Formally verified transformation of non-binary constraints into binary constraints

Catherine Dubois

Formally Verified Transformation of Non-binary Constraints into Binary Constraints

Catherine Dubois[0000−0002−9477−8109]

ENSIIE, Samovar, Évry-Courcouronnes, France
catherine.dubois@ensiie.fr

Abstract. It is well known in the Constraint Programming community that any non-binary constraint satisfaction problem (with finite domains) can be transformed into an equivalent binary one. One of the most well-known translations is the Hidden Variable Encoding. In this paper we formalize this encoding in the proof assistant Coq and prove that any solution of the binary constraint satisfaction problem makes it possible to build a solution of the original problem and vice-versa. This formal development is used to complete the formally verified constraint solver developed in Coq by Carlier, Dubois and Gotlieb in 2012, making it a tool able to solve any n-ary constraint satisfaction problem, The key of success of the connection between the translator and the Coq binary solver is the genericity of the latter.

1 Introduction

Constraint Programming (CP) or Constraint Satisfaction Problems [14] have many real-life applications such as decision making, resource allocation, scheduling, vehicle routing, configuration, planning, program verification, etc. In this paradigm, models are made of variables, domains which define the possible values of the variables and constraints which restrict the space of solutions. For example, modeling a Sudoku game requires 9*9 variables representing the different cells, their domain is the interval 1..9 and the constraints impose that the numbers in the cells must be all different in each column and each line, and that in each square we must find all the numbers from 1 to 9. Here constraints can be expressed using the specialized n-ary constraint AllDifferent [12]. Complex problems are usually naturally modeled with constraints involving a large number of variables. Historically, research in this area has focussed on binary constraints, i.e constraints using only two distincts variables. Then some transformations allowing to translate a non-binary problem containing constraints involving more than two variables, into an equivalent binary problem have been proposed, one of them is the Hidden Variable Encoding (HVE) [13], well-known in the Constraint Programming community. In this paper, we formalize this encoding in Coq and prove that it does provide an equivalent encoding, in the sense that any solution of the encoding binary problem can be translated into a solution of the

original non-binary problem and vice-versa. Furthermore if the original problem is unsatisfiable, then the encoding is also unsatisfiable and vice-versa.

This formal development related to HVE is used to extend the formally verified constraint binary solver developed in Coq by Carlier, Dubois and Gotlieb in 2012 [5], called CoqbinFD, making it a solver able to solve any problem. As far as we know, we provide here the first non-binary constraint solver (for finite domains) formally verified, extracted from a Coq development. It can serve as a reference solver for testing other constraint solvers. It can be compared to the verified LTL model checker developed in Isabelle/HOL proposed as a reference implementation in [8]. It is also a brick of a formal library dedicated to formalize results and classical algorithms about constraints, in the spirit of the project IsaFoL (Isabelle formalisation of Logic)[1] which includes e.g. the formalisation in Isabelle/HOL of a CDCL-based SAT solver using efficient imperative data structures [9].

In [6], we have presented such an encoding verified in Coq for ternary constraints only. This intermediate step was helpful to achieve the n-ary generalization. The two Coq formalisations are close and follow the same process. The main lemmas and theorems are if not identical, very close to each other. The reason why we have first done the ternary case is historical: the translation was implemented in OCaml to encode non-binary arithmetic constraints as a set of ternary constraints (it can always be done as long as only binary and unary operators occur in the non-binary constraint) in order to use CoqbinFD. Then we decided to push this transformation into Coq and to verify it for finally achieve the formalisation we present in this paper. The ternary version does not take into account extensional constraints whereas it is the case here.

The paper is organized as follows. Section2 briefly presents the notion of constraint satisfaction problem, the main ingredients of a constraint solver and the Hidden Variable Encoding. Section 3 describes the Coq formalisation of the Hidden Variable Encoding and highlights the proven properties. Then Section 4 introduces the main characteristics of CoqbinFD. Section 5 presents the extended solver, obtained by reusing CoqbinFD and also some experimentations. We conclude in the last section.

