CLP($\mathcal{PB}$)

A Meta-Interpreter in CLP($\mathcal{R}$)

$\mathcal{P}$eter $\mathcal{B}$arth

MPI–I–92–233 August 1992

Author's Address

Max-Planck-Institut für Informatik
Im Stadtwald
D-6600 Saarbrücken
Peter.Barth@mpi-sb.mpg.de

Acknowledgements

I would like to thank Alexander Bockmayr for his important comments and fruitful discussions.

Abstract

Constraint logic programming is one of the most attractive research areas in logic programming. Due to [JL87] the theoretical foundation of a general constraint logic programming language scheme CLP($\mathcal{X}$) is available. Unfortunately, implementing a CLP($\mathcal{X}$) system for some domain $\mathcal{X}$ is a difficult task. The problematic points are providing a *constraint solver* and ensuring the *incrementality* of the constraint system. We propose here to use an existing CLP system as implementation environment for a new CLP language. We show that under certain conditions we can use the given constraint solver as constraint solver for the new CLP-language. We focus here on prototyping CLP($\mathcal{PB}$), where $\mathcal{PB}$ denotes the structure of pseudo-Boolean functions, in CLP($\mathcal{R}$), where $\mathcal{R}$ denotes the structure of real numbers.

Keywords

Constraint Logic Programming, Operations Research, Pseudo-Boolean Unification, Pseudo-Boolean Optimization, 0-1 Programming

Contents

1 Introduction

Constraint logic programming is one of the most attractive research areas in logic programming. After the theoretical foundation [JL87] of constraint logic programming languages over arbitrary domains $\mathcal{X}$, now the goal is to design CLP-systems with various instances of the computational domain $\mathcal{X}$. Unfortunately, implementing a CLP-system involves a lot of programming effort. Even if we choose a logic programming language like PROLOG there are two new aspects involved by a CLP-system.

- The *constraint solver*:
  Compute the *satisfiability* of a constraint system as well as a *solved form*.
- The *incrementality* of the constraint system :
  Solving of additional constraints must not entail the resolving of old constraints.

These problems are the same for all CLP-systems. A *general* CLP($\mathcal{X}$)-system, where one only needs to specify the constraint solver would be a solution. Unfortunately, up to now no such systems are available. Another approach to implement CLP-systems is to use an existing CLP-system and its constraint solver. This is easily possible under some conditions which are introduced in Sect. 2. Because constraint solving is done by the given constraint solver we can get a prototype with less programming effort. This prototype can later be improved by incorporating special constraint solving algorithms. Let us summarize the advantages of this approach:

- *rapid prototyping*
  - constraint solving is done by the underlying constraint solver
  - incrementality is automatically provided
- *specialized constraint solving algorithms*
  Improvement of the prototype is possible by replacing the calls to the internal constraint solver by special purpose algorithms of the problem domain.

The constraint logic programming language of our interest is CLP($\mathcal{PB}$), where $\mathcal{PB}$ denotes the domain of pseudo-Boolean functions. Many problems from various applications, namely operations research, can be expressed very naturally using pseudo-Boolean functions. For the definition of the language we refer to [Boc91].

The CLP-system we use as programming environment is CLP($\mathcal{R}$), where $\mathcal{R}$ denotes the domain of the real numbers. We refer to [JMSY90] for details.

The organization of the paper is as follows: In Sect. 2 we introduce the problem area and give some general remarks of using CLP-systems as prototyping environment for other CLP-systems. Then we describe in Sect. 3 briefly the two constraint languages CLP($\mathcal{R}$) and CLP($\mathcal{PB}$) and show in Sect. 4 how to express pseudo-Boolean problems in the domain of real numbers. An implementation of the prototype is described in Sect. 5 enhanced by various possible improvements, which are sketched in Sect. 6. Finally we present in Sect. 7 some computational results for solving problems from two different problem domains, followed by the conclusion in Sect. 8.

2 CLP-System Design in CLP-Systems

As already stated, the main problems in implementing CLP-systems are providing the constraint solver as well as ensuring the incrementality of the system. In [JMSY90] it is proposed to use an existing CLP($\mathcal{O}$)-system as implementation language for another CLP($\mathcal{N}$)-system. This seems to be promising if

- the constraints in $\mathcal{N}$ can be expressed as constraints in $\mathcal{O}$.
- the solutions of the transformed constraints in $\mathcal{O}$ can be identified with the original solutions in $\mathcal{N}$.

By using this approach we get rid of the two problems mentioned above.

- We do not need to design a constraint solver, the constraints are solved by the underlying constraint solver.
- Incrementality is therefore automatically provided.

We only need to define *how* constraints in $\mathcal{N}$ can be expressed as constraints in $\mathcal{O}$ and implement a corresponding transformation routine. We precise now the conditions for a structure $\mathcal{N}$ to be *expressible* in $\mathcal{O}$.

The final *state* of the constraint solver is the simplified (solved) set of all constraints collected up to the end of execution. A problem is *solvable* if the final state of the constraint solver (the solution) represents a consistent constraint set in the associated computational domain.

Suppose that there are two computational domains $\mathcal{N}$ and $\mathcal{O}$ and their sets of possible constraints are $C(\mathcal{N})$ and $C(\mathcal{O})$. If there exist functions $\phi : C(\mathcal{N}) \to 2^{C(\mathcal{O})}$ and $\psi : 2^{C(\mathcal{O})} \to 2^{C(\mathcal{N})}$ such that

$$\forall p \in C(\mathcal{N}) : p \text{ is solvable in } \mathcal{N} \iff \phi(p) \text{ is solvable in } \mathcal{O} \quad (1)$$

$$\forall p \in C(\mathcal{N}) : \ S \text{ is solution of } \phi(p) \implies \psi(S) \text{ is solution of } p \quad (2)$$

then $C(\mathcal{N})$ is called *expressible* in $C(\mathcal{O})$.

