# A Certified Constraint Solver over Finite Domains

Matthieu Carlier[1], Catherine Dubois[1,2], and Arnaud Gotlieb[3,4]

[1] ENSIIE, Évry, France
{carlier,dubois}@ensiie.fr
[2] INRIA Paris Rocquencourt, Paris, France
[3] Certus V&V Center, SIMULA RESEARCH LAB., Lysaker, Norway
arnaud@simula.no
[4] INRIA Rennes Bretagne-Atlantique, Rennes, France

**Abstract.** Constraint programs such as those written in modern Constraint Programming languages and platforms aim at solving problems coming from optimization, scheduling, planning, etc. Recently CP programs have been used in business-critical or safety-critical areas as well, e.g., e-Commerce, air-traffic control applications, or software verification. This implies a more skeptical regard on the implementation of constraint solvers, especially when the result is that a constraint problem has no solution, i.e., unsatisfiability. For example, in software model checking, using an unsafe constraint solver may result in a dramatic wrong answer saying that a safety property is satisfied while there exist counter-examples. In this paper, we present a Coq formalisation of a constraint filtering algorithm — AC3 and one of its variant AC2001 — and a simple labeling procedure. The proof of their soundness and completeness has been completed using Coq. As a result, a formally certified constraint solver written in OCaml has been automatically extracted from the Coq specification of the filtering and labeling algorithms. The solver, yet not as efficient as specialized existing (unsafe) implementations, can be used to formally certify that a constraint system is unsatisfiable.

## 1 Introduction

**Context.** Automated software verification relies on constraint resolution [23], either to prove functional properties over programs or to generate automatically test inputs [13]. For example, formal verification involves showing that a formula embedding the negation of a property is *unsatisfiable*, i.e., the formula has no model or solution. While most verification techniques are based on SAT and SMT (Satisfiability Modulo Theory), tools built over *Constraint Programming over Finite Domains*, noted CP(FD) [14], become more and more competitive e.g., CPBPV [11], or OSMOSE [5,4]. In this context, *finite domains* mean finite sets of labels or possible values associated to each variable of the program. Existing results show that CP(FD) is a complementary approach to SMT for certain classes of verification problems [5,3].

**Problem.** Effective constraint-based verification involves using efficient constraint solvers. However, efficiency comes at the price of complexity in the design of these solvers. And even if developing CP(FD) solvers is the craft of a few great specialists, it is nearly impossible to guarantee by manual effort that their results are error-free. A constraint solver declaring a formula being unsatisfiable while it is not the case, can entail dramatic consequences for a safety-critical software system. Thus, an emerging trend in software verification is to equip code with correctness proofs, called *certificates* [12], that can be checked by third-party certifiers [20,8,1,7]. As soon as these certificates involve finite domains constraint systems, external constraint solvers are used without any guarantee on their results.

**Contribution.** Following the research direction opened up by CompCert [17] that offered us a formally certified compiler for a subset of C, the work presented in this paper is part of a bigger project aiming at building a certified testing environment for functional programs based on finite domains constraint solving. A significant first step has been reached by formally certifying the test case generation method [9], provided that a correct constraint solver is available. This paper specifically tackles this second step of the project by building a certified CP(FD) solver. We developed a sound and complete CP(FD) solver able to provide correct answers, relying on the Coq interactive proof assistant. The constraints are restricted to binary normalized constraints, i.e., distinct relations over two variables [10], but are not necessarily represented as set of binary tuples. The language of constraints is in fact a parameter of our formalisation. Our certified CP(FD) solver implements a classical filtering algorithm, AC3 [19] and one of its extension AC2001 [10], thus focuses on arc-consistency. By *filtering algorithm*, we mean a fixpoint computation that applies domain filtering operators to the finite domains of variables. The Coq formalisation is around 8500 lines long. The main difficulties have been to discover or re-discover implicit assumptions and classical knowledge about these algorithms.

Following the Coq proof extraction mechanism, the executable code of the solver in OCaml has been automatically derived from its formal development, and used to solve some constraint systems. The solver, yet not as efficient as specialized existing but unsafe implementations, can be used to formally certify that a constraint system is unsatisfiable or satisfiable. According to our knowledge, this is the first time a constraint solver over finite domains is formally certified. The Coq code and the OCaml extracted files are available on the web at `www.ensiie.fr/~dubois/CoqsolverFD`.

**Outline.** The rest is organized as follows: Sec. 2 introduces the notations and the definitions of the notions of consistency, solution, solving procedure used in our formalisation. Sec. 3 presents the filtering algorithm AC3 and an implementation of the local consistency property called REVISE. It also presents an optimized version of the filtering algorithm called AC2001. Sec. 4 describes the formalisation of the search heuristics. Sec. 5 presents our first experimental results and discusses related work. Finally, sec. 6 concludes the paper.

