LOGICAL MODELING USING LPL

T. Hürlimann

Working Paper

June 1997
(revised September 1997)
(revised November 1997)

INSTITUT D'INFORMATIQUE, UNIVERSITE DE FRIBOURG

INSTITUT FÜR INFORMATIK DER UNIVERSITÄT FREIBURG

Institute of Informatics, University of Fribourg, site Regina Mundi

rue de Faucigny 2 , CH–1700 Fribourg / Switzerland

email: tony.huerlimann@unifr.ch

phone: ++41 26 300 8345 fax: ++41 26 300 9726

This research is supported by the Federal National Fond of Switzerland and financed by the project no. 1217-45922.95.

Logical Modeling using LPL

Tony Hürlimann, Dr. rer. pol.

Key words: Model Building, Modeling Language, Mixed Integer Programming, Logical Constraints.

Abstract: This paper gives an overview of different modeling transformation techniques to translate automatically logigal statements into integer and mixed integer programming constraints. The modeling language LPL is used as vehicle to express the models. The idea of having a unique representation scheme for mathematical and logical models is powerful but its advantages are not yet widely recognized [Williams 1990].
Initially, LPL was built to formulate the structure of bigger LP models to overcome the model management difficulties of large real-live LP models. In the meantime, the language specification has been extended several times to manage more complex models such as logical models.
IP and MIP techniques are applicable to a surprisingly large number of (logical) problems too. Methods are given to convert these models to pure MIP models. The LPL compiler translates pure logical sentences into IP constraints so that a linear MIP solver can solve them.

Stichworte: Modellierung, Modelliersprache, MIP Programmierung.

Zusammenfassung: Dieses Papier gibt einen Überblick über die verschiedenen Modellierungstechniken – ausgedrückt in der LPL Modellierungssprache – in der ganzzahligen und gemischt-ganzzahligen Programmierung. LPL (Linear Modeling Language) war ursprünglich für das Modellmanagement von grösseren LP Modellen entworfen worden. Die Sprache wurde in verschiedenen Etappen erweitert und kann auch für MIP Modelle oder sogar logische Modelle verwendet werden.
Eine überraschend grosse Zahl von Modellen kann als MIP formuliert werden. Viele Modelle, die nicht in MIP-Form vorliegen, können automatisch in solche übersetzt werden. Es werden Methoden und Regeln für eine solche Übersetzung gegeben.

1 INTRODUCTION

> "One of the reasons why IP has not been applied anywhere near as widely as it might to practical situations is the failure to recognize when a problem can be cast in this mould."
>
> Williams H.P.

This paper supposes that the reader is familiar with the basics of the LPL modeling language as described in [Hürlimann 1996] as well as with the basics of the Boolean algebra and predicate logic [Mendelson 1964, Quine 1974]. A brief survey of LPL can also be found in [Hürlimann 1996a].

To summarize, the main features of LPL are:

- A simple syntax of models with indexed expressions close to the mathematical and logical notation, and directly applicable for documentation,
- Transformation rules from logical expressions into 0-1 programs as described in this paper,
- Formulation of both small *and* large LP's with optional separation of the data from the model structure,
- Availability of a powerful index mechanism, making model structuring very flexible,
- An innovative and high-level Input and Report Generator,
- Intermediate indexed expression evaluation (much like matrix manipulation),
- Automatic or user-controlled production of row- and column-names,
- Tools for debugging the model (e.g. explicit equation listing),
- Built-in text editor to enter the LPL model,
- Fast production of the MPS file,
- Open interface to most LP/MIP solver packages.

LPL has been enhanced by several logical operators to exploit the power of integer programming. This extension of the new version 4.20 of LPL is summarized in Section 2. Section 3 introduces some generalities and definitions on IP and MIP modeling. It is well known that logical statements can be translated into IP constraints containing 0–1 variables. Section 4 summarizes the translation rules used by the LPL compiler. Section 5 gives a number of applications. Actually, the translation rules are hardwired within the

modeling language compiler, and the last section exposes some general ideas on model transformation rules and their connections using the term rewriting paradigm.

A surprisingly wide class of practical problems can be modeled using integer programming. There are applications in OR/MS, including operational problems such as distribution of goods, production scheduling, machine sequencing; planning problems such as capital budgeting, facility location, portfolio analysis; design problems such as communication and transportation networks design, VLSI circuit design. There are also many applications in combinatorics, graph theory and logic. An even broader model class are mathematical models combined with some logical conditions.

But the MIP formulation of a problem is sometimes far from being trivial. Ingenious techniques and a lot of modeling experience is needed to formulate such problems. Often, a more natural formulation and representation for many problems in the mentioned domains is (a subset of) predicate logic. An extension of the LPL modeling language has been designed recently: It introduces several logical operators which enhances the expressive power of the language. This allows the modeler to formulate the problem in a subset of predicate logic. By default, the LPL compiler translates such representation into a mixed integer linear mathematical formulation in order to apply a general MIP solver.

Several modeling techniques in integer programming are investigated in this paper. Formulation methods are given for different problems, which can be expressed as MIP problems.

It is well known [Williams 1977], that Boolean expressions can be translated into linear, mathematical constraints such that the solution space is the same. Suppose, for example, that a mathematical model containing different linear constraints is given and the modeler wants – among other constraints – to add the logical constraint "*X or Y*" where X and Y are two propositional statements. Adding "*X or Y*" means that the model is no longer a pure mathematical model, but a mixed model of mathematical and logical constraints. To mould the whole model into a pure mathematical form, the logical statement "*X or Y*" must be replaced by the linear constraint "$x+y\geq 1$", where x and y are 0–1 variables, meaning that they are 1 if the corresponding proposition is true and 0 otherwise. It is not difficult to see that the constraint

"$x+y\geq 1$" holds if and only if "*X or Y*" is true.

Although there are different methods to translate logical statements into 0–1 constraints leading to more or less efficient model representation, a mechanical translation procedure is useful, since it will allow to apply a professional MIP solver to solve such mixed problems. Furthermore, the translation step – coded by hand – would be very time consuming and prone to errors. Hence, an automated translation procedure is especially interesting for mixed models containing symbolic and quantitative knowledge. But even pure logical models such as the SAT problem (satisfiability problem) can be translated into an pure IP model. Since these problems are all NP–complete, it is advantageous to approach their solutions using different methods. Furthermore, an important subset of logical problems that can be formulated as Horn clauses is not NP–complete. Horn clauses translated into IP programs can be solved using an LP solver, since it is sufficient to solve the LP relaxation of the IP program. There may be still no integral solution, but the satisfiability of a set of clauses can be deduced from the LP relaxation. This is an interesting aspect of many (Horn)-rule based knowledge bases: *Instead of using inference and resolution techniques to solve such problems, one may translate the problem into a LP problem and solve the transformed problem.* Today, large LP problems can be solved very quickly.

It is relatively new to integrate a mechanical translation procedure into a modeling system. McKinnon/Williams [1990] present a procedure and its implementation in Prolog, that accepts logical statements and outputs the corresponding linear constraints. Hadjiconstantinou/Lucas/Mitra/Moody [1992] and Mitra/Lucas/Moody [1994] also expose the specification of such a converter procedure which is integrated into the CAMPS modeling system [Lucas/Mitra 1988]. The present paper exposes another translation procedure which is seamlessly integrated into the LPL language.

Although such a general converter is useful for many problems, one should not imply that every logical model should be translated into an IP model to be solved with a general IP solver. Logical models are normally solved more efficiently by specialized and more appropriate solvers. One should here clearly separate the formulation and the solution process. LPL is a language which allows the modeler to *formulate* the model, but the language has no general mechanism to *solve* the problem. Since it is generally admitted that a general solver for mathematical and logical models will never show up, and

since many specialized and efficient solvers exist for different subsets of problems, the motivation is to have – at least – a unique modeling language framework with which all kind of models might be *formulated* and which is formalized enough to be processed automatically by a computer. If there is no hope for a unique universal solver, it might at least be possible to have a unique language paradigm of formulation, which allow the modeler to translate the model specification into a form appropriate for a specific solver.
In this paper, I am concentrating on a *general* translation procedure of all kind of logical statements into mathematical constraint.
Again, such a translator is problematic in many contexts. Neither does there exist a general (best) method on translating a specific logical into a specific mathematical constraint nor is it always a good idea to *translate* logic into mathematics. Nevertheless, for some models this procedure might be very useful. And – what is much more promising – whole families of translators will be invented (I am convinced) which makes the solver independent formulation of problems an extremely powerful mean of computer-based modeling — we shall be prepared with our declarative modeling language!

2 LPL EXTENSION FOR LOGICAL MODELING

Mathematical models can be represented by the LPL language in a form similar to algebraic, indexed notation.
Example: the constraint

$$a_i \le \sum\nolimits_{\varphi=1}^{\nu} \beta_{\iota\varphi} + \sum_{\sigma\in\Sigma}\sum\nolimits_{\kappa=2}^{\mu-1} \chi_{\iota,\kappa-1,\sigma}\xi_{\kappa\sigma} \quad \text{φορ εϖερψ} \in T \qquad (1)$$

would be formulated, using the LPL syntax, as follows:

```
R{i=T}: a[i] <= SUM{j} b[i,j] + SUM{k,s IN S | k>1 AND k<n} c[i,k-1,s]*x[k,s]
```

The statement contains data tables such as c_{iks} or a_i, an indexed variables x_{ks} several indices (i,j,...), and arithmetical as well as logical operators. Several logical operators have been incorporated into the LPL language to evaluate Boolean expressions. However, Boolean expressions were not allowed in expressions containing variables as operands in previous version of LPL. A Boolean expression, such as

$$k > 1 \ \text{ανδ}\ \kappa < \nu \qquad (2)$$

in the example above, can be evaluated immediately, since the operands are all known quantities, no variable is implied.

Yet, consider the following expression, where x is a unknown quantity (a variable in the mathematical sense):

$$((x \le 2)\ o\rho(\xi \ge 5))\ \alpha\nu\delta(\xi \ge 0) \qquad (3)$$

This Boolean expression contains a variable *x* and, therefore, cannot be evaluated immediately, since the value of *x* is unknown. If *x* is between zero and two or greater than 5, the expression must be true otherwise it must be false (Figure 1).

Figure 1: Concave Variable Space

The Boolean expression (3) must be approached differently than (2) since it cannot be evaluated immediately because it contains *variables*. By introducing an additional 0–1 variable *y*, for example, the expression (3) can be expressed by the following three numerical constraints (where *M* is a large number that does not impose any restriction on *x*):

$$\begin{aligned} & x \ge 0 \\ & \xi \le 2 + M(1-\psi) \qquad (4) \\ & \xi \ge 5 - 5\psi \end{aligned}$$

It is easy to verify that (4) expresses exactly the same as (3). We suppose first that $y = 0$, then it follows from (4) that *x* is between 5 and *M*+2 (a big number); we suppose now $y = 1$, than it follows from (4) that *x* must be between zero and 2. That's exactly what is needed. We have expressed a mixed logical-mathematical constraint as a set of purely mathematical constraints.

The steps to translate (3) into (4) can be automated, although this is sometimes tricky and far from straightforward, if the result produced should be as efficient as possible (from the solution point of view).

Nevertheless, a modeling language should accept logical expressions (2) as well as (3), independently of how they are processed. In LPL, a Boolean expression such as $((x \le 2)\ o\rho(\xi \ge 5))\ \alpha\nu\delta(\xi \ge 0)$ can be written in a straightforward way and can be integrated into the model as a ***model constraint*** such as

```
CONSTRAINT MyConstraint: (x<=2 OR x>=5) AND x>=0;
```

To process such a constraint, using a standard solver, it must be translated into a purely logical or into a purely mathematical statement. Since LPL normally, interacts with an LP/MIP-solver, the compiler contains a procedure that

translates them into linear constraints containing 0–1 variables – as shown in (4), in order to apply the LP/MIP-solver.

FLEXIBLE STORAGE: AN EXAMPLE

A more realistic, small example is presented to show how to model logical constraints using LPL [MEIER G al., 1992, p 150 (see model *b1*)].
Suppose a company produces two liquids A and B in unknown quantities x_A and x_B. The liquids must be stored in containers. The company owns two containers with capacities t_1 and t_2. The liquids cannot be mixed in the same container. Normally, liquid A is stored in the first and liquid B in the second container. The company could therefore produce the maximum quantity of t_1 of liquid A and a maximum quantity of t_2 of liquid B. But in some situations, it may be more advantageous to produce more liquid of the same type and none of the other. In this case the company may produce one liquid at a maximum quantity of t_1+t_2 and to store it in both containers. Hence, either the company must produce both liquids at quantities less than t_1 and t_2, or it must produce only one liquid with quantity less than t_1+t_2.
How can we model such a situation? Figure 2 represents the feasible space: Any point on the line AE or on the line AF as well as any point within the rectangle ABCD is a feasible point.

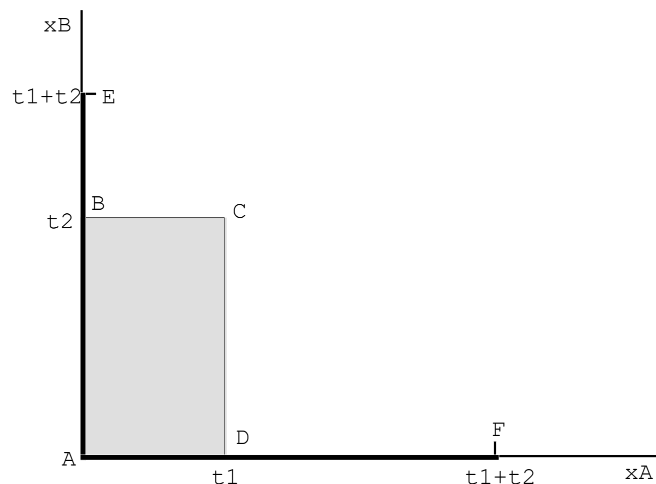


Figure 2: Disjunctive Set

The feasible space is a *disjunctive set*. Therefore, the situation cannot be formulated in a single LP model. What can be done is to solve several LP's: one for each conjunctive (convex) space. Three LP's are solved sequentially

which differ from each other in bound specifications:

LP 1: impose the bounds: $x_A \leq \tau_1$, $\xi_B \leq \tau_2$

LP 2: impose the bounds: $x_A \leq \tau_1 + \tau_2$, $\xi_B \leq 0$

LP 3: impose the bounds: $x_A \leq 0$, $\xi_B \leq \tau_1 + \tau_2$

The overall optimal solution would then be the solution from the LP with the largest (smallest) objective value if one maximizes (minimizes) a function.

