The Boolean Constraint Solver of SWI-Prolog: System Description

Markus Triska

Database and Artificial Intelligence Group
Vienna University of Technology
triska@dbai.tuwien.ac.at
https://www.metalevel.at

Abstract. We present a new constraint solver over Boolean variables, available as `library(clpb)`[1] in SWI-Prolog. Our solver distinguishes itself from other available CLP($\mathcal{B}$) solvers by several unique features: First, it is written entirely in Prolog and is hence portable to different Prolog implementations. Second, it is the first freely available BDD-based CLP($\mathcal{B}$) solver. Third, we show that new interface predicates allow us to solve new types of problems with CLP($\mathcal{B}$) constraints. We also use our implementation experience to contrast features and state necessary requirements of attributed variable interfaces to optimally support CLP($\mathcal{B}$) constraints in different Prolog systems. Finally, we also present some performance results and comparisons with SICStus Prolog.

Keywords: CLP(B), Boolean unification, Decision Diagrams, BDD

1 Introduction

CLP($\mathcal{B}$), Constraint Logic Programming over Boolean variables, is a declarative formalism for reasoning about propositional formulas. It is an important instance of the general CLP($\cdot$) scheme introduced by Jaffar and Lassez [11] that extends logic programming with reasoning over specialized domains. Well-known applications of CLP($\mathcal{B}$) arise in circuit verification and model checking tasks.

There is a vast literature on SAT solving, and there are many systems and techniques for detecting (un)satisfiability of Boolean clauses (see [18], [14], [22] and many others).

However, a CLP($\mathcal{B}$) system is different from common SAT solvers in at least one critical aspect: It must support and take into account *aliasing* and unification of logical variables, even *after* SAT constraints have already been posted. Generally, CLP($\mathcal{B}$) systems are more algebraically oriented than common SAT solvers: In addition to unification of logical variables, they also support variable quantification, conditional answers and easy symbolic manipulation of formulas. In this paper, we discuss several use cases and consequences of these features.

This paper is organized as follows: In Section 2, we briefly outline the current state of available CLP($\mathcal{B}$) systems, followed by a brief discussion of Binary

[1] Documentation: `http://eu.swi-prolog.org/man/clpb.html`

The final publication is available at Springer via:
http://dx.doi.org/10.1007/978-3-319-29604-3_4

Decision Diagrams. In Section 4, we present the interface and implementation of a new CLP($\mathcal{B}$) system, its distinguishing new features, a comparison of attributed variable interfaces and necessary requirements for optimally supporting CLP($\mathcal{B}$) solvers on top of Prolog. Section 5 describes new applications made possible by the new features of our library, followed by performance results and a brief discussion of implementation variants and planned features.

2 Current CLP($\mathcal{B}$) systems and implementation methods

Support of CLP($\mathcal{B}$) constraints has been somewhat inconsistent between and even within different Prolog systems over the last few decades. CHIP [9] was one of the first widely used systems to support CLP($\mathcal{B}$) constraints, and shortly after, SICStus Prolog supported them too [4], up until version 3. However, more recent versions of SICStus Prolog, while shipping with a port of the `clpb` library, do not officially support the solver in any way.[2] In contrast, Prolog IV [1] and GNU Prolog [8] do support Boolean constraints.

Implementation methods of CLP($\mathcal{B}$) systems are likewise diverse. We find two main implementation variants used in major Prolog systems: (1) implementations based on Binary Decision Diagrams (BDDs) and (2) approximation of CLP($\mathcal{B}$) constraints by other constraints, using for example indexicals. SICStus Prolog is an instance of the former variant, and GNU Prolog one of the latter.

Each of these variants has strengths and weaknesses: Among the major advantages of BDD-based implementations we find *completeness* and some algebraic virtues which we will explain in later sections of this paper. In comparison, approximation-based implementations are generally simpler, more scalable and much more efficient on selected benchmarks [5]. However, they are *incomplete* in general and require an explicit search to ensure the existence of solutions after posting constraints.

3 Binary Decision Diagrams (BDDs)

A Binary Decision Diagram (BDD) is a rooted, directed and acyclic graph and represents a Boolean function [2,12]. In this paper, we assume all BDDs to be *ordered* and *reduced*. This means, respectively, that all variables appear in the same order on all paths from the root, and that the representation is minimal in the sense that all isomorphic subgraphs are merged and no redundant nodes occur.

In the Prolog community, BDDs have already appeared several times: Apart from the CLP($\mathcal{B}$) library used in SICStus Prolog, we also find BDDs in the form of small Prolog code snippets. For example, Richard O'Keefe has generously made a small library available for his COSC410 course in the year 2011.[3]

[2] The documentation of SICStus Prolog 4.3.2 contains the exact wording of current support terms of the `clpb` module that ships with the system: "The library module is a direct port from SICStus Prolog 3. It is not supported by SICS in any way."

[3] Source: `http://www.cs.otago.ac.nz/staffpriv/ok/COSC410/robdd.pl`

BDDs also occur in publications that introduce or use closely related data structures [20,19]. Within the logic programming community, important applications of BDDs arise in the context of *probabilistic* logic programming [13] and termination analysis of Prolog programs [3,6].

4 A new CLP($\mathcal{B}$) system: `library(clpb)` in SWI-Prolog

We have implemented a new CLP($\mathcal{B}$) system, freely available in SWI-Prolog [21] as `library(clpb)`. In this section, we present the design choices, interface predicates and implementation. Subsections 4.5, 4.6 and 4.7 are targeted at implementors and contributors of Prolog systems and constraint libraries, and assume familiarity with BDDs and Prolog interfaces for attributed variables.

