CORE
Metadata, citation and similar papers at core.ac.uk
Provided by INRIA a CCSD electronic archive server

SAT-MICRO: petit mais costaud !

Sylvain Conchon, Johannes Kanig, Stéphane Lescuyer

To cite this version:

Sylvain Conchon, Johannes Kanig, Stéphane Lescuyer. SAT-MICRO: petit mais costaud !. JFLA (Journées Francophones des Langages Applicatifs), INRIA, Jan 2008, Etretat, France. pp.91-106. inria-00202831

HAL Id: inria-00202831
https://hal.inria.fr/inria-00202831

Submitted on 8 Jan 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

SAT-MICRO : petit mais costaud !

Sylvain Conchon & Johannes Kanig & Stéphane Lescuyer

LRI, Université Paris-Sud, CNRS, Orsay F-91405
INRIA Futurs, ProVal, Orsay F-91893

Résumé

Le problème SAT, qui consiste à déterminer si une formule booléenne est satisfaisable, est un des problèmes NP-complets les plus célèbres et aussi un des plus étudiés. Basés initialement sur la procédure DPLL, les *SAT-solvers* modernes ont connu des progrès spectaculaires ces dix dernières années dans leurs performances, essentiellement grâce à deux optimisations : le retour en arrière non-chronologique et l'apprentissage par analyse des clauses conflits. Nous proposons dans cet article une étude formelle du fonctionnement de ces techniques ainsi qu'une réalisation en OCAML d'un *SAT-solver*, baptisé SAT-MICRO, intégrant ces optimisations ainsi qu'une mise en forme normale conjonctive paresseuse. Le fonctionnement de SAT-MICRO est décrit par un ensemble de règles d'inférence et la taille de son code, 70 lignes au total, permet d'envisager sa certification complète.

1. Introduction

Le problème de la satisfaisabilité d'une formule propositionnelle est un des problèmes NP-complets les plus célèbres et à ce titre il a été abondamment abordé dans la littérature. Depuis quelques années, les *SAT-solvers* ont été placés au cœur de nombreuses applications industrielles, notamment grâce aux progrès spectaculaires réalisés dans leurs performances. Basés au départ sur la procédure DPLL, les *SAT-solvers* d'aujourd'hui apportent plusieurs optimisations à cet algorithme comme le retour en arrière non-chronologique, l'apprentissage par l'analyse des conflits, la détection efficace des clauses unitaires, etc. Malheureusement, il y a souvent un double fossé à combler entre les avancées théoriques et les applications pratiques : celui du passage de la formalisation à l'implémentation et celui de la preuve formelle de l'implémentation.

Nous proposons dans cet article l'étude et la réalisation d'un *SAT-solver* moderne sous un nouvel angle, de manière à répondre aux points soulevés ci-dessus. Nous présentons une implémentation en OCAML, appelée SAT-MICRO, formalisée à partir d'un système de règles d'inférence. Ces règles, très proches de l'implémentation, sont aisément adaptables aux différentes optimisations mentionnées plus haut. Bien que s'appuyant sur une architecture moderne, SAT-MICRO n'est pas aussi compétitif que les *SAT-solvers* les plus performants d'aujourd'hui. Notre objectif est ici plus de certifier une implémentation, sans sacrifier complètement l'efficacité, que de réaliser un outil hautement performant. La proximité entre la formalisation et le code, ainsi que le style de programmation purement fonctionnel de notre implémentation, font que la taille de SAT-MICRO n'excède pas 70 lignes de code et nous a permis d'entreprendre une preuve formelle de ce système.

Nous présentons en section 2 un système de règles d'inférence modélisant la procédure DPLL ainsi qu'une implémentation basée directement sur ces règles. La section 3 étend ce système avec d'une part la technique de retour en arrière non-chronologique (*backjumping)* et d'autre part un mécanisme d'apprentissage par analyse de conflits. De plus, nous montrons dans la section 4 comment adapter les *SAT-solvers* présentés dans les sections précédentes afin de manipuler des formules en forme normale conjonctive équi-satisfaisable pour éviter l'explosion combinatoire inhérente à une telle transformation. Enfin, nous montrons dans la section 5 l'impact des optimisations présentées dans la section 3 sur les performances de SAT-MICRO.

2. La procédure DPLL

L'algorithme DPLL [7, 6], nommé d'après ses inventeurs Davis, Putnam, Logemann et Loveland, est la procédure la plus classique pour décider si une formule propositionnelle en forme normale conjonctive[1] (FNC) est satisfaisable ou non. Pour cela, l'algorithme essaye de construire une instanciation des variables propositionnelles qui rende la formule vraie. En principe, l'algorithme verifie toutes les 2^n possibilités d'instancier les variables, mais il utilise deux heuristiques intelligentes qui lui permettent d'aller plus vite :

- la *propagation des contraintes booléennes* : à chaque fois qu'une valeur de vérité est choisie pour une variable, la formule est simplifiée en conséquence : les littéraux devenus faux sont supprimés des clauses, et si une clause contient un littéral devenu vrai, toute la clause est supprimée ;
- la règle de la *clause unitaire* : si la formule contient une clause formée d'un seul littéral, la valeur de vérité de la variable de ce littéral est choisie de sorte qu'il soit vrai.

De cette manière, l'algorithme procède en attribuant une valeur de vérité à chaque variable propositionnelle appartenant à la formule, jusqu'à ce que l'un des deux événements suivants arrive :

- la formule simplifiée est la conjonction vide $\emptyset$; dans ce cas, une instanciation des variables a été trouvée qui rend vraie la formule de départ : elle est donc satisfaisable et l'algorithme s'arrête ;
- l'algorithme a atteint un état dans lequel la formule simplifée contient une clause vide (un *conflit*) : dans ce cas, l'algorithme effectue un retour en arrière à l'endroit le plus récent où il peut affecter une autre valeur de vérité à une variable.

Pour la notation des formules en FNC, nous adoptons les conventions suivantes :

- l'ordre des littéraux dans une clause, ou des clauses dans une conjonction, est sans importance ; nous écrirons $l \vee C$ pour décrire une clause qui contient au moins le littéral l et $\{l_1, l_2, l_3\}$ pour la clause formée des littéraux l_1, l_2 et l_3 ;
- une formule en FNC est écrite $C_1, \ldots, C_n$ où les C_i sont les clauses de la formule. Parfois, nous omettrons les accolades des clauses singletons : si Δ est un ensemble de clauses, Δ, l est une formule avec une clause qui ne contient que le littéral l.