Another way to model the disjunctive feasible space is by using logical constraints. The feasible space in Figure 2 could be modeled in a single expression as

$$(x_A \leq \tau_1 \ \alpha\nu\delta \xi_B \leq \tau_2) \ o\rho (\xi_A > \tau_1 \ \alpha\nu\delta \xi_B \leq 0) \ o\rho (\xi_B > \tau_2 \ \alpha\nu\delta \xi_A \leq 0) \quad (1)$$

The expression within the first parentheses represents the space ABCD, the second represents the line AF and the third represents the line AE. Another way to formulate the feasible space – which is essentially the same – is to say, that if x_A exceeds t_1 then x_B must be zero and if x_B exceeds t_2 then x_A must be zero:

$$(x_A > \tau_1 \rightarrow \xi_B \leq 0) \ \alpha\nu\delta (\xi_B > \tau_2 \rightarrow \xi_A \leq 0) \quad (2)$$

It is easy to verify that the expression (2) models exactly the feasible space in figure 2 and is the same as (1).

To add such logical constraints to an otherwise mathematical model, one needs to translate them into purely mathematical statements. This is possible, again, by adding 0–1 variables and the way this is done is explained in a moment. It is not so easy, however, to translate this logical expression into a set of "optimal" pure mathematical constraints. We will see in this example what "optimal" means. (For an introduction into such techniques see Williams 1990). The expression (2) can be translated using four additional 0–1 variables d_1, d_2, d_3, and d_4. It is straightforward to verify that the conjunction of the following six (linear) constraints models exactly the same feasible space as the formula (1) or (2):

$$
\begin{aligned}
&x_A - \tau_1 \le \tau_2(1-\delta_1)\\
&\xi_B - \tau_2 \le \tau_1(1-\delta_3)\\
&\xi_B \le (\tau_1+\tau_2)(1-\delta_2)\\
&\xi_A \le (\tau_1+\tau_2)(1-\delta_4)\\
&\delta_1+\delta_2 \ge 1\\
&\delta_3+\delta_4 \ge 1
\end{aligned}
\qquad (3)
$$

Another more concise way to model (2) uses only two 0–1 variables

$$
\begin{aligned}
&x_A - \tau_1 \le \tau_2(1-\delta_1)\\
&\xi_B - \tau_2 \le \tau_1(1-\delta_2)\\
&\xi_B \le (\tau_1+\tau_2)\delta_1\\
&\xi_A \le (\tau_1+\tau_2)\delta_2
\end{aligned}
\qquad (4)
\qquad\qquad
\begin{aligned}
&x_A - \tau_1 \le \tau_2\delta_1\\
&\xi_B - \tau_2 \le \tau_1\delta_2\\
&\xi_B \le (\tau_1+\tau_2)(1-\delta_1)\\
&\xi_A \le (\tau_1+\tau_2)(1-\delta_2)
\end{aligned}
\qquad (5)
$$

(4) and (5) are essentially the same. It is again easy to verify that (4) or (5) represent the feasible space of (2). To check it, we have to consider four cases:

1) $d_1 = \delta_2 = 0$, then (4) reduces to

$$
\begin{aligned}
&x_A - \tau_1 \le \tau_2\\
&\xi_B - \tau_2 \le \tau_1\\
&\xi_B \le 0\\
&\xi_A \le 0
\end{aligned}
$$

since all four constraints must be true, this means that $x_A = \xi_B = 0$. We suppose that the produced quantities cannot be negative.

2) $d_1 = 1$, $\delta_2 = 0$, then (4) reduces to

$$
\begin{aligned}
&x_A \le \tau_1\\
&\xi_B \le \tau_1+\tau_2\\
&\xi_B \le \tau_1+\tau_2\\
&\xi_A \le 0
\end{aligned}
$$

since all four constraints must be true, this means that $x_A = 0$, $\xi_B \le \tau_1 + \tau_2$.

3) $d_1 = 0$, $\delta_2 = 1$, then (4) reduces to

$$
\begin{aligned}
&x_A \le \tau_1+\tau_2\\
&\xi_B \le \tau_2\\
&\xi_B \le 0\\
&\xi_A \le \tau_1+\tau_2
\end{aligned}
$$

since all four constraints must be true, this means that $x_A = \tau_1 + \tau_2$, $\xi_B = 0$.

4) $d_1 = \delta_2 = 1$, then (4) reduces to

$x_A \leq \tau_1$

$\xi_B \leq \tau_2$

$\xi_B \leq \tau_1 + \tau_2$

$\xi_A \leq \tau_1 + \tau_2$

since all four constraints must be true, this means that $x_A \leq \tau_1$, $\xi_B \leq \tau_2$.

It is easy to translate (2) into (3) by doing the following transformations (explained in more details later on):

1 Original expression: $(x_A > \tau_1 \rightarrow \xi_B \leq 0)$ αvδ $(\xi_B > \tau_2 \rightarrow \xi_A \leq 0)$ (a)

2 Replace implication: $(x_A \leq \tau_1 \vee \xi_B \leq 0)$ αvδ $(\xi_B \leq \tau_2 \vee \xi_A \leq 0)$ (b)

3 Add 0-1 variables for the sub-expressions:
$$\begin{aligned} d_1 &= 1 \rightarrow \xi_A \leq \tau_A \\ \delta_2 &= 1 \rightarrow \xi_B \leq 0 \\ \delta_3 &= 1 \rightarrow \xi_B \leq \tau_B \\ \delta_4 &= 1 \rightarrow \xi_A \leq 0 \end{aligned} \qquad \text{(c)}$$

4 (b) becomes now: $d_1 \vee \delta_2$, $\delta_3 \vee \delta_4$ (d)

5 translate (d) gives : $d_1 + \delta_2 \geq 1, \delta_3 + \delta_4 \geq 1$ (e)

6 translate (c) gives:
$$\begin{aligned} & x_A - \tau_A \leq Y(\xi_A - \tau_A) \cdot (1 - \delta_1) \\ & \xi_B \leq Y(\xi_B) \cdot (1 - \delta_2) \\ & \xi_B - \tau_B \leq Y(\xi_B - \tau_B) \cdot (1 - \delta_3) \\ & \xi_A \leq Y(\xi_A) \cdot (1 - \delta_4) \end{aligned} \qquad \text{(f)}$$

(Note that U(y) is defined as the upper bound of y).

7 Since $U(x_A) = Y(\xi_B) = \tau_A + \tau_B$ it follows that (e) and (f) together are the same as (3).

8 A further reduction ca be obtained by observing that d_2 can be replaced by $1 - \delta_1$ and d_4 by $1 - \delta_3$. In this case, both constraints in (e) are trivially satisfied and can be dropped which generates the constraints in (4).

LPL automatically generates the form (3) from the following code:

```
PARAMETER    t1; t2;              (* container capacities *)
VARIABLE     xA,xB [0,t1+t2];     (* lower and upper bounds of xA and xB *)
CONSTRAINT C:   (xA > t1 -> xB <= 0)   AND   (xB > t2 -> xA <= 0);
```

(It is presently not possible to generate (4) by LPL version 4.11).

By the way, note that (4) (or (5)) is the most compact form to model the feasible space: the model formulation in (4) uses <u>two</u> 0–1 variables to map a space containing <u>three</u> disjunctive subspaces. It is not possible to use less than two 0-1 variables to model this space.

Of course, (4) is logical not equivalent to (2), but (4) implies (2). That is whenever (4) is true than also (2) must be true, if however (4) is false nothing

can be said about the truth of (2) – it could be true or false.

This section only gave a simple but not-trivial example. The theory behind this transformation will now be developed in the rest of this paper.

OVERVIEW OF LOGICAL OPERATORS IN LPL

Table 1 summarizes all logical operators that are defined in LPL and that can be used in the formulation of logical model constraints. Of course, all operators can also be used in Boolean expressions which are evaluated immediately (which do not contain model variables) as in

```
PARAMETER a{i,j};
VARIABLE X{i,j | ATLEAST(3) {i} a[i,j]};
```

This declaration of the variable X is perfectly correct. It means that the variable X is declared for every {i,j}-tuple, such that at least three of a row i in the (known) data matrix a_{ij} are different from zero.

Operator	Alternative formulation	Interpretation
		(x and y are any logical sub-expression containing variables)
unary operators		
`~x`		x is false
binary operators		
`x AND y`	`ATLEAST(2) (x,y)`	both (x and y) are true
`x OR y`	`ATLEAST(1) (x,y)`	at least one of x or y is true
`x XOR y`	`EXACTLY(1) (x,y)`	exactly one is true (either ... or)
`x -> y`	`~x OR y`	x implies y (implication)
`x <-> y`	`(x -> y) AND (y -> x)`	x if and only if y (equivalence)
	`~(x XOR y)`	negation of XOR
`x NOR y`	`~(x OR y)`	none of x and y is true
	`~x AND ~y`	
	`ATMOST(0) (x,y)`	at most none is true
`x NAND y`	`~(x AND y)`	they are not both true
	`~x OR ~y`	
	`ATMOST(1) (x,y)`	at most one is true
indexed operators		
`AND{i} x[i]`	`ATLEAST(#i){i} x[i]`	all x[i] are true*
`OR{i} x[i]`	`ATLEAST(1){i} x[i]`	at least one out of all x[i] is true
`XOR{i} x[i]`	`EXACTLY(1){i} x[i]`	exactly one out of all x[i] is true
`NOR{i} x[i]`	`~OR{i} x[i]`	none of all x[i] is true
	`ATMOST(0){i} x[i]`	at most zero of all x[i] are true
`NAND{i} x[i]`	`~AND{i} x[i]`	not all of x[i] are true
	`ATMOST(#i-1){i} x[i]`	at least one of x[i] is false
`FORALL{i} x[i]`	`ATLEAST(#i){i} x[i]`	all x[i] are true
	`ATLEAST(#i){i} x[i]`	at least all x[i] are true
`EXIST{i} x[i]`	`OR{i} x[i]`	at least one out of all x[i] is true
	`ATLEAST(1){i} x[i]`	at least one out of all x[i] is true
`ATLEAST(k){i} x[i]`		at least k out of all x[i] are true
`ATMOST(k){i} x[i]`		at most k out of all x[i] are true
`EXACTLY(k){i} x[i]`		exactly k out of all x[i] are true

* note that "(#i)" means "cardinality of i"

Table 1: Logical Operators in LPL

The operators ~ (NOT), AND, OR have the usual meaning. XOR is the exclusive OR (either ... or), it is defined as

$$x \dot{\vee} \psi =_{\delta\varepsilon\phi} \leftarrow(\xi \leftrightarrow \psi)$$

-> is the implication; it is defined as

$$x \rightarrow \psi =_{\delta\varepsilon\phi} \leftarrow\xi \vee \psi \quad .$$

<-> is the equivalence; its definition is:

$$x \leftrightarrow \psi =_{\delta\varepsilon\phi} (\xi \rightarrow \psi) \wedge (\xi \leftarrow \psi) = (\xi \vee \leftarrow\psi) \wedge (\leftarrow\xi \vee \psi) = (\xi \wedge \psi) \vee (\leftarrow\xi \wedge \leftarrow\psi) \quad .$$

Two other operators are the NOR (neither ... nor) and the NAND, which are the negations of the OR and the AND operators. Table 2 gives an overview ("0" means "false", "1" means "true").

x	y	x AND y	x OR y	x XOR y	x -> y	x <-> y	x NAND y	x NOR y
1	1	1	1	0	1	1	0	0
1	0	0	1	1	0	0	1	0
0	1	0	1	1	1	0	1	0
0	0	0	0	0	1	1	1	1

Table 2: True-Tables of Logical Connectors

Note that the operators AND, OR, XOR, NOR, and NAND can be used as binary as well as index operators in LPL. As an example, "*AND{i} a[i]*" is the same as "*a[1] and a[2] and ... and a[n]*". Furthermore, "*x AND y*" can also be written as "*AND(x,y)*". For a detailed syntax of logical expressions see the Reference Manual [Hürlimann 1996]. It should also be noted that the *AND{}* has the same meaning as *FORALL{}* and the *OR{}* is the same as *EXIST{}*.

Note, however, that XOR, NOR, and NAND are not associative which means, for example, that the three expressions

NOR(x,y,z) (x NOR y) NOR z x NOR (y NOR z)

are not the same in LPL.

EXACTLY, ATLEAST, and ATMOST are also index operators with a slightly different syntax. The reserved word is followed by a numerical expression surrounded by parentheses. The expression

```
ATMOST (4) {i} a[i];
```

means that "at most 4 out of all a[i] must be true (=non-zero)" to make the whole expression true.

```
ATLEAST (2) {i} a[i];
```

means "at least 2 out of all a[i] most be true" and

```
EXACTLY (k) {i} a[i];
```

means “exactly *k* out of all a[i] must be true”.
Section 5 will show several applications that use these operators.

The following formula holds:

$$ATLEAST(k)_{i\in I}\ \Xi_{\iota} \equiv AN\Delta_{\Sigma\subseteq I:|\Sigma|=\nu-\kappa+1}\,(OP_{\iota\in\Sigma}\Xi_{\iota})$$

which is the conjunction of all $\binom{n}{n-\kappa+1}$ clauses consisting of exactly *k* positive propositions. This expression is also called “an extended clause” in [Barth 1996]. Hence, the ATLEAST()... operator is a convenient operator to condense an otherwise exponentially long Boolean formula.

We mention some identities which are helpful in translating different logical expressions into others. They are:

$$AND^{\nu}_{i=1}\xi_{\iota} \equiv \forall^{\nu}_{\iota=1}\xi_{\iota} \equiv AT\Lambda EA\Sigma T(\nu)^{\nu}_{\iota=1}\xi_{\iota}$$
$$OP^{\nu}_{\iota=1}\xi_{\iota} \equiv \exists^{\nu}_{\iota=1}\xi_{\iota} \equiv AT\Lambda EA\Sigma T(1)^{\nu}_{\iota=1}\xi_{\iota}$$

(If all expressions must be true then at least all are true; if one expression in a set is true then al least one is true.)