4.1 Implementation choices: BDDs, SAT solvers, external libraries

Before presenting the actual features and implementation of our new system, we present a brief high-level overview of the various implementation options and their consequences, and give several reasons that justify the choices we have made in our implementation.

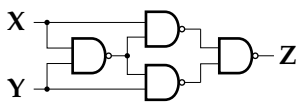
When implementing a new CLP($\mathcal{B}$) system, we typically have a clear idea of what we need from it. Also in our case, the intended use was very clear from the start: Since 2004, the author has been working on facilitating a port of Ulrich Neumerkel's GUPU system [16,17] to SWI-Prolog so that more users can freely benefit from it. GUPU is an excellent Prolog teaching environment, and one of its integrated termination analyzers, cTI [3], heavily depends on the CLP($\mathcal{B}$) implementation of SICStus Prolog. Already a cursory glance at the source code of cTI makes clear that it depends on features that only a BDD-based solver can provide, since cTI goes as far as inspecting the concrete structure of BDDs in its implementation.

Still, we initially hoped for a shortcut: Our hope was that we could simulate the behaviour of a BDD-based CLP($\mathcal{B}$) system by using a simpler (external or internal) SAT solver. For example, we envisioned that checking for tautologies could be easily handled by looking for counterexamples of the accumulated constraints, and checking consistency of accumulated constraints could be handled by trying to generate concrete solutions after posting each constraint.

Alas, such a simplistic approach falls short for several reasons. One of those reasons is efficiency: For example, detecting tautologies (a prominent operation in cTI) is hard in general, but easy after BDDs have been built. Another, more fundamental reason is that many use cases of CLP($\mathcal{B}$) depend on *symbolic* results instead of "only" detecting satisfiability, and such results are much more readily obtained with BDD-based approaches.

As a simple example, consider the integrated circuit shown in Fig. 1 (a). A Prolog program that describes the circuit with CLP($\mathcal{B}$) constraints (see Section 4.2) is shown in Fig. 1 (b). No concrete solutions are asked for by that program: To verify the circuit, we care more about the *symbolic expressions*

that are obtained as residuals goals of this program, and *less* about concrete solutions. For example, with the given program, the query `?- xor(x, y, Z).` yields the residual goal `sat(Z=:=x#y)`, expressing `Z` as a function of the intended input variables, which are universally quantified. From this, we see at one glance that the circuit indeed describes the intended Boolean XOR operation. When producing residual goals, existential quantification is implicitly used by the Prolog toplevel to project away variables that do not occur in the query.



(a)

```
:- use_module(library(clpb)).

nand_gate(X, Y, Z) :- sat(Z =:= ~(X*Y)).

xor(X, Y, Z) :-
        nand_gate(X, Y, T1),
        nand_gate(X, T1, T2),
        nand_gate(Y, T1, T3),
        nand_gate(T2, T3, Z).
```

(b)

Fig. 1: (a) Expressing XOR ($X \oplus Y = Z$) with four NAND gates and (b) describing the circuit with CLP($\mathcal{B}$) constraints. `?- xor(x, y, Z).` yields `sat(Z=:=x#y)`.

To efficiently provide such features and others (see also Section 4.4), we decided to base our implementation on BDDs instead of only emulating them.

Having made the decision to implement a BDD-based CLP($\mathcal{B}$) system, the next arising question was how to actually use BDDs so that they work in the context of CLP($\mathcal{B}$). Even though the excellent implementation description of an existing BDD-based CLP($\mathcal{B}$) system [4] was of course available to us, many unsettled questions still remained, such as: How is an existing BDD changed after unification of two variables? How do we handle unification of two variables that reside in different BDDs? How exactly does the notion of *universally* quantified variables affect all operations on BDDs? How are residual goals produced? Finally, are there not some subtly misguiding mistakes in the implementation description, e.g., is a BDD really represented by a *ground* Prolog term in SICStus Prolog, or are there not variables also involved?

In the face of so many initially unsettled questions, we anticipated a lot of prototyping and rewriting in the initial phase of our implementation, which also turned out to be necessary. To facilitate prototyping, enhance portability, and to study and answer high-level semantic questions separated from lower-level issues, we are consciously *not* hard-wiring our solver with an external BDD package until semantic aspects (see Section 4.7) are settled to provide a more stable basis for low-level changes. Therefore, we have created a new high-level Prolog implementation of BDDs that forms the basis of our new CLP($\mathcal{B}$) system.

We consider the availability of a completely free CLP($\mathcal{B}$) system where the above questions are answered in the form of an executable specification an integral part of our contribution, since it also shows the places where, if at all, external BDD libraries can be most meaningfully plugged in.

4.2 Syntax of Boolean expressions

We have strived for compatibility with SICStus Prolog and provide the same syntax of Boolean *expressions*. Table 1 shows the syntax of all Boolean expressions that are available in both SICStus and SWI-Prolog. Universally quantified variables are denoted by Prolog *atoms* in both systems, and universal quantifiers appear implicitly in front of the entire expression. Atoms are useful for denoting *input* variables: In residual goals, intended output variables are expressed as functions of input variables. The expression `card(Is,Exprs)` is true iff the number of true expressions in the list `Exprs` is a member of the list `Is` of integers and integer ranges of the form `From-To`.

