Modelling Mixed-Integer Optimisation Problems in Constraint Logic Programming

Peter Barth   Alexander Bockmayr

MPI–I–95–2–011   November 1995

Authors' Addresses

Peter Barth, Alexander Bockmayr
Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany
{barth,bockmayr}@mpi-sb.mpg.de

Publication Notes

Acknowledgements

This work was partially supported by the ESPRIT Working Group CCL (contract EP 6028).

Abstract

Constraint logic programming (CLP) has become a promising new technology for solving complex combinatorial problems. In this paper, we investigate how (constraint) logic programming can support the modelling part in solving mixed-integer optimisation problems. First we show that the basic functionality of algebraic modelling languages can be realised very easily in a pure logic programming system like Prolog and that, even without using constraints, various additional features are available. Then we focus on the constraint solving facilities offered by CLP systems. In particular, we explain how the constraint solver of the constraint logic programming language CLP($\mathcal{PB}$) can be used in modelling 0-1 problems.

Keywords

Modelling, Mixed-Integer Optimisation, Constraint Logic Programming

Contents

1 Introduction

Constraint logic programming (CLP) has become a promising new technology for solving complex combinatorial problems. While a lot of research has been dedicated to developing more powerful and efficient constraint solvers, much less effort has been spent to study the role of CLP in modelling. From a practical point of view, however, model building is extremely important. A practitioner does not want to develop a new solver. He would like to use an existing solver and is looking for a tool that allows him to formulate and to solve his problem in the most convenient way. In this paper, we show how (constraint) logic programming can support the modelling process when solving mixed-integer optimisation problems and what the possible benefits are compared to existing modelling systems in mathematical programming.

The organisation of this paper is as follows. In Section 2, we first show how the basic functionalities of algebraic modelling languages can be realised very easily in a logic programming system. Then we discuss what additional features are available in a pure *logic programming* language like PROLOG, even without the use of constraints. In Section 3, we consider *constraint* logic programming over finite domains, CLP($\mathcal{FD}$), and describe various programming language constructs that may allow to model a problem much more compactly than in classical mathematical programming. In Section 4, we focus on modelling 0-1 problems in CLP($\mathcal{PB}$), a constraint logic programming language for pseudo-Boolean constraints. In particular, we show how logical conditions between pseudo-Boolean constraints can be eliminated automatically using the complete constraint solver of this language.

Mathematical programming is much older than programming in the sense of today's computer science. It is time now that the achievements in declarative programming and programming language design are made available for formulating and solving mathematical programming problems. Constraint logic programming is one of the most promising approaches for achieving this goal.

2 Algebraic modelling in logic programming

2.1 Algebraic modelling

Modelling languages are intended to bridge the gap between the modeler's form of a problem, which should be natural and easy to understand by humans, and the algorithm's form, which can be executed on a computer [Fou83]. On the one hand, a modelling language should therefore allow the user to formulate his problem in a natural and declarative way, on the other hand it should be possible to generate from this high-level description automatically a machine-oriented form that can serve as input to a suitable computer package.

Algebraic modelling languages are based on the familiar algebraic notation used in traditional mathematics. They allow the modeler to use standard mathematical notation, for example subscripts and summation, to formulate the objective function and the constraints of his problem. Typical algebraic modelling systems are, for example, AMPL [FGK93] and GAMS [MB93] (see [Sha93] for a guide to modelling software). Williams [Wil93a] lists the following features desirable in any modelling system:

- Separating the data from the statements of the model

```
set PROD;  # products

param rate {PROD} > 0;      # tons produced per hour
param avail >= 0;           # hours available in week
param profit {PROD};        # profit per ton
param market {PROD} >= 0;   # limit on tons sold in week

var Make {p in PROD} >= 0, =< market[p]; # tons produced

maximize total_profit: sum {p in PROD} profit[p] * Make[p];
               # Objective: total profits from all products

subject to Time: sum {p in PROD} (1/rate[p]) * Make[p] =< avail;
               # Constraint: total of hours used by all
               # products may not exceed hours available
```

Figure 1: A simple production model in an algebraic modelling language

- Indexing for variables and parameters
- Summation over index sets
- Relations between indices
- Arithmetic on the coefficients
- Interactive modelling
- Automatic formatting

We illustrate these features by a production model taken from [FGK93]. We are given a number b of hours available, a set of products P and parameters a_j, c_j, and u_j, describing for each $j \in P$ the tons per hour, the profit per ton, and the maximum number of tons of product j. Using variables X_j for the number of tons of product j to be made, our goal is to maximise the objective function

$$\sum_{j \in P} c_j X_j$$

subject to the constraints

$$\sum_{j \in P} (1/a_j) X_j \leq b$$

and

$$0 \leq X_j \leq u_j, \qquad \text{for each } j \in P \ .$$

The corresponding formulation in an algebraic modelling language, here we use AMPL, is given in Figure 1.

2.2 Algebraic modelling in logic programming

We now show how algebraic modelling can be realised very easily in a logic programming language like Prolog.

The general idea of logic programming is to use *logic as programming language*. The most popular logic programming language, Prolog, is based on the Horn fragment of first-order predicate logic. A Horn clause *logic program* is a set of rules of the form

$$A :- B_1, \ldots, B_n$$

with $n \geq 0$. A rule with $n = 0$ is also called a fact. The conclusion A, the head of the rule, and the conditions B_i, the body or premises of the rule, are *logical atoms* of the form $p(t_1, \ldots, t_m)$, $m \geq 0$, where p is a predicate symbol and the t_j are terms, consisting of variables, constants, and function symbols over a suitable signature. The *declarative meaning* of such a rule is given by the universal formula

$$\forall X_1, \ldots, X_k : A \Leftarrow B_1 \wedge \ldots \wedge B_n \ ,$$

where $X_1, \ldots, X_k$ is a list of all the variables occurring in the rule. Given a logic program, one can ask a query of the form

$$?- C_1, \ldots, C_l \ ,$$

which corresponds to the existential formula

$$\exists X_1, \ldots, X_k : C_1 \wedge \ldots \wedge C_l \ .$$

Based on the resolution principle from automated deduction, a logic programming system will then compute an answer substitution

$$X_1 = t_1, \ldots, X_k = t_k$$

such that the query is satisfied. In general, several answer substitutions are possible, which can be enumerated by backtracking (for a comprehensive introduction into logic programming see for example [CM81, SS86]).