Furthermore, we have the identities:

$$NAND^{\nu}_{i=1}\xi_{\iota} \equiv \leftarrow(\forall^{\nu}_{\iota=1}\xi_{\iota}) \equiv \exists^{\nu}_{\iota=1}(\leftarrow\xi_{\iota}) \equiv AT\Lambda EA\Sigma T(1)^{\nu}_{\iota=1}(\leftarrow\xi_{\iota}) \equiv ATMO\Sigma T(\nu-1)^{\nu}_{\iota=1}\xi_{\iota}$$
$$NOP^{\nu}_{\iota=1}\xi_{\iota} \equiv \leftarrow(\exists^{\nu}_{\iota=1}\xi_{\iota}) \equiv \forall^{\nu}_{\iota=1}(\leftarrow\xi_{\iota}) \equiv AT\Lambda EA\Sigma T(\nu)^{\nu}_{\iota=1}(\leftarrow\xi_{\iota}) \equiv ATMO\Sigma T(0)^{\nu}_{\iota=1}\xi_{\iota}$$

(If not all expressions are true then there exists an expression which is not true, at least one is not true, or at most all but one are true. Likewise: If none of the expressions is true, then there does not exist any that is true, all are not true, which means that at least all are not true or at most zero are true.)
Note that using these identities we might as well eliminate the ATMOST-operator and replace it by the ATLEAST operator.

We might use also the following identities to eliminate the NOT operator (~):

$$ATLEAST(k)_{i=1}^{\nu}(\xi_\iota) \equiv ATMO\Sigma T(\nu-\kappa)_{\iota=1}^{\nu}(\leftarrow\xi_\iota)$$
$$ATMO\Sigma T(\kappa)_{\iota=1}^{\nu}(\xi_\iota) \equiv AT\Lambda EA\Sigma T(\nu-\kappa)_{\iota=1}^{\nu}(\leftarrow\xi_\iota)$$
$$\leftarrow(AT\Lambda EA\Sigma T(\kappa)_{\iota=1}^{\nu}(\xi_\iota)) \equiv ATMO\Sigma T(\kappa-1)_{\iota=1}^{\nu}(\xi_\iota)$$
$$\leftarrow(ATMO\Sigma T(\kappa)_{\iota=1}^{\nu}(\xi_\iota)) \equiv AT\Lambda EA\Sigma T(\kappa+1)_{\iota=1}^{\nu}(\xi_\iota)$$
$$\omega\eta\varepsilon\rho\varepsilon\ 0 \le \kappa \le \nu$$

Note that $ATLEAST(k)$ with $k > \nu\ \alpha\nu\delta\ ATMO\Sigma T(\kappa)\ \omega\iota\tau\eta\ \kappa < 0$ is defined to be false; likewise $ATMOST(k)$ with $k \ge \nu\ \alpha\nu\delta\ AT\Lambda EA\Sigma T(\kappa)\ \omega\iota\tau\eta\ \kappa \le 0$ is defined to be true.
Table 2 gives a more complete overview of the relationsship between the ATMOST and the ATLEAST operator.

k	$E \equiv AT\Lambda EA\Sigma T(\kappa)_{\iota=1}^{\nu}(\xi_\iota)\quad (S=\{1K\ \nu\})$ $\equiv ATMOST(n-k)_{i=1}^{n}(\neg x_i)$	$\leftarrow E \equiv ATMOST(k-1)_{i=1}^{n}(x_i)$ $\equiv ATLEAST(n-k+1)_{i=1}^{n}(\neg x_i)$
$k \le 0$	TRUE	FALSE
$k = 1$	$\bigvee_{i=1}^{n}(x_i)$	$\bigwedge_{i=1}^{n}(\neg x_i)$
$k = 2$	$\bigvee_{S:\lvert S\rvert=2}(\bigwedge_{i\in S} x_i) \equiv \bigwedge_{S:\lvert S\rvert=n-1}(\bigvee_{i\in S} x_i)$	$\bigwedge_{S:\lvert S\rvert=2}(\bigvee_{i\in S}\neg x_i) \equiv \bigvee_{S:\lvert S\rvert=n-1}(\bigwedge_{i\in S}\neg x_i)$
…	…	…
$k = j$	$\bigvee_{S:\lvert S\rvert=j}(\bigwedge_{i\in S} x_i) \equiv \bigwedge_{S:\lvert S\rvert=n-j+1}(\bigvee_{i\in S} x_i)$	$\bigwedge_{S:\lvert S\rvert=j}(\bigvee_{i\in S}\neg x_i) \equiv \bigvee_{S:\lvert S\rvert=n-j+1}(\bigwedge_{i\in S}\neg x_i)$
…	…	…
$k = n-1$	$\bigvee_{S:\lvert S\rvert=n-1}(\bigwedge_{i\in S} x_i) \equiv \bigwedge_{S:\lvert S\rvert=2}(\bigvee_{i\in S} x_i)$	$\bigwedge_{S:\lvert S\rvert=n-1}(\bigvee_{i\in S}\neg x_i) \equiv \bigvee_{S:\lvert S\rvert=2}(\bigwedge_{i\in S}\neg x_i)$
$k = n$	$\bigwedge_{i=1}^{n}(x_i)$	$\bigvee_{i=1}^{n}(\neg x_i)$
$k > n$	FALSE	TRUE

Table 2: ATLEAST versus ATMOST

Furthermore, we have:
$$EXACTLY(k)_{i=1}^{\nu}\xi_\iota \equiv ATMO\Sigma T(\kappa)_{\iota=1}^{\nu}\xi_\iota \wedge AT\Lambda EA\Sigma T(\kappa)_{\iota=1}^{\nu}\xi_\iota$$
which could be used to replace the EXACTLY operator.

Predicate variables in LPL

In LPL, one can declare and define predicate variables. They are variables of type BINARY (the 0-1 variables) as in

```
VARIABLE MyPredicate{i} BINARY;
```

To link the predicate with the rest of the otherwise mathematical model, an

expression containing numerical variables can be attached to a predicate. Suppose that a predicate *P* is introduced into the model with the meaning that if it is true, then another (real) variable *x* is greater than *a*. The following declaration introduces this predicate *P*. A real variable *x* is declared. The expression in P links the predicate to the (real) variable.

```
VARIABLE x;                  (* quantity x of product i produced *)
VARIABLE P BINARY : x>=a;    (* predicate P for x >= a *)
```

Using this declaration, one can express the logical condition $P = 1 \rightarrow \xi \geq \alpha$. which means that "if *P* is true (or 1) then *x* is greater than *a*."
It is also possible to link a predicate to an arbitrary, mathematical expression, such as

```
VARIABLE x; y;
VARIABLE Q BINARY : (x>a) OR (y<b);
```

The declaration of the predicate *Q* imposes the logical condition $Q = 1 \rightarrow ((\xi > \alpha) \text{ o}\rho (\psi < \beta))$.

It is also possible to model the inverse implication, for example $x \geq \alpha \rightarrow \Pi = 1$. This is the same as $P = 0 \rightarrow \xi < \alpha$. Such a constraint can be modeled in LPL as follows:

```
VARIABLE x;                     (* quantity x of product i produced *)
VARIABLE P BINARY ~: x<a;    (* predicate P for x >= a *)
```

The ~ operator is append to the type keyword BINARY. Arbitrary expressions are possible as in the example above.

This kind of modeling is very powerful, because it allows the modeler to mix mathematical and logical knowledge in the same model formulation. The modeler does not need to refer to several paradigms to formulate the complete model.

3 IP AND MIP: DEFINITIONS AND MODEL FORMULATION

In this section, some basics and notions in the realm of MIP-models are introduced as described in Nemhauser [1989].

3.1 DEFINITIONS

Integer programming (IP) and mixed integer programming (MIP) deal with problems of maximizing or minimizing a function of many variables subject to inequality and equality constraints, and integrality constraints on some or all

of the variables. Throughout this paper, linear constraints and linear objective functions are assumed. Hence, the *linear mixed programming problem* can be written as

$$\{\max cx + \eta\psi\ A\xi + \Gamma\psi \leq \beta,\ \xi \in \mathbf{Z},\ \psi \in \Re\} \qquad \text{(MIP)}$$

where **Z** is the set of non negative integral n-dimensional vectors, $\Re$ is the set of non-negative real p-dimensional vectors and $x = (x_1,....,x_n)$ and $y = (y_1,...,y_p)$ are the *variables*. An *instance* of the problem is specified by the *data* (c, h, A, G, b), with c an n-vector, h a p-vector, A an mxn matrix, G an mxp matrix, and b an m-vector.
A set $S = \{A\xi + \Gamma\psi \leq \beta,\ \xi \in \mathbf{Z},\ \psi \in \Re\}$ is called the *feasible region* and an $(x,y) \in \Sigma$ is called a *feasible solution*. $z = \chi\xi + \eta\psi$ is called the *objective function*. The feasible point (x^0,y^0) for which the objective function is as large as possible is called the *optimal solution*. The *linear (pure) integer programming problem* is defined as

$$\{\max cx{:}\ Ax \leq \beta,\ \xi \in \mathbf{Z}\} \qquad \text{(IP)}$$

which is a special case of the MIP problem with h=0 and G=0 (or p={} (empty set)).
The *linear programming problem* is defined as

$$\{\max hy{:}\ Gy <= \beta,\ \psi \in \Re\} \qquad \text{(LP)}$$

which is also a special case of the MIP problem with c=0 and A=0.

In LPL, the general MIP problem structure may be formulated as

```
(*/////// General MIP problem specification ///////*)
CONSTRAINT General_Mip;
 SET  m; n; p;
 PARAMETER c{n}; h{p}; A{m,n}; G{m,p}; b{m};
 VARIABLE  x{n} INTEGER; y{p};
 CONSTRAINT    constraints{m}: SUM{n} A*x + SUM{p} G*y <= b
 MAXIMIZE Objective: SUM{n} c*x + SUM{p} h*y;
END
```

The LPL formulation above contains all elements of the model structure. Together with the five data tables c, h, A, G, and b – which are in general sparse tables – as well as the three sets m, n, and p, any MIP instance can be formulated using LPL. From the modeler's point of view, however, this is not a very expressive formulation of the problem. Therefore, different

representational techniques will be considered for different modeling situations.

3.2 Convert general MIP problems to 0–1 MIP problems

In many MIP problems, the integer variables x are only used to represent logical relationships and, therefore, $x \in Z$ may be replaced by $x \in \{0,1\}$. The resulting MIP problem is called *0–1-MIP problem*.

Every MIP problem can be translated into a 0–1-MIP problem by the following procedure [Salkin 89]

Replace every integer variable x by either the expression

(i) $\sum_{k=1}^{\upsilon} t_k$, (where t is a 0–1 variable and u is the upper bound of x),

and omit every $x \le \upsilon$ constraint; or the expression

(ii) $\sum_{k=0}^{\lfloor \log_2 \upsilon \rfloor} 2^k t_k$

and keep the constraint $\sum_{k=0}^{\lfloor \log_2 \upsilon \rfloor} 2^k t_k \le \upsilon$.

Example:

An MIP problem contains the integer variable x with an upper bound 107. Every occurrence of x within the model may be replaced by (where all t_i are 0–1 variables)

(i) $\sum_{i=1}^{107} t_i$ (and drop also the upper bound constraint $x \le 107$)

or by

(ii) $\sum_{i=0}^{6} 2^i t_i$ (and keep the upper bound constraint $\sum_{i=0}^{6} 2^i t_i \le 107$).

Whether the translation of a general MIP to a 0–1 MIP problem is necessary, has to be decided by the modeler. Sometimes it is easier to solve the corresponding 0–1 MIP problem and the transformation is worthwhile, sometimes the solver accepts only 0–1 variables but not general integer variables. (Whether or not it is necessary to convert the model, LPL offers a mechanism to do the transformation automatically. The directive (*$G before the declaration of a general integer variable will suppress the transformation, otherwise LPL transforms each general integer variable into 0–1 variables by default using the (ii) transformation.) The transformation of general integer into 0-1 variables has been removed from LPL, since most solver now accept general integer variables (1997).

3.3 Solver dependent formulations

An important issue in integer programming – from the solver's point of view – is *how* to formulate the model. For the general IP model defined above,

different data tables (A, b) may contain the same solution space S. But the choice of a "good" data set may be essential to solve a model quickly. A more obvious formulation may not be a better one when it comes to solving the problem.

An example shows this important aspect in integer programming [Nemhauser 89] The feasible region

$$S = \{ (0000),(1000),(0100),(0010),(0001),(0110),(0101),(0011) \}$$

may be defined by the constraint

$$S = \{\xi \in \{0,1\}^4 : 93\xi_1 + 49\xi_2 + 37\xi_3 + 29\xi_4 \le 111\} \qquad \text{(a)}$$

Another formulation with the same solution region is

$$S = \{\xi \in \{0,1\}^4 : 2\xi_1 + \xi_2 + \xi_3 + \xi_4 \le 2\} \qquad \text{(b)}$$

Still another formulation of the same set is

$$S = \{\xi \in \{0,1\}^4 : \begin{array}{l} 2\xi_1 + \xi_2 + \xi_3 + \xi_4 \le 2 \\ \xi_1 + \xi_2 \le 1 \\ \xi_1 + \xi_3 \le 1 \\ \xi_1 + \xi_4 \le 1 \end{array} \} \qquad \text{(c)}$$

At first glance, it seems that the formulation (c) is more difficult to solve, since it contains more constraints than the others, and intuitively it may be clear that a problem containing more constraints is also more difficult to solve. This is indeed true in general for pure LP models: the (average) solution time grows with the number of constraints. This is not necessarily the case for IP or MIP models. How to compare the different formulations? Most MIP solver need an upper bound on the value of the objective function, *and the efficiency of the solver is very dependent on the sharpness of this bound*. One such upper bound is the optimal solution to the LP relaxation of the IP problem. Dropping the integral condition for its variables in IP, a new problem is obtained, which is called the *LP relaxation* of the corresponding IP problem. Since (c) gives a tighter bound on the optimal value of the LP relaxation than (b) or (a), the formulation (c) is better than (b) or (a) from the solver's point of view. The relaxation of the formulation (c) gives, in fact, the same optimal solution as the optimal value of the corresponding IP problem. Hence (c) can be solved using LP techniques without recurring to branch and bound methods. The problems (a) and (b), however, cannot be solved to optimality using its LP relaxation.