2.1. DPLL avec des règles d'inférences

$$\textsc{Conflict}\ \frac{}{\Gamma \vdash \Delta, \emptyset} \qquad \textsc{Assume}\ \frac{\Gamma, l \vdash \Delta}{\Gamma \vdash \Delta, l} \qquad \textsc{Bcp}\ \begin{cases} \dfrac{\Gamma, l \vdash \Delta, C}{\Gamma, l \vdash \Delta, \bar{l} \vee C} \\ \dfrac{\Gamma, l \vdash \Delta}{\Gamma, l \vdash \Delta, l \vee C} \end{cases}$$

$$\textsc{Unsat}\ \frac{\Gamma, l \vdash \Delta \qquad \Gamma, \bar{l} \vdash \Delta}{\Gamma \vdash \Delta}$$

FIG. 1 – Une version abstraite de DPLL

La figure 1 montre le fonctionnement de l'algorithme DPLL, à l'aide de cinq règles d'inférences. Elles décrivent l'état de l'algorithme par le *séquent* $\Gamma \vdash \Delta$, où Γ est l'ensemble des littéraux supposés vrais, et Δ est la formule courante. Ces règles doivent être lues de bas en haut : l'état sous le trait est l'état *avant* l'application de la règle d'inférence. La règle CONFLICT correspond au cas où l'algorithme a trouvé la clause vide dans la formule. Cette branche de l'arbre de recherche étant terminée, l'algorithme doit revenir en arrière pour trouver une autre instanciation des variables. La règle ASSUME décrit

[1] *i.e.* une formule de la forme $\bigwedge_{i=1}^{n}(l_1 \vee \cdots \vee l_{k_i})$, où chaque l_j est un littéral, c'est-à-dire une variable propositionnelle ou sa négation.

l'heuristique des clauses unitaires : un littéral dans une clause unitaire doit être supposé vrai, sans quoi la formule serait immédiatement insatisfaisable.

Les règles BCP décrivent la propagation des contraintes propositionnelles imposées par les variables dans Γ. Si un littéral est supposé faux (sa négation est dans Γ), il peut être supprimé de la clause (première règle). Si une clause contient un littéral qui est supposé vrai, la clause peut être entièrement supprimée (deuxième règle).

Finalement, la règle UNSAT, qui doit se lire de bas en haut et de gauche à droite, décrit le choix d'une valeur de vérité pour un littéral de la formule. Il est d'abord supposé vrai (partie gauche), et si aucune instanciation satisfaisable n'a été trouvée (toutes les branches de l'arbre de gauche terminent avec la règle CONFLICT), l'algorithme revient par backtracking à cette règle UNSAT et essaye la branche droite, en supposant cette fois le littéral faux.

Cette procédure a été formalisée en Coq et prouvée correcte et complète. Les tailles de la spécification et des preuves sont respectivement d'environ 400 et 2000 lignes.

2.2. Une implémentation de DPLL en OCAML

Dans cette section, nous voulons montrer qu'il n'est pas difficile de traduire les règles d'inférence de la figure 1 vers du code OCAML qui leur soit proche. Nous appelons cette première implémentation « naïve », puisqu'elle ne contient pas les optimisations que nous allons présenter dans la section 3.

Pour rendre cette implémentation indépendante de la représentation de la FNC et de la mise en FNC, nous avons besoin de deux types abstraits, un type des littéraux et un type qui représente la FNC elle-même :

```
module type LITERAL = sig
  type t
  val mk_not : t → t
  val compare : t → t → int
end
module type FNC = sig
  type formula
  module L : LITERAL
  val make : formula → L.t list list
end
```

À part le type `t` des littéraux, la signature `LITERAL` contient aussi une fonction de négation et une fonction de comparaison (nécessaire pour construire des ensembles de littéraux par le foncteur `Set.Make`). La signature `FNC` vient avec un module de type `LITERAL` et une fonction `make` qui, à partir d'une formule, construit une liste de listes de littéraux, c'est-à-dire une liste de clauses.

Nous pouvons maintenant mettre en place les structures de données pour notre *SAT-solver* naïf, tout en restant indépendant de la représentation de la FNC. Notre foncteur `Sat`, paramétré par un module du type `FNC`, déclare tout d'abord deux exceptions `Unsat` et `Sat`, le module `L` des littéraux et un module `S` d'ensembles de littéraux. Le type `t` de l'état du *SAT-solver* est tout simplement un ensemble `gamma` de littéraux et une formule en FNC `delta`, exactement comme dans les règles d'inférence de DPLL :

```
module Sat(Fnc : FNC) = struct
  exception Unsat
  exception Sat

  module L = Fnc.L
```

```
module S = Set.Make(L)
type t = { gamma : S.t; delta : L.t list list}
```

En suivant ces mêmes règles, nous pouvons procéder à l'implémentation de l'algorithme lui-même. Le code suivant implémente les règles ASSUME et BCP :

```
let rec assume env f =
  if S.mem f env.gamma then env
  else bcp { gamma = S.add f env.gamma; delta = env.delta}

and bcp env =
  List.fold_left
    (fun env l → try
       let l =
         List.filter
            (fun f →
              if S.mem f env.gamma then raise Exit;
              not (S.mem (L.mk_not f) env.gamma) ) l
       in
       match l with
           [] → raise Unsat (* CONFLICT *)
         | [f] → assume env f
         | _ → {env with delta = l ::env.delta}
     with Exit → env
    ) {env with delta=[]} env.delta
```

La règle ASSUME se traduit aisément vers le code présenté ci-dessus. La seule différence avec la règle d'inférence est l'appel de la fonction `bcp` juste après avoir ajouté le littéral `f` dans l'environnement `gamma`. Cette fonction `bcp`, qui est bien sûr la traduction des deux règles BCP, est plus complexe. Elle parcourt toute la formule courante (à l'aide du `fold_left`) et enlève tous les littéraux tels que leur négation est dans l'ensemble `gamma` (à l'aide du `filter`). Si le littéral lui-même apparaît dans `gamma`, toute la clause est enlevée ; ceci est implémenté à l'aide de l'exception `Exit` qui permet de sortir du filtre dès qu'un tel littéral a été trouvé. Si, au cours du parcours de la formule en FNC, on obtient la clause vide, on applique la règle CONFLICT et on revient en arrière en levant une exception `Unsat` (le motif de la liste vide). Si on trouve une clause unitaire (deuxième motif), le littéral correspondant est directement supposé ; les fonctions `bcp` et `assume` doivent donc être mutuellement récursives. Finalement, dans les autres cas, la clause est ajoutée à l'accumulateur du `fold_left`.

On voit bien que ces deux fonctions ne traduisent pas uniquement les règles, mais aussi une *stratégie*, c'est-à-dire une préférence de certaines règles, quand plusieurs peuvent s'appliquer. Ici, quand un littéral est supposé, les règles BCP et ASSUME sont tout de suite appliquées autant que possible. Si un conflit est trouvé, la règle CONFLICT est appliquée. Le point fixe qui est obtenu est une formule qui ne contient plus de clause unitaire, ni de clause vide (sinon l'exception `Unsat` aurait été levée).

