clp(B) : Combining Simplicity and Efficiency in Boolean Constraint Solving

Philippe Codognet and Daniel Diaz

INRIA-Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex, FRANCE
{`Philippe.Codognet, Daniel.Diaz`}`@inria.fr`

Abstract. We present the design and the implementation of `clp(B)`: a boolean constraint solver inside the Constraint Logic Programming paradigm. This solver is based on local propagation methods and follows the "glass-box" approach of compiling high-level constraints into primitive low-level ones. We detail its integration into the WAM showing that the necessary extension is truly minimal since only four new instructions are added. The resulting solver is around an order of magnitude faster than other existing boolean solvers.

1 Introduction

Constraint Logic Programming (CLP) 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 constraint domains currently developed in CLP, booleans are widely investigated and can be found in several CLP languages, as for instance CHIP [19], PrologIII [7] or `clp(FD)` [9].
The history of efficient machine solving of boolean problems goes back to the early 60's with the seminal work of Davis and Putnam, and, since that time, many different algorithms and solvers have been developed and implemented. The study of boolean problems indeed spread over different research communities, such as automated theorem proving, circuit design and verification, artificial intelligence, operational research, pure logic and CLP. It is therefore not surprizing that many tools for solving boolean constraints exist, either as stand-alone solvers, taking as input some special boolean formulation of the problem (e.g. clauses or various other normal forms), or integrated into full CLP languages, making it possible for a much easier and natural formulation of the problem. 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 [18]. 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, developement of which was pioneered in the CLP world by the CHIP language, have also been used to solve boolean constraints with some success; in fact 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. A similar approach has been used to define `clp(B/FD)` where boolean constraint solving is performed on top of the finite domain constraint solver of `clp(FD)` [4]. Indeed, the boolean constraints are encoded thanks to the low-level X *in* r primitive of `clp(FD)`. This idea of compiling complex constraints into simpler low-level constraints has been originally proposed for finite domain constraints by [20] and called the "glass-box" approach. The `clp(B/FD)` approach has shown to be fairly efficient (eight times faster than the CHIP propagation-based boolean solver). Nevertheless, performances can be pushed a step further by simplifying the data-structures used in `clp(FD)`, which are indeed designed for full finite domains constraints. They can be specialized by introducing explicitly a new type and new instructions for boolean variables. It is possible, for instance, to reduce the data-structure representing 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. Also some other data-structures become useless for boolean variables, and can be avoided. The resulting `clp(B)` solver is very compact and simple; it is based again on the glass-box approach, and only uses a single low-level constraint, more specialized than the X *in* r construct, into which boolean constraints such as *and*, *or* or *not* are decomposed. These simplifications improve the performances by more than a factor two giving a solver which is around an order of magnitude faster than other existing boolean solvers. The low-level primitive constraint at the core of `clp(B)` can be implemented into a WAM-based logical engine with a minimal extension: only four new abstract instructions are needed. Therefore this very simple but very efficient boolean constraint solver can be incomporated into any Prolog system very easily.

The rest of the paper is organized as follows: section 2 introduces boolean constraints. Section 3 presents `clp(B)` and the primitive constraint at the core of the solver while section 4 describes its implementation into the WAM. Section 5 details performance evaluation and compares to other boolean solvers. A short conclusion and research perspectives end the paper.

2 The boolean constraint system

2.1 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 suits 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 [16], which can be seen as a first-order generalization of Scott's *information systems* [17]. The emphasis is put on the definition of an *entailment* relation (noted $\vdash$) between constraints, which suffices to define the overall constraint system. A constraint system is thus defined as 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 some basic properties corresponding, roughly speaking, to reflexivity, transitivity in information systems and introduction/elimination of $\wedge$ and $\exists$ in sequent calculus (see [16] for more explanations).
3. $\vdash$ is *generic*: that is $\Gamma[t/X] \vdash d[t/X]$ whenever $\Gamma \vdash d$, for any term t.

2.2 Boolean constraints

Definition
Let $\mathcal{V}$ be an enumerable set of variables. A *boolean constraint on* $\mathcal{V}$ is one of the following formulas:

$$and(X,Y,Z)\ ,\ or(X,Y,Z)\ ,\ not(X,Y)\ ,\ X = Y\ ,\ \text{for } X,Y,Z \in \mathcal{V} \cup \{0,1\}$$

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 first-order theory on $\mathcal{B}$-formulas presented in table 1:

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

Table 1. 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.

