to $\omega' >_{Lex,K} \omega$ (using Definition 8). However, this contradicts the hypothesis $\omega >_{Lex,K} \omega'$. □

Lemma 4. *Let $K_\varphi = (S_0, S_1, \ldots, S_n)$ be a stratified base, and $K'_\varphi = (S_0, L_1, S_1, \ldots, L_n, S_n)$ be the base associated with K_φ using Definition 9. Assume that $S_0 \cup \cdots \cup S_{i-1}$ is consistent and $S_0 \cup \cdots \cup S_i$ is inconsistent. Then, applying Adjustment on K'_φ is equivalent to applying Adjustment on $(S_0, S_1, \ldots, S_{i-1}, L_i)$.*

Proof. The first simplification is justified by the fact that when inconsistency is met at some level then all formulas which are below the inconsistency level are removed by Adjustment.

Suppose now that $S_0 \cup S_1 \cup \cdots \cup S_{i-1}$ is consistent. Indeed, $S_0 \cup S_1 \cup \cdots \cup S_{i-1}$ will certainly belong to $\delta_A(K'_\varphi)$ since their formulas are the consistent prioritized ones. Hence, there is no need to add L_j for $j = 1, \ldots, i-1$ (which are also consistent together with $S_0 \cup S_1 \cup \cdots \cup S_{i-1}$) since they will be classically subsumed by S_j (recall that L_j is the set of all possible disjunctions between formulas of $S_j \cup \cdots \cup S_n$ containing at least one element from S_j). □

Lemma 5. *Let $m = |S_i| \geqslant k > 1$. Let $K_\varphi = (S_0, \ldots, S_n)$ be such that $S_0 \cup \cdots \cup S_{i-1}$ is consistent but $S_0 \cup \cdots \cup S_i$ is inconsistent. Let $K'_\varphi = (S_0, \ldots, S_{i-1}, d_m(S_i) \vee \mathbb{R}_{i+1}, d_m(S_i), \ldots, d_2(S_i) \vee \mathbb{R}_{i+1}, d_2(S_i), d_1(S_i) \vee \mathbb{R}_{i+1})$.*

Assume that $S_0 \cup S_1 \cup \cdots \cup S_{i-1} \cup d_k(S_i)$ is consistent and $S_0 \cup S_1 \cup \cdots \cup S_{i-1} \cup d_{k-1}(S_i)$ is inconsistent. Then, applying Adjustment on K'_φ is equivalent to applying Adjustment on:

$$\big(S_0, S_1, \ldots, S_{i-1}, d_k(S_i), (d_{k-1}(S_i) \vee \mathbb{R}_{i+1})\big).$$

Proof. Assume that $S_0 \cup S_1 \cup \cdots \cup S_{i-1} \cup d_k(S_i)$ is consistent and $S_0 \cup S_1 \cup \cdots \cup S_{i-1} \cup d_{k-1}(S_i)$ is inconsistent.

We know that Adjustment ignores formulas which are under the inconsistency level. Indeed, since $S_0 \cup S_1 \cup \cdots \cup S_{i-1} \cup d_{k-1}(S_i)$ is inconsistent then applying Adjustment on

$$K'_\varphi = \big(S_0, S_1, \ldots, S_{i-1}, (d_m(S_i) \vee \mathbb{R}_{i+1}), d_m(S_i), \ldots, (d_2(S_i) \vee \mathbb{R}_{i+1}), d_2(S_i), (d_1(S_i) \vee \mathbb{R}_{i+1})\big)$$

is equivalent to applying Adjustment on

$$\big(S_0, S_1, \ldots, S_{i-1}, (d_m(S_i) \vee \mathbb{R}_{i+1}), d_m(S_i), \ldots, (d_k(S_i) \vee \mathbb{R}_{i+1}), d_k(S_i), (d_{k-1}(S_i) \vee \mathbb{R}_{i+1})\big).$$

Now since $S_0 \cup S_1 \cup \cdots \cup S_{i-1} \cup d_k(S_i)$ is consistent, consistent with $(d_j(S_i) \vee \mathbb{R}_{i+1}) \cup d_j(S_{i+1})$ for $j > k$, and their formulas are the most prioritized consistent ones in K'_φ then they will belong to $\delta_A(K'_\varphi)$. Indeed, formulas of $(d_j(S_i) \vee \mathbb{R}_{i+1}) \cup d_j(S_{i+1})$ for $j > k$ are subsumed by $d_k(S_i)$ in $\delta_A(K'_\varphi)$. □

Lemma 6. *Let $K_\varphi = S_0 \cup \cdots \cup S_n$ be such that $S_0 \cup \cdots \cup S_{i-1}$ is consistent but $S_0 \cup \cdots \cup S_i$ is inconsistent. Let $m = |S_i| \geqslant k > 1$.*

Moreover, assume that $S_0 \cup \cdots \cup S_{i-1} \cup d_k(S_i)$ is consistent but $S_0 \cup \cdots \cup S_{i-1} \cup d_{k-1}(S_i)$ is inconsistent. Then, applying Adjustment on K'_φ is equivalent to applying it on

$$\big(S_0, S_1, \ldots, S_{i-1}, d_k(S_i), \mathbb{R}_{i+1}\big).$$

Proof. Indeed, since $S_0 \cup \cdots \cup S_{i-1} \cup d_{k-1}(S_i)$ is inconsistent then $S_0 \cup \cdots \cup S_{i-1} \vdash \neg d_{k-1}(S_i)$. Hence, $(d_{k-1}(S_i) \vee \mathbb{R}_{i+1})$ in K'_φ is equivalent to $\mathbb{R}_{i+1}$. □

References

[1] C. Alchourrón, P. Gärdenfors, D. Makinson, On the logic of theory change: Partial meet functions for contraction and revision, J. Symbolic Logic 50 (1985) 510–530.

[2] C. Baral, S. Kraus, J. Minker, V.S. Subrahmanian, Combining knowledge bases consisting in first order theories, Comput. Intelligence 8 (1) (1992) 45–71.

[3] S. Benferhat, D. Dubois, C. Cayrol, J. Lang, H. Prade, Inconsistency management and prioritized syntax-based entailment, in: Proceedings of IJCAI-93, Chambéry, France, 1993, pp. 640–645.

[4] S. Benferhat, D. Dubois, H. Prade, Some syntactic approaches to the handling of inconsistent knowledge bases: A comparative study part 2: The prioritized case, in: E. Orlowska (Ed.), Logic at Work, vol. 24, Physica-Verlag, 1998, pp. 473–511.

[5] S. Benferhat, S. Kaci, D. Le Berre, M. Williams, Weakening conflicting information for iterated revision and knowledge integration, in: Proceedings of IJCAI-01, Seattle, WA, 2001, pp. 109–115.

[6] B. Bessant, E. Grégoire, P. Marquis, L. Sais, Iterated syntax-based revision in a nonmonotonic setting, in: M.-A. Williams, H. Rott (Eds.), Frontiers in Belief Revision, Kluwer Academic, Dordrecht, 2001, pp. 369–391.

[7] R. Bourne, Default reasoning using maximum entropy and variable strength defaults, PhD Thesis, University of London, 1999.

[8] R. Bourne, S.D. Parsons, Connecting lexicographical and maximum entropy entailment, in: Proceedings of the fifth European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty, (ECSQARU'99), in: Lecture Notes in Artificial Intelligence, vol. 1638, Springer, Berlin, 1999, pp. 80–91.

[9] G. Brewka, Preferred subtheories: An extended logical framework for default reasoning, in: Proceedings of IJCAI-89, Detroit, MI, 1989, pp. 1043–1048.
[10] P.D. Bruza, R. Lau, A.H.M. ter Hofstede, K.F. Wong, Belief revision and possibilistic logic for adaptive information filtering agents, in: Proceedings ICTAI-2000, Vancouver, BC, 2000, pp. 19–26.
[11] L. Cholvy, Reasoning about merging information, in: Handbook of Defeasible Reasoning and Uncertainty Management Systems, vol. 3, Kluwer Academic, Dordrecht, 1998, pp. 233–263.
[12] S. Coste-Marquis, P. Marquis, Compiling stratified beliefs bases, in: Proceedings of Fourteenth European Conference on Artificial Intelligence (ECAI'00), Berlin, 2000, pp. 23–27.
[13] S. Coste-Marquis, P. Marquis, On stratified belief bases compilations, 2002, submitted for publication.
[14] A. Darwiche, J. Pearl, On stratified belief bases compilations, Artificial Intelligence 89 (1997) 1–29.
[15] D. Dubois, J. Lang, H. Prade, Possibilistic logic, in: Handbook of Logic in Artificial Intelligence and Logic Programming, vol. 3, 1994, pp. 439–513.
[16] T. Eiter, G. Gottlob, On the complexity of propositional knowledge base revision, updates, and counterfactuals, Artificial Intelligence 57 (1992) 227–270.
[17] P. Gärdenfors, Knowledge in Flux: Modeling the Dynamics of Epistemic States, Bradford Books/MIT Press, Cambridge, MA, 1988.
[18] D.S. Johnson, M.A. Trick, Second DIMACS Implementation Challenge: Cliques, Coloring and Satisfiability, in: DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 26, American Mathematical Society, Providence, RI, 1996.
[19] J. Lang, Possibilistic logic: Complexity and algorithms, in: D. Gabbay, P. Smets (Eds.), Handbook of Defeasible Reasoning and Uncertainty Management Systems, vol. 5, Kluwer Academic, Dordrecht, 2000, pp. 179–200.
[20] D. Lehmann, Another perspective on default reasoning, Ann. Math. Artificial Intelligence 15 (1995) 61–82.
[21] Nayak, Iterated belief change based on epistemic entrenchment, Erkenntnis 41 (1994) 353–390.
[22] B. Nebel, Belief revision and default reasoning: Syntax-based approaches, in: J. Allen, R. Fikes, E. Sandewall (Eds.), Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning (KR'91), Cambridge, MA, 1991, pp. 417–428.
[23] B. Nebel, How hard is it to revise a belief base, in: D. Dubois, H. Prade (Eds.), Belief Change, Handbook of Defeasible Reasoning and Uncertainty Management Systems, vol. 3, Kluwer Academic, Dordrecht, 1998, pp. 77–145.
[24] Y. Shoham, Reasoning About Change, MIT Press, Cambridge, MA, 1988.
[25] W. Spohn, Ordinal conditional functions: A dynamic theory of epistemic states, in: W.L. Harper, B. Skyrms (Eds.), Causation in Decision, Belief Change, and Statistics, vol. 2, Kluwer Academic, Dordrecht, 1988, pp. 105–134.
[26] M.A. Williams, A practical approach to belief revision: reason-based change, in: Proceedings of Fifth International Conference of Principles of Knowledge Representation and Reasoning (KR'96), Trento, Italy, 1996, pp. 412–421.
[27] M.A. Williams, Applications of belief revision, in: B. Freitag, H. Decker, M. Kifer, A. Voronkov (Eds.), Transactions and Change in Logic Databases, in: Lecture Notes in Computer Science, vol. 1472, Springer, Berlin, 1998.
[28] M.A. Williams, A. Sims, Saten: An object-oriented web-based revision and extraction engine, in: International Workshop on Nonmonotonic Reasoning (NMR'2000), 2000, Online Computer Science Abstract, http://arxiv.org/abs/cs.AI/0003059/.
[29] M.A. Williams, Transmutations of knowledge systems, in: Proceedings of Fourth International Conference on principles of Knowledge Representation and reasoning (KR'94), Morgan Kaufmann, San Mateo, CA, 1994, pp. 619–629.
[30] M.A. Williams, Iterated theory base change: A computational model, in: Proceedings of IJCAI-95, Montreal, Quebec, 1995, pp. 1541–1547.
[31] O. Wong, R. Lau, Possibilistic reasoning for intelligent payment agents, in: Proceedings of the Second Workshop on Artificial Intelligence in Electronic Commerce (AIEC'2000), 2000, pp. 1–13.

Available online at www.sciencedirect.com
SCIENCE DIRECT®
ELSEVIER
Artificial Intelligence 157 (2004) 203–237
Artificial Intelligence
www.elsevier.com/locate/artint

Qualitative choice logic

Gerhard Brewka [a,*], Salem Benferhat [b], Daniel Le Berre [b]

[a] *Universität Leipzig, Institut für Informatik, Augustusplatz 10–11, 04109 Leipzig, Germany*
[b] *CRIL-CNRS, Université d'Artois, Rue Jean Souvraz, SP 18, 62307 Lens Cedex, France*

Received 28 December 2002; accepted 14 April 2004

Abstract

Qualitative choice logic (*QCL*) is a propositional logic for representing alternative, ranked options for problem solutions. The logic adds to classical propositional logic a new connective called ordered disjunction: $A \vec{\times} B$ intuitively means: if possible A, but if A is not possible then at least B. The semantics of qualitative choice logic is based on a preference relation among models. Consequences of *QCL* theories can be computed through a compilation to stratified knowledge bases which in turn can be compiled to classical propositional theories. We also discuss potential applications of the logic, several variants of *QCL* based on alternative inference relations, and their relation to existing nonmonotonic formalisms.[1]

Keywords: Preference handling; Nonmonotonic reasoning; Qualitative decision making

1. Introduction

For many AI applications, e.g., in design or configuration, it is necessary to represent intended properties of a particular problem solution. For instance, if we want to book a hotel for a trip to a conference, we intend properties such as being close to the conference site, being close to potential sight-seeing objects, and we want the hotel reasonably priced. Most often not all of the intended properties can be satisfied, that is, we have to make some

* Corresponding author.
E-mail addresses: brewka@informatik.uni-leipzig.de (G. Brewka), benferhat@cril.univ-artois.fr (S. Benferhat), leberre@cril.univ-artois.fr (D. Le Berre).

[1] This is a revised and extended version of the paper that appeared in Proc. 8th Internat. Conference on Principles of Knowledge Representation and Reasoning (KR 2002), Toulouse, France, 2002, pp. 158–169.

0004-3702/$ – see front matter
doi:10.1016/j.artint.2004.04.006

sort of compromises. To do so it is very convenient to be able to express alternative, second (or third, etc.) best options one would like to be satisfied if the best option is unavailable. For the hotel booking example, for instance, we may want to express that we prefer to stay within walking distance of the conference site; if that is not possible transportation provided by the hotel should be available; if this is still not possible, we want at least public transportation (taxis are not being reimbursed according to our university's travel refund policy).

To represent options of this kind we introduce in this paper a new nonmonotonic propositional logic for representing qualitative choices—hence the name qualitative choice logic (*QCL*). The logic is different from existing nonmonotonic logics in the way nonmonotonicity is introduced: we do not use non-standard inference rules, as in Reiter's default logic [32], modal operators expressing consistency or belief, as in autoepistemic logic [30], or abnormality predicates whose extensions are minimized, as in circumscription [27,28]. The non-standard part of our logic is a new logical connective $\vec{\times}$ which is fully embedded in the logical language. Intuitively, if A and B are formulas then $A \vec{\times} B$ says: if possible A, but if A is impossible then (at least) B.

The intended use of this logic can be illustrated using the hotel booking example. Assume we want to represent the options concerning the location. Using mnemonic variable names we express the options as follows:

$$\mathit{walking} \vec{\times} \mathit{hotel\text{-}transport} \vec{\times} \mathit{public\text{-}transport}.$$

Assume there are 4 hotels available out of which we have to pick one:

$$\mathit{hotel}_1 \vee \mathit{hotel}_2 \vee \mathit{hotel}_3 \vee \mathit{hotel}_4.$$

We have the following information about the hotels

$$\mathit{hotel}_1 \rightarrow \mathit{walking},$$

$$\mathit{hotel}_2 \rightarrow \neg\mathit{walking} \wedge \mathit{hotel\text{-}transport},$$

$$\mathit{hotel}_3 \rightarrow \neg\mathit{walking} \wedge \neg\mathit{hotel\text{-}transport} \wedge \mathit{public\text{-}transport},$$

$$\mathit{hotel}_4 \rightarrow \neg\mathit{walking} \wedge \neg\mathit{hotel\text{-}transport} \wedge \neg\mathit{public\text{-}transport}.$$

Given these propositional formulas *QCL* will give us the conclusion hotel_1 since this is the only hotel satisfying our most intended option. Now assume that, after calling the hotel we find out that it is fully booked, that is, we have the additional information $\neg\mathit{hotel}_1$. This means that our most favoured property, being within walking distance of the conference site, cannot be satisfied. We now obtain the conclusion hotel_2 which is not exactly what we wanted but better than nothing.

Our new connective $\vec{\times}$ can be viewed as a kind of disjunction. Classical disjunction allows us to represent alternatives. The new connective uses the order in which options are written down to express additional preference information: $A \vec{\times} B$ is very different form $B \vec{\times} A$. We therefore call the connective ordered disjunction.

The semantics of the new logic will be defined in terms of preferred models. The definition of preferred models will proceed in two steps:

(1) each formula of the logic leads to a ranking of models, based on how well the models satisfy the formula,

(2) a global preference relation on models is defined on the basis of the rankings given by the single formulas.

The rest of the paper is organized as follows: we first introduce syntax and semantics for *QCL* and give a few motivating examples. We then consider aspects of computation. It turns out that the logic has a special normal form. Theories in this normal form can be translated to stratified knowledge bases. These in turn can be transformed into a classical propositional theory [5]. We thus can compute consequences of *QCL* through a compilation process in which the formulas are first translated into propositional logic. Section 4 describes potential applications of *QCL*. In Section 5 we present several alternative definitions of the consequence relation. In Section 6 we investigate the relationship between *QCL* and circumscription, in Section 7 that between *QCL* and possibilistic logic. Section 8 shows how *QCL* and Reiter's default logic [32] can be combined. Section 9 discusses other related work and concludes the paper. Proofs of propositions are contained in Appendix A.

2. Syntax and semantics of *QCL*

2.1. Syntax

We start with standard propositional logic and add a new non-standard kind of disjunction: given formulas $A_1, \ldots, A_n$[2] we will use

$$A_1 \vec{\times} \cdots \vec{\times} A_n$$

to express: some A_j must be true, preferably A_1, but if this is not possible then A_2, if this is not possible A_3, etc. Since $\vec{\times}$ is a binary operator, the formula should be read as shorthand for $(A_1 \vec{\times} (\cdots \vec{\times} A_n) \cdots)$).[3] The idea is that

- from $A \vec{\times} B$ you get A,
- from $A \vec{\times} B, \neg A$ you get B, and
- $A \vec{\times} B, \neg A, \neg B$ is inconsistent.

Clearly, the order matters. We, therefore, call $\vec{\times}$ ordered disjunction.

Ordered disjunction is fully embedded in the logical language to make sure that context dependent options can be expressed, that is we may have formulas like $A \rightarrow (B \vec{\times} C)$ and $\neg A \rightarrow (C \vec{\times} B)$. The precise definition of the syntax is as follows:

Definition 1. Let $\mathcal{V}$ be a set of atoms. The set of well-formed formulas of *QCL* is inductively defined as follows:

(1) every element of $\mathcal{V}$ is a well-formed formula,

[2] Throughout the paper we use capital letters from the beginning of the alphabet to represent formulas.

[3] We will later show that $\vec{\times}$ is associative, so the brackets do not matter.

(2) if F_1 and F_2 are well-formed formulas, then $(\neg F_1)$, $(F_1 \vee F_2)$, $(F_1 \wedge F_2)$ and $(F_1 \vec{\times} F_2)$ are well-formed formulas.

As usual we use $A \rightarrow B$ as an abbreviation for $\neg A \vee B$ and $A \leftrightarrow B$ as an abbreviation for $(A \rightarrow B) \wedge (B \rightarrow A)$. $\top$ represents a (classical) tautology, $\bot$ a (classical) contradiction. We omit unnecessary brackets assuming that all classical connectives have stronger bindings than $\vec{\times}$.

2.2. Semantics

The semantics of *QCL* is based on the degree of satisfaction of a formula in a particular (classical) model. Intuitively, the degree can be viewed as a measure of disappointment: the higher the degree the more disappointing the model (or the farther away from complete satisfaction). As in standard propositional logic, an interpretation I is an assignment of the classical truth values *true* and *false* to the atoms. We identify I with the subset of true atoms.

Interpretations can satisfy formulas to a certain degree. For instance, if A and B are atoms and I contains A then I satisfies the formula $A \vec{\times} B$ as good as possible. If I does not contain A but B, then it also satisfies $A \vec{\times} B$, but only in a suboptimal way: the second best option is now satisfied. We will say that a formula is satisfied to degree 1 if it is satisfied as good as possible, to degree 2 if it is satisfied in the second best way, etc. For classical formulas without ordered disjunction no suboptimal way of satisfaction exists. Hence, such formulas can only be satisfied to degree 1 or not satisfied at all.

For formulas containing ordered disjunction let us first consider a simple special case. Let F be a formula of the form

$$A_1 \vec{\times} \cdots \vec{\times} A_n$$

where each A_i is a classical propositional formula without $\vec{\times}$ and $n > 1$. We will see later that arbitrary *QCL* formulas can be equivalently transformed into formulas of this kind. In this case the satisfaction degree of F in an interpretation I is simply the smallest k such that I satisfies A_k. If none of the ordered disjuncts is satisfied, then also F is not satisfied. More formally, using an index to express the degree of satisfaction, we define the satisfaction relation for formulas of this type as ($\models$ is classical propositional satisfaction):

$$I \models_k (A_1 \vec{\times} \cdots \vec{\times} A_n) \quad \text{iff}$$
$$I \models (A_1 \vee \cdots \vee A_n) \quad \text{and} \quad k = \min\{j \mid I \models A_j\}.$$

For arbitrary formulas determining the degree of disappointment is somewhat more involved. Consider an ordered disjunction $F = (F_1 \vec{\times} F_2)$ where F_1 and F_2 are complex formulas containing $\vec{\times}$. Assume that F_1 is not satisfied by I, and that F_2 is satisfied to degree k. How do we determine the satisfaction degree for F in this case? This degree depends on how many options or, in other words, possible satisfaction degrees F_1 admits. Assume there are j such options for F_1 all of which are, as we said, not satisfied. The satisfaction degree then will be $j + k$ since F is satisfied in the $(j + k)$th best possible way.

We call the number of possible satisfaction degrees of a formula its optionality:

Definition 2. The optionality of a formula is defined as follows:

$$
\begin{aligned}
&opt(A) = 1 \quad \text{if } A \text{ is an atom,}\\
&opt(\neg F) = 1,\\
&opt(F_1 \vee F_2) = max\big(opt(F_1), opt(F_2)\big),\\
&opt(F_1 \wedge F_2) = max\big(opt(F_1), opt(F_2)\big),\\
&opt(F_1 \vec{\times} F_2) = opt(F_1) + opt(F_2).
\end{aligned}
$$

Intuitively, if the optionality of F is n, then there may be a best way, a second best way, etc. and an nth best way of satisfying F. For classical formulas there is only one way to satisfy them, hence they all have optionality 1.

The optionality of a negated formula may seem puzzling at first, but there is not more than one way of making $\neg(A \vec{\times} B)$ true: you must make A and B false, there is no second best solution for this. In a sense negation transforms nested ordered disjunctions into standard disjunctions. Hence $\neg F$ behaves like a classical formula which explains why $opt(\neg F) = 1$.[4] For conjunction and disjunction we obtain the maximum optionality of the subformulas. For instance, if $opt(F_1) = j$ and $opt(F_2) = k$ with $j < k$, then an interpretation which does not satisfy F_1 but satisfies F_2 to degree k will satisfy $F_1 \vee F_2$ to degree k. Similarly, an interpretation which does satisfy F_1 to some degree and satisfies F_2 to degree k will satisfy $F_1 \wedge F_2$ to degree k.

If $F = A \vec{\times} B$, where A and B are classical, 2 degrees are needed. The best way of satisfying F is making A true, but if this does not work there is still the second best option, namely making B true. This generalizes to the sum of optionalities if A and B are arbitrary formulas. Note that for formulas of the form $A_1 \vec{\times} \cdots \vec{\times} A_n$ where all A_i are classical the optionality is n.

We now define the satisfaction relation. The relation is indexed according to the degree of satisfaction of a formula in a model.

Definition 3.

(1) $I \models_k A$ iff $k = 1$ and $A \in I$ (for propositional atoms A);
(2) $I \models_k P \wedge Q$ iff $I \models_m P$ and $I \models_n Q$ and $k = max(m, n)$;
(3) $I \models_k P \vee Q$ iff $I \models_m P$ or $I \models_n Q$ and $k = min\{r \mid I \models_r P \text{ or } I \models_r Q\}$;
(4) $I \models_k \neg P$ iff $k = 1$ and for no m: $I \models_m P$;
(5) $I \models_k P \vec{\times} Q$ iff $I \models_k P$ or $[I \models_1 \neg P,\ I \models_m Q,$ and $k = m + opt(P)]$.

Let us illustrate the satisfiability relation using a simple example. Let $F = A \vee (B \vec{\times} C)$ where A, B and C are atoms. Now all interpretations I containing A as well as those containing B satisfy F with degree 1, that is, we have $I \models_1 F$. Now assume I is an interpretation which contains C but makes A and B false. In this case F is satisfied with degree 2, that is $I \models_2 F$. Interpretations satisfying none of the three atoms do not satisfy F to any degree; they satisfy $\neg F$ with degree 1.

[4] As a consequence the formula $\neg\neg F$ is not equivalent to F, see Section 2.3.

The use of optionalities in (5) can be illustrated using the formula

$$F = (\mathit{walking} \vec{\times} \mathit{hotel\text{-}transport}) \vec{\times} \mathit{public\text{-}transport}$$

Assume $I \models \mathit{public\text{-}transport}$, $I \not\models \mathit{walking}$ and $I \not\models \mathit{hotel\text{-}transport}$. I satisfies *public-transport* with degree 1. Since $\mathit{opt}(\mathit{walking} \vec{\times} \mathit{hotel\text{-}transport}) = 2$ we obtain satisfaction degree 3 for F. This seems intuitive since the 3rd-best option is obtained.

The following lemmata can easily be proven by induction on the structure of formula F:

Lemma 1. *$I \models_k F$ and $I \models_j F$ implies $k = j$.*

We use $\mathit{deg}^I(F)$ to denote the satisfaction degree of F in interpretation I, that is $\mathit{deg}^I(F) = k$ whenever $I \models_k F$. If $I \models \neg F$ we let $\mathit{deg}^I(F) = 0$.

Lemma 2. *$I \models_k F$ implies $k \leqslant \mathit{opt}(F)$.*

The following proposition relates ordered and classical disjunction:

Proposition 1. *Let F be a formula. There is a k such that $I \models_k F$ iff $I \models F^*$ where F^* is obtained from F by replacing each occurrence of $\vec{\times}$ with standard disjunction.*

Definition 4. Let T be a set of formulas. An interpretation I is a model of T if it satisfies each formula in T to some degree.

The satisfaction degrees of formulas help us to determine preferred models. There are different ways of doing this. We will use here a lexicographic ordering of models based on the number of formulas satisfied to a particular degree. In Section 5 alternative preference relations will be discussed. The lexicographic ordering is defined as follows:

Definition 5. Let $M^k(T)$ denote the subset of formulas of T satisfied by a model M to degree k. A model M_1 is T-preferred over a model M_2 if there is a k such that $|M_1^k(T)| > |M_2^k(T)|$ and for all $j < k$: $|M_1^j(T)| = |M_2^j(T)|$. M is a preferred model of T iff M is a maximally T-preferred model.

Intuitively, a preferred model of T is a model of T which satisfies the maximal number of best options of choice logic formulas. Note that *QCL* is a conservative extension of classical propositional logic: for a set of formulas T without appearance of $\vec{\times}$ the preferred models of T coincide with the classical models of T since all classical formulas must be satisfied to degree 1.

We next define a consequence relation based on preferred models. We will restrict our attention here to classical conclusions, that is, the consequence relation we are interested in will be a relation between sets of formulas and classical formulas. The justification for this restriction lies in the intended use of *QCL*: we want to be able to derive properties describing a problem solution based on background knowledge, knowledge about the case at hand and choice formulas describing intended properties. Classical formulas are the formulas describing the intended problem solutions (e.g., the chosen hotel in the booking example).

Definition 6. Let T be a set of formulas, and let A be a classical formula. $T \mid\sim A$ iff A is satisfied in all preferred models of T.

This inference relation is obviously nonmonotonic. As an example consider $T = \{A \vec{\times} B\}$. We have three models $\{A\}, \{A, B\}, \{B\}$ with satisfaction degree 1, 1, 2, respectively. This means that $\{A\}$ and $\{A, B\}$ are maximally preferred and we have $A \vec{\times} B \mid\sim A$. If we add $\neg A$ then the single model, and thus the single preferred model, is $\{B\}$. We thus obtain $\{A \vec{\times} B, \neg A\} \mid\sim \neg A$ and A is no longer a conclusion.

2.3. Properties of QCL

We first define a notion of equivalence:

Definition 7. Let F_1, F_2 be formulas. F_1 is strongly equivalent to F_2, denoted $F_1 \doteq F_2$, iff $opt(F_1) = opt(F_2)$ and for all interpretations I and integers k we have $I \models_k F_1$ iff $I \models_k F_2$.

The optionality of the formulas must be the same in order to guarantee that subformulas can be replaced. For instance, although $A \vec{\times} A$ and A have the same satisfaction degree in all interpretations, $A \vec{\times} B$ and $A \vec{\times} A \vec{\times} B$ clearly have not. Indeed, we have the following substitution result:

Lemma 3. *Let $F(A)$ be a QCL formula containing a subformula A. Let $F(B)$ be obtained from $F(A)$ by substitution of the formula B for an occurrence of A. If $A \doteq B$ then $F(A) \doteq F(B)$.*

Note that for classical formulas without $\vec{\times}$ all classical logical transformations can be performed, but for those with $\vec{\times}$ some standard transformations are not valid: if A is classical, then $\neg\neg A$ is strongly equivalent to A, but $\neg\neg(A \vec{\times} B)$ is not strongly equivalent to $A \vec{\times} B$ due to the different optionalities of the formulas. Indeed, $\neg\neg(A \vec{\times} B)$ is strongly equivalent to $A \vee B$.

Another interesting feature of the logic is that conjunction and comma behave differently: the *QCL* theory $T_1 = \{F_1, F_2\}$ must be distinguished from $T_2 = \{F_1 \wedge F_2\}$. For instance, if $I_1 \models_1 F_1$ and $I_1 \models_2 F_2$, whereas $I_2 \models_2 F_1$ and $I_2 \models_2 F_2$, then I_1 is T_1-preferred over I_2, but not T_2-preferred. In our semantics, sets of formulas allow for more fine-grained distinctions than the conjunction of these formulas.

Proposition 2. *Ordered disjunction is associative, that is for arbitrary formulas F_1, F_2 and F_3 we have $((F_1 \vec{\times} F_2) \vec{\times} F_3) \doteq (F_1 \vec{\times} (F_2 \vec{\times} F_3))$.*

We now show that our inference relation satisfies properties usually considered intended in nonmonotonic reasoning.

Proposition 3. *The inference relation $\mid\sim$ satisfies cautious monotony, that is, $T \mid\sim A$ and $T \mid\sim B$ implies $T \cup \{A\} \mid\sim B$.*

Proposition 4. *The inference relation $\mid\sim$ satisfies cumulative transitivity, that is, $T \mid\sim A$ and $T \cup \{A\} \mid\sim B$ implies $T \mid\sim B$.*

In addition, we can show that $\mid\sim$ is rational, in the sense that it satisfies all rules of System P [21] as well as rational monotony [25]. This follows from the results of Section 3 and from the fact that lexicographic inference satisfies these rules as shown in [4].

3. Computation

In this section we investigate ways to compute consequences of *QCL* theories. The basic idea is to take a set of arbitrary formulas T and to proceed in 3 steps:

(1) translate T to $Norm(T)$, a strongly equivalent normal form where all formulas containing $\vec{\times}$ are basic choice formulas (to be defined below),
(2) construct a stratified knowledge base $Skb(Norm(T))$ such that a classical formula F is lexicographically entailed by $Skb(Norm(T))$ iff $Norm(T) \mid\sim F$,
(3) use the techniques developed in [5] to generate $Extract(Skb(Norm(T)))$, a classical propositional knowledge base whose consequences are exactly the formulas lexicographically entailed by $Skb(Norm(T))$.

From the equivalence results described in the next subsections the following proposition is immediate:

Proposition 5. *Let T be a consistent set of formulas.*

$$T \mid\sim A \quad \textit{iff} \quad Extract\big(Skb\big(Norm(T)\big)\big) \vdash A.$$

3.1. Translation to normal form

We want to translate *QCL* bases into a normal form consisting only of classical formulas and basic choice formulas which are defined as follows:

Definition 8. A formula F is a basic choice formula if it is of the form

$$A_1 \vec{\times} \cdots \vec{\times} A_n$$

where each A_i is a classical propositional formula and $n > 1$.

For the translation to normal form we need the following strong equivalences:

Proposition 6. *Let $A_i, B_j, C_k, \ldots$ be formulas without $\vec{\times}$. Then the following strong equivalences hold*:

(1) $(A_1 \vec{\times} \cdots \vec{\times} A_n) \vee (B_1 \vec{\times} \cdots \vec{\times} B_m) \doteq (C_1 \vec{\times} \cdots \vec{\times} C_k)$ *where* $k = max(m,n)$ *and*

$$C_i = \begin{cases} (A_i \vee B_i) & \text{if } i \leqslant min(m,n), \\ A_i & \text{if } m < i \leqslant n, \text{ and} \\ B_i & \text{if } n < i \leqslant m. \end{cases}$$

Example:

$$(A_1 \vec{\times} A_2) \vee (B_1 \vec{\times} B_2 \vec{\times} B_3) \doteq (A_1 \vee B_1) \vec{\times} (A_2 \vee B_2) \vec{\times} B_3.$$

(2) $(A_1 \vec{\times} \cdots \vec{\times} A_n) \wedge (B_1 \vec{\times} \cdots \vec{\times} B_m) \doteq (C_1 \vec{\times} \cdots \vec{\times} C_k)$ *where* $k = max(m,n)$ *and*

$$C_i = \begin{cases} [(A_1 \vee \cdots \vee A_i) \wedge B_i] \vee [A_i \wedge (B_1 \vee \cdots \vee B_i)] & \text{if } i \leqslant min(m,n), \\ [(A_1 \vee \cdots \vee A_n) \wedge B_i] & \text{if } n < i \leqslant m, \\ [A_i \wedge (B_1 \vee \cdots \vee B_m)] & \text{if } m < i \leqslant n. \end{cases}$$

Example:

$$\begin{aligned}&(A_1 \vec{\times} A_2) \wedge (B_1 \vec{\times} B_2 \vec{\times} B_3) \\ &\quad \doteq (A_1 \wedge B_1) \vec{\times} \big[(A_1 \vee A_2) \wedge B_2\big] \vee \big[A_2 \wedge (B_1 \vee B_2)\big] \vec{\times} \big[(A_1 \vee A_2) \wedge B_3\big].\end{aligned}$$

(3) $\neg(A_1 \vec{\times} \cdots \vec{\times} A_n) \doteq \neg(A_1 \vee \cdots \vee A_n)$.

Repeated application of these transformation rules moves $\vec{\times}$ outside (or eliminates it) until we obtain a classical formula or a basic choice formula:[5]

Proposition 7. *Every formula F can be translated to a strongly equivalent formula F' which is either classical or a basic choice formula.*

We next show how *QCL* bases in normal form can be translated to stratified knowledge bases. For the results of the following section we need an additional condition: we say a set T of formulas is standardized iff different basic choice formulas do not possess syntactically *identical* prefixes[6] (they may, of course, be logically equivalent). This condition considerably simplifies the discussion in the next subsection where we will construct certain sets of prefixes of formulas. In our cardinality based approach it is important to distinguish prefixes coming from different formulas. Without standardization we would have to deal with multi-sets rather than ordinary sets in the following definitions.