La seule règle qui reste à traduire est la règle UNSAT.

```
let rec unsat env = try
  match env.delta with
      [] → raise Sat
    | ([_] | []) ::_ → assert false
    | (a ::_) ::_ →
        (try unsat (assume env a) with Unsat → ());
         unsat (assume env (L.mk_not a))
with Unsat → ()
```

Elle se traduit par une simple analyse par cas sur la formule courante : si on trouve une FNC vide, on a trouvé une instanciation des variables qui rend vraie la formule initiale. La clause vide et une clause unitaire ne sont pas possibles ; c'est l'invariant obtenu par les fonctions `bcp` et `assume`. Sinon, un littéral quelconque `a` est choisi, et supposé vrai. Si cette branche est explorée et aucune instanciation n'a été trouvée, on suppose faux le littéral.

Aucune exception `Unsat` ne doit s'échapper de la fonction `unsat`, puisque c'est précisément ici qu'il faut la traiter et c'est pour cela qu'elle est encadrée d'un bloc `try ... with`. Pourquoi y a-t-il donc un deuxième bloc `try ... with` autour du premier appel récursif `unsat (assume env a)` ? Ce bloc est nécessaire puisqu'une exception `Unsat` peut également être lévée dans l'appel de la fonction `assume`, et dans ce cas il faut bien continuer avec le deuxième appel récursif de `unsat`. Les appels de `assume` juste avant les appels récursifs de `unsat` maintiennent l'invariant qu'il n'y a pas de clause vide ni de clause unitaire dans `delta`.

Il reste la fonction d'entrée du *SAT-solver*, que nous avons appelée `dpll`.

```
  let dpll f = try
    unsat (bcp {gamma = S.empty ; delta = Fnc.make f}) ; false
  with Sat → true | Unsat → false

end (* fin du foncteur Sat *)
```

Elle construit la FNC de la formule `f`, appelle la fonction `bcp` pour éliminer les éventuelles clauses unitaires et établir l'invariant nécessaire afin de pouvoir appeler la fonction `unsat`. Si une exception `Sat` a été levée, la formule est satisfaisable. Si l'appel de `bcp` lève `Unsat`, elle est insatisfaisable. Si la fonction `unsat` termine normalement, elle est également insatisfaisable puisque toutes les branches ont été explorées sans qu'une instanciation des variables n'ait été trouvée.

Au final, on aboutit à un code très court (environ 40 lignes pour le foncteur `Sat`) qui a l'avantage supplémentaire d'être non seulement très proche des règles d'inférence, mais aussi de la formalisation Coq : la stratégie mise en œuvre dans cette implémentation est la même que celle codée « en dur » dans la preuve de complétude.

Nous verrons par la suite comment on peut améliorer l'efficacité de cette procédure en y apportant des modifications minimales.

3. Optimisations

La procédure présentée à la section précédente reste très naïve et les *SAT-solvers* modernes, tout en s'inspirant largement de cette procédure DPLL originale, parviennent à des résultats sensiblement meilleurs grâce à un grand nombre d'optimisations [14, 9].

Un certain nombre de ces optimisations sont de nature heuristique et s'efforcent par exemple de faire les choix de littéraux de décision les plus « pertinents » possibles lorsque la règle UNSAT est appliquée. D'autres, au contraire, sont purement algorithmiques et ont pour but d'élaguer au maximum l'arbre de recherche du *SAT-solver* afin d'éviter de reproduire plusieurs fois les mêmes raisonnements au cours d'une preuve.

Dans cette section, nous allons seulement nous intéresser à ce deuxième type d'optimisations car ce sont celles qui permettent de diminuer à coup sûr la taille de l'arbre de recherche et ne se basent pas sur une quelconque propriété des formules en entrée. Nous allons en présenter deux plus en détail : le retour en arrière non chronologique et l'apprentissage par analyse des clauses conflits. Nous allons voir en particulier comment de très légères modifications du code présenté à la section précédente peuvent conduire à de singulières améliorations des performances.

3.1. Retour en arrière non chronologique

Principe. Le retour en arrière non chronologique [13] consiste, lors de l'application d'une règle UNSAT, à analyser si le littéral l introduit dans la branche de gauche a été ou non « utile » pour l'établissement du conflit dans cette branche. Dans le cas où l n'a pas servi, on peut alors se dispenser d'effectuer la recherche dans la branche de droite. Pour illustrer l'apport de cette méthode, nous avons représenté en figure 2 le fonctionnement de DPLL sur un exemple particulier.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{\bar{x}_4 \vdash \{\}}}{x_3 \vdash \{\bar{x}_4\},\{x_4\}}\,\textsc{Assume} \quad \dfrac{\overline{\bar{x}_5 \vdash \{\}}}{\bar{x}_3 \vdash \{\bar{x}_5\},\{x_5\}}\,\textsc{Assume}}{x_2 \vdash \{\bar{x}_3,\bar{x}_4\},\{\bar{x}_3,x_4\},\{x_3,x_5\},\{x_3,\bar{x}_5\}}\,\textsc{Unsat} \quad \dfrac{\vdots}{\bar{x}_2 \vdash \ldots}}{x_1 \vdash \{\bar{x}_3,\bar{x}_4\},\{\bar{x}_3,x_4\},\{x_2,x_3,x_5\},\{x_3,x_5\},\{x_3,\bar{x}_5\}}\,\textsc{Unsat} \quad \cdots}{x_0 \vdash \{\bar{x}_3,\bar{x}_4\},\{\bar{x}_1,\bar{x}_3,x_4\},\{x_2,x_3,x_5\},\{x_3,x_5\},\{x_3,\bar{x}_5\}}\,\textsc{Unsat} \quad \cdots}{\emptyset \vdash \{\bar{x}_0,\bar{x}_3,\bar{x}_4\},\{\bar{x}_1,\bar{x}_3,x_4\},\{x_2,x_3,x_5\},\{x_3,x_5\},\{x_3,\bar{x}_5\}}\,\textsc{Unsat}$$

FIG. 2 – Exemple de fonctionnement de DPLL

Seules les règles ASSUME et UNSAT y sont représentées, on suppose qu'un maximum de propagation de contraintes booléennes est réalisé après chaque application d'une de ces deux règles. Aussi, seul le dernier littéral ajouté est montré dans le contexte Γ. On peut voir dans cette figure que dans la branche où x_2 a été supposé, les conflits proviennent de l'interaction entre les littéraux x_3, x_4 et x_5. La même dérivation existe certainement dans la branche droite où $\bar{x}_2$ est supposé (marquée par des pointillés verticaux), et donc la recherche dans cette branche est effectuée inutilement par DPLL.