3 Designing `clp(B)`

Here we are interested in designing a specific propagation-based boolean solver that encodes exactly the propagation rules presented in table 1. This solver will follow the glass-box paradigm and will be called `clp(B)`. It will prove that local propagation techniques are a very efficient way to deal with boolean constraints. Moreover this solver indeed reduces to a surprisingly simple instruction set which will make it possible to integrate boolean constraints in any Prolog compiler. It is worth noticing that all well-known boolean solvers (CHIP, PrologIII, etc) are based on the black-box approach, i.e. nobody knows exactly what there is inside these solvers, except [8] which presents a glass-box (re)construction of a boolean solver based on Binary Decision Diagrams (BDDs). From a design point of view, `clp(B)` is very similar to `clp(FD)`. It is based on a low-level primitive constraint $l_0 <= l_1, \ldots, l_n$ and it offers the possibility to define high-level constraints as Prolog predicates. Complex boolean constraints are also translated at compile-time by a preprocessor.

3.1 The primitive constraint $l_0 <= l_1, \ldots, l_n$

Since the initial domain of a boolean variable is 0..1 it can be reduced only once. A constraint is only triggered when some of its variables have become ground, and, if this activation is useful, then the constrained variable will also become ground. Thus, the more appropriate primitive must allows us to express propagation rules which look like "as soon as X is false then set Z to false" and "as soon as both X and Y are true then set Z to true" (for `and(X,Y,Z)`). Note the difference with the `clp(B/FD)` formulation where the primitive X *in* r was used in a computational way to calculate the value (0 or 1) to assign. The behavior of this primitive is very similar to the ask definition of `and(X,Y,Z)` presented in [21]. Thus, we propose a primitive constraint $l_0 <= l_1, \ldots, l_n$ where

```
c ::= l<=[l,...,l] (constraint)

l ::=  X          (positive literal)
      -X          (negative literal)
```

Table 2. Syntax of the constraint $l_0 <= l_1, \ldots, l_n$

each l_i is either a positive literal (X) or a negative literal ($-X$) (see table 2 for a description of the syntax).
Associated to each literal l_i we define X_i as its variable and $Bvalue_i$ as its truth-value. More precisely if $l_i \equiv -X$ or $l_i \equiv X$ then $X_i = X$. Similarly if $l_i \equiv -X$ (resp. $l_i \equiv X$) then $Bvalue_i = 0$ (resp. $Bvalue_i = 1$).

The intuitive meaning of $l_0 <= l_1, \ldots, l_n$ being "l_0 must be entailed in any store which entails $l_1 \wedge ... \wedge l_n$" where l_i is entailed in a store iff $X_i = Bvalue_i$ is entailed in this store.

Without any loss of generality, we can consider that there is only either one or two literals in the body of the primitive constraint. Indeed, the case $n = 0$ comes down to unify X_0 to $Bvalue_0$ (see section 4.1) and the case $n > 2$ can be rewritten by replacing $l_0 <= [l_1, l_2, l_3, ..., l_n]$ by $l_0 <= [l_1, I_2]$, $I_2 <= [l_2, I_3], ...,$ $I_{n-1} <= [l_{n-1}, l_n]$, where each I_k is a distinct new boolean variable. In `clp(B)` a preprocessor is used for these code-rewriting. This decomposition will allow us to implement very efficiently the tell operation as shown below since only remain the two cases $n = 1$ and $n = 2$.

3.2 Defining the constraints

The glass-box approach for designing constraint solvers, as originally proposed by [20] and developed in [9,5], consists in defining high-level constraints such as linear equations and inequations in finite domains or conjunction, disjunction and negation in booleans by means of very simple and unique low-level primitive constraint. We here define a high-level constraint for each boolean constraint thanks to the $l_0 <= l_1, \ldots, l_n$ primitive which allows us to directly encode the propagation rules above presented. The definition of the solver is then quite obvious and presented in table 3.

4 Integration of $l_0 <= l_1, \ldots, l_n$ into the WAM

Let us now specify the abstract instruction set needed to implement the boolean constraint solver, i.e. the unique constraint $l_0 <= l_1, \ldots, l_n$, into the standard abstract machine used in Logic Programming, namely the Warren Abstract Machine. See [22,1] for a comprehensive introduction to the WAM.

```
and(X,Y,Z):-  Z <= [X,Y],  -Z <= [-X],  -Z <= [-Y],
              X <= [Z],    -X <= [Y,-Z],
              Y <= [Z],    -Y <= [X,-Z].

or(X,Y,Z):-  -Z <= [-X,-Y], Z <= [X],    Z <= [Y],
             -X <= [-Z],    X <= [-Y,Z],
             -Y <= [-Z],    Y <= [-X,Z].

not(X,Y):-    X <= [-Y],   -X <= [Y],
              Y <= [-X],   -Y <= [X].
```

Table 3. The boolean solver definition

4.1 Modifying the WAM for boolean variables

Here, we explain the necessary modifications of the WAM to manage a new data type: boolean variables. They will be located in the heap, and an appropriate tag is introduced to distinguish them from Prolog variables. Dealing with boolean variables slightly affects data manipulation, unification, indexing and trailing instructions.

