Boolean Constraint Solving Using `clp(FD)`

Philippe Codognet and **Daniel Diaz**
INRIA-Rocquencourt
Domaine de Voluceau BP 105
78153 Le Chesnay Cedex
FRANCE
{`Philippe.Codognet, Daniel.Diaz`}`@inria.fr`

Abstract

We present a boolean constraint logic language `clp(B/FD)` built upon a language over finite domains `clp(FD)` which uses a local propagation constraint solver. It is based on a single primitive constraint which allows the boolean solver to be encoded at a low-level. The boolean solver obtained in this way is both very simple and very efficient: on average it is eight times faster than the CHIP propagation-based boolean solver, i.e. nearly an order of magnitude faster, and infinitely better than the CHIP boolean unification solver. It also performs on average several times faster than special-purpose stand-alone boolean solvers.

1 Introduction

Constraint Logic Programming combines both the declarativity of Logic Programming and the ability to reason and compute with partial information (constraints) on specific domains, thus opening up a wide range of applications. Among the usual domains found in CLP, the most widely investigated are certainly finite domains, real/rationals with arithmetic constraints, and booleans. This is exemplified by the three main CLP languages: CHIP [22], which proposes finite domains, rational and booleans, PrologIII [5] which includes rationals, booleans and lists, and CLP($\mathcal{R}$) [10] which handles constraints over reals. Whereas most researchers agree on the basic algorithms used in the constraint solvers for reals/rationals (simplex and gaussian elimination) and finite domains (local propagation and consistency techniques), there are many different approaches proposed for boolean constraint solving. Some of these solvers provide pecial-purpose boolean solvers while others have been integrated inside a CLP framework. However, different algorithms have different performances, and it is hard to know if for some particular application any specific solver will be able to solve it in practise. Obviously, the well-known NP-completeness of the satisfiability of boolean formulas shows that we are tackling a difficult problem here.
Over recent years, local propagation methods, developed in the CHIP language for finite domain constraints [22], have gained a great success for many applications, including real-life industrial problems. They stem from consistency techniques introduced in AI for Constraint Satisfaction Problems (CSP) [11]. Such a scheme has also been used in CHIP to solve boolean constraints with some success, to such an extent that it has become the standard tool in the commercial version of CHIP. This method performs better than the original boolean unification algorithm for nearly all problems and is competitive with special-purpose boolean solvers. Thus, the basic idea is that an efficient boolean solver can be derived from a finite domain constraint solver for free.

It was therefore quite natural to investigate such a possiblity with our CLP system `clp(FD)`, which handles finite domain constraints similar to that of CHIP, but being nevertheless more than twice as fast on average [6]. `clp(FD)` is based on the so-called "glass-box" approach proposed by [23], as opposed to the "black-box" approach of the CHIP solver for instance. The basic idea is to have a *single constraint* `X in r`, where r is a *range* (e.g. $t_1..t_2$). More complex constraints such as linear equations and inequations are then defined in terms of this primitive constraint. The `X in r` constraint can be seen as embedding the *core* propagation mechanism for constraint solving over finite domains, and can be seen as an abstract machine for propagation-based constraint solving.
We can therefore directly encode a boolean solver at a low-level with this basic mechanism, and decompose boolean constraints such as *and*, *or*, and *not* in `X in r` expressions. In this way, we obtain a boolean solver which is obviously more efficient than the encoding of booleans with arithmetic constraints or with the less adequate primitives of CHIP. Worth noticing is that this boolean extension, called `clp(B/FD)`, is very simple; the overall solver (coding of boolean constraints in `X in r` expression) being about ten lines long, the glass-box is very clear indeed... Moreover, this solver is surprisingly very efficient, being eight times faster on average than the CHIP solver, which was however reckoned to be efficient, with peak speedup reaching twelve. `clp(B/FD)` is also more efficient than special purpose solvers. It is interesting to remark that, on traditional finite domain benchmarks, `clp(FD)` is twice as fast as CHIP, while on boolean examples `clp(B/FD)` is eight times faster. This can be explained by two main reasons. Firstly, booleans are, to the best of our knowledge, encoded at a finer granularity in `clp(B/FD)` than in CHIP, thanks to the `X in r` decomposition. Secondly, `clp(FD)` integrates some global optimizations on the handling of `X in r`, which benefit all non-basic constraints built upon it, including in this case boolean constraints. This is not the case in CHIP where, while the finite domains part (especially the treatment of linear constraints) certainly contains numerous ad hoc optimizations, the boolean part does not benefit from them. Remark in passing that this is one more argument for the glass-box approach versus the black-box approach. This architecture also has several other advantages, as follows. First, being integrated in a full CLP language, heuristics can be added in the program itself, as opposed to a closed boolean solver with (a finite set of) built-in heuristics. Second, being integrated in a finite domains solver, various extensions such as pseudo-booleans [2] or multi-valued logics [24] can be integrated straightforwardly. Third, being based on a propagation method, searching for a single solution can be done much more quickly if the computation of all solutions is not needed.