Modification des règles. Pour permettre au système de prendre en compte ce phénomène, il faut qu'il puisse calculer grâce à quels littéraux un conflit a été obtenu dans une branche. Pour ce faire, on modifie DPLL de la manière suivante :

- le contexte Γ contient maintenant des littéraux *annotés*, c'est-à-dire des paires $l[\mathcal{A}]$ où l est le littéral ajouté au contexte et $\mathcal{A}$ est un ensemble de littéraux appelé *dépendances* de l ; ces dépendances représentent les littéraux qui ont conduit à l'ajout de l au contexte ;
- les clauses de Δ sont elles aussi annotées par un ensemble de littéraux qui permettent de se souvenir grâce à quels littéraux les clauses ont été réduites au cours de la recherche ;
- enfin, les séquents sont maintenant de la forme $\Gamma \vdash \Delta : \mathcal{A}$ où le nouvel élément $\mathcal{A}$ est l'ensemble des littéraux nécessaires pour établir l'incompatibilité de Γ et Δ. On peut considérer ces séquents comme un algorithme prenant en entrée Γ et Δ et retournant un ensemble de littéraux $\mathcal{A}$.

$$\textsc{Conflict}\ \frac{}{\Gamma \vdash \Delta, \emptyset[\mathcal{A}] : \mathcal{A}} \qquad \textsc{Assume}\ \frac{\Gamma, l[\mathcal{B}] \vdash \Delta : \mathcal{A}}{\Gamma \vdash \Delta, l[\mathcal{B}] : \mathcal{A}} \qquad \textsc{Bcp}\ \begin{cases} \dfrac{\Gamma, l[\mathcal{B}] \vdash \Delta, C[\mathcal{B} \cup \mathcal{C}] : \mathcal{A}}{\Gamma, l[\mathcal{B}] \vdash \Delta, \bar{l} \vee C[\mathcal{C}] : \mathcal{A}} \\ \dfrac{\Gamma, l[\mathcal{B}] \vdash \Delta : \mathcal{A}}{\Gamma, l[\mathcal{B}] \vdash \Delta, l \vee C[\mathcal{C}] : \mathcal{A}} \end{cases}$$

$$\textsc{Unsat}\ \frac{\Gamma, l[l] \vdash \Delta : \mathcal{A} \qquad \Gamma, \bar{l}[\mathcal{A} \setminus l] \vdash \Delta : \mathcal{B}}{\Gamma \vdash \Delta : \mathcal{B}}\ l \in \mathcal{A} \qquad \textsc{BJ}\ \frac{\Gamma, l[l] \vdash \Delta : \mathcal{A}}{\Gamma \vdash \Delta : \mathcal{A}}\ l \notin \mathcal{A}$$

FIG. 3 – Les règles de DPLL avec retour en arrière non-chronologique

Les règles correspondant à ce mécanisme sont détaillées en figure 3. Les annotations sont introduites dans la règle UNSAT au niveau des littéraux de décision, puis elles passent naturellement des littéraux aux clauses lors de la propagation de contraintes. Lorsqu'on arrive à la règle CONFLICT, on sauve dans le membre droit du séquent l'ensemble de littéraux qui ont permis d'obtenir la clause vide et c'est cet ensemble qui permet le *backjumping* via la nouvelle règle BJ. Cette procédure a été formalisée en Coq et prouvée correcte et complète. Les tailles de la spécification et des preuves sont respectivement d'environ 450 et 3500 lignes.

Ainsi, si l'on reprend l'exemple de la figure 2, la dérivation où la règle UNSAT est appliquée avec le littéral « inutile » x_2 va maintenant utiliser la règle BJ comme représenté dans la figure 4, où $\mathcal{A}$ représente l'ensemble de littéraux $\{x_0, x_1, x_3\}$ et $\mathcal{B} = \mathcal{A} \setminus x_3 = \{x_0, x_1\}$. Puisque $\mathcal{A}$, qui décore la branche de gauche, ne contient pas x_2, la règle BJ est appliquée et la branche de droite n'est pas explorée.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\bar{x}_4[x_0,x_3] \vdash \{\}[x_0,x_1,x_3] : \mathcal{A}}\,\textsc{Conflict}}{x_3[x_3] \vdash \{\bar{x}_4\}[x_0,x_3], \{x_4\}[x_1,x_3] : \mathcal{A}}\,\textsc{Assume} \qquad \dfrac{\dfrac{\dfrac{}{\bar{x}_5[x_0,x_1] \vdash \{\}[x_0,x_1] : \mathcal{B}}\,\textsc{Conflict}}{\bar{x}_3[x0,x_1] \vdash \{\bar{x}_5\}[x_0,x_1], \{x_5\}[x_0,x_1] : \mathcal{B}}\,\textsc{Assume}}{x_2[x_2] \vdash \{\bar{x}_3,\bar{x}_4\}[x_0], \{\bar{x}_3,x_4\}[x_1], \{\bar{x}_4,\bar{x}_5\}[], \{x_3,x_5\}[], \{x_3,\bar{x}_5\}[] : \mathcal{B}}\,\textsc{Unsat}}{x_1[x_1] \vdash \{\bar{x}_3,\bar{x}_4\}[x_0], \{\bar{x}_3,x_4\}[x_1], \{x_2,x_3,x_5\}[], \{x_3,x_5\}[], \{x_3,\bar{x}_5\}[] : \mathcal{B}}\,\textsc{BJ}$$

FIG. 4 – Exemple de fonctionnement de DPLL avec *backjumping*

Implémentation. Par rapport à l'implémentation de DPLL donnée à la section 2, il n'y a que très peu de choses à changer pour ajouter le retour en arrière non chronologique. L'environnement `gamma` est maintenant représenté par un dictionnaire où les littéraux sont associés à leurs dépendances, tandis que `delta` est une liste de couples, les clauses étant également associées à des dépendances :

```
module L = Fnc.L
module S = Set.Make(L)
module M = Map.Make(L)

type t = { gamma : S.t M.t; delta : (L.t list × S.t)  list }
```

Les fonctions `assume` et `bcp` fonctionnent comme précédemment, à ceci près qu'elles sont adaptées de telle sorte que la propagation des dépendances soit calculée correctement, et que l'exception `Unsat`, levée lorsqu'une clause vide est rencontrée, renvoie maintenant l'ensemble de dépendances associé à cette clause vide. La fonction `unsat` peut ainsi s'en servir pour effectuer une règle BJ ou non :

```
let rec unsat env = try
  match env.delta with
      [] → raise Sat
    | ([_],_ | [],_) ::_ → assert false
    | ((a ::_),_) ::_ →
      let d =
        try unsat (assume env (a,S.singleton a)) with Unsat d → d in
      if not (S.mem a d) then d
      else unsat (assume env (L.mk_not a,S.remove a d))
 with Unsat d → d
```

Enfin, le point d'entrée de la procédure est la fonction `dpll`, qui utilise la fonction `dispatch` pour annoter toutes les clauses avec l'ensemble vide et appelle alors la fonction `unsat` sur le séquent obtenu.

```
let dispatch d = List.map (fun l→ l,d)

let dpll f = try
  let  _ =
    unsat (bcp {gamma = M.empty ; delta = dispatch S.empty
    (Fnc.make f)}) in
  false
with Sat → true | Unsat _ → false
```

3.2. Apprentissage

Principe. Si l'ajout du retour en arrière non chronologique permet bien d'élaguer certaines parties de l'arbre de recherche à partir des conflits déjà trouvés, il ne tire pas complètement partie des informations à sa disposition. Pour s'en rendre compte, il suffit de considérer la situation schématisée par la figure 5.