Data manipulation. Boolean variables, as standard WAM unbound variables, cannot be duplicated (unlike it is done for terms by structure-copy). For example, loading an unbound variable into a register consists of creating a binding to the variable whereas loading a constant consists of really copying it. In the standard WAM, thanks to self-reference representation for unbound variables, the same copy instruction can be used for both of these kinds of loading. Obviously, a boolean variable cannot be represented by a self-reference, so we must take care of this problem. When a source word W_s must be loaded into a destination word W_d, if W_s is a boolean variable then W_d is bound to W_s or else W_s is physically copied into W_d.

Unification. A boolean variable X can be unified with:

- an unbound variable Y: Y is just bound to X,
- an integer n: if $n = 0$ or $n = 1$ the pair (X, n) is enqueued and the *consistency procedure* is called (see sections 4.3 and 4.4).
- another boolean variable Y: equivalent to $X <= [Y]$, $-X <= [-Y]$, $Y <= [X]$ and $-Y <= [-X]$[1].

Indexing. The simplest way to manage a boolean variable is to consider it as an ordinary unbound variable and thus try all clauses.

[1] we will describe later how constraints are managed.

Trailing In the WAM, unbound variables only need one word (whose value is fully defined by their address thanks to self-references), and can only be bound once, thus trailed at most once. When a boolean variable is reduced (to an integer $n = 0/1$) the tagged word `<BLV,_>` (see section 4.2) is replaced by `<INT, n>` and the tagged word `<BLV,_>` may have to be trailed. So a *value-trail* is necessary. Hence we have two types of objects in the trail: one-word entry for standard Prolog variables, two-word entry for trailing one previous value.

4.2 Data structures for constraints

`clp(B)` uses an explicit queue to achieve the propagation (i.e. each triggered constraint is enqueued). It is also possible to uses an implicit propagation queue as discussed in [9]. The register `BP` (Base Pointer) points to the next constraint to execute, the register `TP` (Top Pointer) points to the next free cell in the queue. The other data structure concerns the boolean variable. The frame of a boolean variable X is shown in table 4 and consist of:

– the tagged word,
– the list of constraints depending on X. For reasons of efficiency two lists are used: constraints depending on $-X$ (`Chain_0`) and constraints depending on X (`Chain_1`).

`Chain_1`		`(pointer to a R_Frame)`
`Chain_0`		`(pointer to a R_Frame)`
`BLV`	`unused`	

Table 4. Boolean variable frame (B_Frame)

Since there are at most 2 literals in the body of a constraint $c \equiv l_0 <= l_1, \ldots, l_n$, if c depends on X (i.e $X_1 = X$ or $X_2 = X$) it is possible to distinguish the case $n = 1$ from the case $n = 2$. Intuitively, in the case $n = 1$ the constraint c can be solved as soon as X is ground while c can still suspend until the other variable is ground in the case $n = 2$. (see section 4.4 for more details). So, the case $n = 2$ requires more information about the constraint to trigger since it is necessary to check the other variable before executing it. The frame associated to a record (R_Frame) of the list `Chain_0/1` consists of:

– the address of the boolean which is constrained (i.e. X_0),
– the value to affect (i.e. $Bvalue_0$),
– only if $n = 2$: the address of the other involved boolean variable
– only if $n = 2$: the value to be satisfied by the other involved variable

Table 5 summarizes the contents of a R_Frame.

It is worth noting that, in the case $n = 2$, a record is necessary in the appropriate list of X_1 with a pointer to X_2 and also in the appropriate list of X_2 with a pointer to X_1. This "duplication" is very limited since it only involves 2 additional words. This is enhanced in figure 1 which shows the data structures involved in the constraint `Z<=[-X,Y]` (which could be used in the definition of `xor(X,Y,Z)`). The alternate solution would use 1 additional word to count the number of variables which suspend (the constraint being told as soon as this counter equals 0).