Logic programming is a natural candidate for a modelling language in mathematical programming. The modeler can use logic to formulate his problem in a high-level and declarative way. Familiar algebraic notation can be incorporated very easily by suitable Prolog predicates. The logic programming system will then automatically generate the corresponding input for a mathematical programming solver.

In Prolog, a set of rules forms a logic program. But, a set of rules can also be seen as a description of a linear mixed-integer programming problem by exploiting the relational aspects of logic programming. For that, we first show that modelling *instances* or *data* of a specific linear programming model is straightforward in Prolog. Next, we illustrate that the *statements* or *constraints* of a linear programming model can be similarly described. Just the result of a special Prolog query has to be interpreted. Finally, some syntactic sugar is introduced that allows to closely mimic a standard modelling language like Ampl or Gams.

2.2.1 Modelling Data

In algebraic modelling languages, the data of a specific instance of a linear programming model consist of *index sets* and *parameters* that relate an index to a data item, which typically is either a coefficient or a variable.

Index Sets: Index sets represent a set of items in the real world. For example, there might be a set of products consisting of the items `a`, `b`, and `c`. In other words, there is a unary relation `product` containing the three items `a`, `b`, and `c`. Declaring such simple relations in PROLOG is straightforward and for the above example we write

```
product(a).
product(b).
product(c).
```

In PROLOG, a clausal definition of an n-ary predicate `p/n` defines an n-ary relation. In the above example we define the unary relation `product/1`, which contains all possible answer substitutions for X in the query

$$?-\ \texttt{product}(X)\ .$$

Hence, we define the set

$$\{X \mid \ \texttt{product}(X) \text{ is satisfied } \} = \{\texttt{a}, \texttt{b}, \texttt{c}\}\ ,$$

which is exactly what we need in algebraic modelling. Furthermore, we are not restricted to declare facts for index sets, but any valid set of rules can be used to declare such a relation. For example,

```
product(X) :- member(X,[a,b,c]).
```

defines the same relation. The predicate `member` is usually available in any PROLOG system or can be recursively defined by the two rules

```
member(X,[X|_]).
member(X,[_|L]) :- member(X,L).
```

One can use the full power of PROLOG in order to declare arbitrary index sets. For example, the `union` and `intersection` of two binary index sets `p1` and `p2` is defined by

```
union(X,Y) :- p1(X,Y).
union(X,Y) :- p2(X,Y).
intersection(X,Y) :- p1(X,Y),p2(X,Y).
```

For a full treatment of set operations and other programming guidelines for PROLOG see for example [SS86].

Parameters: Parameters in algebraic modelling relate an element of an n-ary index relation to a data item. Hence, a parameter specification declares a function from an n-ary index set to a set of data items. Since we have only relations in Prolog, we represent an n-ary function by an (n+1)-ary relation. For example, assume that we have a cost associated to one quantity of each product. Then there is a function

$$\texttt{cost} : \{a, b, c\} \rightarrow \mathbb{R}$$

which might be defined as follows:

$$\texttt{cost(a)} = 4, \texttt{cost(b)} = 5, \texttt{cost(c)} = 3$$

In our relational language Prolog we model such a function by a binary predicate `cost` as follows:

```
cost(a,4).
cost(b,5).
cost(c,3).
```

Again, any facilities offered by the logic programming system can be used. An equivalent formulation is, for example,

```
cost(X,Y) :- member(X-Y,[a-4,b-5,c-3]).
```

Variables can be seen as a special parameter, where the relation does not specify a numerical value (a coefficient) for the (n+1)-th argument, but a unique variable name. In this sense, there is no conceptual difference between parameters and variables. Hence, index sets, parameters, and variables can be naturally formulated in Prolog without further support.

For checking consistency of a model description one often wants to declare the names of all the index sets as well as the parameter and variable names together with their index sets. Again, we can exploit the relational aspect of Prolog and require that the relation `set/1` defines the set of all set names and that `param/2`, resp. `variable/2`, defines the set of all parameter names, resp. variable names, together with a tuple (or a list) of all its index set names, e.g.,

```
set(product).
param(cost,[product]).
variable(x,[product]).
```

While generating the model, we can then assure that only valid, i.e. specification corresponding, index sets and parameters are generated.

2.2.2 Modelling Constraints

For the specification of the constraints again the relational aspect of logic programming is sufficient. The modeler just has to implement a unary predicate `subject_to` such that each element of the relation `subject_to` is a constraint. The complete set of constraints then is the conjunction of all elements in the relation `subject_to`. For example, one rule of the predicate `subject_to` might be

```
subject_to(CostA*XA =< 100) :- cost(a,CostA),x(a,XA).
```

which expresses that the overall cost of making the quantity XA of product a may not exceed 100. Due to the underlying logic inference mechanism, we get parametric statements for free. The same upper bound for all products is modelled by

```
subject_to(CostP*XP =< 100) :-
        cost(P,CostP),
        x(P,XP).
```

Note that we have replaced the constant a with a variable P. The set of all instances of the above rule together with the cost predicate is

$$\{4 * \mathtt{Xa} =< 100, 5 * \mathtt{Xb} =< 100, 3 * \mathtt{Xc} =< 100\} \ .$$

Of course, we should also allow conjunctions of constraints represented as a list of primitive constraints:

```
subject_to(L) :-
        findall(CostP*XP =< 100,(cost(P,CostP),x(P,XP)),L).
```

Here, we use the PROLOG built-in findall, which computes all instances of the relation defined in the second argument. For each of these instances, the first argument, the template, is instantiated accordingly and the list of all these instances is collected in the third argument. If we want to manipulate a specific set of instances, say for constructing a sum of arithmetic terms, we need the functionality of findall. For the objective function we assume to have a single clause defining the unary relation objective.

