Formally Verified Constraints Solvers
A Guided Tour

Catherine Dubois
ENSIE & Samovar

catherine.dubois@ensiie.fr
https://www.ensiie.fr/~dubois

Introduction

Constraint solvers are complex tools implementing tricky algorithms and heuristics manipulating in-
tricate data structures. It is well-known that they have bugs. Certifying the output of such tools is
extremely important in particular when they are used for critical systems or in verification tools:
There are mainly two ways for having confidence in the computed results: making the solver produce not
only the output but also proof logs that can be easily verified by an external checker or proving the
correctness of the solver itself. The former approach is widespread in the Boolean satisfiability com-
munity through formats such as DRAT which can be considered as a standard. The latter approach
has been followed for example for developing Compcert and se4 respectively a C compiler developed
and formally verified with the help of the Coq proof assistant Coq and a micro-kernel developed and
formally verified with the proof assistant Isabelle/HOL.

Main Objectives
• Develop a family of formally verified constraints solver (for finite domains)
• Formalize and understand deeply algorithms and results used in CP

Constraint Satisfaction Problem (CSP)

Definition of a CSP
A CSP is a triple (X, D, C) where:
X: a set of variables,
D: a function that maps each variable of X to its domain here finite set of possible values,
C: a set of constraints (relations btw variables) over variables of X, arity of a constraint = number of its variables.
A solution of (X, D, C) is a valid (compatible with D) assignment defined for all the variables in X
that satisfies all the constraints in C. A CSP is unsatisfiable (UNSAT) when it has no solution.
Solving can mean finding one solution or all solutions, showing UNSAT, completing a solution, find-
ing that satisfies all the constraints in C

CoqbinFD: a formally verified constraints CP(FD) solver

CoqbinFD has been developed with the Coq interactive proof assistant, proved sound and complete
−→ the existing solvers!
−→ The algorithm computes alternating paths, augmenting paths, transfer functions etc.
The proof of its correctness relies on the Berge’s theorem (1957) that we are formalizing in Coq: in a graph is maximum iff there is no augmenting path for

The algorithm

-→ The maximun matching algorithm M is written in OCaml, it returns a matching and a vertex cover
(witness).
−→ The algorithm M is not verified in Coq, considered as untrusted.
The checker verifies that the matching and the vertex cover are 2 sets with the same cardinality, and thus it is a maximal matching.
−→ The checker is developed in Coq, proved correct and extracted as an OCaml executable code.

Approach 2: Verification of a functional algorithm in Coq
The algorithm computes alternating paths, augmenting paths, transfer functions etc.
The proof of its correctness relies on the Berge’s theorem (1957) that we are formalizing in Coq:
−→ Introduction general interfaces, generic parameters, ... yes but there are optional/mandatory fea-
tures, implementation constraints, etc.
−→ Defines a Software Product Line of verified CP solvers

Reuse: For AIIDifferent, and many others global constraints, a large background about graph theory
(matchings, network flows, relaxation methods, . . . ) is needed but . . . there is no large Coq library
about graphs and operational research.
−→ Big effort is needed
−→ Large libraries are needed and must be shared between provers
−→ Interoperability of provers, encyclopedia of proofs

Domains as interval lists
In existing solvers, the main domain representations are range sequences (Geocode, ECLiPSe Prolog,
SICStus Prolog, FaCILE) implemented as lists or binary trees, gap intervals trees (Naxos Solver), bit
vectors (ILSG Solver, Chase), successor vectors, sparse sets (OscaR, Abscon, Castor).

Focus is put on domains represented as minimal sorted sequences of ranges.
Example: the domain [10, 10, 10, 10, 10, 11, 11] is represented by the list [[10, 10, 10, 10, 10], [11, 11]].
Our contribution:
−→ Formalisation in Coq of range sequences;
−→ Partial verification of FaCILE interface (an OCaml constraints library);
−→ Extension to provide a new implementation of finite sets (Coq FSet interface).
From this formal development, we are deriving a Coq formalisation of gap interval lists (work in
progress with A. Ledem).