`Bvalue_2`	`\ (only used`
`Blv_2_Adr`	`/  if Bloc2_Flag is true)`
`Bloc2_Flag`	`(case` $n = 2$ `?)`
`Tell_Bvalue`	
`Tell_Blv_Adr`	`(a pointer to a B_Frame)`
`Next_Record`	`(a pointer to a R_Frame)`

Table 5. Record Frame (R_Frame)

4.3 Compilation scheme and instruction set

The compilation of a constraint $l_0 <= l_1, \ldots, l_n$ consists of two parts:

- loading X_0,...,X_n into WAM temporaries (i.e. `Xi` registers),
- installing and telling the constraint, i.e. creating the necessary R_Frame(s), detecting if the body of the constraint is currently entailed by the store (see section 3.1) to enqueue the pair $(X_0, Bvalue_0)$ and to call the *consistency procedure*. (described in section 4.4).

Loading instructions are:

`b_load_variable(Vi,Xj)`
binds `Vi` to a boolean variable created on top of the heap and puts its address into `Xj`.

`b_load_value(Vi,Xj)`
let `w` be the dereferenced word of `Vi`, if it is:
- an unbound variable: similar to `b_load_variable(w,Xj)`.
- an integer n: fails if $n \neq 0$ and $n \neq 1$ or else n is pushed on the heap and its address is stored into `Xj`.
- a boolean variable: its address is stored into `Xj`.

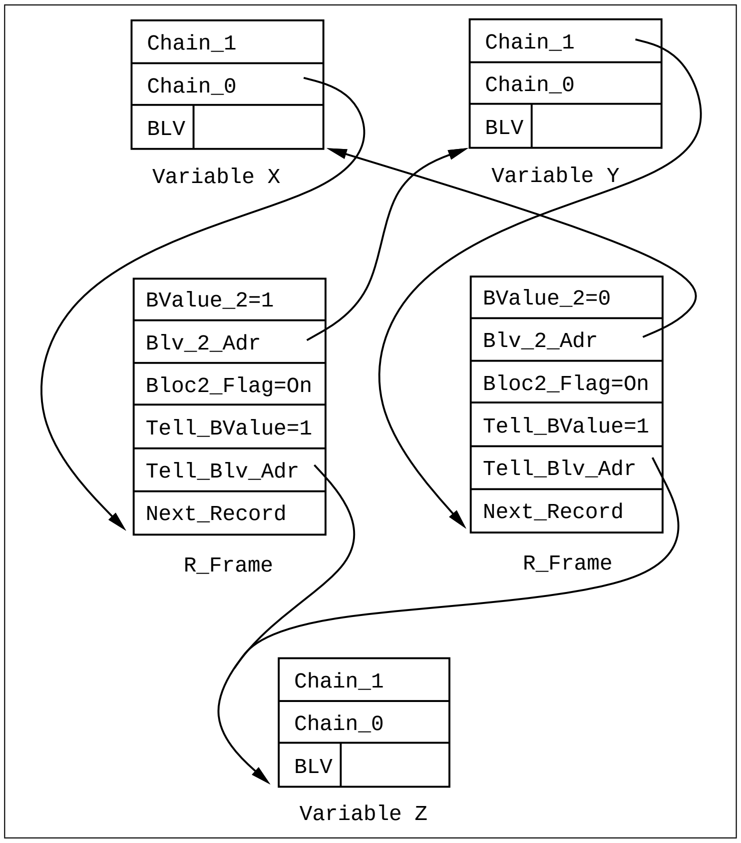

Fig. 1. Data structures involved in the constraint `Z<=[-X,Y]`

Install and telling instructions (defined in the case $n = 1$ or $n = 2$) are:

`b_install_and_tell_cstr1(X0,bvalue0,X1,bvalue1)`
two cases depending on `X1`:

- `X1` is an integer: if `X1`=`bvalue1`, (`X0`, `bvalue0`) is enqueued and the consistency procedure called (else the constraint succeeds immediately as the premise is false).
- `X1` is a boolean variable: an R_Frame (created on the top of the heap) is added to the appropriate list of `X1` recording `X0` and `bvalue0`.

`b_install_and_tell_cstr2(X0,bvalue0,X1,bvalue1,X2,bvalue2)`
three cases depending on `X1` and `X2`:

- `X1` is an integer:
  it behaves like `b_install_and_tell_cstr1(X0,bvalue0,X2,bvalue2)`.
- `X2` is an integer:
  it behaves like `b_install_and_tell_cstr1(X0,bvalue0,X1,bvalue1)`.
- `X1` and `X2` are two boolean variables: an R_Frame (created on the top of the heap) is added to the appropriate list of `X1` recording `X0`, `bvalue0`

and `X2`, `bvalue2`, and similarly an R_Frame is added to the appropriate list of `X2` recording `X0`, `bvalue0` and `X1`, `bvalue1`.