2 Background

A *Constraint Satisfaction Problem* (csp for short) or network of constraints [10] is a triple (X, D, C) where X is a set of variables, C is a set of constraints over X and D is a function that associates a domain $D(x)$ to each variable x in X. In our context, we exclusively consider finite domains. Constraints are relationships between variables, each taking a value in their respective domain: constraints restrict possible values that variables can take. As often in CP literature, we assume that constraints are normalized, meaning that two distinct constraints cannot hold over exactly the same variables. The arity of a constraint is the number of

[1] `https://bitbucket.org/isafol/isafol/wiki/Home`

its variables (assumed as distinct). A n-ary csp contains k-ary constraints with $k \leq n$. A csp is said non-binary as soon as it contains a constraint whose arity is strictly greater than 2. We do not consider unary constraints since the constraint can be directly taken into account in the domain. A solution is defined as a total assignment of the csp variables which satisfies all the constraints simultaneously.

Let us consider as an example the following non-binary csp (X, D, C) where $X = \{x_1, x_2, x_3, x_4, x_5, x_6\}$, $D(v) = \{0, 1\}$ for all v in X and $C = \{c_1 : x_1 + x_2 + x_6 = 1, c_2 : x_1 + x2 - x_3 + x_4 = 1, c_3 : x_4 + x_5 - x_6 \geq 1, c_4 : x_2 + x_5 - x_6 = 0, c_5 : x_1 \geq x_6\}$ inspired from [19]. It has a unique solution defined as $\{x_1 \mapsto 1, x_2 \mapsto 0, x_3 \mapsto 1, x_4 \mapsto 1, x_5 \mapsto 0, x_6 \mapsto 0\}$.

A constraint solver usually alternates propagation and exploration. Propagation prunes the domains of the variables, removing inconsistent values, using the constraints. This step can be decomposed in two interleaved routines: filtering that removes inconsistent values from the domains of the variables of one constraint and propagation that determines the constraints that have to be visited after a filtering step until a fixpoint is reached. Exploration enumerates values for some variables and may backtrack on these choices if necessary. The propagation step enforces a local consistency property that characterizes some necessary conditions on values to belong to solutions. There exist many different local consistencies, e.g. arc consistency, path consistency or bound consistency[3]. One of the oldest is arc consistency - AC for short - (when applied to binary constraints) or generalized arc consistency - GAC for short - (as a generalization of AC to n-ary constraints). Let c be a constraint of a csp (X, D, C) whose variables are $x_1,\ x_2\ \ldots x_k$. The constraint c is (generalized) arc consistent with respect to the csp if and only if for each variable x_i, for each value v in $D(x_i)$, there exist possible values for the other variables of the constraint c that make it true. Thus filtering c consists in removing the values of $x_1,\ x_2\ \ldots x_k$ that invalidate that property. In the previous example, c_1 is generalized arc consistent with respect to the given csp. However if we modify the domain of x_2 as the singleton $\{1\}$, c_1 is not anymore generalized arc consistent because when x_1 has the value 1, there is no value for x_2 that can make the constraint true. In such a case, a filtering algorithm would remove the value 1 from the domain of x_1.

Decomposition of non-binary constraints into equivalent binary constraints is a subject that has been widely discussed in the CP community and for quite a long time. A well-known transformation for constraint satisfaction problems with finite domains is the Hidden Variable Encoding (HVE) [13]. recognized as having nice theoretical properties [11]. In HVE, every non-binary constraint is associated with a variable whose domain is the set of all possible tuples of the original constraint, i.e. the set of tuples (of values of involved variables in the constraint) that satisfy the constraint. Such a variable is called a *dual variable* and written v_c if c denotes the constraint. Thus the variables of the equivalent binary csp are the variables of the original csp called *original variables* and the dual variables. The domains of the original variables remain identical to their domain in the original csp. Original non-binary constraints do not appear anymore in the binary encoding; they are replaced by *hidden constraints* between

a dual variable and each of the original variables in the constraint represented by the dual variable. A hidden constraint enforces the condition that a value of the original variable must be the same as the value assigned to it by the tuple that is the value of the dual variable [2]. In the following we denote them informally as projections: $proj_1, proj_2, \ldots$ A mathematical definition of this transformation (called the *hidden transformation*) can be found in [2] (see Definition 7).