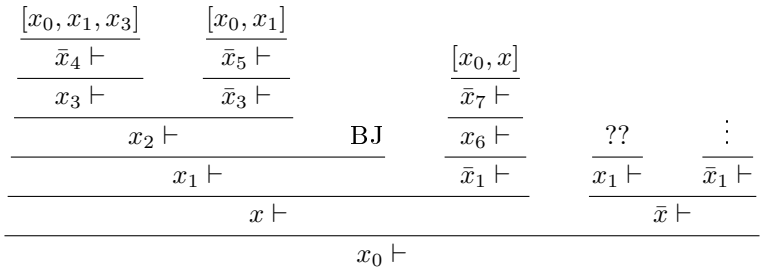

FIG. 5 – Exemple montrant l'insuffisance du *backjumping*

Cette figure schématise une dérivation dans le système de la figure 3 similaire à celle donnée en exemple en figure 2. Seuls les littéraux introduits sont représentés, ainsi que les ensembles de dépendances obtenus aux différentes feuilles. La différence entre la dérivation de la figure 4 et celle-ci est qu'ici, un autre littéral x a été introduit dans le contexte par une règle UNSAT entre les introductions des deux littéraux x_0 et x_1. Or, ce sont ces deux littéraux qui permettaient à eux deux de créer le conflit puisqu'après le *backjumping* sur x_2, les dépendances associées au séquent étaient justement l'ensemble $\mathcal{B} = \{x_0, x_1\}$.

Cela signifie qu'en particulier, supposer x_0 et x_1 vrais en même temps mène à un conflit dans ce problème. Ainsi, on sait que parcourir la branche marquée par les points d'interrogation ne servirait à rien puisque dans cette branche x_0 et x_1 sont vrais. Cependant, le retour en arrière non chronologique ne peut pas nous aider sur ce plan puisque les informations de dépendances qu'il utilise sont perdues dès lors que l'algorithme est repassé « en-dessous » d'un branchement où un des littéraux de dépendance avait été introduit. Ici, en redescendant de la branche où $\bar{x}_1$ est supposé, le nouvel ensemble de dépendances est $[x, x_0]$ et ne peut plus de toute façon faire intervenir x_1 : en revenant dans le branchement sur x, on a perdu l'information comme quoi x_0 et x_1 ne faisaient pas bon ménage et on ne peut pas l'exploiter dans la suite du parcours de l'arbre.

Modification des règles. Pour résoudre ce problème, une solution consiste à stocker, en plus de l'ensemble de dépendances courant, un ensemble de clauses appelées *clauses conflits* représentant l'ensemble des clauses que l'on a déjà « apprises » au cours de l'algorithme. Sur notre exemple, on a appris que x_0 et x_1 impliquent la clause vide, ce que l'on garde sous forme de dépendances : lorsque l'on redescend au branchement sur x, on veut faire en sorte de se souvenir que x_0 implique $\bar{x}_1$: autrement

dit, on veut passer de $\emptyset[x_0, x_1]$ à $\{\bar{x}_1\}[x_0]$. Plus généralement, nous allons considérer que nos clauses conflits sont des clauses annotées et définir une opération sur les ensembles de clauses conflits notée Shift_l et permettant de sortir un litéral l des annotations :

$$\begin{aligned} \mathit{Shift}_l(\emptyset) &= \emptyset \\ \mathit{Shift}_l(\{C[\mathcal{A}, l]\} \cup \mathbb{A}) &= \{\bar{l} \vee C[\mathcal{A}]\} \cup \mathit{Shift}_l(\mathbb{A}) \\ \mathit{Shift}_l(\{C[\mathcal{A}]\} \cup \mathbb{A}) &= \{C[\mathcal{A}]\} \cup \mathit{Shift}_l(\mathbb{A}) \text{ si } l \notin \mathcal{A} \end{aligned}$$

Les séquents s'écrivent maintenant $\Gamma \vdash \Delta : \mathcal{A}, \mathbb{A}$ où le nouvel élément $\mathbb{A}$ est l'ensemble des clauses conflits. Les règles sont très semblables à celles du système de la figure 3 et n'ajoutent que le traitement des clauses conflits ; elles sont présentées dans la figure 6. Celles-ci proviennent des dépendances trouvées lors des conflits, et la règle UNSAT permet d'ajouter $\bar{l}[\mathcal{A} \setminus l]$ aux clauses conflits lorsque l'ensemble de dépendances $\mathcal{A}$ contient l. Les clauses sont maintenues par toutes les règles, à ceci près que les règles UNSAT et BJ appliquent la fonction Shift_l aux clauses conflits trouvées dans la première branche, comme nous le suggérions ci-dessus. Enfin, ces clauses sont utilisées dans la branche droite de la règle UNSAT et sont ajoutées au contexte afin d'aider à accélérer la recherche d'un conflit dans cette branche. En effet, les clauses de $\mathit{Shift}_l(\mathbb{A})$ qui contiennent $\bar{l}$ seront éliminées par BCP, ce sont les clauses restantes qui permettront éventuellement d'obtenir un conflit plus rapidement.

$$\textsc{Conflict}\ \frac{}{\Gamma \vdash \Delta, \emptyset[\mathcal{A}] : \mathcal{A}, \emptyset} \qquad \textsc{Assume}\ \frac{\Gamma, l[\mathcal{B}] \vdash \Delta : \mathcal{A}, \mathbb{A}}{\Gamma \vdash \Delta, l[\mathcal{B}] : \mathcal{A}, \mathbb{A}}$$

$$\textsc{Bcp}\ \begin{cases} \dfrac{\Gamma, l[\mathcal{B}] \vdash \Delta, C[\mathcal{B} \cup \mathcal{C}] : \mathcal{A}, \mathbb{A}}{\Gamma, l[\mathcal{B}] \vdash \Delta, \bar{l} \vee C[\mathcal{C}] : \mathcal{A}, \mathbb{A}} \\ \dfrac{\Gamma, l[\mathcal{B}] \vdash \Delta : \mathcal{A}, \mathbb{A}}{\Gamma, l[\mathcal{B}] \vdash \Delta, l \vee C[\mathcal{C}] : \mathcal{A}, \mathbb{A}} \end{cases}$$

$$\textsc{Unsat}\ \frac{\Gamma, l[l] \vdash \Delta : \mathcal{A}, \mathbb{A} \qquad \Gamma, \bar{l}[\mathcal{A} \setminus l] \vdash \Delta, \mathit{Shift}_l(\mathbb{A}) : \mathcal{B}, \mathbb{B}}{\Gamma \vdash \Delta : \mathcal{B}, \mathit{Shift}_l(\mathbb{A}) \cup \{\bar{l}[\mathcal{A} \setminus l]\} \cup \mathbb{B}}\ l \in \mathcal{A}$$

$$\textsc{Bj}\ \frac{\Gamma, l[l] \vdash \Delta : \mathcal{A}, \mathbb{A}}{\Gamma \vdash \Delta : \mathcal{A}, \mathit{Shift}_l(\mathbb{A})}\ l \notin \mathcal{A}$$

FIG. 6 – Les règles de DPLL avec apprentissage par analyse des clauses conflits

Implémentation. Cette fois encore, les modifications à apporter à l'implémentation afin de mettre en place cette optimisation sont relativement restreintes. Le contexte `gamma` et l'ensemble des clauses `delta` d'un séquent sont les mêmes qu'auparavant. Les fonctions `bcp` et `assume` ne sont pas du tout modifiées par rapport à la section précédente. Seule la fonction `unsat` est en fait modifiée : elle doit retourner non plus seulement un ensemble de dépendances comme avant, mais également un ensemble de clauses annotées.