The rest of this paper is organized as follows. Section 2 introduces boolean constraints and the variety of methods proposed as boolean solvers. Section 3 then briefly presents the `clp(FD)` system and the `X in r` primitive constraint. Sections 4 and 5 detail the propagation-based solver. A rule-based semantics is first introduced, and its equivalence to the usual declarative semantics of boolean formulas is shown (section 4). The solver is then introduced and proved correct (section 5). Performances of `clp(B/FD)` are detailed in Section 6, comparing both with CHIP and several other efficient dedicated boolean solvers. A short conclusion and research perspectives end the paper.

2 Boolean constraints

We first present the basic notions of boolean expressions, and then review the different approaches currently used to solve such constraints in existing systems.

Definition
A *boolean formula* on a set $\mathcal{V}$ of variables is defined inductively as follows:

- the constants 0 and 1 are boolean formulas,
- X is a boolean formula, for $X \in \mathcal{V}$,
- $X \wedge Y$, $X \vee Y$, $\neg X$, $X \equiv Y$ are boolean formulas, for X and Y formulas.

The declarative semantics of the boolean connectives $(\wedge, \vee, \neg, \equiv)$ are given by the well-known truth-tables.
Although the problem of satisfiability of a set of boolean formulas is quite old, designing efficient methods is still an active area of research, and there has been a variety of methods proposed over recent years toward this aim. Moreover, it is usually important not only to test for satisfiability but also to actually compute the models (assigments of variables), if any. To do so, several types of methods have been developed, based on very different data-structures and algorithms. We have chosen to classify existing methods, focusing on implemented systems, by considering as principal criterion the main data-structure used to represent constraints, since it is of primary importance for performances.
Existing boolean solvers can thus be roughly classified as follows:

Resolution-based methods. Clausal representation and SL-resolution can obviously be used for boolean formulas, and such a method is indeed implemented in the current version of Prolog-III [5]. However, the performances of this solver are very poor and limit its use to small problems. An improved resolution-based algorithm using a relaxation procedure is described in [8].

BDD-based methods. Binary Decision Diagrams (BDD) have recently gained great a success as an efficient way to encode boolean functions [3], and it was natural to try to use them in boolean solvers. BDDs are usually encoded as directed acyclic graphs, where common subexpressions are merged. The boolean unification [12] solver of CHIP uses such a BDD representation [4] [21]. Other solvers using BDDs include the Adia solver [16], its improved version (second method of [18]) and the Wamcc-Adia combination [7]. Such solvers are efficient for some circuit verification applications, but do not have as good results for less symmetrical problems, e.g. traditional boolean benchmarks, for the size of the BDD during the computation can become very large. It is also very costly to maintain a normal form (the BDD) and to recompute it each time a new constraint is added. Moreover, these solvers are not apt to selectively compute, if desired, a single solution instead of all possible ones.

Enumerative methods. These methods roughly consist in trying possible assignments by incrementally instanciating variables to 0 or 1 and checking consistency in various sophisticated ways. The seminal algorithm by Davis and Putman falls into this category, although it can also be reformulated in the previous resolution framework. The basic idea is to build (implicitly or explicitly) a decision tree by instanciating variables and backtracking. Boolean constraints are checked for consistency as soon as all their variables become ground. [15] contains various improvements in consistency checking, and [17] shows how to compute most general unifiers representating all possible models. New methods use a matrix-based clausal form to represent constraints for efficiency reasons, either by bit-vector encoding [13] or with a spare-matrix representation (first method of [18]). They also allow fixed variables to be detected quickly. These solvers can be made quite efficient by introducing various heuristics.

Propagation-based methods. These schemes are based on local propagation techniques developed for finite domain constraints and integrated in CLP languages such as CHIP. They are very close in spirit to enumerative methods (especially [13]), but do not use a particular encoding of boolean constraints, and rather reuse a more general module designed for finite domains constraints. Such a boolean solver is integrated in CHIP and performs usually far better than its boolean unification algorithm, being close to specifically developed solvers. `clp(B/FD)` is another instance of such an approach, as it is based on the `clp(FD)` solver for finite domain constraints.

It is also worth distinguishing in the above classification between special-purpose boolean solvers, which are intended to take a set of boolean formulas as input, and solvers integrated in CLP languages, which offer much more flexibility by providing a full logic language to state the problem and generate the boolean formulas. PrologIII, CHIP and `clp(B/FD)` only fall in the latter category.

3 `clp(FD)` in a nutshell

As introduced in Logic Programming by the CHIP language, `clp(FD)` [6] is a constraint logic language based on finite domains, where constraint solving is done by propagation and consistency techniques originating from Constraint Satisfaction Problems [25] [11] [14]. Very close to those methods are the interval arithmetic constraints of BNR-Prolog [1]. The novelty of `clp(FD)` is the use of a unique primitive constraint which allows the user to define his own high-level constraints. The "black box" approach gives way to "glass box" approach.

