Rapport de projet de Preuves de Programmes

Ewen DUFOUR

5 février 2026

Résumé

Dans ce projet, nous proposons une implémentation de l'algorithme DPLL. Nous donnons une spécification et montrons que notre implémentation respecte cette spécification à l'aide de l'outil de vérification déductive Why3.

1 Introduction

1.1 Problème étudié

Nous étudions le problème de la satisfiabilité booléenne. Ici, nous nous intéressons en particulier au problème 3-SAT. Ce problème prend en entrée une liste de clauses, chacune composée de trois littéraux séparés par des opérateurs $\vee$. Un littéral est une variable, possiblement précédée par un opérateur $\neg$. Nous cherchons une assignation de valeur booléenne pour chaque variable qui rend toutes les clauses vraie. L'algorithme DPLL permet de résoudre ce problème.

1.2 Algorithme DPLL

DPLL prend une assignation et une liste de clause à satisfaire. L'algorithme commence par scanner la liste de clauses afin de :

— vérifier si une clause est déja fausse dans l'assignation fournie, auquel cas DPLL renvoie directement unsat ;

— séparer les clauses satisfaites, des clauses à satisfaire

Si aucune clause fausse n'est trouvée, DPLL prend la prochaine variable non assignée et lui assigne une valeur (disons vrai). Il s'appelle récursivement pour tester la satisfiabilité avec la nouvelle assignation. Si l'appel récursif renvoi unsat, alors l'algorithme assigne à la variable la valeur opposée (ici, faux), et s'appelle de nouveau récurssivement. Dans le cas ou l'appel récursif renvoi de nouveau unsat, DPLL renvoi unsat. Si toutes les clauses sont satisfaites, DPLL renvoi sat.

1.3 Description du projet

On nous donne une fonction principale *sat* appelant la fonction *dpll* ainsi qu'une spécification que *sat* doit respecter.

```
let sat (mm: assignment) (cl: array cls) : (sat: bool)
  requires { 0 < length mm }
  requires { forall i. 0 <= i < length cl -> is_cls (length mm) cl[i] }
  ensures  { sat -> is_assignment mm (length mm) }
  ensures  { sat -> forall i. 0 <= i < length cl -> ok_cls mm (length mm) cl[i] }
  ensures  { not sat -> forall mc. is_assignment mc (length mm) ->
```

```
    exists i. 0 <= i < length cl /\ not ok_cls mc (length mm) cl[i] }
  ensures  { permut_all (old cl) cl }
= mm[0] <- 0;
  dpll mm cl (length mm) 1 (length cl)
```

Le contrat de cette fonction indique deux choses. Premièrement, la fonction attend une entrée valide. C'est-à-dire une liste de clauses bien formées (stockée dans cl) ainsi qu'un tableau pouvant accueillir l'assignation de valeur de chaque variable (mm). Deuxièmement, la fonction nous assure trois propriétés sur le résultat et la liste de clauses :

1. (Soundness) si la fonction renvoie sat, elle fourni dans mm une assignation valide de chaque littéral rendant toutes les clauses vraies ;
2. (Completeness) si la fonction renvoie unsat, alors toute assignation valide de littéral possible rend au moins une clause fausse ;
3. (Faithfullness) la fonction ne change pas le problème étudié.

1.4 Détails techniques

Pour un problème donné, nous appelons nv le nombre de variables. Un littéral $-i := \neg x_i$ ou $i := x_i$ est valide lorsque $0 \leq i < nv$. Une clause c valide est une disjonction de trois littéraux valides. Nous représentons ces propriétés par les prédicats suivants dans Why3.

```
type lit = int
type cls = (lit, lit, lit)

predicate is_lit (nv: int) (l: lit) =
  -nv < l < nv

predicate is_cls (nv: int) (c: cls) =
  let l1, l2, l3 = c in
  is_lit nv l1 /\ is_lit nv l2 /\ is_lit nv l3
```

La case i du tableau mm stocke l'assignation de la variable i. L'assignation est vraie lorsque la case i stocke la valeur $-i$, fausse lorsqu'elle stocke la valeur i et invalide dans les autres cas. Une clause est vraie dès lors qu'un de ses littéraux est vrai. Nous utilisons les prédicats suivants dans Why3 :

```
type assignment = array int

predicate is_assignment (mm: assignment) (na: int) =
  0 < na <= length mm /\ mm[0] = 0 /\
  forall i. 0 <= i < na -> abs mm[i] = i

predicate ok_lit (mm: assignment) (na: int) (l: lit) =
  is_lit (length mm) l /\ 0 <= abs l < na <= length mm /\
  mm[abs l] <> l

predicate ok_cls (mm: assignment) (na: int) (c: cls) =
  let l1, l2, l3 = c in
  ok_lit mm na l1 \/ ok_lit mm na l2 \/ ok_lit mm na l3
```

Ici, na représente le numéro de la première variable non assignée. Nous supposons implicitement que les variables sont assignées dans l'ordre de 1 à $nv - 1$.