Of course, *any* theory T can be transformed into standardized form. This can simply be achieved by replacing identical formulas with some logically equivalent but syntactically different formulas. For instance, a standardized form of $T = \{A \vec{\times} B, A \vec{\times} C\}$ can be $\{A \wedge A \vec{\times} B, A \vec{\times} C\}$.

A set of *QCL* formulas T is in normal form if it consists of classical or basic choice formulas only, and if it is standardized.

---

[5] Due to (2), our translation as it stands is exponential. Jérôme Lang (personal communication) has pointed out that a quadratic translation exists. Each *QCL* formula $F_1 \vec{\times} \cdots \vec{\times} F_n$ is equivalent to $F_1 \vec{\times} (F_1 \vee F_2) \vec{\times} \cdots \vec{\times} (F_1 \vee \cdots \vee F_n)$. This transformation (with quadratic increase in size) needs to be done only once in the beginning. Then, under the assumption that $F_1 \vec{\times} F_2$ implies $F_1 \models F_2$, we can reformulate the translation of a conjunction of ordered disjunctions as follows: $(A_1 \vec{\times} \cdots \vec{\times} A_n) \wedge (B_1 \vec{\times} \cdots \vec{\times} B_m) \doteq (C_1 \vec{\times} \cdots \vec{\times} C_k)$ where $k = max(m,n)$ and $C_i = (A_i \wedge B_i)$ if $i \leqslant min(m,n)$, $C_i = (A_n \wedge B_i)$ if $n < i \leqslant m$, and $C_i = (A_i \wedge B_m)$ if $m < i \leqslant n$. The total increase in size is quadratic.

[6] $A_1 \vec{\times} \cdots \vec{\times} A_k$ is a prefix of $A_1 \vec{\times} \cdots \vec{\times} A_n$ whenever $k \leqslant n$.

3.2. Compilation to a stratified knowledge base

We recall the definition of a stratified knowledge base.

Definition 9. A stratified knowledge base is a sequence $(K, S_1, \ldots, S_n)$ of sets of classical propositional formulas.

Since we need K to be a non-default level we slightly modify the definition of lexicographically preferred subbases from [1,24].

Definition 10. Let $KB = (K, S_1, \ldots, S_n)$ be a stratified knowledge base. A maximal consistent subset S of $(K \cup S_1 \cup \cdots \cup S_n)$ is a lexicographically preferred subbase of KB iff

(1) $K \subseteq S$, and
(2) if S' is a maximal consistent subset of $(K \cup S_1 \cup \cdots \cup S_n)$ containing K and for some $k \in \{1, \ldots, n\}$: $|S' \cap S_k| > |S \cap S_k|$ then there is $j < k$ such that $|S \cap S_j| > |S' \cap S_j|$.

The only difference between this definition and the original one [1,24] is that there is no lexicographically preferred subbase if K is inconsistent.

Definition 11. Let $KB = (K, S_1, \ldots, S_n)$ be a stratified knowledge base. A formula F is lexicographically entailed by KB, denoted $KB \vdash_{lex} F$, iff F is entailed by all lexicographically preferred subbases of KB.

We now define the translation:

Definition 12. Let T be a *QCL* base in normal form. The stratified knowledge base associated with T, denoted $Skb(T)$, is

$$Skb(T) = \left(T^*, T_1, \ldots, T_n\right)$$

where $n = max\{k \mid F \in T, opt(F) = k\} - 1$, T^* is obtained from T by replacing each occurrence of $\vec{\times}$ by $\vee$, and

$$T_i = \{A_1 \vee \cdots \vee A_i \mid 1 \leqslant i < k, A_1 \vec{\times} \cdots \vec{\times} A_k \in T\}.$$

The translation is obviously polynomial in time and size.

Proposition 8. *$T \mid\!\sim F$ iff $Skb(T) \vdash_{lex} F$.*

Note that in the definition of T_i the case $i = k$ is not needed since the corresponding disjunctions are already in T^*. It turns out that there is a second, equivalent translation. In Definition 12 we can equivalently define

$$T_i = \{\neg A_1 \wedge \cdots \wedge \neg A_{i-1} \wedge A_i \mid 1 \leqslant i < k, A_1 \vec{\times} \cdots \vec{\times} A_k \in T\}.$$

To illustrate the translation consider the hotel example given in the introduction (after $\neg hotel_1$ was learned). The stratified knowledge base associated with T is $Skb(T) = (T^*, T_1, T_2)$ where T^* consists of:

$\neg hotel_1$,
$walking \vee hotel\text{-}transport \vee public\text{-}transport$,
$hotel_1 \vee hotel_2 \vee hotel_3 \vee hotel_4$,
$hotel_1 \rightarrow walking$,
$hotel_2 \rightarrow \neg walking \wedge hotel\text{-}transport$,
$hotel_3 \rightarrow \neg walking \wedge \neg hotel\text{-}transport \wedge public\text{-}transport$,
$hotel_4 \rightarrow \neg walking \wedge \neg hotel\text{-}transport \wedge \neg public\text{-}transport$,

and

$T_1 = \{walking\}$,
$T_2 = \{walking \vee hotel\text{-}transport\}$.

There is exactly one lexicographically preferred subbase, namely $T^* \cup T_2$. We thus have

$$Skb(T) \vdash_{lex} hotel_2$$

as intended.

Note that a translation also exists in the opposite direction: we can translate a stratified knowledge base $KB = (K, S_1, \ldots, S_n)$ to a *QCL*-theory T. T contains K, and for each formula $F \in S_i$ the formula $\bot \vec{\times} \cdots \vec{\times} \bot \vec{\times} F \vec{\times} \top$ with i occurrences of $\bot$.

3.3. Compilation to a classical KB

This step is described in [5]. To make this paper somewhat more self-contained we briefly describe the main idea underlying the compilation. Lexicographic entailment from the original base is replaced by classical entailment from a compiled base, which contains either formulas from the original base or formulas subsumed by the original ones, obtained from the disjunction of some of the original formulas. This idea has several implementations, namely Disjunctive Maxi-Adjustment (DMA), Iterative DMA and Whole-DMA, which produce logically equivalent results but with different spatial and computational needs.

- DMA replaces formulas involved in a conflict by disjunctions that restore consistency involving a minimum of these formulas (if any). Here the conflicting formulas are only computed once.
- Iterative DMA only adds pairwise disjunctions to the KB and iterates the detection of conflicting formulas in order to stay as close as possible to the original KB. Unfortunately, the approach is computationally costly since determining formulas involved in a conflict is on the second level of the polynomial hierarchy in complexity theory.

- In contrast, Whole-DMA does not detect formulas involved in conflicts, but works with disjunctions built from all the formulas. The advantage is that the complexity of the approach is on the first level of the polynomial hierarchy (SAT), and that in practice SAT solvers are now able to solve instances of the problem with several thousands of variables. The disadvantage is that the size of the compiled KB is likely to explode exponentially.

Note that for the hotel example discussed above the three approaches yield the classical knowledge base $T^* \cup \{\mathit{walking} \vee \mathit{hotel\text{-}transport}\}$.

For the details we refer to [5].

4. Applications

Qualitative choice logic has a number of possible applications.

- In design and configuration, intended properties can be described and ranked according to their desirability.
- In agent systems, intended actions to be performed by agents can be specified together with backup actions covering situations where the standard actions are inappropriate.
- In database and web applications, prioritized queries can be expressed which describe suboptimal results. For instance, one may start a query like: find a used convertible less than 2 years old in red, if red is unavailable then in black. This may be very useful for e-commerce applications.

We will focus on configuration and prioritized queries in this paper.

The configuration task can informally be described as follows: given a set of components *Comp* determine a set $C \subseteq \mathit{Comp}$ such that C is a complete solution (no necessary component is lacking), contains no unnecessary elements, and satisfies certain requirements depending on the case at hand as much as possible. More precisely the following pieces of information are relevant:

- A description of the available components which can be chosen for a particular configuration or subconfiguration.
- A description of additional components which are necessary if some component/subconfiguration is chosen.
- A description of the properties of the components.
- A description of the particular configuration task at hand.
- A description of the desired properties respectively intended components.

Let us illustrate this using a trip planning example. The available main options are going by *plane*, by *train* or by *car*. If *plane* is chosen it is necessary to organize transportation to and from the airport. If *train* is chosen one has to get to the station and from the station to the final destination. For both cases taxis and buses are available. Here is a formalization of this information. It is convenient to use exclusive or ($\otimes$) in configuration examples.

$trip \leftrightarrow plane \otimes train \otimes car$,

$\neg plane \vee \neg train \vee \neg car$,

$train \leftrightarrow toStation$,

$train \leftrightarrow fromStation$,

$plane \leftrightarrow toAirport$,

$plane \leftrightarrow fromAirport$,

$toStation \leftrightarrow taxi\text{-}toStation \otimes bus\text{-}toStation$,

$fromStation \leftrightarrow taxi\text{-}fromStation \otimes bus\text{-}fromStation$,

$toAirport \leftrightarrow taxi\text{-}toAirport \otimes bus\text{-}toAirport$,

$fromAirport \leftrightarrow taxi\text{-}fromAirport \otimes bus\text{-}fromAirport$.

Note that we use $\leftrightarrow$ rather than $\rightarrow$ to describe the available alternatives. This is necessary to make sure that a component is contained in a configuration only if it is necessary.[7]

We next describe our preferences. For short trips we prefer *car* over *train*, for medium trips *train* over *car*. We never use the plane for such trips. Also, if we have heavy *luggage* we never take the train. For long distance trips our first preference is *plane*, followed by *train* and *car*. We also need to represent the background knowledge that a trip belongs to exactly one of the three categories:

$short \rightarrow car \vec{\times} train$,

$medium \rightarrow train \vec{\times} car$,

$long \rightarrow flight \vec{\times} train \vec{\times} car$,

$luggage \rightarrow \neg train$,

$short \otimes medium \otimes long$,

$\neg short \vee \neg medium \vee \neg long$.

The preferences for traveling to and from the airport, respectively station, are as follows:

$toStation \rightarrow taxi\text{-}toStation \vec{\times} bus\text{-}toStation$,

$fromStation \rightarrow taxi\text{-}fromStation \vec{\times} bus\text{-}fromStation$,

$toAirport \rightarrow bus\text{-}toAirport \vec{\times} taxi\text{-}toAirport$,

$fromAirport \rightarrow bus\text{-}fromAirport \vec{\times} taxi\text{-}fromAirport$.

Now given a description of the trip to be planned and the relevant requirements we obtain a suitable configuration. For instance, if in addition to the formulas above we have *trip*, *short* then the preferred model contains *car* and no other component. If we have *trip*, *short*, $\neg car$ we get *taxi-toStation*, *train*, *taxi-fromStation*. In each case we obtain a configuration which best satisfies our preferences.

[7] The second formula is necessary because $plane \otimes train \otimes car$ is true also if all three alternatives are true.

The next example involves prioritized queries. Assume a database *DB* of certain items, say cars, with a description of their properties is given:

$$car_1 \to BMW \wedge red \wedge \neg convertible,$$
$$car_2 \to BMW \wedge green \wedge convertible,$$
$$car_3 \to VW \wedge blue \wedge \neg convertible,$$
$$car_4 \to VW \wedge blue \wedge convertible.$$

We assume *DB* also contains background knowledge of the kind

$$BMW \to \neg VW, \qquad red \to \neg blue, \qquad red \to \neg green,$$

etc. together with information that exactly one car is to be chosen:

$$car_1 \vee car_2 \vee \cdots$$

and the formulas $\{car_i \to \neg car_j \mid i < j\}$. Now a prioritized query is just a set of *QCL* formulas Q. An answer *Ans* is a disjunction of items such that $DB \cup Q \mid\sim Ans$. In the example we might have

$$Q = \{red \vec{\times} blue, convertible \vec{\times} \neg convertible\}.$$

This query would lead to the answer $car_1 \vee car_4$. car_1 is contained in the answer since it has the most preferred colour, car_4 since it is a convertible. If we add to Q the formula $BMW \vec{\times} VW$ the answer becomes car_1.

Note that since the preference relation on models depends on the number of formulas satisfied to a particular degree we can give more weight to a particular criterion by adding a syntactic variant of a formula in Q. Again, syntactic variants are needed to avoid the use of multi-sets of formulas. For instance, if we add $(red \wedge \top) \vec{\times} blue$ to our original query then the colour criterion becomes more important since colour counts twice and we obtain the answer *BMW*. This does not exclude blue cars from being answers in case no red car is available.

5. Alternative definitions of entailment

In Section 2 we defined entailment for *QCL* in terms of a lexicographic ordering on models, based on the number of formulas satisfied to a certain degree. This leads to an approach where solutions are preferred when they contain the highest number of most preferred options. For instance, if there are three choices with three options each, say

$$A_1 \vec{\times} A_2 \vec{\times} A_3, \qquad B_1 \vec{\times} B_2 \vec{\times} B_3, \qquad C_1 \vec{\times} C_2 \vec{\times} C_3$$

then a model M_1 satisfying A_1, B_1 and C_3 is preferred over a model M_2 satisfying A_2, B_2 and C_1 because the number of formulas satisfied in M_1 with degree 1 is 2, the number in M_2 is 1. This may not be wanted for all applications. In our example we might consider M_2 a reasonable alternative: although it gives us only one best choice, we still get two second best options.

In this section we will discuss alternative ways to define preferences on models based on the satisfaction degrees of the premises.

5.1. Inclusion based preference

The following strengthening of the preference relation is based on subsets rather than the number of formulas satisfied with a particular degree.

Definition 13. Let $M^k(T)$ denote the set of formulas of T satisfied by a model M of T to degree k. A model M_1 of T is T-inclusion-preferred over a model M_2 if there is a k such that $M_1^k(T)$ is a strict superset of $M_2^k(T)$ and for all $j < k$: $M_1^j(T) = M_2^j(T)$.

Note that inclusion preference implies the cardinality based preference introduced earlier but not vice versa. In the example above, M_1 is not inclusion-preferred over M_2. We therefore get in general more maximally inclusion-preferred models and thus fewer conclusions, that is, the inference relation $\mid\sim_{inc}$ (defined as $T \mid\sim_{inc} F$ iff F is true in all inclusion preferred models of T) is more cautious than $\mid\sim$.

It turns out that a result corresponding to Proposition 8 can be established for inclusion based preference using the same translation. The definition of an inclusion preferred subbase of a stratified knowledge base is obtained from Definition 10 by replacing clause 2 with

> if S' is a maximal consistent subset of $(K \cup S_1 \cup \cdots \cup S_n)$ containing K and for some $k \in \{1, \ldots, n\}$: $S \cap S_k$ is a proper subset of $S' \cap S_k$ then there is $j < k$ such that $S' \cap S_j$ is a proper subset of $S \cap S_j$.

We have the following proposition:

Proposition 9. *$T \mid\sim_{inc} F$ iff $Skb(T) \vdash_{inc} F$.*

A recent complexity result from Coste-Marquis and Marquis [14] is relevant here: they prove that there is no way to compile any stratified knowledge base under inclusion preference in polynomial space whereas such a translation is possible under lexicographic preference by adding new propositional variables (the compiled base is "query equivalent" to the original one). This is an additional argument in favour of lexicographic preference.

5.2. Preference based on ranking functions

Another possibility for defining preferred models is to use ranking functions. Ranking functions assign an integer rank to a model M of a set of formulas

$$T = \{f_1, \ldots, f_n\}$$

based on the vector of satisfaction degrees

$$\big(deg^M(f_1), \ldots, deg^M(f_n)\big)$$

of the formulas in T.[8] More precisely, let $\ominus$ be a function from a vector of integers to an integer. $\ominus$ will be called a ranking function if it satisfies the following requirements:

Unanimity: If $\forall i = 1, \ldots, n$, $j_i \geqslant k_i$ then: $\ominus(j_1, \ldots, j_n) \geqslant \ominus(k_1, \ldots, k_n)$.
Model preservation: $\ominus(j_1, \ldots, j_n) = 0$ iff $j_i = 0$ for some $i = 1, \ldots, n$.

Intuitively, the second requirement means that a given interpretation should not be considered as a model of T if and only if it falsifies some formula of T. Ranking functions induce preferences on models in a straightforward way:

Definition 14. Let $T = \{f_1, \ldots, f_n\}$ be a set of *QCL* formulas, $\ominus$ a ranking function. A model M_1 of T is a $\ominus$-preferred model of T iff there is no model M_2 of T such that $\ominus(deg^{M_2}(f_1), \ldots, deg^{M_2}(f_n)) < \ominus(deg^{M_1}(f_1), \ldots, deg^{M_1}(f_n))$.

The inference relation induced by this preference relation will be denoted $\mid\sim_{\ominus}$.

As an example consider the ranking function

$$\ominus(j_1, \ldots, j_n) = \begin{cases} 0 & \text{if } j_k = 0 \text{ for some } k \in \{1, \ldots, n\}, \\ \sum_{1 \leqslant k \leqslant n} j_k & \text{otherwise.} \end{cases}$$

This ranking function just adds up the degrees of (dis)satisfaction of all formulas in T for each model and prefers those models whose sum is minimal, that is, whose overall dissatisfaction is minimal. In the example from the beginning of this section, both M_1 and M_2 have overall degree 5 ($1 + 1 + 3$ and $2 + 2 + 1$, respectively), thus none of the models is preferred to the other.

Even more fine grained distinctions could be introduced by adding to each option in an ordered disjunction some integer (increasing in value from left to right) which could be used as a kind of penalty to compute the overall dissatisfaction degree of models.

The relationship between preference based on ranking functions and possibilistic logic will be further investigated in Section 7.

6. Relation to circumscription

Since its invention in the seventies, circumscription [26–28] was certainly one of the most influential nonmonotonic formalisms. In the first order case, circumscription allows the extensions of certain predicates to be minimized. Propositional circumscription makes certain atoms false whenever possible. Semantically, this is achieved by defining a preference relation on models and reasoning from most preferred models. The preference relation depends on the predicates, respectively atoms, chosen for minimization.

Since *QCL* is a propositional logic, we consider propositional circumscription for the comparison in this section. Let us first discuss a short example. According to the

[8] For the reader's convenience we recall the definition of deg^M given after Lemma 1: $deg^M(f) = k$ iff $M \models_k f$ and $deg^M(f) = 0$ iff $M \models_1 \neg f$.

methodology proposed by McCarthy, defaults can be represented using ab atoms which are then circumscribed. For instance, circumscribing ab_1 in

(1) $penguin \wedge \neg ab_1 \rightarrow \neg flies$,

(2) $penguin$,

yields the conclusion $\neg flies$. In *QCL* there is a simple way to model this: we just have to add

$$\neg ab_1 \vec{\times} ab_1$$

and obtain exactly the same conclusions.

For our formal result we will consider one of the most general forms of circumscription, prioritized circumscription with fixed atoms. The priorities are used to handle potential conflicts between minimized atoms, the fixed atoms are not allowed to vary during the minimization.

Assume that, in addition to formulas (1) and (2), we have

(3) $bird \wedge \neg ab_2 \rightarrow flies$,

(4) $bird$.

Minimizing the two ab-atoms now yields two minimal models. Intuitively, we would expect (1) to be preferred over (3) for reasons of specificity. In circumscription this can be achieved by minimizing ab_1 with higher priority than ab_2. This can be modeled in *QCL* by adding to the four premises the formulas

$$\neg ab_1 \vec{\times} ab_1,$$
$$\bot \vec{\times} \neg ab_2 \vec{\times} ab_2.$$

Adding an unsatisfiable option to the first choice formula has the desired effect: making ab_1 false is more important than making ab_2 false.

It is also not difficult to handle fixed atoms. Assume b is fixed. We have to make sure that models which differ in the truth value of b become incomparable. If inclusion based preference is used this can be achieved by adding:

$$\neg b \vec{\times} b,$$
$$b \vec{\times} \neg b.$$

Here are the formal definitions needed for our result:

Definition 15. Let T be a propositional theory, $V_1, \ldots, V_n$ sets of atoms to be circumscribed, W a set of fixed atoms such that $(V_1 \cup \cdots \cup V_n) \cap W = \emptyset$. A formula F is a consequence of the prioritized circumscription of $V_1, \ldots, V_n$ in T with fixed W, denoted $Circ(T; V_1, \ldots, V_n; W) \models F$, iff F is true in all $<_{V_1,\ldots,V_n;W}$-preferred models of T, where $M_1 <_{V_1,\ldots,V_n;W} M_2$ iff[9]

[9] As usual, we identify models with the set of their true atoms.

(1) there is $i \in \{1,\ldots,n\}$ such that $M_1 \cap V_i \subset M_2 \cap V_i$, and for all $j < i$: $M_1 \cap V_j = M_2 \cap V_j$, and
(2) $M_1 \cap W = M_2 \cap W$.

Since the preference criterion used for circumscription is based on subsets, the inclusion based variant of *QCL* is the natural candidate to capture circumscription. Indeed, we have the following proposition:

Proposition 10. *Let T be a propositional theory, $V_1,\ldots,V_n$ sets of atoms to be circumscribed, W a set of fixed atoms such that $(V_1 \cup \cdots \cup V_n) \cap W = \emptyset$. $Circ(T; V_1,\ldots,V_n; W) \models F$ iff $T' \mid\sim_{inc} F$ where*

$$\begin{aligned} T' = T \cup \{&\bot \vec{\times} \cdots \vec{\times} \bot \vec{\times} \neg v \vec{\times} v \mid v \in V_i, \bot \text{ appears } i-1 \text{ times}\} \\ &\cup \{v \vec{\times} \neg v \mid v \in W\} \\ &\cup \{\neg v \vec{\times} v \mid v \in W\}. \end{aligned}$$

We have seen that *QCL* can quite easily capture propositional circumscription. How about the converse? Of course, since circumscription by definition is a minimization technique whereas *QCL* allows us to specify arbitrary preferences we cannot expect modular translations from *QCL* to circumscription which give us exactly the same preferred models.[10] However, if we admit additional symbols in the language we can use circumscription to generate models corresponding to the inclusion preferred *QCL* models *up to the additional atoms*.

We assume that all *QCL* formulas are in normal form, i.e., of the form $F_j = A_1 \vec{\times} \cdots \vec{\times} A_n$. Using additional abnormality atoms this formula can be represented as the set of formulas:

$$\begin{aligned} &(1) \quad \neg ab_{j,1} \rightarrow A_1, \\ &(2) \quad ab_{j,1} \wedge \neg ab_{j,2} \rightarrow A_2, \\ &\quad \vdots \\ &(n-1) \quad ab_{j,1} \wedge \cdots \wedge ab_{j,n-2} \wedge \neg ab_{j,n-1} \rightarrow A_{n-1}, \\ &(n) \quad ab_{j,1} \wedge \cdots \wedge ab_{j,n-1} \rightarrow A_n. \end{aligned}$$

To translate a formula of optionality n in normal form we thus need $n-1$ new ab-atoms not appearing anywhere else in the premises and in the translation of other formulas, and we generate n implications as illustrated above. Intuitively, $ab_{j,i}$ says: option i of formula j is impossible. We have the following result:

Proposition 11. *Let $T = \{F_1, F_2, \ldots\}$ be a set of QCL formulas in normal form and $n = max\{j \mid F_i \in T, opt(F_i) = j\} - 1$. Let T' be the translation of T, and for $i \in \{1,\ldots,n\}$*

[10] A translation *Trans* is modular iff for arbitrary sets S, S' we have $Trans(S \cup S') = Trans(S) \cup Trans(S')$.

let $AB_i = \{ab_{j,i} \mid F_j \in T\}$ *the set of newly introduced abnormality atoms with second index* i. *Moreover, let* F *be a formula not containing any atom in* $AB_1 \cup \cdots \cup AB_n$. *Then*

$$T \mid\sim_{inc} F \quad \textit{iff} \quad Circ(T'; AB_1, \ldots, AB_n; \emptyset) \models F.$$

This result shows that in principle prioritized circumscription is able to express *QCL* under inclusion based preference. However, for each formula with optionality n we need $n-1$ new abnormality atoms, and the representation is quadratic in the size of the original *QCL* theory. We also consider it as an advantage that in *QCL* the knowledge is completely represented by means of formulas whereas circumscription needs the additional specification of a circumscription policy. Moreover, *QCL* offers alternative preference criteria which are not captured by standard versions of circumscription.[11]

7. Possibilistic logic and *QCL*

In Section 5 several alternative definitions of entailment have been proposed. Among them, we defined preferred models based on ranking functions $\ominus$. This section investigates relationships between possibilistic logic and a class of *QCL* based on ranking functions $\ominus$. In particular, we show that (i) any basic choice logic formula can be viewed as a possibilistic knowledge base, and (ii) for each ranking function $\ominus$, the entailment $\mid\sim_{\ominus}$ can be recovered in a possibility theory framework. A corollary of this result is that, for any ranking function, any *QCL* theory can be equivalently transformed into a basic choice logic formula. The following first provides a brief reminder on possibilistic logic.

7.1. A brief reminder on possibilistic logic

7.1.1. Guaranteed possibilistic knowledge bases

Possibilistic logic provides a tool for performing uncertainty reasoning, where uncertain information is semantically represented by means of possibility distributions. Possibility distributions are means to rank order different interpretations of a language. More precisely, a possibility distribution, denoted by π, is a function from a set of mutually exclusive situations (solutions or interpretations) to the interval $[0, 1]$. By convention $\pi(I) = 1$ means that I is among the most normal (or preferred) situations, $\pi(I) = 0$ means that I is impossible or excluded as a possible solution. More generally, $\pi(I) \geqslant \pi(I')$ means that I is at least as preferred as I'. There are several compact (or syntactic) encodings of possibility distributions [2]: necessity based knowledge bases, min-based possibilistic graphs, product-based possibilistic graphs, etc. Recently, another type of compact representation, called guaranteed possibilistic knowledge bases or simply Δ-knowledge bases, has been investigated [3,16]. It is based on the notion of guaranteed possibility measures, which are defined on formulas from a possibility distribution π in the following way:

$$\Delta(\phi) = min\big\{\pi(I) \mid I \models \phi\big\}.$$

[11] For a cardinality based treatment of circumscription see [29].

A Δ-knowledge base is composed of a set of weighted formulas of the form $[\phi_i, \alpha_i]$, where ϕ_i denotes a propositional formula, and α_i is a real number between 0 and 1. The pair $[\phi_i, \alpha_i]$ means that any model of ϕ_i is satisfactory to a degree at least equal to α_i, namely:

$$\Delta(\phi_i) \geqslant \alpha_i$$

or

$$(1) \quad \forall I \models \phi_i, \quad \pi(I) \geqslant \alpha_i.$$

Definition 16. Each Δ-knowledge base Δ induces a unique possibility distribution π defined by:

$$\forall I, \quad \pi(I) = \begin{cases} 0 & \text{iff } I \text{ falsifies all formulas of } \Delta, \\ max\{\alpha_i \mid [\phi_i, \alpha_i] \in \Delta, I \models \phi_i\} & \text{otherwise.} \end{cases}$$

In [16] it has been shown that this possibility distribution corresponds to the most specific possibility distribution[12] satisfying (1) for each weighted formula in Δ. As we will show later, any basic choice logic formula can be equivalently represented by a Δ-knowledge base. However, in order to show the encoding of a set basic choice logic formulas, we need to use possibilistic fusion operators which are recalled in next subsection.

7.1.2. Possibilistic fusion

In [3] several possibilistic fusion operators have been proposed to merge a set Δ-knowledge bases. In this section we restrict the class of merging operators to those which are useful for establishing relationships between *QCL* and possibilistic logic. More precisely, let $\oplus$ be a function from a vector of real numbers in $[0, 1]$ to a real number between $[0, 1]$. $\oplus$ will be called a $[0, 1]$-based merging operator. The requirements for $\oplus$ are similar to those defined for $\ominus$ in Section 5.2:

Unanimity: If $\forall i = 1, \ldots, n, \alpha_i \geqslant \alpha'_i$ then: $\oplus(\alpha_1, \ldots, \alpha_n) \geqslant \oplus(\alpha'_1, \ldots, \alpha'_n)$.
Model preservation: $\oplus(\alpha_1, \ldots, \alpha_n) = 0$ iff $\alpha_i = 0$ for some $i = 1, \ldots, n$.

The second requirement is stronger than the one used in [3] which simply requires that $\oplus(0, \ldots, 0) = 0$. It is strengthened here in order to have an easier connection with *QCL*. Let $\Delta_1, \ldots, \Delta_n$ be the Δ-knowledge bases to merge. We denote by $[\phi_{ij}, \alpha_{ij}]$ the jth weighted formula in Δ_i.

Definition 17. The result of merging $\Delta_1, \ldots, \Delta_n$, denoted by $\Delta_\oplus$, is defined as

$$\Delta_\oplus = \big\{[\phi_{1i} \wedge \cdots \wedge \phi_{nk}, \oplus(\alpha_{1i}, \ldots, \alpha_{nk})] \mid [\phi_{1i}, \alpha_{1i}] \in \Delta_1, \ldots, [\phi_{nk}, \alpha_{nk}] \in \Delta_n\big\}.$$

It can be checked that the possibility distribution associated to $\Delta_\oplus$ using the above definition can be characterized as follows:

[12] π is said to be more specific than π', if $\forall I, \pi(I) \leqslant \pi'(I)$.

Lemma 4. *Let $\Delta_1, \ldots, \Delta_n$ be a set of Δ-knowledge bases. Let $\pi_1, \ldots, \pi_n$ be their associated possibility distributions, respectively. Let $\oplus$ be a $[0,1]$-based merging operator, and $\Delta_\oplus$ be the Δ-knowledge base given by Definition* 17. *Then $\forall I, \pi_{\Delta_\oplus}(I) = \oplus(\pi_1(I), \ldots, \pi_n(I))$.*

7.2. Encoding ranking functions-based QCL in possibilistic logic

We restrict ourselves to sets of basic choice formulas. This is not a limitation since a classical propositional formula p can be represented as a basic choice formula $p \vec{\times} \bot$ without changing the ranking of interpretations. The following lemma is immediate, noticing that models of $\{p_i \vec{\times} \bot \mid p_i$ is a classical formula of $T\}$ are exactly the same as the ones of $\{p_i \mid p_i$ is a classical formula of $T\}$.

Lemma 5. *Let T be a set of formulas. Let T' be obtained from T by replacing each classical formula p in T by a new basic choice formula $p \vec{\times} \bot$. Then T and T' induce the same ranking on the set of interpretations.*

The following lemma establishes a first connection between *QCL* and possibilistic logic when we only have one basic choice formula.

Lemma 6. *Let $F = A_1 \vec{\times} \cdots \vec{\times} A_n$ be a basic choice formula. Let $\alpha_i = \varepsilon^i$ for $i = 1, \ldots, n$, where $0 < \varepsilon < 1$.*[13] *Let $\Delta = \{[A_1, \alpha_1], \ldots, [A_n, \alpha_n]\}$ be the Δ-knowledge base associated to F. Then $\forall I$, $I \models_k F$ if and only if $\pi(I) = \alpha_k$ and $I \models_1 \neg F$ iff $\pi(I) = 0$.*

Hence, each basic choice logic formula can be represented by a Δ-knowledge base. As a corollary of Lemmas 4 and 6, it is possible to provide a possibilistic encoding of a general $QCL^{\ominus}$ theory:

Lemma 7. *Let $T = \{F_1, \ldots, F_n\}$ be a QCL theory, and let $\ominus$ be a ranking function. Let us again denote by $\alpha_i = \varepsilon^i$ for $i \in \aleph$. Let Δ_i be the Δ-knowledge base associated with F_i using Lemma* 6. *Then the Δ-knowledge base associated with T is the one obtained from Definition* 17 *by merging Δ_i's with $\oplus(\alpha_i, \ldots, \alpha_k) = \varepsilon^{\ominus(i,\ldots,k)}$.*

Lemma 7 is interesting since it means that, for any ranking function $\ominus$, any *QCL* theory can be transformed into one basic choice formula. Indeed, let $T = \{F_1, \ldots, F_n\}$ be a set of basic choice formulas. Again, we denote by A_{ij} the propositional formula which is in position j (namely after $j-1$ occurrences of $\vec{\times}$) in the basic choice formula F_i. We denote by

$$\mathcal{C} = A_{1i} \wedge \cdots \wedge A_{nk}$$

any conjunction of exactly one A_{ij} from each F_i. $\mathcal{C}$ is called a complex cube. We denote by *degree*($\mathcal{C}$) the rank associated to $\mathcal{C}$ defined as equal to $\ominus(i, \ldots, k)$. Then, it can be checked

[13] In fact, any function such that $\alpha_i > \alpha_j$ if and only if $i < j$ is appropriate.

that for each ranking function $\ominus$, it is possible to replace T by one basic choice formula defined as $B_1 \vec{\times} \cdots \vec{\times} B_m$ where:

$$m = \ominus\big(opt(F_1), \ldots, opt(F_n)\big),$$

and

$$B_i = \begin{cases} \bot & \text{if there is no } \mathcal{C} \text{ such that } degree(\mathcal{C}) = i, \\ \bigvee_{degree(\mathcal{C})=i} \mathcal{C} & \text{otherwise.} \end{cases}$$

Example 1. Let us assume that T is composed of two basic choice formulas $F_1 = A_{11} \vec{\times} A_{12}$ and $F_2 = A_{21} \vec{\times} A_{22}$. Then we have 4 complex cubes:

$$A_{11} \wedge A_{21}, \qquad A_{11} \wedge A_{22}, \qquad A_{12} \wedge A_{21}, \qquad A_{12} \wedge A_{22}.$$

Let us assume that the ranking function $\ominus$ yields the sum of the satisfaction degrees of the single formulas (as described in Section 5.2). Then, we have:

$$degree(A_{11} \wedge A_{21}) = 2,$$
$$degree(A_{11} \wedge A_{22}) = 3,$$
$$degree(A_{12} \wedge A_{21}) = 3,$$
$$degree(A_{12} \wedge A_{22}) = 4.$$

Therefore, it can be checked that if we rank-order interpretations with respect to sum of degree then this ranking can be recovered from the basic choice formula $C_1 \vec{\times} C_2 \vec{\times} C_3 \vec{\times} C_4$ with:

$$C_1 = \bot,$$
$$C_2 = A_{11} \wedge A_{21},$$
$$C_3 = (A_{12} \wedge A_{21}) \vee (A_{12} \wedge A_{21}), \quad \text{and}$$
$$C_4 = A_{12} \wedge A_{22}.$$

8. Combining *QCL* and default logic

QCL allows us to specify preferences and to reason from most preferred models. So far, we haven't said much about the very nature of the preferences. A model M_1 may be preferred to another model M_2 because it better satisfies some desires, intentions or norms. But it may also be preferred because it describes more normal states of the world, in other words, because it is more in accordance with our expectations about what is true in the world.

In situations where different types of preferences play a role, e.g., desires as well as default information, it may be useful to have different formal tools available to represent them. In this section we discuss how *QCL* and Reiter's default logic [32], one of the standard logics for representing defeasible information, can be combined. The motivation for this section is twofold: (1) we want to show that ordered disjunction cannot only be introduced in classical propositional logic, but also in a non-classical logic, and (2) we

want to propose a formalism where the mechanisms of default logic are used to determine what is normally the case, in other words, what is expected to be true, and the mechanisms of *QCL* to determine what is desired to be true. We assume some familiarity with default logic and refer the reader to Reiter's original paper for more details.

In Reiter's approach a default theory (D, W) is a pair consisting of a set of classical formulas W representing what is known to be true and a set of default rules D. Default theories induce extensions which can be viewed as sets of acceptable beliefs a reasoner may adopt based on the default theory. The extensions are logically closed supersets of W which are closed under the default rules and contain only formulas possessing a non-circular derivation. A (propositional) default rule is of the form $A : B_1, \ldots, B_n/C$ where A, the prerequisite, B_i, the consistency conditions, and C, the consequent (also called head of the default rule), are formulas. The rule is applicable with respect to a set of formulas S iff $A \in S$ and for no $i \in \{1, \ldots, n\}$ $\neg B_i \in S$.