3.1 The constraint `X in r`

The main idea is to use a single primitive constraint `X in r`, where X is a *finite domain (FD) variable* and r is a *range*, which can be not only a *constant range*, e.g. 1..10 but also an *indexical range* using:

- `min(Y)` which represents the minimal value of Y (in the current store),
- `max(Y)` which represents the maximal value of Y,
- `val(Y)` which represents the value of Y as soon as Y is ground.

A fragment of the syntax of this (simple) constraint system is given below.

```
c ::=  X in r   (constraint)

r ::=  t..t     (interval range)
       {t}      (singleton range)
       ...

t ::=  C        (parameter)
       n        (integer)
       min(X)   (indexical min)
       max(X)   (indexical max)
       val(X)   (delayed value)
       t + t    (addition)
       t - t    (subtraction)
       t * t    (multiplication)
       ...
```

The intuitive meaning of such a constraint is: "X must always belong to r".

The initial domain of an FD variable is `0..`∞ and is gradually reduced by `X in r` constraints which replace the current domain of X (D_X) by $D'_X = D_X \cap r$ at each modification of r. An inconsistency is detected when D'_X is empty. Obviously, such a detection is correct if the range denoted by r can only decrease. So, there are some monotone restrictions about the constraints [23]. To deal with the special case of anti-monotone constraints we use the general *forward checking* propagation mechanism [9] which consists in awaking a constraint only when its arguments are *ground* (i.e. with singleton domains). In `clp(FD)` this is achieved using a new indexical term `val(X)` which delays the activation of a constraint in which it occurs until X is ground.
As shown in the previous table, it is possible to define a constraint w.r.t. the *min* or the *max* of some other variables, i.e. reasoning about the bounds of the intervals (*partial lookahead* [22]). `clp(FD)` also allows operations about the whole domain in order to also propagate the "holes" (*full lookahead* [22]). Obviously, these possibilities are useless when we deal with boolean variables since the domains are restricted to 0..1.

3.2 High-level constraints and propagation mechanism

From `X in r` constraints, it is possible to define high-level constraints (called *user constraints*) as Prolog predicates. Each constraint specifies how the *constrained variable* must be updated when the domains of other variables change. In the following examples X, Y are FD variables and C is a *parameter* (runtime constant value).

```
'x+y=c'(X,Y,C):-   X in C-max(Y)..C-min(Y), (C1)
                   Y in C-max(X)..C-min(X). (C2)

'x-y=c'(X,Y,C):-   X in min(Y)+C..max(Y)+C, (C3)
                   Y in min(X)-C..max(X)-C. (C4)
```

The constraint `x+y=c` is a classical FD constraint reasoning about intervals. The domain of X is defined w.r.t. the bounds of the domain of Y.

In order to show how the propagation mechanism works, let us trace the resolution of the system $\{X + Y = 4, X - Y = 2\}$ (translated via `'x+y=c'(X,Y,4)` and `'x-y=c'(X,Y,2)`):
After executing `'x+y=c'(X,Y,4)`, the domain of X and Y are reduced to 0..4 (C_1 is in the current store: $X\ in\ -\infty..4$, $C_2 : Y\ in\ -\infty..4$). And, after executing `'x-y=c'(X,Y,2)`, the domain of X is reduced to 2..4 ($C_3 : X\ in\ 2..6$), which then reduces the domain of Y to 0..2 ($C_4 : Y\ in\ 0..2$).
Note that the unique solution $\{X = 3, Y = 1\}$ has not yet been found. Indeed, in order to efficiently achieve consistency, the traditional method (arc-consistency) only checks that, for any constraint C involving X and Y, for each value in the domain of X there exists a value in the domain of Y satisfying C and vice-versa (see [25] [11] [14] for more details). So, once arc-consistency has been achieved and the domains have been reduced, an enumeration (called labeling) has to be done on the domains of the variables to yield the exact solutions. Namely, X is assigned to one value in D_X, its consequences are propagated to other variables, and so on. If an inconsistency arises, other values for X are tried by backtracking. Note that the order used to enumerate the variables and to generate the values for a variable can improve the efficiency in a very significant manner (see heuristics in [22]).
In our example, when the value 2 is tried for X, C_2 and C_4 are awoken (because they depend on X). C_2 sets Y to 2 and C_4 detects the inconsistency when it tries to set Y to 0. The backtracking reconsiders X and tries value 3 and, as previously, C_2 and C_4 are reexecuted to set (and check) Y to 1. The solution $\{X = 3, Y = 1\}$ is then obtained.

