Solveurs CP(FD) vérifiés formellement

Catherine Dubois[1,2] **Arnaud Gotlieb**[2,3]

[1] CEDRIC-ENSIIE, Évry, France
[2] Inria, France
[3] Certus V&V Center, SIMULA RESEARCH LAB., Lysaker, Norway
dubois@ensiie.fr arnaud@simula.no

Résumé

Les solveurs de contraintes sont utilisés pour résoudre des problèmes d'optimisation, de planification, d'ordonnancement, etc. Leur utilisation dans des domaines critiques réclame un certain degré de confiance et requiert un examen sceptique des résultats fournis par un solveur, tout particulièrement lorsque ce dernier assure qu'un problème n'a pas de solution. Nous proposons de développer un solveur pour les domaines finis - CP(FD) - correct par construction. Nous avons implanté le solveur à l'aide de l'outil de preuve Coq et l'avons prouvé correct par rapport à sa spécification elle-même exprimée en Coq. Il implante l'algorithme AC3 (et AC2001) et met en œuvre une consistance d'arcs. Le mécanisme d'extraction de Coq permet d'obtenir un solveur écrit en OCaml, formellement vérifié, le premier à notre connaissance. Ce solveur peut être utilisé pour résoudre un problème ou encore pour vérifier les résultats d'un autre solveur non vérifié formellement. Ces résultats ont été publiés récemmment dans les actes de la conférence internationale FM 2012 [1]. Nous présentons ici un résumé de ces travaux et étudions leur extension à la consistance de bornes.

Abstract

Constraint solvers are used to solve problems coming from optimization, scheduling, planning, etc. Their usage in critical applications implies a more skeptical regard on their implementation, especially when the result is that a constraint problem has no solution, i.e., unsatisfiability. In this paper, we propose an approach aiming to develop a correct-by-construction finite domain based -CP(FD)- solver. We developed this solver within the Coq proof tool and proved its correctness in Coq. It embeds the algorithm AC3 (and AC2001) and uses arc-consistency. The Coq extraction mechanism allows us to provide a finite domain based solver written in OCaml formally verified, the first one to our knowledge. This solver can be used directly or as *a second shot* solver to verify results coming from an untrusted solver. This result has recently been published in FM 2012 [1]. In this paper, we present a summary of this result and extend it to deal with another local-consistency property, namely, bound-consistency.

1 Introduction

Les solveurs de contraintes à domaines finis (CP(FD) dans la suite) ont pour but de rechercher des solutions à des problèmes exprimés sous la forme de triplets (X, C, D) où X est un ensemble de variables, C un ensemble de contraintes et D associe à chaque variable son domaine, ici un ensemble fini de valeurs possibles. Une solution affecte à chaque variable de X une valeur de son domaine de telle sorte que les contraintes soient toutes satisfaites. Lorsqu'un solveur fournit une solution ou répond UNSAT indiquant ainsi que le problème soumis n'a pas de solution, avons nous confiance dans ces résultats ? Il est souvent facile de vérifier le résultat dans le premier cas, en exécutant les contraintes avec la ou les solutions fournies. Ceci peut néanmoins nécessiter de mettre en œuvre un programme qui fait cette vérification. Mais lorsque le problème est déclaré insatisfaisable i.e. sans solution, comment faire ? La vérification *a posteriori* précédente n'est plus possible. La confiance dans un solveur est cruciale lorsque ce dernier est utilisé dans des applications sensibles voire critiques . Notons également que les solveurs CP(FD) sont fortement utilisés dans des outils de vérification de logiciels [2] ou de génération de tests [4], eux-mêmes utilisés pour vérifier des applications critiques. La complexité des solveurs actuels rend impossible la vérification formelle directe de leur code. Une technique alternative consisterait à faire produire par le solveur une trace post-mortem, qu'un vérificateur (vérifié formellement) pourrait ensuite analyser

pour certifier la correction des résultats fournis. Cette approche requiert cependant de développer un langage approprié de traces ou d'identifier le bon degré d'abstraction des traces car les langages de trace existants ont été développés pour la mise au point des solveurs à contraintes [5] et non pour la preuve formelle. Nous proposons une autre approche qui consiste à développer un solveur certifié formellement, i.e. prouvé correct. Pour cela, nous utilisons l'outil de preuve Coq [8] pour spécifier, programmer et prouver la correction du solveur. Grâce au mécanisme d'extraction de Coq, de ce développement est extrait automatiquement un solveur écrit en OCaml, formellement vérifié (ou certifié) par construction. Ce solveur peut être utilisé directement pour résoudre un problème, il peut aussi être utilisé pour vérifier les résultats d'un autre solveur non vérifié formellement. Nous avons appliqué cette approche pour fournir un solveur CP(FD) formellement certifié s'appuyant sur la consistance d'arc pour des contraintes binaires. A notre connaissance il s'agit du premier solveur CP(FD) formellement vérifié. Ces résultats ont été publiés dans [1]. Ces questions de confiance dans les résultats des solveurs ont été abordées pour les solveurs SAT et SMT avec des approches à base de traces [3] ou avec des approches similaires à la nôtre [6, 7].