2 Fonctions de véracité des clauses

Dans notre implémentation, nous avons tout d'abord besoin de tester si les clauses sont satisfaites pour une assignation donnée.

2.1 Véracité des littéraux

Nous calculons la valeur de vérité d'un littéral avec la fonction suivante :

```
let is_lit_sat (mm : assignment) (nv na : int) (l : lit) : (assgnd: bool, sat : bool)
= if abs l < na then
  (true, mm[abs l] <> l)
  else (false, false)
```

Nous proposons le contrat suivant pour cette fonction :

```
  requires { is_assignment mm na }
  requires { is_lit nv l }
  ensures  { assgnd <-> abs l < na }
  ensures  { sat <-> ok_lit mm na l }
```

Cette fonction suppose que le littéral que nous testons et l'assignation fournie sont valides. Notre spécification reflète les propriétés suivantes :

1. un littéral est assigné si son indice est inférieur à celui de la première variable non assignée ;
2. être satisfait dans l'assignation actuelle correspond au prédicat $ok\_lit\ mm\ na$.

2.2 Véracité des clauses

Nous testons la valeur de vérité d'une clause avec la fonction :

```
let is_clause_satisfied (mm : assignment) (nv na: int) (cl : cls) : (unsat : bool, sat: bool)
= let (l1, l2, l3) = cl in
  let (a1, s1) = is_lit_sat mm nv na l1 in
  let (a2, s2) = is_lit_sat mm nv na l2 in
  let (a3, s3) = is_lit_sat mm nv na l3 in
  let truth_value = s1 || s2 || s3 in
  ( a1 && a2 && a3 && not truth_value, truth_value )
```

Pour cette fonction, nous proposons le contrat qui suit :

```
  requires { is_cls nv cl }
  requires { is_assignment mm na }
  ensures  { sat <-> ok_cls mm na cl }
  ensures  { nv = na /\ (not unsat) -> sat }
  ensures  { unsat -> forall mc.
             ((forall i. 0 <= i < na -> mc[i] = mm[i]) /\
             is_assignment mc (mm.length)) -> not ok_cls mc (length mm) cl}
```

Ce contrat reflète que la fonction suppose que la clause et l'assignation fournies sont valides. Il indique également trois propriétés sur le résultat de la fonction dont nous aurons besoin :

1. la fonction renvoi *sat* si et seulement si la clause est valide et satisfaite ;
2. si toutes les variables sont assignées et que la clause n'est pas fausse, alors elle est forcément vraie ;

3. (Completeness sous l'assignation actuelle) si la fonction renvoie unsat, n'importe quelle extension de l'assignation actuelle ne permettra pas de rendre la clause vraie.

Cette propriété est due au fait que les trois littéraux sont déjà assignés, et par conséquent, assigner d'autres variables ne change pas la valeur de vérité de la clause. Nous nous servons de cette propriété afin de prouver la completeness de notre fonction finale.

3 Fonction Scan

Comme mentionné dans notre description de DPLL, nous avons besoin d'une fonction de scan qui va :

— tester si l'assignation rend une clause fausse ;
— séparer les clauses à satisfaire des clauses satisfaite.

Dans ce but, nous proposons l'implémentation suivante pour la fonction scan :

```
let scan (mm: assignment) (cl: array cls) (nv na nc: int) : (b: bool, mc: int)
= let ref i = 0 in
  let ref mc = nc in
  while i < mc do
    match is_clause_satisfied mm nv na cl[i] with
    | True , _ ->
      return false, mc
    | False, False ->
      i <- i + 1
    | False, True ->
      mc <- mc - 1;
      swap cl i mc
    end
  done;
  true , mc
```

Le contrat de cette fonction est le suivant :

```
  requires { is_assignment mm na }
  requires { nv = mm.length }
  requires { forall i. 0 <= i < length cl -> is_cls (length mm) cl[i] }

  requires { 0 <= nc <= cl.length }
  requires { forall i. nc <= i < cl.length -> ok_cls mm na cl[i] }

  ensures { permut_all (old cl) cl}
  ensures { forall i. mc <= i < cl.length -> ok_cls mm na cl[i] }
  ensures { forall i. nc <= i < cl.length -> cl[i] = old (cl[i]) }
  ensures { 0 <= mc <= nc }
  ensures { na = nv  /\ b -> mc = 0 }
  ensures { not b -> forall mm'.
            ((forall i. 0 <= i < na -> mm'[i] = mm[i]) /\
            is_assignment mm' (mm.length)) ->
            exists i. 0 <= i < length cl /\ not ok_cls mm' (length mm) cl[i]}
```