3.3 Optimizations

The uniform treatment of a single primitive for all complex user constraints leads to a better understanding of the overall constraint solving process and allows for (a few) global optimizations, as opposed to the many local and particular optimizations hidden inside the black box. When a constraint `X in r` has been reexecuted, if $D'_X = D_X$ it was useless to reexecute it (i.e. it has neither failed nor reduced the domain of X). Hence, we have designed three simple but powerful optimizations for the `X in r` constraint which encompass many previous particular optimizations for FD constraints:

- some constraints have the same solutions in any computed store, so only the execution of one of them is needed. In the previous example, when C_2 is called in the store $\{X\ in\ 0..4, Y\ in\ 0..\infty\}$ Y is set to 0..4. Since the domain of Y has been updated, all constraints depending on Y are reexecuted and C_1 ($X\ in\ 0..4$) is awoken unnecessary.
- in some cases, it is useless to reexecute a constraint `X in r` if X is ground. Intuitively, this is possible when the constraints are written in such a way that when X becomes ground, domains of related variables are reduced so that no further check of `X in r` is needed.
- when a constraint is awoken more than once from several distinct variables, only one reexecution is necessary. This optimization is obvious since the order of constraints, during the execution, is irrelevant for correctness.

These optimizations make it possible to avoid on average 45 % of the total number of constraint executions on a traditional set of FD benchmarks (see [6] for full details) and up to 57 % on the set of boolean benchmarks presented below.

4 The boolean constraint system

Let us now detail the formalization of boolean expressions in terms of constraint systems. We give, in this way, an operational semantics to the propagation-based boolean solver and prove its equivalence to the declarative semantics of boolean expressions (truth-tables).

4.1 Constraint systems

The simplest way to define constraints is to consider them as first-order formulas interpreted in some non-Herbrand structure [10], in order to take into account the particular semantics of the constraint system. Such *declarative* semantics is adequate when a non-Herbrand structure exists beforehand and fits well the constraint system (e.g. $\mathcal{R}$ for arithmetic constraints), but does not work very well for more practical constraint systems (e.g. finite domains). Obviously, it cannot address any operational issues related to the constraint solver itself. Recently, another formalization has been proposed by [19], which can be seen as a first-order generalization of Scott's *information sytems* [20]. The emphasis is put on the definition of an *entailment* relation (noted $\vdash$) between constraints, which suffices to define the overall constraint system. Such an approach is of prime importance in the framework of concurrent constraint languages, but is also useful for pure CLP, as it makes it possible to define a constraint system ex nihilo by giving the entailment relation and verifying some basic properties. The entailment relation is given by rules, and we can therefore define a kind of *operational semantics* of the entailment between constraints. This will be particularly useful when defining our propagation-based boolean constraint system, as the entailment relation will accurately represents how information is propagated between constraints.

Definition [19]
A constraint system is a pair $(D, \vdash)$ satisfying the following conditions:

1. D is a set of first-order formulas closed under conjunction and existential quantification.
2. $\vdash$ is an *entailment* relation between a finite set of formulas and a single formula satisfying the following inference rules:

$$\Gamma, d \vdash d \ \ (\textbf{Struct}) \qquad \frac{\Gamma_1 \vdash d \quad \Gamma_2, d \vdash e}{\Gamma_1, \Gamma_2 \vdash e} \ \ (\textbf{Cut})$$

$$\frac{\Gamma, d, e \vdash f}{\Gamma, d \wedge e \vdash f} \ \ (\wedge \vdash) \qquad \frac{\Gamma \vdash d \quad \Gamma \vdash e}{\Gamma \vdash d \wedge e} \ \ (\vdash \wedge)$$

$$\frac{\Gamma, d \vdash e}{\Gamma, \exists X.\, d \vdash e} \ \ (\exists \vdash) \qquad \frac{\Gamma \vdash d[t/X]}{\Gamma \vdash \exists X.\, d} \ \ (\vdash \exists)$$

In $(\exists \vdash)$, X is assumed not free in Γ, e.

3. $\vdash$ is *generic*: that is $\Gamma[t/X] \vdash d[t/X]$ whenever $\Gamma \vdash d$, for any term t.

In order to build constraint systems, it suffices to define a *pre-constraint system* $(D, \vdash)$ satisfying only **(Struct)**, **(Cut)** and the genericity condition. Existential quantification and conjunction can be added in a straightforward way, as stated by the following theorem.

Theorem [19]
Let $(D', \vdash')$ be a pre-constraint system. Let D be the closure of D' under existential quantification and conjunction, and $\vdash$ the closure of $\vdash'$ under the basic inference rules. Then $(D, \vdash)$ is a constraint system.

As an important corollary, a constraint system can be constructed even more simply from any first-order theory, i.e. any set of first-order formulas. Consider a theory T and take for D the closure of the subset of formulas in the vocabulary of T under existential quantification and conjunction. Then one defines the entailment relation $\vdash_T$ as follows. $\Gamma \vdash_T d$ iff Γ entails d in the logic, *with the extra non-logical axioms of* T.
Then $(D, \vdash_T)$ can be easily verified to be a constraint system.
Observe that this definition of constraint systems thus naturally encompases the traditional view of constraints as interpreted formulas.