Note that up to now, we have not introduced any support in PROLOG and are already able to formulate a (mixed-integer) linear programming model and the corresponding data. It can be constructed by executing the query

```
?- findall(Constraint,subject_to(Constraint),Constraints).
```

which instantiates Constraints to a list of all the constraints specified by the modeler. But, using pure PROLOG in such a way for modelling is not very convenient. Hence, we provide now some primitives that facilitate modelling and additionally integrate support for checking the model. Note, however, that we only add syntactic sugar and that the functionality we need is already present in vanilla PROLOG.

2.2.3 Syntactic Sugar

To modelers the functionality of findall is better known as forall. Hence, we support a predicate forall/3 of the form

$$\mathtt{forall(Goal, Template, L)}$$

where Goal is an arbitrary PROLOG goal, Template is a term, and L is a free variable. All possible instantiations of Template obtained by executing the query Goal are collected in L. Similarly, we provide a predicate sum/3

$$\mathtt{sum(Goal, Template, S)}$$

which is equivalent to forall, except that the third argument S is instantiated not to the list, but to the sum of the template instances. For example,

```
subject_to(S =< 200) :-
        sum((cost(P,CostP),x(P,XP)),CostP*XP,S).
```

ensures that the overall cost of all products is less than or equal to 200.

In algebraic modelling one typically wants to make sure that `Prod` is a member of the unary relation `product`. This can be done with an additional goal in the `Goal` part of `sum`.

```
subject_to(S =< 200) :-
        sum((product(P),cost(P,CostP),x(P,XP)),CostP*XP,S).
```

Note that arbitrary PROLOG goals, hence also conjunctions, can be used in the `Goal` part.

It is clumsy to write a function such as `cost` in its relational form. Hence, for variables and parameters we allow to use functional notation in the `Template` part with help of a simple preprocessor. Thus, we can also write

```
subject_to(S =< 200) :-
        sum(product(P),cost(P)*x(P),S).
```

in order to express that the cost of the make is less than or equal to 200. Not only for parameters and variables, but also for expressing `sum`, `forall`, and a conjunction of constraints, the relational form is not very convenient. Thus, we also support a functional notation for these statements.

```
subject_to :-
        sum(product(P),cost(P)*x(P)) =< 200.
```

In general, we now assume that each predicate in the body of a nullary clause `subject_to/0` evaluates to a (conjunction of) constraint(s). Hence, an upper and lower bound of the cost of the make can be given by

```
subject_to :-
        sum(product(P),cost(P)*x(P)) =< 200,
        sum(product(P),cost(P)*x(P)) >= 100.
```

In general, we say an *expression* is an n-ary term p(t_1,...,t_n) such that the query p(t_1,...,t_n,V) assigns to V a unique term. Whenever an expression is allowed, the expression is replaced by the unique term V.

Since it is convenient to also allow constraints while declaring parameters or variables we adopt a similar notion. Hence, a parameter or variable declaration is a constraint declaration, but is unary instead of nullary, where the argument contains the name and index sets. Convenient abbreviations are also supported.

A PROLOG formulation of the production example is given in Figure 2. The close similarity to the formulation in Figure 1 should be obvious. Allowing for such a model formulation in PROLOG requires less than 1000 lines of PROLOG code including interfaces to 3 linear programming systems, some command interpretation, and an expression simplifier. The data for a concrete instance can also be given by a PROLOG program as shown in Figure 3. A nicer syntax for declaring data can be incorporated if needed.

```
set prod.

param rate:prod :- > 0.
param avail :- > 0.
param profit:prod.
param market:prod :- >= 0.

variable make:prod(J) :- >= 0,=< market(J).

objective max:total_profit :- sum(prod(P),profit(P)*make(P)).

subject_to time_res :- sum(prod(P),1/rate(P)*make(P)) =< avail.
```

Figure 2: The production model in PROLOG

```
prod(bands).               % set PROD := bands coils;
prod(coils).

rate(bands,200).           % param:    rate  profit  market :=
rate(coils,140).           %   bands    200    25      6000
profit(bands,25).          %   coils    140    30      4000 ;
profit(coils,30).
market(bands,6000).
market(coils,4000).

avail(40).                 % param avail := 40;
```

Figure 3: Data for the production model in PROLOG

2.2.4 Description of the Prototype

We describe the syntax, features, and usage of the current prototype implementation PLAM (ProLog And Modelling), aiming to simulating a standard mathematical programming modelling language. In general, a model consists of five different parts: sets, parameters, variables, objectives, and constraints. To distinguish these parts we use unary PROLOG operators `set`, `param`, `variable`, `objective`, and `subject_to` respectively.
A file containing a model must start with the line

```
:- plam.
```

Set declarations are of the form

set ⟨*set-name/arity*⟩.

where $\langle$*set-name*$\rangle$ denotes a Prolog predicate of arity *arity* to be defined in the corresponding data module. If /*arity* is missing, the arity defaults to 1. Optionally, arithmetic index sets are supported. For example,

```
set a :- 1..u by 2.
```

declares an index set a containing all odd integer numbers i with $1 \leq i \leq u$, where u is a parameter over no index sets. In the expression `1..u by s` the terms 1, u, and s can be either integer numbers or non-indexed parameters. If `by n` is missing, n = 1 is assumed. Arithmetic index sets are only supported for unary sets. In order to build a, for example, binary index set b ranging from 1 to 3 in the first argument and from 1 to 4 in the second argument you have to exploit the logic programming language by writing

```
set b/2.
b(X,Y) :- between(1,X,3,1),between(1,Y,4,1).
```

The predicate `between(L,X,U,S)`, which holds for all L $\leq$ X $\leq$ U and X $=$ L $+$ $n\cdot$ S, is supported by the system.

