Décomposition formellement vérifiée de contraintes non binaires en contraintes binaires équivalentes

Catherine Dubois

Samovar-ENSIIE, Évry, France

Motivations

Nombreux domaines d'application (y compris critiques) utilisent la résolution de contraintes.

En particulier, beaucoup d'outils de vérification logicielle reposent sur la résolution de contraintes pour :

- prouver des propriétés fonctionnelles
- générer des tests

Solveurs SAT/SMT, solveurs CP(FD) : complémentaires pour certaines classes de problèmes.

Solveurs CP(FD) : efficaces, outils complexes

Confiance ? Crucial quand utilisés pour du logiciel critique.

Validation *a posteriori* :
- OK lorsque le solveur donne une solution, il suffit d'évaluer les contraintes ...
- mais ? ? ? ? quand le solveur répond `UNSAT`

Notre réponse : utiliser un solveur prouvé correct

Travail qui rejoint les objectifs d'autres travaux consacrés à la production d'outils de développement vérifiés : Compcert, Sel4 micro kernel, versat (SAT solver), SGBD ... vérifiés en utilisant des assistants à la preuve comme Coq, Isabelle.

Une présentation ultra-rapide de Coq

Qu'est-ce que Coq ?

- Un assistant à la preuve
- Un langage de programmation fonctionnel (récursion, types de données à la OCaml, pattern-matching, types)
- Un langage de spécification (langage d'ordre supérieur, prédicats inductifs *à la Prolog*)
- Un prouveur interactif, un vérificateur de preuve
- Un générateur de code exécutable (OCaml, Haskell)

A quoi ça sert ?

- Formaliser et vérifier des théorèmes (Théorème des quatre couleurs (2005) - Feit-Thompson (2012))
- Construire des logiciels certifiés/vérifiés formellement (Compcert : un compilateur C entièrement certifié (2008))

Mais ce n'est pas ...

- ... un prouveur automatique
- ... un oracle
- ... très facile à utiliser (malheureusement)

Confiance dans Coq ? oui largement utilisé et noyau de Coq (vérificateur de types) vérifié en Coq.

Vérification formelle

Solveur vérifié formellement

Crédit : Xavier Leroy

Vérification formelle

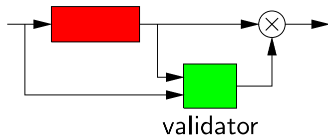
Solveur vérifié formellement

Crédit : Xavier Leroy

Vérification formelle

Validateur vérifié formellement

Solveur vérifié formellement

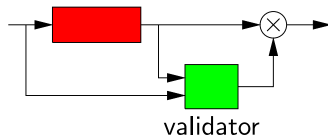


Crédit : Xavier Leroy

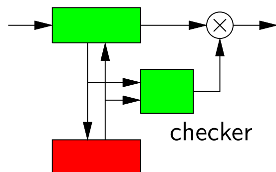
Vérification formelle

Solveur vérifié formellement

Validateur vérifié formellement



Outil externe avec validateur vérifié formellement



= formally verified

= not verified

Crédit : Xavier Leroy

Vérification formelle

Solveur vérifié formellement

+ solveur de référence
+ base pour étude d'extensions
- moyennement efficace
- gros effort de preuve

Crédit : Xavier Leroy

Précédemment, ...

- Un solveur CP(FD) formellement vérifié en Coq CoqbinFD [Carlier, Dubois, Gotlieb, 2012]
- pour contraintes binaires
- prouvé correct et complet
- générique, paramétré par le langage de contraintes
- mettant en œuvre les algorithmes AC3 et AC2001 au coeur de nombreux solveurs existants, reposant sur la consistance d'arc
- extrait en OCaml (code exécutable en OCaml)

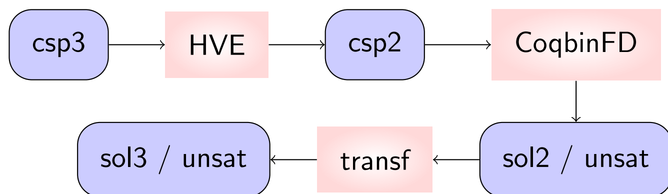
Précédemment, ...