4.2 Boolean constraints

Definition
Let $\mathcal{V}$ be an enumerable set of variables. A *boolean constraint on* $\mathcal{V}$ is one of the following formulas:
$and(X, Y, Z)$, $or(X, Y, Z)$, $not(X, Y)$, $X = Y$, for $X, Y, Z \in \mathcal{V}$.
The intuitive meaning of these constraints are: $X \wedge Y \equiv Z$, $X \vee Y \equiv Z$, $X \equiv \neg Y$, and $X \equiv Y$. We note $\mathcal{B}$ be the set of all such boolean constraints.

Let us now present the rules defining the propagation between boolean constraints.

Definition
Let B be the following first-order theory on $\mathcal{B}$-formulas:

0=0	1=1
and(X,Y,Z), X=0 $\rightarrow$ Z=0	and(X,Y,Z), Y=0 $\rightarrow$ Z=0
and(X,Y,Z), X=1 $\rightarrow$ Z=Y	and(X,Y,Z), Y=1 $\rightarrow$ Z=X
and(X,Y,Z), Z=1 $\rightarrow$ X=1	and(X,Y,Z), Z=1 $\rightarrow$ Y=1
or(X,Y,Z), X=1 $\rightarrow$ Z=1	or(X,Y,Z), Y=1 $\rightarrow$ Z=1
or(X,Y,Z), X=0 $\rightarrow$ Z=Y	or(X,Y,Z), Y=0 $\rightarrow$ Z=X
or(X,Y,Z), Z=0 $\rightarrow$ X=0	or(X,Y,Z), Z=0 $\rightarrow$ Y=0
not(X,Y), X=0 $\rightarrow$ Y=1	not(X,Y), X=1 $\rightarrow$ Y=0
not(X,Y), Y=0 $\rightarrow$ X=1	not(X,Y), Y=1 $\rightarrow$ X=0

Boolean propagation theory B

Observe that it is easy to enrich, if desired, this constraint system by other boolean constraints such as *xor* (exclusive or), *nand* (not and), *nor* (not or), $\Leftrightarrow$ (equivalence), or $\Rightarrow$ (implication) by giving the corresponding rules, but they can also be decomposed into the basic constraints.

We can now define the entailment relation $\vdash_B$ between boolean constraints and the boolean constraint system:

Definitions
Consider a store Γ and a boolean constraint b.
$\Gamma \vdash_B b$ iff Γ entails b with the extra axioms of B.

The *boolean constraint system* is $(\mathcal{B}, \vdash_B)$.

It is worth noticing that the rules of B (and thus $\vdash_B$) precisely encode the propagation mechanisms that will be used to solve boolean constraints. We have indeed given the operational semantics of the constraint solver in this way. The most elegant way to implement such a solver would be to use some *Ask* primitive in a concurrent constraint language, as proposed by [24]. We do not have such a facility in `clp(FD)`, and we will encode this propagation scheme by `X in r` constraints, as is detailed below.

4.3 Correctness and completeness of $(\mathcal{B}, \vdash_B)$

It is important to ensure that our (operationally-defined) constraint system is equivalent to traditional boolean expressions. To do so, we have to prove that our entailment relation derives the same results as the declarative semantics of booleans given by the truth-tables of the *and*, *or* and *not* operators.

Theorem
The $\mathit{and}(X, Y, Z)$, $\mathit{or}(X, Y, Z)$, and $\mathit{not}(X, Y)$ constraints are satisfied for some values of X, Y and Z iff the tuple of variables is given by the truth-tables of the corresponding boolean operators.

Proof.
It must be shown that, for $\mathit{and}(X, Y, Z)$ and $\mathit{or}(X, Y, Z)$, once X and Y are bound to some value, the value of Z is correct, i.e. it is unique (if several rules can be applied, they give the same result) and it is equal to the value given by the corresponding truth-table, and that all rows of the truth-tables are reached.
This can be verified by a straightforward case analysis.
For $\mathit{not}(X, Y)$ it can be easily shown that for any X, Y is given the opposite value.

5 Building `clp(B/FD)`

In this section we specify the constraint solver, i.e. we define a user constraint for each boolean constraint presented above. We then prove the correctness and completeness of this solver, and show how it really encodes the "operational semantics" defined by theory B.