## 2 Formalisation of a Constraint Solving Problem

A *Constraint Satisfaction Problem* (csp for short) or network of constraints [19] is a triple $(X, D, C)$ where $X$ is a set of ordered variables, $C$ is a set of binary normalized constraints over $X$ and $D$ is a partial function that associates a finite domain $D(x)$ to each variable $x$ in $X$. In our setting, the values of a finite domain belong to a set $\mathcal{V}$ equipped with a decidable equality. The set $C$ is composed of *binary* and *normalized constraints*, meaning respectively that constraints hold over 2 variables, and that two distinct constraints cannot hold over exactly the same variables. The function *get_vars* retrieves the ordered pair of variables of a constraint $c$. For example, $get\_vars(c)$ returns $(x_1, x_2)$ iff $x_1$ is smaller than $x_2$, $(x_2, x_1)$ otherwise. Note that this ordering is introduced for convenience, but does not limit the generality of the purpose. We also suppose that each variable of $X$ appears at least once in a constraint of $C$. Restricting to binary normalized constraints does not weaken the contribution as constraints over finite domains with higher arity can always be rewritten into binary constraints [2], and it is always possible to merge two constraints holding over the same two variables into a single one. Omitting unary constraints is not a restriction either since unary constraints are semantically equivalent to domain constraints, that are captured by $D$ in our formal settings

Fig.1 shows the Coq formalisation of constraint network where types of constraints, variables and values are made abstract. To define constraints, we only require the definition of 2 functions *get_vars* and an interpretation function *interp*. We expect the following meaning: if $get\_vars(c) = (x, y)$, then $interp\ c\ u\ v\ = true$ iff $c$ is satisfied by substituting $x$ by $u$ and $y$ by $v$, noted $c(u, v)$ or (*consistent_value c x u y v* ) in Coq code. In the following, Coq excerpts are not true Coq code in the sense that mathematical notations are used when they ease the reading, e.g. $\in$ denotes list or set membership, whereas prefixe notation *In* is kept for membership in domain tables. The formalisation of domains *Doms* is

```
Parameter constraint : Set.
Parameter interp : constraint → value → value → bool.
Parameter get_vars : constraint → variable × variable.
Parameter get_vars_spec : ∀ c x1 x2, get_vars c = (x1, x2) → x1 < x2.
Record network : Type := Make_csp {
CVars : list variable ; Doms : mapdomain ; Csts : list constraint }.
```

**Fig. 1.** Coq formalisation of constraint network

captured by lists without replicates and saved in a table (of type *mapdomain*) indexed by the variables[1].

The Coq record *network_inv csp* that captures well-formedness properties of a constraint network *csp*, is given in Fig2. The first proj. *Dwf* specifies that

[1] The Coq module *Fmap* is used to keep these tables, and the AVL implementation from the Coq's standard library is used in the extracted code.

the network variables (and only those), have an associated domain in the table embedded in *csp*. The second proj. $Cwf1$ specifies that the variables of a constraint are indeed variables of the network *csp*. The third proj. $Cwf2$ specifies that each variable appears at least once in the network. Finally, *norm* specifies that two constraints sharing the same variables must be identical.

```
Record network_inv csp : Prop := Make_csp_inv {
    Dwf : ∀ x, In x (Doms csp) ↔ In x (CVars csp) ;
    Cwf1 : ∀ (c:constraint) (x1 x2 : variable),
                c ∈ (Csts csp) → get_vars c = (x1, x2) →
                  x1 ∈ (CVars csp) ∧ x2 ∈ (CVars csp) ;
    Cwf2 : ∀ x, x ∈ (CVars csp) → ∃ c,
                c ∈ (Csts csp) ∧ (fst (get_vars c) = x ∨ snd (get_vars c) = x);
    Norm : ∀ c c', c ∈ (Csts csp) → c' ∈ (Csts csp) →
                get_vars c = get_vars c' → c = c' }.
```

**Fig. 2.** Well-formedness properties of a constraint network in Coq

## 2.1 Assignment - Solution

Following the definitions given in [6], an *assignment* is a partial map of some variables of the constraint network to values[2], a *valid assignment* is an assignment of some variables to a value from their domain, a *locally consistent assignment* is a valid assignment of some variables that satisfy the constraints that hold over them (and only those), and finally a *solution* is a locally consistent assignment of all the variables of the constraint network. We formalized these notions but do not expose their Coq specification very close to the previous informal definitions.

An important lemma about solutions, named *no_sol* given below, is involved in the completeness proof of the CP(FD) solver. It establishes that as soon as a domain in the constraint network *csp* becomes empty, then *csp* is shown be $unsatisfiable$, i.e., it has no solution. The lemma states that, in this case, any assignment defined over the set of variables of *csp* cannot be a solution. It uses the $find$ function defined on tables such that $find\ x\ a$ returns the value $v$ associated to $x$ in the instantiation $a$ (encoded as $Some\ v$), fails otherwise ($None$ is returned).

```
Lemma no_sol : ∀ csp,
(∃ v, find v (Doms csp) = Some []) → ∀ a , ¬ (solution a csp).
```

## 2.2 Arc-Consistency

The main idea of constraint filtering algorithms such as those used in CP(FD) solvers is to repeatedly filter inconsistent values from the domains. Thus, they reduce the search space while maintaining solutions. Several local consistency properties have been proposed to characterize the pruned domains [14,6], but

[2] Implemented in Coq by using the *Fmap* module, as variable-indexed table.

we focus here on the former and widely used *arc-consistency* property. Roughly speaking, a binary constraint $c(x, y)$ is arc-consistent w.r.t $(X, D, C)$ iff for any value $u$ in the domain of $x$ (i.e., $u \in D(x)$), there exists a value $v$ in the domain of $y$ such as $c(u, v)$ is consistent, and conversely for any value $v \in D(y)$, there exists a value $u \in D(x)$ such as $c(u, v)$ is consistent. A constraint network $(X, D, C)$ is arc-consistent iff any of its constraints $c$ in $C$ is arc-consistent. It is worth noticing that a constraint network can be arc-consistent, while it has no solution [6]. If a constraint network is arc-consistent and all of its domains are singletons, then it has a single solution.