- Un solveur CP(FD) formellement vérifié en Coq [Carlier, Dubois, Gotlieb, 2012]
- pour **contraintes binaires**
- prouvé correct et complet
- générique, paramétré par le langage de contraintes
- mettant en œuvre les algorithmes AC3 et AC2001 au coeur de nombreux solveurs existants, reposant sur la consistance d'arc
- extrait en OCaml (code exécutable en OCaml)

(et aussi un solveur similaire avec consistance de bornes)

Contribution

- Extension du solveur CoqbinFD aux contraintes n-aires (ternaires)
- Formalisation en Coq de l'encodage par variable cachée (Hidden Variable Encoding, HVE [Rossi et al, 1990])
  Correction et complétude du nouveau solveur démontrées en utilisant la correction et la complétude de CoqbinFD
- Point clé ici : généricité du solveur binaire CoqbinFD
  Instanciation avec un langage de contraintes particulier



Encodage des contraintes non-binaires

- 2 méthodes très connues : encodage dual et encodage par variable cachée (Hidden Variable Encoding / HVE)
- HVE : pour chaque contrainte non-binaire c,
  1. on ajoute une variable h_C,
  2. le domaine de h_c est l'ensemble des triplets qui satisfont la contrainte
  3. on remplace c par 3 contraintes de projection : *la ième projection d'un triplet de h_c est la valeur de la variable numéro i* $(1 \leq i \leq 3)$

csp3 $= (X_3, D_3, C_3)$
$X_3 = \{x_1, x_2, x_3, x_4, x_5, x_6\}$,
$D_3(v) = \{0, 1\}$ pour $v \in X$.
$C_3 = \{c_1 : x_1 + x_2 + x_6 = 1$,
$c_2 : x_1 - x_2 + x_4 = 1$,
$c_3 : x_4 + x_5 - x_6 \geq 1$,
$c_4 : x_2 + x_5 - x_6 = 0\}$.

HVE : csp2 $= (X, D, C)$
$X = X_3 \cup \{h_{c_1}, h_{c_2}, h_{c_3}, h_{c_4}\}$.
$D(h_{c_1}) = \{(0, 0, 1), (0, 1, 0), (1, 0, 0)\}$,
$D(h_{c_2}) = \{(0, 0, 1), (1, 0, 0), (1, 1, 1)\}$,
$D(h_{c_3}) = \{(0, 1, 0), (1, 0, 0), (1, 1, 0), (1, 1, 1)\}$
$D(h_{c_4}) = \{(0, 0, 0), (0, 1, 1), (1, 0, 1)\}$.
12 contraintes de projection, ex : $proj_1(h_{c_2}, x_1)$, $proj_3(h_{c_4}, x_6)$.

Notre processus de formalisation en Coq

1. Ecriture du solveur comme si on l'écrivait dans un langage fonctionnel
2. Ecriture des spécifications/propriétés (ici correction et complétude du solveur)
3. Preuve des propriétés (introduction de propriétés intermédiaires et preuve de lemmes intermédiaires)
4. Extraction du code OCaml (c'est le code développé en 1 traduit en OCaml)

Définition d'un csp3

Un CSP (ou réseau de contraintes) est un triplet (X, C, D) où

- X : un ensemble de variables de type `variable3` (type abstrait)
- C : un ensemble de contraintes (relations) sur les variables de X : binaires quelconques de type `basic_constraint` et ternaires de la forme `op X Y Z`
- D : une fonction qui associe à chaque variable de X son domaine (ensemble fini des valeurs possibles du type abstrait `variable3`).