5.1 Designing the constraints

The design of the solver only consists in defining a user constraint for each boolean constraint. As the constraint `X in r` makes it possible to use arithmetic operations on the bounds of a domain, we use some mathematical relations satisfied by the boolean constraints:

```
and(X,Y,Z):-   Z in min(X)*min(Y)..max(X)*max(Y),
               X in min(Z)..max(Z)*max(Y)+1-min(Y),
               Y in min(Z)..max(Z)*max(X)+1-min(X).

or(X,Y,Z):-    Z in min(X)+min(Y)-min(X)*min(Y)..
                    max(X)+max(Y)-max(X)*max(Y),
               X in min(Z)*(1-max(Y))..max(Z),
               Y in min(Z)*(1-max(X))..max(Z).

not(X,Y):-     X in {1-val(Y)},
               Y in {1-val(X)}.
```

Table 1: The boolean solver definition

$and(X, Y, Z)$	satisfies	$Z = X \times Y$
		$Z \leq X \leq Z \times Y + 1 - Y$
		$Z \leq Y \leq Z \times X + 1 - X$
$or(X, Y, Z)$	satisfies	$Z = X + Y - X \times Y$
		$Z \times (1 - Y) \leq X \leq Z$
		$Z \times (1 - X) \leq Y \leq Z$
$not(X, Y)$	satisfies	$X = 1 - Y$
		$Y = 1 - X$

The definition of the solver is then quite obvious and presented in table 1. It only encodes the above relations.

5.2 Correctness and completeness of `clp(B/FD)`

Theorem
The *and*, *or*, and *not* user constraints are correct and complete.

Proof.
The proof of correctness consists in showing that each $\{0, 1\}$ tuple satisfying the relations defined above is an element of the appropriate truth-table. Completeness w.r.t declarative semantics can be easily proved conversely, but we are mainly interested in showing that each time a rule of B can fire, the store is reduced as specified by the rule. Namely, any tuple of variables satisfies the corresponding mathematical relations enforced by the constraint solver. Here again, a case analysis proves the result. For instance if $and(X, Y, Z), Y = 1 \rightarrow Z = X$ fires, $Z \leq X \leq Z \times Y + 1 - Y$ is verified in the resulting store.

6 Performance evaluations

6.1 The benchmarks

In order to test the performances of `clp(B/FD)` we have tried a set of traditional boolean benchmarks:

- `schur`: Schur's lemma. We try to put the integers $\{1 \ldots n\}$ in three boxes so that for any triplet (x, y, z) such that $x + y = z$, x, y and z do not belong

Program	CHIP Time (ms)	`clp(B/FD)` Time (ms)	$\frac{\text{CHIP}}{\texttt{clp(B/FD)}}$
`schur  13`	1200	220	5.45
`schur  14`	1310	240	5.45
`schur  30`	15070	1230	12.25
`pigeon 6/6`	2770	440	6.29
`pigeon 8/8`	177580	29830	5.95
`pigeon 8/7`	18940	2750	6.88
`queens  8`	6720	750	8.96
`queens  9`	26060	2750	9.47
`queens 16 (first)`	42480	4350	9.76
`queens 20 (first)`	$overflow$	57890	
`queens 30 (first)`	$overflow$	2459510	

Table 2: Comparison with CHIP

to the same box. We use $3 \times n$ variables to indicate, for each integer, its box number. This problem has a solution iff $n \leq 13$.

- `pigeon`: pigeon-hole problem. We try to put n pigeons in m holes (at most 1 pigeon per hole). We use $n \times m$ variables to indicate, for each pigeon, its hole number. Obviously, there is a solution iff $n \leq m$.

- `queens`: we have to place n queens on a $n \times n$ chessboard so that there is no couple of queens threatening each other. The boolean formulation uses $n \times n$ variables to indicate, for each square, if there is a queen on it.

All solutions are computed unless otherwise stated. The results presented below for `clp(B/FD)` do not include any heuristics and have been measured on a Sony MIPS R3000 workstation (17 Mips). The following section compares `clp(B/FD)` with the commercial version of CHIP. We have chosen CHIP for the main comparison because it is a commercial product and a CLP language (and not only a constraint solver) and thus accepts the same programs as `clp(B/FD)`. Moreover, it also uses a boolean constraint solver based on finite domains[1]. We also compare `clp(B/FD)` with some specific constraint solvers.

6.2 `clp(B/FD)` vs CHIP

Times for CHIP were measured on a Sun Sparc 2 (28.5 Mips) and normalized by a factor of 1.6. Exactly the same programs were run on both systems.
The average speedup of `clp(B/FD)` w.r.t. CHIP is around a factor of eight, with peak speedup reaching more than twelve (see table 2, where $overflow$ means that too much memory was needed for the corresponding program). This factor of eight can be compared with the factor of two that we have on the traditional FD benchmarks. The main reasons for this gap could be that in CHIP, booleans are written on an existing solver whereas we have developed an appropriate solver thanks to the `X in r` primitive, and that we have global optimizations for primitive constraints from which all user constraints can benefit.

[1]the other solver of CHIP, based on boolean unification, became quickly unpracticable: none of the benchmarks presented here could even run with it, due to memory problems.