In the original presentation of arc-consistency, a constraint network is represented with an undirected graph where nodes are associated to the variables, and edges are used to capture the constraints [19]. An edge between node $x$ and node $y$ exists iff there is a constraint $c$ containing variables $x$ and $y$ ($c(x, y)$) in the constraint network. However, by considering that constraints are undirected relations, this representation is implicitly ambiguous as it does not distinguish constraint $c(x, y)$ from $c(y, x)$. In our Coq formalisation, we tackled this problem by considering an order over the variables, and specified arc-consistency by distinguishing two arcs, denoted $(x, c, y)$ and $(y, c, x)$. Reconsidering the definition of arc-consistency given above, we say that $(x, c, y)$ is arc-consistent if for each value $v$ of the domain of $x$, there exists a value $t$ in the domain of $y$, such that $c$ is satisfied. The value $t$ is usually called the *support* of $v$ for $c$. Note that nothing is required regarding to the values from the domain of $y$. Our Coq formalisation is given in Fig.3, where $d$ is the table of domains and *compat_var_const* is the predicate that associates a constraint and its variables.

```
Definition arc_consistent x y c d :=
compat_var_const x y c →
∀ dx dy, find x d = Some dx → find y d = Some dy →
∀ v, v ∈ dx → ∃ t, t ∈ dy ∧ consistent_value c x v y t.
```

**Fig. 3.** Our Coq formalisation of arc-consistency

## 3 Formalisation and Verification of a Filtering Algorithm

In a CP(FD) solver, local consistency property, such as arc-consistency, is repeatedly applied over each constraint in a fixpoint computation algorithm, i.e., a filtering algorithm. Several distinct filtering algorithms exist, but the most well-known is AC3 [19,6]. At the heart of AC3 is a function that prunes the domain of a variable according to a constraint, commonly named REVISE.

Unlike existing pseudo-code presentations of AC3, we introduce in this section a Coq functional programming code of both algorithms REVISE and AC3.

### 3.1 Formalisation and Verification of Algo. REVISE

In our Coq formalisation shown in Fig.4, the function *revise* takes as arguments $c$, $x$, $y$, $dx$ and $dy$ where $x$ and $y$ are the variables of constraint $c$, with resp. domain *dx* and *dy*. Function *revise* returns a new domain $d'$ for $x$ and a boolean *bool_rev*. If *dx* has been revised, i.e., *dx* has been pruned to $d'$ where $d'$ is strictly included in *dx*, then *bool_rev* is true. Otherwise, *bool_rev* is false. A lot of

```
Fixpoint revise c x y dx dy {struct dx} :=
    match dx with
      nil ⇒ (false, dx)
    | v::r ⇒ let (b, d) := revise c x y r dy in
            if List.existsb (fun t ⇒ consistent_value c x v y t) dy
            then (b, v::d)
            else (true, d)
    end.
```

**Fig. 4.** Coq formalisation of Algo REVISE (function *revise*)

theorems about *revise* are required in the following, but we present only a selection of them in Fig.5. Many of these properties are demonstrated with the help of a functional induction on *revise* which is a tailored induction schema that follows carefully the different paths of the function. Part a. contains theorems showing the conformity of the functional text with respect to the informal specification. Part b. presents 2 theorems: the first one establishes that once a domain *dx* has been revised to $d'$, then its associated arc $(x, c, y)$ is arc-consistent with $d'$. And the second one: when *dx* is not revised, it means that arc $(x, c, y)$ was already locally consistent. Part c. contains a formal explanation of puzzling elements of $AC3$, it is concerned with the relationship between arc $(x, c, y)$ and $(y, c, x)$ w.r.t. arc-consistency. Roughly speaking, it means the modification of the domain of $x$ does not affect the arc-consistency of $y$. Finally, theorems in part d. state that *revise* preserves the solutions of a *csp*, and, equally important, does not add extra solutions.

## 3.2 Formalisation of Algorithm AC3

The main idea behind AC3 consists to revise the domains of all the variables in order to make all arcs arc-consistent. When this is done for arc $(x, c, y)$, we remove only values from the domain of $x$. Hence, other arcs whose target is also $x$ may not be consistent anymore, and they have to be revisited. AC3 maintains a queue containing all the arcs to be visited or revisited. When the queue is empty, AC3 has reached a fixpoint which is a state on which no more pruning is possible. During this fixpoint computation, if a domain becomes empty, then the constraint system is shown to be unsatisfiable.

The corresponding *AC3* function shown in Fig.6 takes as arguments the set of constraints of the network and a pair, composed of an initial map of variables to domains and a queue, containing arcs to be made arc-consistent. It results either in the pruned domains (of type *option mapdomain*), or *None* if the network has no solution or if the network is not well-formed. To add arcs in the queue, we use the function $\oplus$ that appends two lists without repetition.