As an illustration, the binary csp resulting from the HVE transformation applied on the example presented previously has 10 variables: the 6 original ones and 4 dual variables v_{c_1}, v_{c_2}, v_{c_3} and v_{c_4}. Domains of original variables remain identical whereas domains of the dual variables are such that
$D(v_{c_1}) = \{(0,0,1),(0,1,0),(1,0,0)\}$,
$D(v_{c_2}) = \{(0,0,0,1),(0,1,0,0),(0,1,1,1),(1,0,0,0),(1,0,1,1),(1,1,1,0)\}$,
$D(v_{c_3}) = \{(0,1,0),(1,0,0),(1,1,0),(1,1,1)\}$ and
$D(v_{c_4}) = \{(0,0,0),(0,1,1),(1,0,1)\}$.
There are 14 binary constraints: the original binary constraint c_5 and 13 hidden constraints, e.g. $proj_1(v_{c_2}, x_1)$, $proj_3(v_{c_4}, x_6)$.

3 Coq Formalisation of HVE Translation

3.1 N-ary Constraint Satisfaction Problem

The Coq formalisation follows the definition given previously, a csp is encoded as a record, of type *network_n* (see its definition in the code snippet below) containing a list of variables, a map from variables to domains (of type *domain_n*), represented as lists of values and a list of constraints. Types of variables (*variable_n*) and values (*value_n*) are abstract, they can be further defined either in Coq or in OCaml when extraction is used. We expect *value_n* and *variable_n* to be equipped with a strict total order and a decidable equality. Constraints (see below the definition of the type *constraint_n*), either binary or non-binary, are also abstract but the arity of a constraint is made explicit. It means that the type of basic constraints (*basic_constraint*) is abstract, equipped with a function to get the variables and an abstract interpretation function (as in CoqbinFD). A non-binary constraint is defined by a value of the abstract type *OP*, its arity and a list of variables. In order to be as general as possible, we consider extensional constraints as well as intentional ones. In the former case, the semantics is given as a list of acceptable tuples, in the latter case, a boolean function is expected. Constraint c_1 of the example given in Section 2 is represented as *Nary3 p1 [x1 ; x2 ; x6]* where *p1* is associated to the interpretation function $f(a,b,c) := a + b + c - 1 = 0$.

```
Inductive constraint_n : Set :=
| Bin : basic_constraint → constraint_n
| Nary : OP → nat → list variable_n → constraint_n.

Inductive interpretation : Set :=
| Extension : list (list value_n) → interpretation
| Intention : (list value_n → bool) → interpretation.
```

```
Record network_n : Type :=    Make_cspn {
    CVarsn : list variable_n ;
    Domsn : domain_n;
    Cstsn : list constraint_n
  }.
```

Our formalisation choice requires some extra properties about the input constraints language definition, in particular about the interpretation functions, for example *basic_interp* should only be defined for lists of length 2 (should fail for other cases) or the table defining an extensional k-ary constraint should only contain tuples with k components. These requirements can be checked at extraction time, e.g. by testing. They appear in our Coq development as axioms or parameters, in a weak form discovered during the proof of some properties.

```
Parameter interp_op_length_extension : ∀ op ar table,
interp_op op ar = Extension table → ∀ l, In l table → length l = ar.
```

The modeling of constraints is as simple as possible. It allows ill-formed constraints. The ability to deal with potentially ill-formed constraints makes the definition of some functions easier. We define well-formedness in a separate way as the predicate named *network_inv_n* which specifies the following requirements:

1. variables in constraints are exactly the ones that are listed in the csp and defined in the domain map;
2. constraints are normalized, meaning that they do not have the same set of variables;
3. any constraint has distinct variables;
4. in the case of a constraint of arity k represented by *Nary op k l*, the length of the list of variables l is exactly k, with k strictly greater than 2;
5. the tuples defining an extensional constraint must have components compatible with the domains of the related variables.

The property of well-formedness is implicitly introduced when needed.

An alternative way would have been to use dependent types for constraints, giving to the constructor *Nary* the following type *forall n, vector n → OP n → constraint_n* where *vector n* is the type of lists of length n and *OP n* the dependent version of the type *OP*. So a lot of types would become dependent. Another reason not to use dependent types is that we want to be able to easily define the constraints language in OCaml that does not provide such dependent types. A last reason is that the formalisation presented in this paper generalizes the proofs done for ternary constraints [6] and follows the same line.