Parameter declarations are of the form

`param` $\langle$*param-name*$\rangle$: $[\langle$*set-name*$_1$/*arity*$_1\rangle, \ldots, \langle$*set-name*$_n$/*arity*$_n\rangle]$.

where $\langle$*param-name*$\rangle$ denotes an $(\sum_{i=1}^{n} arity_i)$-ary Prolog predicate to be defined in the data module. If $n = 1$, the list parentheses can be omitted. If /*arity*$_i$ is missing, /1 is assumed. Instead of *set-name*$_i$/*arity*$_i$ one can also write *set-name*$_i(X_1, \ldots, X_{arity_i})$. Optionally, a body is allowed containing a conjunction of expressions E_i evaluating to constraints. For example,

```
param p:[i(I)] :- >= q(I).
```

defines a parameter p which is indexed by a unary set i. As restriction we state that for all I we must have `p(I) >= q(I)`. Alternatively, we can write

```
param p:i :- forall(i(I),p(I) >= q(I)).
```

to which the first form is translated.

Variable declarations are similar to parameter declarations but the keyword `param` is replaced by `variable`. Declared variables must not be defined in the data module. The two special expressions `integer` and `binary` restrict variables to be integral, resp. 0-1 variables.

For the objective function, there must be a single declaration of the form

`objective min:<name> :-` S.   or
`objective max:<name> :-` S.

where S evaluates to a linear term.

Constraint declarations are of the form

`subject_to` $\langle$*name*$\rangle$:– $E_1, \ldots, E_m$.

such that each expression E_i evaluates to a constraint or a list of constraints.

A number of commands are supported in order to generate and solve a model. Among them

```
write_model(<filename>,<solver>) ,
```

which writes the modelled instance to the file `filename`, suitable to be read by a solver `<solver>`. Currently, `cplex` and `lpsolve` (see [Sha93]) and the portable format `mps` are supported. With

```
solve(<solver>).
```

the constructed model is generated and solved. Beside `cplex` and `lpsolve` also a built-in solver `clpr` can be used, if supported by the underlying PROLOG system. Several other commands are available for inspecting data, investigating parts of the model, or making the model available as a term.

2.3 Additional power through the logic programming language

Data for sets and parameters can be defined by arbitrary PROLOG rules, not only by facts. Thus, it is no problem to model problems containing complicated data items like, for example, prime numbers. Given a PROLOG predicate `prime(I,P)` that computes the I-th prime number and assigns it to P, we express with

```
set i :- 1..100.
variable x:i :- >= 0, =< 10.
subject_to pc :- sum(between(1,I,100),prime(I)*x(I)) =< 10000.
```

that the sum of the variables x_I weighted by the Ith prime number shall be less than or equal to 10000.

Furthermore, the predicates `sum` and `forall` can be called with an arbitrary PROLOG goal as first argument, which allows all kinds of index calculations. For example, imposing a strict ordering on some variables like

$$\forall 1 \leq i < j \leq n : x_i < x_j$$

is done by

```
param n :- >= 1.

subject_to :-
   forall((n(N),between(1,I,N),SI is I+1,between(SI,J,N)),
          x(I) < x(J)).
```

In graph theory, the fractional stable set polytope is defined by the inequalities

$$\forall (i,j) \in E : x_i + x_j \leq 1 \quad \wedge \quad 0 \leq x \leq 1,$$

where E denotes the edges of the graph. Given a predicate `edge(I,J)` that defines the edges of the graph, we can model the above condition:

```
variable x:1..n :- >= 0, =< 1.
subject_to :- forall(edge(I,J),x(I) + x(J) =< 1).
```

Graph manipulation packages in PROLOG are available [Gro95].

The features of a modelling environment mentioned by Williams [Wil93a] are supported by logic programming. Data and the statements of a model can be separated. The index mechanism for accessing variables and parameters is very powerful. Beside standard index access methods all kinds of index manipulations are available due to the underlying logic language. Providing summation or any other kind of combination over index sets is a matter of adding a few lines of PROLOG code. Relations between index sets are naturally available in PROLOG. Arithmetic on the coefficients is handled by a suitable expression simplifier. Since PROLOG is an interactive language, interactive modelling, debugging, etc. is well supported. Automatic formatting is available and can also be adapted to special needs, since the model representation can be made available as a PROLOG term.

The basic functionality needed for algebraic modelling is available in PROLOG without any further support. Syntactic support can be added with a few lines of code allowing to mimic standard algebraic modelling systems with a small effort. Whenever the supported built-ins are not sufficient, the full power of the logic programming environment can be used to fulfill the needs and can even be made available as built-in if necessary.

3 Modelling in constraint logic programming

Constraint logic programming (CLP) combines the declarative nature of logic programming with the efficiency of constraint solving over specific domains. In addition to *logical atoms* of the form $p(t_1, \ldots, t_m)$, the query or the body of a rule in a constraint logic program may also contain *constraints* $c(x_1, \ldots, x_k)$ over some domain of computation. These constraints may guide the computation, since a rule can be applied to the query only if the current constraint set is consistent. The output of a constraint logic program is in general no longer an answer substitution, but a set of *answer constraints*. The heart of a constraint logic programming system is the underlying constraint solver. It has to perform the following tasks:

- Decide whether a constraint set is satisfiable.
- Simplify a constraint set and compute a solved form.
- Eliminate a set of variables from a constraint set.