In addition to the Boolean expressions shown in Table 1, we have also chosen to support two new Boolean expressions. These new expressions are shown in Table 2. They denote, respectively, the disjunction and conjunction of all Boolean expressions in a list. We have found this syntax extension to be very useful in many practical applications, and encourage their support in other CLP($\mathcal{B}$) systems. This syntax was kindly suggested to us by Gernot Salzer.

expression	meaning
`0`, `1`	**false**, **true**
variable	unknown truth value
atom	universally quantified variable
`~` *Expr*	logical NOT
Expr `+` *Expr*	logical OR
Expr `*` *Expr*	logical AND
Expr `#` *Expr*	exclusive OR
Var `^` *Expr*	existential quantification
Expr `=:=` *Expr*	equality
Expr `=\=` *Expr*	disequality (same as `#`)
Expr `=<` *Expr*	less or equal (implication)
Expr `>=` *Expr*	greater or equal
Expr `<` *Expr*	less than
Expr `>` *Expr*	greater than
`card(Is,Exprs)`	*see description in text*

Table 1: Syntax of Boolean expressions available in both SICStus and SWI

expression	meaning
`+(Exprs)`	disjunction of list `Exprs` of expressions
`*(Exprs)`	conjunction of list `Exprs` of expressions

Table 2: New and useful Boolean expressions in SWI-Prolog

4.3 Interface predicates of `library(clpb)`

Regarding interface predicates of our system, we have again strived primarily for compatibility with SICStus Prolog, and all CLP($\mathcal{B}$) predicates provided by SICStus Prolog are also available in SWI-Prolog with the same semantics. In particular, the interface predicates available in both systems are:

`sat(+Expr)`: True iff the Boolean expression `Expr` is satisfiable.

`taut(+Expr, -T)`: Succeeds with `T=0` if `Expr` cannot be satisified, and with `T=1` if `T` is a tautology with respect to the stated constraints. Otherwise, it fails.

`labeling(+Vs)` Assigns a Boolean value to each variable in the list `Vs` in such a way that all stated constraints are satisfied.

4.4 New interface predicates

BDDs have many important virtues that can be easily made available in a BDD-based CLP($\mathcal{B}$) system. The core idea of efficient algorithms on BDDs is often to combine the solutions for the two children of every BDD node in order to obtain a solution for the parent node.

In addition to the interface predicates presented in the previous section, we have implemented three new predicates that are not yet available in SICStus Prolog:

`sat_count(+Expr, -N)`: `N` is the number of different assignments of truth values to the variables in the Boolean expression `Expr`, such that `Expr` is true and all posted constraints are satisfiable.

`random_labeling(+Seed, +Vs)`: Assigns a Boolean value to each variable in the list `Vs` in such a way that all stated constraints are satisfied, and each solution is *equally likely*, using random seed `Seed` and committing to the first solution.

`weighted_maximum(+Weights, +Vs, -Maximum)`: Assigns 0 and 1 to the variables in `Vs` such that all stated constraints are satisfied, and `Maximum` is the maximum of $\sum w_i v_i$ over all admissible assignments. On backtracking, all admissible assignments that attain the optimum are generated.

As we show in the following section, these predicates are of great value in many applications, and we encourage their support in other CLP($\mathcal{B}$) systems based on BDDs. This is because these predicates are very easy to implement with BDD-based approaches, and omitting them deprives users of these benefits, unnecessarily.

Using the new `+/1` syntax to express the disjunction of Boolean expressions in a list, we also suggest the new idiom `sat_count(+[1|Vs], N)` to count the number of assignments of truth values to variables in `Vs` that satisfy all constraints that are posted so far, without further constraining the set of solutions.

4.5 Implementation

We briefly outline the underlying ideas of our implementation. Perhaps most strikingly, our library is written *entirely* in Prolog. This is a deliberate design decision, facilitating rapid prototyping and portability. To the best of our knowledge, ours is the first BDD-based CLP($\mathcal{B}$) system that is freely available. Our library comprises about 1,700 LOC, including documentation and comments.

Internally, we are using the following representation, using *attributed variables* as in hProlog [7]: Each CLP($\mathcal{B}$) variable belongs to exactly one BDD. Each CLP($\mathcal{B}$) variable gets an attribute of the form `index_root(Index,Root)`, where `Index` is the variable's unique integer index, and `Root` is the root of the BDD that the variable belongs to.

Each CLP($\mathcal{B}$) variable is also equipped with an association table that helps us keep the BDD reduced. The association table of each variable must be rebuilt on occasion to remove nodes that are no longer reachable. We rebuild the association tables of involved variables after BDDs are merged to build a new root. This only serves to reclaim memory: Keeping a node in a local table even when it no longer occurs in any BDD does not affect the solver's correctness.

A *root* is a logical variable with a single attribute, a pair of the form `Sat-BDD`, where `Sat` is the Boolean expression (in original form) that corresponds to `BDD`. `Sat` is necessary to rebuild the BDD after variable aliasing, and to project all remaining constraints to a list of `sat/1` goals.

Finally, a *BDD* is either: (1) The integers **0** or **1**, denoting **false** and **true**, respectively, or (2) a node of the form `node(ID,Var,Low,High,Aux)`, where:

- `ID` is the node's unique integer ID
- `Var` is the node's branching variable
- `Low` and `High` are the node's low (`Var` = 0) and high (`Var` = 1) children
- `Aux` is a free variable, one for each node, that can be used to attach attributes and store intermediate results.