**a. Conformity of** *revise*

```
Lemma revise_true_sublist : ∀ c x y dx dy newdx,
  compat_var_const x y c →
  revise c x y dx dy = (true, newdx) →
    newdx ⊂ dx.

Lemma revise_false_eq : ∀ c x y dx dy newdx,
  revise c x y dx dy = (false, newdx) → newdx = dx.
```

**b.** *revise* **and arc-consistency**

```
Lemma revise_arc_consistent : ∀ csp c x y ,
  c ∈ (Csts csp) → compat_var_const x y c →
  ∀ dx dy dx' b,
    find x (Doms csp) = Some dx → find y (Doms csp) = Some dy →
    revise c x y dx dy = (b, dx') →
      arc_consistent x y c (add x dx' (Doms csp)).

Lemma revise_false_consistent : ∀ csp c x y dx dy,
  c ∈ (Csts csp) → compat_var_const x y c →
  find x (Doms csp) = Some dx → find y (Doms csp) = Some dy →
   ∀ newdx, revise c x y dx dy = (false, newdx) →
      arc_consistent x y c (Doms csp).
```

**c. Relations on arcs** $(x, y, c)$ **and** $(z, x, c)$ **w.r.t. arc-consistency**

```
Lemma revise_x_y_consistent_y_x : ∀ csp c x y dx dy ,
  c ∈ (Csts csp) → compat_var_const x y c →
  find x (Doms csp) = Some dx → find y (Doms csp) = Some dy →
    ∀ newdx, revise c x y dx dy = (true, newdx) →
      arc_consistent y x c (Doms csp) →
        arc_consistent y x c (add x newdx (Doms csp)).

Lemma revise_x_y_consistent_x_z : ∀ d x y dx dy c newdx,
  compat_var_const x y c →
  find x d = Some dx → find y d = Some dy →
  revise c x y dx dy = (true, newdx) →
  ∀ z c0, compat_var_const x z c0 →
    arc_consistent x z c0 d →
      arc_consistent x z c0 (add x newdx d).
```

**d. Completeness of** *revise*

```
Theorem revise_complete : ∀ csp c x y dx dy (a : assign) ,
  network_inv csp →
  c ∈ (Csts csp) → compat_var_const x y c →
  find x (Doms csp) = Some dx → find y (Doms csp) = Some dy →
  solution a csp →
    ∀ newdx, revise c x y dx dy = (true, newdx) →
      solution a (set_domain x newdx csp).

Theorem revise_strict_solution : ∀ csp c x y dx dy ,
  network_inv csp →
  c ∈ (Csts csp) → compat_var_const x y c →
  find x (Doms csp) = Some dx → find y (Doms csp) = Some dy →
    ∀ a newdx, solution a (set_domain x newdx csp) →
      revise c x y dx dy = (true, newdx) →
        solution a csp.
```

**Fig. 5.** Properties of *revise*

```
Function AC3 (csts : list constraint)
      (d_q : mapdomain × list arc) {wf AC3_wf d_q} : option mapdomain :=
   let (doms, qu) := d_q in
   match qu with
   | nil ⇒ Some (doms)
   | (x, c, y)::r ⇒
     match find x doms, find y doms with
     | Some dx, Some dy ⇒
       let (bool_red, newdx) := revise c x y dx dy in
         if bool_red then
           if is_empty newdx
           then None
           else AC3 csts (add x newdx doms, r ⊕ (to_be_revised x y csts))
         else AC3 csts (doms, r)
     | _, _ ⇒ None
   end   end.

Definition measure_map (d: mapdomain) :=
fold (fun x ⇒ fun l ⇒ fun sum ⇒ (length l) + sum) d 0.
```

**Fig. 6.** Formalisation of $AC3$

In this function, *to_be_revised x y c* computes the set of arcs $(z, c', x)$ where $z \neq y$, that is the arcs that may have become inconsistent. Arc $(y, c, x)$ and arcs having $x$ as a source are discarded because the domain of $x$ after revision is necessarily included in the previous domain of $x$, and arc-consistency asks for a support for $y$ for each value of the domain of $x$. In our settings, we proved the above assertions (as captured by theorems *revise_x_y_consistent_y_x* and *revise_x_y_consistent_x_z*) which are required to establish soundness of $AC3$.

$AC3$ is defined as a general recursive function with the Coq construction *Function* which allows us to write the function as in any functional programming language. The overhead includes the definition of a well-founded order and the proof of decrease in the arguments in the recursive calls. For that, a lexicographic ordering, *AC3_wf*, defined on pairs $(d,\ q)$ was built from two measures. The measure of a queue was introduced as its number of elements. For maps, the measure *measure_map* was introduced as the sum of the lengths of the domains, as shown in Fig.6.[3] The proof of the decrease of the arguments required, in the case of the first recursive call, tedious manipulations of maps and lists and application of the lemma *revise_true_sublist* (see Fig.5, part a).

### 3.3 Correction of AC3

**Soundness.** The main soundness theorem, *AC3_sound*, shown in Fig.7 states that $AC3$ reduces the domains in order to achieve arc-consistency for each constraint at the end of the computation. In our formalisation, *complete_graph* computes the graph associated with the constraints, as a list of arcs. Soundness is

[3] Implemented with the map iterator *fold* defined in the module *Fmap*.

proved using a functional induction on $AC3$ and the invariant $PNC$ (for Potentially Non arc-Consistent), given in Fig.7. If $l$ is a list of arcs from the constraint network *csp*, *PNC csp d l* holds iff each arc ($x$, $c$, $y$) not arc_consistent w.r.t. the table of domains $d$ are in $l$. The main idea is to verify that at each step of the computation all the arcs that may be non arc-consistent are in the queue. The corresponding lemmas are given in Fig.7. Difficulties in these proofs have been to discover the invariant and the properties on which correctness relies. As often when formalizing existing algorithms, implicit hypotheses are to be made explicit and it can be hard work.