3.2 Binary Constraint Satisfaction Problem

A binary csp (as it is encoded in CoqbinFD) has a very similar representation, it is a record containing a list of variables, a table that maps variables to finite domains and a list of binary constraints. We define in this subsection the variables, the values and the constraints of a binary csp resulting from the HVE

translation. The type of variables, *variable*, is defined inductively and reflects that variables are either original variables (introduced by the constructor *OVar*) or hidden variables (constructor *HVar*). The latter variables are defined w.r.t an original constraint. We make explicit this association in the way we build variables. For example, the hidden variable v_{c_1} of the example given in Section 2 is encoded in Coq as *HVar p1 3 [x1; x2; x6]*.

```
Inductive variable :=
| OVar : variable_n → variable
| HVar : OP → nat → list variable_n → variable.
```

The type of values, *value*, is also defined inductively, it distinguishes raw values, which are the original variables values, from tuples which are the hidden variables values.

```
Inductive value :=
| Raw_value : value_n → value
| Tuple : nat → tuple → value.
```

A decidable equality and a strict order are defined for both types, following from the required equalities and orders on *value_n* and *variable_n*.

We can now define the type *constraint* whose values are the original binary constraints and the hidden constraints. In our example, the hidden constraint between v_{c_1} and the second original variable is represented in Coq by *Proj p1 3 [x1; x2; x6] 1 x2*. We prove the properties on the constraint language required by CoqbinFD, e.g. any constraint has distinct variables.

```
Inductive constraint : Set :=
| Basic : basic_constraint → constraint
| Proj : OP → nat → list variable_n → nat → variable_n → constraint.
```

3.3 HVE transformation

The Coq function, *translate_csp_n*, that translates a non-binary csp into a binary csp, closely follows the presentation in Section 2 and the mathematical definition given in [2]. It uses several intermediate functions, in particular the function *expand* that computes the domain of a hidden variable, as a list of tuples, from the interpretation function and the domains of the ordinary variables of the constraint corresponding to the hidden variable. The computed domain contains only the tuples that satisfy the interpretation. In the case of an extensional non-binary constraint, the domain of the corresponding hidden variable is obtained by copying the table given as its interpretation. It also uses the function *cstsnTocsts2* which computes, for a list of constraints, the list of original binary and hidden constraints and the list of hidden variables coupled with their list of tuples computed with the help of *expand*. The ordinary binary constraints of the original csp and the corresponding domains are just copied modulo some elementary rewriting. The map containing the domains of the hidden variables is built with the help of the function *new_domain*.

Except some minor differences and the definition of the function *expand*, the function is similar to the one in the ternary case [6].

```
Definition translate_csp_n cspn :=
match (cstsnTocsts2 (Cstsn cspn) (Domsn cspn) ) with
| None ⇒ None
| Some (cs, lvdv) ⇒ Some (Make_csp
  (List.app (List.map (fun x ⇒ OVar x) (CVarsn cspn)) (List.map fst lvdv))
  (new_domain (mapn_to_raw (Domsn cspn) (CVarsn cspn)) lvdv)
  cs)
end.
```

Note that *translate_csp_n* may fail when *cstsnTocsts2* tries to access the domain of unknown variables. We prove that if the non-binary csp is well-formed then the translation does not fail:

```
Lemma network_inv_n_translate_None_False : ∀ cspn,
network_inv_n cspn → ¬ (translate_csp_n cspn = None).
```

We also prove that the binary csp obtained by HVE is well-formed if the original csp is well-formed:

```
Lemma translate_cspn_network_inv : ∀ cspn csp,
network_inv_n cspn → translate_cspn cspn = Some csp →
    network_inv csp.
```

3.4 Focus on tuples and extraction

Let us focus on the *expand* function that, in the case of an intentional constraint, computes the set of tuples. It is merely the computation of the cartesian product of k lists if the arity of the constraint is k. We first compute the result as a list of lists (of length k) representing the tuples. Then we turn these lists into tuples whose type is abstract with the help of an abstract function *tuple_from_list* introduced as a parameter. Yes, this step requires a computational overhead but it allows some flexibility at extraction time. For example we can map the type *tuple* to the OCaml *array* type in order to benefit from a constant time access. We can also keep lists by mapping *tuple_from_list* to the identity function.