If we have a CLP($\mathcal{O}$)-system available and $C(\mathcal{N})$ is expressible in $C(\mathcal{O})$, then obviously every CLP($\mathcal{N}$) problem can be expressed as a CLP($\mathcal{O}$) problem using the function ϕ. After obtaining the (unique) solution S by CLP($\mathcal{O}$) we can compute $S' = \psi(S)$ and we obtain a solution of our original problem.

There is one case where the function ϕ can easily be computed. Suppose that the structure $\mathcal{N}$ is a substructure of $\mathcal{O}$ and that the condition $x \in \mathcal{N}$ can be expressed in $\mathcal{O}$. Then ϕ has only to add additional constraints restricting the solution state such that all ground instances lie in $\mathcal{N}$. The function ψ is then simply the identity.

Another well known technique for implementing solvers of hard constraint theories is the *relaxation* of ϕ. We define a new function ϕ' with the property

$$\forall p \in C(\mathcal{N}) : p \text{ is solvable in } \mathcal{N} \implies \phi'(p) \text{ is solvable in } \mathcal{O} \quad (3)$$

It is often possible to define ϕ' such that the resulting problem is much easier than the one produced by ϕ. Unfortunately we loose correctness and may therefore obtain a false solution S. One technique to achieve correctness is to strengthen the result after solving the relaxed problem. Strengthening must then be done by the function ψ. The main reasons for using the relaxation technique are:

- For the relaxed problem type an efficient constraint solver is available.
- Further information often makes the actual problem easier, so the time spent in solving $\phi(p_i)$ for a subproblem p_i can be saved. Strengthening after collecting all available information can be much easier.

The problem when using a relaxation technique in a CLP-system is the possible exploration of unsatisfiable branches (if $\phi'(p)$ satisfiable but $\phi(p)$ is not).

Example: Let $\forall p \in C(\mathcal{N}) : \phi'(p) = \top$, where $\top$ denotes a tautology in $\mathcal{N}$. Therefore every constraint $\phi'(p)$ is satisfied; constraint solving is *delayed* until the first (probably wrong) solution S is obtained.

In the next sections we describe an implementation where $\mathcal{N}$ is the structure of pseudo-Boolean functions $\mathcal{PB}$. The CLP($\mathcal{O}$)-system we use is CLP($\mathcal{R}$), where $\mathcal{R}$ denotes the structure of real numbers. We define a function ϕ and ϕ'. For efficiency reasons we choose to implement the *relaxed* version, using the linear programming relaxation for 0-1 programming and implicit enumeration for strengthening [NW89].

3 The Constraint Languages

3.1 CLP($\mathcal{PB}$)

The constraint logic programming language CLP($\mathcal{PB}$) was first defined in [Boc91]. The language is an instance of the general constraint logic programming scheme CLP($\mathcal{X}$) [JL86, JL87]. The computational domain is the algebraic structure $\mathcal{PB}$ that allows us to handle equations and inequalities between pseudo-Boolean functions.

A *pseudo-Boolean function* is a mapping $g : \{0,1\}^n \to \mathcal{Z}$, where $\mathcal{Z}$ denotes the ring of integer numbers. Every pseudo-Boolean function can be represented by a (not necessary unique) *pseudo-Boolean term*, constructed using the binary functions $+, *, -$, the constants in $\mathcal{Z}$, associated with their usual meaning and the set of boolean variables $\mathcal{V}_\mathcal{B}$. A *pseudo-Boolean constraint* is a set of equality or inequality-relations between pseudo-Boolean functions. For example

$$X_1 + \ldots + X_n = 1 \tag{4}$$

is a pseudo-Boolean constraint, expressing that exactly one of the X_i is 1. Another pseudo-Boolean constraint is the following set of pseudo-Boolean equations and inequalities.

$$\begin{aligned} A + B + C &\geq 1 \\ A + (1 - B) &\geq 1 \\ A + C &= 0 \end{aligned}$$

This pseudo-Boolean constraint is unsatisfiable. Its unsatisfiability corresponds to a refutational proof in propositional logic [Hoo88] for

$$\{\ A \text{ or } B \text{ or } C\ ,\ A \text{ or } \neg B\ \} \models A \text{ or } C.$$

A CLP($\mathcal{PB}$) program and goal is defined in the usual manner of CLP-systems. Additionally, two facilities are provided for pseudo-Boolean optimization (maximization and minimization), also known as *constrained non-linear 0-1 programming*.

Maximize the pseudo-Boolean function f subject to the pseudo-Boolean constraint C, denoted by *maximize(f,C)*, which also can be written $C \wedge$ *maximize(f)*(resp. for minimization).

Solving pseudo-Boolean constraints is proved to be fitting in the CLP-language scheme in [Boc91]. Various methods for solving pseudo-Boolean problems have already been studied, mainly in the area of operations research.

Typical applications of pseudo-Boolean programming include sequencing problems, time-table scheduling, coding theory, plant location [HR68], inter-city traffic [Rhy70], kinetic energy in spin-glass models [KGV83] or supply-support of space-stations [FGGB66]. Pseudo-Boolean constraints can be used in knowledge representation systems, which gives more compact and comprehensive formulas than in propositional logic. Let the X_i in (4) be facts, then this naturally represents the condition, that exactly one X_i must be valid. A corresponding formula in propositional logic is much larger. Replace in (4) $=$ by $\leq$ then you represent the condition, that all the X_i are mutually exclusive. Furthermore conditions like "at least c facts have to be valid" can naturally be written as

$$X_1 + \ldots + X_n \geq c. \tag{5}$$

For the exact definition of CLP($\mathcal{PB}$), pseudo-Boolean constraint solving algorithms and examples we refer to [Boc91].

3.2 CLP($\mathcal{R}$)

The constraint logic programming language CLP($\mathcal{R}$) was first introduced in [JL86]. The domain of computation is the structure $\mathcal{R}$, the real numbers, including the binary functions $+, -, *, /$ and the binary predicates $<, \leq, >, \geq, =$ with their usual meaning. For the definition of the language see [JMSY90].

Various applications have already been written in CLP($\mathcal{R}$) [HMS87, HL88]. This is due to the fact that there exists a powerful implementation of CLP($\mathcal{R}$) [HJM$^+$91], which we use as implementation environment. This implementation is enriched by some meta-programming facilities, which allows to handle constraints symbolically, according to [HMSY89].