In the incapacitated facility location problem (see below), the same striking observation can be made. The constraints

$y_i \le x \quad \text{for all } i \in \{1 \ldots n\}$ (1)

can be replaced by a single constraint with the same (integer) solution space

$\sum_{i=1}^{n} y_i \le nx$ (2)

But (1) gives a tighter bound to the model than (2). On the other hand, if n is large it can be advantageous to use (2) instead. Wiiliams [1974 and 1978] presents several striking examples.

It is not true neither that the more (non-redundant) constraints a IP formulation has the simpler it is to be solved. As an example take the space

S = {(111), (110), (101), (011)} which can be expressed by

$S = \{x + y \ge 1, x + z \ge 1, y + z \ge 1\}$ (1)

or by

$S = \{x + y + z \ge 2\}$ (2)

(where x, y, and z are 0-1 variables).

However, (2) generates a tighter bound if the LP relaxation is solved. This can be seen by the point (0.5, 0.5, 0.5) which belongs to (1) but not to (2).

This space is the interesting example, which can also be formulated as:

$ATLEAST_{2\ (\text{out of the } 3)}(x,y,z) \equiv (x \wedge y) \vee (x \wedge z) \vee (y \wedge z) \equiv$

$(x \vee y) \wedge (x \vee z) \wedge (y \vee z) \equiv ATMOST_{1\ (\text{out of the } 3)}(\neg x, \neg y, \neg z)$

(see Table 5).

An different story are redundant variables and constraints. Redundant variables and constraints do not tighten the space in any way. A automatic translation of logical formula into mathematical expressions must be carefully designed in order to generate as few redundant variables and constraint as possible. This is all but easy and has revealed to be one of the hardest part in implementation of the following transformation procedure.

4 TRANSFORMATION RULES FOR LOGICAL CONSTRUCTS

The LPL compiler includes a procedure which allows translating logical statements, using the logical operators listed in the last section (Table 1) into pure linear, mathematical statements. The key of such a transformation lays in the fact that the domain {TRUE, FALSE} of a logical proposition is mapped into the two integers {0,1}. For every proposition X, a 0–1 variable x is introduced with the meaning that x=0 if and only if X is FALSE and x=1 if and only if X is TRUE. Different Boolean expressions can then be mapped using this interpretation. An impractical procedure would be to list all kinds of Boolean expressions and to map them into a specific linear constraint using 0–

1 variables. Such an (necessary incomplete) list is given in Table 3 (see also [Hadjiconstantinou al.] or [Yager]).

As can be seen from Table 3, a unique transformation does not exist. Several translation rules, applied in a different order to logical expressions, yield different mathematical constraints which might be more or less tight.

	propositions are X, Y, Z, X_i $(i = 1K\ \nu)$	0–1 variables are x, y, z, x_i $(i = 1K\ \nu)$ ("al" means: an alternative construct)
1	X	$x \geq 1$
2	$\leftarrow X$ (not X)	$x \leq 0$
3	$X \vee \Psi$ $(\Xi\ \text{ορ}\ \Psi)$	$x + \psi \geq 1$
3a	$X_1 \vee \Xi_2 \vee K \vee \Xi_\nu$	$x_1 + \xi_2 + K + \xi_\nu \geq 1$
4	$X \wedge \Psi$ $(\Xi\ \text{ανδ}\ \Psi)$	$x \geq 1,\ \psi \geq 1$ $(\alpha\lambda\ \ \xi + \psi \geq 2)$
4a	$X_1 \wedge \Xi_2 \wedge K \wedge \Xi_\nu$	$x_1 \geq 1, \xi_2 \geq 1, K, \xi_\nu \geq 1$ $(\alpha\lambda\ \ \xi_1 + \xi_2 + K + \xi_\nu \geq \nu)$
5	$X \rightarrow \Psi$	$x - \psi \leq 0$
6	$X \dot{\vee} \Psi\ (\Xi\ \xi o\rho\ \Psi)(\varepsilon\xi\chi\lambda\upsilon\sigma\iota\varpi\varepsilon)$	$x + \psi = 1$
6a	$X_1 \dot{\vee} \Xi_2 \dot{\vee} K \dot{\vee} \Xi_\nu$	$x_1 + \xi_2 + K + \xi_\nu = 1$
7	$X \leftrightarrow \Psi\ (\Xi\ \iota\phi\phi\ \Psi)(\varepsilon\theta\upsilon\iota\varpi\alpha\lambda\varepsilon\nu\chi)$	$x - \psi = 0$
7a	$X_1 \leftrightarrow \Xi_2 \leftrightarrow K \leftrightarrow \Xi_\nu$	$x_1 = \xi_2 = K = \xi_\nu$
8	X nor Y $(\leftarrow(\Xi \vee \Psi))$	$x \leq 0, \psi \leq 0$ $(\alpha\lambda\ \xi + \psi \leq 0)$
8a	X_1 nor X_2 nor K nor X_n	$x_1 \leq 0, \xi_2 \leq 0, K, \xi_\nu \leq 0$ $(\alpha\lambda \xi_1 + \xi_2 + K + \xi_\nu \leq 0)$
9	X nand Y $(\leftarrow(\Xi \wedge \Psi))$	$x + \psi \leq 1$
9a	X_1 nand X_2 nand K nand X_n	$x_1 + \xi_2 + K + \xi_\nu \leq \nu - 1$
10	$X \rightarrow \leftarrow \Psi$	$x + \psi \leq 1$
11	$Y \rightarrow \Xi_1 \wedge \Xi_2 \wedge K \wedge \Xi_\nu$	$x_1 \geq \psi, \xi_2 \geq \psi, K, \xi_\nu \geq \psi (\alpha\lambda\ \xi_1 + \xi_2 + K + \xi_\nu \geq \nu\psi)$
12	$Y \rightarrow \Xi_1 \vee \Xi_2 \vee K \vee \Xi_\nu$	$x_1 + \xi_2 + K + \xi_\nu - \psi \geq 0$
13	$X_1 \wedge \Xi_2 \wedge K \wedge \Xi_\nu \rightarrow \Psi$	$x_1 + \xi_2 + K + \xi_\nu - \psi \leq \nu - 1$
14	$X_1 \vee \Xi_2 \vee K \vee \Xi_\nu \rightarrow \Psi$	$x_1 \leq \psi, \xi_2 \leq \psi, K, \xi_\nu \leq \psi (\alpha\lambda\ \xi_1 + \xi_2 + K + \xi_\nu \leq \nu\psi)$
15	$X \wedge (\Psi \vee Z)$	$x = 1,\ \psi + \zeta \geq 1$
16	$X \vee (\Psi \wedge Z)$	$x + \psi \geq 1, \xi + \zeta \geq 1$ $(\alpha\lambda\ \ 2\xi + \psi + \zeta \geq 2)$
17	$X_1 \wedge \Xi_2 \wedge K \wedge \Xi_\nu \rightarrow \Psi_1 \vee \Psi_2 \vee K \vee \Psi_\mu$ (a clause)	$x_1 + \xi_2 + \ldots + \xi_\nu - (\psi + \psi + \ldots + \psi_\mu) \leq \nu - 1$
18	$X_1 \wedge \Xi_2 \wedge K \wedge \Xi_\nu \rightarrow \Psi_1 \wedge \Psi_2 \wedge K \wedge \Psi_\mu$	$m(x_1 + \xi_2 + \ldots + \xi_\nu) - (\psi + \psi + \ldots + \psi_\mu) \leq \mu(\nu - 1)$
19	atleast(k)$(X_1, X_2, \ldots, X_n)$	$x_1 + \xi_2 + \ldots\ \xi_\nu \geq \kappa$
20	atleast(k)$(X_1, X_2, \ldots, X_n) \rightarrow \Psi$	$x_1 + \xi_2 + \ldots\ \xi_\nu - (\nu - \kappa + 1)\psi \leq \kappa - 1$
21	atmost(k)$(X_1, X_2, \ldots, X_n)$	$x_1 + \xi_2 + \ldots\ \xi_\nu \leq \kappa$

22	exactly$(k)(X_1, X_2, \ldots, X_n)$	$x_1 + \xi_2 + \ldots \xi_\nu = \kappa$
23	$Y \leftrightarrow \Xi_1 \vee \Xi_2 \vee \mathrm{K} \vee \Xi_\nu$	$x_1 + \xi_2 + \mathrm{K} + \xi_\nu \geq \psi, \ \xi_1 \leq \psi, \ \xi_2 \leq \psi \mathrm{K}, \xi_\nu \leq \psi$
24	$Y \leftrightarrow \Xi_1 \wedge \Xi_2 \wedge \mathrm{K} \wedge \Xi_\nu$	$x_1 + \xi_2 + \mathrm{K} + \xi_\nu - \psi \leq \nu - 1,$ $\xi_1 \geq \psi, \ \xi_2 \geq \psi \mathrm{K} \ \xi_\nu \geq \psi$

Table 3: Some Formula and their Corresponding Math. Expression

A sharp formulation [Jeroslov/Lowe 1984] (where the solution of the LP relaxation will produce an integer solution) could be obtained when the logical expressions are entirely expressed in a disjunctive normal form. This would, eventually, produce exponentially large expressions in space and time.

(Note that the rule 13 in Table 3 is a special case of rule 20 if k=n and rule 14 is the special case of 20 with k=1). The two columns in Table 3 are equivalent in the sense that they represent the same feasible space. Example: the Boolean expression "*X or Y*" can be represented by the constraint "$x+y \geq 1$", because both expressions represent the same feasible space as can be verified by inspecting the Table 4. For every expression one can verify the transformation using a corresponding TRUE/FALSE table in which all TRUE/FALSE combinations are verified (as in Table 4 for our example).

X	Y	X or Y	x	y	x+y>=1
TRUE	TRUE	TRUE	1	1	1+1>=1 is TRUE
TRUE	FALSE	TRUE	1	0	1+0>=1 is TRUE
FALSE	TRUE	TRUE	0	1	0+1>=1 is TRUE
FALSE	FALSE	FALSE	0	0	0+0>=1 is FALSE

Table 4: Verifying the Correspondence by Enumeration

The trouble with the translation rules in Table 3 is that it produces too many combinations with different logical expressions; in addition, it is not complete. It may be much more advantageous to reduce the logical operators in a first step by applying Boolean laws. The extreme would be to reduce all expressions to a conjunctive (or disjunctive) normal form. But this may produce large expressions growing exponentially with the number of literals. The procedure described in McKinnon & Williams [1989] reduces all logical operators to the two operators ATLEAST and NOT. This is highly efficient and does not produce large expressions. A subsequent flattening procedure reduces the expression even further. They also propose different variants of rules, which produce more or less sharp constraints.
Mitra/Lucas/Moody [1994] propose another translation procedure. They

represent the expression as an expression tree. The tree is traversed bottom-up and at every intermediate node a new 0-1 variable is introduced and the subtree is detached and separately proceeded. The node is replaced by the newly introduced 0-1 variable. This procedure produces too much 0-1 variables which is not really necessary (redundant variables).

I now describe a new translation method implemented and integrated fully into the LPL modeling language. The translation procedure in LPL proceeds by steps where some subexpressions are replaced successively by some others as shown in the subsequent tables. In the tables this is noted as

```
subExpr1 ::= subExpr2
```

which means that *subExpr1* is replaced by *subExpr2*. How exactly this is done internally depends on the implementation. The rules indicated below are independent of any implementation.

Starting point: an expression in LPL using the operators in Table 1.
Step 1: Several logical operators are eliminated at parse time (T1-rules). This step is applied to *every* expression.
Step 2: Several operators are eliminated at evaluation time (T2-rules). This step and the following steps are only applied to model constraints.
Step 3: the NOT operator is pushed inwards within the expression (T3-rules).
Step 4: (optional) The OR operator is pushed inwards over the AND operators (to make the expression more “cnf-like” (T4-rules).
Step 5: The expression is split into several predicates if necessary (T5-rules).
Step 6: All logical operators are eliminated (T6-rules) and linear constraints are generated.

The six steps are applied in a sequential order. Within each set of transformation rules, every rule is applied recursively until no other rule in the same set can be applied, then the procedure goes to the next step. The subsequent tables numbers the rules. Rules marked by a star are alternative rules which have been tried too, but are not applied in the present implementation. Still I wanted to keep them in the tables. For our translation procedure, the reader can ignore them.