Ce contrat nous exprime plusieurs choses :

1. (Faithfullness) le tableau de clause est seulement permuté ;

2. les clauses situées après mc sont satisfaites ;
3. la zone des clauses satisfaite au départ ($\geq nc$) est inchangée ;
4. le nombre de clauses satisfaites ne peut qu'augmenter ;
5. si toutes les variables sont assignées et que la fonction ne trouve pas de clause fausse, alors toutes les clauses sont vraies ;
6. (Completeness sous l'assignation actuelle) si une clause fausse est trouvée, alors étendre de l'assignation actuelle ne permet pas de rendre cette clause vraie.

3.1 Invariants

Afin montrer que cette implémentation est correcte, nous devons ajouter les invariants (et variant) de boucles suivants :

```
variant { mc - i }
invariant { forall i. mc <= i < cl.length -> ok_cls mm na cl[i] }
invariant { permut_all (old cl) cl }
invariant { forall i. nc <= i < cl.length -> cl[i] = old (cl[i]) }
invariant { 0 <= i <= mc <= nc }
invariant {forall j. 0 <= j < i -> not (ok_cls mm na cl[j]) }
invariant {nv = na -> i = 0 }
```

Ils traduisent les propriétés suivantes :

1. la boucle termine ;
2. les clauses dont l'indice est plus grand que mc sont satisfaites ;
3. (Faithfullness) le tableau est au plus permuté ;
4. la zone $\geq nc$ reste intouchée ;
5. la fonction n'accède qu'à des cases dont on ne sait pas encore si les clauses sont satisfaites ;
6. les cases de cl que la fonction passe (entre 0 et i exclu) contiennent des clauses insatisfaites (mais pas insatisfiables) sous l'assignation actuelle ;
7. Si toutes les variables sont assignées, il ne peut pas y avoir de clause insatisfaite qui n'est pas insatisfiable.

3.1.1 Initialisation des invariants

Au début de la boucle, nous avons $i = 0$ et $mc = nc$. Les invariants sur la zone après mc correspondent aux préconditions de scan. De plus, le tableau n'a pas été modifié, et les bornes sont directement vérifiée.

3.1.2 Préservation des invariants

À chaque passage dans la boucle :
— si la clause courante est insatisfiable, la fonction s'arrête.
— si la clause testée n'est pas satisfaite, mais pas insatisfiable, seul i est incrémenté. Nous en déduisons donc :
  — comme $i < mc$, $i + 1 \leq mc$ et les autres bornes sont préservées ;
  — nous savons que les cases $\leq i$ contient une clause non satisfaite. L'invariant 6 est préservé ;
  — si $nv = na$, aucune clause ne peut être insatisfaite sans être insatisfiable (postcondition 2 de $is\_clause\_satisfied$) ;

— les autres invariants sont trivialement préservés.

— si la clause est satisfaite, alors mc est décrémentée et on échange la case i avec la case $mc - 1$. Ainsi :
  — comme $i < mc$, $i \leq mc - 1$ et les autres bornes sont préservées ;
  — la propriété de permutation est préservée ;
  — la clause qui se retrouve dans la case $mc - 1$ est satisfaite. L'invariant 1 est préservé ;
  — les autres invariants sont trivialement préservés.

3.1.3 Déduction des postconditions

Montrons à présent que les invariants sont préservés. Nous pouvons déduire les postconditions grâce à nos invariants :

1. invariant 2 ;
2. invariant 1 ;
3. invariant 3 ;
4. invariant 4 ;
5. Comme $na = nv$, d'après l'invariant 6, $i = 0$ après la dernière itération de la boucle. De plus, comme $b = true$, nous en concluons que la boucle s'est terminée normalement et $mc \leq i$. Or $0 \leq mc \leq i = 0$, donc $mc = 0$.
6. Comme $b = false$, $is\_clause\_satisfied$ a renvoyé $(faux, \_)$; Donc cette postcondition se déduit de la postcondition 3 de $is\_clause\_satisfied$.