4 Pseudo-Boolean Constraints in CLP($\mathcal{R}$)

In this section we show that for every pseudo-Boolean constraint there is a *simple* equivalent expression in the structure $\mathcal{R}$. We show that solving such expressions entails solving of quadratic equalities[1]. We introduce a relaxation of the constraints which can easily be solved by CLP($\mathcal{R}$) and present a variant of a branch and bound algorithm which calculates the solutions.

4.1 Feasibility

We show how to express pseudo-Boolean problems in the structure $\mathcal{R}$. For that, we assume to have a complete constraint solver over $\mathcal{R}$.

[1] which is not provided by the existing CLP-system [HJM$^+$91]. In fact a relaxation approach is used, as all non-linear constraints are delayed. They are solved only if additional information makes them linear, if not strengthening is *not* applied; the system does not give an answer.

Pseudo-Boolean Equalities and Inequalities.

Obviously $\mathcal{PB}$ is a substructure of $\mathcal{R}$. This implies that every pseudo-Boolean term is a well formed term in $\mathcal{R}$ and every pseudo-Boolean constraint is a well formed constraint in $\mathcal{R}$. Let s_R denote the solution space of a pseudo-Boolean constraint c in $\mathcal{R}$ and s_{PB} the solution space in $\mathcal{PB}$. We know that $s_{PB} \subseteq s_R$. The idea is now to restrict the solution space s_R by additional constraints such that only s_{PB} rests valid. This can be done by fixing the possible values of the (pseudo-Boolean) variables occurring in c to 0 or 1, which leads to the definition of ϕ.

As shown in [Boc91], we can transform every pseudo-Boolean constraint into a pseudo-Boolean equation. So we need to consider only pseudo-Boolean equations. Let $s \doteq t$ be a pseudo-Boolean constraint. Let $\mathcal{V} = \mathcal{V}ar(s \doteq t)$ be the set of variables occurring in $s \doteq t$. We define the function ϕ as follows:

$$\phi(s \doteq t) \stackrel{def}{=} \{s \doteq t\} \cup \bigcup_{X \in \mathcal{V}} \{X * (X - 1) \doteq 0\} \tag{6}$$

The constraint $X*(X-1) = 0$ is in $\mathcal{R}$ equivalent to the condition $X \in \{0, 1\}$. So we have restricted the solution space of $s = t$ to lie in $\mathcal{PB}$ and ϕ satisfies (1). Consequently, every pseudo-Boolean problem can be transformed to an equivalent problem in $\mathcal{R}$ having the same solution space.

Pseudo-Boolean Optimization.

Pseudo-Boolean optimization is not so easy transformable because CLP($\mathcal{R}$) has no primitives concerning optimization. So we need to give a proper algorithm for this problem. We use the underlying constraint solver and not one of the symbolic algorithms [HR68, Basic Algorithm]. We consider only minimization (maximization is done similarly).

The problem is to minimize a given Pseudo-Boolean function $F(\vec{X})$[2]. With the algorithm given below we compute the minimum Min of $F(\vec{X})$ and add the constraint $\phi(Min = F(\vec{X}))$ to the constraint system. This does the required minimization. We present now the algorithm which computes the *Minimum*.

Pseudo-Boolean-Minimization in CLP($\mathcal{R}$)

1. Let *CS* be the initial constraint system.
2. Compute $\vec{X^*} \in \{0, 1\}^n$ such that *CS* is satisfied.
3. Let *CS'* be $CS \cup \{F(\vec{X}) \leq F(\vec{X^*}) - 1\}$.
   IF *CS'* is inconsistent THEN STOP; $F(\vec{X^*})$ =*Minimum*
   ELSE $CS = CS'$;GOTO 2.

Correctness and termination of the above given algorithm is obvious. A ground vector $\vec{X^*}$ can be obtained by *(implicit) enumeration* of the possible solutions, which will be described in the next section.

[2]As noted in Sect. 3.1 additional constraints can assumed to be already in the actual constraint set.

4.2 Relaxation Approach

The CLP($\mathcal{R}$)[3] system [JMSY90] we use is only an approximation of a CLP-system as described in [JL86]. The main problem is that we have introduced *quadratic constraints* to express $\forall_{X \in \mathcal{V}_B} : X \in \{0,1\}$, which are not directly solvable by CLP($\mathcal{R}$). In fact they are delayed and because in general we do not add other constraints making them linear, they will never get awoken. So although our problem transformation is correct the only answer we can get by CLP($\mathcal{R}$) is `*** maybe ***` which of course is not false, but does not solve our pseudo-Boolean problem.

Solving nonlinear polynomials over $\mathcal{R}$ is known to be very hard and seems unacceptable, even for a prototype [Hon92, Col75]. So we have decided to use the relaxation technique introduced in Sect. 2, see also [JMSY90, page 19–23].

We define the function

$$\phi'(p) = \{p\} \cup \bigcup_{X \in \mathcal{V}ar(p)} \{0 \leq X \leq 1\} \tag{7}$$

for all pseudo-Boolean equality and inequality constraints p. The resulting problem is known as *linear programming relaxation*. Obviously ϕ' satisfies (3). After having solved the relaxed problem we strengthen the solution by adding the condition $X \in \{0,1\}$ for all boolean variables X. This can easily be expressed as solving the goal `setbit(`X`)`, where `setbit` is defined as

```
setbit(0).
setbit(1).
```

This forces the variables to be boolean and implements a simple branch and bound algorithm.

Note that we have solved a *clause* and not a constraint. This implies that the constraint solver no longer handles the *constraint* of the variables to be boolean[4]. It is the standard PROLOG backtracking mechanism which forces the variables to be boolean. Therefore we *cannot get symbolic solutions*. Nevertheless we can obtain all solutions of the original pseudo-Boolean problem by backtracking. If necessary it is possible to construct the symbolic solution out of the set of possible solutions.