This representation means that we are using (assuming SWI-Prolog and machine-sized integers) 48 bytes per node on 64-bit systems, and we need to store this roughly *twice* because each node is also represented in the association table of its branching variable.

In addition to this considerable memory overhead, our choice to use association tables incurs a logarithmic runtime overhead compared to hashing. On the plus side, association tables scale very predictably and do not require any *ad hoc* considerations and complex treatment of edge-cases.

Fig. 2 shows an essential predicate of our library: It is called `make_node/4`, and given a branching variable and its two children, it builds (`low_high_key/3`) a unique `Key` and, depending on whether such a node already exists, either yields that node, or builds a new node. A unique ID is generated for each new node by incrementing a global backtrackable variable called `$clpb_next_node`. The predicates `lookup_node/3` and `register_node/3` (implementation omitted) access the branching variable's association table to fetch or store a node. In

```
make_node(Var, Low, High, Node) :-
        (   Low == High -> Node = Low
        ;   low_high_key(Low, High, Key),
            (   lookup_node(Var, Key, Node) -> true
            ;   clpb_next_id('$clpb_next_node', ID),
                Node = node(ID,Var,Low,High,_Aux),
                register_node(Var, Key, Node)
            )
        ).
```

Fig. 2: `make_node/4`, the essential predicate for creating a BDD node

addition, if the two children are identical, then the resulting node is simply that child itself. Thus, `make_node/4` automatically keeps the BDD reduced.

Many of the implemented algorithms use *memoization* to store intermediate results for later use. We are using DCGs and semicontext[4] notation in several internal predicates to implicitly thread through stored results. We refer interested readers to the source code of our library for the fully detailed picture of the implementation.

4.6 Consistency notions in the context of CLP($\mathcal{B}$)

Completeness of our CLP($\mathcal{B}$) system follows from the well-known fact that, for fixed variable order and function, the corresponding BDD is *canonical*. Hence, as long as all BDDs that represent the posted constraints are different from **0**, there is at least one admissible solution.

In addition, the well-known CLP(FD) notion of *domain consistency* is of course equally applicable to CLP($\mathcal{B}$): For example, when posting the constraint `sat(X*Y + ~X*Y)`, then a domain consistent CLP($\mathcal{B}$) solver must yield the unification `Y = 1`. We implement domain consistency in our CLP($\mathcal{B}$) system, and, although this is not documented and does not directly follow from its implementation description, `library(clpb)` in SICStus Prolog seems to implement this as well.

In fact, SICStus Prolog goes even beyond domain consistency, and seems to implement an undocumented additional property that, for lack of an established terminology (see also [10]), we shall call *aliasing* consistency. By this, we mean that if `taut(X =:= Y, 1)` holds for any two variables `X` and `Y`, then `X = Y` is posted. For example, when posting `sat((A#B)*(A#C))`, then an aliasing consistent CLP($\mathcal{B}$) solver must yield the unification `B = C`.

We implement both consistency notions as follows: First, in a single global sweep of the BDD, we collect all variables that are not skipped in any branch of the BDD that leads to **1**. It is easy to see that if a variable is skipped in a branch that leads to **1**, then it can assume both possible truth values, and cannot be involved in any aliasing. The collected variables are further classified into (1) variables that allow only a single truth value, (2) *further-branching* variables

[4] A Prolog DCG primer is available at `https://www.metalevel.at/prolog/dcg`

(i.e., variables that do not have **1** as any child in any node) and (3) *negative-decisive* variables (i.e., variables that have **0** as one child in all nodes). It is easy to see that any potential aliasing must involve one further-branching and one negative-decisive variable, and in additional partial sweeps of the BDD, we determine all unifications that hold among the collected variables.

We have tested the impact of enabling domain and aliasing consistency on a range of benchmarks, and generally found the impact to be very acceptable and sometimes even improving the running time. For this reason, we have opted to enable both consistency notions and benefit from their algebraic properties.

4.7 Unification of attributed variables

At the time of this writing, there is no consensus across different Prolog systems regarding the interface predicates for attributed variables. Two different interfaces used by major implementations are, respectively, the one used by SICStus Prolog, and the one used by hProlog and SWI-Prolog. The most striking difference between these two interfaces (see [7]) is that in SICStus Prolog, unifications are *undone* before `verify_attributes/3` is called, whereas for example in SWI-Prolog, `attr_unify_hook/2` is called with the unification already in place.

Using our implementation experience, we strongly endorse the SICStus interface and its greater generality. We justify this with three different arguments:

(1) The interface used in SWI-Prolog is not general enough to express what we need. For example, according to the documentation of SICStus Prolog, the unification `P = Q` of two CLP($\mathcal{B}$) expressions `P` and `Q` is equivalent to posting `sat(P =:= Q)`. In SWI-Prolog, we cannot fully implement this semantics, because at the time the unification hook is called, the unification has already taken place and may have created a *cyclic* term instead of retaining variables.

(2) The interface used in SWI-Prolog makes it extremely hard to reason about simultaneous unifications. Critically, two variables may be instantiated simultaneously, using for example `[X,Y] = [0,1]`. This may not pose any problem when admissible unifications can be determined from ground values alone, but it is a severe limitation when additional structures such as decision diagrams, typically stored in attributes, are required. This is because when the unification hook is called for `X`, then `Y` is *no longer a variable* and its previous attributes cannot be directly accessed.

