Boolean Constraints in SWI-Prolog: A Comprehensive System Description

Markus Triska[a,*]

[a] *TU Wien, Austria*

Abstract

We present a new constraint solver over Boolean variables, freely available as `library(clpb)`[1] in SWI-Prolog. In this paper, we explain the core algorithms and implementation trade-offs of our system. 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 other systems that provide a few interface predicates that we outline. Second, our system provides new interface predicates, and we show that they allow us to solve new types of problems with CLP($\mathcal{B}$) constraints. Finally, we present performance results and comparisons with the native CLP($\mathcal{B}$) solver of SICStus Prolog, and also with a new SICStus port of our system. Despite being written entirely in Prolog, both versions of our system can solve several benchmark instances that the native CLP($\mathcal{B}$) solver of SICStus Prolog cannot solve.

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($\mathcal{X}$) scheme introduced by Jaffar and Lassez [1]

[*] Corresponding author
Email address: `triska@dbai.tuwien.ac.at` (Markus Triska)
URL: `https://www.metalevel.at` (Markus Triska)

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

The final publication is available at `http://dx.doi.org/10.1016/j.scico.2018.02.001`

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 [2, 3, 4] and many others). Prolog itself in fact turns out to be very suitable for implementing small and clean SAT solvers [5].

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 outline the current state of available CLP($\mathcal{B}$) systems. Section 3 defines Binary Decision Diagrams and describes the key algorithms we are using in our implementation. In Section 4, we present the new interface predicates of our CLP($\mathcal{B}$) system. Further, we describe in detail how we have implemented the mentioned algorithms, and which trade-offs we have made to obtain a portable CLP($\mathcal{B}$) system with acceptable performance. Section 5 describes new applications made possible by the new features of our library, followed by benchmark results in Section 6. In Section 7, we describe a set of test cases.

This paper is a significantly extended and revised version of [6]. The additional material covered in the present version includes completely new material, and also significantly extended explanations and material that previously had to be omitted due to space constraints.

The first group of additions comprises the new Section 3.2 with high-level descriptions of the BDD algorithms we are using, new benchmark results in Section 6.4 that are obtained with the newly available SICStus port of our CLP($\mathcal{B}$) system, and the new Section 7 on testing a CLP($\mathcal{B}$) system.

The second group of extensions includes a much more thorough explanation of our implementation and trade-offs in Section 4, a more compelling sample application of the new interface predicate `sat_count/2` in Section 5.1, and several smaller extensions such as additional benchmark instances in Section 6.

Familiarity with Prolog is assumed to completely understand the Prolog-specific terminology and notation of this paper. As a thorough introduction, we recommend [7]. In addition, we provide accompanying information about modern Prolog with a strong focus on *constraints* and more recent developments in a collection of free essays at `https://www.metalevel.at/prolog`.

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 [8] was one of the first widely used systems to support CLP($\mathcal{B}$) constraints, and shortly after, SICStus Prolog supported them too [9], 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 [10] and GNU Prolog [11] do support Boolean constraints.

Implementation methods of CLP($\mathcal{B}$) systems are likewise diverse. We find at least three different approaches in the literature: (1) implementations based on Binary Decision Diagrams (BDDs) as in [9], (2) implementation of CLP($\mathcal{B}$) constraints by other constraints, using for example indexicals [12], and (3) using an external SAT solver [13].

Each of these variants has strengths and weaknesses: Among the major advantages of BDD-based implementations we find some algebraic virtues and suitability for specific applications which we explain in the following. In comparison, indexical-based implementations are generally simpler, more scalable

[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.”

and much more efficient on selected benchmarks [12]. However, they require an explicit search to ensure the existence of solutions after posting constraints. Using external solvers may render us unable to generate *all* satisfying assignments. Approaches with such properties are called *incomplete*. See also Appendix B.

3. Binary Decision Diagrams (BDDs)

3.1. Definition and related work

A Binary Decision Diagram (BDD) is a rooted, directed and acyclic graph and represents a Boolean function [14, 15]. 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.

Leaves of a BDD are the truth values **true** and **false**. Each internal node in a BDD is associated with a *branching variable* `V` and exactly two children: *low* corresponding to `V` = 0, and *high* corresponding to `V` = 1. Therefore, each internal node can be read as an if-then-else, using (`V`→ *high* `;` *low*) in analogy to well-known Prolog syntax.

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] BDDs also occur in publications that introduce or use closely related data structures [16, 17]. Within the logic programming community, important applications of BDDs arise in the context of *probabilistic* logic programming [18] and termination analysis of Prolog programs [19, 20].

In our CLP($\mathcal{B}$) system which we present in Section 4, it is possible to inspect arising BDDs on the Prolog top-level. See Appendix A for an example.

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

3.2. Algorithms on BDDs

As mentioned in Section 3.1, a BDD represents a Boolean function. In this section, we present a brief overview of the *algorithms* that typically arise in the context of BDDs and which are also used in our CLP($\mathcal{B}$) implementation. The algorithms we are using are a combination of those that are presented in [15] in a sequence of increasingly sophisticated algorithms for reasoning about BDDs. We describe below how we have combined parts of these algorithms in such a way that they become suitable for use in our CLP($\mathcal{B}$) system.

One key observation is that BDDs can be constructed *incrementally*, by reasoning about subformulas of the Boolean functions we want to represent as BDDs. The two simplest cases are:

- The truth values **true** and **false** are represented by themselves, as leaves of any BDD.
- A single Boolean variable X corresponds to the BDD ($X\!\rightarrow$ **true** ; **false**). See 3.1 for the explanation of this syntax: Iff X is *true*, then the whole formula (consisting only of X) is true.

We now describe how to construct the BDD of a compound formula. First, we describe an important operation called *melding* of BDDs. Consider two BDDs f and g of the respective forms $(x\!\rightarrow f_1 \;;\; f_0)$ and $(y\!\rightarrow g_1 \;;\; g_0)$, where x and y are branching variables, and f_i and g_i are again BDDs. In accordance to the definition in Section 3.1, we assume that we have imposed an *order* on the branching variables that arise in these BDDs. We define the *meld* $f \diamond g$ as follows:

$$f \diamond g = \begin{cases} (x\!\rightarrow f_1 \diamond g_1 \;;\; f_0 \diamond g_0), & \text{if } x = y; \\ (x\!\rightarrow f_1 \diamond g \;;\; f_0 \diamond g), & \text{if } x < y; \\ (y\!\rightarrow f \diamond g_1 \;;\; f \diamond g_0), & \text{if } x > y. \end{cases}$$

For any BDD f, we denote with $B(f)$ the total *number of nodes* in f, including the leaves. It is easy to see that $B(f \diamond g) \leq B(f)B(g)$ because each node of $f \diamond g$ corresponds to one node of f and one node of g.

Further, melding lets us derive an algorithm for constructing the BDD of a compound formula: We can plug in one of the Boolean connectives $\wedge$, $\vee$ and $\oplus$ in place of $\diamond$, and augment the definition with the following additional provisions:

1. If at least one of f and g is a concrete Boolean value (**true** or **false**), then computing the BDD corresponding to $f \diamond g$ is straightforward.
2. If, during melding, a new BDD is constructed that is *identical* to one that has already been constructed in earlier steps, then the existing one must be used. This provision ensures that all isomorphic subgraphs are merged, which is a prerequisite for keeping the BDD *reduced*.
3. If, during melding, a BDD of the form $(x \!\rightarrow f \;;\; f)$ arises, then f is used instead. This ensures that no redundant nodes occur, and in combination with (2) ensures that the BDD is *reduced*.

Since melding itself produces only ordered BDDs, the result of these provisions is that all arising BDDs are ordered *and* reduced.

This yields an algorithm with run time cost proportional to at least $B(f)B(g)$ for these binary connectives. The key to make this construction more efficient is to *memoize* intermediate results that have already been computed. We illustrate this algorithm for combining two BDDs by using Boolean *conjunction* as one concrete example. The other Boolean connectives are handled completely analogously. We describe how we have implemented this in Section 4.6:

$$f \wedge g = \begin{cases} \text{If } f \wedge g = r \text{ is in the memo cache, return } r. \\ \text{Otherwise, compute } r \leftarrow f \wedge g \text{ as explained above.} \\ \text{Store } f \wedge g = r \text{ in the memo cache, and return } r. \end{cases}$$

We thus summarize the worst-case run time costs of constructing a conjunction $f \wedge g$ of two BDDs by this method as follows: First, the resulting BDD may have up to $B(f)B(g)$ nodes. To make nodes accessible at all, we must represent them somehow in memory. Nodes and their immediate children are naturally represented as Prolog *terms* (see Section 4.6) and can be accessed in $O(1)$ time by simply referencing these terms. However, this still leaves two questions: First, how to detect whether a node is being constructed that is *identical* to one that was already created? In our implementation, we accomplish this by storing all nodes in an *AVL tree*, which we explain in Section 4.6.3. Second, we need a *memo cache* to store intermediate results. In [15], a *hash table* is suggested for this purpose, but the following option is also mentioned although it again entails some overhead: We can use a balanced *tree* to store intermediate results. This is the method we have chosen, since it is a natural and portable fit for a Prolog-based implementation. We are storing intermediate results *also* in an *AVL tree*. Hence, in the worst case, looking up a single intermediate result has running time that is proportional to $\log\big(B(f)B(g)\big)$.

Let us denote with N the *total number* of nodes that our system stores in memory at the current moment, *plus* the number $B(f)B(g)$ which is the maximal number of new nodes that can arise when computing $f \circ g$ for any binary connective $\circ$.

For each new node that is created, we must look up whether it already exists in the AVL tree of existing nodes, which is accomplished in $O(\log N)$, i.e., logarithmic in the total number of nodes that exist so far. Further, we look up each intermediate result in the memo cache, which is accomplished in $O\big(\log(B(f)B(g))\big)$ in the worst case.