We illustrate the behaviour of the algorithm with an example. Let us solve the pseudo-Boolean constraint

$$X_1 + X_2 + \ldots + X_n = 1$$

Obviously a simply generate and test algorithm has to test 2^n possibilities. Instead the approach presented here can list all solutions in linear time. Suppose that while enumerating, X_1 is instantiated to 1. Then the corresponding constraint set is

$$\{1 + X_2 + \ldots + X_n = 1\} \bigcup_{i=1}^{n} \{0 \leq X_i \leq 1\},$$

which is simplified by the underlying real-number based constraint solver to

$$\{X_2 = X_3 = \ldots = X_n = 0\} \bigcup_{i=1}^{n} \{0 \leq X_i \leq 1\}.$$

[3]From now on we identify CLP($\mathcal{R}$) with the implementation CLP($\mathcal{R}$) 1.1 as described in [HJM$^+$91].

[4]Note that there is no way in $\mathcal{R}$ to express the fact that a variable is boolean beside a quadratic expression. We do not use quadratic expressions and therefore our assumption in Sect. 2, that $x \in \mathcal{PB}$ is expressible in $\mathcal{R}$, fails.

Obviously the whole search space is cut off and instantiating the X_i to 0 is performed in linear time. The same is true for the cases $X_i = 1$, where $i > 1$. Solving 0-1 problems with the above given relaxation approach has been shown to be very efficient [NW89, Hoo88].

We have illustrated how to translate pseudo-Boolean problems into problems in CLP($\mathcal{R}$). This transformation allows us to use the real constraint solver to approximate constraint solving over the discrete domain `BOOL`. Another example using this idea can be found in [HJM+91, page 23–24; smm].

For every pseudo-Boolean constraint p occurring during the execution of a program we

- add the constraint p to the constraint set *as it is*.
- For every (pseudo-Boolean) variable X in p we
  - add the constraints $X \leq 1$ and $X \geq 0$ to the constraint set (the function ϕ')
  - mark the variable X for later treatment by `setbit` (strengthening by ψ).

After executing the pseudo-Boolean program we solve `setbit` for all marked variables and can obtain all possible solutions by backtracking[5].

5 Meta - CLP($\mathcal{PB}$)

We describe now a first implementation of CLP($\mathcal{PB}$) in CLP($\mathcal{R}$). We adapt the meta-interpreter described in [HMSY89] and give new definitions for solving the constraints. No symbolic solutions are given, all possible solutions of the query can be enumerated by backtracking. Pseudo-Boolean optimization is handled as in Sect. 4.

For this section, the reader is assumed to be familiar with PROLOG [CM87] and CLP($\mathcal{R}$) [JMSY90] as well as with meta-programming in general [SS86] and meta-programming in CLP($\mathcal{R}$) [HMSY89]. We refer especially to [HMSY89].

5.1 The Meta-Interpreter

We adapt the meta-interpreter [HMSY89, page 61] for our needs. One major difference is, that for delaying the instantiation of the pseudo-Boolean variables we have to collect them invisible to the user. We distinguish therefore between an *internal meta-interpreter* and a *top-level meta-interpreter*, where the internal meta-interpreter handles the variable collection. The top-level meta-interpreter hides this variable collection and is given by

```
goal(G) :- pbgoal(G,[],PBV),
           bit(PBV).
```

The internal meta-interpreter `pbgoal` is called with the initial empty variable list. All pseudo-Boolean variables occurred while solving the goal `G` are collected in the list `PBV`. Instantiation of all

[5] We do not feel that this is a restriction for *practical* purposes. Once a user has specified a problem, he normally does not want some screens full with the *most general unifier*. In most of the cases he wants to have an applicable solution, that is one of the possible ground instances.

variables in PBV by bit leads to the enumeration of all possible solutions. The predicate bit may be replaced by a predicate yielding symbolic solutions[6].

The internal meta-interpreter pbgoal is defined as in [HMSY89], but additionally handles the variable list in the last two arguments. Variable collection is done in pbconstraint where new pseudo-Boolean variables may occur.

```
pbgoal(true,PBV,PBV).
pbgoal((A,B),PBVin,PBVout) :-
        pbgoal(A,PBVin,PBVtemp),
        pbgoal(B,PBVtemp,PBVout).
pbgoal(minimize(F),PBVin,PBVout) :-
        pbmin(F,PBVin,PBVout).
pbgoal(maximize(F),PBVin,PBVout) :-
        pbmax(F,PBVin,PBVout).
pbgoal(C,PBVin,PBVout) :-
        pbconstraint(C,PBVin,PBVout).
pbgoal(X,PBVin,PBVout) :-
        rule(X,Y),
        pbgoal(Y,PBVin,PBVout).
```

The two additional cases minimize and maximize handle pseudo-Boolean optimization. The syntax[7] of all other possible constraints is identical to the one in CLP($\mathcal{R}$) and can be handled by pbconstraint as in the original meta-interpreter. The last case handles user-defined clauses. The predicate rule behaves like clause in PROLOG, except that arithmetic terms are given syntactically[8] in quoted form [HMSY89]. So rule behaves like quoted_rule in [HMSY89].

5.2 Equality and Inequality Constraints

Equalities and inequalities are handled by pbconstraint. We use the underlying constraint solver of CLP($\mathcal{R}$). Since $\mathcal{PB}$ is a substructure of $\mathcal{R}$ we simply add all equality and inequality constraints to the internal constraint system as in the original meta-interpreter.

```
pbconstraint(A = B,PBVin,PBVout) :-
        eval(A) = eval(B),
        getvarlist(A = B,PBVin,PBVout).
```

The predicate eval [HJM+91] gives syntactic terms their semantics[9] in $\mathcal{R}$. All variables occurring in A and B are Boolean variables. The predicate getvarlist adds these variables to the variable list PBVin yielding PBVout. Moreover, for every new pseudo-Boolean variable V the constraints

[6] for example the predicate setof(Vars,bit(Vars),Table) instantiates Table to the list of all possible solutions and is nothing else than the *table of solutions* [HR68]. A most general pseudo-Boolean unifier can be constructed out of this table.