(3) The interface used in SWI-Prolog makes reasoning about unifications extremely error-prone. For example, when unifying two CLP($\mathcal{B}$) variables, the unification hook is called with the two variables already *aliased* and in fact identical. In our experience, failure to take possible aliasings into account is a common mistake when working with the SWI interface, and it would improve ease of use considerably if, as in the SICStus interface, unifications were *undone* before the unification hook is invoked by SWI-Prolog.

It is clear that the SICStus interface has some performance impact, because unifications have to be *undone*. In our view, this small disadvantage is completely negligible when taking into account the increased generality and ease of use of the SICStus interface.

5 New applications of `library(clpb)`

In this section, we present new applications of our CLP($\mathcal{B}$) system to illustrate the value of the new interface predicates that we provide. Importantly, these applications all rely *exclusively* on the CLP($\mathcal{B}$) interface predicates that are explained in the previous section. In other words, they do not use any low-level primitives that directly manipulate a BDD. Instead, everything is expressed as `sat/1` constraints, and the new interface predicates are used to count solutions and select solutions etc. Similar functionality is also available in many BDD packages. However, a CLP($\mathcal{B}$) system is much more convenient to use than a low-level library, and different formulations of the same problem can be tried more easily.

5.1 Counting solutions

We now apply the new interface predicates of our CLP($\mathcal{B}$) solver to solve a problem that asks for the number of solutions. It is one of the problems posed in the well known set of challenging mathematical tasks called *Project Euler*.[5] Specifically, it is:

Project Euler Problem 172: How many 18-digit numbers n (without leading zeros) are there such that no digit occurs more than three times in n?

One way to solve this problem is to find a recursive formula that breaks the problems into smaller parts, and to use memoization to make computed intermediate results quickly available for later reference.

However, in our experience, such a way to solve this problem is comparatively tedious and error-prone: It is easy to overlook a case, or to accidentally count some combinations multiple times. Thus, it is hard to be absolutely certain about the correctness of a recursive formula in such cases.

In contrast, the problem has a completely straight-forward and short formulation using CLP($\mathcal{B}$) constraints: We can use Boolean variables v_i ($0 \leq i \leq 9$) to represent a single digit d, where $v_i = 1$ indicates that $d = i$. This method naturally scales to multiple digits by using further sets of variables for subsequent digits. Fig. 3 shows how to indicate the number 2016 in this way, where each row corresponds to one digit.

$$\begin{pmatrix} 0&0&1&0&0&0&0&0&0&0 \\ 1&0&0&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&1&0&0&0 \end{pmatrix}$$

Fig. 3: Representing 2016 with a Boolean 4×10 matrix, using one row per digit

[5] See `http://projecteuler.net` for more information.

Obviously, exactly one of v_i must be 1 to specify the unique value of a single digit. Thus, if v_i are represented as a Prolog list `Ls` of 10 Boolean variables, we state `sat(card([1],Ls))` in terms of CLP($\mathcal{B}$).

The main constraint of the puzzle is readily expressed over each *column* of the resulting matrix, using `sat(card([0,1,2,3],Ls))` to constrain each digit to occur at most 3 times.

Fig. 4 shows the complete Prolog code to solve the problem with CLP($\mathcal{B}$). The query `euler_172(N)` yields `N = 227485267000992000` after about 7 hours of computation time, using an Intel Core i7 CPU (2.67 GHz) and about 20 GB RAM with SWI 7.3.7.

```
:- use_module(library(clpb)).
:- use_module(library(clpfd)).

euler_172(N) :-
        findall(Ds, (between(1,18,_),length(Ds,10)), Digits),
        Digits = [[0|_]|_],  % no leading zero
        transpose(Digits, DigitsT),
        maplist(card([0,1,2,3]), DigitsT),
        maplist(card([1]), Digits),
        append(Digits, Vs),
        sat_count(+[1|Vs], N).

card(C, Ls) :- sat(card(C, Ls)).
```

Fig. 4: Using CLP($\mathcal{B}$) constraints to solve Project Euler Problem 172

In contrast to the recursive version and complex combinatorial considerations, there is hardly any room for errors with such a simple CLP($\mathcal{B}$) model.

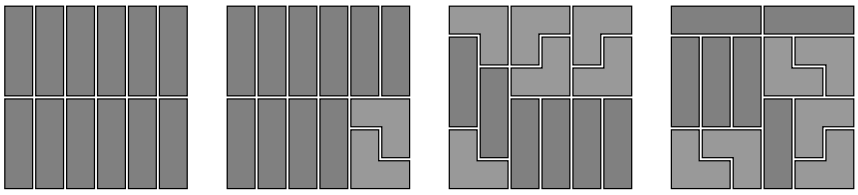
5.2 Random solutions

In our second example, we apply CLP($\mathcal{B}$) constraints to model an *exact cover* problem. The task is to cover an $N \times N$ chessboard with *triominoes*, which are rookwise connected pieces with three cells.

We use the following CLP($\mathcal{B}$) encoding: Each cell of the chessboard corresponds to a column of a matrix (b_{ij}), and each possible placement of a single triomino corresponds to one row. $b_{ij} = 1$ means that placing a triomino according to row i covers cell j. For each row, we introduce a Boolean variable x_k, where $x_k = 1$ means that we choose to place a triomino according to row i. An *exact cover* of the chessboard means that for each set S_l of Boolean variables, $S_l = \{x_k \mid b_{kl} = 1\}$, exactly *one* of the variables in S_l is equal to 1, i.e., `sat(card([1],`$list(S_l)$`))` holds, with $list(S_l)$ denoting a Prolog list corresponding to S_l.