Dans la section 2, nous présentons brièvement l'outil de preuve Coq et la méthodologie suivie. Puis la section 3 résume les points principaux de la formalisation et de la preuve de correction des algorithmes de filtrage et de propagation (voir [1] pour plus de détails). Nous avons également développé un processus d'énumération élémentaire qui, avec les processus de filtrage et de propagation, fournit un solveur correct et complet. Enfin dans la section 4, nous étudions l'extension des spécifications et preuves précédentes à la consistance de bornes.

2 Coq et la méthodologie utilisée

L'outil de preuve Coq [8] permet d'écrire des spécifications et d'énoncer des théorèmes dans un langage proche des mathématiques fortement expressif, puis de prouver ces théorèmes et enfin de vérifier ces preuves. Il ne s'agit pas d'un outil de preuve automatique, il requiert l'interaction avec l'utilisateur mais offre toutefois des procédures de décision qui peuvent être utilisées pour prouver automatiquement certaines formules. C'est aussi un langage de programmation fonctionnel typé qui propose, entre autres, des fonctions simples ou récursives, des types inductifs et du *pattern-matching*. Il impose que toutes les fonctions terminent, que tous les cas d'un pattern-matching soient traités. La programmation en Coq induit donc une programmation plus sûre. Nous avons utilisé ce langage pour implanter les différentes fonctions (par exemple la fonction `revise` qui supprime du domaine d'une variable les valeurs impossibles pour une contrainte donnée) qui composent le solveur comme si nous l'avions écrit directement dans un langage fonctionnel, OCaml par exemple. Nous avons défini des spécifications, la consistance d'arc par exemple, et décrit en Coq des propriétés concernant les fonctions précédentes. Par exemple, *l'application de `revise` sur les variables qui apparaissent dans une contrainte assure l'arc-consistance pour cette contrainte*. Puis nous avons prouvé ces propriétés.

Il est possible d'exécuter directement dans Coq les fonctions écrites en Coq. Néanmoins on n'obtient pas l'efficacité que l'on pourrait obtenir avec un programme écrit directement en OCaml. On peut utiliser le mécanisme d'extraction de Coq pour obtenir un programme OCaml sémantiquement équivalent. Ce mécanisme permet d'extraire le contenu calculatoire d'une preuve, en effaçant ce dernier. Dans notre cas, les propriétés et les preuves sont effacées, les fonctions composant le solveur sont traduites en OCaml.

3 Solveur CF(FD) pour consistance d'arc

Nous résumons ici le développement réalisé en Coq (fonctions, propriétés et preuves) présenté dans [1]. Un point important concerne la généricité de ce développement. Le solveur est en effet paramétré par le type des variables et des valeurs et aussi par le langage de contraintes. En Coq, ces types sont abstraits, supposés équipés d'une égalité décidable. On suppose également que la sémantique des contraintes est donnée par une fonction d'interprétation qui évalue la contrainte en fonction des valeurs de ses deux variables. On requiert aussi la définition d'une fonction qui associe à toute contrainte ses deux variables. Ces types et fonctions sont à définir directement en OCaml pour pouvoir utiliser le solveur extrait dans un contexte particulier.

Un système de contraintes (csp) est alors défini par un enregistrement composé d'une liste finie de variables (champ `X`), d'une liste finie de contraintes (`C`) et d'une table (`D`) associant à chaque variable son domaine, ici une liste finie de valeurs (dans la suite nous utilisons la notation pointée usuelle pour accéder aux différents champs). Un prédicat spécifie la bonne formation d'un csp : le domaine de définition de `D` est exactement `X`, les variables apparaissant dans les contraintes sont exactement celles de `X`, enfin les contraintes sont normalisées. Les deux premières contraintes sont généralement implicites dans la littérature, elles doivent être explicitées lorsqu'il s'agit de preuve formelle. De nombreux théorèmes prendront en

hypothèse la bonne formation du csp.

Il est classique de représenter un csp par un graphe de contraintes symétrique. Nous avons adopté cette vue dans notre formalisation Coq. Ainsi chaque contrainte c de variables x et y donne lieu à deux arcs, l'un reliant le sommet x au sommet y étiqueté par c (noté dans la suite $c(x, y)$) et l'autre reliant y à x également étiqueté par c (noté $c(y, x)$).