We will extend default logic in two ways: we will admit ordered disjunction in W and in the head of default rules. This will allow us to represent desires in W and to derive preferences by default. The general principle is the same as in *QCL*, but instead of preferred models of the premises we have to consider preferred models of extensions. To define the preference relation between models a satisfaction degree of default rules needs to be defined.

Definition 18. A (propositional) choice default theory is a pair (D, W), where W is a set of *QCL* formulas and D is a set of rules of the form $A : B_1, \ldots, B_n/C$ where A and the B_i are classical formulas, C is a *QCL* formula.

To define extensions we will simply consider the default theories obtained by replacing ordered disjunction with ordinary disjunction:

Definition 19. Let $\Delta = (D, W)$ be a choice default theory. E is an extension of Δ iff E is an extension of (D^*, W^*) where D^* and W^* are the propositional counterparts of D, respectively W, obtained by replacing ordered disjunction with ordinary disjunction.

Ordered disjunction is now used to determine the most preferred models of extensions. We will use $Ext(\Delta)$ to denote the set of extensions of Δ. To compare the models of extensions we have to define a satisfaction degree not only for formulas, but also for the rules in D. The satisfaction degree of a rule will not only depend on the satisfaction degree of its consequent, but also on the applicability of the rule within the extension. Since there is no need to punish a rule for being inapplicable we will say that a rule is satisfied to the best possible degree 1 whenever its prerequisite is underivable or one of its consistency conditions is violated.

Definition 20. Let E be an extension of a choice default theory (D, W), M a model of E and $r = A : B_1, \ldots, B_n/C$ a rule in D. The E-satisfaction degree of r in M, denoted $deg_E^M(r)$, is defined as follows:

$$deg_E^M(r) = \begin{cases} 1 & \text{if } A \notin E \text{ or } \neg B_i \in E \text{ for some } i \in \{1, \ldots, n\}, \\ deg^M(C) & \text{otherwise.} \end{cases}$$

Here $deg^M(C)$ denotes the degree of satisfaction of C in M. $deg_E^M(r)$ is well-defined since the consequent of an applicable rule must be in the extension. M must therefore be a model of C.

The definition of the satisfaction degree of a *QCL* formula in a model is independent of a particular extension and remains unchanged. Given the satisfaction degrees of rules and formulas, we can define a preference ordering on the models of an extension E using any of the methods discussed earlier for *QCL*. For instance, in the cardinality based approach we can count the formulas in W and rules in D satisfied to a certain degree, etc.

Assume a preference ordering on the models of each extension E is fixed that way. Let $Pref(E)$ denote the maximally preferred models of E based on this ordering.

Definition 21. A formula F is a consequence of a choice default theory $\Delta = (D, W)$ iff F is true in each model of the set $\bigcup_{E \in Ext(\Delta)} Pref(E)$.

This definition generalizes both *QCL* and default logic (under sceptical inference): if $D = \emptyset$ the consequences coincide with the *QCL* consequences of W, and if the default theory does not contain ordered disjunction then the consequences are the formulas contained in all extensions.

Here is a small example illustrating what can be expressed in choice default logic. Assume you prefer having a Porsche over having a BMW over having a VW, but you also know that you cannot have an expensive car, and that normally a Porsche is expensive. Moreover, you prefer a convertible, unless you live in Germany where it rains quite often:

$$have(Porsche) \vec{\times} have(BMW) \vec{\times} have(VW),$$

$$expensive(Porsche) \rightarrow \neg have(Porsche),$$

$$true : expensive(Porsche)/expensive(Porsche),$$

$$true : \neg residence(Germany)/convertible \vec{\times} \neg convertible.$$

The single extension contains *expensive*(*Porsche*) and thus $\neg$*have*(*Porsche*). Independently of whether a cardinality or an inclusion based preference criterion is used, the most preferred models of the extension contain *have*(*BMW*) and *convertible* which are therefore consequences of the choice default theory. In general, the conclusions obtained this way describe what is true in the most desired worlds which are considered plausible.

9. Discussion

We proposed in this paper a new nonmonotonic propositional logic for representing ranked options. The logic has a new connective called ordered disjunction. Since ordered disjunction is fully embedded in the language, the ranking of the options may depend on the particular context, that is, it may depend on what else is true in the current situation.

We investigated computational aspects of *QCL* and showed how sets of formulas can be translated to stratified knowledge bases and, by results in [5], to classical propositional knowledge bases. We indicated a number of potential applications of the logic, and we presented alternative definitions of the consequence relation for applications where the

lexicographic ordering based on the number of best possible options is not adequate. Moreover, we investigated the relationship between *QCL* on one hand and circumscription respectively possibilistic logic on the other. We also proposed a combination of *QCL* and Reiter's default logic.

In the next subsection we clarify the role of formulas in *QCL*. We then discuss related work.

9.1. Beliefs and desires in QCL

The reader will have noticed that in *QCL* there is no syntactic distinction between formulas representing beliefs and formulas representing desires or intentions. For the applications discussed in Section 4 this did not pose any problems. In contexts where the distinction is important it may be useful (and necessary to avoid wishful thinking, see [36]) to split the premises T into two subsets, a set K of classical formulas representing beliefs about the real world and a set D of *QCL* formulas representing desires.

We call $BD = (K, D)$ where K is a set of propositional formulas and D a set of *QCL* formulas a belief-desire theory. For simplicity, we will assume that D is in normal form, that is $D = \{F_1, \ldots, F_n\}$ where $F_i = C_{i,1} \vec{\times} \cdots \vec{\times} C_{i,k_i}$. Let S be a set of classical formulas. A classical formula F is called *BD*-belief iff $K \vdash F$. F is called conditional desire given S iff

$$\bigvee_{M \in Pref(S \cup D)} \bigwedge_{1 \leqslant i \leqslant n} C_{i,deg^M(F_i)} \vdash F.$$

Here $Pref(S \cup D)$ denotes the preferred models of $S \cup D$, $deg^M(F_i)$ is the satisfaction degree of F_i in M. The disjunction used here can be viewed as a representation of the desires which can be satisfied in the most preferred models satisfying S.

An unconditional desire is a conditional desire given $\emptyset$. A *BD*-desire is a conditional desire given K. Obviously, each belief and each *BD*-desire is a *QCL* consequence of $K \cup D$.

Using this terminology it is possible to explain the behaviour of a "flat" *QCL* theory T which does not distinguish between beliefs and desires. Let $BD_T = (K_T, D_T)$ be an arbitrary partition of T into beliefs and desires. Then there are three categories of conclusions of T: beliefs, BD_T-desires, and "mixed" formulas containing, for instance, conjunctions of beliefs and desires. Moving a classical formula from D to K, or vice versa, may change the category of a conclusion, but never the set of conclusions. Thus, whenever it is unimportant to distinguish between beliefs and desires in the conclusions flat *QCL* can be used.

9.2. Related work

Preference handling in nonmonotonic reasoning and logic programming has received considerable attention in recent years. For an overview of some of the existing approaches see the discussion in [12] or the more recent [33]. Only few approaches allow for context dependent preferences. Existing work on context dependent preferences in nonmonotonic logics is based on explicit representations of a preference ordering together with names for default rules and sophisticated reformulations of the acceptable belief sets [9,10] or

makes heavy use of meta-predicates and compilation techniques [15,20]. The availability of ordered disjunction in *QCL* allows context dependent preferences among properties of a problem solution to be expressed much more conveniently.

QCL is also closely related to approaches in qualitative decision making [17]. Poole [31] aims at a combination of logic and decision theory. His approach incorporates quantitative utilities whereas our preferences are qualitative. Interestingly, Poole uses a logic *without* disjunction ("rather than using disjunction ... we want to use probability and decision theory to handle uncertainty", Section 1.5) whereas we *enhance* disjunction.

In [8] *CP*-networks are introduced, together with corresponding algorithms. These networks are a graphic representation, somewhat reminiscent of Bayes nets, for conditional preferences among feature values under the *ceteris paribus* principle. Our approach differs from *CP*-networks in at least two respects:

- Since ordered disjunction is fully embedded in the logic, we are able to represent more general preferences. Preferences in *CP*-networks are always total orders of the possible values of a variable.
- The ceteris paribus interpretation of preferences is different from our interpretation. The former views the available preferences as (hard) constraints on a global preference order. A set of *QCL* formulas, on the other hand, is more like a set of different criteria in multi-criteria optimization. For example, the *QCL* theory

  $$\left\{A \to (C \vec{\times} D), B \to (D \vec{\times} C), A, B\right\}$$

  is not inconsistent. There is reason to prefer C over D, and reason to prefer D over C. In *QCL* such conflicting preferences may neutralize each other, but do not lead to inconsistency.

In a series of papers [22,23,37], originally motivated by [7], the authors propose viewing conditional desires as constraints on utility functions. Intuitively, $D(a|b)$ stands for: the b-worlds with highest utility satisfy a. Our interpretation of ranked options is very different. Rather than being based on decision theory our approach can be viewed as giving a particular interpretation to the ceteris paribus principle: a model M_1 is preferred over a model M_2 if there is a formula $F \in T$ satisfied to degree j by M_1 and to degree $k > j$ by M_2 provided M_1 satisfies the other formulas in T at least as well as M_2. The last phrase is made precise as follows: for each degree $i \leqslant j$ M_1 satisfies at least as many formulas in $T \setminus \{F\}$ as M_2 to degree i.

Our work is also related to valued (sometimes also called weighted) constraint satisfaction [6,18,19,34]. A classical constraint problem is given by a set of variables V, a domain D_{v_i} for each variable v_i, and a set of constraints specifying conditions for solutions. A solution is an assignment of values from the respective domain to the variables satisfying all constraints.

A valued constraint, rather than specifying hard conditions, yields a ranking of solutions. A global ranking of solutions then is obtained from the rankings provided by the single constraints through some combination rule. In MAX-CSP [18], for instance, constraints assign penalties to solutions and solutions with the lowest penalty sum are

preferred. In fuzzy CSP [19] each solution is characterized by the worst violation of any constraint. Preferred solutions are those where the worst violation is minimal.

Determining a preferred *QCL* model can be viewed as a valued constraint problem where the variables are the atoms, the domains are the truth values, solutions are models, and the valued constraints are expressed as *QCL* formulas. It is a topic of further research whether combination rules used in constraint satisfaction have interesting applications in *QCL*, and vice versa.

In future work we want to investigate combinations of *QCL* and existing product configuration methodologies, e.g., [35], extensions of the inference relation to non-classical formulas, and the first order case. An application of the ideas underlying *QCL* to answer set programming is described in [11], an implementation of this approach based on the answer set solver *Smodels* is described in [13].

Acknowledgements

We would like to thank Jérôme Lang, David Makinson, Pierre Marquis, Ilkka Niemelä, Leon van der Torre and the anonymous referees for very helpful comments. The first author acknowledges support from DFG (Computationale Dialektik: BR 1817/1-5). The second and the third author were supported in part by the IUT de Lens, the Université d'Artois, the Nord/Pas-de-Calais Région under the TACT-TIC project and by the European Community FEDER Program.

Appendix A. Proofs of propositions

This appendix contains proofs of Propositions 2, 6, 8, 9, 10 and 11 and proofs of Lemmas 4 and 6. A few additional lemmas are proven which turn out to be useful for the main proofs. The proofs of Lemmas 1–3 and Proposition 1 are straightforward by induction on the structure of formula F and are therefore omitted. Proofs of Propositions 3 and 4 follow from Proposition 8 and the well-known properties of the lexicographic systems. Proposition 7 is a corollary of Propositions 2 and 6. Proposition 5 is a corollary of Proposition 7 and results in [5].

Fact 1. *Let F_1, F_2 be two arbitrary choice formulas. Let I be an interpretation, and $k \geqslant 1$ be such that $I \models_k F_1$. Then we also have*: $I \models_k F_1 \vec{\times} F_2$.

Proof of Proposition 2 (*Ordered disjunction is associative*). First, it easy to check that the two formulas $((F_1 \vec{\times} F_2) \vec{\times} F_3)$ and $(F_1 \vec{\times} (F_2 \vec{\times} F_3))$ have the same optionality, namely:

$$opt\big((F_1 \vec{\times} F_2) \vec{\times} F_3\big) = opt\big(F_1 \vec{\times} (F_2 \vec{\times} F_3)\big) = opt(F_1) + opt(F_2) + opt(F_3).$$

Let I be an interpretation, let us consider different cases of satisfaction of the formulas F_i by I:

- $I \models_k F_1$ then using Fact 1, we have $I \models_k F_1 \vec{\times} F_2$, and again applying Fact 1, we get $I \models_k (F_1 \vec{\times} F_2) \vec{\times} F_3$.
  On the other hand, using Fact 1, we also have: $I \models_k F_1 \vec{\times} (F_2 \vec{\times} F_3)$.
- $I \models_1 \neg F_1$ and $I \models_k F_2$. Then by definition, we have $I \models_{k+opt(F_1)} F_1 \vec{\times} F_2$, and applying Fact 1, we get $I \models_{k+opt(F_1)} (F_1 \vec{\times} F_2) \vec{\times} F_3$.
  On the other hand, using Fact 1, we also have: $I \models_k F_2 \vec{\times} F_3$, and by definition $I \models_{k+opt(F_1)} F_1 \vec{\times} (F_2 \vec{\times} F_3)$.
- $I \models_1 \neg F_1$, $I \models_1 \neg F_2$ and $I \models_k F_3$. Then by definition, we have $I \models_1 \neg(F_1 \vec{\times} F_2)$, and $I \models_{k+opt(F_1)+opt(F_2)} (F_1 \vec{\times} F_2) \vec{\times} F_3$.
  On the other hand, we also have: $I \models_{k+opt(F_2)} F_2 \vec{\times} F_3$, and $I \models_{k+opt(F_1)+opt(F_2)} F_1 \vec{\times} (F_2 \vec{\times} F_3)$.
- $I \models_1 \neg F_1$, $I \models_1 \neg F_2$ and $I \models_1 \neg F_3$. Then by definition, we have $I \models_1 \neg((F_1 \vec{\times} F_2) \vec{\times} F_3)$ and $I \models_1 \neg(F_1 \vec{\times} (F_2 \vec{\times} F_3))$. □

Proof of Proposition 6 (*Strong equivalences of formulas without* $\vec{\times}$). For the three equivalences, it is easy to check the equality of the optionality of the equivalent formulas.

(1) We assume for the sake of simplicity, and without loss of generality, that $m \leqslant n$. The equivalence becomes:

$$(A_1 \vec{\times} \cdots \vec{\times} A_n) \vee (B_1 \vec{\times} \cdots \vec{\times} B_m) \doteq (A_1 \vee B_1) \vec{\times} \cdots \vec{\times} (A_n \vee B_n)$$

where for $i > m$, $B_i = \bot$ (since A_i is classically equivalent to $A_i \vee \bot$).
Let us consider different cases of satisfaction of A_i's and B_j's by I.

- There exists $i > 0$ and $j > 0$ such that $I \models \neg A_1 \wedge \cdots \wedge \neg A_{i-1} \wedge A_i$ and $I \models \neg B_1 \wedge \cdots \wedge \neg B_{j-1} \wedge B_j$.
  This implies that:
  – for $k < min(i, j)$, $I \models \neg(A_k \vee B_k)$, and for $n \geqslant k \geqslant min(i, j)$: $I \models A_k \vee B_k$,
  – $I \models_i A_1 \vec{\times} \cdots \vec{\times} A_n$,
  – $I \models_j B_1 \vec{\times} \cdots \vec{\times} B_m$.
  This leads by definition to:

$$I \models_{min(i,j)} (A_1 \vec{\times} \cdots \vec{\times} A_n) \vee (B_1 \vec{\times} \cdots \vec{\times} B_m), \quad \text{and}$$
$$I \models_{min(i,j)} (A_1 \vee B_1) \vec{\times} \cdots \vec{\times} (A_n \vee B_n).$$

- $\forall i \leqslant n, I \models \neg A_i$ and there exists $j > 0$ such that $I \models \neg B_1 \wedge \cdots \wedge \neg B_{j-1} \wedge B_j$.
  This implies that:

$$I \models \neg(A_1 \vee B_1) \wedge \cdots \wedge \neg(A_{j-1} \vee B_{j-1}) \wedge (A_j \vee B_j) \wedge \cdots \wedge (A_n \vee B_n),$$
$$I \models_1 \neg(A_1 \vec{\times} \cdots \vec{\times} A_n),$$
$$I \models_j B_1 \vec{\times} \cdots \vec{\times} B_m.$$

  Hence by definition, we get:

$$I \models_j (A_1 \vec{\times} \cdots \vec{\times} A_n) \vee (B_1 \vec{\times} \cdots \vec{\times} B_m), \quad \text{and}$$
$$I \models_j (A_1 \vee B_1) \vec{\times} \cdots \vec{\times} (A_n \vee B_n).$$

- $\forall i \leqslant n, I \models \neg A_i$, and $\forall j \leqslant m, I \models \neg B_j$. This implies that: $\forall i \leqslant n, I \models \neg(A_i \vee B_i)$. Hence: $I \models_1 \neg(A_1 \vec{\times} \cdots \vec{\times} A_n) \vee (B_1 \vec{\times} \cdots \vec{\times} B_m)$, and $I \models_1 \neg((A_1 \vee B_1) \vec{\times} \cdots \vec{\times} (A_n \vee B_n))$.

(2) Again, we assume for the sake of simplicity, and without loss of generality, that $m \leqslant n$. The equivalence becomes:

$$(A_1 \vec{\times} \cdots \vec{\times} A_n) \wedge (B_1 \vec{\times} \cdots \vec{\times} B_m) \doteq (C_1 \vec{\times} \cdots \vec{\times} C_n)$$

where $C_i = [(A_1 \vee \cdots \vee A_i) \wedge B_i] \vee [A_i \wedge (B_1 \vee \cdots \vee B_i)]$, and $B_i = \bot$ for $m < i$. Let us consider different cases of satisfaction of A_i's and B_j's by I.

- There exists $i > 0$ and $j > 0$ such that $I \models \neg A_1 \wedge \cdots \wedge \neg A_{i-1} \wedge A_i$ and $I \models \neg B_1 \wedge \cdots \wedge \neg B_{j-1} \wedge B_j$.
  This implies that:
  – for $k < max(i, j)$, $I \models \neg C_i$, and for $n \geqslant k \geqslant max(i, j)$, $I \models C_k$,
  – $I \models_i A_1 \vec{\times} \cdots \vec{\times} A_n$,
  – $I \models_j B_1 \vec{\times} \cdots \vec{\times} B_m$.
  This leads by definition to:

$$I \models_{max(i,j)} (A_1 \vec{\times} \cdots \vec{\times} A_n) \wedge (B_1 \vec{\times} \cdots \vec{\times} B_m), \quad \text{and}$$
$$I \models_{max(i,j)} C_1 \vec{\times} \cdots \vec{\times} C_n.$$

- $\forall i \leqslant n, I \models \neg A_i$, or $\forall j \leqslant m, I \models \neg B_j$. This implies that: $\forall i \leqslant n, I \models \neg C_i$. Hence:

$$I \models_1 \neg\big[(A_1 \vec{\times} \cdots \vec{\times} A_n) \wedge (B_1 \vec{\times} \cdots \vec{\times} B_m)\big], \quad \text{and}$$
$$I \models_1 \neg[C_1 \vec{\times} \cdots \vec{\times} C_n].$$

(3) $I \models \neg(A_1 \vee \cdots \vee A_n)$ iff $\forall 0 \leqslant i \leqslant n, I \models \neg A_i$
iff $I \models_1 \neg(A_1 \vee \cdots \vee A_n)$ iff $I \models_1 \neg(A_1 \vec{\times} \cdots \vec{\times} A_n)$. □

Proof of Proposition 8 (*Sketch*). $T \mid\sim F$ iff $Skb(T) \vdash_{lex} F$.

Let $Skb(T) = S = (T^*, T_1, \ldots, T_n)$ be the stratified base associated with T. In the following, M_1 and M_2 denote two models of T. $|M_1^p(T)|$ and $|M_2^p(T)|$ denote the number of simple choice logic formulas from T satisfied to a degree p by M_1 and M_2, respectively. In a similar way, $|M_1^p(S)|$ and $|M_2^p(S)|$ denote the number of propositional formulas from T_p satisfied by M_1 and M_2, respectively. The idea of the proof is to show that the lexicographic ordering between models of T which is based on the number of satisfied choice formulas from T is the same as the one based on the number of satisfied propositional formulas from $Skb(T)$.

Fact 2. $|M_1^1(S)| \geqslant |M_2^1(S)|$ *iff* $|M_1^1(T)| \geqslant |M_2^1(T)|$.

Given, this fact, the proof of Proposition 8 follows immediately by applying iteratively the following lemma:

Lemma 8. *Let us assume that*:

$$\forall j = 1, \ldots, i, \quad \big|M_1^j(T)\big| = \big|M_2^j(T)\big| \quad \textit{iff} \quad \big|M_1^j(S)\big| = \big|M_2^j(S)\big|.$$

Then:

- $|M_1^{i+1}(T)| = |M_2^{i+1}(T)|$ *if and only if* $|M_1^{i+1}(S)| = |M_2^{i+1}(S)|$, *and*
- $|M_1^{i+1}(T)| > |M_2^{i+1}(T)|$ *if and only if* $|M_1^{i+1}(S)| > |M_2^{i+1}(S)|$.

Proof. We will only show the first item. The other case follows similarly by replacing the symbol $=$ by $>$.

First note that:

$$\begin{aligned}
&\big|M_1^i(S)\big| + \big|M_1^{i+1}(T)\big| \\
&\quad = \big|\{A_1 \vec{\times} \cdots \vec{\times} A_n \in T\colon\ M_1 \models A_1 \vee \cdots \vee A_i,\ \text{and } n \geqslant i+1\}\big| \\
&\qquad + \big|\{A_1 \vec{\times} \cdots \vec{\times} A_n \in T\colon\ M_1 \models \neg A_1 \wedge \cdots \wedge \neg A_i \wedge A_{i+1},\ \text{and } n \geqslant i+1\}\big| \\
&\quad = \big|\{A_1 \vec{\times} \cdots \vec{\times} A_n \in T\colon\ M_1 \models A_1 \vee \cdots \vee A_i \vee A_{i+1},\ \text{and } n \geqslant i+1\}\big|.
\end{aligned}$$

Hence:

$$(1)\quad \big|M_1^{i+1}(S)\big| = \big|M_1^i(S)\big| + \big|M_1^{i+1}(T)\big| - \big|T^{i+1}\big|$$

where $T^{i+1} = \{A_1 \vec{\times} \cdots \vec{\times} A_n \in T\colon\ n = i+1\}$.

It is clear that any model M of T satisfies $A_1 \vee \cdots \vee A_n$ where $A_1 \vec{\times} \cdots \vec{\times} A_n \in T^{i+1}$.

Given (1) the proof follows straightforwardly. Indeed,

- if $|M_1^i(S)| = |M_2^i(S)|$ and $|M_1^{i+1}(T)| = |M_2^{i+1}(T)|$ then using (1) we also have $|M_1^{i+1}(S)| = |M_2^{i+1}(S)|$;
- if $|M_1^i(S)| = |M_2^i(S)|$ and $|M_1^{i+1}(S)| = |M_2^{i+1}(S)|$ then using again (1) we also have $|M_1^{i+1}(T)| = |M_2^{i+1}(T)|$. □

Proof of Proposition 9 (*Sketch*). $T \mid\sim_{inc} F$ iff $Skb(T) \vdash_{inc} F$.

In the following, $M_1^p(T)$ and $M_2^p(T)$ denote the set of simple choice logic formulas from T satisfied to a degree p by M_1 and M_2, respectively. In a similar way, $M_1^p(S)$ and $M_2^p(S)$ denote the set of propositional formulas from T_i satisfied by M_1 and M_2, respectively. We define *Skb*-inclusion-preference between models of T exactly like in Definition 13, by replacing $M_1^j(T)$ and $M_2^j(T)$ by $M_1^j(S)$ and $M_2^j(S)$, respectively. The idea of the proof is then to show that the T-inclusion-based ordering coincides with *Skb*-inclusion-based ordering. We first give two facts and show a lemma.

Fact 3. *If* $M \models_k A_1 \vec{\times} \cdots \vec{\times} A_n$ *then for any* $j = k, \ldots, n$ *we have* $M \models A_1 \vee \cdots \vee A_j$.

Fact 4. *If* $M \models A_1 \vee \cdots \vee A_j$ *then there exists some* $k \leqslant j$ *such that* $M \models_k A_1 \vec{\times} \cdots \vec{\times} A_n$.

Lemma 9. *If there exists some* k *such that for all* $j = 1, \ldots, k$ *we have* $M_1^j(T) = M_2^j(T)$ *then for all* $j = 1, \ldots, k$ *we have* $M_1^j(S) = M_2^j(S)$. *The converse is also true.*

Proof.

- Assume that for all $j = 1, \ldots, k$ we have

$$M_1^j(T) = M_2^j(T) \tag{A.1}$$

and there exists some $i \leqslant k$ such that $M_1^i(S) \neq M_2^i(S)$.

Let $A_1 \vee \cdots \vee A_i$ be a formula in S_i satisfied by M_1 but falsified by M_2. Using Fact 4, we have $M_1 \models_j A_1 \vec{\times} \cdots \vec{\times} A_n$ from some $j \leqslant i$. Using the hypothesis (1) this implies that $M_2 \models_j A_1 \vec{\times} \cdots \vec{\times} A_n$. Applying Fact 3, we get $M_2 \models A_1 \vee \cdots \vee A_i$ and hence a contradiction.

- The other direction follows the same schema. Namely, assume that for all $j = 1, \ldots, k$ we have

$$M_1^j(S) = M_2^j(S) \tag{A.2}$$

and there exists some $i \leqslant k$ such that $M_1^i(T) \neq M_2^i(T)$.
Let $A_1 \vec{\times} \cdots \vec{\times} A_n$ be a formula in T satisfied by M_1 to a degree i but is not satisfied by M_2 to a degree i. Using Fact 3, we get $M_1 \models A_1 \vee \cdots \vee A_i$ (and by definition for $k < i$, $M_1 \not\models A_1 \wedge \cdots \wedge A_k$). Hypothesis (2) implies $M_2 \models A_1 \vee \cdots \vee A_i$. Applying Fact 4, we get $M_2 \models_k A_1 \vec{\times} \cdots \vec{\times} A_n$ for some $k \leqslant i$, hence a contradiction. □

Given this lemma, we have.

Lemma 10. *M_1 is Skb-inclusion-preferred to M_2 iff M_1 is T-inclusion-preferred to M_2.*

Proof.

- Let M_1 be *Skb*-inclusion-preferred to M_2 but M_1 is not T-inclusion-preferred to M_2. By definition, M_1 is *Skb*-inclusion-preferred to M_2 implies that there exists some i such that for all $j < i$ we have $M_1^j(S) = M_2^j(S)$ and $M_2^i(S) \subset M_1^i(S)$. From Lemma 5, we also get: for all $j < i$ we have $M_1^j(T) = M_2^j(T)$.
Moreover, $M_2^i(S) \subset M_1^i(S)$ means that there exists a formula $A_1 \vee \cdots \vee A_i$ which is satisfied by M_1 but not by M_2. This means that there exists a choice logic formula $A_1 \vec{\times} \cdots \vec{\times} A_n$ which is satisfied to degree i by M_1 but not by M_2 (which means that $M_2^i(T)$ is not included in $M_1^i(T)$).
Now, assume that there exists a simple choice logic formula $A_1 \vec{\times} \cdots \vec{\times} A_n$ which is satisfied by M_2 to a degree i but not by M_1. From Fact 3, this implies that $M_2 \models A_1 \vee \cdots \vee A_i$, which implies $M_1 \models A_1 \vee \cdots \vee A_i$ (since $M_2^i(S) \subset M_1^i(S)$) and this contradicts the fact that $A_1 \vec{\times} \cdots \vec{\times} A_n$ is not satisfied by M_1 to a degree i.
- The proof is symmetric. Let M_1 be T-inclusion-preferred to M_2 but M_1 is not *Skb*-inclusion-preferred to M_2. By definition, M_1 is Incl-T-preferred to M_2 implies that there exists some i such that for all $j < i$ we have $M_1^j(T) = M_2^j(T)$ and $M_2^i(T) \subset M_1^i(T)$. From Lemma 5, we also get: for all $j < i$ we have $M_1^j(S) = M_2^j(S)$.
Moreover, $M_2^i(T) \subset M_1^i(T)$ means that there exists a simple choice logic formula $A_1 \vec{\times} \cdots \vec{\times} A_n$ which is satisfied to a degree i by M_1 but not by M_2. This means that there exists a propositional formula $A_1 \vee \cdots \vee A_i$ in S_i which is satisfied by M_1 but not by M_2 (which means that $M_2^i(S)$ is not included in $M_1^i(S)$).
Now, assume that there exists a formula $A_1 \vee \cdots \vee A_i$ in S_i which is satisfied by M_2 but not by M_1. Using Fact 4, this implies that $M_2 \models_i A_1 \vec{\times} \cdots \vec{\times} A_n$, which implies $M_1 \models_i A_1 \vec{\times} \cdots \vec{\times} A_n$ (since $M_2^i(T) \subset M_1^i(T)$) and this contradicts the fact that $A_1 \vee \cdots \vee A_i$ is not satisfied by M_1. □

Proof of Proposition 10 (*Representing circumscription in QCL*). We first observe that T and T' have exactly the same models since the propositional counterpart of each formula in $T' \setminus T$ (obtained by replacing $\vec{\times}$ with $\vee$) is a tautology. Now assume M is not $<_{V_1,\ldots,V_n;W}$-preferred. Then there exists a model M' of T and an i such that M' agrees with M on atoms in W and on atoms in $V_1,\ldots,V_{i-1}$ and makes fewer atoms in V_i true than M. Thus the formulas in $T' \setminus T$ satisfied to degree $1,\ldots,n-1$ by M' are the same as those satisfied to degree $1,\ldots,n-1$ by M. Moreover, M' satisfies a superset of those satisfied by M to degree i and thus M is not an inclusion preferred model of T'.

Conversely, let M be a non-preferred model of T'. Then there is a model M' and an i such that the formulas in $T' \setminus T$ satisfied to degree i by M' are a superset of those satisfied to degree i by M, and those satisfied to any degree $j < i$ by the two models coincide. Since changing the truth value of a variable in W always changes the satisfaction degree of some formula in T' from 1 to 2, M and M' agree on atoms in W and there must be a variable in V_i false in M' and true in M. Since all variables in V_i which are false in M are also false in M', it follows that M is not $<_{V_1,\ldots,V_n;W}$-preferred. □

Proof of Proposition 11 (*Representing QCL in circumscription*). The proof is based on the following

Lemma 11. *Let $F_{i,k}$ denote the kth ordered disjunct of formula $F_i \in T$. If M is a model of T, then $M' = M \cup \{ab_{i,h} \mid h \leqslant j, M \models_j F_i\}$ is a model of T'. Vice versa, if M' is a model of T', then $M' \setminus AB_1 \cup \cdots \cup AB_n$ is a model of T.*

To prove the lemma consider a model M of T. M satisfies each formula F_i to some degree j. Implications $(1),\ldots,(j-1)$ of the translation of F_i are satisfied in M' because M' contains $ab_{i,1},\ldots,ab_{i,j-1}$. Implication (j) is satisfied because M and thus M' satisfies $F_{i,j}$. Implications (k) for $k > j$ are satisfied because M' does not contain $ab_{i,j}$.

Conversely, let M' be a model of T', F_j a formula in T with optionality $m+1$. M' contains an arbitrary subset S (possibly empty) of $\{ab_{j,1},\ldots,ab_{j,m}\}$. S satisfies at least one of the preconditions of the implications obtained from translating F_j and thus the consequent of the implication which is one of the options of F_j. Since F_j does not contain any of the new ab-atoms it follows that M satisfies F_j. This concludes the proof of the lemma.

Now assume M_1 is a non-preferred model of T, that is, there is a model M_2 inclusion preferred over M_1. Let j be the smallest satisfaction degree at which the two models differ, and let F_m be a formula satisfied to degree j in M_2, but not in M_1.

Consider a model M_1' of T' such that $M_1' \setminus AB_1 \cup \cdots \cup AB_n = M_1$. Let M_2' be the model of T' constructed from M_2 according to the lemma. Since M_1 and M_2 agree on formulas satisfied to any degree smaller than j and M_2 satisfies more formulas to degree j we have, for each $h \leqslant j$, $ab_{r,h} \in M_2'$ implies $ab_{r,h} \in M_1'$. Moreover, since M_1' must contain $ab_{m,j}$ whereas M_2' does not, it is straightforward to show that $M_2' <_{AB_1,\ldots,AB_n;\emptyset} M_1'$.

Conversely, assume M_1 is a maximally preferred model of T. Let M_1' be the model of T' obtained through the construction in the lemma. We show by contradiction that M_1' is maximally $<_{AB_1,\ldots,AB_n;\emptyset}$-preferred. Assume for some model M_2' of T' we have $M_2' <_{AB_1,\ldots,AB_n;\emptyset} M_1'$. Since every model different from M_1' which agrees with M_1' on

atoms in T must contain more $ab_{i,j}$ literals than M_1', M_2' cannot agree with M_1' on atoms in T. Using the lemma we thus have that $M_2 = M_2' \setminus AB_1 \cup \cdots \cup AB_n$ is a model of T different from M_1. Let j be the smallest index such that, for some k, $ab_{k,j}$ is false in M_2' but true in M_1'. M_2 satisfies all formulas $F \in T$ with $deg^{M_1}(F) \leqslant j$ with a degree at least as good as M_1. Moreover, M_2 satisfies F_k at least to degree j whereas M_1 does not. Therefore M_2 is inclusion preferred to M_1, contrary to our assumption.

We thus have for each maximally inclusion preferred model of T a corresponding model of T' agreeing on F and vice versa. □

Proof of Lemma 4 (*Possibility distribution associated with* $\Delta_\oplus$). We only show the case where two Δ-knowledge bases are merged. The general case follows in a similar way.

Recall that

$$\pi_{\Delta_\oplus} = \big\{\big[\phi_i \wedge \psi_j, \oplus(\alpha_i, \beta_j)\big]\colon\ [\phi_i, \alpha_i] \in \Delta_1 \text{ and } [\psi_j, \beta_j] \in \Delta_2\big\}.$$

Note first that if for some interpretation I, $\pi_1(I) = 0$ or $\pi_2(I) = 0$ then this means that I falsifies all propositional formulas of Δ_1, or I falsifies all propositional formulas of Δ_2. Hence, I also falsifies all propositional formulas of $\Delta_\oplus$. Then, $\pi_{\Delta_\oplus} = 0 = \oplus(\pi_1(I), \ldots, \pi_n(I))$, since $\oplus(0, 0) = 0$.

Now assume that I satisfies at least one propositional formula from Δ_1 and at least one propositional formula from Δ_2. Then $\pi_{\Delta_\oplus}$ is computed as follows:

$$\begin{aligned}
\pi_{\Delta_\oplus}(I) &= max\big\{\oplus(\alpha_i, \beta_j)\colon\ I \models \phi_i \wedge \psi_j \text{ and } \big[\phi_i \wedge \psi_j, \oplus(\alpha_i, \beta_j)\big] \in \Delta_\oplus\big\} \\
&= max\big\{\oplus(\alpha_i, \beta_j)\colon\ I \models \phi_i \wedge \psi_j \text{ and } [\phi_i, \alpha_i] \in \Delta_1,\ [\psi_j, \beta_j] \in \Delta_2\big\} \\
&= max\big\{\oplus(\alpha_i, \beta_j)\colon\ I \models \phi_i, [\phi_i, \alpha_i] \in \Delta_1 \text{ and } I \models \psi_j,\ [\psi_j, \beta_j] \in \Delta_2\big\}.
\end{aligned}$$

Since $\oplus$ satisfies the unanimity condition then when α_i and β_j are maximal then $\oplus(\alpha_i, \beta_j)$ is also maximal.