In Fig. 5, subfigures (a) and (b) illustrate a common phenomenon when using CLP(FD) constraints to solve such tasks: Successive solutions are often very much alike. Simply adding randomization to `labeling/2` is in general *not* sufficient to guarantee random solutions due to potential clustering of

solutions. Subfigures (c) and (d) illustrate that solutions can be selected with uniform probability with CLP($\mathcal{B}$) constraints, using the new interface predicate `random_labeling/2`.



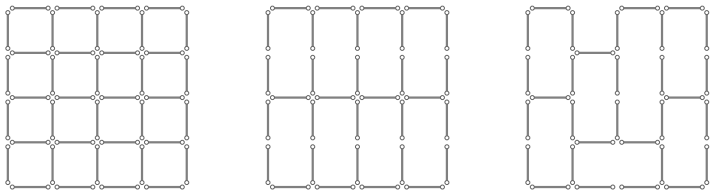
(a) (b) (c) (d)

Fig. 5: Exact covers of a 6×6 chessboard. (a) and (b) are successive solutions found with CLP(FD) constraints. (c) and (d) are found with CLP($\mathcal{B}$), using random seeds 0 and 1, respectively.

5.3 Weighted solutions

In the third example, we use the new interface predicate `weighted_maximum/3` to *maximize* the number of Boolean variables that are **true**.

The example we use to illustrate this concept is a simple matchsticks puzzle. The initial configuration is shown in Fig. 6 (a), and the task is to keep as many matchsticks as possible in place while at the same time letting no subsquares remain. For example, in Fig. 6 (b), exactly 7 subsquares remain, including the 4×4 outer square. Fig. 6 (c) shows an admissible solution of this task.



(a) (b) (c)

Fig. 6: (a) A grid of matchsticks, (b) exactly 7 subsquares remaining and (c) removing the minimum number of matchsticks so that no subsquares remain

Such puzzles are readily formulated with CLP($\mathcal{B}$) constraints, using one Boolean variable to indicate whether or not a matchstick is placed at a particular position. Our new interface predicates make it easy to find and count solutions, and also to maximize or minimize the number of used matchsticks.

CLP($\mathcal{B}$) constraints are not limited to very small puzzles and toy examples though: For tasks of suitable structure, CLP($\mathcal{B}$) constraints scale quite well and let us solve tasks that are hard to solve by other means.

In the next example (taken from [12]), we use CLP($\mathcal{B}$) constraints to express maximal independent sets of graphs: Boolean variables b_i denote whether node i is in the set. In addition, each node i is assigned a weight w_i. The task is to find a maximal independent set that maximizes the total weight $\sum b_i w_i$. For concreteness, let us consider the cycle graph C_{100}, and assign each node i the weight $w_i = (-1)^{\nu(i)}$, where $\nu(i)$ is the number of ones in the binary representation of i. The grey nodes in Fig. 7 show a maximal independent set of C_{100} with maximum total weight. In the figure, nodes with negative weight are drawn as squares, and nodes with positive weight are drawn as circles.

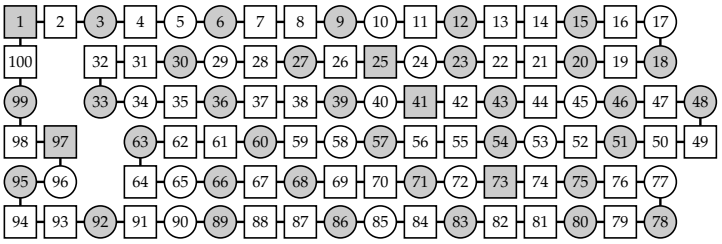


Fig. 7: Maximal independent set of C_{100} with maximum weight (= 28)

CLP($\mathcal{B}$) constraints yield the optimum (28) within a few seconds in this example. Moreover, we can use our new interface predicates to compute other interesting facts. For example, C_{100} has exactly 792,070,839,848,372,253,127 independent sets, and exactly 1,630,580,875,002 *maximal* independent sets.

6 Benchmark results

We now use several benchmarks to compare the performance of our system with the CLP($\mathcal{B}$) library that ships with SICStus Prolog. We are using SWI-Prolog version 7.3.7, and SICStus Prolog version 4.3.2. All programs are run on an Intel Core i7 CPU (2.67 GHz) with 48 GB RAM, using Debian 8.1.

The benchmarks comprise examples[6] from the literature that are also used in [4] and other publications:

`pigeon8x9`: The task of attempting to place 9 pigeons into 8 holes in such a way that each hole contains at most one pigeon. Clearly, this problem is unsatisfiable.

`queens` N: Placing N queens on an $N \times N$ chessboard in such a way that no queen is under attack.

[6] The code of all benchmarks is available at `https://www.metalevel.at/clpb-bench`

`schur` N: Distribute the numbers $1, \ldots, N$ into 3 *sum-free* sets. A set S is sum-free iff $i, j \in S$ implies $i + j \notin S$. This is satisfiable for all N up to and including the *Schur number* $S(3) = 13$, and unsatisfiable for $N > 13$.

`triominoes` N: Triomino cover (see Section 5.2) of an $N \times N$ chessboard.