Nous avons implanté le solveur à l'aide de 3 fonctions récursives, l'une réalise le filtrage (`revise`), une autre la propagation (`ac3` qui appelle la précédente) et la dernière réalise l'énumération (`labeling` qui utilise `ac3`). Nous décrivons les deux premières à la figure 1. Ces fonctions suivent les présentations usuelles de la littérature adaptées au paradigme fonctionnel. La fonction `ac3` met en œuvre une récursion plus complexe que celle de `revise`, elle requiert explicitement une preuve de terminaison alors que la preuve de terminaison de `revise` est faite automatiquement par Coq. Hormis quelques fonctions auxiliaires et la fonction d'énumération non présentée ici, le code du solveur est complet. Le code extrait en OCaml correspond à cette partie à laquelle est ajoutée la traduction en OCaml des fonctions de la librairie utilisées (comme celles manipulant les listes et les tables).

Il nous faut exprimer la notion de consistance locale mise en œuvre dans ce processus de résolution, à savoir ici la consistance d'arc. Nous formalisons la propriété, sous la forme du prédicat `loc_consistent` $c\ x\ y\ D$, de la manière suivante : l'arc $c(x, y)$ est arc-consistant pour D si et seulement si pour toute valeur u de $D(x)$, il existe au moins une valeur (appelée support) v dans $D(y)$ telle que $c[x/u, y/v]$ (i.e la contrainte quand on a remplacé x par u et y par v) est satisfaite.

La correction du filtrage consiste à montrer que la consistance locale est établie après une étape de filtrage (voir le théorème *revise_loc_consistent* à la figure 1). Quant à la complétude, elle exprime que si on dispose d'une solution compatible avec une table de domaines D et si une étape de filtrage amène à réduire un domaine alors la solution initiale est toujours compatible avec les nouveaux domaines. On ne perd pas de solution ! La preuve repose sur un lemme qui consiste à montrer que les valeurs éliminées du domaine touché par le filtrage ne peuvent pas être éléments d'une solution. La correction d'`ac3` consiste à montrer que l'étape de propagation conduit à des domaines tels que tous les arcs du graphe des contraintes sont arc-consistants. La preuve de correction repose sur la correction du filtrage et aussi sur des lemmes relatifs à la fonction `visitagain`. Par exemple on démontre (A) que si l'arc $c(x, y)$ a provoqué une réduction du domaine de x alors la consistance des arcs qui ne sont pas dans `visitagain` $x\ y\ g$ n'est pas affectée.

La complétude de `ac3` consiste à montrer que cette fonction ne perd pas de solution ; elle utilise la complétude du filtrage.

En nous limitant au filtrage et à la propagation, le code Coq complet (définitions et preuves) représente environ 5300 lignes. L'adaptation à AC2001 requiert principalement de modifier la fonction `revise` de manière à lui faire gérer une table qui garde les supports. Les deux développements sont en fait deux instances d'un même développement modulaire, permettant de partager une grande partie du code.

4 Vers un solveur certifié pour la consistance de bornes

Lorsque les domaines des variables sont trop grands, la recherche d'un support pour chaque valeur d'un domaine peut devenir trop coûteuse. Une alternative est alors d'utiliser la consistance de bornes qui consiste à raisonner sur les bornes des domaines. Ainsi rechercher la consistance de bornes pour un arc $c(x, y)$ consiste à trouver la borne inférieure (resp. supérieure) du domaine de x, m_x (resp. M_x), telle que la contrainte soit vérifiée. Le domaine de chaque variable x est dorénavant représenté par le couple (m_x, M_x). La fonction de filtrage est redéfinie de manière à modifier éventuellement les bornes du domaine d'une variable. Quant à la fonction de propagation, elle reste identique, à l'adaptation près de la fonction `visitagain` : en effet après le filtrage de l'arc $c(x, y)$, la propagation requiert de visiter également l'arc inverse $c(y, x)$ (en plus de ceux nécessités par l'arc-consistance). Les preuves de correction et de complétude de la propagation restent identiques, si on peut fournir les preuves de correction et de complétude de la fonction de filtrage ainsi que celle de la propriété (A) énoncée à la section précédente.

Plus généralement, on obtient une formalisation (spécification et preuve) modulaire et générique de la fonction de propagation, paramétrée par la représentation des domaines, la notion de consistance locale et la fonction de filtrage. Elle peut être instanciée pour la consistance d'arc ou la consistance de bornes.