Thus, the total worst case running time of constructing the BDD of $f \circ g$ is: $O\big((\log N + \log(B(f)B(g))) \times B(f)B(g)\big)$, and hence $O\big(B(f)B(g)\log N\big)$. The asymptotically *optimal* worst-case performance is $O\big(B(f)B(g)\big)$. Thus, the inherent overhead of our implementation is proportional to $\log N$. We have chosen to *accept* this overhead, because the limiting factor when reasoning over BDDs is usually *memory*, and for typical N (say, up to a few million nodes), $\log N$ is acceptably small for our purposes. Moreover, we apply a technique that is suggested in [15] to further reduce this overhead in general: In our implementation, we maintain one AVL tree *per branching variable*, and each such tree only contains those nodes where the corresponding variable occurs at the root. Section 4.6 contains more information about the representation we are using and its consequences. In cases where many intermediate results can be reused, the running time is significantly *less* than the worst case, due to the use of the memo cache. As noted in [15], only $B(f) + B(g)$ new nodes arise in many cases of practical relevance when combining two BDDs. In such cases, the running time of our implementation is $O\big((B(f) + B(g)) \log N\big)$.

Once a BDD f is constructed, many important operations on it can be performed in time $O\big(B(f)\big)$ or even $O(1)$. In the context of our system, we benefit from the following features after a BDD is constructed: First, we can determine in $O(1)$ whether f is *satisfiable*. This follows from the important fact that reduced and ordered BDDs are *canonical* and hence such a BDD is satisfiable *iff* it is different from the single leaf **false**. Second, if f is satisfiable, we can compute a concrete *satisfying assignment* in time and space that is *linear* in the number of variables of f, by traversing the BDD and searching for a path that leads to **true** which is known to exist in that case.

The following two algorithms share an important design principle: They work by first recursively *counting* the number of solutions in each of the children of any node, storing intermediate results as attributes of those nodes that were already visited. Considering that for each node, a k-bit number must be stored and computed, this step takes time and space proportional to $O\big(kB(f)\big)$ and is straightforward to implement by inductively computing the number of solutions in the form of powers of 2, one for each level of branching variables. Once this is available, we can efficiently compute (1) the *total number* of solutions (2) a *random* solution in the sense that each satisfying assignment is *equally likely*. We have followed the descriptions contained in [15] for both cases.

The computation of a *weighted solution* which attains minimum (or maximum) weight among all satisfying assignments when each Boolean variable is assigned an integral weight is similarly straightforward [15] to accomplish in $O\big(n + B(f)\big)$ time and space, where n is the number of variables.

Critically though, the BDD algorithms themselves are not sufficient to implement a CLP($\mathcal{B}$) system: In a logic programming language like Prolog, variables may become *aliased*. The existing literature on BDDs does not explain how to handle this case. We explain this aspect in tandem with the Prolog implementation of these algorithms in Section 4.6, and show examples of their application in Section 5.

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

We have implemented a new free CLP($\mathcal{B}$) system, available in SWI-Prolog [21] as `library(clpb)`. In this section, we present the design choices, interface predicates and implementation. Subsections 4.6 and 4.7 are targeted at implementors and contributors of Prolog systems and constraint libraries, and assume familiarity with the algorithms and preliminaries presented in Sections 3.2 and 4.5.

4.1. 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 answers to queries, 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`. Note that this syntax in fact permits us to express[4] *Pseudo-Boolean* constraints, which generalize *cardinality* constraints by allowing integer coefficients in Boolean formulas. For example, the Pseudo-Boolean constraint $2a + b + \bar{c} \geq 2$ can be expressed in CLP($\mathcal{B}$) as `sat(card([2-4],[A,A,B,~C]))`. Cardinality constraints are readily translated to BDDs that represent *counter networks* [9].

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

[4]See [22] for the reduction scheme and some related results.

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

4.2. 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 satisfied, and with `T=1` if `T` is a tautology with respect to the stated constraints.

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

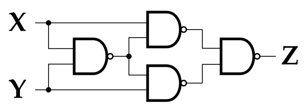
4.3. Implementation choices: BDDs, SAT solvers, external libraries

Before presenting the 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 [23, 24] 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 [19], 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



```
:- 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).
```

(a) (b)

Figure 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)`.

in cTI) is hard in general, but straightforward if the arising BDDs are small. 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 4.1) 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 answers, and *less* about concrete solutions. For example, with the given program, the query `?- xor(x, y, Z).` yields the answer `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 answers, existential quantification is implicitly used by the Prolog top-level to project away variables that do not occur in the query.

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

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 [9] 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 does the notion of *universally* quantified variables affect all operations on BDDs? Which kind of consistency is guaranteed? 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 4.5) 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.

4.4. New interface predicates

As already mentioned, BDDs have many important virtues that can be easily made available in a BDD-based CLP($\mathcal{B}$) system. As explained in Section 3.2 and [15], the key idea of several efficient algorithms on BDDs is 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 Section 4.2, we have implemented three new predicates that are not yet available in SICStus Prolog, and which are high-level interfaces to the algorithms described in Section 3.2:

`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 Section 5, these predicates are of great value in many applications, and we encourage their support in other CLP($\mathcal{B}$) systems based on BDDs. We describe design principles for new interface predicates in Appendix C.

4.5. Attributed variables

Before explaining the actual implementation of our new system, a short discussion of *attributed variables* is highly appropriate. This is because Prolog application programmers typically encounter attributed variables only indirectly, whereas they play a major role in the internal implementation of our CLP($\mathcal{B}$) system.

Attributed variables are a ternary relation between (i) a variable, (ii) the name of the attribute, and (iii) the value of the attribute. Importantly, attributes can be fetched and changed in such a way that all modifications are *undone* on backtracking. Moreover, when an attributed variable is involved in a *unification*, then a special user-provided predicate is invoked that can analyse existing attributes and either allow or veto the unification based on the available information. Attributed variables are an important mechanism for implementing constraint solvers.

In SWI-Prolog, the interface of attributed variables follows that of hProlog [25], and we now briefly explain the built-in interface predicates for attributed variables in SWI-Prolog. These predicates can be classified into three groups: (1) accessing and changing attributes, (2) reasoning about *unifications* of attributed variables and (3) displaying attributed variables in *answers* to queries. In SWI-Prolog, the name of an attribute is always a Prolog *atom* and at the same time corresponds to a Prolog *module*.

The most important predicates of the first category are `get_attr/3` and `put_attr/3`. We do not explain these predicates in detail here: For the following, it suffices to understand that attributes can be used to attach information to variables, and that this information can be taken into account when such variables are being unified. We refer readers to the SWI-Prolog documentation for more information about these two predicates.

In the second category, there is an important interface predicate called `attr_unify_hook/2`:

`attr_unify_hook(+AttValue, +VarValue)`: A predicate that must be defined in the module to which an attributed variable refers. It is called *after* the attributed variable has been unified with a non-free term, possibly another attributed variable. `AttValue` is the value of the attribute that was associated to the variable in this module. `VarValue` is the new value of the variable. If this predicate fails, the unification fails. If `VarValue` is another attributed variable, the hook often combines the two attributes and associates the combined attribute with `VarValue` using `put_attr/3`.

In the third category, we find the predicate `project_attributes/2` and the nonterminal[5] `attribute_goals//1`. The predicate `project_attributes/2` is used to project all remaining constraints onto variables that appear in the original query, and the nonterminal `attribute_goals//1` – if it is defined in a module – relates the attributes of that module to a *list* of Prolog goals that express the pending constraints. The Prolog system automatically calls these predicates when necessary. We refer interested readers to the SWI-Prolog manual for the exact definition. Other Prolog systems such as SICStus Prolog provide similar interfaces for attributed variables that are more convenient and less error-prone.

[5]This concept is used in Prolog *Definite Clause Grammars* (DCGs), which we explain at: `https://www.metalevel.at/prolog/dcg`. We also provide a more comprehensive introduction to *attributed variables* at: `https://www.metalevel.at/prolog/attributedvariables`.

4.6. Implementation

We now explain the actual implementation of our system. Perhaps most strikingly, our system 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 1 800 LOC, including documentation and comments. Some knowledge of Prolog is required to understand this section. If more information is required about any predicate `P` that is mentioned in this section, we recommend to run the following query in SWI-Prolog: `?- help(P).` For example: `?- help(sat).`

The following sections explain the implementation of our system in considerable detail. For example, we outline how we *represent* BDDs (Section 4.6.2), how we use AVL trees (Section 4.6.3), how we construct new BDD nodes (Section 4.6.4), and how we have implemented various other aspects of our system. We summarize and discuss implementation options and trade-offs in Section 4.9.

4.6.1. Prerequisites

On a very high level, the implementation can be reduced to expressing the algorithms explained in Section 3.2 in Prolog. This is because every Prolog goal involving the interface predicates explained in Sections 4.2 and 4.4 performs one of these algorithms. However, to obtain a full CLP($\mathcal{B}$) system, there are additional assumptions and subtleties involved, which we now explain.

In particular, we rely on the following assumptions regarding the underlying Prolog system:

1. There is a way to obtain a sequence of strictly increasing *integers* that can be accessed from any part of our code. These integers serve to assign an *order* to all occurring CLP($\mathcal{B}$) variables, and also provide unique identifiers of BDD *nodes*.
2. There is a way to attach information to logical variables. This is necessary to keep track of all constraints these variables are involved in.

3. There is a way to take existing constraints into account when two logical variables are *unified*.

SWI-Prolog, SICStus Prolog, and also several other Prolog systems all satisfy these requirements. In particular:

- Requirement (1) is satisfied in *all* Prolog systems: For example, the so-called *global database* can be used to store and change global information via the standard predicates `assertz/1` and `retract/1`.
- Requirements (2) and (3) are satisfied by the interface predicates to attributed variables mentioned in Section 4.5.

4.6.2. Representing BDDs in Prolog

We now discuss how we represent and reason about BDDs in our system. First, let us consider the representation of a BDD that is already constructed. In our system, we represent every BDD as a Prolog *term*. We distinguish two basic cases: (1) the *leaves* **false** and **true** are respectively represented by the integers `0` and `1`, and (2) *internal* nodes are represented as `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) at least 48 bytes per node on 64-bit systems. Using the notation of Section 3.1, this is the node (`Var`→ `High ; Low`), identified by the unique integer `ID`, and additional information that can be stored in `Aux`.

Note that more must hold: We must ensure that BDDs are *reduced* (see Section 3.1) and in particular that *all isomorphic subgraphs are shared*. Before we explain how we have implemented this, we consider the representation of *branching variables*: In our system, each branching variable is a Prolog variable, and we can therefore attach attributes to it (Section 4.5). The attributes of each branching variable `V` are:

- the *index* of `V`. The index is a unique integer for each branching variable.
- the BDD to which `V` belongs. We explain this point below.
- an *AVL tree* that stores all nodes where `V` is a branching variable. See Section 3.2.