STEP 1: PROCEEDING T1-RULES

The first set, containing the T1-rules (Table 5), is applied to any logical expression at parse time. Note also that "#i" always means "cardinality of i".

```
1   X -> Y              ::=   ~X OR Y
1a  X <- Y              ::=   X OR ~Y
2   X NOR Y             ::=   ~(X OR Y)
3   X NAND Y            ::=   ~(X AND Y)
4   AND{i} X[i]         ::=   FORALL{i} X[i]
5   OR{i} X[i]          ::=   EXIST{i} X[i]
6   XOR{i} X[i]         ::=   EXACTLY(1){i} X[i]
7   NOR{i} X[i]         ::=   ATMOST(0){i} X[i]
8   NAND{i} X[i]        ::=   ATMOST(#i-1){i} X[i]

9   X RelOp1 Y RelOp2 z ::=   X RelOp1 Y AND Y RelOp2 z
      (for any RelOp1 or RelOp2 in [=,<>,<,<=,>,>=])

10a  r: X ~< Y           ::=   r : X - Pr < Y
10b  r: X ~> Y           ::=   r : X + Nr > Y
10c  r: X ~= Y           ::=   r : X - Pr + Nr = Y
      (where Pr and Nr are two slack variables)

*2a  X NOR Y             ::=   ~X AND ~Y
*3a  X NAND Y            ::=   ~X OR ~Y
*4a  AND{i} X[i]         ::=   ATLEAST(#i){i} X[i]
*4b  AND{i} X[i]         ::=   ATMOST(0) {i} (~X[i])
*5a  OR{i} X[i]          ::=   ATLEAST(1){i} X[i]
*7a  NOR{i} X[i]         ::=   FORALL{i} (~X[i])
*7b  NOR{i} X[i]         ::=   ATLEAST(#i){i} (~X[i])
*8a  NAND{i} X[i]        ::=   EXIST{i} (~X[i])
*8b  NAND{i} X[i]        ::=   ATLEAST(1){i} (~X[i])
```

Table 5: T1-rules: Eliminate some Operators at Parse Time

The corresponding Boolean operators ->, <-, NOR, NAND, AND{}, OR{}, XOR{}, NOR{} (sometimes also called the NONE-operator), and NAND{} are eliminated and replaced by other constructs where X and Y are any logical expressions and X[i] and Y[i] are indexed expressions. This greatly reduces the number of operators. (The operator "::=" means that the left side is replaced by the right side.) These operators are eliminated at parse time, so the interpreter and evaluator does not see them anymore. The consequence of this is that these operators are eliminated in *all* expressions of a LPL model. This is different of the second set of rules (T2-rules) where further operators are eliminated, but they are only eliminated if there is need to translate the constraint to an IP model.

Keep in mind that the operators ->, NOR, NAND, XOR, and <-> are not associative which means, for example, that "x NOR y NOR z" will not be translated into "~x OR ~y OR ~z" as we might expect. The result will rather be "~(~x OR ~y) OR ~z" or "~x OR ~(~y OR ~z)" depending on whether

NOR is left- or right-associative. (In LPL most operators are left-associative.)

Rule 9 implements the convention that several relational operators in sequence must hold individually. An example shows this practical rule. The expression:

$x \le \psi \le \zeta \le \omega$

for example, is directly translated into the following expression:

$(x \le \psi) \wedge (\psi \le \zeta) \wedge (\zeta \le \omega)$.

The rules 10 introduce slack variables as already discussed in the Reference Manual of LPL.

STEP 2: PROCEEDING THE T2-RULES

The second set of rules (Table 6) eliminates the FORALL and the EXIST operators and simplifies some special cases, such as “EXACTLY(0)” which is the same as “ATMOST(0)”, and ”EXACTLY(<all>)” which is the same as “ATLEAST(<all>)”.

```
11   FORALL{i} X[i]              ::=   ATLEAST(#i){i} X[i]
12   EXIST{i} X[i]               ::=   ATLEAST(1){i} X[i]
13   EXACTLY(0){i} X[i]          ::=   ATMOST(0){i}  X[i]
14   EXACTLY(#i){i} X[i]         ::=   ATLEAST(#i){i} X[i]

*13a EXACTLY(0){i} X[i]          ::=   ATLEAST(#i){i}  (~X[i])
*15  ATMOST(0){i} X[i]           ::=   ATLEAST(#i){i} (~X[i])]
*16  ATMOST(k){i} X[i]           ::=   ATLEAST(#i-k){i} (~X[i])
```

Table 6: T2-rules: Eliminates some Operators at Evaluation Time

Rules 13a, 15, and 16 are not applied in the present implementation. However, this could be a good place to eliminate the ATMOST operator. I have decided not to eliminate it because its presence simplifies several other expressions later on. Only at the very end it will be eliminated.

STEP 3: PROCEEDING T3-RULES

The third set of rules, the T3-rules, pushes the NOT operator as far inwards in the expression as possible (Table 7).

```
20   ~(~X)                       ::=   X
21   ~(X AND Y)                  ::=   ~X OR ~Y
```

```
22   ~(X OR Y)                      ::=   ~X AND ~Y
23   ~(X <-> Y)                     ::=   X XOR Y
24   ~(X XOR Y)                     ::=   X <-> Y
25   ~(X,Y,Z)                       ::=   ~X , ~Y , ~Z
26   ~(X [<=,>=,<,>,=,<>] Y)        ::=   X [>,<,>=,<=,<>,=] Y
27   ~(ATLEAST(k){i} X[i])          ::=   ATMOST(k-1) {i} X[i]
28   ~(ATMOST(k){i} X[i])           ::=   ATLEAST (k+1) {i} X[i]
29   ~(EXACTLY(k){i} X[i])          ::=   ATMOST(k-1){i} X[i]
                                            OR ATLEAST(k+1){i} X[i]
29a** ~(FORALL {i} X[i])            ::=   ATMOST(#i-1){i} X[i]
29b** ~(EXIST {i} X[i])             ::=   ATMOST(0){i} X[i]

30   ~c  (c is a const. expr)       ::=   (no replacement)
31   ~x  (x is an binary var)       ::=   (no replacement)
32   ~z  (z is any other var)       ::=   z<=0
33   ~Z  (Z is an var expr)         ::=   Z<>0

*31a ~x  (x is an binary var)       ::=   x<=0

*27a ~(ATLEAST(k){i} X[i])          ::=   ATLEAST(#i-k+1){i} (~X[i])
*29a ~(EXACTLY(k){i} X[i])          ::=   ATLEAST(#i-k+1){i} (~X[i])
                                             OR ATLEAST(k+1){i} X[i]
** obsolete, if T2-rules have been applied
```

Table 7: T3-rules: Push the NOT Operator inwards

The ~ operator disappears from all expressions except in rules 30 and 31 where c is a constant expression and x is a binary variable. "~c" (if c is a constant) will be interpreted as "0" if c if different from zero and as "1" otherwise. "~x" (if x is a binary variable) will be modified later (see rule 68). If z is a variable (but not a binary variable then "~z" is translated into $z \leq 0$ (rule 32). In all other cases, if Z is a numerical expression containing any variables, rule 33 is applied. It means – as a convention – that the following expression (where *x* is a real variable)

```
~(45*x + 1)
```

is translated into

```
45*x + 1 <> 0
```

Rules 27a and 29a would be needed if the ATMOST-operator had already been eliminated.

Next it would be a good place to eliminate the three operators "<", ">", and "<>" and to apply the rules *34–*36. Again, these rules are postponed until the very end. The reason is that the unequal-operator can be eliminated more elegantly (see rule 90). Since the rule 90 introduces a less and a greater operator, the rules *35 and *36 also are postponed (rules 91 and 92).

```
*34  x <> y                         ::=   x < y OR x > y
```

```
*35  x < y                              ::=   x + e <= y
*36  x > y                              ::=   x >= y + e
```

The rule *34 would eliminate the unequal operator, where x and y are arbitrary expressions. The rules *35 and *36 would replace the greater and the less operator by the greater-equal and less-equal operator. By doing so, we need to introduce a small constant e. If x and y are integer quantities then e can be chosen to be one.

STEP 4: PROCEEDING THE T4-RULES

At the beginning of this step eight logical operators are left: OR, XOR <->, ~, AND, ATLEAST, ATMOST, and EXACTLY. Let us ignore for a moment the last five operators and analyze expression that contain only the three operators OR, XOR, <->. The T3-rules has pushed all NOT operators inwards in such a way that it appears only before 0-1 variables and constant expressions. It is now easy to generate a conjunctive normal form (CNF) [a disjunctive normal form (DNF)] just be applying the rules 40–44 [*40a–*44a] recursively. For the ultimate production of linear constraints, it is important to apply the rules 40–44 (and not *40a–*44a, which yield a DNF). More exactly, we proceed as follows:

1 apply the rules 40 and 41 recursively to eliminate all XOR and <->,
2 apply the T3-rules to every subtree expression to push the NOT operators inwards eventually,
3 apply the rules 42–44 to generate a CNF.

It is straightforward to translate conjunctive normal forms (CNF) into 0-1 constraints. Therefore, we might only need to express all logical expressions as CNFs (as we have done in rules 40–44). Unfortunately, the expression can become exponential in the size of the propositions. To make an expression nevertheless 'more cnf-like' the T4-rules 40–44 are applied only *once* at each subexpression node. This is an arbitrary decision. One could also apply them *twice* at each subexpression. For some expressions this would generate simpler constraints.

```
40   x XOR y                    ::=  (x OR y) AND (~x OR ~y)
41   x <-> y                    ::=  (~x OR y) AND (x OR ~y)
```

```
42   x OR (y AND z)              ::=  (x OR y) AND (x OR z)
43   (x AND y) OR z              ::=  (x OR z) AND (y OR z)
**44   (x AND y) OR (v AND w)     ::=  (x OR v) AND (x OR w) AND
                                      (y OR v) AND (y OR w)

*40a  x XOR y                    ::=  (x AND ~y) OR (~x AND y)
*41a  x <-> y                    ::=  (x AND y) OR (~x AND ~y)

*42a   x AND (y OR z)            ::=  (x AND y) OR (x AND z)
*43a   (x OR y) AND z            ::=  (x AND z) OR (y AND z)
**44a   (x OR y) AND (v OR w)     ::=  (x AND v) OR (x AND w) OR
                                      (y AND v) OR (y AND w)
** rules implied by the others
```

Table 8: T4-rules: Push outwards the AND (OR) Operator

The rules marked by a star in Table 8 must be applied to generate a DNF expression. This can be advantageous if a solver based on resolution is used to solve the problem.

At first sight, it might seem that the following two rules are simpler then 40 and 41.

```
40b   x XOR y                    ::=  x + y = 1
41b   x <-> y                    ::=  x = y
```

However, 40 generates the two linear constraints:

```
x+y >= 1
x+y <=1
```

which is exactly the same as 40b. Rule 40b does not make the problem formulation any sharper! A similar argument counts for 41 and 41b.

At this step, furthermore, a subexpression which is an operand of XOR, <->, or OR is detached if it contains a (logical or mathematical) operator different from one of the four operators XOR, <->, OR, ~, or if it contains operands different from binary variables. The detachment is applied *before* the T4-rules are applied.
Detaching a subexpressions means to replace it by an additional binary variables as explained in the next section (T5-rules).

STEP 5: PROCEEDING THE T5-RULES

The T5-rules decomposes an expression if necessary, that is, certain subexpressions are detached from the main expression and replaced by an additional binary variable. To do the decomposition, let's construct a parse tree

for the corresponding expression. The parse tree is a linked tree where each operators and each operand occupy a node within the tree. (For a extended discussion of parse tree see Aho/Sethi/Ullman [1986].) Traversing the tree in an inorder way yields the expression. For example, the parse tree of

a OR (y ≤ z)

(where *x* is a binary variable and *y* and *z* are real variables) is as follows:

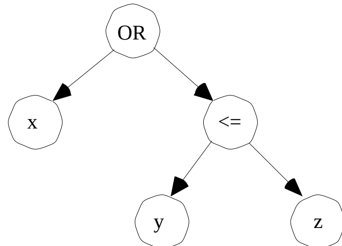


Figure 3: A Syntax Tree

The rule of decomposition goes as follows. Traverse the tree in any order. At each node *n* of the tree – except the root – consider the type of the node and the type of the parent node *pn*. The Table 9 indicates whether at a n–pn (node–parent node) pair the subtree at node *n* must be deconnected (detached) or not. "yes" means "yes, it must be deconnected", "no" means "no, it mustn't".

n \ *pn*	And	Or	Xor	<->	Atleast	Atmost	Exact.	else
And	no	yes	yes	yes	yes	yes	yes	yes
Or	no	no	yes	yes	yes	yes	yes	yes
Xor	no	yes	no	yes	yes	yes	yes	yes
<->	no	yes	yes	no	yes	yes	yes	yes
Atleast	no	yes	yes	yes	yes	yes	yes	yes
Atmost	no	yes	yes	yes	yes	yes	yes	yes
Exact.	no	yes	yes	yes	yes	yes	yes	yes
Not,bar	no	no	no	no	no	no	no	no
X_b	no	no	no	no	no	no	no	no
else	no	yes	yes	yes	yes	yes	yes	no

Table 9: Parent/Child Pair to Decide when to Detach a Subtree

In Table 9 the rows (columns) must be interpreted as follows:

And	means	AND or ATLEAST(all)... or ATMOST(0)...
Or	means	OR or ATLEAST(1)... or ATMOST(all–1)...
Xor	means	XOR or EXACTLY(1)

X_b	means	any expressions containing only binary or no variables.
else	means all other symbols	

“Yes” means that the corresponding subtree at node *n* is detached and replaced by a newly introduced binary variable, say *d*. The detached tree is attached to the definition of *d*.
As an example, let's check for the expression (in Figure 3) (in LPL syntax):

```
CONSTRAINT C: x OR ( y <= z);
```

The table indicates yes only at the node AND (parent is XOR). So we detach it and attach it at a new variable (Figure 4).

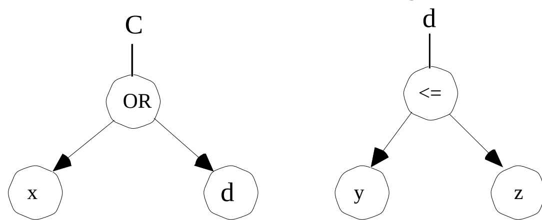


Figure 4: Two syn tax Trees, one Detached

We have now introduced a new predicate *d*. The previous constraint C will be replaced by the following two definitions (in LPL syntax):

```
CONSTRAINT C : x OR d;       (* replacing previous model constraint C *)
VARIABLE d BINARY: y <= z;    (* new predicate definition *)
```

By the way, the LPL compiler after proceeding the constraint C will return (in the debug-file) exactly these two definitions.

STEP 6: PROCEEDING THE T6-RULES

Finally we need to eliminate all logical operators and replace them by mathematical ones. Three sets of rules are applied.

The first set of rules is applied to model constraints, the second and third set of rules are applied to predicate definitions. Each rule of the first set has the form

1>0 --> *Expr1* ::= *Expr1a*

where *Expr1* is the constraint expression. This is nothing else than a (complicated) way of saying that *Expr1* must be true. In fact it imposes the Boolean constraint $TRUE \rightarrow E\xi\pi\rho1$, which is the same as imposing just *Expr1*. The whole rule says that *Expr1* must be replaced by *Expr1a*.

The reason while this slightly more complicated form is used will become clear from the rules in second and third set.

