

Rapport Projet Proofs of Programs

Boolean Satisfiability and the DPLL Algorithm

Hugo Barreiro

MPRI 2025–2026

Table des matières

| | | |
|----------|-----------------------------------|-----------|
| 1 | Introduction | 3 |
| 2 | Implémentation | 3 |
| 3 | Fonction <code>sat</code> | 4 |
| 4 | Fonction <code>dpll</code> | 5 |
| 4.1 | Pré-condition | 6 |
| 4.2 | Écriture | 6 |
| 4.3 | Variant | 6 |
| 4.4 | Post-condition | 7 |
| 4.5 | Invariant de boucle | 7 |
| 5 | Fonction <code>scan</code> | 8 |
| 5.1 | Pré-condition | 9 |
| 5.2 | Écriture | 9 |
| 5.3 | Variant | 9 |
| 5.4 | Post-condition | 9 |
| 5.5 | Invariant de boucle | 10 |
| 6 | Conclusion | 11 |
| 6.1 | Synthèse | 11 |
| 6.2 | Problèmes rencontrés | 11 |
| 6.3 | Remarques | 11 |

1 Introduction

L'objectif de ce projet est d'implémenter un algorithme DPLL pour 3-SAT vérifié avec Why3 comme décrit dans le sujet. L'archive contenant le code source est disponible sur GitHub.

Pour ce faire, nous avons dû compléter le squelette de code fourni, c'est-à-dire compléter la définition des fonctions scan dpll et sat du fichier dpll.mlw. Ainsi, dans les sections suivantes, on présentera l'implémentation et la vérification de ces fonctions.

2 Implémentation

Pour la fonction sat, je propose une implémentation légèrement différente de celle présentée dans le sujet :

```
let sat (mm: assignment) (cl: array cls) : (sat: bool)
=
  mm[0] <- 0;
  dpll mm cl (length mm) 1 (length cl)
```

À présent, en plus d'initialiser aux bonnes valeurs les paramètres de la fonction dpll, j'initialise la première case du tableau mm pour qu'il ait la bonne valeur afin d'être un *assignment* correct. Ainsi, je n'ai pas à gérer cette valeur "dégénérée" dans la fonction dpll, où j'aurais dû écrire une fonction annexe ou mettre à 0 la valeur de la case 0 de mm à chaque appel de fonction.

Pour les fonctions dpll et scan, je me suis basé sur les indications du sujet. Ainsi, je propose cette implémentation pour la fonction dpll :

```
let rec dpll (mm: assignment) (cl: array cls) (nv na nc: int) : (s: bool)
=
  let (b, mc) = scan mm cl nv na nc in
    if mc = 0 then begin
      for i = na to nv - 1 do
        mm[i] <- i
      done;
      true
    end
    else if (not b) || (na = nv) then false
    else begin
      mm[na] <- na;
      if dpll mm cl nv (na + 1) mc then true
      else begin
        mm[na] <- -na;
        dpll mm cl nv (na + 1) mc
      end
    end
  end
```

Ainsi, si toutes les clauses sont satisfaites, on s'assure que toutes les variables soient assi-

gnées, puis on renvoie true. Sinon, si l'*assignment* actuel ne satisfait pas toutes les clauses, alors on renvoie false. De plus, si toutes les variables sont assignées mais que toutes les clauses ne sont toujours pas satisfaites, alors on renvoie également false, car l'*assignment* actuel ne convient pas. Sinon, on assigne une nouvelle variable à false, puis à true si cela ne convenait pas.

Pour la fonction scan, je propose cette implémentation :

```
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
    let (l1, l2, l3) = cl[i] in
    if (abs l1 < na && mm[abs l1] <> l1) ||
       (abs l2 < na && mm[abs l2] <> l2) ||
       (abs l3 < na && mm[abs l3] <> l3)
    then begin
      mc <- mc - 1;
      swap cl i mc
    end
    else if (abs l1 < na && mm[abs l1] = l1) &&
           (abs l2 < na && mm[abs l2] = l2) &&
           (abs l3 < na && mm[abs l3] = l3)
    then
      return (false, mc)
    else
      i <- i + 1
  done;
  (true, mc)
```

Pour toutes les clauses encore non satisfaites, on regarde si elles sont satisfaites ou si elles sont fausses avec l'*assignment* actuel. Si la clause courante est satisfaites, alors on la retire de la zone active. Si la clause courante est fausse, alors on renvoie false. Si aucune clause n'est fausse avec l'*assignment* actuel, alors on renvoie true.