AIIDifferent global constraint
AII Different(x1, . . . , xN) means that all variables x1, . . . , xN must be pairwise different.

It can be solved by decomposing it into a set of binary inequalities: ∪(i,j)(xi ̸= xj) (loss in filtering level)
or using a dedicated filtering/propagation algorithm (e.g. Regin 94).
Following Regin’s algorithm, a solution of AII Different(x1, . . . , xN) is a matching (of the value graph)
that covers V = {x1, . . . , xN}.

Enforcing GAC means removing all the edges that cannot be in any matching covering V .

Work in progress: formalisation in Coq of the first step of the Regin’s filtering algorithm, epicompu-
tation of a maximal matching in the graph of values

Approach 1: veriﬁcation a posteriori — development of a verified checker [3]
The maximal matching algorithm M is written in OCaml, it returns a matching and a vertex cover
(witness).
−→ The algorithm M is not verified in Coq, considered as untrusted.
The checker verifies that the matching and the vertex cover are 2 sets with the same cardinality, and thus it is a maximal matching.
−→ The checker is developed in Coq, proved correct and extracted as an OCaml executable code.

Approach 2: veriﬁcation of a functional algorithm in Coq
The algorithm computes alternating paths, augmenting paths, transfer functions etc.
The proof of its correctness relies on the Berge’s theorem (1957) that we are formalizing in Coq:
- a matching in a graph is maximum iff there is no augmenting path for m in G.

Conclusion and future work

We have developed and verified in Coq a binary constraints solver for finite domains and some vari-
ants of it. Some challenges remain.

Eﬃciency: Eﬃciency of the extracted solver CoqbinFD is reasonable but we cannot compete with
the existing solvers!
We expect it to serve as a reference implementation but some efforts are still needed.
−→ More eﬃcient data structures for variables, domains, queues, etc.
−→ Eﬃcient dedicated ﬁlters/propagators (but loss of generality?)
−→ Imperative data structures (reﬁnement, primitive arrays)

Modularity: CoqbinFD has a too rigid structure: it is diﬃcult to plug-in new domain representations,
new local consistencies,
−→ Introduce general interfaces, generic parameters, ... yes but there are optional/mandatory fea-
tures, implementation constraints, etc.
−→ Deﬁne a Software Product Line of veriﬁed CP solvers

Reuse: For AIIDifferent, and many others global constraints, a large background about graph theory
(matchings, network ﬂows, relaxation methods, . . . ) is needed but there is no large Coq library
about graphs and operational research.
−→ Big eﬀort is needed
−→ Large libraries are needed and must be shared between provers
−→ Interoperability of provers, encyclopedia of proofs

References

[2] Catherine Dubois. Formally verified decomposition of non-binary constraints into binary con-

Acknowledgements
This is joint work with A. Butant (ENSIEE 2016), M. Carlier, V. Clément (ENSIEE 2014), S. Eloumi
(ENSIA), A. Gotlieb (Simula, Norway), A. Ledem (ENSIEE 2020) and H. Mlodecki (ENSIEE 2017).
Many thanks to them!

From non-binary CSPs to binary CSPs

A non-binary CSP can be solved by translating/encoding it into an equivalent binary one and using
well-established binary CSP techniques or using extended versions of binary techniques directly on
the non-binary problem (usually the recent case, however still research interest for encodings).
One of the most popular encodings is the hidden variable encoding (HVE) (1999).

original CSP
{X, D, C}
⇒ HVE Encoding
{X′, D′, C′}
x ∈ X
⇒ Ordinary variable
D′(x) = {D(x)}
binary constraint
⇒ Propositional constraint
n-ary constraint
⇒ n-ary constraint
∀v(0) = (x1, . . . , xn) ⇒ D′(v) = D(x1) ∧ D(x2) ∧ . . . D(xn)
∀v(n) = (x1, . . . , xn, z) ∈ C′, i ∈ [1, n]

Our work
• Formalisation in Coq of the hidden variable encoding, proof of its correctness;
• Extension of CoqbinFD as a formally verified CPFD solver for both binary and non-binary con-
straints.