On définit donc via le foncteur `Set.Make` le module `C` des ensembles de clauses annotées, ainsi que l'implémentation de la fonction *Shift* sur ces ensembles.

```
module C = Set.Make(
  struct
    type t = L.t list × S.t
```

```
    let compare (cl1,s1) (cl2,s2) = ...
  end)

let shift a c =
  C.fold (fun ((cl,d) as x) c →
            let x = if S.mem a d then
              ((L.mk_not a) ::cl,S.remove a d) else x in
            C.add x c ) c C.empty
```

Ensuite, la fonction `unsat` se contente d'appliquer la fonction `shift` aux clauses retournées, fait le *backjumping* si c'est possible ou bien ajoute les clauses conflits au contexte de la branche droite dans le cas contraire.

```
let rec unsat env = try
  match env.delta with
      [] → raise Sat
    | ([_],_ | [],_) ::_ → assert false
    | ((a ::_),_) ::_ →
        let d , c =
          try unsat (assume env (a,S.singleton a))
          with Unsat r → r , C.empty in
        let c = shift a c in
        if not (S.mem a d) then d , c
        else
          let (n,dn) as x = L.mk_not a,S.remove a d in
          let d , c' =
            unsat
              (assume
                 { env with delta = (C.elements c)@env.delta} x)
          in d , C.add ([n],dn) (C.union c c')
with Unsat d → d , C.empty
```

4. Formes normales conjonctives

Comme nous l'avons vu dans les sections précédentes, la procédure DPLL — et ses extensions — manipulent exclusivement des formules en forme normale conjonctive (FNC). Une transformation est donc nécessaire si l'on veut vérifier la satisfaisabilité d'une formule arbitraire à l'aide de cette procédure. Maleureusement, il n'existe pas d'algorithme polynomial de mise en forme clausale[2] et cette transformation peut donc s'avérer catastrophique en pratique. Une solution à ce problème consiste à engendrer des FNC qui ne sont plus équivalentes à la formule de départ mais seulement équi-satisfaisables.

Nous présentons dans cette section une extension de DPLL pour manipuler, de manière incrémentale, des FNC équi-satisfaisables. Nous proposons également une implémentation du module `Fnc`, basée sur la technique du *hash-consing*, pour construire ces FNC équi-satisfaisables de manière efficace.

[2]S'il en était autrement on aurait alors également, par dualité, un algorithme polynomial de mise en DNF et donc une preuve de $P = NP$.

4.1. FNC équi-satisfaisables

Afin de déterminer la satisfaisabilité d'une formule booléenne ψ quelconque, les *SAT-solvers* présentés en sections 2 et 3 réalisent une mise en forme clausale à l'aide de la fonction `Fnc.make` qui convertit ψ en une liste de clauses de type `L.t list list`, où `L.t` représente le type des littéraux.

Bien que nécessaire, cette transformation peut néanmoins constituer un véritable frein à l'efficacité de ces outils. Dans le pire des cas, la taille de la forme normale conjonctive d'une formule ψ peut être $O(2^{|\psi|})$. Afin de se faire une idée de l'explosion combinatoire de cette transformation, le résultat de la mise en FNC de la formule ψ de la forme $(a_1 \wedge a_2 \ldots \wedge a_n) \vee (b_1 \wedge b_2 \ldots \wedge b_k)$ est la formule suivante :

$$\bigwedge_{i=1}^{n} \bigwedge_{j=1}^{k} (a_i \vee b_j)$$

Une solution bien connue pour contourner cette difficulté est de construire une forme normale conjonctive ϕ de taille proportionnelle à ψ, telle que ϕ soit *seulement* équi-satisfaisable, mais non-équivalente, à ψ (voir par exemple [12, 8]). L'idée consiste à introduire de nouvelles variables X (appelées *proxies*) pour remplacer les sous-formules φ de ψ et à définir les clauses représentant l'équivalence $\varphi \leftrightarrow X$. Par exemple, on peut remplacer les sous-formules $a_1 \wedge \ldots \wedge a_n$ et $b_1 \wedge \ldots \wedge b_k$ de ψ par deux *proxies* X et Y et ajouter les clauses pour représenter les deux équivalences $X \leftrightarrow (a_1 \wedge \ldots \wedge a_n)$ et $Y \leftrightarrow (b_1 \wedge \ldots \wedge b_k)$. On obtient alors la FNC suivante :

$$(X \vee Y) \wedge \bigwedge_{i=1}^{n} (\bar{X} \vee a_i) \wedge \bigwedge_{j=1}^{k} (\bar{Y} \vee b_j) \wedge \left(X \vee \bigvee_{i=1}^{n} \bar{a}_i \right) \wedge \left(Y \vee \bigvee_{j=1}^{k} \bar{b}_j \right)$$

Bien que cette solution réduise considérablement la taille des FNC, on peut encore réduire le nombre de clauses manipulées par un *SAT-solver* en retardant l'introduction des clauses définissant les *proxies* jusqu'au moment où ces variables se voient affecter une valeur par le *SAT-solver*. Par exemple, seule la formule $X \vee Y$ de la FNC ci-dessus peut être donnée au démarrage du *SAT-solver*, les clauses définissant X et Y peuvent elles être ajoutées seulement quand ces variables se voient assigner une valeur (positive ou négative).

Nous détaillons dans la suite de cette section un algorithme de mise en forme clausale utilisant cette technique des *proxies*, ainsi qu'un mécanisme d'ajout de clauses *incrémental* qui ne nécessite qu'un changement minimal dans les codes des *SAT-solvers* présentés dans les sections précédentes.