Program	Bdd worst Time (ms)	Bdd best Time (ms)	`clp(B/FD)` Time (ms)	$\frac{\text{Bdd worst}}{\texttt{clp(B/FD)}}$	$\frac{\text{Bdd best}}{\texttt{clp(B/FD)}}$
`schur 13`	5220	1780	220	23.72	8.09
`schur 14`	8080	2290	240	33.66	9.54
`pigeon 6/6`	1250	160	460	2.71	↓ 2.87
`pigeon 8/8`	24800	930	29830	↓ 1.20	↓ 32.07
`pigeon 8/7`	7280	500	2750	2.64	↓ 5.50
`queens 6`	3860	1620	30	128.66	54.00
`queens 8`	94740	86000	750	126.32	114.66

Table 3: Comparison with a BDD method

Program	Enum Time (ms)	`clp(B/FD)` Time(ms)	$\frac{\text{Enum}}{\texttt{clp(B/FD)}}$
`schur 13`	1350	220	6.13
`schur 14`	1470	240	6.12
`pigeon 5/5`	350	30	11.66
`pigeon 6/6`	3850	440	8.75
`queens 8`	2420	750	3.22
`queens 9`	11600	2750	4.21

Table 4: Comparison with an enumerative method

6.3 `clp(B/FD)` vs the rest

In this section, we compare `clp(B/FD)` with other specific boolean solvers. These solvers are not programming languages, they accept a set of constraints as input and solve it. So there are as many formulations as problem instances. On the other hand, `clp(B/FD)` generates constraints at runtime (the overhead thus introduced is limited to 20 %, so we do not need to worry too much about that). Another important point to mention is that were not able to run exactly the same programs, and we have used time measurements provided by the referenced papers (which usually incorporate a large number of heuristics).

6.3.1 `clp(B/FD)` vs BDD methods

Adia is an efficient boolean constraint solver based on the use of BDDs. Time measurements presented below are taken from [18] who tries four different heuristics on a Sun Sparc IPX (28.5 Mips). We have chosen the worst and the best of these four timings for Adia (normalized with a factor of 1.6). Note that the BDD approach computes all solutions and is thus unpracticable when we are only interested in one solution for big problems such as `queens` for $n \geq 9$ and `schur` for $n = 30$). Here again, `clp(B/FD)` has very good speedups (see table 3).

6.3.2 `clp(B/FD)` vs enumerative methods

[17] provides time measurements for an enumerative method for boolean unification on a Sun 3/80 (1.5 Mips). We normalized these measurements by a factor of 1/11.3. The average speedup is 6.5 (see table 4).

Program	L. Propag Time (ms)	`clp(B/FD)` Time (ms)	$\frac{\text{L. Propag}}{\texttt{clp(B/FD)}}$
`schur  13`	130	220	↓ 1.69
`schur  14`	140	240	↓ 1.71
`pigeon 6/6`	1480	440	3.36
`pigeon 8/8`	113580	29830	3.80
`queens  8`	3040	750	4.05
`queens  9`	13030	2750	4.73
`queens 14 (first)`	5280	1180	4.47
`queens 16 (first)`	30200	4350	6.94

Table 5: Comparison with a local consistency method

6.3.3 `clp(B/FD)` vs a local consistency method

Here, we refer to [13] who presents results of a boolean constraint solver based one the use of local consistency techniques. Times are given on a Macintosh SE/30 equivalent to a Sun 3/50 (1.5 Mips). We normalized them with a factor of 1/11.3. This solver includes two labeling heuristics, the most important being the ability to dynamically order the variables w.r.t. the number of constraints still active on them. On the other hand, `clp(B/FD)` only uses a static order (standard labeling). An interesting point is that the factors are quite constant within a class of problem. `clp(B/FD)` is slower on the `schur` benchmark by a factor of 1.7, three times faster on `pigeon` and four times faster on `queens` (see table 5 for more details). We conjecture that this is because both solvers certainly perform much the same pruning, although they are based on very different data-structures for the constraints and constraint network.

7 Conclusion and perspective

We have presented a very simple boolean constraint solver `clp(B/FD)` built upon the finite domain constraint logic language `clp(FD)`.
We have formally defined the boolean constraint system by a rule-based operational semantics for the entailment relation which encodes the propagation scheme for boolean constraint and proved its equivalence w.r.t the declarative definition of boolean expressions through truth-tables. Moreover, we also proved that the `clp(B/FD)` solver really encodes that semantics.
The `clp(B/FD)` solver is very efficient, being eight times faster than the CHIP boolean solver on average, and also several times faster than special-purpose stand-alone boolean solvers. This proves firstly that the propagation techniques proposed for finite domains are very competitive for booleans and secondly that, among such solvers, the “glass-box”approach of using a single primitive constraint `X in r` is very interesting and it makes it possible to encode other domains (such as boolean domains) at a low-level, with better performances than the “black-box”. An additional advantage is the complete explicitation of the propagation scheme.
Nevertheless, performances can be improved by simplifying the data-structures used in `clp(FD)`, which are designed for full finite domain constraints, and specializing them for booleans by explicitly introducing a new type and new instructions for boolean variables. For instance, it is possible to reduce the variable frame repre-