**a. Soundness theorem**

```
Theorem AC3_sound : ∀ csp d',
  network_inv csp →
  AC3 (Csts csp) (Doms csp, complete_graph (Csts csp)) = Some d' →
  ∀ x y c, (x, c, y) ∈ (complete_graph (Csts csp)) →
    arc_consistent x y c d'.
```

**b. Invariant**

```
Definition PNC csts (d : mapdomain) (l : list arc): Prop := ∀ x y c,
  (x, c, y) ∈ (complete_graph csts) → ¬(arc_consistent x y c d) →
    (x, c, y) ∈ l.

Lemma PNC_invariant_to_be_revised : ∀ csp c x y dx dy r newdx,
network_inv csp → (x, c, y) ∈ (complete_graph (Csts csp)) →
find x (Doms csp) = Some dx → find y (Doms csp) = Some dy →
revise c x y dx dy = (true, newdx) →
PNC (Csts csp) (Doms csp) ((x, c, y)::r) →
  PNC (Csts (set_domain x newdx csp)) (add x newdx (Doms csp))
        r ⊕ (to_be_revised x y (Csts csp))).

Lemma PNC_invariant_tail : ∀ csp d x y c r,
  (x, c, y) ∈ (complete_graph (Csts csp)) → arc_consistent x y c d →
  PNC (Csts csp) d ((x, c, y)::r) →
    PNC (Csts csp) d r.
```

**c. Completeness**

```
Theorem AC3_complete : ∀ csp (a : assign) d',
  network_inv csp → solution a csp →
  AC3 (Csts csp) (Doms csp, (complete_graph (Csts csp))) = Some (d') →
    solution a (set_domains d' csp).
```

**Fig. 7.** Correction of AC3

**Completeness.** Completeness means that *AC3* preserves the set of solutions. If $a$ is a solution of a constraint network, then filtering with $AC3$ will preserve it. In the formal settings of Fig.7 part c, the constraint networks before and after filtering only differ by their map of domains. In addition, a more general theorem where an arbitrary queue $q$ is introduced under the hypothesis that it is included in (*complete_graph* (*Csts csp*)), has been proved by functional induction on *AC3*, relying mainly on *revise* completeness. We have also proved that $AC3$ does not add any supplementary solution. The statement and proof of this theorem relies on the analogous property for *revise*. Those theorems are not given here to save space in the paper, but they are all available on our webpage.

## 3.4 An Optimization: AC2001

AC2001 is an improvement of AC3 published in [10] which achieves optimal time complexity of arc-consistency. When a support has to be found for a value, AC3 starts to search in the entire domain for the support, without remembering what happened in a previous step of filtering. AC2001 improves searching of a support by maintaining a structure named `last` which records, for each arc $(x, c, y)$ and value $v \in D(x)$, the smallest value $t \in D(y)$ such that $c(v, t)$ holds. So AC2001 requires a set of ordered values, while it is not the case for AC3. Hence, each time an arc is enforced to arc-consistency, the algorithm checks for each value $v$ in $D(x)$ whether the value $t$ recorded in `last` still belongs to $D(y)$. When it is not the case, AC2001 looks for a new value by enumerating all values in $D(y)$ greater than $t$. AC2001 shares all the AC3 formalisation items, but the *revise* function. Let us call this function *revise2001* for AC2001. It takes an extra argument, the `last` structure which we call a memory (of type *memory*) in our formalisation. The function returns also the new memory state, since it is modified when a revision takes place. A memory $m$ is represented by a table from *variable* $*$ *variable* to *list* (*value* $*$ *value*): if the variables of the constraint $c$ are $x$ and $y$ (in this order), then for $(v, t) \in m(x, y)$, $t$ is the smallest support found for $v$. It means that $c$ is satisfied for these values $(c(v, t))$ and that $c$ is not satisfied when assigning $v$ to $x$ and a value $w < t$ to $y$. Furthermore, for efficiency reasons, we require the list $m(x, y)$ to be ordered (on the first components of the pairs). All these properties are recorded in an invariant we call *memory_inv*. The core of $revise2001$ is the function formalized in Fig. 8 which acts directly on the list $last$ defined as $m(x, y)$. Cases (1) and (2) are very similar but differ wrt the existence or not of a support for the value $vx$ in $last$. The former happens when it is the first time the revision is done (*revise_exists c x y vx dy* tries to find a support for $vx$ in the entire domain $dy$), the latter is the nominal case (*revise2001_a_value c x y vx vy dy* tries to find a support for $vx$ in $dy$, starting from the old recorded support $vy$). Initial memory is the memory where each pair of variables is assigned the empty list. The final function $revise2001$ is just a wrapping embedding a memory $m$. All theorems that we proved for $revise$ can be established for $revise2001$, in particular soundness and completeness. Of course, they are modified w.r.t. the input and output memories. We also demonstrated that $revise2001$ preserves the memory invariant.