3 Fonction sat

Pour la fonction sat, j'ai repris exactement le même contrat que celui proposé dans le sujet :

```
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 { permute_all (old cl) cl }
```

On doit donc seulement supposer que le tableau *assignment* soit d'une longueur au moins égale à 1 et que tous les éléments du tableau des clauses soient bel et bien des clauses.

Avec ces hypothèses, on doit prouver que, si l'ensemble de clauses est satisfiable, alors toutes les variables doivent être assignées. De plus, si l'ensemble des clauses est satisfiable, alors toutes les clauses doivent être satisfaites avec cet *assignment*. En revanche, si l'ensemble de clauses n'est pas satisfiable, alors aucun *assignment* n'est possible pour satisfaire toutes les clauses en même temps. Enfin, on peut effectuer des opérations sur l'ensemble de clauses, mais celui-ci doit rester une permutation de l'ensemble de clauses initial.

4 Fonction dpll

Pour la fonction dpll, je propose le contrat suivant :

```
let rec dpll (mm: assignment) (cl: array cls) (nv na nc: int) : (s: bool)
  requires { nv = length mm }
  requires { 0 < na <= nv }
  requires { 0 <= nc <= length cl }

  requires { mm[0] = 0 }
  requires { is_assignment mm na }

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

  writes { cl }
  writes { mm }

  variant { nv - na }

  ensures { s -> is_assignment mm (length mm) }

  ensures { forall i. nc <= i < length cl -> cl[i] = old cl[i] }
  ensures { forall i. 0 <= i < na -> mm[i] = old mm[i] }
  ensures { s -> forall i. 0 <= i < length cl -> ok_cls mm nv cl[i] }

  ensures { permute_all (old cl) cl }
```

accompagné des invariants de boucle suivants :

```
for i = na to nv - 1 do
  invariant { forall j. 0 <= j < na -> mm[j] = old mm[j] }
  invariant { is_assignment mm i }
  ...
done
```

4.1 Pré-condition

J'ai déclaré les pré-conditions suivantes :

```
requires { nv = length mm }
requires { 0 < na <= nv }
requires { 0 <= nc <= length cl }

requires { mm[0] = 0 }
requires { is_assignment mm na }

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

Premièrement, `nv = length mm` signifie que `nv` est égal au nombre de variables plus 1. Deuxièmement, `0 < na <= nv` signifie que la première variable non assignée est comprise entre la première et la dernière variable. Troisièmement, `0 <= nc <= length cl` signifie que le nombre de clauses encore non satisfaites est compris entre 0 et le nombre total de clauses. Si les arguments concernés ne respectent pas ces conditions, alors ils ne sont pas valides, car ils seraient inconsistants avec la logique de l'algorithme. De plus, ces pré-conditions previennent des accès hors bornes des différents tableaux.

Ensuite, `mm[0] = 0` assure que la première case du tableau *assignment* est correctement initialisée. Cette condition nous permettra d'exprimer plus simplement les post-conditions et les invariants. De plus, `is_assignment mm na` assure que toutes les variables jusqu'à `na` exclu sont bien assignées. Par exemple, cette dernière condition est vraie avec `na = 1` si `mm[0] = 0`. Ces deux conditions sont essentielles pour garantir qu'à la fin de l'algorithme, si l'ensemble de clauses est satisfiable, alors toutes les variables sont assignées.

Enfin, `forall i. nc <= i < length cl -> ok_cls mm na cl[i]` exprime le fait que toutes les clauses dans la zone *inactive* sont satisfaites. Cette condition est vraie lorsque `nc = length cl`, car aucune clause n'est alors dans la zone *inactive*. Cette condition est essentielle pour prouver que, si l'ensemble de clauses est satisfiable, alors à la fin de l'algorithme toutes les clauses seront satisfaites.

4.2 Écriture

J'ai déclaré les écritures suivantes :

```
writes { cl }
writes { mm }
```

En effet, dans la fonction `scan`, on écrit dans le tableau `cl`, et dans la fonction `dpll`, on écrit dans le tableau `mm`. Il est donc nécessaire de déclarer ces écritures.

4.3 Variant

J'ai déclaré le variant suivant :

```
variant { nv - na }
```

En effet, na est initialisé entre 1 et nv . De plus, na est strictement croissant et tend vers nv . Ainsi, comme nv est fixé, l'expression $nv - na$ tend bien vers 0.

4.4 Post-condition

J'ai déclaré les post-conditions suivantes :