Hence,

$$\begin{aligned}
\pi_{\Delta_\oplus}(I) &= \oplus\big(max\big\{\alpha_i\colon\ I \models \phi_i,\ [\phi_i, \alpha_i] \in \Delta_1\big\}, max\big\{\beta_j\colon\ I \models \psi_j,\ [\psi_j, \beta_j] \in \Delta_2\big\}\big) \\
&= \oplus\big(\pi_1(I), \pi_2(I)\big).
\end{aligned}$$

Proof of Lemma 6 (*Basic choice formulas as* Δ*-knowledge bases*). The proof is immediate. Indeed, if for all i, $I \not\models A_i$, then from Definition 16 we have $\pi(I) = 0$, and from Definition 3 we have $I \models_1 \neg F$. Now, assume that I satisfies some A_i. Then $\pi(I) = \alpha_k$ means that $I \models A_k$, and for $i = 1, \ldots, k-1$, we have $I \not\models A_i$, and this is exactly equivalent to $I \models_k F$. □

References

[1] S. Benferhat, C. Cayrol, D. Dubois, J. Lang, H. Prade, Inconsistency management and prioritized syntax-based entailment, in: Proc. IJCAI-93, Chambéry, France, 1993, pp. 640–645.

[2] S. Benferhat, D. Dubois, S. Kaci, H. Prade, Bridging logical, comparative and graphical possibilistic representation frameworks, in: Proceedings of the 6th European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty (ECSQARU'01), Toulouse, France, 2001, pp. 422–431.

[3] S. Benferhat, S. Kaci, A logical representation and fusion of prioritized information based on guaranteed possibility measures: application to the encoding of distance-based merging for classical bases, Artificial Intelligence 148 (1–2) (2003) 291–333.
[4] S. Benferhat, D. Dubois, H. Prade, Some syntactic approaches to the handling of inconsistent knowledge bases: a comparative study. Part II: The prioritized case, in: E. Orlowska (Ed.), Logic at Work, vol. 24, Physica-Verlag, Heidelberg, 1998, pp. 473–511.
[5] S. Benferhat, S. Kaci, D. Le Berre, M.-A. Williams, Weakening conflicting information for iterated revision and knowledge integration, in: Proc. IJCAI-01, Seattle, WA, 2001, pp. 109–115.
[6] S. Bistarelli, U. Montanari, F. Rossi, Semiring-based constraint solving and optimization, J. ACM 44 (2) (1997) 201–236.
[7] C. Boutilier, Towards a logic for qualitative decision theory, in: Proc. Principles of Knowledge Representation and Reasoning (KR-94), Bonn, 1994, pp. 75–86.
[8] R.I. Boutilier, C. Brafman, H.H. Hoos, D. Poole, Reasoning with conditional ceteris paribus preference statements, in: Proc. UAI-99, Stockholm, Sweden, 1999, pp. 71–80.
[9] G. Brewka, Well-founded semantics for extended logic programs with dynamic preferences, J. Artificial Intelligence Res. 4 (1996) 19–36.
[10] G. Brewka, Representing meta-knowledge in Poole-systems, Studia Logica 67 (2001) 153–165.
[11] G. Brewka, Logic programming with ordered disjunction, in: Proc. 17th National Conference on Artificial Intelligence (AAAI-02), Edmonton, AB, 2002, pp. 100–105.
[12] G. Brewka, T. Eiter, Preferred answer sets for extended logic programs, Artificial Intelligence 109 (1999) 297–356.
[13] G. Brewka, I. Niemelä, T. Syrjänen, Implementing ordered disjunction using answer set solvers for normal programs, in: Proc. JELIA, Cosenza, Italy, 2002, Springer, Berlin, 2002, pp. 444–455.
[14] S. Coste-Marquis, P. Marquis, On stratified belief base compilations, Ann. Math. Artificial Intelligence, in press.
[15] J. Delgrande, T. Schaub, H. Tompits, Logic programs with compiled preferences, in: Proc. European Conference on Artificial Intelligence, Berlin, 2000, pp. 392–398.
[16] D. Dubois, P. Hajek, H. Prade, Knowledge-driven versus data-driven logics, J. Logic Language Inform. 9 (2000) 65–89.
[17] J. Doyle, R.H. Thomason, Background to qualitative decision theory, AI Magazine 20 (2) (1999) 55–68.
[18] E.C. Freuder, R.J. Wallace, Partial constraint satisfaction, Artificial Intelligence 58 (1) (1992) 21–70.
[19] H. Fargier, J. Lang, T. Schiex, Selecting preferred solutions in fuzzy constraint satisfaction problems, in: Proc. First European Congress on Fuzzy and Intelligent Technologies, 1993.
[20] B. Grosof, Diplomat: compiling prioritized rules into ordinary logic programs for e-commerce applications, in: Proc. National Conference on Artificial Intelligence (AAAI-99), Orlando, FL, 1999, pp. 912–913 (intelligent systems demonstration abstract).
[21] S. Kraus, D. Lehmann, M. Magidor, Nonmonotonic reasoning, preferential models and cumulative logics, Artificial Intelligence 44 (1990) 167–207.
[22] J. Lang, Conditional desires and utilities—an alternative logical approach to qualitative decision theory, in: Proc. 12th European Conference on Artificial Intelligence (ECAI-96), Budapest, Hungary, 1996, pp. 318–322.
[23] J. Lang, L. van der Torre, E. Weydert, Utilitarian desires, Autonomous Agents and Multi-Agent Systems 5 (3) (2002) 329–363.
[24] D. Lehmann, Another perspective on default reasoning, Ann. Math. Artificial Intelligence 15 (1) (1995) 61–82.
[25] D. Lehmann, M. Magidor, What does a conditional knowledge base entail?, Artificial Intelligence 55 (1992) 1–60.
[26] V. Lifschitz, Circumscription, in: Handbook of Logic in AI and Logic Programming, vol. 3, Oxford University Press, Oxford, 1994, pp. 298–352.
[27] J. McCarthy, Circumscription—A form of nonmonotonic reasoning, Artificial Intelligence 13 (1980) 27–39.
[28] J. McCarthy, Applications of circumscription to formalizing common sense knowledge, Artificial Intelligence 28 (1986) 89–116.
[29] Y. Moinard, Note about cardinality-based circumscription, Artificial Intelligence 119 (1–2) (2000) 259–273.
[30] R.C. Moore, Semantical considerations on nonmonotonic logic, Artificial Intelligence 25 (1985) 75–94.

[31] D. Poole, The independent choice logic for modelling multiple agents under uncertainty, Artificial Intelligence 94 (1–2) (1997) 7–56.
[32] R. Reiter, A logic for default reasoning, Artificial Intelligence 13 (1980) 81–132.
[33] T. Schaub, K. Wang, A comparative study of logic programs with preference, in: Proc. IJCAI-01, Seattle, WA, 2001, pp. 631–637.
[34] T. Schiex, H. Fargier, G. Verfaillie, Valued constraint satisfaction problems: Hard and easy problems, in: Proc. IJCAI-95, Montreal, Quebec, 1995, pp. 631–637.
[35] T. Soininen, I. Niemelä, J. Tiihonen, R. Sulonen, Representing configuration knowledge with weight constraint rules, in: Proc. of the AAAI Spring Symposium on Answer Set Programming: Towards Efficient and Scalable Knowledge, Stanford, CA, 2001.
[36] R. Thomason, Desires and defaults: a framework for planning with inferred goals, in: Proc. 7th Conference on Principles of Knowledge Representation and Reasoning (KR-2000), Breckenridge, CO, 2000, pp. 702–713.
[37] L. van der Torre, E. Weydert, Parameters for utilitarian desires in a qualitative decision theory, Appl. Intelligence 14 (2001) 285–301.

Exploiting the real power of unit propagation lookahead

Daniel Le Berre

Business & Technology Research Laboratory, The University of Newcastle, Newcastle NSW 2308 Australia

Abstract

One of the best SAT solvers for random 3-SAT formulae, SATZ, is based on a heuristic called unit propagation lookahead (UPL). Unfortunately, it does not perform so well on specific structured instances, especially on the ones coming from an area where a huge interest for SAT has emerged in recent years: symbolic model checking (SMC). We claim that all the power of this heuristic is not used in SATZ, and that UPL can be extended to solve some real world structured problems, where the major competitors are using intelligent backtracking or specific deduction rules. We introduce a preprocessing technique that can be applied to simplify instances containing equivalent literals. This technique is based on UPL, so it can be easily added to any solver using this heuristic. We compare our approach to the new extension of SATZ for equivalency reasoning (EqSATZ) and another approach, the Stålmarck method, which is mainly used in SMC.

1 Introduction

During the last decade, a lot of work has been done around the satisfaction problem (SAT). Most techniques can be classified either as complete or incomplete, depending on their ability to prove the inconsistency of a formula. In this paper we focus on the first group, since one of our applications, symbolic model checking, will be used to prove the inconsistency of formulae. Most of the solvers from this group inherit the backtracking version of the Davis and Putnam algorithm, DPLL [1]. But at least two different approaches can be found: the one suited for random 3-SAT formulae, with C-SAT [2], POSIT [3], SATZ [4,5], and another which is more effective on real world problems, SATO [6], RELSAT [7], GRASP [8] and recently Chaff [9]. The former is based on a powerful lookahead technique which reduces the search space whereas the lat-

ter uses "intelligent backtracking" to achieve the same goal [1]. More recently new solvers which use an entirely different approach, deduction rules, have been developed: HeerHugo [10] and EqSATZ [11]. EqSATZ is to the best of our knowledge the only solver able to solve the full parity-32 problem, hence achieving the second propositional challenge proposed by [12] [2]. Such approach is not new in an area where a huge interest for SAT has emerged in recent years: symbolic model checking. One widely used solver reporting good results is based on the patented deduction rules based Stålmarck method [14,15]. Constrastly, DPLL-based algorithms do not perform very well on these instances [16,17]. We propose an explanation.

The development of SAT solvers for randomly generated formulae is based on the assumption that if a problem has a specific structure, then there exists a way to use this structure to reduce the size of the problem to a "kernel" CNF formula without any structure, such as a random formula. We believe that deduction rules may well be one option, and that the success of Stålmarck method on symbolic model checking instances and EqSATZ on parity32 instances tend to confirm this assumption. One of the two arising problems is to decide what kind of rules to use for a given structured problem, and the other to implement an efficient method to fire these rules.

We address in this paper both problems. We will show that the lookahead technique used in SATZ can be extended to retrieve both implied and equivalent literals, the same kind of information handled by deduction rules based EqSATZ, and that symbolic model checking instances can be considerably reduced using this information. Since only classical DPLL data are used, this technique can be easily implemented on existing DPLL based implementation. The other contribution of this paper is to show the link between techniques based on DPLL and Stålmarck.

2 SATZ and EqSATZ

The efficiency of SATZ on random 3-SAT formulae is due to the very efficient lookahead technique used to choose the next branching variable. The idea is that before deciding which variable to branch on, taking a look to the surrounding search space using a simple unit propagation can help. This branching rule used in SATZ can be defined as in algorithm 1 described below.

[1] Chaff is also heavily based on randomization and restarts, but discussions about that matter are out of the scope of this paper.

[2] [13] proposed a technique solving the compressed version of the instances

This technique was previously used in C-SAT and POSIT, but SATZ is the result of a vast amount of experiment to correctly choose the lookahead search space [4]. The original *propagate*() method in SATZ is a simple unit propagation. An extension of SATZ (SATZ213) including in *propagate*() a second level of UPL on variables occurring in reduced binary clauses is described in [5]. One of the consequence of this extension is to possibly increase the number of literals fixed during the *propagate*() process.

Algorithm 1: SATZBranchingRule(KB, V)

```
Data: a set of clauses KB ;
      a set of variables V ;
begin
    for each x ∈ V do
        % propagate() returns the set of ;
        % literals fixed during the process. ;
        L1 ← propagate(KB ∧ x) ;
        L2 ← propagate(KB ∧ ¬x) ;
        if ⊥ ∈ L1 and ⊥ ∈ L2 then
            return "KB is not satisfiable";
        if ⊥ ∈ L1 then
            x ← 0;
        else
            if ⊥ ∈ L2 then
                x ← 1;
            else
                % neither L1 nor L2 contain ⊥ ;
                % (1) ;
                % compute a heuristic H(x) for x. ;
    % branch on x ∈ V maximizing H(x) ;
end
```

In the following, $UPL(KB, l)$ will refer to the set of literals satisfied during the *propagate*$(KB \wedge l)$ process. Moreover $\bot \in UPL(KB, l)$ iff the process falsifies a clause in the base. Obviously, we have $l \in UPL(KB, l)$.

Let $IMPL(KB)$ be the set of implied literals of KB. $(\forall l \in IMPL(KB), KB \models l)$. We have $UPL(KB, l) \subseteq IMPL(KB \wedge l)$, that is, $UPL(KB, l)$ is an approximation of $IMPL(KB \wedge l)$.

Let $UPL_1(KB, l)$ be the set of literals satisfied by unit propagation when l is

satisfied in KB. $UPL_1(KB, l)$ denotes the result of the original *propagate*() process in SATZ. In the same way, $UPL_2(KB, l)$ will refer to the set of literals satisfied by unit propagation plus a second level of UPL when l is satisfied in KB as in SATZ213. From the definition of these sets we have $UPL_1(KB, l) \subseteq UPL_2(KB, l) \subseteq IMPL(KB \wedge l)$.

$UPL_2(KB, l)$ is a better approximation of $IMPL(KB \wedge l)$ than $UPL_1(KB, l)$ but the complexity of these procedures are different: $UPL_1(KB, l)$ relies on $|V|$ calls to a unit propagation algorithm whereas $UPL_2(KB, l)$ relies on possibly $|V| * (|V| - 1)$ calls to the same algorithm. (Depending of the choice of the set of variables for the second level of UPL).

$UPL_2(KB, l)$ has been reported useful to solve hard random 3-SAT formulae [5]. Unfortunately, structured problems are still really difficult for SATZ [18].

EqSATZ[11] can be viewed as an extension of SATZ with deduction rules, that can deal with some structured problems (e.g. Parity32). EqSATZ is interestingly also suited for a range of bounded model checking problems introduced by [16] which we will refer to as BMC instances.

EqSATZ focuses on a special kind of formula, biconditionals, especially those of the form (1) $l_1 \leftrightarrow l_2$ denoting equivalent literals, and those of the form (2) $l_1 \leftrightarrow l_2 \leftrightarrow l_3$. On the latter, some deduction rules are used to discover new equivalent literals:

$(R5)\ l_1 \leftrightarrow l_2 \leftrightarrow l_3, l_1 \leftrightarrow l_2 \leftrightarrow l_4 \quad \equiv l_3 \leftrightarrow l_4$

$(R6)\ l_1 \rightarrow (l_3 \leftrightarrow l_4), \neg l_1 \rightarrow (l_3 \leftrightarrow l_4) \quad \equiv l_3 \leftrightarrow l_4$

$(R7)\ l_1 \rightarrow (l_3 \leftrightarrow l_4), \neg l_1 \rightarrow (\neg l_3 \leftrightarrow l_4) \equiv l_1 \leftrightarrow l_3 \leftrightarrow l_4$

Note that formula (1) appears in CNF as 2 binary clauses $(\neg l_1 \vee l_2, l_1 \vee \neg l_2)$ and (2) as 4 ternary clauses $(\neg l_1 \vee \neg l_2 \vee l_3, \neg l_1 \vee l_2 \vee \neg l_3, l_1 \vee l_2 \vee l_3, l_1 \vee \neg l_2 \vee \neg l_3)$.

EqSATZ is based on a syntactical recognition of equivalent literals and biconditionals from their clausal form. Unfortunately, EqSATZ can miss some formulas: take for instance the three clauses $l_1 \rightarrow l_2, l_2 \rightarrow l_3, l_3 \rightarrow l_1$, EqSATZ is unable to detect that $l_1 \leftrightarrow l_3$. (Note that the "syntactical" reasoning is used to initiate a more "semantical" one using the above rules, as pointed out by Chu-Min Li in a personal communication.)

We will show that it is possible to recover this information, and more, *semantically* using only unit propagation lookahead (UPL). Furthermore, EqSATZ is a complete re-implementation of SATZ for deduction rules. We will show that EqSATZ like deduction rules could be added to SATZ with minor modifications in the code.

3 Stålmarck's method

Stålmark's approach [14] is a patented method that uses deduction rules instead of resolution. The input (any propositional formula) is translated into a conjunction of equivalent literals or sub-formulas of the form $x \leftrightarrow u \otimes v$, where $\otimes$ is a binary connective. The deduction rules allow to either fix the truth value of a variable or to detect new equivalent literals. The use of these rules on the initial formula is called 0-saturation. Using the same idea as UPL, 1-saturation consists of fixing the truth value of a given variable to true then false, and each time using 0-saturation (instead of unit propagation for UPL) to deduce new formulae. Those found in both cases are kept. More generally, n-saturation involves fixing the truth value of one variable and applying (n-1)-saturation on the remaining variables. n-saturation is used sequentially on each variable until no more information can be deduced, then (n+1)-saturation is applied. One can view this method as a breadth-first search compared to the depth-first search of DPLL-based techniques. It appears experimentally that 2-saturation is usually sufficient to solve symbolic model checking instances, which explains the success of this method in this domain. Unlike DPLL-based approaches, this method requires a lot of memory and can be compared in that way to the original Davis and Putnam algorithm [19].

HeerHugo [10] is a research oriented implementation of Stålmarck's method adapted for CNF, with several improvements, such as the integration of local search for instance.

4 The real power of Unit Propagation Lookahead

Let us now highlight the semantics of $UPL(KB, l)$.

Proposition 1 $u \in UPL(KB, l) \Rightarrow KB \models l \rightarrow u$

This is an obvious result, based on the definition of UPL. There is also the particular case of $\bot$. This proposition denotes the way SATZ fixes the truth value of some variables during unit propagation lookahead.

Proposition 2 $\bot \in UPL(KB, l) \Rightarrow KB \models \neg l$

The following results concern more interesting cases. For instance, the first one was indirectly used by C-SAT [2] to compute implied literals in a preprocessing step. (It was based on production of binary clauses instead of making explicitly an intersection.)

Proposition 3 $u \in UPL(KB, l) \cap UPL(KB, \neg l) \Rightarrow KB \models u$

Here we obtain *some* implied literals. Note that we do not claim to find all the implied literals of KB with this method, but a subset of them ($(UPL(KB, l) \cap UPL(KB, \neg l)) \subseteq IMPL(KB)$) . Hence it is easy to detect and propagate during the search some implied literals by keeping track of literals satisfied during the $propagate()$ process. In that way, we extend the process used in C-SAT at the root of the search tree to all the nodes.

We can easily modify the algorithm 1 by adding at (1) the statement:

for *each* $y \in L_1 \cap L_2$ **do**
$\lfloor\ y \leftarrow 1;$

To illustrate the impact that such simple result can have on an UPL based algorithm, let us relate the effect of adding the implied literals propagation rule in SATZ. Chu-Min Li has added to the latest version of SATZ (SATZ214) the implied literal propagation, leading to SATZ215[3]. This new version of SATZ can solve within 15s each the two instances 3bitadd_31 and 3bitadd_32 from the Beijing Challenge[4] on SAT-Ex. No other complete SAT solver has been reported to solve 3bitadd_31 in reasonable time and only SATZ213 has been reported to solve 3bitadd_32 (in more than 800s [18]). This result can be compared to the one of the distributed version a SATZ214: //SATZ [20] needs 159s to solve 3bitadd_32 on 20 comparable computers. However, these instances have been solved using randomized algorithms [21] or local search [22].

The next result provides a way to detect equivalent literals similarly to the detection of implied literals.

Proposition 4 $u \in UPL(KB, l)$ *and* $\neg u \in UPL(KB, \neg l) \Rightarrow KB \models l \leftrightarrow u$

[3] Available at http://www.laria.u-picardie.fr/ cli/satz215.c
[4] Available at http://www.cirl.uoregon.edu/crawford/beijing/

This result is important since it leads to the first approach to remove some equivalent literals in a SATZ-like DPLL. The set of literals satisfied during the unit propagation lookahead is the only information required. Then a simple intersection between two sets of literals will lead to a set of equivalent literals.

The treatment made on these equivalent literals depends on the location of their detection:

- if this process occurs during a preprocessing step (at the root of the search tree), the equivalent literals can all be replaced by a unique one then removed from the base KB. We will see later that this preprocessing technique can drastically reduce the number of variables of some instances. This is good news for DPLL like algorithms, since they are exponential relatively to the number of variables in the formula. Unfortunately, it does not always lead to an improvement to the resolution time in practice.
- If the process occurs during the search, then the variables can only be replaced within the subtree. A mechanism to restore them while backtracking is needed. Our implementation currently does not have this feature, so we will focus on the preprocessing step in the rest of the article.

Let us sketch a first preprocessing technique based on UPL. We use the same lookahead technique as SATZ, except for the third step, noted (1), when $\bot$ does not appear in L_1 or L_2. We first try to detect implied literals using the Proposition 3. Then, we try to detect equivalent literals using Proposition 4. If some equivalent literals $u \leftrightarrow x$ are detected, then we replace the literal u by x and $\neg u$ by $\neg x$ in KB. In both case, we remove from V a variable. The process is repeated until no new equivalent or implied literal can be found.

This technique is implemented in an instance compressor in Java under the name CompressLite [5] . It uses the $UPL_1(KB,l)$ approximation as in the original SATZ. The set V of variables contains all the variables of the instances (-cm parameter). The compressor can be asked to use only variables that occur in binary clauses (-cl parameter). Here is the result obtained using this preprocessing technique to compress Miters instances [6] .

The result is quite surprising. Most of the instances are drastically reduced: up to 48 percent of the variables are removed in the c1908 instances. We recall that these instances are coming from equivalence checking, so it seems

[5] The compresser is part of the tool ADS. ADS is available at http://cafe.newcastle.edu.au/daniel/ADS/

[6] Available at ftp://algos.inesc.pt/pub/benchmarks/cnf/equiv-checking/MITERS.tgz

Algorithm 2: COMPRESS(KB, V)

```
Data: a set of clauses KB ;
      a set of variables V ;
Result: a compressed version of KB preserving consistency.
begin
    while no more changes do
        for each x ∈ V do
            % propagate() returns the set of ;
            % literals fixed during the process. ;
            L1 ← propagate(KB ∧ x) ;
            L2 ← propagate(KB ∧ ¬x) ;
            if ⊥ ∈ L1 and ⊥ ∈ L2 then
                return "KB is not satisfiable";
            if ⊥ ∈ L1 then
                x ← 0;
            else
                if ⊥ ∈ L2 then
                    x ← 1;
                else
                    % neither L1 nor L2 contain ⊥ ;
                    for each y ∈ L1 ∩ L2 do
                        y ← 1;
                        remove y from V ;
                    for each y ∈ L1 ∩ ¬L2 do
                        replace y by x in KB ;
                        remove y from V;
    return KB
end
```

reasonable that they contain some equivalent literals. One exception is the c6288 instances. We do not know why. (It is worth noting that these two instances are the only ones that zChaff [9] cannot solve in this benchmark set). Interestingly, a new implementation of CompressLite using $UPL_2(KB, l)$ approximation, available in JSAT [7] under the name RemoveEQ, solves these two instances within 100s each.

The compressed instances are still difficult for SATZ. To the best of our knowledge, only GRASP, using the recursive learning strategy described in [23] can solve all the original instances. The results of state-of-the-art solvers on them

[7] http://cafe.newcastle.edu.au/daniel/JSAT/

instance	N	K	Nc	Kc	diff	%
C432-s	392	1128	298	942	94	23.98
C432	389	1115	294	917	95	24.42
C499-s	606	1878	430	1557	176	29.04
C499	606	1870	430	1549	176	29.04
C1355-s	1294	3670	1086	3221	208	16.07
C1355	1294	3662	1086	3213	208	16.07
C1908-s	1919	5108	991	3226	928	48.36
C1908	1917	5096	986	3208	931	48.57
C1908_bug	1919	5100	986	3208	933	48.62
C2670-s	2864	7436	1687	4780	1177	41.10
C2670	2627	6756	1513	4240	1114	42.41
C2670_bug	2632	6696	1440	4023	1192	45.29
C3540-s	3499	9526	2017	6507	1482	42.35
C3450	3450	9326	1958	6296	1492	43.25
C5315-s	5408	15110	3338	10468	2061	38.11
C5315	5399	15024	3338	10468	2061	38.17
C5315_bug	5396	15004	3335	10448	2061	38.19
C6288-s	5025	14882	4860	14376	165	3.28
C6288	5008	14813	4843	14375	165	3.29
C7752-s	7766	20812	4706	14093	3060	39.40
C7752	7651	20423	4588	13706	3063	40.03
C7752_bug	7558	20109	4520	13442	3038	40.20
C880-s	957	2590	712	1968	245	25.60
C880	957	2590	712	1968	245	25.60

Fig. 1. Detection of equivalent literals in Miters instances

can be found on Laurent Simon SAT-Ex site [18, instances "Joao"]. On the same site, one can note that HeerHugo and the recent zChaff perform very well on these instances (23 out of 25 instances solved).

The following instances come from the "Planning As Satisfiability" framework. We compare the power of compression of our preprocessing step against Ronen Brafman's 2-Simplify simplifier [24]. This simplifier principle is similar

to ours (use of deduction rules to fix the truth value of variables and removal of equivalent literals) but based on the information provided by an implication graph instead of UPL. One of the key idea behind this simplifier is to use a linear-time algorithm [25] to compute equivalent literals (strongly connected components in the implication graph).

			CompressLite		2-Simplify	
Instance	Var	Clauses	rem.	time(s)	rem.	time(s)
log-dir.a	828	6718	425	42	152	0.02
log-dir.b	843	7301	421	48	152	0.02
log-dir.c	1141	10719	542	86	186	0.04
log.d	4713	21991	799	2779	753	2.4
log-gp.a	1782	20895	644	972	148	0.19
log-gp.b	2069	29508	671	7473	169	0.31
log-gp.c	2809	48920	795	14041	191	0.52
log-un.a	1415	14346	750	1179	160	0.13
log-un.b	1729	21943	615	3621	161	0.21
log-un.c	2353	37121	729	11249	179	0.37
bw-dir.a	459	4675	459	55	173	0.08
bw-dir.b	1087	13772	477	643	351	0.2
bw-dir.c	3016	50457	1008	5966	824	0.96

Fig. 2. CompressLite vs 2-Simplify on planning instances

In all the cases, CompressLite removes more variables than 2-simplifier. The big problem is the running time of our implementation. We do not use a special purpose implementation for simplifying the instance, but our own implementation of a DPLL algorithm (in Java!). A lot of things are computed but not required in this context (some heuristics, backtracking mechanism, etc.). We are currently re-implementing our compresser to improve its running time.

5 More biconditionals with UPL

We have seen how we can use the unit propagation lookahead for *semantically* retrieving some equivalent literals. Another way to obtain equivalent literals is to reason with biconditional formulae, using EqSATZ rule R5 for instance.

We will now focus on a way to semantically find some of these biconditionals using *double* unit propagation lookahead. Let us note $DUPL(KB, l_1, l_2)$ the set of literals satisfied by the $propagate(KB \wedge l_1 \wedge l_2)$ process such that $\bot \in DUPL(KB, l_1, l_2)$ iff the process falsifies a clause. Obviously, we have $l_1 \in DUPL(KB, l_1, l_2)$ and $l_2 \in DUPL(KB, l_1, l_2)$.

Let us now highlight the semantics of this set of literals.

Proposition 5 $u \in DUPL(KB, l_1, l_2) \Rightarrow KB \models l_1\ l_2 \rightarrow u$

In the same way as with simple propagation, we can obtain *some* implied literals.

Proposition 6 $\bot \in DUPL(KB, l_1, l_2) \cap DUPL(KB, \neg l_1, l_2) \Rightarrow KB \models \neg l_2$
$\bot \in DUPL(KB, l_1, l_2) \cap DUPL(KB, l_1, \neg l_2) \Rightarrow KB \models \neg l_1$
$u \in DUPL(KB, l_1, l_2) \cap DUPL(KB, l_1, \neg l_2) \cap DUPL(KB, \neg l_1, l_2) \cap DUPL(LB, \neg l_1, \neg l_2) \Rightarrow KB \models u$

We can also retrieve some equivalent literals.

Proposition 7 $\bot \in DUPL(KB, l_1, l_2) \cap DUPL(KB, \neg l_1, \neg l_2) \Rightarrow KB \models l_1 \leftrightarrow \neg l_2$

Let us now describe how we can recover some biconditionals.

Proposition 8 $u \in DUPL(KB, l_1, l_2) \cap DUPL(KB, \neg l_1, \neg l_2)$ *and* $\neg u \in DUPL(KB, l_1, \neg l_2) \cap DUPL(KB, \neg l_1, l_2) \Rightarrow KB \models l_1 \leftrightarrow l_2 \leftrightarrow u$

This can be viewed as a semantical alternative to EqSATZ $R7$ rule. Now a more powerful property which is very close to the preceding one:

Proposition 9 $u, v \in DUPL(KB, l_1, l_2) \cap DUPL(KB, \neg l_1, \neg l_2)$ *and* $\neg u, \neg v \in DUPL(KB, l_1, \neg l_2) \cap DUPL(KB, \neg l_1, l_2) \Rightarrow KB \models u \leftrightarrow v$

It is exactly the rule R5 of EqSATZ. It simply means that we can recover non-trivial equivalent literals using double unit propagation without explicitly using a deduction rule process (as in EqSATZ or Stålmarck/HeerHugo). The following rule is equivalent to the rule R6:

Proposition 10 $u \in DUPL(KB, l_1, l_2) \cap DUPL(KB, \neg l_1, l_2)$ *and* $\neg u \in DUPL(KB, l_1, \neg l_2) \cap DUPL(KB, \neg l_1, \neg l_2) \Rightarrow KB \models l_2 \leftrightarrow u$

Now let us expand an algorithm so that it can handle equivalent literals:

- first fix the truth value of implied literals and apply the proposition 4 to remove equivalent literals using simple unit propagation (using the original

SATZ *propagate*() process).
- then fix the truth value of implied literals and apply the propositions 9 and 10 to remove equivalent literals found using double unit propagation (still using the original SATZ *propagate*() process).
- use a DPLL-like algorithm to solve the rest of the problem.

This algorithm is implemented as two different variants: DavPutSt uses all the variables for V for the second step, whereas DavPutSt2 uses only the variables that appear in clauses with the length of at least 3. This choice is consistent with current implementations of DPLL since they usually maintain the number of positive and negative occurrences in binary clauses for each variable. The selected variables are those with both counters to zero. The first step is in both cases applied to all the variables. DavPutSt can be viewed as the upper bound of the semantical power of the approach since we use each time all the variables. But it is really slow. DavPutSt2 is a tradeoff between what can be deduced and the time spent. We report here the result of DavPutSt2 on the BMC barrel instances below. (No results are provided for the barrel9 instance since the instance is too large for the solver).

Instance	var	clauses	eq	r7	r6	r5	bic.	time
barrel2	50	159	10	0	0	0	0	0.1
barrel3	275	942	44	0	0	60	4	3.2
barrel4	578	2035	76	0	0	171	9	7.4
barrel5	1407	5383	170	0	0	598	26	39
barrel6	2306	8931	238	0	0	1080	40	169
barrel7	3523	13765	321	0	0	1767	57	767
barrel8	5106	20083	413	0	0	2695	77	4355
barrel9	8903	36606	-	-	-	-	-	-

The R5 rule in conjunction with the removal of equivalent literals with a single unit propagation (column "eq") removes more than half of the variables each time. R6 and R7 seem useless: on other instances (eg. dubois), R6 works very well, but not R5. Consequently, it can be seen that both R5 and R6 are efficient to detect equivalent literals. In our implementation, when a new biconditional is found using proposition 8, we try to add the associated ternary clauses to the base. R7 denotes the number of ternary clauses added. These results mean that we do not create new biconditionals, we just retrieve some of them. The "bic." column denotes how many biconditionals have been found.

The following table reflects the runtime of the best DPLL-like SAT solvers

for the barrel instances found on Laurent Simon SAT-Ex site. Since the comparison between our algorithm implemented in Java and these state-of-the-art solvers in C is unfair, we show the result of the preprocessing step of DavPutSt2 [8] followed by a call to SATZ to solve the remaining problem. The duration for each step and the total time consumed are given.

Instance	HH	EqSATZ	SATZ	Preprocessing+SATZ		
				Prep.	SATZ	Total
barrel2	0.06	0.04	0.02	0.91	-	0.91
barrel3	0.15	0.08	0.14	2.42	1.60	4.02
barrel4	0.42	0.21	0.38	7.30	6.52	13.8
barrel5	2.13	0.67	380	6.05	20.2	26.2
barrel6	5.48	1.39	2788	11.1	52.6	63.7
barrel7	12.64	1.82	53	19.6	112	132
barrel8	26.46	2.81	4.43	27.4	250	277
barrel9	-	6.13	>10000	57.3	475	532

The instance barrel2 is solved during the preprocessing step. The runtime of both HeerHugo (HH) and EqSATZ come from SAT-Ex site. The runtime of SATZ on SAT-Ex is slightly better than on our computer. If we compare the result of SATZ with and without preprocessing, we can see that the preprocessing step seems to extract enough information from the base to reduce the search space to a tractable one. Nevertheless, this technique cannot compete with EqSATZ (even though the deduction rules seem identical) or HeerHugo.

The figure 3 compares the power of compression of our preprocessing step against Joao Marquez Silva simplifier [26].

The results were obtained using Sun java 2 SE with hotspot server (jdk 1.3 with -server command line option, Perl version 5.6.0, both on a Linux box Mandrake 7.2, PIII 500, 128 Mo RAM Time for simplify obtained using the unix time command. Time for Compress displayed by the tool.

Joao Marques Silva simplifier works very well. One can think that the compressed instances are the expected "kernel" ones, on which SATZ performs very well. The only drawback is the running time of the simplifier (in perl). Our simplifier followed by SATZ can solve the instances quicker than with

[8] Compress, invoked using the -c parameter in ADS.

Instance	Compress (*java -server ADS -c*)					Simplify				
	Var	Clauses	Time	SATZ	Total	Var	Clauses	Time	SATZ	Total
barrel3	98	573	3.6	1.0	4.6	48	183	7.6	0.0	7.6
barrel4	211	1419	5.3	4.2	9.5	102	423	21.0	0.0	21.0
barrel5	399	4043	5.9	13.3	19.2	310	1295	52.0	0.1	52.1
barrel6	650	7053	8.3	35.2	43.5	519	2209	106.6	0.4	107.0
barrel7	986	11259	15.1	75.4	90.5	805	3465	212.3	0.1	212.4
barrel8	1419	16859	25.9	170.4	196.3	1180	5213	214.1	0.8	214.9
barrel9	2006	31323	58.5	330.7	389.2	1701	8887	796.2	0.5	796.7

Fig. 3. Compress vs Simplify on barrel instances

simplify+SATZ. Nevertheless, this result seems to consolidate the assumption underlying the work on random 3-SAT formulae: it does exist some ways to take into account the structure of "hard" instances as a preprocessing step for classical SAT solvers. (We will see in the next section that the preprocessing step can even be sufficient to solve the barrel instances).

5.1 Other approach

This work on the double unit propagation for solving the barrel and the dubois instances resulted from the failure to detect equivalent literals in these instances using the original SATZ *propagate*() process in CompressLite. It simply meant that the *propagate*() process did not return enough literals to allow us to detect the equivalent literals. We focussed on the double unit propagation scheme after discovering the Stålmarck method. However, a simpler approach is to improve the approximation of $IMPL(KB \wedge l)$. The new version of CompressLite available in JSAT is based on this idea: it is using SATZ213 *propagate*() process instead of the one from SATZ. The resulting preprocessing algorithm, RemoveEQ, can solve all 13 dubois instances within 2 second. The running time of RemoveEQ (from JSAT 0.1.27) and the number of equivalent literals found on the barrel instances are reported in the table below.