Principe. On remarque dans la FNC précédente que les clauses qui définissent, par exemple, le *proxy* X se réduisent à la formule $\bigwedge_{i=1}^{n} a_i$ quand X est vrai et à la formule $\bigvee_{i=1}^{n} \bar{a}_i$ quand X est faux. C'est donc l'une ou l'autre de ces deux formules que l'on souhaite insérer dans le *SAT-solver* quand X se voit attribuer une valeur. La même remarque s'applique aux clauses définissant les *proxies* des sous-formules de la forme $f \vee g$, $f \rightarrow g$ et $f \leftrightarrow g$. Le tableau de la figure 7 résume la forme de ces clauses en fonction de la valeur assignée au *proxy* par le *SAT-solver*.

	X	$\bar{X}$
$X \leftrightarrow f \wedge g$	$f \wedge g$	$\bar{f} \vee \bar{g}$
$X \leftrightarrow f \vee g$	$f \vee g$	$\bar{f} \wedge \bar{g}$
$X \leftrightarrow (f \rightarrow g)$	$\bar{f} \vee g$	$f \wedge \bar{g}$
$X \leftrightarrow (f \leftrightarrow g)$	$(\bar{f} \vee g) \wedge (f \vee \bar{g})$	$(f \vee g) \wedge (\bar{f} \vee \bar{g})$

FIG. 7 – Clauses définissant les *proxies*

Modification des règles. On modifie les règles ASSUME et UNSAT de la procédure DPLL afin d'introduire dans Δ les clauses qui définissent un *proxy* au moment où cette variable est ajoutée à Γ. La figure 8 montre les modifications apportées aux règles de la figure 1. La fonction `expand` prend en argument un littéral l et retourne un ensemble de clauses. Cet ensemble doit contenir des clauses de la forme de celles présentées dans le tableau de la figure 7, si l est un *proxy*. Il doit être vide si l est un littéral « traditionnel ». Des modifications similaires peuvent être apportées aux règles de DPLL avec *backjumping* et apprentissage : il faut simplement attacher les informations de dépendances portées par l aux clauses de `expand(`l`)`.

Implémentation. Une fonction `expand` de type `L.t → L.t list list` est ajoutée à la signature `FNC`. Le type `L.t` représente donc maintenant à la fois les littéraux « traditionnels » et les *proxies*. La nature des variables est indifférente au *SAT-solver*, seule la fonction `expand` doit pouvoir les distinguer.

```
module type FNC = sig
  type formula
  module L : LITERAL
  val make : formula → L.t list list
  val expand : L.t → L.t list list
end
```

L'unique modification à apporter au *SAT-solver* de la section 2 est la concaténation à `delta` de la liste de clauses retournées par `Fnc.expand l`, au moment où le littéral `l` est passé en argument à la fonction `assume`. On obtient la fonction `assume` suivante :

```
  let rec assume env l =
    if S.mem l env.gamma then env
    else bcp { gamma = S.add l env.gamma ;
               delta = (Fnc.expand l)@env.delta }
```

Une modification similaire doit être réalisée pour la fonction `assume` des *SAT-solvers* de la section 3. On utilise la fonction `dispatch` pour propager les informations de dépendances associées au littéral `l`. On obtient le code suivant :

```
  let rec assume env (l,d) =
    if M.mem l env.gamma then env
    else bcp { gamma = M.add l d env.gamma ;
               delta = (dispatch d (Fnc.expand l))@env.delta}
```

4.2. Implémentation du module `Fnc`

Nous présentons brièvement dans cette section une implémentation efficace d'un module `Fnc` pour construire des FNC équi-satisfaisables. La méthode mise en œuvre repose sur la la technique de *hash-consing*. Le lecteur intéressé trouvera de plus amples détails dans [5].

$$\textsc{Assume}\ \frac{\Gamma, l \vdash \mathtt{expand}(l), \Delta}{\Gamma \vdash \Delta, l} \qquad \textsc{Unsat}\ \frac{\Gamma, l \vdash \mathtt{expand}(l), \Delta \qquad \Gamma, \bar{l} \vdash \mathtt{expand}(l), \Delta}{\Gamma \vdash \Delta}$$

FIG. 8 – Les règles de DPLL avec *proxies*

Principe. La méthode suit la structure récursive des formules. Par exemple, si ψ est une formule de la forme $f \wedge g$, on crée récursivement les *proxies* X_f et X_g de f et g, respectivement, puis on construit les deux listes de clauses ψ^+ et ψ^- en utilisant X_f et X_g, comme indiqué par le tableau de la figure 7. Maintenant, plutôt que de créer une nouvelle variable pour remplacer ψ, nous allons simplemement *hash-conser* la paire (ψ^+, ψ^-) et utiliser la valeur obtenue comme *proxy* pour ψ. Ainsi, nous réalisons non seulement une FNC équi-satisfaisable mais nous assurons également un partage maximal des sous-formules (donc des *proxies*), ce qui peut s'avérer très efficace pour les *SAT-solvers* (cf. section 5).

Implémentation. Nous utilisons la librairie `Hashcons` présentée dans [5] pour fabriquer des FNC équi-satisfaisables. Ce module contient une fonction `hashcons`, de type α `t` $\rightarrow \alpha \rightarrow \alpha$ `hash_consed`, permettant d'*hash-conser* des valeurs d'un type quelconque α. Le premier argument de la fonction est une table contenant les valeurs déjà partagées. Le résultat est une valeur dont le type, α `hash_consed`, permet de distinguer les valeurs *hash-consées* de celles qui ne le sont pas encore (très utile pour assurer le partage maximal). `hash_consed` est un type enregistrement ayant une étiquette `node` de type α dans laquelle est stockée la valeur passée à la fonction `hashcons`. Le type `L.t` des littéraux est défini à l'aide des trois types récursifs ci-dessous. Le type `t` représente les *proxies* des sous-formules. Il correspond aux paires *hash-consées* des clauses de la figure 7, représentées ici par des enregistrements de type `view` avec deux étiquettes `pos` et `neg`. Les clauses sont définies récursivement par le type `clause` qui représente soit une liste de clauses *hash-consées*, soit directement un littéral « traditionnel ».

```
module L = struct
 type t = view Hashcons.hash_consed
 and view = { pos : clauses ; neg : clauses}
 and clauses = C of t list list | LT of string × bool
```

La fonction `mk_not` construit la négation d'un *proxy* x simplement en échangeant le contenu des champs `x.node.pos` et `x.node.neg`, puis en *hash-consant* le nouvel enregistrement :

```
 let mk_not x = let n = x.node in hashcons table {pos=n.neg ;neg=n.pos}
```

La fonction `expand` du module `Fnc` se contente de retourner la liste des clauses contenues dans le champ `pos` du *proxy* `x`. Elle renvoie la liste vide si `x` représente un littéral « traditionnel » :

```
let expand x = match x.node.pos with C l → l | LT _ → []
```

Enfin, on construit le *proxy* d'une sous-formule comme $f \wedge g$ à l'aide de la fonction `mk_and` ci-dessous. Cette fonction prend en argument deux *proxies* f et g de type `L.t` et retourne le *proxy* correspondant au *hash-consing* des FNC $f \wedge g$ et $\bar{f} \vee \bar{g}$.

```
let mk_and f g = hashcons table {pos=C[[f] ;[g]] ; neg=C[[mk_not f ;mk_not g]]}
```

5. Performances

Nous détaillons dans cette section les résultats des tests de performances des *SAT-solvers* présentés dans les sections précédentes. Les formules utilisées pour effectuer ces expérimentations sont extraites du jeu de tests du concours annuel sur les *SAT-solvers* [1]. Ces formules sont déjà en forme normale conjonctive ; elles ne seront donc d'aucune utilité pour mesurer les effets des FNC équi-satisfaisables. Nous invitons le lecteur intéressé par les performances de la technique présentée en section 4 à consulter les résultats détaillés dans [5].