We benchmark each example in three different ways, and the results are summarized in Table 3: First, we build a single conjunction `C` of all clauses and post `sat(C)`. The columns titled `sat` show the timing results (in seconds) of this call for SWI and SICStus, respectively. Then, we build a list `Cs` of clauses and post `maplist(sat, Cs)`. The timing result of this is shown in the `sats` columns. Finally, we invoke `taut(C, _)`, and the timing results of this call are shown in the `taut` columns.

name	*vars*	*clauses*	SWI 7.3.7			SICStus 4.3.2		
			`sat`	`sats`	`taut`	`sat`	`sats`	`taut`
`pigeon8x9`	72	17	1.2	1.2	1.2	0.07	0.01	0.05
`queens6`	36	302	12.7	12.8	12.9	0.01	2.7	0.01
`queens7`	49	490	65.7	65.9	67.3	3.62	22166	0.03
`schur13`	39	139	10.6	10.7	10.7	0.31	2.8	0.19
`schur14`	42	161	13.1	13.2	13.3	0.57	7.63	0.41
`triominoes5`	94	25	3.6	3.7	3.7	0.01	–	0.02
`triominoes6`	148	36	22.6	22.7	23.3	–	–	0.08

Table 3: Running times (in seconds) of different benchmarks

There are several things worth pointing out about these results: First, it is evident that the CLP($\mathcal{B}$) solver of SICStus Prolog often vastly outperforms our library. We can safely expect the SICStus library to be at least two orders of magnitude faster than ours on many benchmarks. In part, this huge difference in performance may certainly be attributed to the fact that SWI-Prolog itself is already more than three times slower than SICStus Prolog on benchmarks that are in some sense deemed to be representative of many applications. Neng-Fa Zhou, the author of B-Prolog, kindly maintains a collection of these results at `http://www.picat-lang.org/bprolog/performance.htm`. Since our library is written entirely in Prolog, it strongly depends on the performance of the underlying Prolog system.

Second, there is a large relative difference between the `sat` and `sats` columns within SICStus Prolog. In the `queens7` case, it is particularly pronounced. In SWI-Prolog, there is virtually no difference between these variants, because we *always* implicitly post individual `sat/1` constraints if the given formula is a compound term with principal functor `*/2`, i.e., a conjunction.

Third, some of the benchmarks cannot be solved at all with SICStus Prolog on this machine: We use "–" to denote an *insufficient memory* exception.

7 CLP($\mathcal{B}$) with other types of decision diagrams

BDDs are not the only kind of decision diagrams that are practically useful, and the question arises whether other types of decision diagrams are not at least equally suitable as the basis of CLP($\mathcal{B}$) systems.

To collect preliminary experiences with different implementation variants, we have created a variant[7] of `library(clpb)` that is based on Zero-suppressed Binary Decision Diagrams (ZDDs). The key idea of ZDDs [15] is to assign a slightly different meaning to the diagram: In ZDDs, a branch leading to **1** only means **true** if all variables that are *skipped* in that branch are *zero*. ZDDs are therefore especially useful when many variables are zero in solutions.

The ZDD-based variant of `library(clpb)` does not feature all the functionality that the BDD-based version provides. This is due to two main reasons: The first reason is that, due to the different semantics of the diagrams, a ZDD-based approach necessitates that all variables be known *in advance*, at least if we want to avoid rebuilding all ZDDs every time a new variable occurs. Therefore, a special library predicate must be called before using the ZDD-based version in order to "declare" all Boolean variables that appear in the formulation. The ZDD-based variant is thus not a drop-in replacement of the BDD-based version that ships with SWI-Prolog.

The second reason is that the shortcomings (see Section 4.7) of SWI-Prolog's interface predicates for attributed variables are especially severe when ZDDs are involved. This is because simultaneous unifications, such as `[A,B] = [0,1]`, significantly complicate the reasoning when deciding whether a variable (still) occurs in a ZDD. With BDDs, these limitations are a bit less severe, because a variable that does not occur in a BDD can assume either truth value.

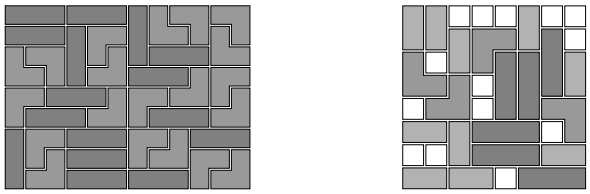
So far, we have collected only very limited experience with ZDDs, in part also due to the mentioned limitations of SWI-Prolog's interface predicates. Nevertheless, we would like to point out two interesting tasks for which the ZDD-based variant is very well suited, and hint at planned future developments.

First, we extend the triomono tiling task to a 9×12 grid. One solution is shown in Fig. 8 (a). Project Euler Problem 161 asks for the *number* of such tilings. Using the ZDD-based variant, it takes about 13 GB RAM and 2 days of computation time to construct a ZDD that represents all solutions and compute the number (which is 20,574,308,184,277,971). Using the BDD-based version of `library(clpb)` requires more than 4 times as much memory.

Second, we allow, in addition to triominoes, also monominoes and dominoes, and cover an 8×8 chessboard. Fig. 8 (b) shows one solution. With the ZDD-based variant, 1 GB RAM suffices to compute the number of possible coverings (there are exactly 92,109,458,286,284,989,468,604 of them). Using BDDs takes about 10 times as much memory.

Many other interesting applications of ZDDs are described in [12], and we plan to make many of them accessible in future versions of this library variant. This may require suitable additional interface predicates.