The second set of rules has the form

d>0 --> *Expr2* ::= *Expr2a*

where *d* is the predicate name and *Expr2* is the expression which is attached to the predicate *d*. This expression imposes the Boolean constraint $d \rightarrow Expr2$, that is "if *d* is true then *Expr2* must be true". The whole rule says that d>0 --> *Expr2* must be replaced by *Expr2a*.

The third set of rules has the form:

d=0 --> *Expr3* ::= *Expr3a*

where *d* is the predicate name and *Expr3* is the expression which is attached to the predicate *d*. This expression imposes the Boolean constraint $\neg Expr3 \rightarrow d$ (which is $\neg d \rightarrow Expr3$), that is "if *d* is false then *Expr3* must be true". The whole rule says that d>0 --> *Expr3* must be replaced by *Expr3a*.

Note that only an implication (not an equivalence is enforced. To enforce equivalence between the predicate *d* and an expression *Expr*, a fourth set of rules would be used as:

d>1 <--> *Expr4* ::= *Expr4a*

which is the same as the conjunction of both rules defined as:

d>0 --> *Expr4* ::= *Expr4a*

d=0 --> ~*Expr4* ::= ~*Expr4a*

```
50   1>0 --> X AND Y AND ...            ::=  x, y, ...
51   1>0 --> X OR Y OR ...              ::=  x + y + ... >= 1
*52  1>0 --> X <-> Y <-> Z              ::=  x = y = z          (x=y AND y=z)
*53  1>0 --> X XOR Y XOR Z              ::=  x + y + z = 1
54   1>0 --> ATLEAST(k){i} X[i]         ::=  SUM{i} x[i] >= k
55   1>0 --> ATMOST(k){i} X[i]          ::=  SUM{i} x[i] <= k
56   1>0 --> EXACTLY(k){i} X[i]         ::=  SUM{i} x[i]  = k
57   1>0 --> X                          ::=  X
58   ~x   (x is a binary var)        ::=  x<=0
58a  ~x   (x is a binary var)        ::=  1-x
59   x       (x is a binary variable)   ::=  x>=1

60   d>0 --> X AND Y AND ...            ::=  x >= d , y >= d , ...
61   d>0 --> X OR Y OR ...              ::=  x + y + ... >= d
*62  d>0 --> X <-> Y <-> Z              ::=  d>0 --> x=y AND y=z
*63  d>0 --> X XOR Y XOR Z              ::=  x + y + z = d
64   d>0 --> ATLEAST(k){i} X[i]         ::=  SUM{i} x[i] >= k*d
65   d>0 --> ATMOST(k){i} X[i]          ::=  SUM{i} x[i] <= k*d
66   d>0 --> EXACTLY(k){i} X[i]         ::=  SUM{i} x[i]  = k*d
67   d>0 --> X                          ::=  X >= d

70   d>0 --> ∑ ax <= b           ::=  ∑ ax-b <= U(∑ ax-b)*(1-d)
 71  d>0 --> ∑ ax >= b           ::=  ∑ ax-b >= L(∑ ax-b)*(1-d)
72   d>0 --> ∑ ax = b            ::=  d>0 --> ∑ax >= b AND d>0 --> ∑ax <= b
```

```
80  d=0 --> ∑ ax <= b           ::=  ∑ ax-b <= U(∑ ax-b)*d
81  d=0 --> ∑ ax >= b           ::=  ∑ ax-b >= L(∑ ax-b)*d
82  d=0 --> ∑ ax = b            ::=  d=0 --> ∑ax >= b AND d=0 --> ∑ax <= b

90  x <> y                      ::= d>0 --> x < y ; d=0 --> x > y
91  x < y                       ::=   x + e <= y
92  x > y                       ::=   x >= y + e
```

Table 10: T6-rules: Eliminate All Logical Operators

Rule 50 just says that a conjunction of constraints is considered as a set of constraints. Rule 51 replaces all ORs by the addition operator. The addition must be greater or equal than one. This enforces at least one operand to be true. Rule *52 and *53 are not used since both operator have been eliminated in rules 40 and 41. Rule 57 is the starting rule for every constraint. Rule 58 is applied if "~x" is a complete constraint while 58a is applied if "~x" is a subexpression and belongs to a constraint.
(Rules 65 and 66 do not exactly enforce the intended implication. The correct two rules would be

```
65a  d>0 --> ATMOST(k){i} X[i]    ::=  SUM{i} x[i] <= n -(n-k)*d
66a  d>0 --> EXACTLY(k){i} X[i]   ::=  d>0 --> ATLEAST(k){i} X[i] AND
                                      d>0 --> ATMOST(k){i} X[i]
```

The rule 65 and 66 enforce the implications *supposing d is true*. This can be suppose in most of the time. However the modeler should use EXACTLY and ATMOST with care in a LPL model.)

At this point, all logical operators are eliminated as can be verified by the T6-rules. The rules 50–57 are straightforward and implement the first set of the T5-rules. Rules 60–67 implements the second set of the T5-rules. Hence, rule 67, for example, means that if d is true then X must also be true. The constraint "X >= d" reflects this fact: if d is 1 then X must be atleast 1, otherwise X can be true or not. Similar arguments hold for the other rules.

The rules 70–71 makes the link between mathematical constraints and logical proposition where the expressions $L(X)$ and $U(X)$ are lower and upper bounds on the expression X. Normally, they are applied on a linear constraints such as

$$L \leq \sum_{\varphi=1}^{\nu} \alpha_\varphi \xi_\varphi - \beta \leq Y$$

These bounds may be given by the modeler or they may be calculated automatically from the lower and upper bounds l_j and u_j of the variables x_j involved in the linear expression. (In LPL both solutions are possible.) In the second case, they are calculated using the following formula [Brearley/Mitra/Williams 1975]:

$$L = \sum_{\varphi \in \{\varphi | \alpha_\varphi > 0\}} \alpha_\varphi \lambda_\varphi + \sum_{\varphi \in \{\varphi | \alpha_\varphi < 0\}} \alpha_\varphi \upsilon_\varphi - \beta$$

$$Y = \sum_{\varphi \in \{\varphi | \alpha_\varphi > 0\}} \alpha_\varphi \upsilon_\varphi + \sum_{\varphi \in \{\varphi | \alpha_\varphi < 0\}} \alpha_\varphi \lambda_\varphi - \beta$$

where $l_j \le \xi_\varphi \le \upsilon_\varphi$.

In LPL it is easy to bound variables as well as constraints. A range specification [...,...] must be attached to the corresponding declaration, as in

```
VARIABLE x [100,500];  (* lower and upper bounds of the variable x are 100 and 500 *)
CONSTRAINT R [l,u];    (* lower and upper bounds of the constraint R are l and u *)
```

These bounds are used by LPL to calculate *U* and *L* of a constraint.

Table 11 gives a collection of alternative translation rules that could have been used. These rules are closely related with rules used by McKinnon/Williams [1989]. None of them is applied in LPL.

```
*1b  X AND Y AND ...      ::= ATLEAST (#) (X,Y,...)
*2b  X OR Y OR ...        ::= ATLEAST (1) (X,Y,...)
*3b  X NOR  Y NOR ...     ::= ATMOST  (0) (X,Y,...)
*4b  X NAND Y NAND ...    ::= ATMOST  (1) (X,Y,...)
*3b  X -> Y               ::= ATLEAST (1) (~X,Y)
*6b  X XOR Y XOR ...      ::= EXACTLY (1) (X,Y,...)
*6b  X XOR Y              ::= (X OR Y) AND (~X OR ~Y)
*7e  X <-> Y              ::= ~(EXACTLY(1) (X,Y))
*7b  X <-> Y              ::= (~X OR Y) AND (X OR ~Y)
*7c  X <-> Y              ::= (~X AND Y) OR (X AND ~Y)
*7d  X <-> Y            ::= ATLEAST(2)(ATLEAST(1)(~X,Y),ATLEAST(1)(X,~Y))

*17  ~(X = Y)          ::= ATLEAST (1) (X<Y , X>Y)
*16  ~(X = Y)          ::= (X<Y) OR (X>Y)
*21  EXACTLY(k){i} X[i]   ::= ATLEAST(2) (ATLEAST(k){i}(X[i]),
                               ATMOST(k){i}(X[i]))
```

Table 11: Further Transformation Rules (not used)

This completes the transformation procedure implemented in LPL. Several examples are now given to illustrate the transformation.

EXAMPLES

Example 1: (Y AND Z) OR X

```
Applying rule 41  gives:  (X OR Y) AND (X OR Z)
Applying rule 50 gives:  (X OR Y) , (X OR Z)
Applying rule 51 (twice) gives: (x + y ≥ 1) , (x + z ≥ 1)
```

Example 2: ATLEAST(k){j} X[j]

```
Applying 54 gives:  ∑j xj >= k
```

Example 3: ATLEAST(k){h} (~Y[h])

```
Applying 54 gives :   ∑h (~yh) >= k
Applying 58 gives:   ∑h (1-yh) >= k
```

Example 4: ATLEAST(n){h} Y[h] -> ATLEAST(k){j} X[j]

```
Applying 1  gives:  ~(ATLEAST(n){h} Y[h])  OR  ATLEAST(k){j} X[j]
Applying 27 gives:  ATMOST(n-1){h} Y[h]  OR  ATLEAST(k){j} X[j]
Applying T5-rules introduces two binaries:
                   (1) d1 OR d2
                   (2) d1>0 --> ATMOST{n-1){h} Y[h]
                   (3) d2>0 --> ATLEAST(k){j} X[j]
Applying 51 on (1) gives:  d1 + d2 >= 1
Applying 65 on (2) gives:  ∑h yh <= (n-1)*d1
Applying 64 on (3) gives:  ∑j xj >= k*d2
```

Other examples are now briefly reviewed to see advantages and disadvantages of the rules.

Let's try the expression: x or (y xor z)

LPL first applies the rule 40 to generate directly the CNF-form:

(x or y or z) and (x or not y or not z)

The rules 51 and 58a then produce the two following linear constraints:

$$x + y + z \geq 1$$
$$x - y - z \geq -1$$

Let's see whether these two constraints model our constraint correctly by inspection into the true-table:

x	y	z	x or (y xor z)	x+y+z≥1	x−y−z≥−1
true	true	true	true	true	true
true	true	false	true	true	true
true	false	true	true	true	true
true	false	false	true	true	true
false	true	true	false	true	false
false	true	false	true	true	true
false	false	true	true	true	true
fasle	false	false	false	false	true

The conjunction of the two linear constraint give exactly the same truth function. Hence

$$x + \psi + \zeta \geq 1$$
$$\xi - \psi - \zeta \geq -1 \qquad \text{is the same as } x \vee (\psi \dot{\vee} \zeta)$$
$$\xi, \psi \zeta \in \{0,1\}$$

Let's see now how the expression would have been translated if a T5-rule had detached the subexpression (y XOR z). In this case a new binary variable *d* would have been introduced. The expression now takes the form:

```
x or d
d>0 --> y xor z
```

Applying Rules 51 to the first and 40 to the second gives:

```
x+d≥1
d>0 --> (y or z) and (not y or not z)
```

applying first 60, then 61 and 58a, and finally 70 and 71 we get

```
x+d ≥ 1
y+z-1 ≤ U*(1-d)
y+z-1 ≥ L*(1-d)
```

U can be maximally 1 since y and z can be 1 maximally therefore: U(y+z–1) = 1. L can be minimally –1 since y and z can be minimally 0, therefore: L(y+z–1) = –1 Therefore we get the three linear constraints:

```
x+d ≤ 1
y+z+d ≤ 2
y+z-d ≥ 0
```

Let us interpret the last two linear constraints from a geometrical point of view to see whether they reflect the logical constraint d>0 --> y xor z. It is clear what we want is to forbid that y and z have the same value whenever d is forced to be 1. That's exactly what this constraint says.

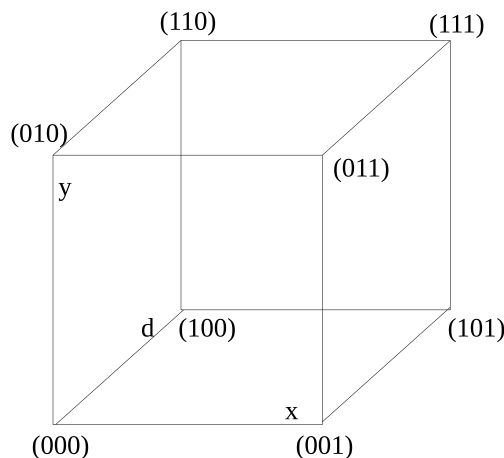


Figure 5: The Three-dimensional 0-1 Simplex

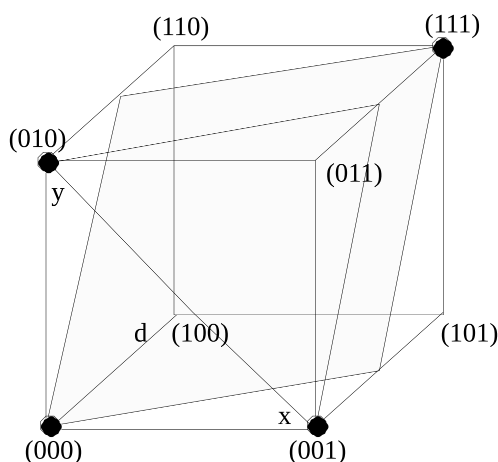
Figure 5 is a simplex that reflects all value combinations of the three variables (d,y,z). We need hyperplanes which exclude the point (111) and the point (100). It is easy to see, that a single hyperplane cannot do the job; but two can. For example, the hyperplane excluding only (111) going through the three points (110), (011), and (101) is $y+z+d \leq 2$, the second constraint. The hyperplane excluding (100) going through the three points (000), (110), and ((101) is exactly $y+z-d \geq 0$, the third constraint.

This second translation generates also an expression which is equivalent to the original expression, *supposing that d is true if x is not true*. However, this last constraint is imposed by the first linear constraint.
However, this translation leads to three linear constraints and an additional binary variable. The resulting constraints are clearly less sharp than the two constraints produced by LPL.