4.6.3. AVL trees in Prolog

To reason about AVL trees in Prolog, we use `library(assoc)`: This library ships with SWI-Prolog and provides AVL trees based on a public domain library that was originally written by Richard O'Keefe. An AVL tree as implemented by this library is a data structure that provides an *association* between unique ground Prolog terms called the *keys*, and corresponding *values* which are arbitrary Prolog terms. For this reason, AVL trees and other balanced trees are also called *association lists* in Prolog. Since `library(assoc)` uses AVL trees as its underlying data structure, inserting, changing and fetching an association all take $O(\log N)$ time in the worst case (and average case), where N is the number of associations stored in the tree.

There are only 3 operations on AVL trees that are needed to implement our CLP($\mathcal{B}$) system:

- `list_to_assoc(Pairs, Assoc)`: `Pairs` is a given list of `Key-Value` *pairs*, and `Assoc` is the resulting AVL tree. For example, the *empty* association list `E` can be obtained with `list_to_assoc([], E)`.
- `get_assoc(Key, Assoc, Value)`: Given association list `Assoc` and key `Key`, fetch the associated value `Value` if it exists. Otherwise, *fail*.
- `put_assoc(Key, Assoc0, Value, Assoc)`: Given an existing AVL tree `Assoc0`, as well as key `Key` and value `Value`, compute a new AVL tree

`Assoc` where `Key` is associated with `Value`. This can be used to *add* as well as to *change* an association.

The mentioned AVL trees which are stored as *attributes* of branching variables are critical to keep all BDDs reduced. They are used as follows: Suppose we are given two existing nodes `Low` and `High`, and want to create a new node `M` to represent the BDD (`V`→ `High ; Low`). Further, we denote with *id*(`N`) the unique integer ID of the node `N` (if it is an inner node) or the atoms `true` or `false` (if it is a leaf node). Then the Prolog term `node`(*id*(`Low`), *id*(`High`)) is a unique identifier of the new node `M` with respect to the decision variable `V`.

Importantly, terms of the form `node(I,J)`, where `I` and `J` are integers or atoms, are *ground* Prolog terms (they contain no variables) and can hence be used as *keys* of an AVL tree.

4.6.4. Constructing BDD nodes in Prolog

We are now ready to describe how a new node (`V`→ `High ; Low`) is constructed if its children `Low` and `High` are already computed. In our system, this construction is implemented as a Prolog *predicate*. However, we consciously present it here in functional terminology to emphasize the *directional* aspect of this computation: The branching variable `V` and the two children `Low` and `High` must be *given*, and the synthesized node `N := make_node(V, Low, High)` is computed. In pseudo-code:

$$\texttt{make\_node}(\texttt{V},\texttt{Low},\texttt{High}) = \begin{cases} \text{If } \texttt{Low == High}\text{, return } \texttt{Low}. \\ \text{Otherwise, set } \texttt{Key} := \texttt{node}(id(\texttt{Low}), id(\texttt{High})). \\ \text{If } \texttt{Key} \text{ exists in the AVL tree of } \texttt{V}\text{, return} \\ \qquad \text{the associated node.} \\ \text{Otherwise, obtain a new unique integer } \texttt{ID}. \\ \text{Set } \texttt{Node} := \texttt{node}(\texttt{ID},\texttt{V},\texttt{Low},\texttt{High},\texttt{\_Aux}). \\ \text{Register } \texttt{Key-Node} \text{ in the AVL tree of } \texttt{V}. \\ \text{Return } \texttt{Node}. \end{cases}$$

In the above description, `_Aux` is a new fresh variable that can be used to attach attributes to the new node when needed. For example, this is used by `sat_count/2` to recursively compute the number of possible assignments by the method explained in Section 3.2. In such computations, `_Aux` is used to store results that are already computed for a node.

In our system, these steps are executed every time a new node is constructed. This guarantees that all arising BDDs are *reduced*. The reason for this is that internally, the underlying Prolog system uses *references* to existing terms when yielding results from association lists. This happens completely automatically, due to the way association lists are implemented in `library(assoc)`. Therefore, when a node already exists, it is reused in the construction.

Since each existing node is also represented in the association list of its branching variable, the space that is necessary to represent any BDD – and which we have considered above – is roughly *doubled*. Therefore, compared to hashing as it is used in [15], using association lists incurs linear space overhead and logarithmic time overhead. On the plus side, association lists scale very predictably and do not require any *ad hoc* considerations and complex treatment of edge-cases. In addition, they are well suited for a Prolog implementation.

4.6.5. Interaction with logical variables

We now discuss how Prolog *variables* interact with these algorithms. In a CLP($\mathcal{B}$) system, we want to treat Prolog variables as actual *logical* variables, which means that two variables may become *aliased* by *unification*. Before we explain how to make this possible, we first describe how we represent CLP($\mathcal{B}$) variables in general.

In our system, each CLP($\mathcal{B}$) variable belongs to exactly one BDD and gets an attribute of the form `index_root(Index,Root)`, where `Index` is the variable's unique integer index, and `Root` is the *root* (see below) of the BDD that the variable belongs to. When a new logical variable is encountered by one of the CLP($\mathcal{B}$) interface predicates, the variable gets assigned a unique integer ID. This is used to make BDDs *ordered*. The mechanism for creating unique IDs for variables is analogous to how IDs are assigned to new *nodes*, and is ensured by requirement (1).

With these provisions in place, constructing the BDD of any binary connective is simply a matter of performing the steps outlined in Section 3.2 while using the algorithm of Section 4.6.4 to create nodes. When a new BDD is being constructed, intermediate results are stored in a (fresh) association list for *memoization*. The *keys* in this association list are ground Prolog terms of the form $\circ(id(\texttt{F}), id(\texttt{G}))$, where the *functor* $\circ$ denotes one of the binary connectives $\wedge$, $\vee$ or $\oplus$, and `F` and `G` are BDD nodes. The associated *value*, if it exists, denotes the resulting *node* of this operation, and can be used instead of re-computing the (*necessarily identical*) node. This association list is discarded after the BDD is fully constructed, i.e., upon completion of the solver interface predicates such as `sat/1`.

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`.

We say that two BDDs are *disjoint* if they have no branching variable (and hence no inner node) in common. In precisely two situations, two or more disjoint BDDs need to be combined into a single new BDD. These situations are: (a) the occurrence of a new constraint which contains variables from disjoint BDDs and (b) *aliasing* of two logical variables that belong to disjoint BDDs.

We consider (a) first also because it is, in a sense, the more general case: Declaratively, we can interpret the unification of two variables `X` and `Y` as the constraint `sat(X=:=Y)`. However, this does not completely explain what happens operationally upon unification, and we therefore discuss unification separately below. We illustrate (a) with a simple example: From the above description, it follows that the conjunction of constraints `(sat(A+B),sat(C+D))` yields precisely two (disjoint) BDDs, one for each of the two constraints. Suppose that further, the constraint `sat(B+C)` is posted. This expression contains variables from disjoint BDDs. Hence, it is necessary to build a new BDD that corresponds to the totality of all involved constraints. We observe that posting several successive constraints that give rise to BDDs corresponds to a *conjunction* of SAT formulas. In this case, such a newly formed BDD must therefore correspond to the *conjunction* `(A+B)*(C+D)*(B+C)`. We thus build the BDD of that conjunction (recall that each original formula is accessible via the *root* of each involved BDD), and then assign the result to each of the involved variables. This retains the invariant that each CLP($\mathcal{B}$) variable belongs to exactly one BDD. This is readily generalized to $(n+1)$-fold conjunctions in cases where variables from n ($n \geq 2$) disjoint BDDs are involved. The combined symbolic conjunction is assigned to the *root* of the new BDD, since this conjunction is the corresponding formula.

We now turn to (b), i.e., variable *aliasing*. When two logical variables that both participate in CLP($\mathcal{B}$) constraints are *unified*, then the unification hook (see Section 4.5) is automatically invoked by the Prolog system. When two variables `X` and `Y` are unified, we first assign them the same *index* (for example, that of `X`), so that they become truly indistinguishable. Note that this may destroy the *ordering* of any BDDs, and we must take steps to restore the ordering. For this, we distinguish two cases: First, suppose `X` and `Y` are already part of the *same* BDD. In that case, we need to *rebuild* that BDD. We do this by fetching the formula `Sat` that is associated with the root of `X` (and hence also of `Y`), and build the BDD of `Sat` from scratch, via the synthesis algorithm outlined in Section 3.2. This works because the variables are automatically aliased also in `Sat` due to the way Prolog handles unifications.

Second, suppose that `X` and `Y` occur in *different* BDDs. These BDDs must be *merged* to form a single BDD in which the newly aliased variables both participate. This is carried out in exactly the same way as explained above.

Using these techniques, we *combine* and *rebuild* BDDs when necessary, to ensure that each CLP($\mathcal{B}$) variable belongs to exactly one BDD, and that all arising BDDs are *ordered* and *reduced*. Every time two or more BDDs are combined due to one of the reasons outlined above, we purge and recreate from scratch the AVL trees that are associated with each branching variable of the new BDD. This is a substitute for *garbage collection* as suggested in [15]: By recreating these AVL trees, we automatically free memory that would otherwise be used to keep track of nodes that no longer occur in any BDD. The cost of each such cycle is $O(N \log N)$, where N is the number of existing nodes. Note that we rely on the underlying Prolog system's garbage collector to reclaim this memory.

When a variable `V` is unified with a concrete Boolan *value*, i.e., `0` or `1`, the Prolog system likewise calls the unification hook. In that case, we compute the so-called *restriction* of the BDD f to which the variable belongs: This means the BDD where each occurrence of the node (`V`$\rightarrow f_i$ `;` f_j) is replaced by either f_i or f_j, depending on the truth value of `V`. When computing the restriction of a BDD, we use a fresh AVL tree to store nodes whose restriction has already been computed. The *keys* of this AVL tree are the unique node IDs, and the associated *values* are the corresponding results of computing the restriction. This means that such an assignment is performed in $O(B(f) \log(B(f)))$, again incurring a logarithmic factor over asymptotically optimal performance.

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

For fixed variable order and Boolean function, the corresponding BDD is *canonical*. Hence, as long as all BDDs that represent the posted constraints are different from the single leaf **false**, there is at least one admissible solution. In our system, after a BDD is constructed, we always check whether it is identical to **false** (which, as explained in Section 4.6, is represented by `0`), and automatically *fail* if that is the case. Therefore, in our system, when a constraint *succeeds*, there is at least one solution. Appendix B defines these notions in depth.