[7] the negation of a pseudo-Boolean variable $\overline{X}$ has to be written $(1 - X)$.

[8] For example CLP($\mathcal{R}$) cannot distinguish between $3 + 4$, $2 + 5$ and 7. Because this is necessary in some cases, the concept of *quoting* arithmetic terms is introduced. So quote(3+4) reduces to 3 ŧ 4, where ŧ is an *uninterpreted* function symbol. For further information we refer to [HMSY89].

[9] The reverse process of quoting. For example eval(3 ŧ 4) is equivalent to 7.

```
V >= 0,V <= 1
```

are added to the constraint system.

The predicates for `<=` and `>=` are implemented similarly. A minor improvement is done for `<` (resp. `>`).

```
pbconstraint(A < B,PBVin,PBVout) :-
        eval(A) <= eval(B)-1,
        getvarlist(A < B,PBVin,PBVout).
```

Note that we have used our knowledge over the domain $\mathcal{Z}$ while transforming the constraint $S < T$ into $S \leq T - 1$ (resp. for $>$). This yields a smaller search space for the constraint solver, that is some inconsistencies or conclusions, like fixations of variables to 0 or 1, are detected earlier.

5.3 Pseudo-Boolean Optimization

We use the optimization algorithm described in Sect. 4.1.

```
pbmin(F,PBVin,PBVout) :-
      getvarlist(F,PBVin,PBVout),
      min_iterate(F,PBVout,Minimum),
      eval(F) = Minimum.
```

The predicate `min_iterate` implements the branch and bound algorithm which computes the `Minimum`. Note that all constraints collected up to now, as well as the new generated constraints `eval(F) <= LocalMinimum-1` are active at every moment of computation. So `get_value` only produces values, such that all constraints are satisfied.

```
min_iterate(F,PBV,Minimum) :-
      get_value(F,PBV,LocalMinimum),!,
      eval(F) <= LocalMinimum-1,
      min_iterate(F,PBV,Minimum).
min_iterate(F,PBV,Minimum) :-
      get_value(F,PBV,Minimum).
```

In `get_value` the possible solutions of the constraint set are enumerated and the first solution gives a value of `F`. The instantiation of the variables in `Vars` must be temporarily[10]. We have to store the computed result in the global database and undo the variable bindings produced by `bit` with `!,fail`.

```
get_value(F,Vars,Value) :- do_get_value(F,Vars,Value).
get_value(F,Vars,Value) :- value(Value).

do_get_value(F,Vars,_) :-
      retractall(value(X)),
```

[10]Unfortunately the concept of *implication* [PD91] is not available. Then `get_value` would be `get_value(F,Vars,Value) :- bit(Vars) => eval(F) = Value.`.

```
bit(Vars),eval(F) = Value,
fasserta(value(Value)),
!,fail.
```

The predicate `fasserta` is equivalent to `asserta` in PROLOG.

This implements the desired optimization algorithm by implicit enumeration. The implementation can be improved by heuristics which try to find a value near the *Minimum*. The simplest heuristic is trying to instantiate all variables in the objective function having a positive (negative) coefficient with 0 (1) first.

6 Further Improvements

6.1 Linearization of Non-Linear Constraints

As stated in [Boc91] any pseudo-Boolean constraint can be represented in the form

$$a_1 * \vec{P_1} + \ldots + a_n * \vec{P_n} \,\square\, c$$

where a_i and c are integer constants and $\square \in \{=, <=, >=\}$. The $\vec{P_i}$ are products of the form

$$X_1 * \ldots * X_l * \overline{X_{l+1}} * \ldots * \overline{X_k}.$$

Note that $\overline{X_i}$ is an abbreviation for $X_i - 1$.

In order to *linearize* the constraint, we replace every product $\vec{P_i}$ by a new variable Y_i and solve the resulting problem. Obviously the constraint is not solvable if the linearized form is not solvable. So some inconsistencies are directly detected. If the linearized constraint set remains consistent we add the constraints

$$\overline{X_1} + \ldots + \overline{X_l} + X_{l+1} + \ldots + X_k + Y_i \geq 1 \tag{8}$$

$$-(\overline{X_1} + \ldots + \overline{X_l} + X_{l+1} + \ldots + X_k) + k * \overline{Y_i} \geq 0 \tag{9}$$

which are equivalent to [For60]

$$Y_i = X_1 * \ldots * X_l * \overline{X_{l+1}} * \ldots * \overline{X_k} \tag{10}$$

and continue the computation.

We explain briefly the equivalence of {(8),(9)} and (10). If $\vec{P_i} = 1$ then obviously $X_1 = \ldots = X_i = \overline{X_{i+1}} = \ldots = \overline{X_k} = 1$. Therefore $S = \overline{X_1} + \ldots + \overline{X_i} + X_{i+1} + \ldots + X_k = 0$ and (8) forces $Y_i = 1$. On the other side if $\vec{P_i} = 0$ then $k \geq S \geq 1$ and (9) forces $Y_i = 0$. Vice versa if $Y_i = 0$, then $S \geq 1$ is implied by (8), therefore $\vec{P_i} = 0$. On the other side if $Y_i = 1$, then (9) implies $S = 0$, therefore $\vec{P_i} = 1$.

Much work has already be done for the linearization of pseudo-Boolean constraints, which does not introduce new variables [BM84a]. These techniques have to be further investigated and tested.

6.2 Instantiation Ordering

In order to speed up a boolean satisfiability checker [HF90], Jeroslow and Wang [JW90] have designed a variable selection rule, which provides a heuristic on which variable with which value to branch first. They have introduced a weightening function

$$w(S, j, v) = \sum_{k=1}^{\infty} N_{jkv} * 2^{-k} \tag{11}$$

where N_{jkv} is the number of clauses in a set S of clauses having k literals in which x_j occurs positively (if $v = 1$) or negatively (if $v = 0$). Then they branch on the variable x_{j^*} with value v^*, where (j^*, v^*) maximizes $w(S, j, v)$. Roughly spoken they branch such that *many short* clauses will become satisfied.

The Pseudo-Boolean Case.