Let's try the expression: d <-> (x AND y)

The expression models the equivalence between d and a conjunction: $d \leftrightarrow (\xi \wedge \psi)$. Graphically, the expression is true for the corners with the small circle and false otherwise. These TRUE-points can be separated from the others by two parallel planes containing the points {(1,1,1), (0.5,0,1),(0,0,0)} and {(0,1,0), (0.5,1,1), (0,0,1)} as following:

$0 \leq \xi + \psi - 2\delta \leq 1$



A sharper formulation uses three planes (Jeroslow) as following.

$-x - y + d \geq -1, \quad x \geq d, \quad y \geq d$

These planes are represented by the points {(010, (111), 001)}, {(111),(101),(000),(010)}, and {(110),(000),(001),(111)} in Figure 6.

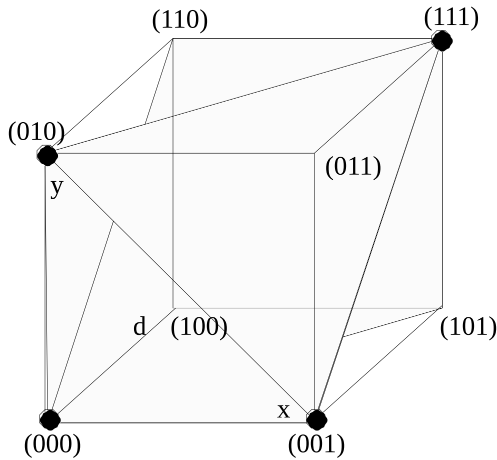


Figure 6: Tighter Planes

Let us see how LPL treat this expression. First the rule 41 is applied giving

```
(not d or (x and y)) and (d or not (x and y))
```

The ~is pushed down giving

```
(not d or (x and y)) and (d or not x or not y)
```

Finally, the AND is pushed up (rule 42).

```
(not d or x) and (not d or y) and (d or not x or not y)
```

Note that the expression is automatically "flattened" (McKinnon), that is the parentheses separating different ANDs can be removed. LPL generates also the sharpest possible form.

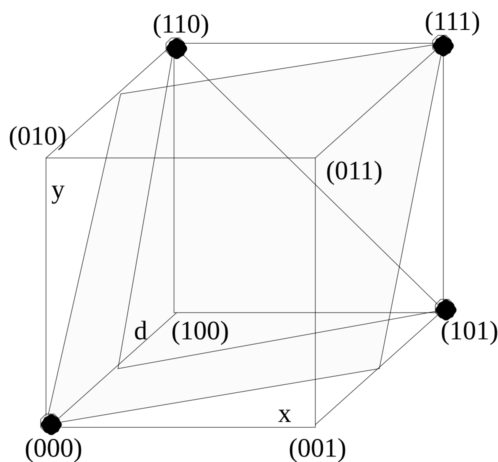
We can generalized these formulas [Darby-Dowman al 1988]. The expression
$d \leftrightarrow (\xi_1 \wedge \xi_2 \wedge \ldots \wedge \xi_\nu)$
can be translated into a 0-1 formulations as
$0 \leq \xi_1 + \xi_2 + \ldots + \xi_\nu - \nu\delta \leq \nu - 1$
or by the sharper formulation:
$-x_1 - x_2 - \ldots - x_n + d \geq 1 - n$
$x_1 - d \geq 0$
$x_2 - d \geq 0$
$\ldots \ldots$
$x_n - d \geq 0$
Since the heuristic principle in rule 42 has been adopted in LPL to apply it only once and to detach a subtree with a root node containing the label AND, LPL will not generate the sharp formulation. It introduces just a single additional binary variable. The advantage is that the expression cannot get an

exponential size in the number of propositions.

Let us try the expression: d <-> x OR y

Likewise we can formulated a equivalence of a proposition with a disjunction as $d \leftrightarrow (\xi \vee \psi)$. Graphically, the expression is true for the corners with the small circle and false otherwise. These TRUE-points can be separated from the others by two parallel planes containing the points {(1,1,0), (0,0,0.5),(1,0,1)} and {(1,1,1), (0.5,0,1), (0,0,0)} as following:

$-1 \leq x + y - 2d \leq 0$



A sharp formulation uses three planes as follows:

$+x + y - d \geq 0, \quad x \leq d, \quad y \leq d$

These planes are represented by the points {(110, 000, 101)}, {(111),(101),(000), (010)}, and {(110),(000),(001),(111)} in Figure 6.

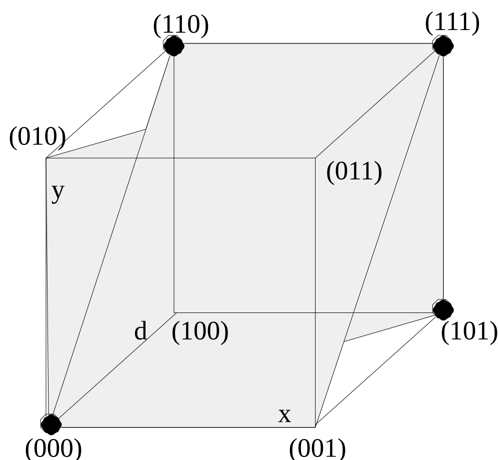


Figure 6: Tighter Planes

Let us see how LPL treat this expression. First the rule 41 is applied giving

```
(not d or x or y) and (d or not (x or y))
```

The ~is pushed down giving

```
(not d or x or y) and (d or (not x and not y))
```

Finally, the AND is pushed up (rule 42).

```
(not d or x or y) and (d or not x) and (d or not y)
```

Note that the expression is automatically "flattened" (McKinnon), that is the parentheses separating different ANDs can be removed. LPL generates also the sharpest possible form.

We can generalized these formulas. The expression

$d \leftrightarrow (\xi_1 \vee \xi_2 \vee \ldots \vee \xi_\nu)$

can be translated into a 0-1 formulations as

$-(n-1) \le x_1 + x_2 + \ldots + x_n - nd \le 0$

or into the sharp formulation:

$x_1 + \xi_2 + \ldots + \xi_\nu - \delta \ge 0$

$-\xi_1 + \delta \ge 0$

$-\xi_2 + \delta \ge 0$

$\ldots \ldots$

$-\xi_\nu + \delta \ge 0$

The same can be said here. If the disjunction only contains two operands then LPL generate the sharp formulation; otherwise an additional binary variable is introduced.

Let us try expression: d <-> (x XOR y)

It is more involving to find a sharp formulation of the equivalence:

$d \leftrightarrow (\xi_1 \dot{\vee} \xi_2 \dot{\vee} \ldots \dot{\vee} \xi_\nu)$

In the simplified version for n=2 with $d \leftrightarrow (\xi \dot{\vee} \psi)$, we have the following sharp formulation:

$x + \psi - \delta \ge 0$

$\xi + \psi + \delta \le 2$

$\xi - \psi + \delta \ge 0$

$-\xi + \psi + \delta \ge 0$

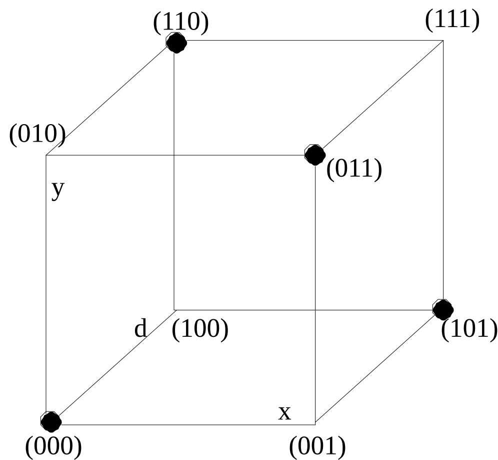


Figure 7

The four inequations represent the planes

{(000),(011),(110)} excluding the point (010)
{(110),(011),(101)} excluding the point (111)
{(000),(011),(101)} excluding the point (001)
{(000),(101),(110)} excluding the point (100).

LPL finds this sharp formulation by first applying rule 41 then 24 giving:

```
(not d or (x xor y)) and (d or (x <-> y))
```

Again the rule 41 and 42 are applied with the not pushed down giving

```
(not d or ((x or y) and (not x or not y)))  and
(d or ((not x or y) and (x or not y))
```

This time the rule 42 is applied on several nodes giving finally the CNF:

```
(not d or x or y) and (not d or not or not y)  and
(d or not x or y) and (d  or x or not y)
```

This translation reveals an extremely important fact about translating logical expressions into mathematical notation: It is significant in which order the rules are applied. This is the main reason why in LPL all rules are partitioned into the six T1-T6-rules classes, which are applied sequentially. However within the classes, the rules are also applied sequentially *on each node of the syntax tree* and they are repeated recursively.

If we do not want to enforce equivalence but implication then we have the formula:
$d \rightarrow (\xi \dot{\vee} \psi)$
which can be formulated by the plane: $x + \psi = \delta$, containing the points: {(000,(110),(101)}.

There are two ways in LPL to formulate this constraint. The first is simply by a constraint:

```
C: d -> (x XOR y);
```

The second way is by defining a predicate as follows:

```
VARIABLE d BINARY : x XOR y;
```

Both formulations generate exactly the same linear constraints

```
x+y <= d
x+y >= d
```

This is the same as the equality constraint. (For the simplex algorithm it is sometimes preferable to replace equality constraints by two inequalities, depending on the implementation.)

Let us try expression: d <-> (x <-> y)

It is also involving to find a sharp formulation of the equivalence:
$d \leftrightarrow (\xi_1 \leftrightarrow \xi_2 \leftrightarrow \ldots \leftrightarrow \xi_v)$

In the simplified version for n=2 with $d \leftrightarrow (\xi \leftrightarrow \psi)$, we have the following sharp formulation:

$-x + y - d \geq -1$

$x - y - d \geq -1$

$x + y + d \geq 1$

$-x - y + d \geq -1$

The feasible points are depicted in Figure 7.

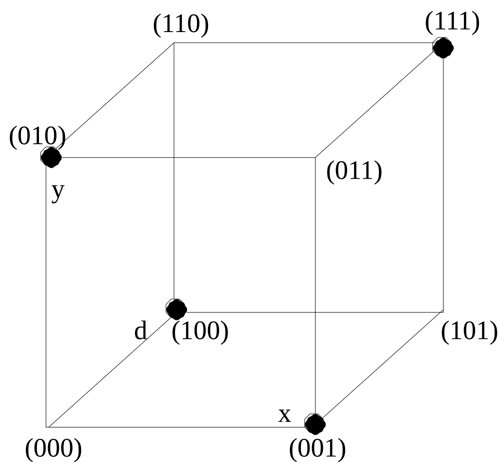


Figure 7

It can be easy seen from Figure 7, that there is no simpler formulation (using less constraints).

5 APPLICATIONS

This section gives a collection of applications where the logical operators and mathematical operators are useful in representing different models.

5.1 PROBLEMS USING DISCRETE (INTEGER) ENTITIES

The most obvious – but also the least important – class of problems for using integer variables is in situations where integral quantities as for example cars, planes, houses etc. must be modeled.
A very small model may show why IP, in these cases, is sometimes necessary. Consider the following model instance [Williams 1990, p.155]

```
! --- LP formulation --- !
VARIABLE x; y;
CONSTRAINT
 R1: -2*x + 2*y >= 1;
 R2: -8*x + 10*y <= 13;
MAXIMIZE A: x + y;
END

! Solution is: x=4, y=4.5 !
```

```
! --- IP formulation --- !
VARIABLE x INTEGER; y INTEGER;
CONSTRAINT
 R1: -2*x + 2*y >= 1;
 R2: -8*x + 10*y <= 13;
MAXIMIZE A: x + y;
END

! Solution is: x=1, y=2 !
```

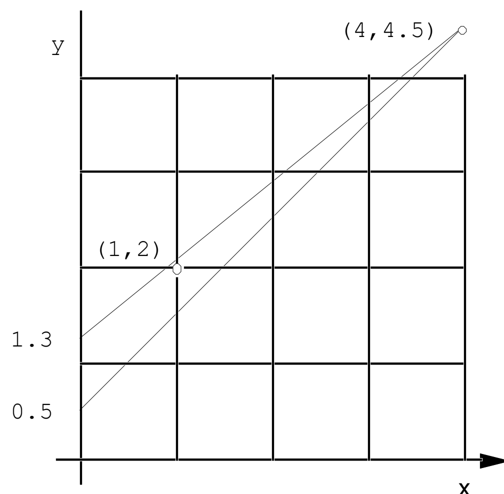


Figure 8: A Very Pointed Space

Figure 8 represent this model instance graphically. If x and y are divisible quantities such as gallons, this model can be treated as an LP problem and the solution will be x=4 and y=4.5. If the variables, however, are restricted to be integral, since they represent indivisible goods, such as aeroplanes, then the optimum solution is quite different: x=1 and y=2 (see Figure 8). This model shows that the rounding of the LP optimum to its nearest integer values (x=4, y=4) can be arbitrarily far off from its integer solution and sometimes even

yields an infeasible solution. Hence, the modeler must carefully decide whether some entities are integers or not.

But this kind of problems does not turn up very often and integer variables should be avoided as much as possible in such situations. For example, there is no need to use integer variables for the number of hens or cows of a nation, although this number is certainly integral! Fractional parts of a solution can easily be cut without any harm to the model solution. While it unquestionably makes a big difference whether *one* or *no* truck travels between two locations, this may not be true for the situation where 2301 or 2300 trucks take the same journey. However, the mathematical reason is that the solution space is almost always very flat for such problems.
Of course, the main reason why integral variables should be avoided, if possible, is that MIP problems are much harder to solve than LP problems. From the modeler's point of view, however, there is no logical reason to make this difference. But the modeling system cannot decide for the modeler in such a situation. The best one can do is to warn the modeler, if he or she defines an integer variable with an upper bound higher than a small positive value.