Now all the instances are solved during the preprocessing step. The running time of RemoveEQ is worst than Compress but it can be improved: unit subsumption could be disabled during the second level of UPL for instance.

Instance	var	clauses	eq	time (s)
barrel2	50	159	0	0.2
barrel3	275	942	117	1.2
barrel4	578	2035	368	3.6
barrel5	1407	5383	978	22
barrel6	2306	8931	1666	65
barrel7	3523	13765	2616	196
barrel8	5106	20083	3868	441
barrel9	8903	36606	7042	1585

Fig. 4. RemoveEQ (JSAT 0.1.27) on barrel instances

5.2 The problem of pret instances

We have compared our approach to EqSATZ, and emphased where it is a semantical approach to EqSATZ. This relationship does not always hold since the two approaches do not always work in a similar way. They are orthogonal. We have seen that EqSATZ is unable to detect in $KB = l_1 \rightarrow l_2, l_2 \rightarrow l_3, l_3 \rightarrow l_1$ that $l_1 \leftrightarrow l_3$. One can check that $UPL(KB, l_1) = \{l_2, l_3\}$ and $UPL(KB, \neg l_1) = \{\neg l_2, \neg l_3\}$ so using proposition 3 we find $l_1 \leftrightarrow l_3$. Furthermore, the Miters instances c6288 that RemoveEq can solve in 100s cannot be solved by EqSATZ in 10000s! On the other hand, Dimacs pret instances are solved in linear time by EqSATZ, but neither Compress nor RemoveEq approach can extract biconditional information from these sets of clauses. The explanation has been pointed out by Chu-Mi Li in a personal communication. Pret instances contains biconditionals of the form: $l_1 \leftrightarrow l_2 \leftrightarrow l_3, l_2 \leftrightarrow l_4 \leftrightarrow l_5, l_3 \leftrightarrow l_4 \leftrightarrow l_6$,

From these biconditionals, EqSATZ can produce the new biconditional $l_1 \leftrightarrow l_5 \leftrightarrow l_6$, and add it to the base. Indeed, satisfying l_1 produces $l_2 \leftrightarrow l_3$ so l_3 can be replaced in the last biconditional by l_2. We can then apply R5 between the two last biconditionals, which leads to $l_5 \leftrightarrow l_6$. So we have $l_1 \rightarrow (l_5 \leftrightarrow l_6)$ (1). Similarly, falsifying l_1 produces $\neg(l_2 \leftrightarrow l3)$ so l_3 can be replaced in the last biconditional by $\neg l_2$. From the first rule, this conditional is equivalent to $l_2 \leftrightarrow l_4 \leftrightarrow \neg l_6$. Applying R5 we obtain $l_5 \leftrightarrow \neg l_6$. So we also have $\neg l_1 \rightarrow \neg(l_5 \leftrightarrow l_6)$ (2). Using the R7 rule between (1) and (2) give us the expected biconditional.

What happens with the DavPutSt approach? The new biconditional should

be retrieved using a double unit propagation on l_1 and l_5 variables.

Suppose we satisfy both l_1 and l_5. This gives the base $KB_1 = \{l_2 \leftrightarrow l_3, l_2 \leftrightarrow l_4, l_3 \leftrightarrow l_4 \leftrightarrow l_6\}$. It is easy to see that $KB_1 \models l_6$. So our idea of recovering the biconditional semantically should work here. Unfortunately, this is not obvious since l_6 is not satisfied during the double unit propagation. We need to find a way to capture more implied literals after this double unit propagation. One can note that this information can be retrieved using a single unit propagation lookahead, as seen in Proposition 3! This is really convenient. After a double unit propagation, we may be forced to add $|V| - 2$ single unit propagation lookahead. Therefore for this unit propagation we will focus on variables appearing in binary clauses only. Note that does not mean that the implied literal will be one of them (look at the example: l_6 does not appear in a binary clause but is found as an implied literal of KB_1 when l_2 is one of the other literals used for a single unit propagation lookahead).

Let $FDUPL(KB, l_1, l_2)$ be the set of:

- literals satisfied by double unit propagation of l_1 and l_2
- implied literals found by simple unit propagation lookahead on variables occurring in binary clauses when the double unit propagation is stopped. DavPutSt3 is a modified version of DavPutSt which uses $FDUPL$ instead of $DUPL$.

This algorithm can handle pret instances.

Instance	var	clauses	SATZ213 (s)	DavPutSt3 (s)
pret60_25	60	160	100.84	13.16
pret60_40	60	160	101.58	3.02
pret60_60	60	160	102.17	2.80
pret60_75	60	160	100.35	2.71

For the four pret150 instances, SATZ is unable to solve them. DavPutSt3 is also slow on these instances. Using the preprocessing step of DavPutSt3 [9] (which takes 35 s for each instance) adds 188 clauses to each instance. SATZ can then prove their inconsistency in less then 50 ms each!

If the pret instances are hard for SATZ, they are easy for "learning" algorithms, that can learn new clauses during the search, often based on conflict

[9] CompressHeavy, invoked by -ch in ADS

analysis [6–8]. Using an algorithm such as DavPutSt3 produces new clauses before the search, leading to the same result. If usually it is impossible to semantically describe the clauses learned, thus explaining what kind of clauses are required to help solving the instances, our little experiment shows that expliciting biconditionals is sufficient to help solving pret instances.

6 Comparison with previous works

[26] has concurrently developed a simplifier based on the use of implication graph instead of unit propagation lookahead. This work complete the one on 2-simplify [24] since a pattern matching approach is used to simplify the formulae and some deduction rules are used to discover more binary clauses. (Hence completing the original implication graph). From our first experiments on the barrel instances, this technique looks very powerful.

6.1 AVAL

[27] introduced another way to detect some implied literals in the AVAL SAT solver. They use the basic proposition 2 to determine the implied literals, but take into account the following property to avoid useless literal testing:

Proposition 11 (Proposition 2 in [27]) *if* $u \in UPL(KB,l)$, $u \neq \bot$ *and* $\bot \notin UPL(KB,l)$, *then* $\bot \notin UPL(KB,u)$.

However, they do not obtain the same good behavior on the two instances 3bitadd_31 and 3bitadd_32 mentioned before. We think that the key of SATZ215 success on these instances is in the use of $UPL_2(KB,l)$ to approximate $IMPL(KB \wedge l)$. ([27] are using $UPL_1(KB,l)$). Their result should not be used in conjunction with our technique to detect equivalent literals because we use the information returned by $UPL(KB,l)$ to detect equivalent literals even if it does not contain $\bot$.

6.2 EqSAT

We already mentioned several similarities and differences between our approach and EqSATZ. Here is a summary: (1) both use deduction rules, but while EqSATZ uses a specific implementation to fire them, DavPutSt just uses the usual unit propagation lookahead. (2) EqSATZ works syntactically, DavPutSt semantically. (3) EqSATZ (like HeerHugo for instance) uses deduction rules during all the search, whereas DavPutSt uses them as a preprocess-

ing step. This behavior is not related to the technique used, but the difficulty to implement it in a classical DPLL (handling equivalent literals is not easy during the search).

6.3 HeerHugo

[10] proposed a first link between the Stålmarck method and the usual DPLL/Local search techniques, concluding to use deduction rules as a preprocessing tool on structured SAT instances. The good behavior of this method on Miters instances give us some clues about the way to handle them: one can note that the compressed instances obtained after removing equivalent literals contain mainly clauses representing formulas of the form $x \leftrightarrow y_1 \wedge y_2 \wedge ... \wedge y_n$. Since HeerHugo uses deduction rules on triplets of the form $x \leftrightarrow y_1 \wedge y_2$, one can expect that the good behavior of HeerHugo on Miters instances is due to these particular deduction rules. Since neither EqSATZ nor DavPutSt have such deduction rules, it may explain why they do not perform well on these instances.

6.4 Intelligent backtracking

If the best results on random 3CNF can be attributed to SATZ [4], on structured problems, SATO [6], RELSAT [7], GRASP [8] seems to be more appropriate. The common approach of these three solvers is their use of an "intelligent backtracking process" that let them avoid repeating mistakes. Basically, their knowledge is recorded by some added clauses during the search, computed from a conflict analysis.

The main difference between the two approaches (deduction rules vs intelligent backtracking) is that deduction rules can be successfully applied if the instances have a well known structure (which is the case for Stålmarck's symbolic model checking problems, or for EqSATZ and the parity32 problem) and result in highly efficient algorithms for these particular instances. The aim of intelligent backtracking is to learn from the instance where nothing is known about them. In this sense, the approaches are complementary.

7 Conclusion

We have shown that unit propagation lookahead can be used to retrieve semantical information from SAT instances. We have verified the efficiency of

such a technique on different benchmarks. Since our technique relies only on unit propagation lookahead, a classical feature of some highly optimized DPLL SAT solvers, it can be easily added to any of them, or implemented using existing tuned data structures and algorithms. For instance, Chu-Min Li implemented the detection of implied literals in the last version of SATZ215, the only complete SAT solver able to solve 3bitadd_31 instance we are aware of.

This semantical information can be also used to simplify some structured instances, especially the ones from symbolic model checking (SMC), containing equivalent literals. We gave some experimental results confirming the efficiency of this simplification as preprocessing of a traditional SAT solver (SATZ) on some SMC instances.

This preprocessing technique is a first step to the simplification of SMC instances into a "kernel" formula. We have underlined that some compressed instances still have a specific structure (for instance Miters compressed instances). We are currently working on a way to take it into account.

Finally, the question of the use of a CNF input for classical solvers can be raised. Stålmarck/Heerhugo solvers take advantage of the non imposed form of the formula to initiate their reasoning about equivalent literals for instance, that must be retrieved syntactically (EqSATZ) or semantically (DavPutSt) when using the CNF input. It makes sense to work with CNF if you suppose that the translation of the problem to CNF is made by another tool. But when the translation is closely related to a deduction process, it is clearly difficult to see when the translation ends. In one sense, Stålmarck and Heerhugo methods are more powerful than DPLL techniques since they handle propositional formulas, not only CNF. However the comparison with DPLL will be unfair if an unintelligent translation from the general formula into CNF is applied.

Acknowledgements

The author would like to thank Yacine Boufkhad for his insights about C-SAT and 2-simplify, Mamoun Filali-Amine for pointing out the Stålmark method, Chu-Min Li for his discussions about EqSATZ and his implementation of the implied literal propagation in SATZ, Laurent Simon for his SAT-Ex web site, Miroslav Velev for his remarks on the early version of the paper and the referees for their helpful comments. This research was funded by an ARC Large Grant at The University of Newcastle.

References

[1] M. Davis, G. Logemann, D. Loveland, A machine program for theorem proving, Communications of the ACM 5 (1962) 394–397.

[2] O. Dubois, P. André, Y. Boufkhad, J. Carlier, SAT versus UNSAT, in: Johnson and Trick [28], pp. 415–436.

[3] J. W. Freeman, Improvements to propositional satisfiability search algorithms, Ph.D. thesis, Departement of computer and Information science, University of Pennsylvania, Philadelphia (1995).

[4] C.-M. Li, Anbulagan, Heuristics based on unit propagation for satisfiability problems, in: Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97), Nagoya (JAPAN), 1997, pp. 366–371.
URL `http://www.laria.u-picardie.fr/ cli/Publis/up.ps`

[5] C.-M. Li, A constrained based approach to narrow search trees for satisfiability, Information processing letters 71 (1999) 75–80.
URL `http://www.laria.u-picardie.fr/ cli/Publis/ipl0199.ps`

[6] H. Zhang, SATO: an efficient propositional prover, in: Proceedings of the International Conference on Automated Deduction (CADE'97), volume 1249 of LNAI, 1997, pp. 272–275.

[7] R. J. J. Bayardo, R. C. Schrag, Using CSP Look-Back Techniques to Solve Real-World SAT Instances, in: Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97), Providence, Rhode Island, 1997, pp. 203–208.
URL `http://www.almaden.ibm.com/cs/people/bayardo/ps/aaai97.ps.Z`

[8] J. P. Marques-Silva, K. A. Sakallah, GRASP - A New Search Algorithm for Satisfiability, in: Proceedings of IEEE/ACM International Conference on Computer-Aided Design, 1996, pp. 220–227.
URL
`http://sat.inesc.pt/ jpms/research/papers/iccad96/iccad96.ps.gz`

[9] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: Engineering an Efficient SAT Solver, in: Proceedings of the 38th Design Automation Conference (DAC'01), 2001.

[10] J. F. Groote, J. P. Warners, The propositional formula checker HeerHugo, in: Gent et al. [29], pp. 261–281.

[11] C.-M. Li, Integrating Equivalency reasoning into Davis-Putnam procedure, in: Proceedings of AAAI'2000, Austin, Texas, USA, 2000, pp. 291–296.
URL `http://www.laria.u-picardie.fr/ cli/Publis/aaai2000.ps`

[12] B. Selman, H. A. Kautz, D. A. McAllester, Ten challenges in propositional reasoning and search, in: Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97), 1997, pp. 50–54.

URL
http://www.cs.cornell.edu/home/selman/papers-ftp/97.ijcai.challenge.ps

[13] J. P. Warners, H. Van-Maaren, A two phase algorithm for solving a class of hard satisfiability problems, Operations Research letters 23 (1998) 81–88.

[14] G. Stålmarck, A system for determining propositional logic theorems by applying values and rules to triplets that are generated from a formula, Tech. rep., European Patent N 0403 454 (1995), US Patent N 5 276 897, Swedish Patent N 467 076 (1989) (1989).

[15] J. Harrison, Stålmarck's Method as a HOL Derived Rule, in: J. von Wright, J. Grundy, J. Harrison (Eds.), Proceedings of the 9th international Conference on Theorem Proving in Higher Order Logics (TPHOLs'96), volume 1125 of Lecture Note in Computer Science, Finland, 1996, pp. 221–234.

[16] A. Biere, A. Cimatti, E. M. Clarke, Y. Zhu, Symbolic Model Checking without BDDs, in: Proceedings of Tools and Algorithms for the Analysis and Construction of Systems (TACAS'99), number 1579 in LNCS, 1999.

[17] P. A. Abdulla, P. Bjesse, N. Eén, Symbolic Reachability Analysis Based on SAT-Solvers, in: Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2000), 2000.
URL http://www.docs.uu.se/ parosh/publications/smc.ps

[18] L. Simon, P. Chatalic, SATEx: a Web-based Framework for SAT Experimentation, in: Proceedings of the Workshop on Theory and Applications of Satisfiability Testing (SAT2001) [30], http://www.lri.fr/ simon/satex/satex.php3.

[19] M. Davis, H. Putnam, A computing procedure for quantification theory, Journal of the ACM 7 (1960) 201–215.

[20] B. Jurkowiak, C.-M. Li, G. Utard, Parallelizing SATZ Using Dynamic Workload Balancing, in: Proceedings of the Workshop on Theory and Applications of Satisfiability Testing (SAT2001) [30], to appear.

[21] C. P. Gomes, B. Selman, N. Crato, H. Kautz, Heavy-Taily Phenomena in Satisfiability, in: Gent et al. [29], pp. 15–41.

[22] B. Selman, H. Kautz, B. Cohen, Local search strategies for satisfiability testing, in: Johnson and Trick [28], pp. 521–532.
URL
http://www.cs.cornell.edu/home/selman/papers-ftp/96.dimacs.walksat.ps

[23] J. P. Marques-Silva, T. Glass, Combinational Equivalence Checking Using Satisfiability and Recursive Learning, in: Proceedings of the IEEE/ACM Design, Automation and Test in Europe Conference (DATE), 1999.
URL http://sat.inesc.pt/ jpms/research/papers/date99/cec.ps.gz

[24] R. I. Brafman, A simplifier for propositional formulas with many binary clauses, in: Proceedings of the Seventeenth International Joint Conference on Artificial

Intelligence (IJCAI'01), Seattle, Washington, USA, 2001, to appear.
URL http://www.cs.bgu.ac.il/ brafman/ijcai01-binsat.ps

[25] B. Aspvall, M. Plass, R. Tarjan, A linear-time algorithm for testing the truth of certain quantified boolean formulas, Information Processing Letters 8 (1979) 121–123.

[26] J. P. Marques-Silva, Algebraic Simplification Techniques for Propositional Satisfiability, in: Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP'2000), 2000.
URL
http://sat.inesc.pt/ jpms/research/papers/cp2000/cp2000-algeb.ps.gz

[27] G. Audemard, B. Benhamou, P. Siegel, AVAL: An enumerative method for SAT, in: Proceedings of the First international conference on Computational Logic (CL'00), Londres, 2000, pp. 373–383.

[28] D. Johnson, M. Trick (Eds.), Second DIMACS implementation challenge : cliques, coloring and satisfiability, Vol. 26 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, 1996.
URL http://dimacs.rutgers.edu/Challenges/

[29] I. Gent, H. van Maaren, T. Walsh (Eds.), SAT20000: Highlights of Satisfiability Research in the year 2000, Frontiers in Artificial Intelligence and Applications, Kluwer Academic, 2000.

[30] Proceedings of the Workshop on Theory and Applications of Satisfiability Testing (SAT2001), to appear.
URL http://www.cs.washington.edu/homes/kautz/sat2001/

Annals of Mathematics and Artificial Intelligence (2005) 43: 307–342

The SAT2002 Competition

Laurent Simon [a], Daniel Le Berre [b,*] and Edward A. Hirsch [c,**]

[a] *LRI, U.M.R. CNRS 8623, Université Paris-Sud, 91405 Orsay Cedex, France*
E-mail: simon@lri.fr
[b] *CRIL, F.R.E. CNRS 2499, Faculté Jean Perrin, Université d'Artois, Rue Jean Souvraz SP 18, 62300 Lens Cedex, France*
E-mail: leberre@cril.univ-artois.fr
[c] *Steklov Institute of Mathematics at St. Petersburg, 27 Fontanka, 191023 St. Petersburg, Russia*
http://logic.pdmi.ras.ru/~hirsch/

SAT Competition 2002 held in March–May 2002 in conjunction with SAT 2002 (the Fifth International Symposium on the Theory and Applications of Satisfiability Testing). About 30 solvers and 2300 benchmarks took part in the competition, which required more than 2 CPU years to complete the evaluation. In this report, we give the results of the competition, try to interpret them, and give suggestions for future competitions.

Keywords: Boolean satisfiability (SAT), empirical evaluation

AMS subject classification: 68W20, 03B05

1. Introduction

The SAT2002 solver competition, involving more than 30 solvers and 2300 benchmarks, took place in Cincinnati a decade after the first SAT solver competition held in Paderborn in 1991/1992 [7]. Two other SAT competitions were organized since that date: the Second DIMACS Challenge, held in 1992/1993 [29], and the *Beijing* competition, in 1996.[1] In the last few years, the need for such a competition was more and more obvious, reflecting the recent and important progress in the field. A lot of papers have been published concerning "heuristics" algorithms for the NP-complete satisfiability problem [11], and even software exists that people use on real-world applications. Many techniques are currently available, and it is difficult to compare them. This comparison can hardly be only on the theoretical level, because it often does not tell anything from a practical viewpoint. A competition can lead to some empirical evaluation of current algorithms (as good as possible) and thus can be viewed as a snapshot of the state-of-the-art solvers at a given time. The data collected during this competition

* This work has been supported in part by the IUT de Lens, the Université d'Artois and by the "Région Nord/Pas-de-Calais" under the TACT-TIC project.
** Supported in part by RFBR grant No. 02-01-00089 and by grant No. 1 of the 6th RAS contest-expertise of young scientists projects (1999).

[1] Benchmarks available at `http://www.cirl.uoregon.edu/crawford/beijing/`.

will probably help to identify classes of hard instances, solvers limitations and allow to give appropriate answers in the next few years. Moreover, we think that the idea of such a competition takes place in a more general idea of empirical evaluation of algorithms. In a number of computer science fields, we need more and more knowledge about the behavior of the algorithms we design and about the characteristics of benchmarks. This competition is a step in a better – and crucial – empirical knowledge of SAT algorithms and benchmarks [26,27]. The aim of this paper is to report what organizers learned during this competition (about solvers, benchmarks and the competition itself), and to publish enough data to allow the reader to make is own opinion about the results.

As it was mentioned, in the first half of the last decade, some competitions were organized to compare solvers. However, one major – and recent – step in that area was the creation of the SAT-Ex web site [54], an online collection of results concerning various SAT solvers on some classical benchmarks. Among all the advantages of this kind of publication, SAT-Ex allows to check every solver output, generate dynamically synthesis and add constantly new solvers and benchmarks.

More and more researchers would like to see how their solver compares with other solvers on the current benchmark set. This is not really a problem because only a few CPU days are needed to update SAT-Ex database with a new solver: usually, the new solver will outperform old ones for some series of benchmarks. A problem arises with the introduction of new benchmarks: the benchmarks have to be tested on each solver, and they are likely to give them a hard time. Since all the results available on SAT-Ex were obtained on the same computer (the only way to provide a fair comparison based on the CPU time) it will take ages before seeing results on new benchmarks. To solve that problem, there are several solutions:

– working with a cluster of computers instead of a single one. Laurent Simon is currently working that out, preparing a new (and updated) release of SAT-Ex;
– using SAT-Ex system as a convenient way to take a picture of some SAT solvers efficiency on a given set of benchmarks.

This last point is one of our major technical choices: using SAT-Ex architecture for the competition, providing a *SAT-Ex style* online publication. In order to enlight some of the other choices we made during the competition, let us first recall some SAT statements. Currently, approaches to solve SAT can be divided into two categories: complete and incomplete ones. A complete solver can prove satisfiability as well as unsatisfiability of a boolean formula. On the contrary, an incomplete solver can only prove that a formula is satisfiable, usually by providing a model of the formula (a *certificate* of satisfiability).

Most complete solvers descend from the backtrack search version of the initial resolution-based Davis and Putnam algorithm [14,15], often referred to as DPLL algorithm. It can be viewed as an enumeration of all the truth assignments of a formula, hence if no model is found, the formula is unsatisfiable. Last decade has resulted in many im-

provements of that algorithm in various aspects both in theory (exponential worst-case upper bounds; the most recent are [13,24]) and in practice (heuristics, especially for k-SAT formulas: [16,17,20,39], data structures [66] and local processing). Forward local processing is used to reduce the search space in collaboration with heuristics (unit propagation lookahead [37], binary clause reasoning [62], equivalence reasoning [38], etc.). Backward local processing tries to correct mistakes made by the heuristics: learning, intelligent backtracking, backjumping, etc. [5,42,65]. Also randomization is used to correct wrong heuristics choices: random ties breaking and rapid restart strategies have been shown successful for solving some structured instances (planning [22], Bounded Model Checking (BMC) [4]).

Another use of randomization is the design of incomplete solvers, where randomness is inherent. There was an increased interest in their experimental study after the papers on greedy algorithms and later WalkSAT [51,52]. Encouraging average-case time complexity results are known for this type of algorithms (see, e.g., [33]). In theory, incomplete solvers could perform (much) better than complete ones just because they belong to a wider computational model. Indeed, there are benchmarks (especially coming from various random generators) on which incomplete solvers perform much better. Worst-case time bounds are also better for incomplete algorithms [50].

A revolution? Furthermore, a completely new approach to solve SAT appeared last year, resulting from the existence of huge SAT instances encoding some specific problems, such as planning [18,31,32] or more recently Bounded Model Checking [1,6,63]. While most of the underlying techniques are not new (DPLL with intelligent backtracking, learning and a rapid restart strategy), one of the main idea was to focus on a carefully engineered solver: when dealing with a huge instance, choosing the right algorithm or data structure is as important as choosing the right heuristics to reduce the search space. Chaff [43,67] was designed from the begining to handle large formulas (more than 100000 variables) from a very specific area (mostly Bounded Model Checking) using "lazy" data structures. Since there is no heuristics shown to be efficient on EDA instances, Chaff also integrated a new form of learning, taking advantage of the overall lazy data structures used: Chaff makes mistakes, but learns quickly! Chaff outperformed existing SAT solvers on Bounded Model Checking instances, and a large set of "structured" (as opposed to random) instances [54]. It looked interesting to establish a new overall picture of SAT kingdom after that "revolution".

Such a competition allows to obtain both new solvers and new benchmarks. It was proved to stimulate the community (more than just by providing *awards* to it). Many breakthroughs in the last years were due to empirical evaluation of algorithms, leading to a better knowledge of algorithms behaviors and of benchmarks hardness. This knowledge allow to propose (and test) new answers. Such a competition can thus be viewed as one of the fundamental part of the research around the topic.

2. Rules and submissions

In order to ensure fairness, all the rules concerning the competition were available a few months before the competition on the web,[2] after public discussions on a SATLive! forum.

The solver and benchmark submission processes were running in parallel. The submitters did not know who else submitted solvers or benchmarks, and what was submitted. All submissions were received and processed by Laurent Simon who kept them in secret from everybody including the two other organizers. After that, he alone (+ system administrators) was running the competition computers. That allowed Edward A. Hirsch to participate in the competition despite of being among the organizers.

2.1. The rules

The general idea was to award the most "generic" solver, i.e. the one that is able to solve the widest range of problems. However, it looked like a nonsense to compare a solver tailored for 3-SAT random instances and one tailored for Electronic Design Automation (EDA) instances, and the same remark applies for complete and incomplete solvers. So we divided the space of SAT experiments into 6 categories: industrial, handmade, random benchmarks for either complete (which could solve both satisfiable and unsatisfiable benchmarks) or all solvers (in the latter case, only satisfiable and "unknown" benchmarks were used, and only satisfiable ones were counted). Submitters were asked to stamp their benchmarks with the correct category.

To rank the solvers in each category, we decided to use the notion of series: a series of benchmarks is a set of closely related benchmarks (for instance, pigeon-hole series consists of hole-2, hole-3, etc. instances). We considered that a series was solved if and only if at least one of the instances of that series was solved. Thus the idea was to award a solver solving a maximum number of series in a given category. To break ties, we decided to count the total number of instances solved. We planned to use CPU time as a last resort but we did not have to use it (note that, besides its effects with the CPU cut-off limit, pure CPU time performances do not play a crucial role in the results: two solvers have the same performance if they solve a benchmark, whatever the exact CPU time it takes). Benchmarks were grouped in series by us, authors were only allowed to submit *families* of benchmarks (a series was one or more families of benchmarks).

Furthermore, if there was a scaling factor between the instances of the series (`hole-2` $\leqslant$ `hole-3` $\leqslant$ `hole-4`, etc.) then we did not launch a solver on the biggest instances if it failed to solve any smaller. The initial idea of this "heuristic" (well-founded in practice on the *pigeon hole* example) was to save CPU time (allowing to discard quickly any weak solver). Later, it happened that this choice had an important impact on results and was not well-founded in general (we will discuss this later in the paper). The scaling information of families of benchmarks was only given by benchmarks author.

[2] `http://www.satlive.org/SATCompetition/cfs.html`.

2.1.1. Input and output formats

We asked submitters to send benchmarks in DIMACS file format.[3] One of the ideas underlying this format is that benchmarks are in CNF and are easy to read (for instance, variables are already indexed by integers). Of course, *generators* of benchmarks were allowed, assuming that authors gave clues for the *interesting* parameters to use with.

The output format (printed by solvers) was detailed in our call for solvers.[4] Briefly, the idea was to allow any solver to print any "comment" line (any information judged as "interesting" by authors) and some special lines for automated interpretation purposes. Information lines are important if one wants to understand results and to be able to interpret the huge amount of data collected during the competition. The output format allows to print the answer (SAT, UNSAT or unknown), and requests a certificate if SAT was claimed. If no answer was (syntactically) found in the output (for instances if the solver crashed or was timed out), then *unknown* was assumed.

2.1.2. Checking results and outputs: What makes a solver buggy

Let us notice a tricky consideration about *buggy* solvers. If SAT was claimed on a satisfiable instance, but the certificate was not correct, then *unknown* (and not *buggy*) was assumed as an answer. Each SAT result is thus certified, and we did not consider as *buggy* a solver that gave a wrong certificate (this can be due for instance to a CPUs exceed while printing the certificate or to a data structure problem if the certificate is displayed on a single line). As a matter of fact, we only considered as "buggy" all solvers that answered incorrectly, UNSAT on a SAT instance (previously known SAT or proved by any other solver during the competition). In addition, solvers are by essence incomplete, because of memory and CPU limitation. Thus, if a solver crashed during the competition (which can be due or not to bugs), we did not consider it as *buggy*. We only considered that its answer was "*unknown*".

Each time a *buggy* solver was found, it was tagged *hors-concours* and discarded from the awards (results were still available "unofficially").

2.1.3. Competition steps

From a practical point of view, the competition ran in several steps, going from March to May 2002. The initial step was exclusively for authors: a machine was opened over the web to allow them to compile/test their sources code in "realistic environment". After that, the competition began:

– *compliance testing*:

- *solvers*: each of them was compiled and tested on a few benchmarks to check that the solvers conformed input/output requirements of SAT-Ex framework. During that step, some bugs (in the usual sense) were detected and reported to authors. But note that *it was not the aim of that step*. Some fixed version were accepted.

[3] This was more precisely a restriction of this format, as described in our call for benchmarks (`http://www.satlive.org/SATCompetition/cfb.html`).

[4] `http://www.satlive.org/SATCompetition/cfs.html`.

- *benchmarks*: at the same time, new submitted benchmarks were shuffled (literal renaming, clauses reordering). Comment line were also removed. Some benchmarks were discarded because of incorrect syntactical format.

– *first round*: all solvers ran on all "correct" benchmarks during 40 minutes (see section 2.4 for the computer description). We first ran all the solvers on industrial benchmarks, then handmade benchmarks and finally randomly generated ones (this last ones were run for 20 minutes only, on faster machines).
In this step, the launching heuristic was applied, and, according to it, each complete solver was launched on each applicable benchmark one time. Randomized solvers (incomplete or not) were launched 3 times on each applicable benchmarks, on industrial and hand-made benchmarks only. To take these 3 executions into account, the median CPU time was taken and the instance was solved if at least one execution solved it (that means that a randomized solver can solve a particular instance and be charged of the maximum CPU time, if only one of the three launches has succeeded).

- *second round*: the top five solvers ran on a part of the remaining unsolved instances during 6 hours. If a solver returned an incorrect result (typically, UNSAT instead of SAT) in the first round, then it was not qualified for this stage (even "unofficially").

2.2. Benchmark submission

The following benchmarks were submitted to **Industrial** category:

bart, homer from Fadi Aloul. Represent FPGA Switch-Box problems, all instances should imply a lot of symmetries, as it is described in [19]. Bart instances are all satisfiable, Homer instances are unsatisfiable.

cmpadd from Armin Biere. These benchmarks encode the problem of comparing the output of a carry ripple adder with the output of a fast propagate and generate adder. They are all unsatisfiable.

dinphil from Armin Biere. These benchmarks are generated from bounded model checking from the well-known dining philosophers example. The instances have the generic name '`dp i t k cnf`', where '`i`' is the number of philosophers, '`k`' is the model checking bound and '`t`' is '`u`' for unsatisfiable or '`s`' for satisfiable. The model for '`i`' philosophers may reach a bug not faster than in '`i`' steps.

cache, comb, f2clk, fifo8, ip, w08, w10 from Emmanuel Dellacherie (TNI-Valiosys, `http://www.tni-valiosys.com/`, France). All these problems represent 18 industrial model-checking examples and 3 combinational equivalence examples.

bmc1 from Eugene Goldberg. Bounded Model Checking (BMC) examples (76 CNFs, 30% of them are unsatisfiable) encoding formal verification of the open-source Sun PicoJava II (TM) microprocessor. These CNFs were generated by Ken Mcmillan (Cadence Berkeley Labs). The complete description of the benchmarks is given at `http://www-cad.eecs.berkeley.edu/~kenmcmil/satbench.html`.

bmc2 from Eugene Goldberg, suggested by Ken Mcmillan (Cadence Berkeley Labs). This small set of 6 BMC instances encodes testing whether a sequential N-bit counter

(file `cntN.cnf`) can reach a final state from an initial state in 2^{N-1} cycles. In the initial state all the bits of the counter are set to 0 and, in the final state, all the bits of the counter are set to 1. All CNFs are satisfiable.

fpga_routing from Eugene Goldberg and Gi-Joon Nam (32 CNFs submitted by E. Goldberg and 6 by G.-J. Nam separately, but all instances were generated by G.-J. Nam). These Boolean SAT problems are constructed by reducing FPGA (Field Programmable Gate Array) detailed routing problems into Boolean SAT. More information on transforming FPGA routing problems into SAT, are available at `http://andante.eecs.umich.edu/sdr/index.html`.

rand_net from Eugene Goldberg. This is a set of *miter* CNFs (all unsatisfiable) produced from randomly generated circuits. To produce a miter, a random circuit is generated first. This circuit is specified by the number of primary input variables (`N`), the number of levels in the circuit (`M`) and the "length" (`K`) of wires connecting gates of the circuit (`K=1` means that the output of a gate may be connected only to the input of a gate of the next level). A circuit consists of AND and OR gates and does not contain inverters. So any circuit implements a monotone function (by adding inverters to a randomly generated circuit one can make it very redundant). Circuits are "rectangular", i.e. the number of primary inputs, the number of gates of mth level, and the number of primary outputs are all equal to `N`. Now, to check if a circuit is equivalent to itself, a miter is formed. This class of benchmarks allow one to vary the "topology" of the circuit by changing the "length" of wires. Each instance is named `rand_netN_M_K.miter.cnf` where `N`, `M` and `K` are the values of parameters described above.

mediator from Steven Prestwich. The encoded problem (described in [47]) is to construct a query plan to supply attributes in a mediator system (e.g., an online bookstore). These problems combine set covering with plan feasibility and involve chains of reasoning that should make them hard for pure local search. Symmetry breaking constraints were not added, in order to make the problems harder for systematic backtrack search. A file `medN.cnf` contains a problem with shortest known plan length `N`.

IBM from Emmanuel Zarpas (IBM). Bounded Model Checking for real hardware formal verification. Benchmarks are partitioned by difficulty in {Easy, Medium, Hard} by the submitter.

The following benchmarks were submitted to **Handmade** category:

lisa from Fadi Aloul. Those instances represent integer factorization problems. They are all satisfiable (see [19]). Note that other factorization problems (given as generators) were submitted (described below).

matrix, polynomial from Chu-Min Li (with Bernard Jurkowiak and Paul W. Purdom). Those instances encode respectively the multiplication of two $n \times n$ matrices using m products, and the multiplication of two polynomials of degree-bound n using m products. Both problems should involve a lot of symmetries (see [8]).

urquhart from Chu-Min Li (with Sebastien Cantarell and Bernard Jurkowiak) [38] and independently from Laurent Simon [10]. All instances are unsatisfiable and proved very hard for all DLL and DP approaches (hard for all resolution-based procedures, in general [57]). Chu-Min Li benchmarks are 3-SAT encoding of Urquhart problems and Laurent Simon are non-reduced encoding (clauses can be long).

hanoi from Eugene Goldberg (but generated by Henry Kautz). These instances represent the classical problem of the Towers of Hanoi, hand-encoded axioms around 1993 (similar to the ones used in [29], but larger instances available).

graphcolor*K* from Dan Pehoushek. Random regular graph coloring problems. Above some number of vertices, most of them should be colorable.

ezfact from Dan Pehoushek. SAT encoding of factorization circuits.

glassy-sat-sel from Federico Ricci-Tersenghi. Selected instances (by the submitter) of medium hardness from the glassy-sat generator (see below).

gridmnbench from Allen Van Gelder. Encode (negated) propositional theorem about a (non realistic) fault-tolerant circuit family.

checkerinterchange from Allen Van Gelder (with Fumiaki Okushi). Planning problem to solve checker interchange problem within deadline.

ropebench from Allen Van Gelder. A linear family of graph coloring problems (sequence of unsatisfiable formulas in 3-CNF). The formula length is linear in the number of variables (namely, $36n$).

qgbench from Hantao Zhang. Small instances of quasigroups with constraints 0–7.

sha from Lintao Zhang and Sharad Malik. CNF encoding of secure hashing problems.

xor-chains (among them, the smallest unsolved unsatisfiable instance with 106 variables, 282 clauses and 844 literals), from Lintao Zhang and Sharad Malik. This encodes verification problems of 2 xor chains.

satex-challenges from Laurent Simon. Selection of (heterogenous) unsolved instances from SAT-Ex [54].

pyhala from Tuomo Pyhàlà. Submitted as a generator. Depending on arguments, it can generate a SAT encoding of factoring of primes (unsat instances) or products of two primes (sat instances). The benchmarks encode multiplication circuits, with predefined output. Two circuits are available (*braun* or *adder-tree* multipliers).