We consider first linear pseudo-Boolean inequalities, which can be brought to the normal form [HR68]

$$a_1 * X_1^{s_1} + \ldots + a_n * X_n^{s_n} \geq b \tag{12}$$

such that $b \geq a_1 \geq \ldots \geq a_n \geq 1$ ($s_j \in \{0, 1\}$ and $X_j^1 \equiv X_j$, $X_j^0 \equiv \overline{X_j}$). We define now a function

$$g(j, v) = \begin{cases} \frac{a_j}{b} & \text{if } s_j = v \\ 0 & \text{otherwise} \end{cases}$$

which gives a measure of how much the assignment to the variable X_j with value v satisfies (12). Taking this into account in (11) we obtain

$$w_{pb}(S, j, v) = \sum_{k=1}^{\infty} g(j, v) * N_{jkv} * 2^{-k}$$

as new weightening function. Note that the function g gives always 1 or 0 if applied to clausal inequalities ($a_1 = \ldots a_n = b = 1$) and therefore the new weightening function specializes to the one of Jeroslow and Wang for clausal inequalities.

For the weightening function, we can view every equality $f = c$ as a pair of inequalities $f \geq c$ and $f \leq c$ and after linearization we obtain a weightening function for arbitrary pseudo-Boolean constraints.

6.3 Cutting Non-Integer Solutions

Consider again the normal form

$$a_1 * \vec{P}_1 + \ldots + a_n * \vec{P}_n \,\square\, c$$

of a pseudo-Boolean constraint. Let $\gcd(a_i)$ denote the greatest common divisor of all the a_i.

Then we can replace the original constraint by

$$\frac{a_1}{\gcd(a_i)} * \vec{P}_1 + \ldots + \frac{a_n}{\gcd(a_i)} * \vec{P}_n \,\square\, r_{\square}(\frac{c}{\gcd(a_i)}),$$

where $r_{\square}$ is a rounding function depending on $\square$. If $\frac{c}{\gcd(a_i)}$ is an integer value, then we cannot further restrict the search space and $r_{\square}$ is the identity. If $\frac{c}{\gcd(a_i)}$ is not integral and

- $\Box \equiv =$, then the constraint is not solvable.
- $\Box \equiv \geq$, then $r_\Box$ rounds $\frac{c}{\gcd(a_i)}$ up to the next integer value.
- $\Box \equiv \leq$, then $r_\Box$ rounds $\frac{c}{\gcd(a_i)}$ down to the next integer value.

This can significantly reduce the search space. For solving propositional satisfiability problems there is an algorithm using exclusively a *cutting plane* technique [Hoo89]. In this algorithm all valid linear combinations of the actual constraint set are built and with a similar rounding technique stronger constraints are obtained. We illustrate the idea on an example. Let

$$A + B + (1 - C) \geq 1 \tag{13}$$
$$A + C \geq 1 \tag{14}$$

be the constraint set. Then a valid linear combination is $2 * A + B \geq 1$. With the above described method we deduce

$$A + B \geq 1. \tag{15}$$

Note that (15) is the resolvent of (13) and (14).

6.4 Optimization

Alternative approaches for solving the optimization problem are to optimize the linear programming relaxation (for example with the Simplex algorithm) and applying a branch and bound algorithm [NW89]. Other alternatives are symbolic algorithms, such as the *Basic algorithm* described in [HR68, CHJ90] and adaptations of boolean constraint solving algorithms [BES$^+$90].

Much work has already be done in this area which has to be investigated and incorporated in future versions of the prototype.

7 Computational Experience

We have tested the prototype on some test problems for boolean satisfiability checkers [MR91] and for constrained non-linear 0-1 programming [Tah72]. We run all the tests on a Solbourne-SPARC-Server (16 Mips) under Sun-OS-4.1.1.

7.1 Boolean Satisfiability Problems

In [MR91], a large set of boolean satisfiability problems has been tested on various algorithms. We have solved a subset of these problems with our prototype. We represent every clause

$$C = L_1 \vee \ldots \vee L_n$$

as a linear pseudo-Boolean inequality

$$C' = L'_1 + \ldots + L'_n \geq 1,$$

where for negated (positive) literals $L_i = \overline{V_i} (= V_i)$ we set $L'_i = 1 - V_i (= V_i)$. A set of clauses $\{C_1, \ldots, C_n\}$ is satisfiable iff the set of pseudo-Boolean inequalities $\{C'_1, \ldots, C'_n\}$ is satisfiable [Hoo88].

In Table 1 we give the results of the tests. The table is organized as follows. The first three columns report on using no heuristic for instantiating the variables, whereas the next three columns report on using the Jeroslow-Wang like heuristic described in Sect. 6.2. For each problem we report the cputime (**time**) used for finding a solution (or proving the problem to be unsatisfiable). Additionally, the total number of branches while enumerating (**nodes**) and the total number of wrong choices, that is fathomed nodes (**fn**) is reported. Because we have implemented the JW-heuristic in CLP($\mathcal{R}$), it runs rather slow. So we have given in brackets the time used without counting the computation of the heuristic. The last three columns are copied from [MR91]. They report the cputime used on a Appollo-Workstation (4 Mips) for

Jer/Wa a version of the Davis/Puttnam/Loveland algorithm [HF90], using the variable selection heuristic described in Sect. 6.2,

ProIII Prolog-III [Col90] using rational terms and a branch and bound algorithm and

CHIP-2 CHIP [VH89] using the finite domain constraint solver

on a Sun 3/60 (3 Mips). For further information we refer to [MR91].

Taking into account that CLP($\mathcal{PB}$) is not designed to solve satisfiability problems and that the current prototype is written as a meta-interpreter, the results are quite encouraging.

7.2 Constrained Non-Linear 0-1 Problems

We have used the test problems given in [Tah72], which are constrained non-linear 0-1 minimization problems having a special structure. They have been obtained by replacing in linear 0-1 programs each variable X_i by a product P_i of new linear 0-1 variables. We solve the original linear master problem under the additional constraints $X_i = P_i$.