```
ensures { s -> is_assignment mm (length mm) }

ensures { forall i. nc <= i < length cl -> cl[i] = old cl[i] }
ensures { forall i. 0 <= i < na -> mm[i] = old mm[i] }
ensures { s -> forall i. 0 <= i < length cl -> ok_cls mm nv cl[i] }

ensures { permute_all (old cl) cl }
```

Tout d'abord, $s \rightarrow \text{is_assignment } mm (\text{length } mm)$ signifie que, si l'ensemble de clauses est satisfiable, alors toutes les variables sont assignées. Grâce à la pré-condition, cette post-condition est vérifiée jusqu'à na . On aura donc besoin de l'invariant de boucle dans le cas où $na < nv$, c'est-à-dire lorsque l'on doit compléter les *assignment* restantes.

Ensuite, premièrement, $\text{forall } i. nc \leq i < \text{length } cl \rightarrow cl[i] = \text{old } cl[i]$ signifie que l'on ne modifie plus l'ordre des clauses déjà satisfaites. Comme on écrit dans cl dans la fonction `scan`, cette propriété devra également être vérifiée dans la fonction `scan`. Deuxièmement, $\text{forall } i. 0 \leq i < na \rightarrow mm[i] = \text{old } mm[i]$ signifie que l'on ne modifie pas l'*assignment* déjà effectué avant l'entrée dans la fonction, c'est-à-dire que l'on n'assigne que les variables encore non assignées. Là encore, un invariant de boucle est nécessaire dans le cas où $na < nv$. Troisièmement, $s \rightarrow \text{forall } i. 0 \leq i < \text{length } cl \rightarrow \text{ok_cls } mm nv cl[i]$ signifie que, si l'ensemble de clauses est satisfiable, alors à la fin de l'algorithme toutes les clauses sont satisfaites. Pour prouver cette propriété, il faudra l'exprimer dans la fonction `scan`, car c'est elle qui vérifie la satisfiabilité des clauses. Les deux premières propriétés sont nécessaires uniquement pour prouver la troisième. En effet, si l'on ne modifie pas les clauses déjà satisfaites, que l'on ne change pas l'*assignment* déjà effectué, et que l'on progresse monotonement vers la satisfiabilité (cf. contrat de `scan`), alors on finira par satisfaire toutes les clauses si un *assignment* le permet.

Enfin, `permute_all (old cl) cl` exprime le fait que l'on effectue uniquement des *swaps* dans le tableau cl , et que l'on obtient donc une permutation de l'ensemble de clauses initial. Comme la fonction `dpll` ne modifie pas directement le tableau cl , cette propriété devra être prouvée dans la fonction `scan`.

4.5 Invariant de boucle

J'ai déclaré les invariants de boucle suivants :

```
for i = na to nv - 1 do
    invariant { forall j. 0 <= j < na -> mm[j] = old mm[j] }
    invariant { is_assignment mm i }
    ...

```

done

Tout d'abord, `forall j. 0 <= j < na -> mm[j] = old mm[j]` signifie que l'on ne modifie pas l'*assignment* déjà effectué avant l'entrée dans la fonction, c'est-à-dire que l'on n'assigne que les variables encore non assignées. Cette condition est vérifiée ici, car la boucle démarre à $i = na$ et progresse jusqu'à nv , ce qui implique que seules les variables non encore assignées sont modifiées. Cet invariant sert à prouver la stabilité de la post-condition `forall i. 0 <= i < na -> mm[i] = old mm[i]` au sein de la boucle.

Ensuite, `is_assignment mm i` assure que toutes les variables jusqu'à i exclu sont bien assignées. Cet invariant sert à prouver la post-condition $s \rightarrow \text{is_assignment } mm (\text{length } mm)$, qui garantit que toutes les variables sont assignées si l'ensemble de clauses est satisfiable. Comme i tend vers $nv = \text{length } mm$, on obtient bien `is_assignment mm (length mm)` lorsque l'ensemble de clauses est satisfiable.

5 Fonction scan

Pour la fonction `scan`, je propose le contrat suivant :

```
let scan (mm: assignment) (cl: array cls) (nv na nc: int) : (b: bool, mc: int)
  requires { nv = length mm }
  requires { 0 < na <= nv }
  requires { 0 <= nc <= length cl }
  requires { forall i. nc <= i < length cl -> ok_cls mm na cl[i] }

  writes { cl }

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

  ensures { permut_all (old cl) cl }
```

accompagné des invariants de boucle suivants :