The benchmarks of **Random** category were submitted as generators (except for plainoldcnf and twentyvars):

3sat from the organizers. This generator produces uniform 3-CNF formulas. Checks are performed to prevent duplicate or opposite literals in clauses. In addition, no duplicate clause are created.

glassy-sat from Federico Ricci-Tersenghi (with W. Barthel, A.K. Hartmann, M. Leone, M. Weigt, and R. Zecchina). Generator of hard and solvable 3-SAT instances, corresponding to a glassy model in statistical physics. A description is available as a preprint at `http://xxx.lanl.gov/abs/cond-mat/0111153`.

okrandgen from Oliver Kullman [34,36], k-CNF uniform random generator, based on encryption functions to ensure *strong* and *reliable* random formulae. Detailed descrip-

tion and sources available at `http://cs-svr1.swan.ac.uk/~csoliver/OKgenerator.html`.

hgen2 from Edward A. Hirsch (available from `http://logic.pdmi.ras.ru/~hirsch/benchmarks/hgen2.html`). An instance generated by this generator (3-CNF, 500 variables, 1750 clauses, 5250 literals, seed 1 216 665 065) was the smallest satisfiable benchmark that remained unsolved during the competition. Description: First a satisfying assignment is chosen; then clauses ($3.5n$ of them for n variables) are generated one by one. A literal cannot be put into a clause if

1. There is a less frequent literal.
2. The corresponding variable already appears in the current clause.
3. A variable dependent on it (i.e., occurred together in another clause) already appears in the current clause.
4. A variable dependent on a dependent variable already appears in the current clause.
5. The opposite literal is not satisfying and occurs not more frequently (except for the case that choosing a satisfying literal is our last chance to satisfy this clause).

If the generation process fails (no literal can be chosen), it is restarted from the beginning.

hgen1 from Edward A. Hirsch. Similar to hgen2 except for condition 5.

hgen3 from Edward A. Hirsch. Similar to hgen1, but formulas are *not* required to be satisfiable.

hgen4 from Edward A. Hirsch. Similar to hgen2, but formulas are in 4-CNF, with $9n$ clauses. Also condition 4 is not applied.

hgen5 from Edward A. Hirsch. Similar to hgen2, but formulas are a mix of 3-CNF ($1.775n$ clauses) and 4-CNF ($5.325n$ clauses).

g3 from Mitsuo Motoki. Generates positive instances at random. These instances have only one solution with high probability. Benchmarks were discarded because of a bug in the generator.

plainoldcnf from Dan Pehoushek. Selection of regular random 5-CNF.

twentyvars from Dan Pehoushek. Small instances (in terms of their number of variables) of k-CNF, with $k \in \{6, 7, 8\}$.

2.3. Solver submission

We wanted the competition to be as fair and open as possible. So we did not want to restrict people to a given language (such as C or C++): the only condition was that the solver can run on a standard Linux/Unix box. The solver sources had to be provided, with a suitable makefile. Additional libraries were statically linked to the code. All but one solvers were in C/C++, one was in Java.

limmat Armin Biere. Complete deterministic solver. This is a zchaff-like SAT solver (implemented in C) with an early detection of conflicts in the BCP queue; a constant time lookup of the 'other' watched literal; an optimized ordering of decision

variables and a robust code through sophisticated test framework. More informations and sources are available at http://www.inf.ethz.ch/personal/biere/projects/limmat/.

saturn by Steven Prestwich [48]. Incomplete randomized solver.

2clseq by Fahiem Bacchus. Complete deterministic solver. DPLL with binary clause and equivalence reasoning plus intelligent backtracking and learning [2,3]. Available in source (C++) at http://www.cs.toronto.edu/~fbacchus/2clseq.html.

marchI, marchIse, marchII, marchIIse by Marijn Heule, Hans van Maaren, Mark Dufour, Joris van Zwieten. Complete deterministic solver. Those solvers were designed by postgraduate students for a course given by Hans van Maaren. The heuristics used in those solvers can be found in [64]. Note that some of them (marchIse-hc, marchII-hc, marchIIse-hc) were received after the deadline so we decided to run them hors-concours.

blindsat by Anatoly Plotnikov and Stas Busygin. Complete deterministic solver. A report and the solver source (C++) are available at http://www.vinnica.ua/~aplot/current.html.

ga by Anton Eremeev and Pavel Borisovsky. Incomplete randomized solver. A greedy crossover genetic algorithm.

berkmin by Eugene Goldberg and Yakov Novikov [21]. Complete deterministic solver.

> "Berkmin inherits such features of GRASP, SATO, and Chaff as clause recording, fast BCP, restarts, and conflict clause "aging". At the same time Berkmin introduces a new decision making procedure and a new procedure for the management of the database of conflict clauses. The key novelty of Berkmin is that this database is organized as a chronologically sorted stack. Berkmin always tries to satisfy the topmost unsatisfied clause of the stack. When removing clauses Berkmin tries to get rid of the clauses that are at the bottom of the stack in the first place" [Eugene Goldberg].

The version used for the competition was 62. Berkmin 56 binaries are available at http://eigold.tripod.com/.

unitwalk by Edward A. Hirsch and Arist Kojevnikov [25]. Incomplete randomized solver. UnitWalk is a combination of unit clause elimination (particularly, the idea of Paturi, Pudlák and Zane's randomized unit clause elimination algorithm [46]) and local search. The solver participated in the competition extends this basic algorithm with adding some of 2-resolvents using incBinSat [68], and mixes its random walks with WalkSAT-like [51] walks. The version used for the competition was 0.98. Available in source (C) at http://logic.pdmi.ras.ru/~arist/UnitWalk/.

jquest by Joao Marques-Silva and Inês Lynce. Complete deterministic solver. Jquest is a SAT platform in Java containing various heuristics, data structures and search strategies [40]. The solver was configured with lazy data structures (inspired by both SATO and Chaff), non-chronological backtracking and clause recording (like in Grasp), chaff-like heuristic, randomized backtracking [41] and rapid restarts strat-

egy. JQuest source code is available at `http://sat.inesc.pt/sat/soft/jquest/jquest-src.tgz`.

lsat by Richard Ostrowski, Bertrand Mazure and Lakhdar Sais [45]. Complete deterministic. LSAT detects some boolean functions (equivalence chains, and/or gates) and uses them

- to simplify the original CNF,
- to detect independent variables.

Then a classical DPLL is launched on the simplified CNF, branching only on independent variables.

usat05, usat10 by Bu Dongbo. Incomplete randomized solver. No description available.

sato by Hantao Zhang [65]. Complete deterministic solver.

simo by Armando Tacchella, Enrico Giunchiglia, Marco Maratea [12]. Complete deterministic (wrongly noted randomized in the competition). In Simo3.0 there are features the most recent and effective in SAT like UIP-based learning, restart, 2-literals watching. Simo3.0 is characterized by a new type of heuristic(called GMT). GMT tries to combine Chaff-like and SATZ-like heuristics. The idea is to switch between SATZ-like and Chaff-like heuristics by introducing measures of "probably successful search" and "probably unsuccessful search". The default is to use a Chaff-like heuristic. When the measure of unsuccessful search exceeds a given threshold, SIMO switches to a SATZ-like heuristic. SIMO resumes the Chaff-like heuristic once the measure of successful search exceeds a given threshold. SIMO 2.0 is available in source (C++) at `http://www.mrg.dist.unige.it/~sim/simo/`.

OKsolver by Oliver Kullmann [35]. Complete deterministic solver.

> "OKsolver has been designed to be a "clean solver" as possible, minimising the use of "magical numbers", and for 3-CNF indeed the algorithm is completely generic. OKsolver is a DPLL-like algorithm, with reduction by failed literals (complete and iterated at each node) and autarkies (found when searching for failed literals), while the branching heuristic chooses a variable creating as many new clauses as possible (exploiting full unit clause propagation for all variables), and the first branch is chosen maximising an approximation of the probability, that a branching formula is satisfiable" [Oliver Kullmann].

OKsolver 1.2 source code is available at `http://cs-svr1.swan.ac.uk/~csoliver/`.

dlmsat1, dlmsat2, dlmsat3 by Benjamin Wah and Alan Zhe Wu [53]. Incomplete randomized solvers. Available in source at `http://manip.crhc.uiuc.edu/Wah/programs/SAT_DLM_2000.tar.gz`.

modoc by Allen Van Gelder [44,58,61]. Complete deterministic solver. Binaries available at `ftp://ftp.cse.ucsc.edu/pub/avg/Modoc/`.

rb2cl by Allen Van Gelder [59,60,62]. Complete deterministic solver.

It applies reasoning in the form of certain resolution operations, and identification of equivalent literals. Resolution produces growth in the size of the formula, but within a global quadratic bound; most previous methods avoid operations that produce any growth, and generally do not identify equivalent literals. Computational experience so far suggests that the method does substantially less "guessing" than previously reported algorithms, while keeping a polynomial time bound on the work done between guesses [Allen van Gelder].

zchaff by Lintao Zhang and Sharad Malik [43,67]. Complete deterministic solver. This solver is a carefully engineered DPLL procedure with non-chronological backtracking, learning (clause recording), restarts, randomized branching heuristic and an innovative notion of "heuristic learning" (VSIDS). Zchaff source code is available at `http://www.ee.princeton.edu/~chaff`.

partsat by John Kolen. Complete deterministic solver. No description available.

2.4. Computers available

The competition was held on 2 clusters of Linux PCs, kindly provided by the University of Cincinnati, thanks to John Franco. The first cluster of 32 dual PIII-450 computers with 1 GB of RAM was used to run most of the competition: compliance testing, first stage for handmade and industrial benchmarks, second stage for all benchmarks. The second cluster, consisting of 16 Athlon 1800+ machines, was used to run the first stage on random instances. Only one processor was used, virtual (hard drive) memory was disabled and each program was given 900 MB of memory.

3. The results

The results of the competition were released during the SAT2002 symposium. The detailed results can be found on the competition web page `http://www.satlive.org/SATCompetition/2002/`.

Some of the solvers were found buggy by the organizers during compliance testing, returned to their authors, and corrected (some of them). However, this was not the aim of this phase, and cannot be considered as a guarantee of any kind of testing. Only wrong answers (or obvious crashes) were reported as bugs. We also noticed problems with some solvers during the first round but we did not accept new version of the solvers. Here are some of the things one must be aware of before reading the results.

But, first of all, we must begin with a word of caution: The following results should be considered with care, because they correspond to the behavior of a particular version of each solver (the one that was submitted) on the benchmarks accepted for the competition, on a particular computer under a particular operating system.[5] The competition results should increase our knowledge about solvers, but one inherent risk of such snap-

[5] Linux-SMP 2.4.3, solvers binaries compiled by `gcc 2.96`.

shot is that results can be misinterpreted, and thus lead to wrong pictures of the state of the art.

We discarded some of the solvers from the competition (ran *hors concours*) because they demonstrated unexpected behavior; mainly, claimed UNSAT for a satisfiable instance. Most of the time, the problem showed up only on a few families of benchmarks. Also some versions of the marchXYZ solvers were hors concours from the beginning, because these versions were submitted substantially after the deadline (but before the first stage of the competition). Hors concours solvers results *are* also displayed on the competition web page. For instance, the `lsat` solver answered incorrectly on some instance (there was actually a bug in the code), but the corrected version (as well as the buggy version) solved easily all `urquhart` instances and the `xor-chains` benchmarks, awarded during the competition (adding the detection of boolean functions pays on small but hard hand-made formulas). But it was officially discarded because of its bug.

Some specific problems occured with other solvers (not hors concours). In the following two cases, the picture given by the competition results does not reflect the real solvers performance:

Berkmin was composed of two engines, one for small/medium instances, and one for large instances. The latter just crashed on Linux (the authors tested it under MS Windows and Solaris only). That problem was not detected during the compliance testing (for more details, see `http://www.satlive.org/SATCompetition/2002/berkmin.html`). Note that other solvers also crashed sometimes, especially during the second round where benchmarks were larger.

JQuest did not output a correct certificate when the instance had less variables than the `nbvar` parameter provided in the "`p cnf nbvar nbclause`" line (because in that case, it renames internally the variables ids). For that reason, JQuest is reported not solving those instances.

The best way to view the detailed results of the competition is to take a look at all the traces at `http://www.ececs.uc.edu/sat2002/scripts/menu_choix2.php3`; the summaries of the results per competition stage per category follow. The instances used for the competition are available on SATLIB.[6]

3.1. First stage

In tables 1–3, letfmost number is the number of solved series (a series is solved if at least one of its instances is solved). Rightmost number (where breaking a tie is necessary) denotes the total number of instances solved.

In each category, the top five solvers went to the second stage.

[6] `http://www.satlib.org/`.

Table 1
First stage results on Industrial instances.

	Complete solvers on Industrial benchmarks	
23	zchaff	
22	limmat	
18	berkmin	
15	simo	
14	2clseq	
12	jquest	
11	oksolver	
10	rb2cl, march2[se], modoc	
3	blindsat	
	All solvers on satisfiable Industrial benchmarks	
11	zchaff	
10	limmat	
8	berkmin	
7	simo	
6	unitwalk	57
6	2clseq	55
6	dlmsat2	54
6	dlmsat1	53
6	rb2cl	50
6	saturn	49
6	oksolver	47
6	march2	44
6	march2se	43
6	jquest	32
5	usat10/usat05, modoc, dlmsat3	
3	blindsat	
1	ga	

3.2. Second stage

In this stage, Top 5 solvers were run on smallest remaining unsolved instances for 6 hours. For industrial benchmarks, only 31 instances remained unsolved. So, we used all of them for complete solvers, and only satisfiable instances for all solvers. Thus, in table 4, berkmin, oksolver, 2clseq, simo and zchaff were launched on all instances. unitwalk was only launched on all instances that were not known to be unsatisfiable. Over the 31 instances, only 15 were solved. One can notice that, surprisingly, berkmin and 2clseq are able to solve comb/comb3 instance in less than 2000 s. This benchmark was not solved during the first stage of the competition (recall that CPU cut-off was 2400 s on the same machines), due to the use of our launch-heuristic (comb1 and comb2 are still unsolved, and considered as easiest by the heuristic). Let us also notice how

Table 2
First stage results on Handmade instances.

Complete solvers on Handmade benchmarks		
20	berkmin, oksolver	
19	2clseq, limmat, zchaff	
18	simo	
17	march2se	
16	jquest	
15	rb2cl, march2	
14	modoc	
2	blindsat	
All solvers on satisfiable Handmade benchmarks		
11	berkmin, oksolver, unitwalk	
10	zchaff	73
10	limmat	65
10	2clseq	63
9	dlmsat2, simo, usat05/10	
8	dlmsat3, dlmsat1, jquest, march2se, saturn	
7	march2, rb2cl	
5	modoc	
2	ga, blindsat	

zchaff seems well-tuned for this category: it is able to solve instances with millions of literals.

For handmade instances (table 5), only a few families remained (but several instances per family) so we took the smallest 2 SAT+UNSAT instances in each family for complete solvers, and 2 smallest SAT benchmarks in each family for all solvers (families are urq/urq*bis, urq/urq, xor-chain/x1_*, xor-chain/x1.1_*, xor-chain/x2_*, matrix, Lisa, satex-c/par32-*-c, satex-c/par32, pbu-4, pbs-4 and hanoi). One can notice that the only incomplete solver used in this stage (i.e., uniwalk) was not able to solve any instance during this stage. Let us just recall that most instances in this table are easy for lsat (xor-chains, urq, par32), which was hors-concours.

The selection of random benchmarks was guided by the following considerations: the smallest unsolved unsatisfiable instance had already been found in the handmade category (the smallest unsolved random instance was larger than it). Concerning SAT instances, all the SAT instances in the industrial and handmade categories were tested for the second stage and the smallest unsolved one had 82 345 literals (handmade pbs4 instance). One feature of random category is that most instances are unknown. So, to be sure to award the smallest SAT instance, we needed to keep the smallest instances for the second stage, independently of their series. As a consequence, if all the series were present in that second stage, the number of instances per series varied.

Table 3
First stage results on randomly generated instances.

Complete solvers on randomly generated benchmarks		
34	2clseq, oksolver	
32	march2, march2se	
31	rb2cl	616
31	simo	569
31	berkmin	541
30	zchaff	
28	limmat, modoc	
17	jquest	
4	blindsat	
All solvers on satisfiable randomly generated benchmarks		
23	dlmsat1,dlmsat2,dlmsat3	
22	unitwalk	
21	oksolver	261
21	usat10	257
21	saturn	255
21	2clseq	228
20	march2[se], rb2cl, simo, usat05	
19	berkmin	
18	modoc, zchaff	
16	limmat	
9	jquest	
5	ga	
4	blindsat	

Table 6 shows all the results for randomly generated instances, sorted by their respective length. Note that all solved instances were previously known as SAT (by forced-SAT generators).

We finally give as summary of results, in tables 7–9. The leftmost number denotes the total number of instances solved during the second stage. Rightmost numbers (if any) denote the 1st stage result (number of solved series and total number of instances solved to break ties).

3.3. Benchmarks

We awarded the two smallest (one satisfiable and one unsatisfiable) instances that remained unsolved during the competition. Of course, both instances participated in the second stage, i.e., the top 5 solvers were run on them for 6 hours! Note that we did not take into account here the instances that were submitted as “unknown”.

The smallest hard unsatisfiable instance **xor-chain/x1_36** (106 variables, 844 literal occurrences) was submitted by Lintao Zhang and Sharad Malik.

Table 4
Industrial benchmarks used for the second stage. "N*", in the "SAT?" colmun, denotes a previously-unknown benchmark claimed to be Unsat (recall that no proof were given, and solver have to be trusted on this answer). All fpga-r/file are fpga-routing/k2fix_gr_file, 6pipe_o for 6pipe_6_ooo, and satex-c/cnf-r4-i for satex-challenges/cnf-r4-b1-k1.i-comp.

Name	Nb Var	Nb Clauses	Length (Max)	SAT?	Solved by
Homer/homer17	286	1 742	3 718 (12)	N	limmat (6957 s)
Homer/homer18	308	2 030	4 312 (12)	N	
Homer/homer19	330	2 340	4 950 (12)	N	
Homer/homer20	440	4 220	8 800 (12)	N	
dinphil/dp11u10	9 197	25 271	59 561 (12)	N	
dinphil/dp12u11	11 137	30 792	72 531 (13)	N	
comb/comb1	5 910	16 804	38 654 (29)	–	
comb/comb2	31 933	112 462	274 030 (14)	–	
comb/comb3	4 774	16 331	39 495 (14)	N*	berkmin (1025 s) 2clseq (1772 s)
f2clk/f2clk_40	27 568	80 439	186 255 (26)	–	
f2clk/f2clk_50	34 678	101 319	234 655 (26)	–	
fifo8/fifo8_300	194 762	530 713	1 200 865 (12)	N*	zchaff (5716 s)
fifo8/fifo8_400	259 762	707 913	1 601 865 (12)	N*	zchaff (16083 s)
ip/ip36	47 273	153 368	366 122 (21)	N*	limmat (20919 s) zchaff (6982 s)
ip/ip38	49 967	162 142	387 080 (21)	N*	limmat (5640 s) zchaff (13217 s)
ip/ip50	66 131	214 786	512 828 (21)	–	
w08/w08_14	120 367	425 316	1 038 230 (16)	Y	zchaff (16359 s)
w08/w08_15	132 555	469 519	1 146 761 (16)	–	
bmc2/cnt10	20 470	68 561	187 229 (4)	Y	
fpga-r/2pinvar_w8	3 771	270 136	1 620 816 (7)	–	
fpga-r/2pinvar_w9	5 028	307 674	2 438 766 (9)	–	
fpga-r/2pin_w8	9 882	295 998	1 727 100 (7)	–	
fpga-r/2pin_w9	13 176	345 426	2 606 340 (9)	–	
fpga-r/rcs_w8	10 056	271 393	550 328 (9)	–	
satex-c/cnf-r4-1	2 424	14 812	39 764 (25)	Y	limmat (21339 s) berkmin (13071 s)
satex-c/cnf-r4-2	2 424	14 812	39 764 (25)	Y	limmat (20454 s)
fvp-unsat/6pipe	15 800	394 739	1 157 225 (116)	N	zchaff (12714 s)
fvp-unsat/6pipe_o	17 064	545 612	1 608 428 (188)	N	zchaff (4398 s)
fvp-unsat/7pipe	23 910	751 118	2 211 468 (146)	N	
sha/sha1	61 377	255 417	769 041 (5)	Y	
sha/sha2	61 377	255 417	769 041 (5)	Y	

The smallest hard satisfiable instance **hgen2-v500-s1216665065** (500 variables, 5250 literal occurrences) was generated by Edward A. Hirsch's random instance generator hgen2.

Note that instances with fewer variables also remained unsolved, but the winner was determined by the total number of *literal occurrences* in the formula (note that a

Table 5
Handmade benchmarks used for the second stage. pbu and pbs are pyhala-braun-unsat and pyhala-braun-sat, respectively. Comp/Un. denotes which solvers were used: "C" means that we tried berkmin, oksolver, 2clseq, limmat and zchaff on the considered benchmark. "U" means that unitwalk was also launched (see previous section for the Top 5 in Handmade category) and note that berkmin, oksolver, limmat and zchaff where common in both Complete/All categories.

Name	Nb Var	Nb Clauses	Length (Max)	SAT?	Comp/Un.	Solved by
urq/urq3_25bis	99	264	792 (4)	N	C	berkmin (2825 s)
xor-chain/x1_36	106	282	844 (4)	N	C	
xor-chain/x1.1_40	118	314	940 (4)	N	C	
xor-chain/x1_40	118	314	940 (4)	N	C	zchaff (3165 s)
xor-chain/x2_40	118	314	940 (4)	N	C	
xor-chain/x1.1_44	130	346	1 036 (4)	N	C	
xor-chain/x2_44	130	346	1 036 (4)	N	C	
urq/urq3_25	153	408	1 224 (4)	N	C	
urq/urq4_25bis	192	512	1 536 (4)	N	C	
urq/urqu4_25	288	768	2 304 (4)	N	C	
matrix/Mat26	744	2 464	6 432 (4)	N	C	zchaff (18604 s)
satex-c/par32-2-c	1 303	5 206	15 246 (4)	Y	C,U	
satex-c/par32-1-c	1 315	5 254	15 390 (4)	Y	C,U	
Lisa/lisa21_99_a	1 453	7 967	26 577 (23)	Y	C,U	berkmin (20459 s)
satex-c/par32-2	3 176	10 253	27 405 (4)	Y	C,U	
satex-c/par32-1	3 176	10 277	27 501 (4)	Y	C,U	
pbu-4/p-b-u-35-4-03	7 383	24 320	62 950 (4)	N	C	berkmin (3693 s) oksolver (3073 s) 2clseq (10821 s) limmat (4718 s) zchaff (2738 s)
pbs-4/p-b-s-40-4-03	9 638	31 795	82 345 (4)	Y	C,U	
pbs-4/p-b-s-40-4-04	9 638	31 795	82 345 (4)	Y	C,U	zchaff (3182 s)
pbu-4/p-b-u-40-4-01	9 638	31 795	82 345 (4)	N	C	
hanoi/hanoi6	4 968	39 666	98 346 (10)	Y	C,U	berkmin (2551 s)
matrix/Mat317	24 435	85 050	227 610 (4)	–	C,U	

hard randomly generated formula in 4-CNF will have a much greater clauses/variables ratio, not to say about the length of its clauses).

4. Other views of the competition

As we said, one of the risks of such competition is that it results can be misleading (how strong are the results w.r.t. the performance of solvers in a *real* situation: embeded component in a model checker or a planning system for instance?). As long as the competition was running, we had to make decisions, each of them having a direct impact on final results. We have collected a large amount of data, much more valuable than just the name of the final winner. Here, we try to interpret these data from a different point of view. Note that the following is based on the data collected during the first stage of the competition.

Table 6
Random benchmarks used for the second stage. Clause max length is not reported (it is exactly 4 for all benchmarks). Comp is "C" if 2clseq, oksolver, marchII, marchIIse and rb2cl were launched; and "U" if dlmsat [1–3], unitwalk and oksolver were launched (see previous section for Top 5 solvers and random instances).

Name	Nb Var	Nb Cl.	Length	SAT?	Comp/Un.	Solved by
hgen3-v300-s1766565160	300	1 050	3 150	–	C	
hgen3-v300-s1817652174	300	1 050	3 150	–	C	
hgen3-v300-s229883414	300	1 050	3 150	–	C	
hgen3-v350-s1711636364	350	1 225	3 675	–	C	
hgen3-v350-s524562458	350	1 225	3 675	–	C	
hgen2-v400-s161064952	400	1 400	4 200	Y	C,U	unitwalk (20199 s)
hgen3-v400-s344840348	400	1 400	4 200	–	C	
hgen3-v400-s553296708	400	1 400	4 200	–	C	
hgen2-v450-s41511877	450	1 575	4 725	Y	C,U	dlmsat3 (94 s)
hgen3-v450-s432353833	450	1 575	4 725	–	C	
unif-c1700-v400-s734590802	400	1 700	5 100	–	C	
okgen-c1700-v400-s2038016593	400	1 700	5 100	–	C	
hgen2-v500-s1216665065	500	1 750	5 250	Y	C,U	
hgen3-v500-s1349121860	500	1 750	5 250	–	C	
hgen3-v500-s1769527644	500	1 750	5 250	–	C	
hgen3-v500-s1803930514	500	1 750	5 250	–	C	
hgen3-v500-s1920280160	500	1 750	5 250	–	C	
glassybp-v399-s382874052	399	1 862	5 586	Y	C,U	oksolver (13034 s) marchII (7100 s) marchIIse (7064 s)
glassybp-v399-s499089820	399	1 862	5 586	Y	C,U	
glassyb-v399-s500582891	399	1 862	5 586	Y	C,U	oksolver (13834 s)
glassyb-v399-s732524269	399	1 862	5 586	Y	U	oksolver (8558 s)
glassy-v450-s1188040332	450	2 100	6 300	Y	U	oksolver (15444 s)
glassy-v450-s1679149003	450	2 100	6 300	Y	U	oksolver (18766 s)
glassy-v450-s1878038564	450	2 100	6 300	Y	U	
glassy-v450-s2052978189	450	2 100	6 300	Y	U	
glassy-v450-s325799114	450	2 100	6 300	Y	U	
glassybp-v450-s1173211014	450	2 100	6 300	Y	U	
glassybp-v450-s1349090995	450	2 100	6 300	Y	U	
glassybp-v450-s1976869020	450	2 100	6 300	Y	U	
glassybp-v450-s2092286542	450	2 100	6 300	Y	U	
glassybp-v450-s40966008	450	2 100	6 300	Y	U	
glassyb-v450-s1529438294	450	2 100	6 300	Y	U	
glassyb-v450-s1709573704	450	2 100	6 300	Y	U	
glassyb-v450-s1729975696	450	2 100	6 300	Y	U	

4.1. SOTA view

Geoff Sutcliffe and Christian Suttner are running the CASC[7] competition for many years now, and provide in [55] some clues about what is a fair way to evaluate au-

[7] CASC = "CADE ATP System Competition", CADE = "Conference on Automated Deduction", ATP = "Automated Theorem Proving".

Table 7
Second stage results on Industrial benchmarks (summary).

Complete solvers		All solvers on satisfiable benchmarks		
7	**zchaff**	2	**limmat**	
5	limmat	1	zchaff	[11]
2	berkmin	1	berkmin	[8]
1	2clseq	0	simo	[7]
0	simo	0	unitwalk	[6]

Table 8
Second stage results on Handmade benchmarks (summary).

Complete solvers			All solvers on satisfiable benchmarks		
3	**zchaff**		2	**berkmin**	[11]
2	berkmin		2	zchaff	[10]
1	oksolver	[20]	1	oksolver	[11]
1	limmat	[19 140]	1	limmat	[10]
1	2clseq	[19 126]	0	unitwalk	

Table 9
Second stage results on randomly generated benchmarks (summary).

Complete solvers			All solvers on satisfiable benchmarks		
4	**oksolver**		5	**oksolver**	
3	march2, march2se		1	dlmsat3	[23]
0	2clseq	[34]	1	unitwalk	[22]
0	rb2cl	[31]	0	dlmsat1, dlmsat2	

tomated theorem provers. One of the key ideas of their work is the notion of *State Of The Art (SOTA)* solver. It is based on a subsumption relationship between the set of instances solved by the competing solvers: "a solver A is better than a solver B iff solver A solves a strict subset of the instances solved by solver B". The underlying idea is that any solver being the only one to solve an instance is meaningful. The subsumption relationship provides a partial order between the competing solvers, and a virtual solver representing advances of the whole community, *the SOTA solver*. This solver can solve every problem solved by any of the competing solvers (so would be the unique maximal element for the subsumption relationship). There is a little chance that the SOTA solver is a real solver, and thus the notion of *SOTA contributors* is introduced by the authors. They correspond to the maximal elements of the subsumption relationship. The SOTA solver is equivalent to the set of SOTA contributors running in parallel.

Tables 10–12 show the problems solved by only one solver during the first stage of the competition, for the three categories of benchmarks (hors-concours solvers are discarded). We can now find SOTA contributors from each table:

Table 10
Uniquely solved Industrial benchmarks during the first stage.

Solver	Bench (shortname)	CPU (s)
2clseq	bmc2/cnt09.cnf	198.00
2clseq	rand_net/50-60-10	96.33
2clseq	rand_net/60-40-10	16.03
2clseq	rand_net/60-60-10	180.45
2clseq	rand_net/60-60-5	172.22
2clseq	rand_net/70-40-10	35.02
2clseq	rand_net/70-40-5	52.15
2clseq	rand_net/70-60-10	188.11
2clseq	rand_net/70-60-5	611.71
2clseq	satex-c/c6288-s	0.73
2clseq	satex-c/c6288	0.77
berkmin	dinphil/dp10u09	321.67
berkmin	fpga-r/rcs_w9	2 054.50
berkmin	satex-c/cnf-r4-b2-k1.1	1 402.24
limmat	Homer/homer16	2 315.05
limmat	dinphil/dp12s12	5.21
modoc	Homer/homer15	843.87
zchaff	fifo8/fifo8_200	1 417.68
zchaff	w10/w10_70.cnf	661.74
zchaff	satex-/9vliw_bp_mc	1 274.94
zchaff	fvp-u.2.0/5pipe	186.58
zchaff	fvp-u.2.0/5pipe_3_oo	533.11
zchaff	fvp-u.2.0/5pipe_4_ooo	1 667.30
zchaff	fvp-u.2.0/5pipe_5_ooo	814.92
zchaff	fvp-u.2.0/7pipe_bug	452.14

Table 10 (Industrial) SOTA contributors for industrial instances are 2clseq, berkmin, limmat, modoc and zchaff. Note that 2clseq worked very well on rand_net benchmarks, same thing for zchaff and pipe instances.

Table 11 (Handmade) SOTA contributors for hand-made benchmarks are berkmin, limmat, rb2cl and zchaff.

Table 12 (Random) SOTA contributors for randomly generated benchmarks are 2clseq, dlmsat1, dlmsat3, oksolver, unitwalk, usat05, usat10.

In order to obtain a total order among the solvers, one must first classify benchmarks themselves. For this, the notion of SOTA contributors can be used [55]: benchmarks solved by all SOTA contributors are said *easy*, those solved by *at least one SOTA contributor* (but not all) are called *difficult*.[8] Note that the benchmarks not solved by any solver are not considered here.

Now, it is easy to rank the solvers accordingly to the number of *difficult* instances they can solve. Tables 13 and 14 provide a summary for SAT2002 competition, for respectively complete solvers and satisfiable benchmarks.

[8] A degree of difficulty can be computed using the ratio number of failing SOTA contributors over the total number of SOTA contributors.

Table 11
Uniquely solved Handmade benchmarks during the first stage.

Solver	Benchmark	CPU (s)
berkmin	hanoi6_on	295.04
berkmin	rope_1000	2 058.62
berkmin	x1.1_32	57.13
berkmin	x1_32	82.18
limmat	pyhala-braun-sat-40-4-01	1 295.77
rb2cl	lisa21_1_a	1 955.62
zchaff	lisa21_2_a	287.74
zchaff	pyhala-braun-sat-40-4-02	971.59
zchaff	pyhala-braun-unsat-35-4-01	1 474.30
zchaff	pyhala-braun-unsat-35-4-02	1 653.80
zchaff	Urquhart-s3-b5	1 507.11
zchaff	x1.1_36	786.60
zchaff	x2_36	1 828.81

What can we conclude? First of all, we awarded SOTA contributors. Looking at the SOTA ranking per category, berkmin, zchaff and oksolver would be awarded. Limmat, our fourth awarded, is most of the time third after zchaff and berkmin in industrial and hand-made categories. So the result of the SAT2002 competition looks reasonable. Berkmin certainly deserves a special attention from the community, since despite the bug that made it crashed on more than 100 benchmarks, it is ranked first in the industrial category using the SOTA system. This little impact of crashes can certainly be due to the fact that only the second SAT-engine of Berkmin crashed, which means that most instances on which Berkmin crashed are hard for all solvers anyway (this engine is called for hardest instances only).

Furthermore, we now have difficult and unsolved instances for the next competition. The degree of difficulty provided by the SOTA system can be used to tune solvers for the next competition: first try to solve instances of medium difficulty, then try the really hard ones. All that information will be included in the instances archive.

4.2. Graphical analysis of SOTA CPU performances

The SOTA ranking also allows to focus on subsets of solvers/benchmarks. We can for instance represent the above results in a graphical way, mixing this time complete and incomplete solvers. Doing this, we can take CPU time results into account to give a better picture of SOTA contributors performance. Figures 1–3 show for all respective SOTA contributors how much CPU time is needed to solve an increasing number of instances.

Such representation is important for the validation of the CPU time slice parameter. For this competition, it was arbitrarily chosen[9] and one can ask whether this value can play a crucial role in the results.

[9] Based only on the number of benchmarks/solvers and machines.

Table 12
Uniquely solved randomly generated benchmarks during the first stage.