The theory of constraint logic programming provides a general programming language scheme CLP($\mathcal{X}$) that can be instantiated in various different ways, depending on the computational domain that is chosen. Among the most important domains that have been considered so far are [Col87, JL87, DvHS+88, ASS+88, Boc93]

- Linear arithmetic over the real or rational numbers, CLP($\mathcal{R}$) or CLP($\mathcal{Q}$),
- Boolean algebra, CLP($\mathcal{B}$),
- Finite domains, CLP($\mathcal{FD}$), and
- Pseudo-Boolean or 0-1 constraints, CLP($\mathcal{PB}$).

To solve discrete optimisation problems, the computational domain $\mathcal{X}$ is usually instantiated to finite domains $\mathcal{FD}$. This means that the constraints are defined over variables that take their values in finite sets of natural numbers. Since solving such constraints is in general, an NP-complete problem, the most important operation on constraints, a test for consistency, has been relaxed in most systems to local consistency, e.g. [DvHS+88, ACD+94, HSS+92, OB93, DC93, BMvH94, Pug94]. Local consistency procedures remove some inconsistent values from the domains of the variables by a constraint propagation mechanism, but in general they cannot achieve global consistency. This has to be ensured by the programmer, using an extra enumeration predicate. Therefore, a typical finite domain program has the form

```
problem(<Vars>) :- <state constraints over Vars>,
                   <enumerate domain of Vars>.
```

The search for a feasible solution of the collected constraints is performed by the Prolog backtracking mechanism, which can be guided in a high-level way by suitable heuristics, while the finite domain solver is responsible for pruning the search tree. Bounded integer linear programs

$$\max\{c^T x \mid Ax \begin{array}{c}\geq\\=\\\leq\end{array} b, 0 \leq x \leq u\}$$

can be expressed directly in this framework.

Example 1 *The linear 0-1 program*

$$\max\{-100x_1 + 72x_2 + 36x_3 \mid -2x_1 + x_2 \leq 0, -4x_1 + x_3 \leq 0, x_1 + x_2 + x_3 \geq 1, x \in \{0,1\}^3\}$$

can be formulated in CLP($\mathcal{FD}$) as follows:

```
example(L,Z) :-
    L = [X1,X2,X3],
    domain(L,0,1),
    -2*X1 + X2 #<= 0,
    -4*X1 + X3 #<= 0,
    X1 + X2 + X3 #>= 1,
    Z #= -100*X1 + 72*X2 + 36*X3,
    maxof(labeling(L),Z).
```

The system predicate `domain(L,0,1)` *constrains each variable in the list* `L` *to the domain* `0..1`*. The predicate* `labeling(L)` *enumerates the possible values of the variables in* `L` *and* `maxof` *computes the maximum of* `Z`*.*
Asking the query

```
?- example(L,Z).
```

yields the answer

```
L = [1,1,1], Z = 8
```

corresponding to the optimal solution.

From the modelling point of view, one of the most important differences of CLP($\mathcal{FD}$) compared to classical integer programming is that in addition to the standard arithmetical constraints, i.e., linear equations and linear inequalities, various other types of constraints are available, which are supported by special constraint solving algorithms.

Roughly, we can distinguish the following classes of constraints:

- Basic numerical constraints: `#=, #>=, #<=, #>, #<`.
- Symbolic constraints, e.g. `#\=` (disequality), `alldifferent`, `atmost`, `atleast`, `element`.
- Domain-specific "global" constraints, e.g. `cumulative, diffn, cycle, among`.
- Meta constraints, e.g. Big-M, disjunction, cardinality (cf. Section 4).

The constraint `alldifferent(L)` states that all values in a list of finite domain variables `L = [X1,...,Xn]` have to be pairwise different. In traditional integer programming, $n(n-1)/2$ constraints would be needed to express this condition. But, the `alldifferent` constraint can also be handled algorithmically. Such an algorithmic treatment avoids the creation of a quadratic number of constraints while the same pruning effect is achieved. Furthermore, even better pruning can be obtained by using special information on the `alldifferent` relation. For example, the property that for every subset of k variables the cardinality of the union of the associated domains must be greater than or equal to k can be exploited.

The constraints `atleast(N,L,V)` and `atmost(N,L,V)`, with a list `L` of domain variables and integers `N,V` state that at least or atmost `N` elements of `L` have the value `V`. The constraint `element(X,L,Y)`, with domain variables `X` and `Y` and a list of integers `L`, states that `Y` is equal to the `X`-th element of `L`.

The "global" constraints `cumulative, diffn, cycle` and `among` were first introduced in the constraint logic programming language Chip [AB93, BC94]. Similar constraints are also available in the Ilog system [Pug94, Ber95]. The `cumulative` constraint [AB93] is used to express cumulative resource limits over a period. The `diffn` constraint [BC94] is an n-dimensional generalisation of `alldifferent` and expresses non-overlapping constraints on n-dimensional rectangles. The `among` constraint [BC94] extends both `atmost` and `atleast` and enforces constraints on sequences of numbers. The `cycle` constraint [BC94] finds cycles in directed graphs.

Example 2 *We illustrate by the* `cycle` *constraint, how a classical mathematical programming problem like the traveling salesman problem can be modelled with these new constraint abstractions. The* `cycle` *constraint finds one or several cycles in a directed graph $G = (V, E)$. Suppose the graph consists of n nodes numbered from 1 to n. For each node $i \in V$ we introduce a finite domain variable X_i for the successor node of i with domain $D_i = \{j \mid (i,j) \in E\}$. The constraint*

```
cycle(N,[X1,...,Xn])
```

states that there are exactly `N` *non-overlapping cycles covering the graph G. In the special case* `N` *= 1, the X_i describe a Hamiltonian circuit. Note that this modelling with finite domain variables for the nodes of the graph is completely different from the standard approach for Hamiltonian circuits based on 0-1 variables for the edges. In order to associate weights w_{ij} to the edges $(i,j) \in E$ we introduce for each node $i \in V$ a constraint*

```
element(Xi,[Wi1,...,Win],Wi).
```

which states that `Wi` *is the weight of the edge leading from node i to its successor* `Xi`.