En Coq :

```
Record network3 : Type := Make_csp3 {
  CVars3 : list variable3 ;
  Doms3 : mapdomain3 ;
  Csts3 : list constraint3 }.
```

avec *mapdomain*3 : type des tables indexées par des variables avec des valeurs de type *list value*3

Propriétés de bonne formation d'un réseau de contraintes :

`Record` *network_inv3 csp3* : `Prop` := *Make_csp_inv3* {

Dwf3 : $\forall$ *x*, *In x* (*Doms3 csp3*) $\leftrightarrow$ *In x* (*CVars3 csp3*) ;

Le domaine de la map contient les variables du csp, et uniquement celles-ci

Cwf1 : $\forall$ (*c* :*constraint3*) (*x* : *variable3*),
c $\in$ (*Csts3 csp3*) $\rightarrow$ *is_var_of_c3 x c3* $\rightarrow$
x $\in$ (*CVars csp*) ;

Les variables qui apparaissent dans les contraintes sont des variables du csp

Cwf2 : $\forall$ *x*, *x* $\in$ (*CVars3 csp3*) $\rightarrow$ $\exists$ *c*, *c* $\in$ (*Csts3 csp3*) $\wedge$
is_var_of_c3 x c3 ;

Toute variable du csp apparaît dans une contrainte au moins.

Cwf3 : $\forall$ *c*, *c* $\in$ (*Csts3 csp3*) $\rightarrow$ *diff_vars3 c* ;

Les variables d'une contrainte sont différentes

Norm3 : *norm3 csp3*

Le csp est normalisé

}.

Définition d'un csp binaire

Le solveur binaire est générique : le type des variables (variable), le type des valeurs (value), et le type des contraintes (constraint) sont des paramètres.

Ici ils sont instanciés.

Définition d'un csp binaire

Le solveur binaire est générique : le type des variables (variable), le type des valeurs (value), et le type des contraintes (constraint) sont des paramètres.

Ici ils sont instanciés.

```
Inductive variable :=
  OVar : variable3 → variable
| HVar : variable3 → OP → variable3 → variable3 → variable.
```

Une variable de csp2 est soit une variable originale soit une variable cachée

Définition d'un csp binaire

Le solveur binaire est générique : le type des variables (variable), le type des valeurs (value), et le type des contraintes (constraint) sont des paramètres.

Ici ils sont instanciés.

```
Inductive value :=
|Raw_value : value3 → value
|Triple : value3 → value3 → value3 → value.
```

Une valeur est soit une valeur originale soit un triplet

Définition d'un csp binaire

Le solveur binaire est générique : le type des variables (variable), le type des valeurs (value), et le type des contraintes (constraint) sont des paramètres.

Ici ils sont instanciés.

```
Inductive constraint : Set :=
  Basic : basic_constraint → constraint
| Triplefst : OP → variable3 → variable3 → variable3 → variable3 → constraint
| Triplesnd : OP → variable3 → variable3 → variable3 → variable3 → constraint
| Triplethd : OP → variable3 → variable3 → variable3 → variable3 → constraint
```

Une constrainte de csp2 est soit une contrainte binaire originale soit une contrainte de projection

Définition d'un csp binaire

Le solveur binaire est générique : le type des variables (variable), le type des valeurs (value), et le type des contraintes (constraint) sont des paramètres.

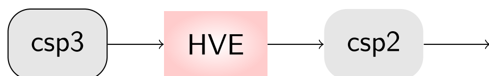
Ici ils sont instanciés.

```
Record network : Type := Make_csp {
CVars : list variable;
Doms : mapdomain;
Csts : list constraint }.
```

```
Record network_inv csp : Prop := ....
```

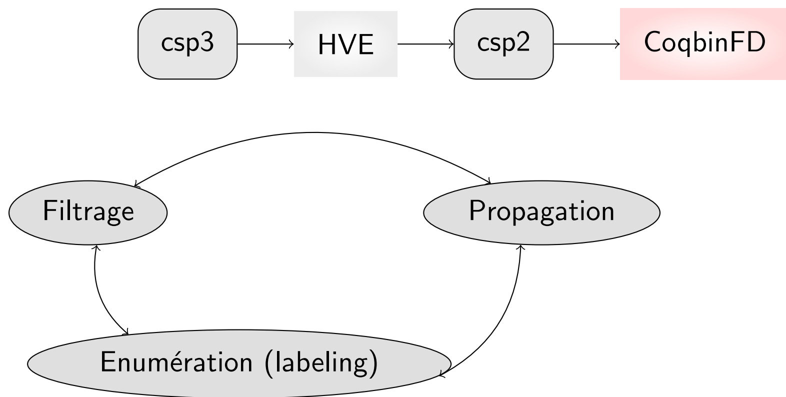
Définition du type d'un csp binaire + des propriétés de bonne formation (similaires aux précédentes)