In addition, the well-known and general notion of (global) *consistency* as defined in [26] is of course equally applicable to CLP($\mathcal{B}$): A Boolean constraint satisfaction problem is *consistent* iff all variables that admit only a single truth value are assigned that value. In other words, there must not be remaining domain elements that do *not* participate in any solution. For example, when posting the constraint `sat(X*Y + ~X*Y)`, then a consistent CLP($\mathcal{B}$) solver must yield the unification `Y = 1`. We implement this notion of 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 global consistency, and seems to implement an undocumented additional property that, for lack of an established terminology (see also [27]), 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 at least one branch of the BDD that leads to **true**. It is easy to see that if a variable is skipped in a branch that leads to **true**, then it can assume both possible truth values, and cannot be involved in any aliasing. In the following, only the collected variables are considered. Among these variables, we can easily detect those that admit only a single satisfying assignment: These are all branching variables V for which all nodes in which they occur are of the same shape, and either *all* of them are ($V\rightarrow$ **false** ; *any*) or *all* of them are ($V\rightarrow$ *any* ; **false**). Such a variable is *necessarily* identical to 0 (in the first case) or 1 (in the second case). In our system, these assignments are automatically materialized by means of unification.

The remaining variables are further classified into: (1) variables that do not have **true** as any child in any node, and (2) variables that have **false** as one child in all nodes. It is easy to see that any potential aliasing between different variables must involve one variable from category (1), and one variable from category (2). Therefore, in a single sweep of the BDD, we have reduced the task of finding potential aliasings from $O(n^2)$ (in the number n of variables) to considering pairs of variables of *only* the described types, which we then iteratively test.

We have tested the impact of enabling global 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. See also Section 6.3.

The combination of these two consistency notions ensures the following interesting property: *All entailed unifications* between CLP($\mathcal{B}$) variables and truth values appear *explicitly* in answers to queries. An interesting application[6] of this property is the derivation of *all entailed constraints* also in other domains, such as described in [28]. By "entailed", we mean that these unifications and constraints *necessarily hold* in all solutions.

4.8. Quantification of variables

Existential quantification of a variable v is expressed as the *disjunction* of two BDD restrictions (see Section 4.6), in which v is respectively **true** and **false**.

Further, recall from Section 4.1 that Prolog *atoms* denote universally quantified variables in CLP($\mathcal{B}$) expressions, and the quantifiers implicitly appear *in front* of the entire expression.

As shown in Section 4.3, an important use case of universally quantified variables is to indicate intended *input* variables of Boolean functions. Full propagation would lead to unintended unifications in such cases and prevent users from seeing such functional dependencies in answers. Therefore, during constraint propagation, we internally treat universally quantified variables in such a way that they do not lead to any additional domain restrictions on other variables.

4.9. Summary of implementation options and trade-offs

We briefly recall the different available implementation options, and summarize the trade-offs we have made in the preceding sections. When implementing a CLP($\mathcal{B}$) system, we have to make the following key decisions:

- **implementation language**: In particular, should we use Prolog or a different programming language to implement the system?

[6]Taus Brock-Nannestad recognized and mentioned this application in personal communication with the author.

- **algorithmic approach**: Should we base the solver on decision diagrams, or use approaches that are typically used in SAT solvers, or choose a different approach altogether? See Section 4.3.

- **data structures**: Should we use BDDs or other types of decision diagrams? Should we use hashes, AVL trees or other data structures?

- **algebraic guarantees**: In particular: Which *consistency notion* should we guarantee? See Section 4.7 and Section 6.3.

Our aim was to provide a *simple* and sufficiently *efficient* CLP($\mathcal{B}$) system with the eventual goal of running GUPU in a completely free environment. Therefore, we have chosen a Prolog-based implementation to facilitate prototyping and to keep the system simple, even though a C-based implementation would likely be faster. As one major benefit of this choice, our CLP($\mathcal{B}$) system is *portable* to other Prolog systems (see Section 6.4).

As the primary data structures in our current system, we are using ordered and reduced BDDs. This turned out to be a rather versatile choice, and BDDs are sufficiently fast on a range of benchmarks. The native CLP($\mathcal{B}$) system of SICStus Prolog uses a variant of these BDDs, also allowing negated nodes. This may be an advantage in certain benchmarks, but also comes with additional complexities. Therefore, to keep the system as simple as possible, we have chosen the most basic BDDs as they were originally introduced. However, many other types of decision diagrams also appear in the literature and could be useful for various types of tasks. ZDDs [29] are a notable example.

As explained in Section 3.2, we are using AVL trees to store nodes and intermediate results. It is also possible to implement *hash tables* in Prolog, for example by using *attributed variables* (Section 4.5). As explained, we have chosen AVL trees because they are a natural fit for Prolog, a common library is already available in many Prolog systems, and the logarithmic overhead is acceptable for our purposes. For comparison, SICStus Prolog uses a dedicated *unique table* that is deeply integrated into the core engine and managed by the system's garbage collector. This is a very efficient, though also a quite involved solution.

The system we describe in this paper is simple, portable and – as we show in Section 6 – even faster than the native CLP($\mathcal{B}$) system of SICStus Prolog on several benchmarks. Therefore, we consider it an acceptable compromise among the available implementation options.

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 Section 4. 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.

The full potential of CLP($\mathcal{B}$) constraints is often realized when solving counting tasks that *lack* a highly regular structure and therefore cannot be solved by more abstract combinatorial considerations. To illustrate this, we use the adjacency map of the contiguous United States and DC as they appear in [30]. Fig. 2 (a) shows the syntax that is used, and Fig. 2 (b) shows our translation to Prolog *facts* of the predicate `edge/2`.

We now use CLP($\mathcal{B}$) constraints to express *independent sets* of this graph. Each node i corresponds to one Boolean variable b_i. An independent set means that for each undirected edge $u - v$ in the graph, b_u and b_v are not both **true**. That is, the following conjunction IND must hold:

```
AL FL                    edge(al, fl).
AL GA                    edge(al, ga).
AL MS                    edge(al, ms).
...                      ...
```

(a) (b)

Figure 2: (a) Adjacency map and (b) corresponding Prolog facts

$$\text{IND} \equiv \bigwedge_{u-v} \neg b_u \vee \neg b_v \tag{1}$$

Fig. 3 shows the *complete* Prolog formulation of IND, using CLP($\mathcal{B}$) constraints and assuming suitable `edge/2` facts. The semantics of `list_to_assoc/2` and `get_assoc/3` are described in Section 4.6. The other used predicates are very common Prolog predicates, and their meaning can be obtained from the SWI documentation. Note that due to commutativity of $\vee$, it suffices to post constraints on the *ordered* pairs as they appear in Fig. 2.

Using our CLP($\mathcal{B}$) system and the program shown in Fig. 3 together with the "US" `edge/2` facts[7], the query `?- independent_set(Sat), sat_count(Sat, Count).` yields the solution `Count = 211954906` within a few seconds.

```
independent_set(*(NBs)) :-
        findall(U-V, edge(U, V), Edges),
        setof(U, V^(member(U-V, Edges);member(V-U, Edges)), Nodes),
        pairs_keys_values(Pairs, Nodes, _),
        list_to_assoc(Pairs, Assoc),
        maplist(not_both(Assoc), Edges, NBs).

not_both(Assoc, U-V, ~BU + ~BV) :-
        get_assoc(U, Assoc, BU),
        get_assoc(V, Assoc, BV).
```

Figure 3: Expressing (1) with CLP($\mathcal{B}$) constraints

5.2. Random solutions

In this section, 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.

[7]All `edge/2` facts are available from: `https://www.metalevel.at/clpb/edges.pl`

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.

We give two concrete examples to clarify the encoding. First, consider the very small case of a 2×2 chessboard. Clearly, only a single triomino can be placed on such a small board, and there are exactly 4 ways to place it. The matrix in Fig. 4 therefore comprises 4 *rows*, because each row corresponds to one way to place a single triomino on the board. An *exact cover* means an assignment to the Boolean variables x_i such that in *each* of the following sets of variables, exactly *one* is set to **true**: $\{x_1, x_2, x_3\}$, $\{x_1, x_2, x_4\}$, $\{x_1, x_3, x_4\}$ and $\{x_2, x_3, x_4\}$. Clearly, this is impossible.

$$\begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix}$$

Figure 4: Matrix (b_{ij}) indicating 4 ways to place a triomino on a 2×2 chessboard

For comparison, Fig. 5 shows the case of a 6×6 board. It involves 148 decision variables, one for each possible placement of a single triomino on the board.

In Fig. 6, 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 so-

$$\begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{148} \end{pmatrix}$$

Figure 5: Matrix (b_{ij}) indicating 148 ways to place a triomino on a 6×6 chessboard

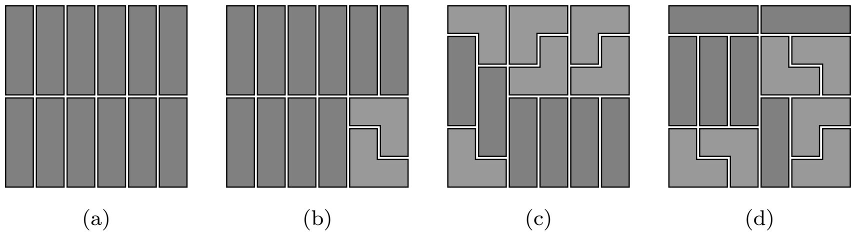


Figure 6: Exact covers of a 6×6 chessboard. (a) and (b) are successive solutions found with CLP(FD) constraints. (c) and (d) are found with CLP($\mathcal{B}$), using random seeds 0 and 1, respectively.

lutions. 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`.

5.3. Weighted solutions

We now use the new interface predicate `weighted_maximum/3` to *maximize* the number of Boolean variables that are **true**.

The first example we use to illustrate this concept is a simple matchsticks puzzle. The initial configuration is shown in Fig. 7 (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. 7 (b), exactly 7 subsquares remain, including the 4×4 outer square. Fig. 7 (c) shows an admissible solution of this task.