It is worth noticing that only 4 instructions are needed to implement this boolean solver into the WAM. The extension is really minimal. Our experience has shown that in this way only a few days are necessary to integrate boolean constraints into a Prolog compiler whose sources are well-known.
Table 6 shows an example of code generated for the constraint `and(X,Y,Z)`.

```
and/3: b_load_x_value(0,0)                  X(0) = address of X
       b_load_x_value(1,1)                  X(1) = address of Y
       b_load_x_value(2,2)                  X(2) = address of Z
       b_install_and_tell_cstr2(2,1,0,1,1,1)  Z <= [X,Y]
       b_install_and_tell_cstr1(2,0,0,0)     -Z <= [-X]
       b_install_and_tell_cstr1(2,0,1,0)     -Z <= [-Y]
       b_install_and_tell_cstr1(0,1,2,1)      X <= [Z]
       b_install_and_tell_cstr2(0,0,1,1,2,0) -X <= [Y,-Z]
       b_install_and_tell_cstr1(1,1,2,1)      Y <= [Z]
       b_install_and_tell_cstr2(1,0,0,1,2,0) -Y <= [X,-Z]
       proceed                              Prolog return
```

Table 6. Code generated for `and(X,Y,Z)`

4.4 The consistency procedure

This procedure is responsible for ensuring the consistency of the store. It repeats the following steps until the propagation queue is empty (or until a failure occurs):
Let $(X, Bvalue)$ be the pair currently pointed by `BP`.

- If X is an integer, there are two possibilities:
  - $X = Bvalue$: success (*Check Ok*)
  - $X \neq Bvalue$: failure (*Fail*)
- else the boolean variable X is set to $Bvalue$ (*Reduce*) and each constraint depending on X (i.e each record of `Chain_`*Bvalue*) is reconsidered as follows:
  - case $n = 1$: the pair $(X_0, Bvalue_0)$ is enqueued.
  - case $n = 2$: let us suppose that $X = X_1$, the case $X = X_2$ being identical. The variable X_2 must be tested to detect if the constraint can be solved:
    - ∗ X_2 is an integer: if $X_2 = Bvalue_2$ then the pair $(X_0, Bvalue_0)$ is enqueued or else the constraint is already solved (*Solved*).
    - ∗ X_2 is a boolean variable: the constraint still suspends (*Suspend*).

Each constraint $(X, Bvalue)$ in the queue will be activates and can have one of the following issues:

- *Reduce*: the boolean variable X is set to the integer $Bvalue$,
- *Check Ok*: X already equals $Bvalue$,
- *Fail*: X is an integer different from $Bvalue$.

When a constraint $(X, Bvalue)$ has *Reduce* as issue, the propagation reconsider all constraints depending on X. Each such constraint will be enqueued in order to be activated (and taken into account by the above cases) or ignored (only if $n = 2$) due to:

- *Suspend*: the other variable of the constraint is not yet ground,
- *Solved*: the other variable is ground but does not correspond to the "sign" of its literal, i.e. the premise is false.

4.5 Optimizations

Obviously, *Check Ok* corresponds to a useless tell since it neither reduces the variable nor fails. As first pointed out in the design of `clp(FD)` we can avoid some of such tells [9]. However, in the simpler framework of `clp(B)`, empirical results show that there is no gain in terms of efficiency. Indeed, a useless tell only consists in a test between two integers and the detection of the possibility to avoid such a tell also involves a test between integers.

The *Solved* issue also corresponds to a useless work since the constraint is already entailed (see [5]).

Figure 2 makes it possible to estimate the proportion of each issue for some instances of our benchmarks.

5 Performances of `clp(B)`

5.1 The benchmarks

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

- `schur`: Schur's lemma. The problem consists in finding a 3-coloring of the integers $\{1 \ldots n\}$ such that there is no monochrome triplet (x, y, z) where $x + y = z$. The formulation uses $3 \times n$ variables to indicate, for each integer, its color. This problem has a solution iff $n \leq 13$.
- `pigeon`: the pigeon-hole problem consists in putting n pigeons in m pigeon-holes (at most 1 pigeon per hole). The boolean formulation uses $n \times m$ variables to indicate, for each pigeon, its hole number. Obviously, there is a solution iff $n \leq m$.