4 Fonction DPLL

Nous proposons d'implémenter l'algorithme DPLL avec la fonction récursive suivante :

```
let rec dpll (mm: assignment) (cl: array cls) (nv na nc: int) : (s: bool)
= match scan mm cl nv na nc with
  | False, _ -> false
  | True, mc ->
    if mc  = 0 then (
      let ref i =  na in
      while i < nv do
        mm[i] <- -i;
        i <- i+1;
      done;
      true)
    else (
        mm[na] <- -na;
        if dpll mm cl nv (na + 1) mc then true
        else (
          mm[na] <- na;
          dpll mm cl nv (na + 1) mc
      )
    )
  end
```

Le contrat de dpll est le suivant :

```
  requires { 0 < na <= nv }
  requires { is_assignment mm na }
  requires { nv = mm.length }
  requires { 0 <= nc <= cl.length }
  requires { forall i. 0 <= i < length cl -> is_cls (length mm) cl[i] }
  requires { forall i. nc <= i < cl.length -> ok_cls mm na cl[i] }

  ensures { forall i. 0 <= i < na -> mm[i] = (old mm[i])}
  ensures { forall i. nc <= i < cl.length -> cl[i] = old (cl[i]) }
  ensures { permut_all (old cl) cl }
  ensures { s -> forall i. 0 <= i < length cl -> ok_cls mm (length mm) cl[i] }
  ensures { s -> is_assignment mm (mm.length) }
  ensures { not s -> forall mc.
            ((forall i. 0 <= i < na -> mc[i] = mm[i]) /\
            is_assignment mc (mm.length)) ->
            exists i. 0 <= i < length cl /\ not ok_cls mc (length mm) cl[i]}
  variant { nv - na }
```

Ce contrat reflète le fait que cette fonction attend une entrée valide et nous assure plusieurs propriétés :

1. les variables déjà assignées ne sont pas changées ;
2. les clauses déjà satisfaites ne sont pas touchées ;
3. (Faithfullness) le tableau est au plus permuté ;
4. (Soundness) si dpll renvoie *true*, elle fourni une assignation dans mm qui satisfait toutes les clauses ;
5. (Completeness sous l'assignation actuelle) si *dpll* renvoie $false$, aucune extension de l'assignation courante ne peut satisfaire toutes les clauses ;
6. nous avons un nombre fini d'appels récursif.

4.1 Invariants

Afin de prouver que ce code respecte notre contrat, nous devons ajouter les invariants suivant dans notre boucle de complétion de l'assignation :

```
variant {nv - i}
invariant { forall j. 0 <= j < length cl -> ok_cls mm i cl[j] }
invariant { is_assignment mm i }
invariant { na <= i <= nv }
invariant { forall j. 0 <= j < na -> mm[j] = (old mm[j]) }
```

Ils capturent le fait que la boucle termine et assigne des variables non assignées sans changer celles qui sont déjà assignées. De plus, l'extension de l'assignation par la boucle préserve le fait que toutes les clauses sont satisfaites.

4.1.1 Initialisation des invariants

À l'initialisation, i = na. Nous pouvons donc vérifier que les invariants sont initialement vrais :

1. découle directement de la postcondition 2 de *scan* couplée au fait que $mc = 0$;
2. $is\_assigment\ mm\ na$ est une précondition de *dpll* ;

3. $na \leq nv$ d'après une précondition de *dpll* ;
4. le tableau n'a pas encore été touché.

4.1.2 Préservation de l'invariant

Vérifions que ces invariants sont bien préservés :

1. les invariants 4 et 1 de l'itération précédente indiquent que les variables déjà assignées satisfont déjà toutes les clauses et ne sont pas changées ;
2. l'invariant 2 de l'itération précédente combiné au fait que l'on assigne à la case i la valeur $-i$ font que la nouvelle assignation est valide ;
3. par l'invariant 3, au début de la boucle $na \leq i \leq nv$. i est incrémenté de 1 dans la boucle. Nous voulons montrer que $i + 1 \leq nv$. Or comme nous sommes dans une itération de la boucle, nous avons $i < nv$. Donc $i + 1 \leq nv$;
4. Par l'invariant 3 de l'itération précédente, les cases changées sont après na.

4.1.3 Déduction des postconditions

Nous établissons d'abord la correction de la branche $mc = 0$. Après la dernière itération, $i = nv = mm.length$. La postcondition 1 découle de l'invariant 4, la postcondition 4 de l'invariant 1, et la postcondition 5 de l'invariant 2 et du fait que $i = nv$.

La correction de l'autre branche est assez directe grâce aux appels récursifs à *dpll*. La subtilité principale réside dans la preuve de Completeness. Elle est due au fait que si les deux appels récursifs à *dpll* ont échoués, cela signifie que l'on a testé toutes les valeurs possibles pour la variable d'indice na et que dans tous les cas, étendre l'assignation après na mène directement à une clause insatisfiable.

5 Conclusion

Le contrat que nous avons donné à *dpll* permet de prouver directement le contrat de *sat*. Notre implémentation est donc correcte.

Ressenti général

Lors de ce projet, je n'ai pas eu de très grandes difficultés. Je dirais que la partie qui m'a posé le plus de problème était la preuve de Completeness. En particulier, trouver les bons invariants de boucle et où mettre le coeur de la propriété.