senting the domain of a variable and its associated constraints to only two words: one pointing to the chain of constraints to awake when the variable is bound to 0 and the other when it is bound to 1. Time-stamps also become useless for boolean variables, and a further improvement can be made in the computation of ranges if direct execution of C functions is possible in ranges. Such a specialized solver should provide a speedup two or three times greater than `clp(B/FD)`.
It is worth noticing that in `clp(FD)` the only heuristic available for labeling is the classical "first-fail" based on the size of the domains which is obviously useless for boolean constraints. Some more flexible primitives (e.g. number of constraints on X, number of constraints using X) would be necessary in order to express, at the language level, some complex labeling heuristics [22], [13].

References

[1] BNR-Prolog User's Manual. Bell Northern Research. Ottawa, Canada, 1988.

[2] A. Bockmayr. Logic Programming with Pseudo-Boolean Constraints. Research report MPI-I-91-227, Max Planck Institut, Saarbrucken, Germany, 1991.

[3] R.E. Bryant, Graph Based Algorithms for Boolean Fonction Manipulation. *IEEE Transactions on computers*, no. 35 (8), 1986, pp 677–691.

[4] W. Büttner and H. Simonis. Embedding Boolean Expressions into Logic Programming. *Journal of Symbolic Computation*, no. 4 (1987), pp 191-205.

[5] A. Colmerauer. An introduction to PrologIII. *Communications of the ACM*, no. 28 (4), 1990, pp 412-418.

[6] D. Diaz and P. Codognet. A Minimal Extension of the WAM for `clp(FD)`. In *proc. ICLP'93, 10th International Conference on Logic Programming*, Budapest, Hungary, MIT Press 1993.

[7] G. Dore and P. Codognet. A Prototype Compiler for Prolog with Boolean Constraints. In *proc. GULP'93, Italian Conference on Logic Programming*, Gizzeria Lido, Italy, 1993.

[8] G. Gallo, G. Urbani, Algorithms for Testing the Satisfiability of Propositional Formulae. *Journal of Logic Programming*, no. 7 (1989), pp 45-61.

[9] R. M. Haralick and G. L. Elliot. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence 14 (1980)*, pp 263-313

[10] J. Jaffar and J-L. Lassez. Constraint Logic Programming. In *proc. POPL-87, Principles Of Programming Languages*, Munich, Germany, January 1987.

[11] A. K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence 8 (1977)*, pp 99-118.

[12] U. Martin, T. Nipkow, Boolean Unification – The story so far. *Journal of Symbolic Computation*, no. 7 (1989), pp 191-205.

[13] J-L. Massat. Using Local Consistency Techniques to Solve Boolean Constraints. In *Constraint Logic Programming: Selected Research*, A. Colmerauer and F. Benhamou (Eds.). The MIT Press, 1993.

[14] B. A. Nadel. Constraint Satisfaction Algorithms. *Computational Intelligence 5 (1989)*, pp 188-224.

[15] A. Rauzy. *L'Evaluation Semantique en Calcul Propositionnel*. PhD thesis, Universite Aix-Marseille II, Marseille, France, January 1989.

[16] A. Rauzy. Adia. Technical report, LaBRI, Université Bordeaux I, 1991.

[17] A. Rauzy. Using Enumerative Methods for Boolean Unification. In *Constraint Logic Programming: Selected Research*, A. Colmerauer and F. Benhamou (Eds.). The MIT Press, 1993.

[18] A. Rauzy. Some Practical Results on the SAT Problem, Draft, 1993.

[19] V. Saraswat. The Category of Constraint Systems is Cartesian-Closed. In *proc. LICS'92, Logic In Computer Science*, IEEE Press 1992.

[20] D. S. Scott Domains for Denotational Semantics. In *proceedings of ICALP'82, International Colloquium on Automata Languages and Programmaing*, 1982.

[21] H. Simonis, M. Dincbas, *"Propositional Calculus Problems in CHIP"*, ECRC, Technical Report TR-LP-48, 1990.

[22] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* Logic Programming Series, The MIT Press, Cambridge, MA, 1989.

[23] P. Van Hentenryck, V. Saraswat and Y. Deville. Constraint Processing in cc(FD). Draft, 1991.

[24] P. Van Hentenryck, H. Simonis and M. Dincbas. Constraint Satisfaction Using Constraint Logic Programming. *Artificial Intelligence* no 58, pp 113-159, 1992.

[25] P. Van Hentenryck, Y. Deville and C-M. Teng. A Generic Arc-Consistency Algorithm and its Specializations. *Artificial Intelligence 57 (1992)*, pp 291-321.