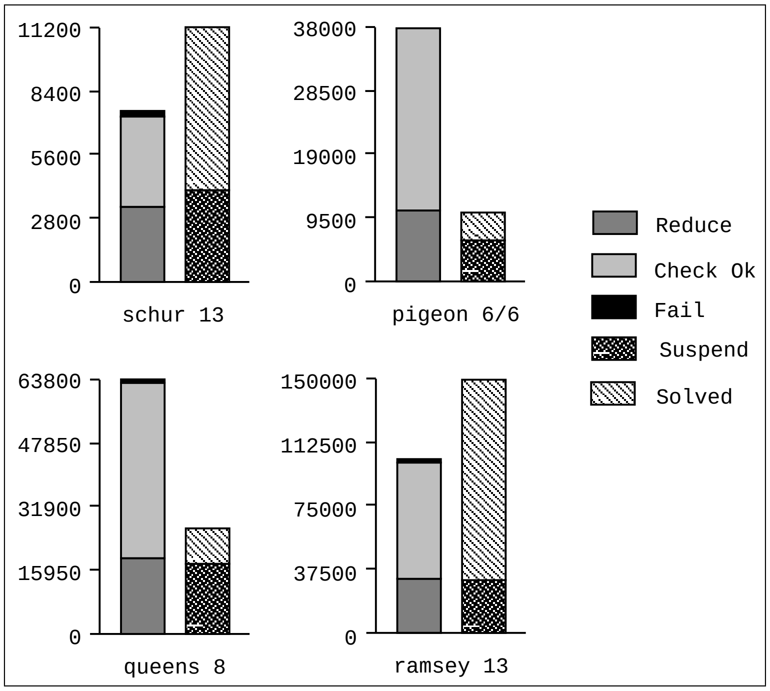

Fig. 2. Proportion of each issue of the consistency procedure

- `queens`: place n queens on a $n \times n$ chessboard such that there are no queens threatening each other. The boolean formulation uses $n \times n$ variables to indicate, for each square, if there is a queen on it.
- `ramsey`: find a 3-coloring of a complete graph with n vertices such that there is no monochrome triangles. The formulation uses 3 variables per edge to indicate its color. There is a solution iff $n \leq 16$.

All solutions are computed unless otherwise stated. The results presented below for `clp(B)` do not include any heuristics and have been measured on a Sun Sparc 2 (28.5 Mips).

5.2 Presentation of other boolean solvers

Here, we present the set of boolean solvers, based on different techniques, which will be used in the following section to evalute the performances of `clp(B)`. It is worth noticing that many of them are special-purpose boolean solvers, which are intended to take a set of boolean formulas as input while only `clp(B/FD)` and CHIP provide boolean solvers integrated into full CLP languages, offering therefore more flexibility. These solvers are described in more details in [6].

`clp(B/FD)` is a CLP language on booleans built on the top of `clp(FD)` [4]. Times for `clp(B/FD)` were also measured on a Sun Sparc 2. Exactly the same programs were run on both systems.

CHIP is also a CLP language (and not only a constraint solver) and thus accepts the same programs as `clp(B)`. Moreover, it also uses a boolean constraint solver based on finite domains[2]. Times for CHIP were also measured on a Sun Sparc 2. Exactly the same programs were run on both systems.

Adia is an efficient boolean constraint solver based on the use of BDDs [13]. Time measurements presented below are taken from [15] who tries four different heuristics on a Sun Sparc IPX (28.5 Mips). We have chosen the best of these four timings for Adia. 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 $n = 30$.

Enumeration method is presented in [14] who provides time measurements on a Sun 3/80 (1.5 Mips). We normalized these measurements by a factor of 1/19.

Boolean local consistency method is presented in [12] who presents results on a Macintosh SE/30 equivalent to a Sun 3/50 (1.5 Mips). We normalized them with a factor of 1/19. 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)` only uses a static order (standard labeling).

FAST93 is a boolean solver based on 0-1 programming techniques from Operational Research [3]. The time measurements are given for a Sparc Station 1+ (18 MIPS), and therefore normalized by a factor 1/1.6. It should be noted that on the benchmark problems, only the first solution is computed.

5.3 Comparison with other solvers

Table 7 shows the performances of `clp(B)` and the corresponding speedup w.r.t. all other solvers above presented except for FAST93 since it was on a different set of benchmarks (table 8 presents a comparison with FAST93). The sign "$ovflw$" means that the program exhausted available memory, the symbol "?" means that the timing was not available to us and the symbol ↓ before a number means in fact a slowdown of `clp(B)` by that factor. We can summarize the figures of table 7 and table 8 as follows:

[2] 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 limitations.