HVE



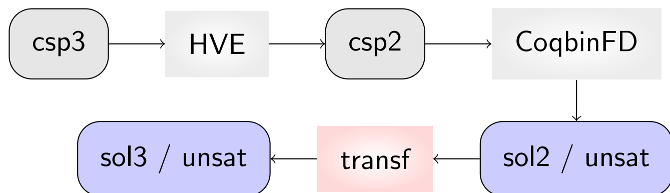

```
Definition translate_csp3 csp3 :=
match csts3Tocsts2 csp3.Csts3 csp3.Doms3 with
| None ⇒ None
| Some (cs, lvdv) ⇒ Some (Make_csp
      (List.map (fun x ⇒ OVar x) (csp3.CVars3) ++ List.map first
lvdv)
      (new_domain (map3_to_raw (csp3.Doms3) (csp3.CVars3)) lvdv)
      cs)
end.
```

CoqbinFD



utilise AC3 - AC 2001
fournit une fonction `solve_csp` (unsat ou une solution)
prouvée correcte et complète

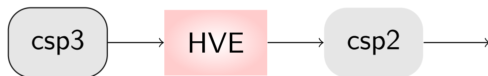
Calcul du résultat : une solution ou unsat



translate_sol : *assign* → *list variables3* → *assign3*

`translate_sol a l` : reconstruit une solution en filtrant dans `a` les variables de `l`

Lemmes et théorèmes clés



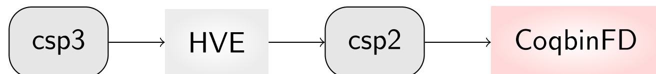
Nombreux lemmes techniques sur HNE

`Lemma` *csp3Tocsp2_basic_bin* : $\forall$ *csp3 csp*,
network3_inv csp3 $\rightarrow$ *translate_csp3 csp3* = *Some csp* $\rightarrow$
($\forall$ *b* : *basic_constraint*, *b* $\in$ (*Csts csp*) $\longleftrightarrow$ *b* $\in$ (*Csts3 csp3*)).

Un théorème complémentaire :

`Lemma` *HA_HVE_AC* : $\forall$ *csp3 csp*,
network3_inv csp3 $\rightarrow$ *translate_csp3 csp3* = *Some csp* $\rightarrow$
HA_consistent3_csp csp3 $\rightarrow$ *arc_consistent_csp csp*.

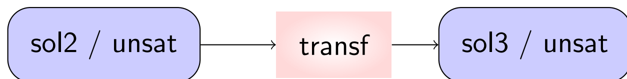
Lemmes et théorèmes clés



La fonction *solve_csp* de CoqbinFD prend 2 paramétres : un csp binaire et la preuve que celui-ci est bien formé, preuve obtenue par le lemme suivant.

`Lemma` *translate_csp3_network_inv* : $\forall$ *csp3 csp*,
network3_inv csp3 $\rightarrow$ *translate_csp3 csp3* $=$ *Some csp* $\rightarrow$
network_inv csp.

Lemmes et théorèmes clés

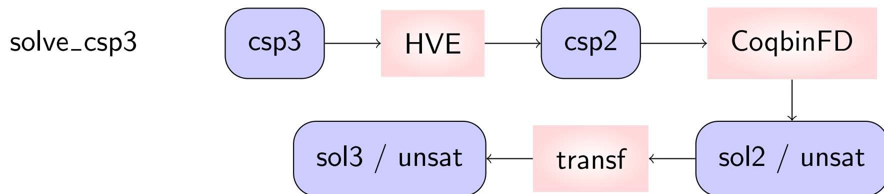


`Lemma` *translate_sol* : $\forall$ *csp3 csp a a3* ,
network3_inv csp3 $\rightarrow$ *translate_csp3 csp3* $=$ *Some csp* $\rightarrow$
solution a csp $\rightarrow$ *solution3* (*translate_sol a*) *csp3*.