Now we consider an asymmetric traveling salesman problem with distance matrix [Wil93b]:

From/To	*A*	*B*	*C*	*D*	*E*	*F*
A	*0*	*42*	*62*	*53*	*96*	*105*
B	*52*	*0*	*49*	*29*	*54*	*84*
C	*70*	*42*	*0*	*77*	*65*	*129*
D	*42*	*35*	*56*	*0*	*57*	*56*
E	*105*	*63*	*81*	*41*	*0*	*80*
F	*101*	*93*	*111*	*72*	*75*	*0*

The following CHIP program solves this problem

```
tsp(L) :-
    L = [X1,X2,X3,X4,X5,X6],
    L :: 1..6,
    cycle(1,L),
    element(X1, [  0, 42,  62, 53, 96, 105], C1),
    element(X2, [ 52,  0,  49, 29, 54,  84], C2),
    element(X3, [ 70, 42,   0, 77, 65, 129], C3),
    element(X4, [ 42, 35,  56,  0, 57,  56], C4),
    element(X5, [105, 63,  81, 41,  0,  80], C5),
    element(X6, [101, 93, 111, 72, 75,   0], C6),
    Cost :: 1..1000,
    Cost #= C1 + C2 + C3 + C4 + C5 + C6,
    min_max(labeling(L),Cost).
```

Asking the query `?- tsp(L)` *yields the optimal solution* `L = [3, 6, 2, 1, 4, 5]` *with cost 346. Due to the underlying domain propagation, finite domain constraints can be combined freely. Note that this would not be possible when using a special purpose algorithm. For example, if we want to express the condition that node 5 should be among the first three nodes, we simply add the constraint*

```
atleast(1,[X1,X2,X3],5).
```

Now our program computes the optimal solution `L = [2, 3, 5, 1, 6, 4]` *of cost 350. The general* `cycle(N,L)` *constraint has been used in various contexts. Typical application areas include vehicle routing [BC94] or airline crew management [BKC94].*

Using the logic programming language, a user may define his own new constraint abstractions on top of the existing constraints in the system. If a constraint turns out to be useful in many areas the language designers may decide to build it into their CLP system and to support it by special algorithms.

4 Modelling in CLP($\mathcal{PB}$)

Pseudo-Boolean constraints are equations or inequalities between multilinear integer polynomials in 0-1 variables. On the one hand, they generalise Boolean constraints, on the other hand they are a restricted form of finite domain constraints, where all domains are equal to the two-element set $\{0, 1\}$. In operations research, pseudo-Boolean constraints correspond to non-linear 0-1 programming problems.

A constraint logic programming language CLP($\mathcal{PB}$) for pseudo-Boolean constraints has been introduced in [Boc93]. A prototype implementation of CLP($\mathcal{PB}$) has been developed in [Bar94]. The prototype solver of CLP($\mathcal{PB}$) is not intended to be used for solving large 0-1-problems. This should be better done with the 0-1 optimisation software OPBDP [Bar95]. However, the prototype solver can be very useful in modelling 0-1 problems. Given a set of possibly non-linear pseudo-Boolean constraints it computes an equivalent set of extended clauses of the form

$$L_1 + \cdots + L_k \geq d \ ,$$

saying that at least d out of k literals have to be true, where a literal is either a 0-1 variable X or its negation $1 - X$ (for an in-depth treatment of the underlying constraint solving techniques see [Bar96]).

4.1 Simplifying pseudo-Boolean constraints

First, we show how a constraint set can be simplified by the solver of CLP($\mathcal{PB}$). The main steps of the simplification procedure are the following:

- Linearisation of non-linear pseudo-Boolean constraints.
- Transformation of linear pseudo-Boolean constraints to equivalent sets of extended clauses.
- Derivation of stronger extended clauses including a check of consistency.

Example 3 *Given the non-linear constraint*

```
?- A*B + A*C + B*C #>= 1.
```

the solver computes the solved form

```
    A + B + C #>= 2.
```

saying that at least two of the three 0-1 variables `A,B,C` *have to be true. If we ask*

```
?- A*B + A*C + B*C #>= 2.
```

we get the answer

```
    A = 1, B = 1, C = 1.
```

Example 4 *Given the linear pseudo-Boolean inequality [Wil93a, p. 216]*

```
?- 3*X1 + 3*X2 -2*X3 + 2*X4 +2*X5 #<= 4.
```

the solver computes the solved form

```
~X1 + ~X2 +      ~X4 + ~X5 #>= 2,
~X1 + ~X2 + X3       + ~X5 #>= 2,
~X1 + ~X2 + X3 + ~X4       #>= 2
```

which, in this case, corresponds to three facets of the convex hull of the set of feasible 0-1 solutions of the original inequality (for a general discussion of the polyhedral properties of the solved form see [Bar96, Section 7.7])

4.2 Reasoning with user-defined constraints

Additional power is gained once again by combining constraint solving with the possibilities offered by the surrounding logic programming system. We can use logic programming for meta-programming with constraints.