Besides *tuple_from_list*, we need two other functions: *tuple_to_list* (of type *nat → tuple → list value_n*) and *proj_tuple* (of type *nat → tuple → value_n*). The first one is not used in the translation itself but only in the proofs. These functions are specified by three properties or axioms which are given below:

```
Parameter tuple_to_from_list : ∀ a ,
tuple_to_list (length a) (tuple_from_list a) = a.

Parameter proj_tuple_nth_error : ∀ n n0 t v0,
n > 0 → n0 < n →
proj_tuple n0 t = v0 ↔ nth_error (tuple_to_list n t) n0 = Some v0.

Parameter length_tuple_to_list : ∀ n t,
length (tuple_to_list n t) = n.
```

In order to gain some more confidence when we extract OCaml code from the Coq code, we have tested these three properties using the QuickChick property testing tool for Coq programs [7] with 10 000 test cases randomly generated. It allowed the discovery of a missing hypothesis (the blue one in the second statement).

An alternative could be to use primitive persistent arrays in the Coq code for implementing tuples (without going through intermediate lists). The type of such arrays is axiomatized. Primitive arrays internally are implemented using a persistent data structure. This has been integrated into a very recent version of Coq while it was previously available as a separate implementation [1]. We plan to experiment with these primitive arrays. However cartesian product of domains implemented in the *expand* function is a bit more complicated when dealing with arrays.

A last proposition could be to implement tuples as finite functions, and then to use the coq library proposed by Sakaguchi in [16] to extract these tuples to OCaml arrays.

3.5 Correctness of the HVE translation

To prove the correctness of the translation, we prove that satisfiability is preserved by the HVE translation. Two related properties are illustrated below.

A solution is defined as usual as an assignment of the csp variables which is total, valid (i.e. values are compatible with the domains) and locally consistent (i.e. making each constraint satisfied). It is implemented as a map from variables to values. A solution of a non-binary csp (resp. a binary encoding csp) is characterized by the predicate *solution_n* (resp. *solution*).

Lemma *translate_nosol* states that if the original non-binary csp is UNSAT (i.e. it admits no solution) then the binary encoding is also UNSAT. Lemma *translate_complete* explains that if the non-binary csp admits a solution, then its mapping to the hidden and original variables (computed by the function *translate_sol_n*) is a solution of the binary encoding.

```
Lemma translate_nosol: ∀ cspn csp ,
  network_inv_n cspn → translate_csp_n cspn = Some csp →
  (∀ a, ¬ (solution a csp)) → ∀ an, ¬ (solution_n an cspn).

Lemma translate_complete: ∀ an cspn csp,
  network_inv_n cspn → translate_csp_n cspn = Some csp →
  solution_n an cspn → solution (translate_sol_n an cspn) csp.
```

3.6 Local Consistencies

We have completed the formalisation by the proof of a result about local consistency: if the original csp is generalized arc consistent then its binary encoding is arc consistent. Unsurprisingly, the proof of this property reuses a large part of the script and intermediate lemmas developed for soundness and completeness. The interesting point worth noticing is that this proof requires the introduction

of the requirement 5 in *network_inv_n* specifying the proper formation of tuples defining an extensional constraint.

4 Brief Presentation of the Formally Verified Solver CoqbinFD

In this section we briefly describe the binary solver CoqbinFD that we want to reuse. For more details please consult [5]. An important point in this development and crucial for the present work is its genericity. In the following we mainly emphasize the requirements about the generic parameters. The solver is indeed parameterized by the type of variables (*variable*) and values (*value*) and also by the constraint language (*constraint*). In Coq, these types are abstract, assumed to accept a decidable equality. It is also assumed that the semantics of the constraints is given by an interpretation function as a Boolean function of the values of its two variables and a function that retrieves, for any constraint, its two variables. So a constraint is abstracted as a relation over two distinct variables, represented by an interpretation predicate. These types and functions must be defined either in Coq or OCaml in order to use the extracted solver in a particular context. Here they are given Coq concrete values according to HVE.