The Coq model corresponding to Sec. 2 and Sec. 3 contains $\approx$ 6000 lines of code. A functor has been implemented in order to factorize the formalisation of AC3 and AC2001 allowing us to share around 1500 lines.

```
Fixpoint revise2001_aux (c : constraint) (x y : variable) dx dy
                        (last_elem : list (value × value)) {struct dx}
                        : (bool × (list value × list (value × value))) :=
  match dx with
    nil ⇒ (false, (dx, nil))
  | vx::dx ⇒ let (o_vy, last_elem) := last_elem_get vx last_elem in
    let (bool_red, (dx, last)) := revise2001_aux c x y dx dy last_elem in
    match o_vy with
(1)      | None ⇒      match revise_exists c x y vx dy with
                       | None ⇒ (true, (dx, last))
                       | Some vy ⇒ (bool_red, (vx::dx, (vx, vy)::last))
                       end
(2)      | Some vy ⇒  match revise2001_a_value c x y vx vy dy with
                       | None ⇒ (true, dx_last)
                       | Some vy ⇒ (bool_red, (vx::dx, (vx, vy) :: last))
                       end
    end   end.
```

**Fig. 8.** Coq definition of AC2001

## 4 Labeling Search

Labeling implements a systematic search based on backtracking interleaved with local-consistency domain filtering. *Backtracking* incrementally attempts to extend an assignment toward a complete solution, by repeatedly choosing a value for an uninstantiated variable. Thus, the algorithm chooses a not yet assigned variable $x$ and a value $v$ in its domain following a given search heuristics, enforces the unary constraint $x = v$ (by assigning the domain to this unique value), re-establishes local consistency by applying the filtering algorithm (e.g., $AC3$) on each constraint. At this stage if filtering fails, it means there is no solution with $v$ as a value for the variable $x$, then backtrack to another value for $x$ or another variable, if possible. If filtering succeeds then go on with another variable if any. The labeling search procedure is *complete* if it can explore the overall search space. In our Coq formalisation, we implemented a complete search procedure with a simple heuristics, taking the first non assigned variable, with the first value met in the domain. Furthermore the labeling search procedure is independant from the filtering algorithm (e.g., $AC3$ or $AC2001$). In our settings, labeling is formalized in a module parameterized by the filtering algorithm and its required properties. Thus proofs about labeling are done only once, whatever be the filtering algorithm. Quantitatively, it means 1800 shared Coq lines vs 30 lines per instance. We adopted a style mixing computation and proof with the help of dependent types and the *Program Definition* construct, that eases a lot that style. The labeling function takes a well-formed constraint network and returns the first found solution if any, $None$ otherwise. It uses an auxiliary function that takes as argument a list of constraints *csts*, a list of variables to be assigned *vars*, the map $d$ of non empty domains for those variables satisfying arc-consistency. The type of the result is written as follows: {*ret*: *option domain*

| *result_ok ret csts vars d*}. The result is either *None* or *Some d'* and it must verify the expected soundness property, that is: if *None*, the csp (*vars*, *d*, *csts*) has no solution, if *Some d'*, then *d'* can be turned in a solution of (*vars*, *d*, *csts*): $d'$ assigns a unique value to the variables of *vars*, and all the constraints are arc-consistent w.r.t. *d'*. *Program Definition* generates 28 proof obligations, e.g., each recursive call requires to receive arguments verifying the embedded pre-conditions. In particular, the proof relies on properties about $AC3$ such as its soundness, and the fact that it reduces the domains. Some of these proof obligations concern the termination of the function, it relies again on a measure on maps of domains. We do not expose the code of the labeling function, as it just follows the informal description given above and encodes chronological backtracking in a recursive functional manner. All can be found on the webpage.

## 5 Evaluation

### 5.1 Extracting the Certified CP(FD) Solver

The extraction mechanism of Coq allows one to transform Coq proofs and functions into functional programs, erasing logical contents from them. Currently, we extract executable OCaml code only from Coq functions, as our proofs have no computational content. In a function defined with *Function*, such as $AC3$, or with *Program Definition*, such as labeling, proofs attached to proof obligations are just erased. So we can extract an operational certified CP(FD) solver for any language of binary constraints and arbitrary values. By *certified*, we mean a CP(FD) solver that returns provably-correct results in both cases, satisfiable and unsatisfiable formulas.

However the user still has to provide the constraint language, including (i) the OCaml type for the variables and the associated equality and ordering, (ii) the OCaml type for the variables and also the associated equality and ordering if AC2001 is used, and (iii) the OCaml type of constraints with the OCaml implementation of the *get_vars* and *interp* functions. It is worth noticing that the user still has to ensure the conformity of the *interp* function with the expected behaviour of the constraints. For our experiments, we introduce a language of binary constraints including operators <, =, >, the $\neq$ (e.g. $x > z$), conjunctions ($x > y \wedge x \; mod \; y = 0$) and disjunctions (e.g., $x \; mod \; y = 0 \vee x \; mod \; y = 2$) and the add/mult/sub/mod operators over 2 variables and a constant (e.g, $x = y{+}3$). We also implemented in OCaml a function translating addition constraints between 3 variables into binary constraints. Again the correctness of this preprocessing step, or more ambitious decomposition approaches, is not ensured by our formalisation and could be an extension. However, it seems that constraint decomposition requires source-to-source semantics preserving proofs which are less challenging than proving the correctness of filtering algorithms.