[7] The variant is freely available at `https://www.metalevel.at/clpb-zdd`



(a) (b)

Fig. 8: (a) Project Euler Problem 161: Covering a 9×12 grid with triominoes; (b) Covering a chessboard with monominoes, dominoes and triominoes

8 Conclusion and future work

We have presented the first BDD-based CLP($\mathcal{B}$) system that is freely available. It features new interface predicates that allow us to solve new applications with CLP($\mathcal{B}$) constraints.

Implementing the system in Prolog has allowed us to prototype many ideas quickly. The implementation provides a high-level description of all relevant ideas, and is portable to other Prolog systems that support attributed variables.

We hope that the availability of a free BDD-based CLP($\mathcal{B}$) system leads to increased interest in CLP($\mathcal{B}$) constraints within the Prolog community, and encourages other vendors to likewise support such libraries.

Ongoing and future work is focused on additional test cases to ensure the system's correctness, porting the system to other Prolog systems such as YAP and SICStus Prolog, and improving performance. Stefan Israelsson Tampe is currently porting the solver to Guile-log, a Prolog system based on Guile.

Additional interface predicates may be needed to cover further applications of BDDs and ZDDs. Careful design of these predicates is necessary to provide sufficient generality without exposing users to low-level details of the library.

9 Acknowledgments

First and foremost, I thank Ulrich Neumerkel for introducing me to constraint logic programming and to testing constraint solvers. For their encouragement about CLP($\mathcal{B}$), I thank Nysret Musliu and Fred Mesnard. My gratitude also goes to Jan Wielemaker for providing a robust and free Prolog system, for his fast reaction times and much appreciated support when discussing and implementing new features. I thank Mats Carlsson for the visionary CLP($\mathcal{B}$) solver of SICStus Prolog and sending me a complimentary version of his system. For their supremely well-written documents about BDDs, I thank Donald Knuth and Henrik Reif Andersen. These books and papers further increased my interest in the subject and were very useful during development. I also thank the anonymous reviewers for their helpful comments.

With all my heart, I thank my partner Barbara for her love.

References

1. Benhamou, F., Touraïvane: Prolog IV : langage et algorithmes. In: JFPLC. pp. 51–64 (1995)
2. Bryant, R.E.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Trans. Computers 35(8), 677–691 (1986)
3. Burckel, S., Hoarau, S., Mesnard, F., Neumerkel, U.: cTI: Bottom-up termination inference for logic programs. In: 15. WLP. pp. 123–134 (2000)
4. Carlsson, M.: Boolean Constraints in SICStus Prolog. SICS TR T91:09 (1991)
5. Codognet, P., Diaz, D.: A Simple and Efficient Boolean Solver for Constraint Logic Programming. J. Autom. Reasoning 17(1), 97–129 (1996)
6. Colin, S., Mesnard, F., Rauzy, A.: Un module Prolog de mu-calcul booléen: une réalisation par BDD. In: JFPLC'99, Huitièmes Journées Francophones de Programmation Logique et Programmation par Contraintes. pp. 23–38 (1999)
7. Demoen, B.: Dynamic attributes, their hProlog implementation, and a first evaluation. Report CW 350, Dept. of Computer Science, K.U. Leuven (Oct 2002)
8. Diaz, D., Abreu, S., Codognet, P.: On the implementation of GNU Prolog. TPLP 12(1-2), 253–282 (2012)
9. Dincbas, M., Hentenryck, P.V., Simonis, H., Aggoun, A., Graf, T., Berthier, F.: The constraint logic programming language CHIP. In: FGCS. pp. 693–702 (1988)
10. Hooker, J.N.: Projection, consistency, and George Boole. Constraints 21(1), 59–76 (2016)
11. Jaffar, J., Lassez, J.L.: Constraint Logic Programming. In: POPL. pp. 111–119 (1987)
12. Knuth, D.E.: The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams. Addison-Wesley Professional, 12th edn. (2009)
13. Mantadelis, T., Rocha, R., Kimmig, A., Janssens, G.: Preprocessing Boolean Formulae for BDDs in a Probabilistic Context. In: Proceedings of the 12th European Conference on Logics in Artificial Intelligence. pp. 260–272. JELIA'10, Springer-Verlag, Berlin, Heidelberg (2010)
14. Marques-Silva, J.P.: Algebraic simplification techniques for propositional satisfiability. In: CP'00. LNCS, vol. 1894 (2000)
15. Minato, S.: Zero-suppressed BDDs for set manipulation in combinatorial problems. In: Design Automation Conference (DAC). pp. 272–277 (1993)
16. Neumerkel, U.: Teaching Prolog and CLP (tutorial). ICLP (1997)
17. Neumerkel, U., Kral, S.: Declarative program development in Prolog with GUPU. In: Proceedings of the 12th International Workshop on Logic Programming Environments, WLPE. pp. 77–86 (2002)
18. Selman, B., Kautz, H., Cohen, B.: Local search strategies for satisfiability testing. Second DIMACS Implementation Challenge (1993)
19. Tarau, P.: Pairing Functions, Boolean Evaluation and Binary Decision Diagrams. CoRR abs/0808.0555 (2008), `http://arxiv.org/abs/0808.0555`
20. Tarau, P., Luderman, B.: Boolean Evaluation with a Pairing and Unpairing Function. In: SYNASC 2012. pp. 384–390 (2012)
21. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. TPLP 12(1-2), 67–96 (2012)
22. Zhang, H.: SATO: An Efficient Propositional Prover. In: CADE. LNAI, vol. 1249 (1997)