The results are given in Table 2. The table is organized as follows. The first two columns report on using no linearization, that is all constraints are delayed until they become linear. The next two columns report on using the Fortet linearization [For60]. As heuristic for implicit enumeration we use the heuristic as described in Sect 5.3. In the column **iterations** we give the number of iterations until the optimum is reached by the implicit enumeration algorithm. In the column **time** we give the cputime used for solving the problem. The time in brackets is the cputime used for establishing the optimum solution. The fifth column reports the cputime used by the algorithm from [Tah72] (**TAHA**) on a IBM 7040. The last two columns are copied from [HJM89] and report the cputime used by the algorithm HJM [HJM89] with linear objective function (**HJM**) and from [BM84b] (**BM**) on a SUN 3/50 (3 Mips).

It can be seen, that our prototype behaves well on Taha's problems. It is interesting to note, that without linearization the results are much better. It seems that the problems are to small (in the number of variables) to explode exponentially.

8 Conclusion

We have given an implementation of the new constraint logic programming language CLP($\mathcal{PB}$) in CLP($\mathcal{R}$). Emphasis was on the *ease* of prototyping CLP-systems in other CLP-systems under

Table 1: Boolean satisfiability problems

	no heuristic			JW heuristic			Jer/Wa	Chip2	ProIII
Problem	**nodes**	**fn**	**time**	**nodes**	**fn**	**time**	**time**	**time**	**time**
ulmbc024	32	18	2.02	11	1	3.99 (0.9)	0.86	4.54	3.28
ulmbc040	106	52	25.11	20	2	23.04 (6.93)	2.26	9.58	11.95
ulmbc060	863	299	476.73	31	2	86.71 (25.64)	6.93	19.58	397.00
ulmbp048*	39364	30294	312.55	209	69	172.32 (26.68)	1.90	44.74	–
ulmbs040	840	760	24.14	23	2	7.68 (0.35)	0.79	14.62	1.60
ulmbs060	–	–	%	30	2	20.71 (1.0)	1.23	177.6	8.63
ulmbs084	–	–	–	38	4	58.64 (3.23)	2.30	#	87.58
ulm027r0	25	13	0.25	11	1	2.59 (0.21)	0.70	1.9	0.32
ulm027r1	23	12	0.45	23	4	3.77 (0.62)	0.80	2.42	0.70
ulm027r2	25	12	0.22	11	0	2.04 (0.18)	0.70	1.92	0.32
ulm054r0	50	26	0.50	21	1	13.36 (0.51)	1.22	3.84	1.13
ulm054r1	46	24	0.94	78	12	35.31 (1.98)	1.97	18.7	1.55
ulm054r2	50	24	0.54	22	0	11.69 (0.5)	1.20	3.84	1.68
ulm081r0	75	39	0.81	31	1	40.82 (0.7)	2.18	5.7	2.15
ulm081r1	69	36	1.53	181	28	177.1 (6.0)	%	328.7	2.90
ulm081r2	75	36	0.78	33	0	34.86 (0.67)	2.18	5.68	8.47
ulm216r0	200	104	2.52	–	–	%	18.13	15.16	72.98
ulm216r1	184	96	4.20	–	–	%	114.39	#	102.23
ulm216r2	200	96	2.42	–	–	%	17.77	15.12	–
real1a12	18	6	2.4	15	0	8.21 (1.79)	0.80	10.02	6.75
real1b12	15	4	0.35	11	0	1.66 (0.38)	0.63	3.3	0.50
real2a12	18	6	2.5	15	0	8.4 (1.86)	0.80	10.16	4.82
real2b12	15	4	0.51	11	0	1.61 (0.32)	0.63	3.24	0.48
real2c12	17	6	2.67	14	0	11.22 (1.72)	0.85	14.18	36.67
alu1	79	28	8.36	59	1	83.48 (7.71)	2.49	5.36	–
dc1	79	47	10.31	49	1	61.78 (16.27)	2.22	5.9	–
example	9	3	0.26	5	0	0.25 (0.04)	0.57	0.52	–
newtes1	56	9	0.41	52	0	22.88 (0.34)	0.90	2.68	13.53
newtest*	–	–	%	146	85	184.55 (11.59)	0.92	46.8	–
tomstest	35	5	0.23	30	0	5.76 (0.22)	0.66	2.14	3.15
twoinvr1	30	18	0.75	17	0	3.7 (0.43)	0.77	2.06	1.43
twoinvrt*	18	11	2.21	7	1	4.69 (1.49)	1.28	1.58	–

* unsatisfiable problem

program aborted after some time

% program run out of memory

– problem not tested

Table 2: Constrained non-linear 0-1 problems

	no linearization		Fortet linearization		TAHA	HJM	BM
Problem	**iterations**	**time**	**iterations**	**time**	**time**	**time**	**time**
1–A	1	0.06	1	0.29 (0.26)	0.27	0.50	3.34
1–B	1	0.13	1	0.47 (0.43)	0.15	2.74	86.38
1–C	1	10.7	1	1.62 (1.47)	0.23	325.82	303.98
2–A	1	0.26	1	1.15 (0.88)	1.88	1.44	8.06
2–B	1	0.76	1	2.15 (2.01)	0.58	2.66	20.10
2–C	1	0.55	1	2.93 (2.05)	3.33	125.86	1391.26
2–D	2	14.12	2	56.4 (46.24)	1.02	356.90	648.30
2–E	1	0.39	1	0.96 (0.84)	5.02	0.66	3.52
3–A	1	0.39	1	0.54 (0.50)	2.12	0.42	1.04
3–B*	0	0.28	0	0.59	8.83	0.80	3.14
3–C	1	0.37	1	1.01 (0.81)	2.35	0.38	8.12
3–D	2	0.95	1	2.19 (1.66)	3.42	1.10	15.22
3–E	1	1.15	1	2.94 (1.92)	11.29	2.56	36.86

* no solution

certain conditions. The direct transformation of pseudo-Boolean constraints into equivalent constraints in $\mathcal{R}$ has been found to produce too hard problems and we have introduced and applied a relaxation based approach for the transformation.