### 5.2 Experimental Results

The goal of our experiment was to evaluate the capabilities of the automatically extracted CP(FD) solver to solve classical benchmark programs of the

Constraints community. Of course, we did not expect our solver to compete with optimized (but unsafe) existing CP(FD) solvers, but we wanted to check whether our approach was feasible or not. We selected five well-known problems that may have interesting unsatisfiable instances, as we believe that a certified CP(FD) solver is much more interesting in this case. We selected a small puzzle (`sport`) (find a place to go to do sport with friends), the generic SEND+MORE=MONEY (`smm`) puzzle problem, the SUDOKU (`sudoku`) problem, the pigeon-hole problem (`pigeon`) and the Golomb rulers (`golomb`) problem. All problems but `sport` rely on a symbolic language of constraints whereas `sport` is defined via relations and tuples. Unlike the first four, the last one is a constraint optimization problem. Certifying unsatisfiability in this case is interesting for demonstrating that a given value for a cost objective function is actually a minimum value, i.e., any smaller value leads to the unsatisfiability of the problem. The Golomb rulers problem has various applications in fields such as Radio communications or X-Ray crystallography. A Golomb ruler is a set $x_1, .., x_m$ of $m$ ordered marks such as all the distances $\{x_j - x_i | \ 1 \leq i < j \leq m\}$ between two marks are distinct. The goal of Golomb rulers problem is to find a ruler of order $m$ with minimal length (*minimize* $x_m$). For example, $[0, 2, 5, 6]$ is an optimal Golomb ruler with 4 marks. All our experiments[4] have been performed on a standard *3.06Ghz clocked Intel Core 2 Duo* with *4Gb 1067 MHz DDR3 SDRAM* and are reported in Tab.1. The results show that extracting a reasonably-efficient certified CP(FD) solver is feasible. The solver is powerful enough to handle some classical problems of the CP Community, and useful to certify unsatisfiability. For instance, certifying that there is no Golomb ruler with 6 marks of length less than 17 takes about 23 sec (i.e., the Golomb ruler found by our solver is $[0, 2, 7, 13, 16, 17]$).

**Table 1.** CPU Time required with our certified AC3-based CP(FD) solver

| Examples | `sport` | `smm` | `sudoku` | `p(6)` | `p(7)` | `p(8)` | `p(9)` | `p(10)` | `g(4)` | `g(5)` | `g(6)` |
|---|---|---|---|---|---|---|---|---|---|---|---|
| in ms | 0,02 | 117 | 253 | 6 | 17 | 158 | 1611 | 17541 | 7 | 350 | 23646 |

## 5.3 Related Work and Discussion

A first concretization of automated certifying processes lates back to the middle of the nineties with the work of Necula on proof-carrying code [21]. The idea was to join correctness proof evidence to mobile code in order to offer the receiver some guarantee over the code. Since then, several large initiatives have been undertaken to build certifying compilers [22] or certified compilers [17] But, it is only recently that the needs for certifying/certified constraint solvers, i.e. SMT solvers, emerged from formal verification [12,20]. For certifying a SMT solver, one can think of two distinct approaches. A first approach is to make the

4 We have no specialized implementation for the global constraint ALLDIFF. It is translated into a list of binary difference constraints.

constraint solver produce an external trace of its computations in addition of its *sat*/*unsat* result. This external trace, sometimes called a *certificate* [12], can then be formally verified by a proof checker. Examples of such an approach include HOL Light used to certify results of the CVC Lite solver [20], or Isabelle/HOL to certify results of Harvey [12] and Z3 [8]. More recently, proof witnesses based on Type Theory in the proof assistant Coq have also been used to certify the results of decision procedures in SMT solvers [1,7]. A second approach, which is the one we picked up in our work even if it is considered harder than the first one, is 1) to develop the solver within the proof assistant, 2) to formally prove its correctness and 3) to extract automatically its code. For SAT/SMT solver, [18] is the only work we are aware of following this research direction. In this work, a Coq formalisation of an algorithm deciding the satisfiability of SAT formulas is proposed, and a fully reflexive tactic is automatically extracted to solve these formulas. According to our knowledge, the approach reported in this paper is the first attempt to certify a CP(FD) constraint solver. CP(FD) solving is currently outside the scope of arithmetic decision procedures, and SMT solvers rely on the BitVectors Theory to handle finitely-encoded integers [5]. We selected the second approach discussed above, to build our certified CP(FD) solver, because unsatisfiability in these solvers is not reported with certificates or proof trees. It means that using an external certified checker is not possible in that case. As a drawback, our approach cannot currently be used to certify directly the most advanced CP(FD) solvers such as Gecode or Zinc[5] that are used in industrial applications of CP. But, although our certified CP(FD) is not competitive with these hand-crafted solvers, it could be integrated as a back-end to certify a posteriori the unsatisfiable constraint systems detected by these solvers.

## 6 Conclusion

This paper describes a formally certified constraint solver over finite domains, i.e. CP(FD) with Coq[6]. Our formal model contains around 8500 lines ($\approx$ 110 definitions and 200 lemmas). The OCaml code of the solver has been automatically extracted. The solver implements either AC3 or AC2001 as a filtering algorithm, and can be used with any constraint language, provided that a constraint interpretation is given. According to our knowledge, this is the first time a CP(FD) solver can be used to formally certify the absence of solution, or guarantee that an assignment is actually a solution. Our main short-term future work involves the application of this certified solver to software verification. For that, we envision to integrate the solver within FocalTest our formally certified test case generator [9]. We will also parametrize the solver with another local consistency property, called bound-consistency [14], mainly used because it handles efficiently large sized finite domains. Other longer-term perspectives include the usage of our certified solver for solving constraint systems extracted from business or critical constraint models, e.g., in e-Commerce [15] or Air-Traffic Control.