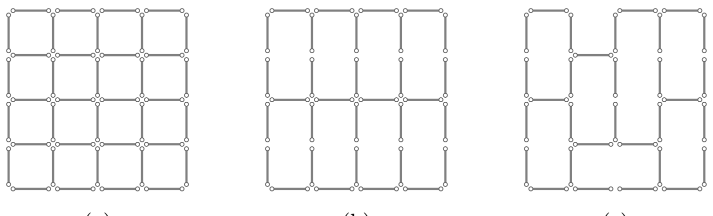


Figure 7: (a) A grid of matchsticks, (b) exactly 7 subsquares remaining and (c) removing the minimum number of matchsticks so that no subsquares remain

Such puzzles are readily formulated[8] 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 [15]), we use CLP($\mathcal{B}$) constraints to express *maximal* independent sets of graphs, which are also called *kernels*. The formulation is based on the constraints described in Section 5.1: Boolean variables b_i again denote whether node i is in the set, and the following constraints KER are posted to enforce *maximal* independent sets, using IND as defined in Eq. (1):

$$\text{KER} \equiv \text{IND} \wedge \bigwedge_{v} (b_v \vee \bigvee_{u-v} b_u) \tag{2}$$

KER can be easily expressed in CLP($\mathcal{B}$) using a suitable extension[9] of the program shown in Fig. 3. 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 rep-

[8]A sample solution is available at: `https://www.metalevel.at/clpb/matchsticks.pl`

[9]The complete program is available at: `https://www.metalevel.at/clpb/kernels.pl`

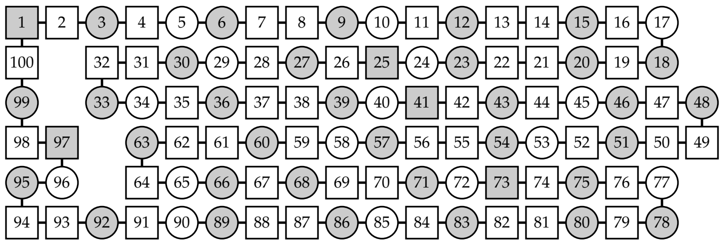


Figure 8: Maximal independent set of C_{100} with maximum weight (= 28)

resentation of i. The grey nodes in Fig. 8 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.

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. Benchmarks

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 7.5.12, 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.

6.1. Benchmark instances

The benchmarks comprise examples from the literature that are also used in [9] and other publications:

`langford` N: Describe a *Langford Sequence* of order N. This is a sequence of the numbers 1, 1, 2, 2, ..., N, N such that the two occurrences of all $k \in \{1, 2, \ldots, N\}$ are k units apart.

`pigeon` N: The task of attempting to place $N+1$ pigeons into N holes in such a way that each hole contains at most one pigeon. Clearly, this problem is unsatisfiable. One interesting property of this task is that it can be formulated with `card/2` constraints alone. Another interesting aspect is that this task has no short resolution refutations in general [31].

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

`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.

The code of all instances is available at `https://www.metalevel.at/clpb/`.

6.2. Benchmark results

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.

These benchmarks are a superset of those we presented in [6]. For the present publication, we have re-run all benchmarks because we have since applied the following changes: First, in order to more accurately measure specific running times, we now create a fresh Prolog process for each of the shown goals (previously, all goals of the same instance were run in the same Prolog process), and second, we have upgraded SWI-Prolog to the latest released version (previously we used 7.3.7). These changes and expected variations over different runs due to task scheduling, memory management and other factors cause small differences in running times between the two publications. For greater precision, we now also state all running times (less than 1 000 seconds) rounded to two decimal places, whereas we previously used only one decimal place in some cases.

We highlight in **bold** the running times of those instances where our system outperforms the native CLP($\mathcal{B}$) solver of SICStus Prolog.

name	*vars*	*clauses*	SWI 7.5.12			SICStus 4.3.2		
			`sat`	`sats`	`taut`	`sat`	`sats`	`taut`
`langford6`	45	18	0.77	**0.81**	0.77	0.01	–	0.01
`langford7`	63	21	3.21	**3.16**	3.22	0.81	–	0.04
`langford8`	84	24	**11.69**	**11.67**	11.79	–	–	0.17
`pigeon8`	72	17	1.12	1.16	1.12	0.08	0.03	0.07
`pigeon9`	90	19	2.91	2.94	2.92	0.23	0.03	0.30
`pigeon10`	110	21	7.49	7.34	7.33	0.92	0.04	0.95
`queens6`	36	302	12.59	12.66	12.73	0.01	2.43	0.01
`queens7`	49	490	65.52	**65.79**	66.64	3.26	20 753	0.04
`queens8`	64	744	388.08	**389.18**	392.32	368.42	–	0.33
`schur13`	39	139	10.54	10.68	10.68	0.28	2.58	0.17
`schur14`	42	161	12.80	12.79	12.81	0.48	8.74	0.49
`schur15`	45	183	17.31	**17.40**	17.54	1.14	28.01	1.17
`triominoes5`	94	25	3.48	**3.54**	3.57	0.01	–	0.03
`triominoes6`	148	36	**21.95**	**21.84**	22.03	–	–	0.09
`triominoes7`	214	49	153.77	**153.02**	152.65	0.63	–	0.62

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

6.3. Performance analysis

There are several things worth pointing out about the results of Section 6.2, and we now analyse them in more detail.

First, it is evident that the native 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. At least in part, this huge difference in performance may 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 and its interface predicates that are used by the solver.

Second, within SICStus Prolog, there is a large relative difference between the `sat` and `taut` columns on one side, and the `sats` column on the other. In the `queens7` case, this difference is particularly pronounced, and it is also observable in almost all other instances. We observe that the native CLP($\mathcal{B}$) system of SICStus Prolog outperforms our library on almost all instances in the `sat` and `taut` columns, but only on 6 of 15 instances in the `sats` column.

The cause of these phenomena is explained in [9], and can be verified by looking into the available Prolog layer of `library(clpb)` that ships with SICStus Prolog: The Prolog part comprises about 700 LOC, and reveals that *preprocessing* is applied if a *conjunction* of formulas is posted via a single `sat/1` or `taut/2` call. In SWI-Prolog, we implicitly post individual `sat/1` constraints if the given formula is a conjunction. At the time of this writing, we do not apply any preprocessing. Therefore, the three columns are almost identical in SWI-Prolog. These results suggest that our system could benefit from applying similar preprocessing. We note though that the `sats` column is arguably more representative of how a constraint solver is typically used in practice, since constraints are often posted one after another instead of all at once.

Third, some of the benchmarks cannot be solved at all with the native CLP($\mathcal{B}$) implementation of SICStus Prolog on this machine: We use "–" to denote an *insufficient memory* exception.

Performance *profiles* are available from `https://www.metalevel.at/clpb/` and show: With SICStus, on the 36 instances that can be run and profiled, on *average* more than 80% of the executed instructions are accounted for by the 3 predicates `clpb:bdd_univ/[4,5,8]`, i.e., `clpb:bdd_univ` with respective arity 4, 5 and 8. In fact, in more than half of the instances, these 3 predicates account for more than 93% of the executed instructions. However, the granularity of profiling SICStus Prolog is restricted by the fact that these predicates internally call built-in C-functions (such as `'$bdd_build'/4` and `'$bdd_parts'/4`) that remain opaque to the system's profiler. The comparatively high memory requirements ($> 47\,\text{GB}$) of SICStus Prolog in some of these instances may hint at opportunities for improvements of internal mechanisms in its CLP($\mathcal{B}$) system.

For comparison, predicates for reasoning about *association lists* (see Section 4.6.3) account, on average, for about one third of the run time of each instance in SWI-Prolog. The rest is dispersed across different predicates.

In SWI-Prolog, *aliasing consistency* (see Section 4.7) incurs less than 25% CPU time overhead in 22 of these 45 instances. On *average*, aliasing consistency incurs an overhead of ca. 50% in each instance. The highest relative overhead is attained in the `queens6` and `schur13` instances, which take almost thrice as long due to aliasing consistency. By removing *all* propagation from the constraint solver, and thus forfeiting both notions of consistency that are explained in Section 4.7, a 3-fold speedup can be achieved on average. Since we value these algebraic properties, we guarantee both notions of consistency in our system.

6.4. Benchmark results with the SICStus port of our system

As already mentioned, our CLP($\mathcal{B}$) system is written entirely in Prolog and is hence *portable* to other Prolog systems that support suitable interfaces for attributed variables. To illustrate this characteristic of our implementation, we have ported the whole system to SICStus Prolog and present the benchmark results of this port in Table 4. We are presenting two sets of times in this table: One with the native `library(assoc)` of SICStus Prolog, and one with a custom `assoc` implementation based on AVL trees, as in SWI-Prolog. The differences between the two sets of benchmarks stem from the fact that `library(assoc)` of SICStus Prolog does not *rebalance* trees upon insertion of elements, and hence accessing an element may take time *linear* in the size of the tree. The $\max_n$ column shows the number of inner nodes of the *largest* BDD that arises while running the respective benchmark instance. In these instances, operations on association lists take, respectively, about 90% and 50% of the time on average.

From these benchmarks, it is clear that our system is often significantly slower than the native CLP($\mathcal{B}$) solver that ships with SICStus Prolog (see Table 3). This is to be expected, because our system is written entirely in Prolog and – as explained in Section 6.3 – does not apply any preprocessing. Nevertheless, our system can solve several benchmark instances that *cannot* be solved with the native SICStus implementation due to insufficient RAM (48 GB) on this machine configuration. We again use **bold** text to indicate where our port outperforms the native CLP($\mathcal{B}$) system of SICStus Prolog.

On average, the SICStus port of our system is already more than 3 times faster than the SWI version, and our goal is to improve it further in the future. For example, we plan to add *preprocessing* as it is applied in the native CLP($\mathcal{B}$) system of SICStus Prolog. The SICStus port is freely available from `https://www.metalevel.at/clpb/`.

7. Testing a CLP($\mathcal{B}$) System

How can we make sure that what we have described in this paper is also what we have actually implemented? On a very high level, we distinguish between *white-box* and *black-box* tests. Tests of the former category assess internal features of the code, whereas tests of the latter category do not.

In this section, we give an example of stating and testing a property that is called *extra-logical* or *meta-logical* because it falls outside the realm of pure logic programs. In this section, we use *black-box* tests to try to find a counterexample of this property. See also Appendix G.

Example: The documentation of `labeling/1` (see Section 4.2) in SICStus Prolog contains the following description: "Enumerates all solutions by backtracking, but creates choicepoints *only if necessary.*" (emphasis ours).