Jusque là, nous n'avons fait aucune hypothèse quant à la forme des contraintes. Or, dans le filtrage de $c(x, y)$, le domaine de y doit être parcouru pour trouver un support pour m_x et M_x, ce qui n'est pas efficace. En nous plaçant dans le cadre d'un langage de contraintes numériques, nous proposons de rendre ce filtrage plus efficace en utilisant des fonctions de projection qui permettent de mettre à jour les bornes des domaines des variables, sans parcourir les domaines. Afin de réutiliser les développements précédents nous conservons la méthode de filtrage arc par arc. Nous pouvons réutiliser toutes les définitions et les preuves

- x et y sont les variables de c, de domaine resp. dx et dy
- *revise c x y dx dy = (b, dx')*
si $b = true$, dx' est le domaine de x après filtrage ($dx' \subsetneq dx$), sinon $dx' = dx$.

```
Function revise c x y dx dy {...} :=
    match dx with
      nil ⇒ (false, dx)
    | v ::r ⇒ let (b, d) := revise c x y r dy in
              if exists_support c v dy
              then (b, v ::d)
              else (true, d)
    end.
```

avec *exists_support c v dy* = true si il existe $t \in dy$ tel que $c[x/u, y/t]$ soit satisfaite, false sinon.

```
Theorem revise_loc_consistent : ∀ csp c x y ,
    c ∈ csp.C → x ∈ (vars_of c) → y ∈ (vars_of c) →
    ∀ dx dy dx' b,
    csp.D(x) = Some dx → csp.D(y) = Some dy →
    revise c x y dx dy = (b, dx') →
      loc_consistent x y c (csp.D(x) ← dx').
```

avec $(D(x) \leftarrow dx')$, la table obtenue à partir en D en modifiant le domaine de x qui devient dx'.

- g : graphe des contraintes, D : table des domaines, qu : liste des arcs à visiter (*worklist*)
- Si découverte d'un domaine vide, alors *ac3 g D qu* = *None* sinon, *ac3 g D qu* = *Some D'* : la propagation se termine avec les domaines D'

```
Function ac3 g D qu { ...} :=
match qu with
| nil ⇒ Some (D)
| (x, c, y) : :r ⇒
  match D(x), D(y) with
  | Some dx, Some dy ⇒
    let (b, dx') := revise c x y dx dy in
      if b then
        if is_empty dx' then None
        else ac3 g (D(x) ← dx') (r ⊕ (visitagain x y g))
      else ac3 g D r
  | _, _ ⇒ None
  end
end
```

avec $\oplus$ = concaténation stricte de deux listes et *visitagain x y g* = liste des arcs à revisiter = liste de tous les arcs de la forme $c'(z, x)$ avec $z \neq y$
Appel initial : qu = liste de tous les arcs de g.

FIGURE 1 – `revise` et `ac3` en Coq

concernant la propagation moyennant les preuves requises sur le filtrage, ici reportées au niveau de chaque fonction de projection. Il convient en particulier de montrer que chaque fonction de projection assure la consistance de bornes et ne perd pas de solution. Ces preuves sont actuellement en cours.

5 Conclusion et perspectives

Ce papier résume les travaux que nous avons menés sur le développement d'un solveur de contraintes sur les domaines finis, formellement certifié en Coq. A notre connaissance, c'est la première fois qu'un tel développement est proposé. La difficulté de ces travaux réside dans la spécification méticuleuse de toutes les fonctions du solveur, et dans les preuves de correction et complétude associées. Une perspective à court terme de ce travail concerne le traitement de contraintes ternaires et N-aires, qui nécessite une révision partielle de la spécification. A plus long terme, nous envisageons d'utiliser ce solveur certifié dans le contexte de la vérification formelle de code.

Références

[1] M. Carlier, C. Dubois, and A. Gotlieb. A certified constraint solver over finite domains. In *Formal Methods (FM)*, volume 7436 of *LNCS*, pages 116–131, Paris, 2012.

[2] H. Collavizza, M. Rueher, and P. Van Hentenryck. Cpbpv : A constraint-programming framework for bounded program verification. *Constraints Journal*, 15(2) :238–264, 2010.

[3] A. Van Gelder. Verifying rup proofs of propositional unsatisfiability. In *Int. Symp. on Artificial Intelligence and Mathematics (ISAIM), Fort Lauderdale, USA*, 2008.

[4] A. Gotlieb, B. Botella, and M. Rueher. A CLP Framework for Computing Structural Test Data. In *Computational Logic (CL)*, pages 399–413, London, 2000.

[5] L. Langevine, P. Deransart, and M. Ducassé. A generic trace schema for the portability of cp(fd) debugging tools. In *Int. Workshop on Constraint Solving and Constraint Logic Programming (CSCLP 2003), Budapest, Hungary, Selected Papers*, volume 3010 of *LNCS*. Springer, 2004.

[6] S. Lescuyer and S. Conchon. A Reflexive Formalization of a SAT Solver in Coq. In *Emerging Trends - Int. Conf. on Theorem Proving in Higher Order Logics (TPHOLs)*, 2008.

[7] F. Maric. Formal verification of a modern sat solver by shallow embedding into isabelle/hol. *Theor. Comput. Sci.*, 411(50) :4333–4356, 2010.

[8] The Coq Development Team. *Coq, version 8.4*. Inria, August 2012. http://coq.inria.fr/.