The work was done to obtain a prototype of the new constraint logic programming language CLP($\mathcal{PB}$). Several examples [Boc91, Sect. 6] [HR68] have been implemented and tested. The different approaches, symbolic constraint solving versus adapted constraint solving mechanisms in $\mathcal{R}$ need now to be compared. The goal is to obtain a powerful (pseudo-)Boolean constraint solver.

On the other side, it is necessary to give some larger applications in CLP($\mathcal{PB}$) which show the usefulness of this language. Equivalences between pseudo-Boolean programming and theorem-proving have been already sketched in [Hoo88] and need to be further investigated.

References

[BES+90] W. Büttner, K. Estenfeld, R. Schmid, H.-A. Schneider, and E. Tidén. Symbolic constraint handling through unification in finite algebras. *Applicable Algebra in Engineering, Communication and Computing*, 1:97–118, 1990.

[BM84a] E. Balas and J. B. Mazzola. Nonlinear 0-1 programming: I. Linearization techniques. *Mathematical Programming*, 30:1–21, 1984.

[BM84b] E. Balas and J. B. Mazzola. Nonlinear 0-1 programming: II. Dominance relations and algorithms. *Mathematical Programming*, 30:22–45, 1984.

[Boc91] A. Bockmayr. Logic programming with pseudo-boolean constraints. Technical Report mpii-91-227, Max-Planck-Institut für Informatik, Saarbrücken, 1991.

[CHJ90] Y. Crama, P. Hansen, and B. Jaumard. The basic algorithm for pseudo-boolean programming revisited. *Discrete Applied Mathematics*, 29:171 – 185, 1990.

[CM87] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer, third rev. and ext. edition, 1987.

[Col75] G. E. Collins. Quantifier elimination for the elementary theory of real closed fields by cylindrical algebraic decomposition. In *Lecture Notes in Computer Science*, volume 33, pages 134–183. Springer-Verlag, 1975.

[Col90] A. Colmerauer. An introduction to Prolog III. *Communications of the ACM*, 33(7):69–90, 1990.

[FGGB66] R.J. Freeman, D.C. Gogerty, G.W. Graves, and R.B.S. Brooks. A mathematical model of supply support for space operations. *Oper. Research*, 14:1–15, 1966.

[For60] R. Fortet. Applications de l'algèbre de boole en recherche opérationelle. *Rev. Française Recherche Opér.*, 4:17–26, 1960.

[HF90] J. Hooker and C. Fedjki. Branch-and-cut solution of inference problems in propositional logic. *Annals of Mathematics and Artificial Intelligence*, 1:123–139, 1990.

[HJM89] P. Hansen, B. Jaumard, and V. Mathon. Constrained nonlinear 0-1 programming. Technical Report RRR 47-89, Rutgers Center for Operations Research, 1989.

[HJM+91] Nevin C. Heintze, Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland Yap. The CLP($\mathcal{R}$) programmer's manual – version 1.1. Technical report, IBM Thomas J Watson Research Center, November 1991.

[HL88] Tien Huynh and Catherine Lassez. A CLP($\mathcal{R}$) options trading analysis system. In *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, pages 59–69, Seattle, Washington, U.S.A., 1988.

[HMS87] Nevin Heintze, Spiro Michaylov, and Peter Stuckey. CLP($\mathcal{R}$) and some electrical engineering problems. In Jean-Louis Lassez, editor, *Logic Programming: Proceedings of the 4th International Conference*, pages 675–703. MIT Press, May 1987.

[HMSY89] Nevin Heintze, Spiro Michaylov, Peter Stuckey, and Roland Yap. On meta-programming in CLP($\mathcal{R}$). In Ewing Lusk and Ross Overbeek, editors, *Logic Programming: Proceedings of the North Amercian Conference, 1989*, pages 52–68. MIT Press, October 1989.

[Hon92] Hoon Hong. Non-linear constraints solving over real numbers in constraint logic programming (introducing RISC-CLP). RISC-Linz Report Series 92-08, Research Institute for Symbolic Computation, Johannes Kepler Institute, A-4040 Linz, Austria, Europe, January 27 1992.

[Hoo88] J. N. Hooker. A quantitative approach to logical inference. *Decision Support Systems*, 4:45 – 69, 1988.

[Hoo89] J. N. Hooker. Input proofs and rank one cutting planes. *ORSA Journal for Computing*, 1:137 – 145, 1989.

[HR68] P.L. Hammer and S. Rudeanu. *Boolean Methods in Operations Research and Related Areas*. Springer-Verlag, 1968.

[JL86] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. Technical Report 86/73, Monash University, Victoria, Australia, June 1986.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages, Munich, Germany*, pages 111–119. ACM, January 1987.

[JMSY90] Joxan Jaffar, Spiro Michaylov, Peter Stuckey, and Roland Yap. The CLP($\mathcal{R}$) language and system. Technical Report RC 16292 (#72336) 11/15/90, IBM Research Division, November 1990.

[JW90] R.E. Jeroslow and J. Wang. Solving propositional satisfiability problems. *Annals of Mathematics and AI*, 1, 1990.

[KGV83] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[MR91] I. Mitterreiter and F.J. Radermacher. Experiments on the running time behaviour of some algorithms solving propositional logic problems. Technical report, Forschungsinstitut für anwendungsorientierte Wissensverarbeitung Ulm, 1991.

[NW89] G. L. Nemhauser and L. A. Wolsey. Integer programming. In G. L. Nemhauser et al., editor, *Optimization*, volume 1 of *Handbooks in Operations Research and Management Science*, chapter VI, pages 447–527. Elsevier, 1989.

[PD91] Frank Pfenning and Scott Dietzen. A declarative alternative to *assert* in logic programming. Draft, 1991.

[Rhy70] J. Rhys. A selection problem of shared fixed costs and networks. *Manag. Science*, 17:200–207, 1970.

[SS86] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1986.

[Tah72] H. A. Taha. A Balasian-based algorithm for zero-one polynomial programming. *Management Science*, 18B:328–343, 1972.

[VH89] Pascal Van Hentenryck. *Contraint satisfaction in logic programming.* MIT Press, 1989.