[5] `http://www.gecode.org/` and `http://g12.research.nicta.com.au/`

[6] Available at `http://www.ensiie.fr/~dubois/CoqsolverFD`

**Thanks.** We are very grateful to Benoit Robillard who helped us proving formally the termination of the *AC3* function. We also thank the anonymous referees.

## References

1. Armand, M., Faure, G., Grégoire, B., Keller, C., Théry, L., Werner, B.: A modular integration of sat/smt solvers to coq through proof witnesses. In: Jouannaud, Shao (eds.) [16], pp. 135–150.
2. Bacchus, F., Chen, X., Beek, P., Walsh, T.: Binary vs. non-binary constraints. Artificial Intelligence 140(1-2), 1–37 (2002)
3. Bardin, S., Gotlieb, A.: FDCC: A Combined Approach for Solving Constraints over Finite Domains and Arrays. In: Beldiceanu, N., Jussien, N., Pinson, É. (eds.) CPAIOR 2012. LNCS, vol. 7298, pp. 17–33. Springer, Heidelberg (2012)
4. Bardin, S., Herrmann, P.: Osmose: Automatic structural testing of executables. Software Testing, Verification and Reliability (STVR) 21(1), 29–54 (2011)
5. Bardin, S., Herrmann, P., Perroud, F.: An Alternative to SAT-Based Approaches for Bit-Vectors. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 84–98. Springer, Heidelberg (2010)
6. Bessiere, C.: Constraint propagation. In: Handbook of Constraint Programming, ch. 3. Elsevier (2006)
7. Besson, F., Cornilleau, P.-E., Pichardie, D.: Modular smt proofs for fast reflexive checking inside coq. In: Jouannaud, Shao (eds.) [16]
8. Böhme, S., Fox, A., Sewell, T., Weber, T.: Reconstruction of z3's bit-vector proofs in hol4 and isabelle/hol. In: Shao, Jouannaud (eds.) [16]
9. Carlier, M., Dubois, C., Gotlieb, A.: A First Step in the Design of a Formally Verified Constraint-Based Testing Tool: FocalTest. In: Brucker, A.D., Julliand, J. (eds.) TAP 2012. LNCS, vol. 7305, pp. 35–50. Springer, Heidelberg (2012)
10. Bessiere, R.Y.C., Régin, J.-C., Zhang, Y.: An optimal coarse-grained arc consistency algorithm. Artificial Intelligence, pp. 165–185 (2005)
11. Collavizza, H., Rueher, M., Van Hentenryck, P.: Cpbpv: A constraint-programming framework for bounded program verification. Constraints Journal 15(2), 238–264 (2010)
12. Fontaine, P., Marion, J.-Y., Merz, S., Nieto, L.P., Tiu, A.F.: Expressiveness + Automation + Soundness: Towards Combining SMT Solvers and Interactive Proof Assistants. In: Hermanns, H. (ed.) TACAS 2006. LNCS, vol. 3920, pp. 167–181. Springer, Heidelberg (2006)
13. Godefroid, P., Klarlund, N.: Software Model Checking: Searching for Computations in the Abstract or the Concrete. In: Romijn, J.M.T., Smith, G.P., van de Pol, J. (eds.) IFM 2005. LNCS, vol. 3771, pp. 20–32. Springer, Heidelberg (2005)
14. Van Hentenryck, P., Saraswat, V., Deville, Y.: Design, implementation, and evaluation of the constraint language cc(fd). JLP 37, 139–164 (1998)
15. Holland, A., O'Sullivan, B.: Robust solutions for combinatorial auctions. In: Riedl, J., Kearns, M.J., Reiter, M.K. (eds.) ACM Conf. on Electronic Commerce (EC 2005), Vancouver, BC, Canada, pp. 183–192 (2005)
16. Jouannaud, J.-P., Shao, Z. (eds.): CPP 2011. LNCS, vol. 7086. Springer, Heidelberg (2011)
17. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM 52(7), 107–115 (2009)

18. Lescuyer, S., Conchon, S.: A Reflexive Formalization of a SAT Solver in Coq. In: Emerging Trends of the 21st Int. Conf. on Theorem Proving in Higher Order Logics, TPHOLs (2008)
19. Mackworth, A.: Consistency in networks of relations. Art. Intel. 8(1), 99–118 (1977)
20. McLaughlin, S., Barrett, C., Ge, Y.: Cooperating theorem provers: A case study combining hol-light and cvc lite. ENTCS, vol. 144(2) (January 2006)
21. Necula, G.C.: Proof-carrying code. In: POPL 1997, pp. 106–119 (1997)
22. Necula, G.C., Lee, P.: The design and implementation of a certifying compiler. In: PLDI 1998, pp. 333–344 (1998)
23. Rushby, J.: Verified software: Theories, tools, experiments. In: Automated Test Generation and Verified Software, pp. 161–172. Springer (2008)