```
while i < mc do
  variant { mc - i }

  invariant { 0 <= i <= mc <= nc }
  invariant { forall j. nc <= j < length cl -> cl[j] = old cl[j] }
  invariant { forall j. mc <= j < length cl -> ok_cls mm na cl[j] }

  invariant { permut_all (old cl) cl }
  ...
done
```

5.1 Pré-condition

J'ai déclaré les pré-conditions suivantes :

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

Ces pré-conditions sont déjà présentes dans la fonction `dpll`; leur explication et leur signification sont donc identiques. À noter que `forall i. nc <= i < length cl -> ok_cls mm na cl[i]` sera utile pour prouver que toutes les clauses de la nouvelle zone *inactive* sont déjà satisfaites.

5.2 Écriture

J'ai déclaré l'écriture suivante :

```
writes { cl }
```

En effet, dans la fonction `scan`, on écrit dans le tableau `cl`. Il est donc nécessaire de déclarer cette écriture.

5.3 Variant

J'ai déclaré le variant suivant :

```
while i < mc do
  variant { mc - i }
  ...
done;
```

En effet, `i` est initialisé à 0 et `mc` est initialisé à `nc`, donc à `length cl`. De plus, à chaque itération de la boucle qui n'est pas un `return`, soit `i` est incrémenté, soit `mc` est décrémenté. Ainsi, `mc - i` tend bien vers 0, car cette quantité est strictement décroissante et converge vers 0.

5.4 Post-condition

J'ai déclaré les post-conditions suivantes :

```
ensures { 0 <= mc <= nc }
ensures { forall i. nc <= i < length cl -> cl[i] = old cl[i] }
ensures { forall i. mc <= i < length cl -> ok_cls mm na cl[i] }

ensures { permute_all (old cl) cl }
```

Tout d'abord, `0 <= mc <= nc` signifie que la nouvelle zone *active* est de taille inférieure ou égale à l'ancienne, et inversement que la nouvelle zone *inactive* est de taille supérieure

ou égale à l'ancienne. Cette propriété est vérifiée car on ne modifie jamais les clauses déjà satisfaites et l'on ne peut que diminuer nc dans l'algorithme. On devra prouver la stabilité de cette propriété dans la boucle à l'aide d'un invariant. C'est cette propriété qui exprime le fait que l'on progresse uniquement vers la satisfiabilité.

Deuxièmement, `forall i. nc <= i < length cl -> cl[i] = old cl[i]` est déjà présente dans la fonction `dpll`; son explication est donc la même. Elle sera prouvée dans la boucle grâce à un invariant.

Troisièmement, `forall i. mc <= i < length cl -> ok_cls mm na cl[i]` signifie que toutes les clauses dans la nouvelle zone *inactive* sont satisfaites. Cette condition sera prouvée grâce à un invariant dans la boucle. Ces trois propriétés sont nécessaires pour prouver, dans la fonction `dpll`, que si l'ensemble de clauses est satisfiable, alors toutes les clauses le sont également.

Enfin, `permut_all (old cl) cl` est déjà présente dans la fonction `dpll`; son explication est donc identique. Cette propriété est prouvée dans la boucle grâce à un invariant, car c'est à cet endroit que l'on effectue les *swaps*.

5.5 Invariant de boucle

J'ai déclaré les invariants de boucle suivants :

```
while i < mc do
    invariant { 0 <= i <= mc <= nc }
    invariant { forall j. nc <= j < length cl -> cl[j] = old cl[j] }
    invariant { forall j. mc <= j < length cl -> ok_cls mm na cl[j] }

    invariant { permut_all (old cl) cl }
    ...
done
```

Tout d'abord, `0 <= i <= mc <= nc` signifie que la nouvelle zone *active* est plus petite ou égale à l'ancienne, et que les clauses traitées par la boucle appartiennent uniquement à la zone *active*. Cet invariant permet de prouver la post-condition `0 <= mc <= nc`.

Ensuite, `forall j. nc <= j < length cl -> cl[j] = old cl[j]` permet de prouver la stabilité de cette propriété à travers la boucle, car on ne modifie pas les clauses situées dans la zone *inactive*.

De plus, `forall j. mc <= j < length cl -> ok_cls mm na cl[j]` permet de prouver que toutes les clauses de la nouvelle zone *inactive* sont satisfaites par l'*assignment*. Cette propriété est préservée par la boucle, car on ne place dans cette zone que des clauses nouvellement satisfaites par l'*assignment*, en plus de celles déjà satisfaites auparavant (ce qui est assuré par la pré-condition).

Enfin, `permut_all (old cl) cl` assure que les *swaps* produisent uniquement une permutation des clauses à travers les différentes itérations de la boucle.

6 Conclusion

6.1 Synthèse

TODO

6.2 Problèmes rencontrés

TODO

6.3 Remarques

TODO