In a first step, we associate with a linear pseudo-Boolean inequality `L #>= R` a Boolean variable `B` such that `B` logically implies `L #>= R`:

```
pb_switch_constraint(L #>= R, B) :-
    pb_lowerbound(L - R, M),
    L - M * ~BVar #>= R.
```

Here, `pb_lowerbound` computes a lower bound `M` for `L - R`, for example by summing up the negative coefficients. Similarly, we can impose the condition that `B` should be logically equivalent to `L #>= R`:

```
pb_equiv_constraint(L #>= R, B) :-
    pb_switch_constraint(L #>= R, B),
    pb_switch_constraint(L #< R, ~B).
```

After having defined indicator variables we can now express logical conditions on the constraints [MLM94]. For example, we can define for pseudo-Boolean constraints `C1, C2` the disjunction

```
pb_disj(C1,C2) :-
    pb_switch_constraint(C1,B1),
    pb_switch_constraint(C2,B2),
    B1 + B2 #>= 1.
```

or the implication

```
pb_impl(C1,C2) :-
    pb_equiv_constraint(C1,B1),
    pb_switch_constraint(C2,B2),
    B2 #>= B1.
```

Example 5 *We consider a non-trivial example from [MW89]. The problem is to model by a conjunction of linear pseudo-Boolean inequalities the condition:*

If 3 or more of products (1 to 5) are made, or less than 4 of products (3 to 6, 8, 9) are made then at least 2 of products (7 to 9) must be made unless none of products (5 to 7) are made.

If we represent the decision to make product i by a 0-1 variable `Pi` *this condition can be expressed as follows:*

```
pb_condition(P1,P2,P3,P4,P5,P6,P7,P8,P9) :-
    pb_equiv_constraint(P1+P2+P3+P4+P5 #>= 3, A),
    pb_equiv_constraint(P3+P4+P5+P6+P8+P9 #< 4, B),
    pb_equiv_constraint(P5+P6+P7 #>= 1, C),
    pb_equiv_constraint(P7+P8+P9 #>= 2, D),
    (A + B) * C #<= 2*D.
```

Note that $(A + B) * C \leq 2 * D$ is the pseudo-Boolean equivalent of $(A \vee B) \wedge C \rightarrow D$. Given the query

```
?- pb_condition(P1,P2,P3,P4,P5,P6,P7,P8,P9).
```

the solver of CLP($\mathcal{PB}$) computes the answer

```
P7 + P9 + ~P3 + ~P4 + ~P5 #>= 1,
P7 + P8 + ~P2 + ~P3 + ~P5 #>= 1,
P7 + P9 + ~P2 + ~P3 + ~P5 #>= 1,
P3 + P4 + P7 + P9 + ~P5 #>= 1,
P3 + P4 + P7 + P8 + ~P5 #>= 1,
P4 + P6 + P7 + P9 + ~P5 #>= 1,
P3 + P6 + P7 + P9 + ~P5 #>= 1,
P4 + P6 + P7 + P8 + ~P5 #>= 1,
P3 + P6 + P7 + P8 + ~P5 #>= 1,
P8 + P9 + ~P5 #>= 1,
P7 + P8 + ~P1 + ~P4 + ~P5 #>= 1,
P7 + P9 + ~P1 + ~P4 + ~P5 #>= 1,
P7 + P8 + ~P1 + ~P3 + ~P5 #>= 1,
P7 + P9 + ~P1 + ~P3 + ~P5 #>= 1,
P7 + P8 + ~P3 + ~P4 + ~P5 #>= 1,
P8 + P9 + ~P7 #>= 1,
P7 + P8 + ~P1 + ~P2 + ~P4 + ~P6 #>= 1,
P7 + P9 + ~P1 + ~P2 + ~P4 + ~P6 #>= 1,
P7 + P8 + ~P1 + ~P3 + ~P4 + ~P6 #>= 1,
P7 + P8 + ~P1 + ~P2 + ~P5 #>= 1,
P7 + P9 + ~P1 + ~P3 + ~P4 + ~P6 #>= 1,
P7 + P8 + ~P2 + ~P3 + ~P4 + ~P6 #>= 1,
P7 + P9 + ~P2 + ~P3 + ~P4 + ~P6 #>= 1,
P7 + P8 + ~P1 + ~P2 + ~P3 + ~P6 #>= 1,
P7 + P9 + ~P1 + ~P2 + ~P5 #>= 1,
P7 + P9 + ~P1 + ~P2 + ~P3 + ~P6 #>= 1,
P4 + P5 + P7 + P9 + ~P6 #>= 1,
P3 + P5 + P7 + P9 + ~P6 #>= 1,
P3 + P4 + P7 + P9 + ~P6 #>= 1,
P4 + P5 + P7 + P8 + ~P6 #>= 1,
P3 + P5 + P7 + P8 + ~P6 #>= 1,
P3 + P4 + P7 + P8 + ~P6 #>= 1,
P7 + P8 + ~P2 + ~P4 + ~P5 #>= 1,
P7 + P9 + ~P2 + ~P4 + ~P5 #>= 1,
P8 + P9 + ~P6 #>= 1
```

A tighter representation is possible using compact extended clauses [Bar96]. But, the main point here is that a complex Boolean relationship between constraints can be easily expressed in the logic programming environment.

Another convenient modelling operator is *cardinality*, which states an upper and lower bound for the number of constraints in a list that must hold.

```
pb_card(Lower,L,Upper) :-
      pb_switch_sum(L,S),S #>= Lower,S #<= Upper.

pb_switch_sum([],0).
pb_switch_sum([C|Cs],S+BV) :-
      pb_switch_constraint(C,BV),pb_switch_sum(Cs,S).
```

Hence, a disjunction of constraints can be expressed by

```
pb_card(1,[C1,...,CN],N).
```

and a conjunction by

```
pb_card(N,[C1,...,CN],N).
```

Parts of a problem can be semantically investigated using the interactivity of the logic programming environment together with the powerful constraint solving capabilities offered by constraint logic programming. For example, we can investigate what a disjunction or conjunction of constraints is equivalent to. The query

```
?- pb_card(1,[4*A + 3*B + 2*C #>= 3,4*A + 2*B + 3*C #>= 3],2).
```

gives the answer

```
   C + B + A #>= 1
```

Note that in this example already many features of the constraint solver are used, i.e., transformation, simplification, and projecting out the Boolean variables introduced by `pb_switch_constraint`.