Le tableau ci-dessous récapitule les résultats obtenus sur des instances des trois familles de problèmes suivantes :

– AIM, instances satisfaisables du problème 3-SAT générées aléatoirement
– UF, instances 3-CNF mettant en jeu le phénomène de transition de phase
– DUBOIS, instances insatisfaisables générées aléatoirement

Les chiffres qui suivent les noms des formules de la première colonne indiquent le nombre de variables et le nombre de clauses de la FNC. Les trois colonnes suivantes donnent les temps de réponse (pour un Pentium 4 2GHz avec 512Mo de mémoire) des *SAT-solvers* présentés dans les Sections 2, 3.1 et 3.2, respectivement.

	DPLL	DPLL-B	DPLL-C
aim-50-1.6 (50,80)	4s	40ms	4ms
aim-100-2.0 (100,200)	> 10m	33s	0.3s
aim-200-2.0 (200,400)	> 10m	7m	4s
uf-125 (125,538)	22s	12s	10s
dubois (66,176)	8m30s	47s	52s

On peut constater que, sur ces exemples, les optimisations présentées en section 3 ont un impact fort sur les performances des *SAT-solvers*. Cet impact est à relativiser par le fait que notre système n'utilise aucune stratégie particulière pour sélectionner les littéraux de décision. Ainsi, il est très sensible à l'ordre des clauses et en particulier dans le cas de l'apprentissage, à l'ordre dans lequel les clauses conflit sont ajoutées. Ceci peut expliquer les différences de gain observées par exemple entre DUBOIS et AIM-200-2.0.

6. Travaux Connexes et Conclusion

Nous avons présenté une implémentation concise d'un *SAT-solver* moderne en OCaml à partir d'une description du système à base de règles d'inférence. Nous avons montré comment des optimisations permettant de diminuer l'espace de recherche peuvent être mises en œuvre en modifiant les règles; d'autre part, comme ces règles ont été conçues avec le souci de la proximité entre formalisation et implémentation, le code du DPLL naïf a pu être facilement adapté afin de prendre en compte ces optimisations. Les comparaisons des résultats obtenus présentés dans la section 5 démontrent les effets importants de ces améliorations. Par ailleurs, l'implémentation de notre *SAT-solver* étant indépendante de la représentation des formules proprement dites, nous avons montré comment tirer profit de techniques permettant une mise en FNC efficace : ainsi, nous avons présenté comment un partage des sous-formules par *hash-consing* et un calcul paresseux de la FNC pouvaient être implémentés de manière transparente pour le *SAT-solver*.

Les travaux les plus proches de notre démarche sont ceux de Nieuwenhuis, Oliveras et Tinelli sur la formalisation de DPLL [11]. Leur système, basé sur des règles de réécriture, décrit une version de la procédure DPLL où les conditions de garde des règles sont exprimées de manière *abstraite* par des formules logiques. Bien que rendant plus aisée la preuve de correction, ce genre de formalisation rend plus difficile la réalisation pratique du solveur et *a fortiori* la preuve de son implémentation. Or, de part l'utilisation croissante de ces outils dans l'industrie, il est important de s'assurer de leur correction [15] : nous avons justement été en mesure de prouver formellement la correction et la complétude de nos deux premiers systèmes, et la preuve du système avec apprentissage est en cours.

Sat-Micro peut être largement amélioré en intégrant des optimisations comme le *restart* ou les *two-matched literals* [10], ainsi que de « bonnes » heuristiques pour le choix des littéraux de décision. La recherche des littéraux dont dépendent les conflits pourrait aussi être raffinée de telle sorte que seuls les littéraux « dominateurs » du conflit soient retournés. Ces littéraux, comme expliqué dans [2], permettent souvent un *backjumping* plus efficace que celui que nous avons présenté. Il serait enfin intéressant de voir comment adapter le système pour résoudre le problème de la satisfaisabilité modulo une théorie, afin de pouvoir l'exploiter plus efficacement au sein d'un solveur SMT tel Ergo [4, 3].

Références

[1] The international SAT Competitions web page. `http://www.satcompetition.org/`.

[2] P. Beame, H. Kautz, and A. Sabharwal. Understanding the power of clause learning. 2003.

[3] S. Conchon and E. Contejean. The Ergo automatic theorem prover. `http://ergo.lri.fr/`.

[4] S. Conchon, E. Contejean, J. Kanig, and S. Lescuyer. Lightweight Integration of the Ergo Theorem Prover inside a Proof Assistant. In J. Rushby and N. Shankar, editors, *AFM07 (Automated Formal Methods)*, 2007.

[5] S. Conchon and J.-C. Filliâtre. Type-Safe Modular Hash-Consing. In *ACM SIGPLAN Workshop on ML*, Portland, Oregon, Sept. 2006.

[6] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7) :394–397, 1962.

[7] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3) :201–215, 1960.

[8] T. B. de la Tour. Minimizing the number of clauses by renaming. In *CADE-10 : Proceedings of the tenth international conference on Automated deduction*, pages 558–572, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[9] J. W. Freeman. *Improvements to propositional satisfiability search algorithms.* PhD thesis, Philadelphia, PA, USA, 1995.

[10] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff : engineering an efficient sat solver. In *DAC '01 : Proceedings of the 38th conference on Design automation*, pages 530–535, New York, NY, USA, 2001. ACM Press.

[11] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Abstract DPLL and abstract DPLL modulo theories. In F. Baader and A. Voronkov, editors, *Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'04), Montevideo, Uruguay*, volume 3452 of *Lecture Notes in Computer Science*, pages 36–50. Springer, 2005.

[12] D. A. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *J. Symb. Comput.*, 2(3) :293–304, 1986.

[13] J. P. M. Silva and K. A. Sakallah. Grasp : a new search algorithm for satisfiability. In *ICCAD '96 : Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society.

[14] L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In *CADE-18 : Proceedings of the 18th International Conference on Automated Deduction*, pages 295–313, London, UK, 2002. Springer-Verlag.

[15] L. Zhang and S. Malik. Validating sat solvers using an independent resolution-based checker : Practical implementations and other applications. In *DATE '03 : Proceedings of the conference on Design, Automation and Test in Europe*, page 10880, Washington, DC, USA, 2003. IEEE Computer Society.