A csp is defined as a record of type *network_csp* consisting of a finite list of variables (*CVars* field), a finite list of constraints (*Csts*) and a map (*Doms*) associating each domain with its variable, here a finite list of values. A predicate (*network_inv*) specifies the well-formedness of a csp: the entries of the domain map are exactly the variables of the csp, variables appearing in the constraints are exactly those declared, constraints are normalized and finally the two variables of any constraint are distinct.

The solving process is based on arc consistency, it implements a generic version of the propagation algorithm AC3 [10], allowing the use of AC2001 [10]. However here, it is transparent, the binary solver being used as a black-box.

5 Extension of the Solver CoqbinFD to Non-binary Constraints

We propose to build a constraint solver able to deal with binary and non-binary constraints by extending the CoqbinFD solver (whose main function is the *solve_csp* function) with the HVE translation acting as a pre-processor and the solution translation acting as a post-processing. The different steps are illustrated on Fig. 1.

The extended solver is mainly embodied by the following *solve_n* function which follows the steps of Fig. 1 and is built using the tactic `Program` [17] (as its counterpart in CoqbinFD):

```
Program Definition solve_n (cspn : network_n) (Hn: network_inv_n cspn)
  : {res : option (assign_n ) | result_n_ok res csp} :=
match (translate_csp_n cspn) with
```

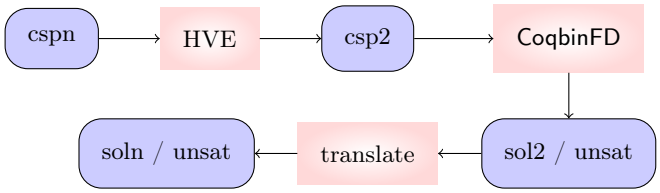

Fig. 1. Behavior of the Non-binary Solver

```
  None ⇒ None
| Some csp ⇒ match (solve_csp csp _) with
               None ⇒ None
             | Some a ⇒ Some (translate_sol a cspn.CVarsn)
             end
end.
```

The type of the result is a kind of subtype *à la PVS*, it describes not only the type of the computed result *res* (*None* or a solution) but it also contains a proof that the result is sound (specified by the predicate *result_n_ok*), i.e. if the result is *None* then the original csp has no solution and if it is *Some a*, then *a* is a solution of the original csp. This definition generates proof obligations that correspond to the expected properties of the result. Another proof obligation comes from the underscore appearing in the call of *solve_csp* that expects as a third argument a proof that its second argument is well-formed. This proof obligation is solved by the lemma *translate_cspn_network_inv* shown previously in Subsection 3.3.

Completeness of the extended solver is also proved. It follows from the completeness of CoqbinFD and from the properties of the *translate_sol_n* function regarding solutions.

The main task to extend CoqbinFD to n-ary constraints is to provide the binary encoding exactly as it is expected by CoqbinFD. As this solver is generic in the input constraint language, the task was made easier.

After extraction, we ran the extended solver (with the AC3 instance of CoqbinFD) to solve some problems. First we have used it with binary and ternary csps, for non-regression testing. The time overhead is not significant. We also solved some problems, intensional and extensional ones, from the XCSP2.1 library [15] where csps are represented as XML definitions. For example the problem named `normalized_g_4x4` with 16 variables with $\{0, 1\}$ as domain and 15 constraints with arity from 3 to 5 is solved in 0.0033 sec on a laptop (2,3 GHz Intel Core i5 8 Go 2133 MHz LPDDR3) whereas for the problem known as `normalized-graceful-K2-P3` with 15 variables (whose domain is either 0..9 or 1..9) and 60 constraints, 9 of them being ternary and the rest being binary, we obtain a solution within 2.8 sec.

The manual transcription of XCSP2.1 problems in OCaml is however tedious and error prone. Our solver could be completed with a tool allowing the trans-

lation of XML definitions into OCaml or Coq definitions. Following Stergiou and Samaras in [18], we could obtain a better efficiency by using specialized arc consistency and search algorithms for the binary encodings requiring to further prove some variants for propagation and exploration algorithms.