To reason about choicepoints *within* a Prolog program, we can use the predicate `call_cleanup/2`. This predicate takes two arguments, `Goal` and `Cleanup`, and provides the following key feature: `Cleanup` is called after `Goal` *succeeds*

name	$\max_n$	`library(assoc)`			AVL tree `assoc`		
		`sat`	`sats`	`taut`	`sat`	`sats`	`taut`
`langford6`	1 757	0.80	**0.76**	0.85	0.28	**0.24**	0.26
`langford7`	6 045	8.19	**8.01**	7.94	1.12	**1.04**	1.00
`langford8`	22 191	**90.12**	**90.28**	91.94	**3.27**	**3.16**	3.12
`pigeon8`	8 976	3.40	3.66	3.54	0.25	0.35	0.23
`pigeon9`	22 546	21.32	21.57	21.41	0.76	0.75	0.71
`pigeon10`	55 316	127.08	127.84	126.82	1.81	1.89	1.90
`queens6`	956	4.49	4.23	4.27	3.93	3.81	3.61
`queens7`	3 044	14.57	**15.03**	13.93	12.73	**15.74**	23.72
`queens8`	10 726	**75.30**	**80.03**	76.08	**93.35**	**61.05**	53.65
`schur13`	3 107	7.91	7.44	6.55	2.88	2.94	2.72
`schur14`	3 989	9.55	9.70	10.13	3.70	**3.51**	3.86
`schur15`	4 888	15.43	**15.66**	15.45	5.12	**4.93**	5.16
`triominoes5`	4 295	9.81	**9.85**	10.38	1.35	**1.30**	1.44
`triominoes6`	22 103	**280.89**	**286.58**	281.59	**6.27**	**6.34**	6.02
`triominoes7`	103 003	10 199	**10 129**	10 187	46.48	**47.22**	44.05

Table 4: Running times (CPU, in seconds) using the SICStus *port* of our system

deterministically, which means that no more choicepoints remain. The predicate `call_cleanup/2` is available in both SWI-Prolog and SICStus, and we refer to the documentation of these systems for further semantic details. The core idea of our approach is to use `call_cleanup/2` to post the unification `Det=det` as soon as no more choicepoints remain. Hence, if `dif(Det, det)` *succeeds* after the call (`dif/2` means that its arguments are *different*), then a choicepoint remains.

Fig. 9 (a) shows how a subset of all CLP($\mathcal{B}$) expressions (see Section 4.1) can be systematically generated. A DCG[10] is used to generate expressions of

[10]We explain this in our DCG primer at `https://www.metalevel.at/prolog/dcg`

```
sat(a)   --> [].
sat(_)   --> [].
sat(~X)  --> [_], sat(X).
sat(X+Y) --> [_], sat(X), sat(Y).
sat(X#Y) --> [_], sat(X), sat(Y).
```

(a)

```
counterexample(Expr) :-
        length(Ls, _), phrase(sat(Expr), Ls),
        sat(Expr), term_variables(Expr, Vs),
        setof(Vs, labeling(Vs), Sols),
        setof(Vs, labeling_nondet(Vs), Sols).

labeling_nondet(Vs) :-
        call_cleanup(labeling(Vs), Det=true),
        dif(Det, true).
```

(b)

Figure 9: (a) Generating valid CLP($\mathcal{B}$) expressions (b) describing a counterexample to the intended determinism.

bounded depth.

Fig. 9 (b) describes a *counterexample* to the property stated above. We are looking for an expression `Expr` such that the set `Sols` of solutions that are found by exhaustively labeling all variables `Vs` of `Expr` is *equal* to the set of solutions that are found while a choicepoint still *remains*. This means that the remaining choicepoint was in fact *unnecessary*.

Running the code in SICStus Prolog 4.3.2 immediately yields:

```
| ?- counterexample(E).
E = a+ ~_A,
sat(_A=:=_B*a) ?
```

This means that the property in fact does *not* hold, which came as a surprise to us. The same result is also found in our CLP($\mathcal{B}$) system. For this reason, we do not state such a property in the documentation of our system.

Note also that the found expression exemplifies the provision explained in Section 4.8.

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, and can solve several benchmark instances that other systems cannot solve.

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 easily portable to other Prolog systems that support attributed variables, which we have shown by providing a port to SICStus Prolog.

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, improving the performance of the SICStus port, and porting the system to further Prolog systems. Stefan Israelsson Tampe is currently porting the solver to Guile-log, a Prolog system based on Guile. Guile-log is available from: `https://gitlab.com/gule-log/guile-log`.

Additional interface predicates may be needed to cover further applications of BDDs and other types of decision diagrams, such as those that are outlined in [32]. 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.

[1] J. Jaffar, J.-L. Lassez, Constraint Logic Programming, in: POPL, 1987, pp. 111–119.

[2] B. Selman, H. Kautz, B. Cohen, Local search strategies for satisfiability testing, Second DIMACS Implementation Challenge.

[3] J. P. Marques-Silva, Algebraic simplification techniques for propositional satisfiability, in: CP'00, Vol. 1894 of LNCS, 2000.

[4] H. Zhang, SATO: An Efficient Propositional Prover, in: CADE, Vol. 1249 of LNAI, 1997.

[5] J. M. Howe, A. King, A Pearl on SAT and SMT Solving in Prolog, Theoretical Computer Science 435 (2012) 43–55.

[6] M. Triska, The Boolean constraint solver of SWI-Prolog: System description, in: FLOPS, Vol. 9613 of LNCS, 2016, pp. 45–61.

[7] L. Sterling, E. Shapiro, The Art of Prolog (2Nd Ed.): Advanced Programming Techniques, MIT Press, Cambridge, MA, USA, 1994.

[8] M. Dincbas, P. V. Hentenryck, H. Simonis, A. Aggoun, T. Graf, F. Berthier, The constraint logic programming language CHIP, in: FGCS, 1988, pp. 693–702.

[9] M. Carlsson, Boolean Constraints in SICStus Prolog, SICS TR T91:09.

[10] F. Benhamou, Touraïvane, Prolog IV : langage et algorithmes, in: JFPLC, 1995, pp. 51–64.

[11] D. Diaz, S. Abreu, P. Codognet, On the implementation of GNU Prolog, TPLP 12 (1-2) (2012) 253–282.

[12] P. Codognet, D. Diaz, A Simple and Efficient Boolean Solver for Constraint Logic Programming, J. Autom. Reasoning 17 (1) (1996) 97–129.

[13] M. Codish, V. Lagoon, P. J. Stuckey, Logic programming with satisfiability, Theory and Practice of Logic Programming 8 (1) (2008) 121–128.

[14] R. E. Bryant, Graph-Based Algorithms for Boolean Function Manipulation, IEEE Trans. Computers 35 (8) (1986) 677–691.

[15] D. E. Knuth, The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams, 12th Edition, Addison-Wesley Professional, 2009.

[16] P. Tarau, B. Luderman, Boolean Evaluation with a Pairing and Unpairing Function, in: SYNASC 2012, 2012, pp. 384–390.

[17] P. Tarau, Pairing Functions, Boolean Evaluation and Binary Decision Diagrams, CoRR abs/0808.0555. URL `http://arxiv.org/abs/0808.0555`

[18] T. Mantadelis, R. Rocha, A. Kimmig, G. Janssens, Preprocessing Boolean Formulae for BDDs in a Probabilistic Context, in: Proceedings of the 12th European Conference on Logics in Artificial Intelligence, JELIA'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 260–272.

[19] S. Burckel, S. Hoarau, F. Mesnard, U. Neumerkel, cTI: Bottom-up termination inference for logic programs, in: 15. WLP, 2000, pp. 123–134.

[20] S. Colin, F. Mesnard, A. Rauzy, Un module Prolog de mu-calcul booléen: une réalisation par BDD, in: JFPLC'99, Huitièmes Journées Francophones de Programmation Logique et Programmation par Contraintes, 1999, pp. 23–38.

[21] J. Wielemaker, T. Schrijvers, M. Triska, T. Lager, SWI-Prolog, TPLP 12 (1-2) (2012) 67–96.

[22] I. Abío, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, V. Mayer-Eichberger, A new look at bdds for pseudo-boolean constraints, Journal of Artificial Intelligence Research (JAIR) 45 (2012) 443–480.

[23] U. Neumerkel, Teaching Prolog and CLP (tutorial), ICLP.

[24] U. Neumerkel, S. Kral, Declarative program development in Prolog with GUPU, in: Proceedings of the 12th International Workshop on Logic Programming Environments, WLPE, 2002, pp. 77–86.

[25] B. Demoen, Dynamic attributes, their hProlog implementation, and a first evaluation, Report CW 350, Dept. of Computer Science, K.U. Leuven (Oct. 2002).

[26] E. C. Freuder, Synthesizing constraint expressions, Communications of the ACM 21 (11) (1978) 958–966.

[27] J. N. Hooker, Projection, consistency, and George Boole, Constraints 21 (1) (2016) 59–76. `doi:10.1007/s10601-015-9201-2`.

[28] T. Brock-Nannestad, Space-efficient planar acyclicity constraints - A declarative pearl, in: Functional and Logic Programming - 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings, 2016, pp. 94–108.

[29] S. Minato, Zero-suppressed BDDs for set manipulation in combinatorial problems, in: Design Automation Conference (DAC), 1993, pp. 272–277.

[30] D. E. Knuth, The Stanford GraphBase: A Platform for Combinatorial Computing, ACM, New York, NY, USA, 1993.

[31] A. Haken, The intractability of resolution, Theoretical Computer Science 39 (1985) 297–308.

[32] D. Bergman, A. A. Cire, W.-J. v. Hoeve, J. Hooker, Decision Diagrams for Optimization, 1st Edition, Springer Publishing Company, Incorporated, 2016.

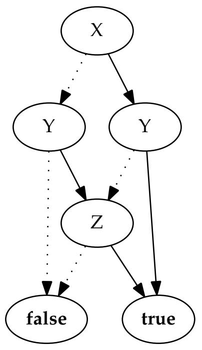
[33] J. Wielemaker, T. Lager, F. Riguzzi, SWISH: swi-prolog for sharing, CoRR abs/1511.00915.

[34] T. Schrijvers, V. Santos Costa, J. Wielemaker, B. Demoen, Logic Programming: 24th International Conference, ICLP 2008 Udine, Italy, December 9-13 2008 Proceedings, Springer Berlin Heidelberg, 2008, Ch. Towards Typed Prolog, pp. 693–697.