Program	clp(B) Time (s)	clp(B/FD) / clp(B)	CHIP / clp(B)	Bdd best / clp(B)	Enum / clp(B)	BCons / clp(B)
`schur 13`	0.040	2.50	20.57	27.75	20.25	1.75
`schur 14`	0.040	2.50	22.00	35.75	22.00	2.00
`schur 30`	0.100	2.50	93.70	*ovflw*	?	?
`schur 100`	0.620	1.89	322.83	*ovflw*	?	?
`pigeon 6/5`	0.020	2.50	15.00	3.00	2.00	6.50
`pigeon 6/6`	0.180	2.00	10.00	↓ 1.80	12.72	4.88
`pigeon 7/6`	0.110	2.81	15.45	1.00	7.63	7.90
`pigeon 7/7`	1.390	1.91	9.67	↓ 5.56	?	5.20
`pigeon 8/7`	0.790	2.81	16.12	↓ 2.54	?	8.63
`pigeon 8/8`	12.290	1.97	9.58	↓ 21.18	?	5.49
`queens 6`	0.040	1.50	?	25.25	1.75	?
`queens 7`	0.090	1.88	?	50.55	4.11	?
`queens 8`	0.230	2.34	19.17	233.73	6.26	7.86
`queens 9`	0.860	2.48	19.37	*ovflw*	8.02	9.01
`queens 10`	3.000	2.75	22.27	*ovflw*	?	10.90
`queens 14 1st`	0.500	1.74	12.56	*ovflw*	?	6.28
`queens 16 1st`	1.510	2.17	17.47	*ovflw*	?	11.89
`queens 18 1st`	4.450	2.35	20.27	*ovflw*	?	?
`queens 20 1st`	17.130	2.51	22.93	*ovflw*	?	?
`ramsey 12 1st`	0.130	1.46	10.53	*ovflw*	?	?
`ramsey 13 1st`	0.690	2.17	11.13	*ovflw*	?	?
`ramsey 14 1st`	1.060	2.28	31.30	*ovflw*	?	?
`ramsey 15 1st`	292.220	2.39	32.10	*ovflw*	?	?
`ramsey 16 1st`	721.640	2.52	44.17	*ovflw*	?	?

Table 7. `clp(B)` vs all other solvers

`clp(B/FD)` Basically, `clp(B)` is about twice as fast as `clp(B/FD)` on average. This factor varies only slightly between 1.5 and 2.5 (depending on the problem), showing that the two systems perform the same pruning. `clp(B)` achieves better performances because of its more simple data-structures and internal computations.

CHIP The average speedup of `clp(B)` w.r.t. CHIP is around a factor of 20, with peak speedup reaching more than two orders of magnitude. It can be noted that on the `schur` and `ramsey` benchmarks the speedup of `clp(B)` grows up as the size of the problem grows. This is certainly due to the fact that the management of constraints is more complex in CHIP.

Adia The average speedup of `clp(B)` w.r.t. Adia is around a factor 20. It is however worth noticing that for big problems like `schur` ($n \geq 30$), `queens` ($n \geq 8$) or `ramsey` Adia overflows due to memory limits. This is intrinsic to the BDD approach and not only to the Adia implementation. However, Adia is slightly faster on the `pigeon` example because BDDs make it possible to factorize the very large number of solutions. Also note that we have chosen the timmings of Adia corresponding to the best heuristics for each benchmark (which is not always the same [15]); with the worst heuristics performances of Adia decrease greatly and for instance it is also slower on the `pigeon` problem.

Enumeration method The average speedup is around an order of magnitude. The speedup however differs substantially on the different bechmarks and even within the same benchmark with different problem sizes showing that the two solvers do not perform the same search space reduction.

Boolean local consistency method The average speedup is around a factor 6. An interesting point is that the factors are quite constant within a class of problem. 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.

Program	FAST93 Time (s)	`clp(B)` Time (s)	$\frac{\text{FAST93}}{\texttt{clp(B)}}$
`pigeon 7/7 1st`	0.250	0.010	25.00
`pigeon 8/7 1st`	1.940	0.790	2.45
`pigeon 8/8 1st`	0.630	0.020	31.50
`pigeon 9/8 1st`	4.230	6.840	↓ 1.61
`pigeon 9/9 1st`	0.690	0.030	23.00
`ramsey 10 1st`	11.500	0.070	164.28
`ramsey 12 1st`	81.440	0.130	626.46

Table 8. `clp(B)` vs an Operational Research method

FAST93 We have the timmings for `pigeon` and `ramsey` problems only and for the computation of the first solution only since this method cannot compute all solutions (see table 8). On the `pigeon` problem, `clp(B)` is about 30 times faster than FAST93 when the problem is satisfiable (less pigeons than holes) whereas FAST93 discovers inconsistencies more quickly when the problem does not admit any solution because of the underlying Operational Research techniques. For instance on `pigeon 9/8` FAST93 is 1.6 times faster than `clp(B)`. This method does not seem adequate to solve the `ramsey` problem since it is 160 times slower than `clp(B)` for `ramsey 10` and more than 600 times slower for `ramsey 12`. The speedup of `clp(B)` seems to grow exponentially, showing again that the two solving techniques perform very different search space reduction.