`Lemma` *translate_nosol* : $\forall$ *csp3 csp* ,
network3_inv csp3 $\rightarrow$ *translate_csp3 csp3* $=$ *Some csp* $\rightarrow$
($\forall$ *a*, $\neg$ (*solution a csp*)) $\rightarrow$ $\forall$ *a3*, $\neg$ (*solution3 a3 csp3*).

`Lemma` *translate_complete* : $\forall$ *a3 csp3 csp*,
network3_inv csp3 $\rightarrow$ *translate_csp3 csp3* $=$ *Some csp* $\rightarrow$
solution3 a3 csp3 $\rightarrow$ *solution* (*translate_sol3 a3 csp3*) *csp*.

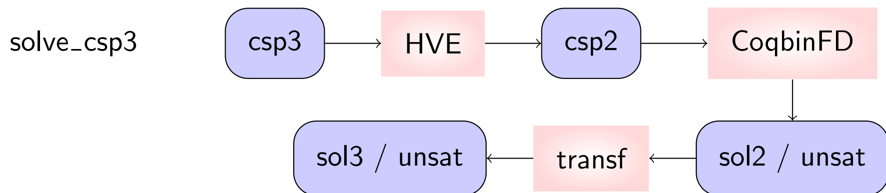
Correction et complétude du solveur étendu



`Theorem` *solve3_correct_sol* : $\forall$ *csp3* (*Hinv3* : *network3_inv csp3*) *a3*, *solve_csp3 csp3 Hinv3* = *Some a3* $\rightarrow$ *solution3 a3 csp3*.

`Theorem` *solve3_correct_nosol* : $\forall$ *csp3* (*Hinv3* : *network3_inv csp3*), *solve_csp3 csp3 Hinv3* = *None* $\rightarrow$ ($\forall$ *a3*, $\neg$*solution3 a3 csp3*).

Correction et complétude du solveur étendu



`Theorem` *solve3_complete_sol* : $\forall$ *csp3* (*Hinv3* : *network3_inv csp3*) *a3*, (*solution3 a3 csp3*) $\rightarrow$
 $\exists$ *a3*, *solve_csp3 csp3 Hinv3* = *Some a3* $\wedge$ *solution3 a3 csp3*.

`Theorem` *solve3_complete_unsat* : $\forall$ *csp3* (*Hinv3* : *network3_inv csp3*), ($\forall$ *a3*, $\neg$ (*solution3 a3 csp3*)) $\rightarrow$ *solve_csp3 csp3 Hinv3* = *None*.

Effort de preuve

- code CoqbinFD : 8500 loc
- décomposition en contraintes binaires : 3000 loc
  dont 70 loc (paramètrès génériques), 300 loc (instanciation des paramètres génériques de CoqbinFD)
- 60 définitions de fonctions (dont 33 uniquement pour les preuves), 12 définitions inductives et 110 lemmes et théorèmes.

Expérimentations du solveur extrait

- Quelques exemples jouets
- Golomb : golomb 6 4, golomb 5 4, golomb 7 4, golomb 10 6, golomb 16 6, golomb 17 6, golomb 24 7 (unsat, 100 s), golomb 25 7 (23 s) golomb n p : existe-t-il une règle de Golomb de longueur n avec p marques ?
- Problème normalized-g-2X2 (XCSP 2.1) (4 variables and 4 contraintes ternaires)
- Problème normalized-graceful-K2-P3 (15 variables and 60 contraintes dont 9 ternaires) (0,5 s)

Conclusion et perspectives

Extension du solveur binaire : formellement vérifiée en Coq, exécutable OCaml

Perspectives

- Généralisation aux contraintes n-aires
- Décomposition plus spécialisée
- Utilisation de structures de données plus efficaces (par exemple, hashing, représentation plus efficace des domaines)
- Formalisation d'autres propriétés de consistence
- Formalisation de contraintes globales (*alldifferent* en cours)
- Structuration (modules/*features*) plus flexible pour construire un solveur à la carte

Formalisons, formalisons, formalisons, ...

Formalisons, formalisons, formalisons, ...

mais tous ensemble dans Caviar