Appendix A. Inspecting BDDs in our system

Our system allows *inspection* of BDDs: If the Prolog flag `clpb_residuals` is set to the value `bdd`, then Prolog terms that represent the BDDs are emitted as answers, using pairs of the form `Node-ITE`, where `Node` is a unique node identifier, and `ITE` is an if-then-else node as explained in Section 3.1.

Fig. A.10 shows an example of a BDD and its resulting representation as Prolog terms. In this figure, the dotted lines correspond to $\texttt{V} = 0$. The Boolean function that is represented by this BDD is **true** iff precisely *two* of the three variables `X`, `Y` and `Z` are `1`. In our CLP($\mathcal{B}$) system, this function can be expressed (see Section 4.1) as `card([2],[X,Y,Z])`.



```
?- set_prolog_flag(clpb_residuals, bdd).
true.

?- sat(card([2],[X,Y,Z])).
bdd([node(4)- (X->node(3);node(2)),
     node(2)- (Y->node(0);false),
     node(0)- (Z->true;false),
     node(3)- (Y->node(1);node(0)),
     node(1)- (Z->false;true)]).
```

Figure A.10: A BDD corresponding to the CLP($\mathcal{B}$) constraint `card([2],[X,Y,Z])`

In our system, the *order* of the variables reflects the order in which the variables are first encountered in CLP($\mathcal{B}$) constraints, using a left-to-right, depth-first traversal of the Prolog terms that represent the constraints. Hence, different orders can easily be tried by first using the variables in the desired order in a suitable tautology. The tautology `+[1,V1,V2,...]` is a suitable idiom for enforcing the variable order `V1`, `V2` etc., because it denotes the *disjunction* $\textbf{true} \lor V_1 \lor \cdots$ which is always true, yet assigns suitable indices to the variables. See Section 4.1 for more information about this syntax.

Appendix B. Monotonicity, completeness and incompleteness

In this section, we define several key concepts and use them to explain properties of our system that are important in more specialized applications.

First, let us consider the declarative meaning of Prolog queries. Declaratively, we can read a query as the *question* "Are there any solutions subject to the stated constraints?" In queries, logical variables are thus *existentially* quantified.

In response to a (terminating) query, the Prolog system provides one or more *answers*. Each answer describes a set of *solutions*. The most important answers are:

- `true`
- `false`
- ground solutions
- *residual goals* that indicate constraints that are still pending.

There are also other answers such as *errors* for various exceptional situations. We say that a query *fails* if the answer is `false` ("no" etc. in some systems), and we say it *succeeds* if the answer is `true` ("yes" etc. in some systems) or a concrete solution is reported. If there are alternatives, the system is said to leave a *choicepoint* after reporting the first answer.

If the system reports *residual goals*, then these goals are said to *flounder*. These goals may be *satisfiable* or *unsatisfiable*. When residual goals are reported, we also say that the query *succeeds conditionally*. The notion of residual goals is especially important in the context of constraint solvers, and also motivates the following definition:

Definition (completeness): A CLP system is called *complete iff* the following property holds: If residual goals are reported, then they are satisfiable.

In general, constraint systems do *not* satisfy this property. Instead, they typically require an explicit additional step that uses exhaustive *search* to determine whether the residual goals admit a solution. We call such systems *incomplete*, alluding to the fact that constraint propagation has not derived the ultimate conclusion (which is unsatisfiability of the whole query) from the posted constraints. Note that the words *complete* and *incomplete* may also be used in a different context to denote whether or not a predicate *reports all solutions that exist*.

We now define an important class of Prolog programs that we call *monotonic* programs.

Definition (monotonicity): A Prolog program P is called *monotonic iff* the following properties both hold for all queries Q that can be asked over P: (1) If Q *fails*, then adding further constraints *cannot* make it succeed. (2) If Q *succeeds*, then removing any constraint from it *cannot* make it fail.

Monotonicity is of great importance in some applications. For example, various search strategies such as *iterative deepening* rely on it and will in general not work as intended over logic programs without it. Also, *declarative debugging*[11] in the form of automated generalizations and specializations relies on this property.

To the best of our knowledge, all currently available CLP($\mathcal{B}$) systems are *not monotonic*. For instance, in available systems, *adding* a constraint can yield *new* solutions, which clearly runs counter to the declarative semantics we expect from constraints. Consider for example the following queries and answers with the CLP($\mathcal{B}$) system of SICStus Prolog:

```
| ?-          sat(X), X = 1+1.
no
| ?- X = 1+1, sat(X), X = 1+1.
X = 1+1 ? ;
```

From a logical point of view, this behaviour is highly problematic: It runs counter to our expectation that making a query more general can at most *in-*

[11]For more information, please see: `https://www.metalevel.at/prolog/debugging`

crease the set of solutions. The reason for this behaviour is the so called *defaulty* representation of SAT formulas: When a *variable* is encountered in CLP($\mathcal{B}$) expressions, the constraint system impurely *commits* to regarding it as standing for a concrete Boolean value.

For compatibility with SICStus Prolog and for convenience, this is also the behaviour we have adopted. However, as of SWI-Prolog 7.3.20, ours is the first CLP($\mathcal{B}$) system that supports a *monotonic* execution mode in addition to the non-monotonic default mode. In our system, the monotonic execution mode is enabled by setting the Prolog flag `clpb_monotonic` to `true`. In the SICStus port, the mode is enabled by asserting the fact `clpb:monotonic`. The setting influences a specific and comparatively easily assessable source fragment of our solver, which is executed when CLP($\mathcal{B}$) expressions are being parsed.

When the monotonic execution mode is enabled, then the system yields an *instantiation error* if the SAT formula is not yet sufficiently instantiated. In the cases shown above, we get respectively:

```
?-           sat(X), X = 1+1.
ERROR: Arguments are not sufficiently instantiated

?-  X = 1+1, sat(X), X = 1+1.
X = 1+1.
```

In the monotonic execution mode, we need a way to explicitly mark variables that stand for *concrete* Boolean truth values. Only for such variables, further inferences can be soundly derived. In the monotonic execution mode, you have to write `v(X)` instead of `X` to denote the logical variable `X` that stands for a concrete Boolean value. With this syntax, the issue above cannot arise.

If we consistently use this wrapper for variables that occur in CLP($\mathcal{B}$) expressions, then adding constraints cannot yield new solutions, and removing constraints can at most yield more solutions.

Appendix C. Design principles for new interface predicates

The new interface predicates described in Section 4.4 are not yet found in any other CLP($\mathcal{B}$) system, and we were therefore free to devise their semantics as we saw fit. There are two general design principles that we applied, and which we now illustrate by using the two predicates `sat_count/2` and `weighted_maximum/3` as examples.

The first design principle we applied was to preserve, as far as possible, the *relational* nature of user programs. Enabling the monotonic execution mode that we explained in Appendix B is a necessary precondition for this property. As explained in that appendix, not even `sat/1` is a true relation in the default execution mode.

From this principle, it already follows that `weighted_maximum/3` must be *complete*. This means that, as described in Section 4.4, it generates *all* solutions that attain the optimum. When implementing such an optimizing predicate and designing its interface, it is tempting to impurely *commit* to one of several optima. In fact, we see from CLP(FD) systems and their optimization predicates that committing to a single solution is the behaviour that is *most widely* implemented.

However, it is sometimes impossible to attain logical purity in this sense. For example, `sat_count/2` is intrinsically non-monotonic, since it refers to the extra-logical *number* of admissible assignments subject to the *currently posted* constraints. For this reason, we cannot expect to preserve *commutativity* of conjunction when `sat_count/2` is one of the goals. We can still approximate monotonicity even in this case though, and attain the goal of preserving declarative properties as far as possible: In the case of `sat_count/2`, it is easy to see that (again, assuming that the monotonic execution mode is enabled) *adding* a constraint can at most *reduce*, never increase the number of solutions that are reported.

Observation: Extending the number of variable assignments that make the posted constraints succeed implies that additional variables are introduced by means of unifications like `Var = (X+Y)`, where `X` and `Y` are fresh variables. Such expressions are not valid CLP($\mathcal{B}$) expressions if they are wrapped with the dedicated `v/1` functor, so the term `v(X+Y)` yields a *domain error* if it occurs in `sat_count/2` in place of `v(Var)`. $\square$

The second design principle we applied was to increase the versatility of interface predicates as far as possible.

Let us consider again the semantics of `sat_count/2`. When thinking about a suitable semantics of `sat_count/2` or a similar predicate, implementors of CLP($\mathcal{B}$) systems may at first glance be tempted to somehow refer to *the BDD corresponding to* `Expr`. There are at least two reasons for this: First, it would simplify the implementation of such interface predicates. Second, many text books and publications that outline similar computations on BDDs almost necessarily allude to BDDs when discussing the implementation of this operation, so it is tempting to adopt a similar BDD-centric view also in interface predicates of a CLP($\mathcal{B}$) system.

We have consciously chosen *not* to do this, and instead adopted a view that is based solely on the semantics of the formula as it occurs in the first argument of this interface predicate. Note that, in comparison with the BDD-centric view, this incurs an overhead in the implementation and execution of the interface predicate. Let S denote the set of variables that occur in the same BDD as one of the variables in `Expr`, but do not themselves occur in `Expr`. To compute the number of solutions, `sat_count/2` first constructs a BDD for the conjunction of `Expr` and all constraints posted so far. Before actually counting the number of solutions via this BDD, the variables in S must be projected away.

Suppose, on the other hand, that we adopt the BDD-centric view. Then `sat_count/2` can be implemented without even *computing* S, let alone projecting away such variables.

The reason we have chosen the former semantics over the latter becomes clear after a bit of reflection: Given the former semantics, the latter can be easily obtained *by users themselves.* One way to do this is to simply involve the variables of S also in `Expr`, in such a way that it does not change the number of solutions. Most users will easily find a way to do this. The reverse does *not* hold: Many users who are interested in CLP($\mathcal{B}$) constraints may find it exceedingly hard to selectively quantify variables existentially in the way described above, although the semantics described above are useful for them in many cases.

A concrete example will make this clear. Consider the following Prolog query and answer:

```
?- sat(A+B), sat_count(B+C, Count).
Count = 3,
sat(A=\=A*B#B).
```

Here, $S = \{$`A`$\}$, and therefore `A` must be projected away in the BDD before counting solutions. If `A` were *not* projected away, we would obtain 5 solutions instead of 3, because the expression `(A+B)*(B+C)` admits 5 satisfying assignments. We can easily see this by incorporating `A` in the expression in such a way that the new subformula is a tautology:

```
?- sat(A+B), sat_count((1+A)*(B+C), Count).
Count = 5,
sat(A=\=A*B#B).
```

Adding appropriate tautologies to the formula is easy to accomplish, whereas projecting away exactly the right variables is not as easy for many practitioners.