Solver	Benchmark	CPU (s)
2clseq	5col100_15_1	1 148.08
2clseq	5col100_15_4	1 097.78
2clseq	5col100_15_5	1 018.13
2clseq	5col100_15_6	1 353.14
2clseq	5col100_15_7	748.21
2clseq	5col100_15_8	1 160.66
2clseq	5col100_15_9	821.42
dlmsat1	hgen5-v300-s1895562135	391.94
dlmsat1	hgen5-v300-s528975468	512.81
dlmsat1	hgen2-v300-s1807441418	651.35
dlmsat1	hgen3-v450-s646636548	141.16
dlmsat1	hgen4-v200-s2074278220	16.79
dlmsat1	hgen4-v200-s812807056	1 158.20
dlmsat3	hgen3-v450-s356974048	821.42
dlmsat3	4col280_9_2	1 766.60
dlmsat3	4col280_9_4	6.60
dlmsat3	4col280_9_6	1 792.59
dlmsat3	4col280_9_7	108.43
oksolver	glassy-v399-s1993893641	1 162.07
oksolver	glassy-v399-s524425695	814.44
oksolver	glassybp-v399-s1499943388	273.31
oksolver	glassybp-v399-s944837607	615.27
oksolver	glassyb-v399-s1267848873	1 001.71
oksolver	5cnf_3900_3900_160	243.24
oksolver	5cnf_3900_3900_170	911.32
oksolver	5cnf_3900_3900_180	1 790.46
oksolver	5cnf_3900_3900_190	1 778.96
oksolver	5cnf_4300_4300_090	1 390.55
oksolver	5cnf_4300_4300_100	1 311.92
unitwalk	hgen2-v650-s2139597266	536.09
unitwalk	hgen2-v700-s543738649	32.38
unitwalk	hgen2-v700-s548148704	0.62
unitwalk	hgen3-v500-s1754870155	157.03
usat05	okgen-c2550-v600-s552691850	5.57
usat10	hgen3-v450-s1400022686	0.39

The first observation from all the figures is that, in general, the order of the respective curves does not change after a certain CPU time (there is mostly no crossing of lines after 100 s). There are however two exceptions. First, on figure 1, for 2clseq and zchaff: 2clseq did not solve any other problem after 1000 s, which allowed zchaff to solve more problems in the given CPU time. 2clseq would probably have been considered in a better way if the cut-off had been fixed to less than 1000 s. Secondly, on Handmade benchmarks, the Berkmin curve crosses the one of zchaff just after 2000 s (this can be due to the internal bug of Berkmin). One can ask the question of what happened to the results if

Table 13
Number of difficult SAT+UNSAT benchmarks solved by each solver during the first stage.

Industrial		Handmade		Randomly generated	
Solver	# solved	Solver	# solved	Solver	# solved
berkmin	121	berkmin	68	oksolver	548
zchaff	104	zchaff	68	marchII	543
2clseq	93	limmat	47	marchIIse	543
limmat	87	simo	43	2clseq	369
simo	57	2clseq	34	rb2cl	338
rb2cl	36	oksolver	33	zchaff	305
oksolver	35	jquest	16	simo	298
jquest	31	marchIIse	13	berkmin	263
marchIIse	29	rb2cl	10	limmat	221
modoc	26	marchII	9	modoc	219
blindsat	20	modoc	7	jquest	122
marchII	16	blindsat	0	blindsat	3

Table 14
Number of difficult satisfiable benchmarks solved by each solver during the first stage.

Industrial		Handmade		Randomly generated	
Solver	# solved	Solver	# solved	Solver	# solved
berkmin	68	zchaff	35	oksolver	548
zchaff	64	berkmin	33	marchII	543
limmat	54	limmat	29	marchIIse	543
2clseq	46	simo	27	2clseq	369
unitwalk	41	oksolver	23	rb2cl	338
simo	39	2clseq	21	zchaff	305
dlmsat2	37	unitwalk	11	simo	298
dlmsat1	36	dlmsat1	11	berkmin	263
rb2cl	35	marchIIse	10	dlmsat1	255
dlmsat3	35	dlmsat3	10	dlmsat2	234
saturn	34	saturn	9	dlmsat3	233
usat10	33	usat05	9	unitwalk	227
usat05	31	usat10	9	limmat	221
oksolver	30	dlmsat2	9	modoc	219
marchIIse	26	marchII	7	saturn	208
jquest	24	rb2cl	6	usat10	206
blindsat	20	modoc	5	usat05	205
modoc	20	jquest	4	jquest	122
ga	18	blindsat	0	blindsat	3
marchII	13	ga	0	ga	3

the CPU time slice was less than 2000 s. At last, as expected, one can notice on figure 3 that uncomplete solvers obtain the best performances on SAT instances (the rightmost curve is oksolver on SAT+UNSAT benchmarks). Obviously – oksolver is complete – the picture is quite different as soon as we take into account all (SAT+UNSAT) answers

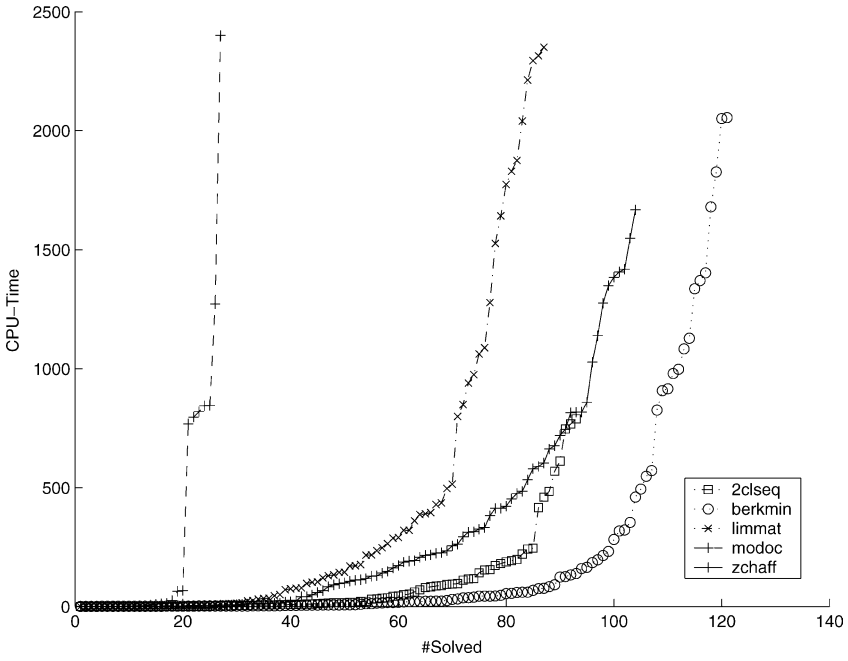


Figure 1. Number of instances solved vs. CPU time for SOTA contributors: Industrial benchmarks.

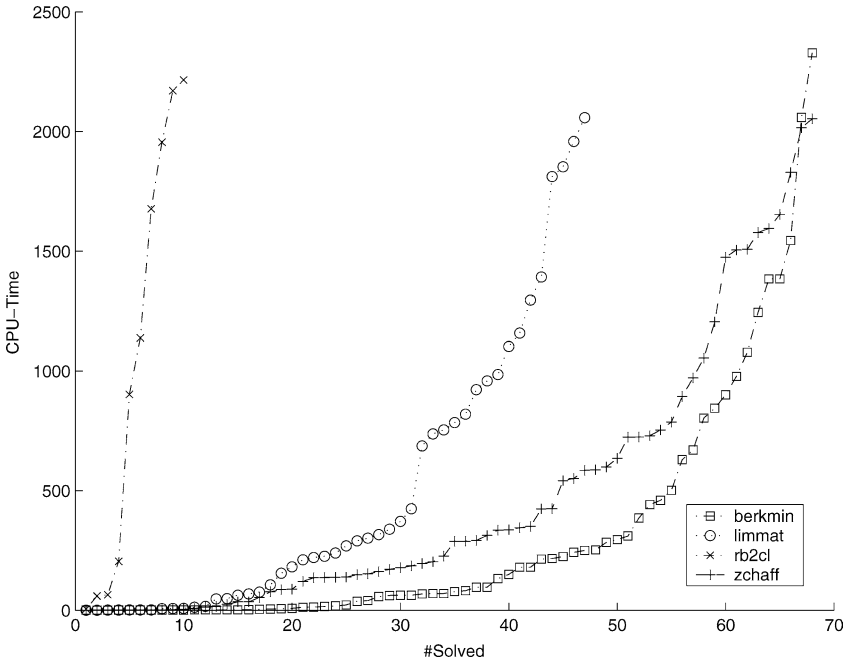


Figure 2. Number of instances solved vs. CPU time for SOTA contributors: Handmade benchmarks.

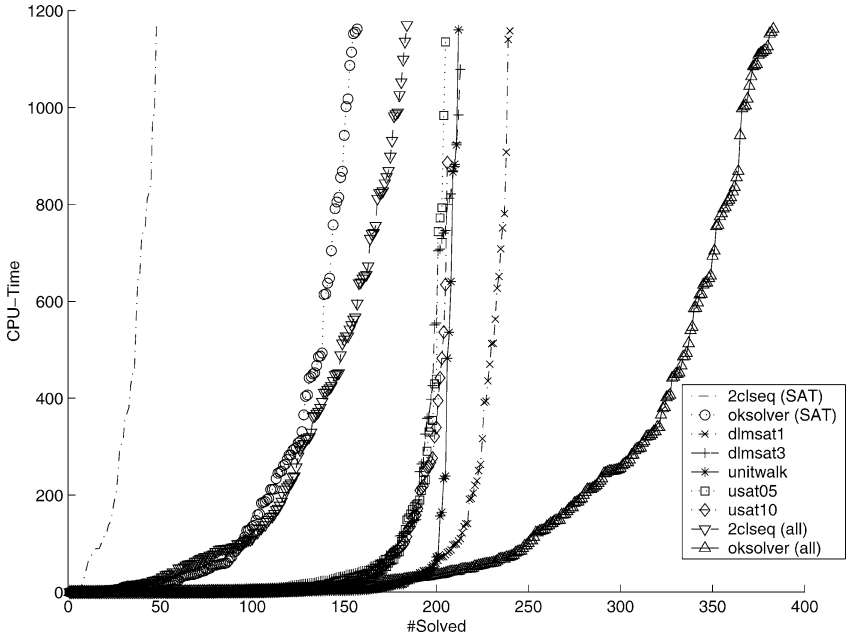


Figure 3. Number of instances solved vs. CPU time for SOTA contributors: Randomly generated benchmarks (note that the CPU time slice was smaller for these benchmarks). For each complete solvers (2clseq and oksolver), we plot two curves: (1) a curve for SAT results only (in order to compare their performances to uncomplete solvers) and (2) for all benchmarks (SAT+UNSAT).

for oksolver. It may be at last interesting to notice that all uncomplete solvers obtain more or less the same curve (observe how close are the curves for dlmsat1, dlmsat2, unitwalk, usat05 and usat10).

Two conclusions may be drawn. First, the growth of the curves on industrial benchmarks clearly shows that 2500 s is enough. Moreover, we can guess that a CPU time of 1500 s would have been sufficient. On hand-made instances, the picture is not so clear. May be this is due to the scalability of generated instances, which sometimes allows a smooth growth of the needed CPU time (a smooth growth may also be observed on random instances).

4.3. Cumulative CPU analysis of results

There is a lot of different ways to study the data collected during the first stage. Here, we give a ranking similar to the one of SAT-Ex. Each solver is given a maximal amount of time to solve an instance (recall that the launching heuristic also applies here). If it cannot solve the instance, then we only penalize it with the maximal CPU time. The ranking is given by the sum of all CPU time. However, this kind of ranking implies to penalize solvers that cannot solve a given instance, instead of just counting successes, we also have to count failures. So, a problem occurs with incomplete solvers on Unknown instances (whose can be Unsat). Thus, we reconsider how to compare

Table 15
Cumulative CPU time on Industrial benchmarks. Beware, we partition here complete vs. incomplete solvers.

Complete			Incomplete		
Solver	CPU (hours)	# solved (245)	Solver	CPU (hours)	# solved (137)
berkmin	54	175	**unitwalk**	54	57
zchaff	66	163	dlmsat1	57	53
2clseq	70	146	dlmsat2	57	54
limmat	79	144	saturn	60	49
simo	97	105	dlmsat3	63	51
rb2cl	114	81	usat05	64	44
oksolver	115	78	usat10	64	46
modoc	117	79	ga	79	20
marchIIse	118	72			
jquest	122	71			
marchII	128	59			
blindsat	146	25			

Table 16
Cumulative CPU time on Handmade benchmarks. Beware, we partition here complete vs. incomplete solvers.

Complete			Incomplete		
Solver	CPU (hours)	# solved (276)	Solver	CPU (hours)	# solved (156)
berkmin	83	160	**dlmsat1**	71	56
zchaff	88	160	dlmsat3	71	54
limmat	99	139	dlmsat2	76	52
simo	105	134	saturn	80	44
2clseq	107	125	unitwalk	84	45
oksolver	109	122	usat05	84	33
rb2cl	120	102	usat10	85	31
jquest	126	98	ga	96	15
marchII	129	91			
marchIIse	129	98			
modoc	130	89			
blindsat	172	18			

solvers, and we rather partition solvers in Complete/Uncomplete solvers rather than by solver/benchmarks (e.g., Complete on ALL, All on SAT).

Results are given in tables 15 (Industrial), 16 (Handmade) and 17 (Random). For randomized solvers, the median CPU time needed for a given benchmark is taken into account. As already noted using the SOTA system, berkmin is in head on industrial benchmarks, the whole picture tends to reinforce our awards, in all categories.

Table 17
Cumulative CPU time on Random benchmarks. Beware, we partition here complete vs. incomplete solvers.

Complete			Incomplete		
Solver	CPU (hours)	# solved (1502)	Solver	CPU (hours)	# solved (1502)
oksolver	248	834	**dlmsat1**	325	541
marchII	253	829	unitwalk	332	513
marchIIse	254	829	dlmsat2	332	517
2clseq	310	655	dlmsat3	335	519
rb2cl	316	616	usat10	339	492
zchaff	326	573	usat05	340	491
berkmin	333	541	saturn	345	493
simo	337	569	ga	462	116
limmat	361	461			
modoc	370	448			
jquest	422	252			
blindsat	464	115			

5. Difficulties and future competitions

Despite a good maturity level in solvers, pure-SAT competitions are not so frequent (the previous one took place 7 years ago). In some aspect, this competition has surpassed all previous competitions (in the number of benchmarks and solvers, the availability of results on the web), but some choices were hard to take. Some of them were good, some were not. Because another competition will be held next year, it is important to take stock of this one, in order to think about the next one.

5.1. Some lessons

Let us begin our first assessments with our own difficulties during the competition. Despite of our efforts to make everything automatic, life is always more complex than predictions. In particular, we got unexpected bugs (or should we say that we did not expect too many *expected* unexpected bugs?), in all the stages of the competition, including bugs in benchmarks:

- problems with input/output format (solvers),
- duplicate literals or opposite literals in clauses (benchmarks),
- internal timeout hardcoded in solvers,
- wrong declaration of benchmarks (SAT vs. UNSAT) and solvers (randomized vs. deterministic);

Even worse, the side effects of bugs were often important (e.g., if an instance is buggy, it should be dropped, which affects the whole procedure; the running scripts had to be modified to ignore buggy results). Thus, some experiments had to be repeated which had lead to smaller amount of available CPU time than we expected. For all these reasons,

human action was frequently needed to understand whether a bug was due to solver, due to benchmark, or due to running scripts, and, of course, how to fix it appropriately.

Despite we tried to set up as strict rules as possible, still human action was needed in the choice of benchmarks (how to separate them into series/categories; how many benchmarks to take for the second stage?; how many benchmarks to generate from each random generator?). The human action was especially hard to implement since one of the organizers submitted his own solver and benchmarks, another one submitted benchmarks (though not qualified for the awards), and the remaining one planned to submit a solver (but did not). Therefore, Laurent Simon had to make a lot of decisions almost alone while was extremely busy with actually running the competition.

Also a wrong decision has been made concerning the selection of benchmarks *from* series. We expected that larger benchmarks of the same series should be harder, and therefore decided not to run a solver on larger benchmarks if it failed on smaller benchmarks of the same series. However, our conjecture was false (especially for industrial problems, where BMC SAT-checking of depth n can be harder than depth $n + 1$, and for some of the random generators, where the conjecture can be true for the median CPU time over a lot of formulas, but false for just 4 formulas), and it produced completely wrong decisions concerning mixed SAT/UNSAT series (clearly, any incomplete solver fails all (smaller) unsatisfiable benchmarks and thus is not run on (larger) satisfiable ones). We had to fix the effect of this either by dropping SAT/UNSAT series from the consideration for incomplete solvers, or by running the experiments on all benchmarks (possibly wasting CPU time).

Two other difficulties we encountered were (briefly):

- SAT-Ex scripts had to be tuned during the competition, e.g., to present results according to the competition rules, and to process solvers and benchmarks differently in different categories;
- the lack of live action (the system was not automated enough to put everything on the web without the help of the organizers who were extremely busy; also the results would be quite misleading if putted on the web immediately because once a bug had been eventually found, it changed the picture of the competition).

However, despite of the problems emphasized above, we treat the whole competition as a success in many aspects.

5.2. About the next competition

Because any of the SOTA contributors is important for SAT research, we are thinking about delivering a SOTA contributor certificate for the next competition. But, to adopt the SOTA system for awarding solvers, we need a better classification of available SAT benchmarks as in TPTP. Let us emphasize some differences between the CASC competition (where SOTA ranking *is* the rule) and the SAT2002 competition. Differences are essentially due to the different level of maturity of these competitions:

1. In the CASC competition, most instances are part of the TPTP library, so they are well known and classified. This is not the case for the SAT competition since we received a lot of completely new benchmarks for running the competition.
2. The ranking proposed above is made on *Specialist Problem Classes*, whose granularity is finer than our simple (Industrial, Handmade, Randomly generated) partition (for instance, the pigeon-hole problem may occur as an industrial problem ... How can we classify such a benchmark?).
3. We used a heuristic to save CPU time, so not all systems were run on all instances.

Many other improvements are possible for the next competition. Still it seems like human action is unavoidable. To make it more fair and less time-consuming for the organizers, we propose to appoint a board of *judges* similarly to CASC. When the competition rules do not give an explicit answer, it is up to the judges to decide what to do. They can also play a role for instance in the partition of benchmarks.

For the **future competitions** we propose the following:

- Allow more time for submitters to experiment with their solvers (prior to submission) on one of the *actual* computers of the competition.
- Make clear which version of a solver is used. Sometimes, under the same name, quite different algorithms or techniques are used (e.g., Berkmin 56 that is a single engine solver whose results were published in [21] and Berkmin 62 that was the 2 engines solver submitted to the competition). This would prevent a spectator from a suspicion when an instance is solved by a particular solver as reported in a previously published paper but not during the competition.
- Make competition more automatic both technically (better scripts) and semantically (more strict rules with no exceptions). This could allow to make the results online as soon as they are available, for instance to allow solver submitters to check the behavior of their solver during the first stage.
- After benchmarks are selected, every solver is run on every benchmark even if it seems a waste of CPU time (however, solver submitters could be allowed to submit their solvers only for some categories of benchmarks).
- Run the winners of the previous competition anyway (or the SOTA contributors). If an old winner wins a category, no award is given for this category. The same principle applies for the 2nd stage benchmarks.
- Limit the number of submissions per author groups: if one technique performs well in a category, it is likely that all its variants perform equally well (see, e.g., dlm-sat1/2/3 in 1st stage random SAT category). Then the second stage is biased. One solution is to limit the number of variants and to qualify only the best one for the final stage.
- Provide right now the scripts/programs used to check input/output/instances format to delegate that step to the submitters. (Then a fully automatic machinery can reject submissions not respecting the formats.)

Some other points, more or less prospective, are possible:

- Classify new benchmarks accordingly to the performances of previous year SOTA contributors. For instance, "*easy*" benchmarks (according to SOTA) could thus be discarded, freeing CPU time.
- Discuss the notion of series and how to score solvers on series. For instance, if many benchmarks in a series are easy, then most of solvers would have solved the series. This can be resolved for instance by giving the solvers points according to their performance for each particular series (and not 1 vs. 0 as it was in SAT 2002 competition). For example, the series winner could get 5 points, the next solver could get 4, etc.
- More prospective, one can use only a timeout per series instead of a timeout per instance (maybe as a new category). A solver able to solve quickly the first instances of a series will have more time to solve the remaining instances. Beware here: the order in which the instances are provided to the solver matters, and it is not easy to determine the best order.
- At last, we could use a larger cluster of machines, as the ones that they use for VLSI/CAD applications and simulations, where very large empirical evaluations are often performed. For instance, the new version of `bookshelf.exe` [9] should allow to use hundreds of identical computers with a simple-web interface (automatic report, ...). However, modifications are needed to merge the SAT-Ex architecture into this cluster interface.

The final point is to run the next competition only before the SAT Symposium (only publishing results and giving awards at the meeting) ...

6. Conclusions

The competition revealed many expected results as well as a few surprising ones. First of all, *incomplete solvers* appeared to be much weaker than complete ones. Note that no incomplete solver won any category of satisfiable formulas while these categories were intended rather for them than for complete solvers (note that in theory randomized one-sided error algorithms accept potentially more languages than deterministic ones). In Industrial and Handmade categories only one incomplete solver (UnitWalk) was in the top five. This can be due to the need of specific tuning for local search algorithms (noise, length of walk, etc.) and, probably, to the lack of new ideas for the use of randomization (except for local search). If automatic tuning (for similar benchmarks) of incomplete solvers is possible, how to incorporate it in the competition?

On Industrial and Handmade benchmarks, *zchaff and algorithms close to it* (Berkmin and limmat) were dominating. On the other hand, the situation on *randomly generated instances* was quite different. These algorithms were not even in the top five! Also, randomly generated satisfiable instances were the only category where incomplete algorithms were competitive (four of them: dlmsat1(2,3) and UnitWalk, were in the top five).

Concerning the unsatisfiable instances, the top five list is also looking quite differently to other categories.

In fact, only two solvers appeared in all the top five lists for the three categories Industrial/Handmade/Random: a *non*-zchaff-like complete solver 2clseq for SAT+UNSAT and an incomplete solver UnitWalk for SAT. However, they did not win anything. The (unsurprising) conclusion is that specialized solvers indeed perform better on the classes of benchmarks they are specialized for. Also it confirms that our choice of categories was right. But maybe an award should be given to algorithms performing uniformly on all kinds of instances (while some part of the community was against an "overall winner" for the present competition).

Another conclusion of the competition is that almost all benchmarks that remained *unsolved within 40 minutes* on P-III-450 (or a close number of CPU cycles on a faster machine) have not been solved in 6 hours either. This can be partially due to the fact that few people experimented with the behaviour of their solvers for that long. Note that the greatest number of second stage benchmarks was solved in Industrial–SAT+UNSAT category, the one where probably the greatest number of experiments is made by the solvers authors. Also many solvers crashed on huge formulas (probably due to the lack of memory).

It is no surprise that the *smallest unsolved unsatisfiable* benchmark (xor-chain instance by L. Zhang) belongs to Handmade category. In fact, many of unsatisfiable benchmarks in this category are also very small. However, it seems like all these benchmarks are hard only for resolution (and hence DP- and DLL-like algorithms) where exponential lower bounds are known for decades (see, e.g., [56,57]). Therefore, if non-resolution-based complete algorithms come, these benchmarks will be probably easy for them. For example, LSAT (fixed version) and eqsatz (not participated) which employ equality reasoning can easily solve parity32 instances that remained unsolved in the competition.

On the other hand, the *smallest unsolved* (provably) *satisfiable* benchmark (hgen2 instance by Edward A. Hirsch) is made using a random generator. Other small hard satisfiable benchmarks also belong to Random category. These benchmarks are much larger than hard unsatisfiable ones (5250 vs. 844 literal occurrences). This is partially due to the fact that no exponential lower bounds are known for DPLL-like algorithms for *satisfiable* formulas (in fact, the only such lower bounds we know for other SAT algorithms are of [23]). In contrast, no random generator was submitted for (provably) unsatisfiable instances. (Of course, some of the handmade unsatisfiable instances can be generated using random structures behind them; however, this does not give a language not known to be in coRP (and even in ZPP).) Note that the existence of an efficient generator of a *coNP-complete* language would imply NP = coNP (random bits form a short certificate of membership).

Probably, at the end, the main thing about competition is that it attracted completely new solvers (e.g., 2clseq and limmat) and a lot of new benchmarks.

Some challenging questions, drawn from the conclusion, are:

1. Construct a *random* generator for a "hard" language of provably *unsatisfiable* formulas.

2. Design an incomplete algorithm that would outperform complete algorithms (on satisfiable formulas), or explain why it is not possible.
3. Construct *satisfiable* formulas giving exponential lower bounds for the worst-case running time of *DPLL-like* algorithms.[10]
4. For the competition: how to request/represent a certificate for "unsatisfiable" answers without violating the rights of non-DPLL-like algorithms?

Acknowledgements

We would like to thank John Franco, Michal Kouril, and the University of Cincinnati for providing us the computers and local maintenance. We also thank Hans van Maaren and the computational logic group of the Technical University of Delft for providing the awards. Of course, not to forget the many benchmarks and solvers authors. We are also very grateful to all SAT2002 conference participants for their feedback.

References

[1] P.A. Abdulla, P. Bjesse and N. Eén, Symbolic reachability analysis based on SAT-solvers, in: *Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2000)* (2000).
[2] F. Bacchus, Enhancing Davis Putnam with extended binary clause reasoning, in: *Proceedings of National Conference on Artificial Intelligence (AAAI-2002)* (2002).
[3] F. Bacchus, Exploring the computational tradeoff of more reasoning and less searching, in: [49, pp. 7–16] (2002).
[4] L. Baptista and J.P. Marques-Silva, Using randomization and learning to solve hard real-world instances of satisfiability, in: *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP)* (2000).
[5] R.J.J. Bayardo and R.C. Schrag, Using CSP look-back techniques to solve real-world SAT instances, in: *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)* (AMS, Providence, RI, 1997) pp. 203–208.
[6] A. Biere, A. Cimatti, E.M. Clarke, M. Fujita and Y. Zhu, Symbolic model checking using SAT procedures instead of BDDs, in: *Proceedings of Design Automation Conference (DAC'99)* (1999).
[7] M. Buro and H.K. Büning, Report on a SAT competition, Bulletin of the European Association for Theoretical Computer Science 49 (1993) 143–151.
[8] C.-M. Li, B. Jurkowiak and P.W. Purdom Jr, Integrating symmetry breaking into a DLL procedure, in: [49, pp. 149–155] (2002).
[9] A.E. Caldwell, A.B. Kahng and I.L. Markov, Toward CAD-IP reuse: The MARCO GSRC bookshelf of fundamental CAD algorithms, IEEE Design and Test (May 2002) 72–81.
[10] P. Chatalic and L. Simon, Multi-resolution on compressed sets of clauses, in: *Twelth International Conference on Tools with Artificial Intelligence (ICTAI'00)* (2000) pp. 2–10.
[11] S.A. Cook, The complexity of theorem-proving procedures, in: *Proceedings of the Third IEEE Symposium on the Foundations of Computer Science* (1971) pp. 151–158.

[10] The question has been partially resolved in M. Alekhnovich, E.A. Hirsch, D. Itsykson, Exponential lower bounds for the running time of DPLL algorithms on satisfiable formulas, in: *Proceedings of ICALP 2004*. Springer, to appear.

[12] F. Copty, L. Fix, E. Giunchiglia, G. Kamhi, A. Tacchella and M. Vardi, Benefits of bounded model checking at an industrial setting, in: *Proc. of CAV* (2001).
[13] E. Dantsin, A. Goerdt, E.A. Hirsch, R. Kannan, J. Kleinberg, C. Papadimitriou, P. Raghavan and U. Schöning, Deterministic $(2 - 2/(k + 1))^n$ algorithm for k-SAT based on local search, Theoretical Computer Science 189(1) (2002) 69–83.
[14] M. Davis, G. Logemann and D. Loveland, A machine program for theorem proving, Communications of the ACM 5(7) (1962) 394–397.
[15] M. Davis and H. Putnam, A computing procedure for quantification theory, Journal of the ACM 7(3) (1960) 201–215.
[16] O. Dubois, P. André, Y. Boufkhad and J. Carlier, SAT versus UNSAT, in: [29, pp. 415–436] (1996).
[17] O. Dubois and G. Dequen, A backbone-search heuristic for efficient solving of hard 3-SAT formulae, in: *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI'01)*, Seattle, WA (2001).
[18] M.D. Ernst, T.D. Millstein and D.S. Weld, Automatic SAT-compilation of planning problems, in: [28, pp. 1169–1176] (1997).
[19] F. Aloul, A. Ramani, I. Markov and K. Sakallah, Solving difficult SAT instances in the presence of symmetry, in: *Design Automation Conference (DAC)*, New Orleans, LO (2002) pp. 731–736.
[20] J.W. Freeman, Improvements to propositional satisfiability search algorithms, Ph.D. thesis, Departement of Computer and Information Science, University of Pennsylvania, Philadelphia, PA (1995).
[21] E. Goldberg and Y. Novikov, BerkMin: A fast and robust SAT-solver, in: *Design, Automation, and Test in Europe (DATE '02)* (2002) pp. 142–149.
[22] C.P. Gomes, B. Selman and H. Kautz, Boosting combinatorial search through randomization, in: *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI'98)*, Madison, WI (1998) pp. 431–437.
[23] E.A. Hirsch, SAT local search algorithms: Worst-case study, Journal of Automated Reasoning 24(1/2) (2000) 127–143. Also reprinted in *Highlights of Satisfiability Research in the Year 2000*, Frontiers in Artificial Intelligence and Applications, Vol. 63 (IOS Press, 2000).
[24] E.A. Hirsch, New worst-case upper bounds for SAT, Journal of Automated Reasoning 24(4) (2000) 397–420. Also reprinted in *Highlights of Satisfiability Research in the Year 2000*, Frontiers in Artificial Intelligence and Applications, Vol. 63 (IOS Press, 2000).
[25] E.A. Hirsch and A. Kojevnikov, UnitWalk: A new SAT solver that uses local search guided by unit clause elimination, Annals of Mathematics and Artificial Intelligence 43 (2005) 91–111.
[26] J.N. Hooker, Needed: An empirical science of algorithms, Operations Research 42(2) (1994) 201–212.
[27] J.N. Hooker, Testing heuristics: We have it all wrong, Journal of Heuristics (1996) 32–42.
[28] IJCAI97, *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI'97)*, Nagoya, Japan (1997).
[29] D. Johnson and M. Trick (eds.), *Second DIMACS Implementation Challenge: Cliques, Coloring and Satisfiability*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 26 (American Mathematical Society, 1996).
[30] H. Kautz and B. Selman (eds.), *Proceedings of the Workshop on Theory and Applications of Satisfiability Testing (SAT2001)*, *LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT 2001)* (Elsevier Science, 2001).
[31] H.A. Kautz and B. Selman, Planning as satisfiability, in: *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI'92)* (1992) pp. 359–363.
[32] H.A. Kautz and B. Selman, Pushing the envelope: Planning, propositional logic, and stochastic search, in: *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI'96)* (1996) pp. 1194–1201.
[33] E. Koutsoupias and C.H. Papadimitriou, On the greedy algorithm for satisfiability, Information Processing Letters 43(1) (1992) 53–55.

[34] O. Kullmann, First report on an adaptive density based branching rule for DLL-like SAT solvers, using a database for mixed random conjunctive normal forms created using the Advanced Encryption Standard (AES), Technical Report CSR 19-2002, University of Wales Swansea, Computer Science Report Series (2002). (Extended version of [36].)

[35] O. Kullmann, Investigating the behaviour of a SAT solver on random formulas, Annals of Mathematics and Artificial Intelligence (2002).

[36] O. Kullmann, Towards an adaptive density based branching rule for SAT solvers, using a database for mixed random conjunctive normal forms built upon the Advanced Encryption Standard (AES), in: [49] (2002).

[37] C.-M. Li, A constrained based approach to narrow search trees for satisfiability, Information Processing Letters 71 (1999) 75–80.

[38] C.-M. Li, Integrating equivalency reasoning into Davis–Putnam procedure, in: *Proceedings of the 17th National Conference in Artificial Intelligence (AAAI'00)*, Austin, TX (2000) pp. 291–296.

[39] C.-M. Li and Anbulagan, Heuristics based on unit propagation for satisfiability problems, in: [28, pp. 366–371] (1997).

[40] I. Lynce and J.P. Marques Silva, Efficient data structures for backtrack search SAT solvers, in: [49] (2002).

[41] I. Lynce, L. Baptista and J.P. Marques Silva, Stochastic systematic search algorithms for satisfiability, in: [30] (2001).

[42] J.P. Marques-Silva and K.A. Sakallah, GRASP – A new search algorithm for satisfiability, in: *Proceedings of IEEE/ACM International Conference on Computer-Aided Design* (1996) pp. 220–227.

[43] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang and S. Malik, Chaff: Engineering an efficient SAT solver, in: *Proceedings of the 38th Design Automation Conference (DAC'01)* (2001) pp. 530–535.

[44] F. Okushi and A. Van Gelder, Persistent and quasi-persistent lemmas in propositional model elimination, in: *(Electronic) Proc. 6th Int'l Symposium on Artificial Intelligence and Mathematics* (2000).; Annals of Mathematics and Artificial Intelligence 40(3–4) (2004) 373–402.

[45] R. Ostrowski, E. Grégoire, B. Mazure and L. Sais, Recovering and exploiting structural knowledge from CNF formulas, in: *Proc. of the Eighth International Conference on Principles and Practice of Constraint Programming (CP'2002)*, Ithaca, NY (2002).

[46] R. Paturi, P. Pudlák and F. Zane, Satisfiability coding lemma, in: *Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science, FOCS'97* (1997) pp. 566–574.

[47] S. Prestwich, A SAT approach to query optimization in mediator systems, in: [49, pp. 252–259] (2002).

[48] S.D. Prestwich, Randomised backtracking for linear pseudo-Boolean constraint problems, in: *Proceedings of Fourth International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimisation Problems* (2002).

[49] SAT2002, *Fifth International Symposium on the Theory and Applications of Satisfiability Testing*, Cincinnati, OH (2002).

[50] R. Schuler, U. Schöning, O. Watanabe and T. Hofmeister, A probabilistic 3-SAT algorithm further improved, in: *Proceedings of 19th International Symposium on Theoretical Aspects of Computer Science, STACS 2002* (2002).

[51] B. Selman, H.A. Kautz and B. Cohen, Noise strategies for improving local search, in: *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI'94)*, Seattle (1994) pp. 337–343.

[52] B. Selman, H. Levesque and D. Mitchell, A new method for solving hard satisfiability problems, in: *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI'92)* (1992) pp. 440–446.

[53] Y. Shang and B.W. Wah, A discrete Lagrangian-based global-search method for solving satisfiability problems, Journal of Global Optimization 12(1) (1998) 61–99.

[54] L. Simon and P. Chatalic, SATEx: a Web-based framework for SAT experimentation, in: [30] (2001); `http://www.lri.fr/~simon/satex`.

[55] G. Sutcliff and C. Suttner, Evaluating general purpose automated theorem proving systems, Artificial Intelligence 131 (2001) 39–54.

[56] G.S. Tseitin, On the complexity of derivation in the propositional calculus, in: *Structures in Constructive Mathematics and Mathematical Logic, Part II*, ed. A.O. Slisenko (Consultants Bureau, New York, 1970) pp. 115–125. Translated from Russian.

[57] A. Urquhart, Hard examples for resolution, Journal of the Association for Computing Machinery 34(1) (1987) 209–219.

[58] A. Van Gelder, Autarky pruning in propositional model elimination reduces failure redundancy, Journal of Automated Reasoning 23(2) (1999) 137–193.

[59] A. Van Gelder, Extracting (easily) checkable proofs from a satisfiability solver that employs both preorder and postorder resolution, in: *Seventh Int'l Symposium on AI and Mathematics*, Fort Lauderdale, FL (2002).

[60] A. Van Gelder, Generalizations of watched literals for backtracking search, in: *Seventh Int'l Symposium on AI and Mathematics*, Fort Lauderdale, FL (2002).

[61] A. Van Gelder and F. Okushi, Lemma and Cut strategies for propositional model elimination, Annals of Mathematics and Artificial Intelligence 26(1–4) (1999) 113–132.

[62] A. Van Gelder and Y.K. Tsuji, Satisfiability testing with more reasoning and less guessing, in: [29, pp. 559–586] (1996).

[63] M. Velev and R. Bryant, Effective use of Boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors, in: *Proceedings of the 38th Design Automation Conference (DAC '01)* (2001) pp. 226–231.

[64] J. Warners and H. van Maaren, Solving satisfiability problems using elliptic approximations: Effective branching rules, Discrete Applied Mathematics 107 (2000) 241–259.