5 Conclusion

Due to the progress in programming language design, implementing a modelling language must no longer be considered a difficult task [Fou83]. Moreover, implementing a special algebraic modelling language is not necessary, since existing declarative programming languages, like PROLOG, can be easily extended to suit the modeler's needs. All necessary extensions can be provided in PROLOG itself. Compared to existing modelling languages the full power of a complete programming language is available whenever needed, which offers extensibility and flexibility with respect to the problem to be modelled. By the introduction of constraints into logic programming, not only modelling is possible, but also constraint solving within the system. Beside just solving a completely specified problem, the available constraint operators can also be used to implement domain specific constraint solvers with small effort. For complex applications, parts of the problem that do not admit a natural encoding as constraints can be handled by exploiting the programming language power.

References

[AB93] A. Aggoun and N. Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. *Mathl. Comput. Modelling*, 17(7):57 – 73, 1993.

[ACD+94] A. Aggoun, D. Chan, P. Dufresne, E. Falvey, H. Grant, A. Herold, G. Macartney, M. Meier, D. Miller, B. Perez, E. van Rossum, J. Schimpf, P. A. Tsahageas, and D. H. de Villeneuve. ECLIPSE 3.4, ECRC Common Logic Programming System. Technical report, ECRC, Munich, July 1994.

[ASS+88] A. Aiba, K. Sakai, Y. Sato, D.J. Hawley, and R. Hasegawa. Constraint logic programming language CAL. In *Fifth Generation Computer Systems, Tokyo, 1988.* Springer, 1988.

[Bar94] P. Barth. *Short Guide to CLP($\mathcal{PB}$).* Max-Planck-Institut für Informatik, 1994. System available: ftp://www.mpi-sb.mpg.de/pub/tools/CLPPB/clppb.html.

[Bar95] P. Barth. A Davis-Putnam based enumeration algorithm for linear pseudo-Boolean optimization. Technical Report MPI-I-95-2-003, Max-Planck-Institut für Informatik, Saarbrücken, January 1995. System available: http://www.mpi-sb.mpg.de/~barth/opbdp/opbdp.html.

[Bar96] P. Barth. *Logic-based 0-1 constraint programming.* Operations Research/Computer Science Interfaces Series. Kluwer, 1996.

[BC94] N. Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *Mathl. Comput. Modelling*, 20(12):97 – 123, 1994.

[Ber95] H. Beringer. Global constraints for real-life problems. In *PACT'95, Paris*, 1995.

[BKC94] G. Baues, P. Kay, and P. Charlier. Constraint based resource allocation for airline crew scheduling. *ATTIS'94, Paris*, 1994.

[BMvH94] F. Benhamou, D. McAllester, and P. van Hentenryck. CLP(Intervals) revisited. In *Logic Programming. Proceedings of the 1994 International Symposium, ILPS'94*, 1994.

[Boc93] A. Bockmayr. Logic programming with pseudo-Boolean constraints. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming. Selected Research*, chapter 18, pages 327 – 350. MIT Press, 1993.

[CM81] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer, 1981.

[Col87] A. Colmerauer. Introduction to PROLOG III. In *4th Annual ESPRIT Conference, Bruxelles.* North Holland, 1987.

[DC93] D. Diaz and P. Codognet. A minimal extension of the WAM for clp(FD). In *Proc. 10th Intern. Conf. Logic Programming, Budapest*, 1993.

[DvHS+88] M. Dincbas, P. van Hentenryck, H. Simonis, A. Aggoun, and T. Graf. The constraint logic programming language CHIP. In *Fifth Generation Computer Systems, Tokyo, 1988.* Springer, 1988.

[FGK93] R. Fourer, D. Gay, and B. W. Kernighan. *AMPL: a modeling language for mathematical programming.* The Scientific Press, San Francisco, 1993.

[Fou83] R. Fourer. Modeling languages versus matrix generators for linear programming. *ACM Trans. Math. Software*, 9(2):143 – 183, 1983.

[Gro95] Programming Systems Group. *SICStus v3 User's Manual.* Swedish Institute of Computer Science, 1995.

[HSS+92] W. S. Havens, S. Sidebottom, G. Sidebottom, J. Jones, and R. Ovans. Echidna: a constraint logic programming shell. In *Pacific Rim Int. Conf. Artificial Intelligence, Seoul, Korea*, pages 165 – 171, 1992.

[JL87] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proc. 14th ACM Symp. Principles of Programming Languages*, Munich, 1987.

[JM94] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 1994.

[MB93] Alexander Meeraus and Anthony Brooke. *GAMS: A User's Guide.* Boyd and Fraser publishing, 1993.

[MLM94] G. Mitra, C. Lucas, and S. Moody. Tools for reformulating logical forms into zero-one mixed integer programs. *Europ. J. Oper. Res.*, 72:262 – 276, 1994.

[MW89] K. I. M. McKinnon and H. P. Williams. Constructing integer programming models by the predicate calculus. *Annals of Operations Research*, 21:227–246, 1989.

[OB93] W. Older and F. Benhamou. Programming in CLP(BNR). In *Principles and Practice of Constraint Programming PPCP'93, Newport, RI*, 1993.

[Pug94] J.-F. Puget. A C++ implementation of CLP. In *Proceedings Second Singapore International Conference on Intelligent Systems*, Singapore, 1994.

[Sha93] R. Sharda. *Linear & discrete optimization and modeling software.* UNICOM, 1993.

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

[Wil93a] H. P. Williams. *Model building in mathematical programming.* John Wiley, third revised edition, 1993.

[Wil93b] H. P. Williams. *Model solving in mathematical programming.* John Wiley, 1993.