6 Conclusion and perspective

We have presented a boolean constraint solver in the CLP paradigm wich is both very simple and efficient. It is based on the glass-box approach, and only uses a single low-level constraint into which boolean constraints such as *and*, *or* or *not* are decomposed. The main idea was to simplify the data-structures used for full finite domain constraints in `clp(FD)` that were used in the `clp(B/FD)`. These simplifications improve the performances by more than a factor two giving a solver which is around an order of magnitude faster than other existing boolean solvers. The low-level primitive constraint at the core of `clp(B)` can be implemented into a WAM-based logical engine with a minimal extension : only four new abstract instructions are needed. Therefore this very simple but very efficient boolean constraint solver can be incomporated into any Prolog system very easily. We are currently investigating the use of `clp(B)` for real-life applications such as fault diagnosis. We will also investigate the integration of flexible primitives to allow the user to define complex labeling heuristics.

References

1. H. Aït-Kaci. *Warren's Abstract Machine, A Tutorial Reconstruction.* Logic Programming Series, MIT Press, Cambridge, MA, 1991.
2. H. Bennaceur and G. Plateau. FASTLI: An Exact Algorithm for the Constraint Satisfaction Problem: Application to Logical Inference. Research Report, LIPN, Universite Paris-Nord, Paris, France, 1991.
3. H. Bennaceur and G. Plateau. Logical Inference Problem in Variables 0/1. in *IFORS 93 Conference*, Lisboa, Portugal, 1993.
4. P. Codognet and D. Diaz. Boolean Constraint Solving Using `clp(FD)`. In *International Logic Programming Symposium*, Vancouver, British Columbia, Canada, MIT Press 1993.
5. P. Codognet and D. Diaz. Compiling Constraint in `clp(FD)`. draft, 1993.
6. P. Codognet and D. Diaz. Local Propagation Methods for Solving Boolean Constraints in Constraint Logic Programming. draft, 1993.
7. A. Colmerauer. An introduction to PrologIII. *Communications of the ACM*, no. 28 (4), 1990, pp 412-418.
8. M. M. Corsini and A. Rauzy. CLP($\mathcal{B}$): Do it yourself. In *GULP'93, Italian Conference on Logic Programming*, Gizzeria Lido, Italy, 1993.
9. D. Diaz and P. Codognet. A Minimal Extension of the WAM for `clp(FD)`. In *10th International Conference on Logic Programming*, Budapest, Hungary, The MIT Press 1993.
10. J. Jaffar and J-L. Lassez. Constraint Logic Programming. In *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. 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.
13. A. Rauzy. Adia. Technical report, LaBRI, Université Bordeaux I, 1991.

14. A. Rauzy. Using Enumerative Methods for Boolean Unification. In *Constraint Logic Programming: Selected Research*, A. Colmerauer and F. Benhamou (Eds.). The MIT Press, 1993.
15. A. Rauzy. Some Practical Results on the SAT Problem, Draft, 1993.
16. V. Saraswat. The Category of Constraint Systems is Cartesian-Closed. In *Logic In Computer Science*, IEEE Press 1992.
17. D. S. Scott Domains for Denotational Semantics. In *ICALP'82, International Colloquium on Automata Languages and Programming*, 1982.
18. T. E. Uribe and M. E. Stickel Ordered Binary Decision Diagrams and the Davis-Putnam Procedure. In *1st International Conference on Constraints in Computational Logics*, Munich, Germany, 1994.
19. P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. Logic Programming Series, The MIT Press, Cambridge, MA, 1989.
20. P. Van Hentenryck, V. Saraswat and Y. Deville. Constraint Processing in cc(FD). Draft, 1991.
21. P. Van Hentenryck, H. Simonis and M. Dincbas. Constraint Satisfaction Using Constraint Logic Programming. *Artificial Intelligence* no 58, pp 113-159, 1992.
22. D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI International, Oct. 1983.