[65] H. Zhang, SATO: An efficient propositional prover, in: *Proceedings of the International Conference on Automated Deduction (CADE'97)*, Lecture Notes in Artificial Intelligence, Vol. 1249 (1997) pp. 272–275.

[66] H. Zhang and M.E. Stickel, An efficient algorithm for unit propagation, in: *Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics (AI-MATH'96)*, Fort Lauderdale, FL (1996).

[67] L. Zhang, C.F. Madigan, M.W. Moskewicz and S. Malik, Efficient conflict driven learning in a Boolean satisfiability solver, in: *International Conference on Computer-Aided Design (ICCAD'01)* (2001) pp. 279–285.

[68] L. Zheng and P.J. Stuckey, Improving SAT using 2SAT, in: *Proceedings of the Twenty-Fifth Australasian Computer Science Conference (ACSC2002)*, ed. M.J. Oudshoorn, Melbourne, Australia (2002).

Aggregating Interval Orders by Propositional Optimization

Daniel Le Berre, Pierre Marquis, and Meltem Öztürk*

Université Lille Nord de France, F-59000 Lille, France
Université d'Artois, CRIL, F-62307 Lens, France
CNRS, UMR 8188, F-62307 Lens, France
{leberre, marquis, ozturk}@cril.univ-artois.fr

Abstract. Aggregating preferences for finding a consensus between several agents is an important issue in many fields, like economics, decision theory and artificial intelligence. In this paper we focus on the problem of aggregating interval orders which are special preference structures allowing the introduction of tresholds for the indifference relation. We propose to solve this problem by first translating it into a propositional optimization problem, namely the Binate Covering Problem, then to solve the latter using a MAX-SAT solver. We discuss some properties of the proposed encoding and provide some hints about its practicability using preliminary experimental results.

Keywords: Interval orders, preference modelling and aggregation, propositional reasoning, Boolean optimization

1 Introduction

Aggregating preferences for finding a consensus between several agents is an important issue in many fields, like economics, decision theory, and artificial intelligence. Given the preferences of a set of agents (or voters) over a set of alternatives (or candidates), where preferences are generally formulated as binary relations such as strict preference, indifference, etc., preference aggregation aims at determining a collective preference relation representing as much as possible the individual preferences.

However many works have shown through paradoxes and impossibility theorems that preference aggregation is not an easy task, the famous ones are Condorcet's paradox [3], Arrow's theorem [2] .

A common approach is to consider a preference relation as a complete preorder (i.e., a reflexive and transitive relation). In the above results each voter is supposed to present a complete preoder over the set of alternatives. However, such a model for preferences does not prove adequate to all situations, and other models (generalizing the complete preorder one) have been pointed out. In particular, different structures have been introduced for defining thresholds as in the famous example given by Luce [10] about a cup

* This work has been supported in part by two Projects: ANR-05-BLAN-0384 and PEPS-09 37. This support is gratefully acknowledged by the autors. The authors also thank anonymous referees for their helpful comments and suggestions.

of coffee. Indeed, in contrast to the strict preference relation, the indifference relation induced by such structures is not necessarily transitive. Semiorders may form the simplest class of such structures and they appear as a special case of interval orders. The axiomatic analysis of what we call now interval orders has been given by Wiener [16], then the term "semiorders" has been introduced by Luce [10] and many results about their representations are available in the literature (for more details see [5,13]). Roughly speaking, within an interval order, alternative x_1 is strictly preferred to alternative x_2 if and only if the evaluation of x_1 is greater than the evaluation of x_2 plus a threshold. It is easy to see that preorders are special cases of interval orders where the value of threshold is fixed to zero.

In this paper we consider the interval order aggregation problem; to solve it, we propose a method based on the Kemeny distance which makes use of a translation into the Binate Covering Problem [4]. More precisely, we consider the case where the preferences of voters are interval orders and we try to find a final interval order which will be "as close as possible" to the set of voter's preferences. Let us note that having an interval order as a result of an aggregation is not a drawback for pointing out an undominated alternative since it is known that when the asymmetric part of a binary relation is transitive, which is the case of interval orders, there is always at least one such undominated alternative [14]. Moreover it is natural to ask an interval order as a result when preferences of voters are interval orders. Finally, as we will show it, even when the input preferences are preorders, focusing on interval orders as outputs is a way to get an aggregation which is closer to the given preferences than when preorders are targeted (just because the set of all interval orders over the! alternatives is a superset of the set of all preorders over the alternatives).

2 Aggregation as Optimisation

In this paper, we consider a finite set of alternatives A on which preference relations are applied ($|A| = n$), we represent with $a, b, c, ...$ specific elements of A and $x_1, x_2, ...$ or $x, y, z, \ldots$ variables ranging over the set A. We have a finite set of voters $V = \{v_1, \ldots, v_m\}$ ($|V| = m$). Voters express their preferences by the help of two binary relations represented in an explicit way as n^2-matrices: the notation aP_ib (resp. aI_ib) means that the voter v_i prefers strictly alternative a to b (resp. is indifferent between a and b). $\#p(a, b)$ (resp. $\#i(a, b)$) is the number of voters v_i for whom aP_ib (resp. aI_ib) holds. We call a *profile*, the set of voter's preference relations and denote it by $X = \{\langle P_1, I_1\rangle, \langle P_2, I_2\rangle, \ldots, \langle P_m, I_m\rangle\}$; its size is in $\mathcal{O}(m.n^2)$.

The result of the agregation is also expressed by two relations that we denote by P and I (P^{-1} represents the inverse of P: $\forall x, y \in A,\ xP^{-1}y$ iff yPx). aPb (resp. aIb) means that alternative a is preferred to alternative b (resp. a and b are indifferent) in the resulting order. We denote it as $f(\langle P_1, I_1\rangle, \langle P_2, I_2\rangle, \ldots, \langle P_m, I_m\rangle) = \langle P, I\rangle$.

The pair $\langle P, I\rangle$ is called a *preference structure* if and only if P is asymmetric, I is reflexive and symmetric, $P \cup I$ is complete and $P \cap I$ is empty. Such a pair is an interval order if and only if it is a preference structure and satisfies a property called Ferrers relation.[1]

[1] $\forall x, y, z, t \in A,\ xPy \wedge yIz \wedge zPt \Rightarrow xPt$.

Definition 1. *Let P and I be binary relations on $A \times A$, $\langle P, I\rangle$ is an interval order if and only if*

i) $P \cup I \cup P^{-1} = A \times A$ *(completeness) ,*
ii) $P \cap I =$ *(exclusivity),*
iii) P *is asymmetric,* I *is symmetric and reflexive,*
iv) $P.I.P \subset P$ *(Ferrers relation).*

The numerical representation of interval orders is as in the following:

Proposition 1. *[5] Let P and I be binary relations on $A \times A$, $\langle P, I\rangle$ is an interval order if and only if there exist a mapping g from A to $\mathbb{R}$ and a mapping q from $\mathbb{R}$ to $\mathbb{R}^+$ such that for any $x, y \in A$, we have:*
$xPy \Leftrightarrow g(x) > g(y) + q(g(y))$.
$xIy \Leftrightarrow g(x) \leq g(y) + q(g(y))$.

Interval orders are quasi-orders (i.e., orders with a transitive asymmetric part). Gibbard ([6]) has showed that Arrow's theorem can be generalized to the case of quasi-orders, hence we have this impossibility result for interval orders. Pirlot and Vincke ([13]) have focused also on this theorem with a special attention to interval orders. Before presenting this theorem we first need the following definitions in order to state it formally:

weak unanimity an aggregation procedure satisfies the weak unanimity condition if and only if, for all voters $v_i \in V$ and for all $a, b \in A$, $aP_ib \Longrightarrow aPb$;

non-dictatorship an aggregation procedure satisfies the non-dictatorship condition if and only if, for no voter $v_i \in V$ such that for all possible preferences of other voters and for all alternatives a and b $aP_ib \Longrightarrow aPb$;

independence of irrelevant alternatives an aggregation procedure satisfies the independence of irrelevant alternatives condition if and only if $\forall(\langle P_1, I_1\rangle, \ldots, \langle P_m, I_m\rangle)$, $(\langle P'_1, I'_1\rangle, \ldots, \langle P'_m, I'_m\rangle), \forall a, b \in A$,
$(\langle P_1, I_1\rangle, \ldots, \langle P_m, I_m\rangle)/\{a, b\} = (\langle P'_1, I'_1\rangle, \ldots, \langle P'_m, I'_m\rangle)/\{a, b\} \Longrightarrow$
$(\langle P, I\rangle), /\{a, b\} = (\langle P', I'\rangle)/\{a, b\}$
where $\langle P, I\rangle$ is the result on $(\langle P_1, I_1\rangle, \ldots, \langle P_m, I_m\rangle)$, and $\langle P, I\rangle/\{a, b\}$ is the restriction of $\langle P, I\rangle$ to $\{a, b\}$, etc.

Theorem 1 (Generalized Arrow's Theorem). *[13] If $|A| \geq 4$, if X is the set of all n-tuples of interval orders on A and if Y is the set of all interval orders on A, then there is no (X-Y)-aggregation procedure*[2] *satisfying simultaneously weak unanimity, non-dictatorship and independence conditions.*

Note that if all the considered relations are complete preorders, Theorem 1 is exactly Arrow's theorem with $|A| \geq 3$. We need four alternatives for interval orders because of the definition of Ferrers relation.

There exist a number of papers addressing the aggregation issue for binary relations as an optimization problem. Typically, a 0/1 linear program is targeted. Contrastingly,

[2] X represents here the set of voter's preferences and Y the resulting order.

in our approach, we associate to each profile of binary relations an instance of BCP, the so-called *Binate Covering Problem* [4], where the set of constraints is not any set of 0/1 linear inequations but a SAT instance. This problem has been studied for decades by the circuit community where it is important for logic synthesis (minimizing the number of components needed to perform a given operation).

From a theoretical standpoint, like 0/1 linear programming, BCP is an NP-hard optimization problem (and the associated decision problem is in NP) (see e.g. [12]). In practice, each clause can be translated into an equivalent 0/1 linear inequation, but the converse does not hold. The specific format of the constraints considered in BCP (compared to 0/1 linear programs) enables us to take advantage of the power of existing MAX-SAT solvers in order to solve its instances in a more efficient way from the practical side.

To our knowledge there is a limited number of studies related to the aggregation of interval orders. Pirlot and Vincke [13] have shown that the schemes that work well for complete preorders such as lexicographic procedure or Borda's sum of ranks do not lean themselves easily to the generalization with interval orders. They proposed two types of aggregation procedures: one consisting in aggregating numerical representations into a "global evaluation" function, and the other inspired from pairwise comparison methods.

In this paper we propose a hybrid approach consisting in finding an interval order being optimal in the sense of minimal Kemeny distance [8] to the input profile. Intuitively, ranking the alternatives according to Kemeny's rule can be seen as the best compromise since on average it gives the "closest" social preference to the individual preferences. Our idea can be summarized as in the following:

1. Determine all pairwise comparisons for which all the voters have the same opinion and build a partial order that preserves those comparisons.
2. Search within the set of feasible interval orders in order to find a closest one to the input profile.

The first step can be easily achieved the following way:
$\forall v_i \in V,\ \forall x, y \in A,\ xP_iy, \Longrightarrow xPy,$
$\forall v_i \in V,\ \forall x, y \in A,\ xI_iy, \Longrightarrow xIy.$

The resulting $\langle P, I\rangle$ is a partial order.

From partial order to interval orders Naturally this step provides in the majority of cases many interval orders. The worst case that we may expect is when the partial order provided in the first step is empty. In this case we have to find all the interval orders containing n objects (n being the cardinality of A). This case gives an idea on the number of interval orders that we may have. Stanley [15] has precised the number of interval orders with n elements; for this he has made use of relations between interval orders and hyperplanes arrangement. The coefficient of the following polynomial provides the number of interval orders:

$z = \sum_{k\geq 0} c_k \frac{x^k}{k!}$
$z = 1 + x + 3\frac{x^2}{2!} + 19\frac{x^3}{3!} + 195\frac{x^4}{4!} + 2831\frac{x^5}{5!} + 53703\frac{x^6}{6!} + 1264467\frac{x^7}{7!} + \ldots$

z is the unique power series satisfying $\frac{z'}{z} = y^2,\ z(0) = 1$ where $1 = y(2 - e^{xy})$. The value c_k of the serie z is the number of interval orders on k alternatives. This number grows exponentially on the number of alternatives: for instance, with just 7 alternatives we have more than one million interval orders. However, we will see in the following that we do not need to represent those interval orders explicitly. We denote by $\langle P^{(1)}, I^{(1)}\rangle$,$\langle P^{(2)}, I^{(2)}\rangle$, ... these interval orders.

Discriminating interval orders In our approach, the distance of an interval order $\langle P^{(i)}, I^{(i)}\rangle$ to the input profile X, $D(\langle P^{(i)}, I^{(i)}\rangle, X)$, will be calculated as the sum of its distance to each voter's order $\langle P_j, I_j\rangle$.
Let us denote this distance by $d(\langle P^{(i)}, I^{(i)}\rangle, \langle P_j, I_j\rangle)$:

$$D(\langle P^{(i)}, I^{(i)}\rangle, X) = \sum_{\langle P_j, I_j\rangle \in X} d(\langle P^{(i)}, I^{(i)}\rangle, \langle P_j, I_j\rangle)$$

The distance d is computed using the difference between pairwise comparisons in the following way:

$$d(\langle P^{(i)}, I^{(i)}\rangle, \langle P_j, I_j\rangle) = \sum_{(x,y)\in A^2} \delta_{\langle P^{(i)}, I^{(i)}\rangle, \langle P_j, I_j\rangle}(x, y)$$

$$\delta_{(\langle P^{(i)}, I^{(i)}\rangle, \langle P_j, I_j\rangle)}(x, y) = \begin{cases} p2p\ if & (xP^{(i)}y \text{ and } yP_jx) \text{ or } (yP^{(i)}x \text{ and } xP_jy) \\ 0 \quad if & (xP^{(i)}y \text{ and } xP_jy) \text{ or } (xI^{(i)}y \text{ and } xI_jy) \\ p2i\ otherwise & \end{cases}$$

Here $p2p$ and $p2i$ are nonnegative constant numbers. The rationale for this definition of d is to put a penalty when there is a discrepancy of preference relation between the comparison given by a voter and the one of the interval order. Naturally, a discrepancy of a strict preference (for instance xPy) to the inverse of this preference (yPx) is at least as problematic as a discrepancy of a strict preference (for instance xPy) to an indifference (xIy) for this reason we suggest that $p2p \geq p2i$. Even more one can impose the strict inequality ($p2p > p2i$) which will guarantee to have as a result aIb when the profile with three voters is aP_1b, aI_2b and bP_3a. Note that the distance used by Hudry ([7]) imposes $p2p = p2i = 1$ and provides as a result three interval orders (aPb, bPa and aIb) for this example.

We propose to represent the set of interval orders to be *implicitly* considered in the second step using propositional constraints (clauses). Then, computing the interval orders closest to the profile is encoded as minimizing an objective function. Accordingly, we reduce our interval order optimization problem to the BCP one.

3 Translation into the Binate Covering Problem

We first need propositional variables v_{xPy} and v_{xIy} to represent all pairs of the form xPy ($\forall x \neq y \in A$) and xIy ($\forall x, y \in A, x \leq y$). As a consequence, $n^2 - n + \frac{n\times(n-1)}{2} + n = \frac{3\times n^2 - n}{2}$ variables must be considered. For instance, for 4 alternatives, we need 22 propositional variables. For 16 alternatives, we need 376 propositional variables.

3.1 Implicit representation of interval orders

Structural constraints The following constraints express that the result of the aggregation must be an interval order. They do not depend on the voters.

- $P \cup I$ is complete: $\forall x < y \in A\ v_{xPy} \vee v_{xIy} \vee v_{yPx}$,
- P is asymmetric: $\forall x < y \in A,\ \neg(v_{xPy} \wedge v_{yPx}) \equiv \neg v_{xPy} \vee \neg v_{yPx}$,
- P and I are exclusive: $\forall x \neq y \in A,\ \neg(v_{xPy} \wedge v_{xIy}) \equiv \neg v_{xPy} \vee \neg v_{xIy}$,
- I is symmetric by construction because a single propositional variable v_{xIy} represents both xIy and yIx.
- I is reflexive: $\forall x \in A$, v_{xIx} is forced to be true,
- $P \cup I$ is Ferrers: $\forall x, y, z, t \in A, x \neq y, z \neq t, x \neq t, y \neq t, x \neq z\ (v_{xPy} \wedge v_{yIz} \wedge v_{zPt}) \Rightarrow v_{xPt}$,

Note that we need to generate $2n(n-1) + n + n(n-1)(n-2)^2 = n(n^3 - 5n^2 + 10n - 5)$ structural constraints plus the unit clauses needed to preserve unanimity (see below). For 4 alternatives, it means at least 76 constraints. For 16 alternatives, it means at least 47536 constraints. The $O(n^4)$ space required by the above encoding is clearly dominated by the cost of ensuring Ferrers condition.

Unanimity constraints Those additional constraints encode unanimity for both P and I. They are generated according to the votes. Since they force the truth value of some variables, they simplify in practice the computation of the best interval order.

- Unanimity for P: $\forall x \neq y \in A$, if $\#p(x, y) = |V|$ then xPy is forced to be true,
- Unanimity for I: $\forall x \neq y \in A$, if $\#i(x, y) = |V|$ then xIy is forced to be true.

3.2 Distance between interval orders and the profile

The coefficient associated to each variable is computed according the individual penalty δ defined earlier and the number of voters that disagree with the interval order.

- $\forall x, y \in A$, satisfying $I(x, y)$ entails that voters that strictly prefer x to y or y to x disagree with that fact, with a simple individual penalty of $p2i$. As a consequence, the coefficient of the variables is exactly $p2i(\#p(x, y) + \#p(y, x))$,
- $\forall x, y \in A$, satisfying $P(x, y)$ entails that voters that are indifferent between x and y disagree with that fact with a simple penalty of $p2i$, while the voters that strictly prefer y to x disagree with that fact with an individual penalty of $p2p$. So the coefficient of those variables is exactly $p2i\#i(x, y) + p2p * \#p(y, x)$.

The objective function of the binate covering problem associated with X is denoted by $score_X(\langle P, I\rangle)$ and is

$$\sum_{x \leq y \in A} p2i(\#p(x, y) + \#p(y, x))v_{xIy} + \sum_{x \neq y \in A} (p2i\#i(x, y) + p2p * \#p(y, x))v_{xPy}.$$

Thus the space needed to represent the objective function is in $\mathcal{O}(n^2.log_2(m))$. Interestingly, the space needed by the encoding (constraints and objective function)

is only logarithmic in the number of voters. This renders the approach feasible for a large number of voters. On the other hand, the space needed by the encoding is in $\mathcal{O}(n^4)$; considering that MAX-SAT solvers are currently able to solve *some* instances with millions of variables, it might be possible to solve aggregation problems up to roughly 40 alternatives (which leads to 2 millions of clauses using the above encoding).

The result of the aggregation step is any interval order which minimizes the value of the objective function. An important issue is to determine whether it makes sense to use of sophisticated SAT engine (or 0/1 linear program solver) to solve those specific BCP instances stemming from a translation from instances of the aggregation problem. [7] gave a positive answer to this query, by identifying the complexity of the following decision problem: SCORE:

Input: A finite profile X of binary relations $\langle P, I\rangle$ on A and a nonnegative integer k.
Question: Does there exist an interval order $\langle P, I\rangle$ on A such that $score_X(\langle P, I\rangle) \leq k$?

In a nutshell Hudry showed that SCORE is NP-complete as soon as the number of voters m is "sufficiently" large compared to the number n of alternatives, even in the restricted case when X consists of linear orders only, provided that $p2p = p2i = 1$. This justifies to take advantage of algorithms running in exponential time (as MAX-SAT solvers) in the worst case, since polynomial time ones are hardly expected.

Hudry's NP-hardness result extends easily to our framework when the parameters $p2p$ and $p2i$ are such that $p2p = p2i$ since linear orders are interval orders; on the other hand, the membership to NP of the SCORE probllem is obvious in our setting: in order to determine that an instance of this decision problem is positive, it is enough to guess a binary relation $\langle P, I\rangle$ on A (its size is $\mathcal{O}(n^2)$), then to check that it is an interval order (this can be easily achieved in polynomial time in the size of the relation), and finally to compute in polynomial time $score_X(\langle P, I\rangle)$ in order to compare it with k.

Our MAX-SAT algorithm for the BCP problem is a branch-and-bound algorithm. During the search, each time a (partial) assignment is found that satisfies all the constraints, the corresponding score is computed (each unassigned variable is set to 0) and a constraint which eliminates all the assignments leading to a greater bound is added, so that whenever a partial assignment leads to a score which is worse than this bound, a backtrack occurs. Its worst-case time complexity is simply exponential in the number of variables under consideration (hence linear in the size of X) and its space complexity is linear in the size of the constraints (hence quadratic in the size of X).

3.3 Examples

As a matter of illustration, let us consider the following examples. For these examples we suppose that $p2i = 1$ and $p2p = 2$.

Example 1. Consider first a case with 5 voters and 4 alternatives with the preferences of voters shown in Table 1.
These preferences of voters can be compactly represented in a matrix where $\forall x_i, x_j$, $P(x_i, x_j) = \alpha$ means that there are α voters who prefer alternative x_i to alternative

V_1	a	b	c	d
a	I	P	P	P
b	P^{-1}	I	I	P
c	P^{-1}	I	I	P
d	P^{-1}	P^{-1}	P^{-1}	I

V_2	a	b	c	d
a	I	P	I	P
b	P^{-1}	I	P^{-1}	P
c	I	P	I	P
d	P^{-1}	P^{-1}	P^{-1}	I

V_3, V_4, V_5	a	b	c	d
a	I	P	P	P
b	P^{-1}	I	I	P
c	P^{-1}	I	I	P
d	P^{-1}	P^{-1}	P^{-1}	I

Table 1. Pairwise comparisons on 4 alternatives given by 5 voters

x_j. Table 2 represents the matrix related to the previous example. Accordingly, this matrix contains all the information needed for running our aggregation procedure. Our method find as a result the following interval order: aPb, cPa, dPa, cPb, dPb, cPd (its distance to the profile is 12).

P	a	b	c	d
a	0	5	1	2
b	0	0	0	2
c	3	4	0	5
d	3	3	0	0

Table 2. The number of voters agreeing for a strict preference

Example 2. Here is a second example; Table 3 shows the pairwise comparisons given by three voters on three alternatives (a, b, c).

V_1	a	b	c
a	I	P	I
b	P^{-1}	I	P^{-1}
c	I	P	I

V_2	a	b	c
a	I	P	P
b	P^{-1}	I	I
c	P^{-1}	I	I

V_3	a	b	c
a	I	I	I
b	I	I	I
c	I	I	I

Table 3. The profile of Example 2

The result of our aggregation procedure provides a unique interval order as close as possible to the input profile. It is not a preorder (a is preferred to b and all the other comparisons are indifference), despite the fact that each preference relation in the input profile is a preorder.

3.4 More than one solution is often the case

Clearly enough, there is no guarantee in general that a unique interval order $\langle P, I\rangle$ exists, leading to a minimal value s^* for the objective function $score_X(\langle P, I\rangle)$. This problem is inherent to the fact that voters may have different preferences, and it may happen in very simple scenarios, for instance when A consists of two alternatives a and b and X consists of two interval orders $\langle P_1, I_1\rangle$ and $\langle P_2, I_2\rangle$ on A so that aP_1b and aI_2b: in

such a case, both $\langle P_1, I_1\rangle$ and $\langle P_2, I_2\rangle$ lead to the minimal value $s^* = p2i$, but not to the same sets of undominated alternatives. Nevertheless, this plurality is problematic since decisions made using only one of such optimal interval orders are not necessarily robust, in the sense that the choice of another optimal interval order could question them. Typically decisions are made by comparing alternatives or determining undominated ones. While robustness is a complex notion, a sufficient condition for a comparison to be robust is when it holds for every optimal interval order, and similarly an alternative is robustly undominated when it is undominated for all optimal interval orders. Formally, the following decision problems have to be considered: NEC-COMP(R):

Input: A finite profile X of binary relations $\langle P, I\rangle$ on A and two alternatives a, b from A.
Question: Is it the case that every interval order $\langle P, I\rangle$ on A satisfying $score_X(\langle P, I\rangle) = s^*$ is such that aRb? (where R = P or $R = I$)?

NEC-UNDOM:
Input: A finite profile X of binary relations $\langle P, I\rangle$ on A and an alternative a from A.
Question: Is it the case that for every interval order $\langle P, I\rangle$ on A such that $score_X(\langle P, I\rangle) = s^*$, we have $a(P \cup I)b$ for every $b \in A$?

Those decision problems are "mildly" hard, since they belong to the complexity class Θ_2^p, consisting of all decision problems which can be solved in deterministic polynomial time using logarithmically many calls to an NP oracle. In order to prove the membership of NEC-COMP(R) and NEC-UNDOM to Θ_2^p, we consider the complementary problems and show them in Θ_2^p as well (this class is closed under complementation). We have already seen that SCORE is in NP. Now, the value of s^* can be computed by binary searching it within the bounds 0 and $m.n^2.max(p2p, p2i)$ which is a (rough) upper bound of s^*, and has a value linear in the size of X since $max(p2p, p2i)$ is a constant. Hence, s^* can be computed in deterministic polynomial time using logarithmically many calls to an NP oracle (used to solve the SCORE instances encountered during the search, associated to the successive values of k) . Once this is done, it remains to guess a binary relation $\langle P, I\rangle$ on A using a last call to the NP oracle, check that it is an interval order such that $score_X(\langle P, I\rangle) = s^*$, and finally check that $a\bar{R}b$ (resp. that there exists a $b \in A$ such that bPa). We conjecture that NEC-COMP(R) and NEC-UNDOM are Θ_2^p-complete. Noticeably, when s^* is part of the input, the complexity of NEC-COMP(R) and NEC-UNDOM falls down to coNP. From the practical side, when several instances of NEC-COMP(R) or NEC-UNDOM sharing the same profile X are to be solved, it can prove useful to compute s^* once for all during a pre-processing phase, then to exploit it in order to solve those instances in a more efficient way.

4 Some Theoretical Results

We analyze here some expected properties for aggregation procedures such as respect of unanimity, independance, majority, etc., and our objective is to determine whether or

not our approach satisfies some of them. We begin by the properties at work in Arrow's theorem:

Universality An aggregation procedure is universal if it accepts all configurations for the input profile. Since the input of our procedure can be any finite set of interval orders, we can conclude that our procedure is universal.

Transitivity Arrow's theorem imposes the transitivity of the preference and the indifference relation. Our procedure provides an interval order which has a transitive preference relation P but the indifference relation I is not necessarily transitive. However as we mentioned in the introduction, in order to find an undominated alternative, transitivity of P is enough.

Weak-unanimity Our procedure satisfies the weak unanimity condition since unanimity is imposed by our formulation as a hard constraint to be respected.

Non-dictatorship Our procedure obviously satisfies the non-dictatorship condition.

Independence Our procedure does not satisfy the condition of independence of irrelevant alternatives: let us show it on a new example. The set of alternatives is $A = \{a, b, c, d\}$ and we have two different profiles X and X' which have the same votes on the subset $A' = \{c, d\}$ of A:

Example 3. Table 3 shows the compact matrix of each profile. Our procedure concludes that for the profile X there are two optimum solutions, in the first one c is indifferent to d and in the second one d is preferred to c. However, even if the profile X' has the same votes for the comparison between c and d, our procedure concludes for X' that c is preferred to d.

X	a	b	c	d
a	0	0	0	1
b	0	0	6	2
c	4	0	0	6
d	8	6	4	0

X'	a	b	c	d
a	0	3	0	0
b	3	0	0	3
c	8	1	0	6
d	2	2	4	0

Table 4. The compact matrix of profile X and X' 4

We consider now some other properties that an aggregation procedure should preferably satisfy.

Anonymity The result of the aggregation depends only on the preferences of voters (and for instance, not on the age, sex or seniority of candidates). Let $\mathcal{P}$ be the set of permutations of A, π one element of $\mathcal{P}$. We denote by $\pi(R)$, the binary relation such as $\pi(a)\pi(R)\pi(b) \iff aRb$. An aggregation procedure is anonymous if and only if $\forall \pi \in \mathcal{P}$,$f(R_1, R_2, \ldots, R_m) = \pi(f(\pi(R_1), \pi(R_2), \ldots, \pi(R_m)))$.
It is easy to see that our procedure is anonymous.

Loyalty If there is just one voter the procedure must provide as a result the same preference as her: $m = 1 \Longrightarrow f(R_1) = \{a \in A : aR_1b, \forall b \in A\}$. Again, it is easy to check that our procedure satisfies the loyalty condition

Majority condition If there is a majority of voters who prefers a to b then the result of the aggregation procedure must agree with this comparison: f satisfies the *majority condition* if and only if $\forall (R_1, R_2, \ldots, R_m) \in X,\ \forall a, b \in A$

$$\#p(a,b) > \#p(b,a) \Longrightarrow aPb,$$
$$\#p(a,b) = \#p(b,a) \Longrightarrow aIb.$$

Our aggregation procedure does not satisfy the majority condition as the following example shows it.

Example 4. Table 5 gives the number of votes for pairwise comparisons between four alternatives given by 11 voters

P	a	b	c	d
a	0	6	5	1
b	2	0	8	3
c	6	2	0	1
d	4	3	3	0

Table 5. The number of voters agreeing for a strict preference

Even if the majority of voters prefer c to a, the result of our procedure concludes that a is preferred to c (the output is the interval order such that aPb, aPc, aId, aIa, bPc, bId, bIb, cId, cIc, dId and its distance to the profile is 39).

5 Conclusion

In this paper, we have presented an optimization-based approach to interval orders aggregation. In this approach, to every profile of interval orders, one associates an instance of a propositional optimization problem (namely the Binate Covering Problem); solving the latter gives in a straightforward way an interval order (the "closest" to the input profile in some sense), which is considered as the aggregation looked for. Among other things, we have computed an upper bound of the size of the BCP instance associated to every profile (showing that it is only logarithmic in the number of voters), identified some properties satisfied (or not) by the aggregation approach. An interesting feature of such an optimization-based approach to aggregation is that it can be easily tuned to fit with other preference structures (e.g. preorders, semiorders, etc.). Indeed, it is enough to point out the corresponding hard constraints. Investigating in more depth such extensions is a perspective for further research.

The Binate Covering Problem can be seen as a very specific case of an Integer Linear Program in which case efficient ILP frameworks exist (e.g. CPLEX). However, it looks that tools dedicated to Boolean reasoning are better suited to solve such problems: Weighted Partial MAX SAT [1] and Pseudo Boolean Optimization [11] engines are currently receiving a lot of attention since international evaluations are organized regularly and many systems are freely available for the research community.

We designed a proof of concept tool based on the SAT4J library[9], a library of Boolean search engines dedicated to solving SAT, MAX SAT and Pseudo Boolean problems. That tool can be downloaded from `http://sat4j.ow2.org/`.

In order to have an idea of the applicability of our approach on a real scenario, we used the publicly available results of the SAT RACE 2006[3]. It is a competitive event between 16 SAT solvers on a set of 100 benchmarks. Here each benchmark is a voter and each solver is an alternative. A given benchmark b prefers the SAT solver x to the SAT solver y iff x solved b faster than y. A given benchmark b is indifferent between the SAT solvers x and y iff none of x and y solved b or both of them solved b but with a roughly the same CPU time (the difference is less than 1 second). By definition, each vote is an interval order. Computing the aggregation of such votes means solving a binate covering problem with 376 variables and 47536 clauses. Our aggregator takes less than one second to generate the BCP from the compact matrix. SAT4J takes one second to find a solution, but fails to prove it is optimal even after running for several hours.

We plan to test several MAXSAT and Pseudo Boolean engines on aggregation of real interval orders instances (including LP based ones).

References

1. Josep Argelich, Chu-Min Li, Felip Many, and Jordi Planes. The first and second max-sat evaluations. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 4:251–278, 2008.
2. K.J. Arrow. *Social choice and individual values*. J. Wiley, New York, 1951. 2nd edition, 1963.
3. Marquis de Condorcet. *Essai sur l'application de l'analyse à la probabilité des décisions rendues à la pluralité des voix*. Imprimerie Royale, Paris, 1785.
4. Olivier Coudert. On solving covering problems. In *Design Automation Conference*, pages 197–202, 1996.
5. P.C. Fishburn. *Interval Orders and Interval Graphs*. J. Wiley, New York, 1985.
6. A. Gibbard. Social choice and the arrow conditions. *unpublished*, 1969.
7. O. Hudry. Np-hardness results for the aggregation of linear orders into median orders. *Annals of Operations Research*, 163:63–88, 2008.
8. J.G. Kemeny. Mathematics without numbers. *Daedalus*, 88:575–591, 1959.
9. Daniel Le Berre and Anne Parrain. SAT4J, a SATisfiability library for java. `http://www.sat4j.org`.
10. R.D. Luce. Semi-orders and a theory of utility discrimination. *Econometrica*, 24, 1956.
11. V. Manquinho and O. Roussel. The first evaluation of pseudo-boolean solvers (pb'05). *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 2:103–143, 2006.
12. CH. Papadimitriou and I. Steiglitz. *Combinatorial Optimization: algorithms and complexity*. Prentice-Hall, Englewood Cliffs, 1982.
13. M. Pirlot and Ph. Vincke. *Semi Orders*. Kluwer Academic, Dordrecht, 1997.
14. A.K. Sen. *Collective Choice and Social Welfare*. North Holland, Amsterdam, 1970.
15. R. Stanley. Hyperplanes arrangements, interval orders and trees. *Proc. Nat. Acad. Sci.*, 93:2620–2625, 1996.
16. N. Wiener. A contribution to the theory of relative position. *Proc. of Cambridge Philosophical Society*, 17:441–449, 1914.

[3] `http://fmv.jku.at/sat-race-2006/`

Résumé

Nos travaux de recherche se situent dans le cadre de la logique propositionnelle. Nous nous intéressons à la modélisation de problèmes dans ce formalisme et à leur résolution pratique. Nous présentons comme fil conducteur du document le problème de la gestion des dépendances entre logiciels. Nous montrons comment divers aspects de ce problème peuvent être modélisés comme des problèmes d'optimisation pseudo-booléens ou de type MaxSat, et indiquons aussi comment la logique du choix qualitatif peut être utilisée comme cadre de représentation de préférences complexes à prendre en compte pour gérer les dépendances. Nous présentons ensuite en détail les fonctionnalités de la bibliothèque libre Sat4j, permettant de résoudre divers problèmes de décision et d'optimisation en logique propositionnelle. Ses performances sont évaluées contre l'état de l'art dans chaque domaine (SAT, pseudo-booléen, MaxSat). Nous présentons enfin les motivations et le fonctionnement des diverses éditions de la compétition internationale de prouveurs SAT que nous avons organisées.

Mots-clés: Logique propositionnelle, résolution pratique de SAT, contraintes pseudo-booléennes, MaxSat, optimisation, gestion des dépendances, logiciel libre

Abstract

This work takes place in the area of propositional logic. We are interested in modeling and solving problems in such logic. We use the problem of software dependency management as central theme. We show how various aspects of that problem can be modeled as pseudo-boolean optimization or MaxSat problems. We exemplify the use of our qualitative choice logic to express complex preferences to be taken into account for dependency management. Then, we present in detail the features available in the open source library sat4j, that allows to solve both some decision and some optimization problems in propositional logic. Its efficiency is evaluated against state-of-the-art solvers in each domain (SAT, pseudo-boolean, MaxSat). Finally, we motivate the creation and the design decisions behind the internal SAT competition that we have organized.

Keywords: Propositional logic, practical resolution of SAT, pseudo-boolean constraints, MaxSat, optimization, dependency management, open source software