In the future, when deciding between semantic variants of further interface predicates, we will again apply the design principle derived from the above considerations to guide us.

Appendix D. Boolean logic, SAT and BDDs

CLP($\mathcal{B}$) constraints and their implementation using BDDs are found at the intersection of several topics that are both of great interest and comparatively easy to understand for students of logic and computer science. This makes CLP($\mathcal{B}$) constraints particularly well suited as supplementary teaching material when introducing students to Boolean logic, expressiveness of SAT formulas, digital circuits, decision diagrams and several other topics that arise in this context.

We illustrate this important didactic application of CLP($\mathcal{B}$) constraints by means of a simple logic puzzle that appears in Raymond Smullyan's *What Is the Name of this Book* and Maurice Kraitchik's *Mathematical Recreations*. This example also serves to illustrate and reinforce the concepts that are mentioned in this paper, such as global consistency.

Task: You are on an island where every inhabitant is either a knight or a knave. Knights always tell the truth, and knaves always lie. You meet 3 inhabitants. **A** says: "**B** is a knave." **B** says: "**A** and **C** are of the same kind." What is **C**?

We use Boolean variables `A`, `B` and `C` to represent the inhabitants. Each variable is `1` iff the respective inhabitant is a knight. The task can be formulated as the conjunction of two CLP($\mathcal{B}$) goals that relate the statements to inhabitants:

```
?- sat(A =:= ~B), sat(B =:= (A=:=C)).
C = 0,
sat(A=\=B).
```

Note that no search via `labeling/1` is necessary in this example: The consistency method described in Section 4.7 has deduced that **C** is a *knave*. The residual goals also show that **A** and **B** are different kinds of inhabitants.

To further help teaching Boolean constraints and BDDs, we have implemented a *BDD renderer* in the web-based SWISH platform [33]. The renderer allows all users to easily create and export drawings of BDDs.

Appendix E. Attributed variables: Properties and Desiderata

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. An excellent comparison of the similarities and differences between the two interfaces is contained in [25]. Most of the commonly desired functionality of attributed variables can be expressed with both interfaces, but one particularly striking difference remains: In SWI-Prolog, `attr_unify_hook/2` (see Section 4.5) is called *with* all bindings in place after a unification, whereas in SICStus Prolog, unifications are *undone* before an analogous interface predicate (`verify_attributes/3`) is called.

During the implementation of CLP(FD) and CLP($\mathcal{B}$) systems, we have collected considerable implementation experience with both interfaces, and, from the available alternatives, endorse the SICStus interface and its greater generality for the following important reason: The interface used in SWI-Prolog is not general enough to express important classes of constraint solvers. We substantiate this statement with a detailed example in Appendix F.2.

In addition to its limited generality, the interface used in SWI-Prolog also makes reasoning about unifications more 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.

Based on these considerations, we formulate:

Desideratum 1: Attributed variable interfaces must allow reasoning at a point where the unifications are *not* (yet) in place.

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.

In both SWI-Prolog and SICStus, there is a predicate called `copy_term/3`, which allows us to explicitly reason about the constraints an attributed variable is involved in. In SWI-Prolog, the third argument is a *list* of goals, whereas in SICStus Prolog, it is either a single goal *or* a conjunction of several goals. In this case, we prefer the SWI-Prolog variant of `copy_term/3`, because it makes it easier to symbolically reason about such goals. We therefore formulate:

Desideratum 2: Goals stemming from attributed variables should be made accessible as *lists* of goals.

Obviously, a simple wrapper predicate can be used to obtain this behaviour in both system, but it is still worth pointing out the desired behaviour for future implementors of attributed variable interfaces.

Appendix F. 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.

Appendix F.1. Zero-suppressed Binary Decision Diagrams

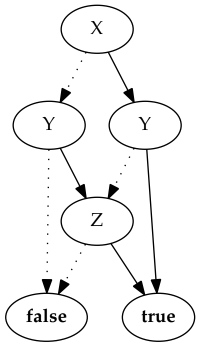
To collect preliminary experiences with different implementation variants, we have created a variant[12] of `library(clpb)` that is based on Zero-suppressed Binary Decision Diagrams (ZDDs). The key idea of ZDDs [29] is to assign a slightly different meaning to the diagram: In ZDDs, a branch leading to `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. This is the case in many covering and tiling tasks such as the one shown in Section 5.2.

As an illustrating example, Fig. F.11 shows a ZDD that represents the same Boolean function whose BDD is shown in Fig. A.10: `card([2],[X,Y,Z])` is true iff precisely two of the three variables are `1`. Note that due to the new conventions, fewer internal nodes are needed in this case. This is of course not always the case. Although one can show (see [15]) that the relative difference between the number of nodes in a BDD and a ZDD that represent the same function cannot be arbitrarily far apart, the choice of either representation is often a significant performance factor in practice.

Appendix F.2. ZDD-based implementation

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

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

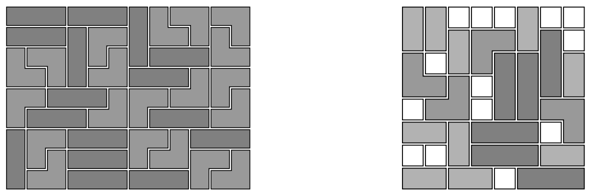


```
?- Vs = [X,Y,Z],
   zdd_set_vars(Vs),
   sat(card([2],Vs)).
Vs = [X, Y, Z],
zdd([node(11)- (X->node(10);node(9)),
     node(9)- (Y->node(6);false),
     node(6)- (Z->true;false),
     node(10)- (Y->true;node(6))]).
```

Figure F.11: A ZDD corresponding to the CLP($\mathcal{B}$) constraint `card([2],[X,Y,Z])`

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. This shortcoming is surmountable in principle, for example by *rebuilding* existing ZDDs when a new variable is encountered.

The second reason is that the shortcomings of SWI-Prolog's interface predicates for attributed variables are especially severe when ZDDs are involved. One way to see this is to consider the ZDD shown in Fig. F.11 and the following conjunction of goals: `X=1, Y=1, Z=1`. After the first two unifications (`X=1,Y=1`), the ZDD shows that only `Z=0` can still make the Boolean function `true`, because `Z` is a variable that does not occur on this path. We therefore expect a ZDD-based CLP($\mathcal{B}$) system to deduce `Z=0`, and therefore the third unification (`Z=1`) to `fail`. Note in particular that it is necessary to reason about whether *variables* (still) occur in the ZDD. Suppose, on the other hand, that we had posted all unifications equivalently in the form of a single simultaneous unification `[X,Y,Z]=[1,1,1]`. With the interface as used in SWI-Prolog, we can no longer explicitly reason about the variables occurring in the ZDD, because at the time the unification hook is called, all variables are already instantiated to ground integers in this example. To the best of our knowledge, we are the



(a) (b)

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

first implementors to consider ZDD-based reasoning with attributed variables, and therefore also the first to find this shortcoming of the SWI interface.

Appendix F.3. Applications of a ZDD-based CLP($\mathcal{B}$) system

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. F.12 (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. F.12 (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 [15], and we plan to make many of them accessible in future versions of this library variant. This may require suitable additional interface predicates.

Appendix G. Further approaches for testing a CLP($\mathcal{B}$) system

In this appendix, we outline two further approaches that we applied for *testing* our CLP($\mathcal{B}$) implementation. This augments Section 7 with more detailed information that may be valuable for system implementors.

Appendix G.1. Testing properties dynamically

Prolog is a very dynamic language and does not provide static type checking as a built-in feature. Nevertheless, Prolog code can be very easily[13] inspected and analyzed, and it is possible to build libraries that augment Prolog with static type checking (see [34] for an example of this approach).

However, the properties we would like to formulate and test go beyond verifying commonly available types. For example, in our CLP($\mathcal{B}$) system, one of the most important properties is without doubt that all BDDs must be *ordered* and *reduced*, as explained in Section 3.1. Ensuring that this property holds throughout all BDD transformations is currently out of reach for static analysis tools. Therefore, we test such important properties *dynamically*: If the Prolog flag `clpb_validation` is set to `true`, then our system tests at *run time* whether:

- every BDD is ordered and reduced
- all occurring BDD nodes are correctly registered in their branching variable's association list (see Section 4.6).

Such dynamic tests were extremely helpful during development of our system. We have found several cases where at least one of the desired invariants was violated due to mistakes in our implementation. During development, we always used our system with all dynamic tests enabled. Of course, executing such (white-box) tests at run time may slow down the solver considerably, and therefore these tests are by default turned off in the released version of our system.

[13]Prolog *code* is naturally represented as a Prolog *term*; for this reason, Prolog is a so-called *homoiconic* language.

Appendix G.2. Black-box tests of interface predicates

We now turn to *black-box* tests of the interface predicates that are explained in Section 4.2. In contrast to the approach described in the previous section, the focus is now on properties that can be observed without analyzing or changing the source code of the system in any way. For concreteness, let us consider `sat/1` as an important representative of our system's interface predicates. How can we make sure that `sat/1` behaves as described in this paper? We distinguish two approaches:

1. test against a reference implementation such as SICStus Prolog
2. test invariants within the same system.

Option (1) is clear: We simply try out `sat/1` with different arguments, and compare its results with those of SICStus Prolog. We have run many such test cases and found several discrepancies between the two systems during development. All of them turned out to be mistakes in our system, and we have corrected all mistakes we found. As of this writing, our test cases have been running for several weeks without finding any further discrepancies, making us more confident about our system's correctness.

Option (2) again tests properties that need to be preserved, and the focus is now on properties that can be tested within the same system. For example, one property that is easy to formulate and test is *commutativity* of conjunction, which holds for CLP($\mathcal{B}$) constraints and unifications if the monotonic execution mode described in Appendix B is enabled. Again, we tried to find cases where this property is violated in our system. We found it particularly worthwhile to test commutativity of conjunctions that involve both `labeling/1` and `sat/1` goals, because the available constraints typically work correctly if all variables are already instantiated and serve as a useful reference in this case. During development, we have corrected several mistakes that were found with such tests. As of this writing, no further mistakes have been found after several weeks of computation time.