6 Conclusion

In this paper we have formalized in Coq the well-known Hidden Variable Encoding that performs the translation of a non-binary constraint satisfaction problem into an equivalent binary constraint satisfaction problem. This translation is used to extend the CoqbinFD solver, developed in Coq several years ago. The Coq code is available at `www.ensiie.fr/~dubois/HVE_nary`. From the whole Coq development, an OCaml executable solver can be extracted. It can be considered as a reference implementation and used to test other solvers, for example the FaCiLe OCaml constraint library [4].

References

1. M. Armand, B. Grégoire, A. Spiwack, and L. Théry. Extending coq with imperative features and its application to SAT verification. In M. Kaufmann and L. C. Paulson, editors, *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6172 of *Lecture Notes in Computer Science*, pages 83–98. Springer, 2010.
2. F. Bacchus, X. Chen, P. van Beek, and T. Walsh. Binary vs. non-binary constraints. *Artificial Intelligence*, 140(1):1 – 37, 2002.
3. C. Bessière. Constraint propagation. In *Handbook of Constraint Programming*, chapter 3. Elsevier, 2006.
4. P. Brisset and N. Barnier. FaCiLe : a Functional Constraint Library. In *CICLOPS 2001, Colloquium on Implementation of Constraint and LOgic Programming Systems*, Paphos, Cyprus, 2001.
5. 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.
6. C. Dubois. Formally verified decomposition of non-binary constraints into equivalent binary constraints. In N. Magaud and Z. Dargaye, editors, *Journées Francophones des Langages Applicatifs 2019*, Les Rousses, France, Jan. 2019.
7. M. Dénès, L. Lampropoulos, Z. Paraskevopoulou, and B. C. Pierce. Quickchick: Property-based testing for coq, 2014.
8. J. Esparza, P. Lammich, R. Neumann, T. Nipkow, A. Schimpf, and J. Smaus. A Fully Verified Executable LTL Model Checker. In N. Sharygina and H. Veith, editors, *Computer Aided Verification 2013, Saint Petersburg, Russia*, volume 8044 of *LNCS*, pages 463–478. Springer, 2013.
9. M. Fleury, J. C. Blanchette, and P. Lammich. A verified SAT solver with watched literals using imperative HOL. In J. Andronick and A. P. Felty, editors, *7th ACM SIGPLAN Int. Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA*, pages 158–171. ACM, 2018.

10. A. Mackworth. Consistency in Networks of Relations. *Art. Intel.*, 8(1):99–118, 1977.
11. N. Mamoulis and K. Stergiou. Solving non-binary Csps using the hidden Variable encoding. In T. Walsh, editor, *Principles and Practice of Constraint Programming - CP 2001, 7th Int. Conference, CP 2001, Paphos, Cyprus*, volume 2239 of *LNCS*, pages 168–182. Springer, 2001.
12. J.-C. Régin. A Filtering Algorithm for Constraints of Difference in CSPs. In *12th National Conference on Artificial Intelligence (AAAI'94)*, pages 362–367, 1994.
13. F. Rossi, C. J. Petrie, and V. Dhar. On the Equivalence of Constraint Satisfaction Problems. In *ECAI*, pages 550–556, 1990.
14. F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming*. Elsevier Science Inc., USA, 2006.
15. O. Roussel and Christophe Lecoutre. XML representation of Constraint networks: Format XCSP 2.1. *CoRR*, abs/0902.2362, 2009.
16. K. Sakaguchi. Program extraction for mutable arrays. *Sci. Comput. Program.*, 191:102372, 2020.
17. M. Sozeau. Subset coercions in coq. In T. Altenkirch and C. McBride, editors, *Types for Proofs and Programs, International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers*, volume 4502 of *Lecture Notes in Computer Science*, pages 237–252. Springer, 2006.
18. K. Stergiou and N. Samaras. Binary encodings of non-binary constraint satisfaction problems: Algorithms and experimental results. *J. Artif. Intell. Res.*, 24:641–684, 2005.
19. K. Stergiou and T. Walsh. Encodings of Non-Binary Constraint Satisfaction Problems. In J. Hendler and D. Subramanian, editors, *Sixteenth National Conference on Artificial Intelligence and Eleventh Conference on Innovative Applications of Artificial Intelligence, 1999, Orlando, Florida, USA.*, pages 163–168. AAAI Press / The MIT Press, 1999.