5.2 Problems with a set of discrete values (multiple choice variables)

Some quantities are allowed to take on only one of several integral values. This occurs in many multiple choice situations. Suppose a variable z may be assigned to a limited number of known values v_i with i={1,..,n}, but it can only be one of them. A natural way to model this kind of problem is to use the XOR operator:

$$z=v_1 \text{ XOR } z=v_2 \text{ XOR } \dots \text{ XOR } z=v_n$$

which means precisely that 'either ($z=v_1$ is true) or ($z=v_2$ is true) or ...'.
Using LPL this can be formulated as

```
CONSTRAINT MultipleChoice: XOR(z=v[1] , z=v[2] , .... , z=v[n]);
```

(Note that

```
CONSTRAINT Choice: z = (v[1] XOR v[2] XOR .... XOR v[n]);
```

is not the same. The expression (v[1] xor v[2] xor .... xor v[n]) can be evaluated immediately and returns true or false (zero or one) depending on the values v[i]: if just one of them is different from zero, the whole expression is one otherwise it is zero. Therefore, the whole equation will be reduced to z=1 or z=0 and produces a fixed bound for z.)

A more concise formulation of the constraint *MultipleChoice* is

```
CONSTRAINT MultipleChoice: XOR{i} (z=v[i]);
```

where XOR{} is the exclusive OR operator extended to a list and means 'there exist exactly one *i* so that *z=v[i]*'.
Note that this model fragment is not an explicit MIP formulation of this problem. It can, however, be converted manually to an MIP formulation by introducing explicitly a 0–1 variable *x* for every *i* meaning that

x_i = \B\LC\{(\S(0 if $z \neq v_i$, 1 if $z=v_i$))

and by replacing the logical constraint *MultipleChoice* by the two linear constraints *S* and *T*. In LPL this can also be formulated as

```
VARIABLE  x{i} BINARY;
CONSTRAINT  S: z = SUM{i} v[i]*x[i];
         T: SUM{i} x[i] = 1;
```

The constraint *T* certifies that only one of all *x* variables has the solution 1, and all others must be 0. Constraint *S*, on the other hand, fixes *z* to exactly one of the *v* data. This is precisely what is needed: *z* must be either v_1 or v_2 or ... or v_n.
An alternative translation of the *MultipleChoice* constraint into an MIP form with a smaller solution space and an additional constraints is

```
VARIABLE  x{i} BINARY;
CONSTRAINT  S{i}: z = v[i]*x[i];
          T: SUM{i} x[i] = 1;
```

By default, LPL will translate the logically *MultipleChoice* constraint into the last MIP-form. By te way, The constarint T is a SOS1 constraint type and can be handled very efficiently by a MIP-solver.

5.3 Problems using dichotomies

More common are applications where the choice is limited to be binary. Consider an event that may or may not occur. This dichotomy can be modeled by introducing a binary (or logical) proposition *x*

x = \B\LC\{(\S(true if the event occurs,false if the event does not occur))

Since it is unknown whether the event occurs or not, the value of the proposition is unknown. Like any other unknown this is modeled by a 0–1 variable in LPL. In LPL (like in C), the values TRUE and FALSE can be simulated by the integer values 0 and 1. A logical proposition is defined as a 0–1 variable *x* using the following syntax

```
VARIABLE x BINARY;
```

Example: The 0–1 knapsack problem:

Suppose there are n projects. Each project i={1..n}, if realized, has a certain reward of p_i and costs the amount w_i. Each project is either done or not. It is not possible to realize a fractional part of a project. There is also a global budget c, available to realize some or all projects. The problem is to select a subset of projects that maximizes the sum of rewards without violating the budget constraint. This problem is also called *the budget selection* or *the capital budgeting problem*. By introducing a binary variable for every project where the dichotomy 'project i is done – project i is not done', this problem can be formulated as

```
SET       i;
PARAMETER     p{i}; w{i}; c;
VARIABLE      x{i} BINARY;
CONSTRAINT    R: SUM{i} w[i]*x[i] <= c;
MAXIMIZE obj: SUM(i) p[i]*x[i];
END
```

This problem is called the 0–1 knapsack problem because of the analogy to the hiker's problem of deciding what to put in a knapsack, given a weight limitation.

Although the 0–1 knapsack problem can be formulated as an IP problem and, therefore, be solved by a general IP solver, this would be highly inefficient for large instances. Many far more specialized solver procedures exist for this and similar problems [Martello & Toth 1990]. However, for small n, the solution of this IP problem – when using a general solver – may be found in a reasonable amount of time. Furthermore, the formulation given above, together with the definition data, *is* a complete model of the problem at hand.

5.4 Problems with a concave solution space

A model with a concave feasible space cannot be modeled using LP. The feasible space of an LP is always convex, since a set of constraints can be interpreted as an intersection of all constraints in the set. In LPL notation this means that the semicolon ';' between the constraints of the CONSTRAINT statements can be interpreted as AND operator.

Hence the region ABCD in Figure 9 normally formulated in LPL as

```
CONSTRAINT
 ABCD1: x<=4;                                            (1a)
 ABCD2: y<=2;
```

can also be formulated as

```
CONSTRAINT ABCD: (x<=4) AND (y<=2);                              (1)
```

The region AEF can be formulated as:

```
CONSTRAINT
 AEF1: y<=x;                                                  (2a)
 AEF2 4*x+3*y<=24;
```

Again in LPL, it is possible to state this as

```
CONSTRAINT AEF: (y<=x) AND (4*x+3*y<=24);                           (2)
```

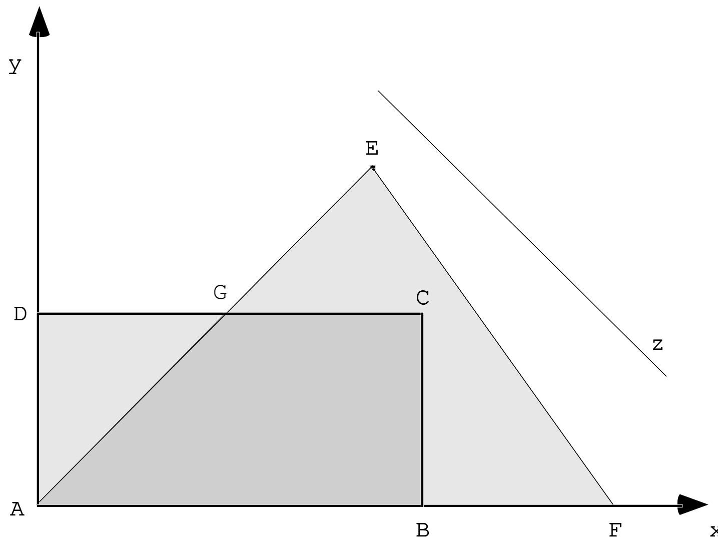


Figure 9: A Concave Space

The region AGCB again is convex and is defined as the intersection between ABCD and AEF. The 'traditional' way to formulate the intersection of ABCD and AEF is to state explicitly all linear constraints which are necessary to model the space AGCB as in

```
CONSTRAINT
 ABCD1: x<=4;
 ABCD2: y<=2;
 AEF1: y<=x;
 AEF2: 4*x+3*y<=24;
```

Another way to model the intersection using LPL is simply to write

```
CONSTRAINT AGCB : ABCD AND AEF;
```

Now, the union of ABCD and AEF is the region ADGEF which is not convex. The non-convex space can be formulated using MIP modeling, by introducing two indicator variables, *d1* and *d2*, and by modeling the space with the following five constraints (where *M* is a convenient upper bound on *x* and *y*).

```
CONSTRAINT
 A1: 4-x + M*d1 >= 0;
 A2: 2-y + M*d1 >= 0;
 A3: x-y + M*d2 >= 0;
 A4: 24-4*x-3*y + M*d2 >= 0;
```

```
A5: d1+d2 = 1;
```

It is easy to see that the five constraints, together, model the union of ABCD and AEF: If *d1*=1 (*d2*=0) then the feasible space is AEF (as defined by the constraint AEF), and if *d1*=0 (*d2*=1) then the feasible space is ABCD (as defined by the constraint ABCD).

In LPL, there exists a more concise way to formulate the region ADGEF: simply be connecting the two regions ABCD and AEF with the union operator OR like the intersection above

```
CONSTRAINT ADGEF: ABCD OR AEF;
```

If the subspaces ABCD and AEF have not been defined before within the model, then the union can simply be stated as

```
CONSTRAINT ADGEF: x<=4 AND y<=2 OR y<=x AND 4*x+3*y<=24;
```

LPL now does the job to translate this logical formulation into a MIP formulation. Of course, bounds on the variables (or the constraints must exists i order to generate the bound M.

5.5 Satisfiability problems

Logical inference means to deduce logical propositions from a set of premises. The conventional way to solve logical problems is by truth tables, Boolean algebra (transformation laws), and by a well known tree searching procedure, called *resolution*. Another *symbolic* method to this inference problem is natural deduction. Still another way is to use *numeric* (or quantitative) methods to solve an inference problem; that is to convert the Boolean expressions into linear (or non-linear) constraints in order to prove the logical argument using mathematical programming. This has two advantages [Chandru al]: a) in some cases this leads to faster algorithms, b) the analysis of the quantitative models leads to the unveiling of hidden mathematical structure. Two further advantages are: c) symbolic (logical) and numeric (mathematical) knowledge can be mixed and represented in the same framework; d) default or other non-monotonic knowledge can also be modeled using the same framework [Yager].

The general SAT problem is as following: Let E and F be two arbitrary Boolean statements. The problem is to decide whether the statement: $E \rightarrow \Phi$ is satisfiable. One way to solve this problem using mathematical programming is to translate E and F into two CNF. From there it is easy to build the two linear

system $Ax \geq \alpha$ and $Bx \geq \beta$. Then we solve the problem [Hooker 1988a p. 52]:

$$\begin{aligned} \min \quad & x_0 \\ & x_0 e + B\xi \geq \beta \\ & A\xi \geq \alpha \\ & \xi \in \{0,1\} \end{aligned}$$

(where e is a column unit vector). If the minimum value x_0 is 1, then E implies F. Another method is to pick an arbitrary inequations (clause) within $Bx \geq \beta$, say $B^{(k)}x \geq \beta_\kappa$, and to solve the linear problem

$$\begin{aligned} \min \quad & B^{(k)}x \\ & B^{(i)}x \geq \beta_\iota \quad \forall \iota\!: \iota \neq \kappa \\ & A\xi \geq \alpha \\ & \xi \in \{0,1\} \end{aligned}$$

if $\lceil B^{(k)}x^* \rceil \geq \beta_\kappa$ (with the minimal solution x^*), then E implies F. (LPL adopts this second method.) Interesting, the expression $B^{(k)}x^*$ needs not to be integer. Hooker [1988a] asserts that 88% of all random generated SAT problems can be solved this way without branching.

An simple example is the expression:

$$(\bar{x} \vee \bar{\psi} \vee \zeta) \wedge (\xi \vee \bar{\psi} \vee \zeta) \mid\mid\stackrel{?}{\rightarrow} (\bar{\psi} \vee \zeta)$$

We solve the linear system:

min $-\psi - \zeta$

συβφεχτ το

$$- \xi - \psi - \zeta \geq -2$$

$$\xi - \psi - \psi \geq -1$$

The LP-relaxation of this problem has a solution of –1.5. Since $\lceil -1.5 \rceil = -1$. The implication follows. Note that the LP-relaxation do not have a integer solution. Nevertheless, we can conclude that the implication is true. (The reason why the LP-relaxation solves the problem is that all clauses in the CNF-form are Horn-clauses.) By the way, it is easy to see that $(\bar{y} \vee \zeta)$ follows from $(\bar{x} \vee \bar{\psi} \vee \zeta) \wedge (\xi \vee \bar{\psi} \vee \zeta)$, since it is a resolvent in the resolution procedure, or in other words $-y - z \geq -1$ is a Chvatal-cut of $-x - y - z \geq -2$ and $x - \psi - \zeta \geq -1$. In Figure 10 the two linear constraints are defined by the points {(1,1,0), (0,1,1), (1,0,1)} and {(1,1,0), (0,1,1), (1,0,1)}, the first excludes point (1,1,1) and the second excludes the point (1,1,0). The Chvatal cut $-y - z \geq -1$ goes through the four points {(0,1,0), (0,1,1), (1,0,0), (1,0,1)} and excludes both points at the same time. Therefore, it is a sharp formulation.

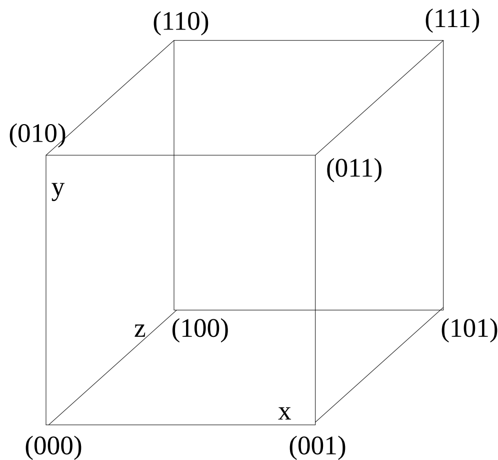


Figure 10

Several examples [Mendelson 1964, p.23–24] now show how to model Boolean expressions using LPL. An IP formulation of both problems was also given by [Williams 1977].

Example 1: a simple SAT-problem

Is the following argument correct? "If fallout shelters are built, other countries will feel endangered and our people will get a false sense of security. If other countries will feel endangered they may start a preventive war. If our people will get a false sense of security, they will put less effort into preserving peace. If fallout shelters are not built, we run the risk of tremendous losses in the event of war. Hence, either other countries may start a preventive war and our people will put less effort into preserving peace, or we run the risk of tremendous losses in the event of war."

The conventional way to handle such problems is to introduce Boolean variables (propositions) for the English sentences. Hence, six variables are introduced with the following meaning

p means "fallout shelters are built"
q means "other countries will feel endangered"
r means "our people will get a false sense of security"
s means "other countries may start a preventive war"
t means "our people will put less effort into preserving peace
u means "we run the risk of tremendous losses in the event of war"

The variables are connected by logical operators in order to obtain Boolean expressions according to the English sentences, as follows

(1) $p \rightarrow \theta \wedge \rho$

(2) $\theta \rightarrow \sigma$

(3) $\rho \rightarrow \tau$

(4) $\leftarrow \pi \rightarrow \upsilon$

(5) ΠΡΟςΕ : $\sigma \wedge \tau \vee \upsilon$

To prove the argument, one must check, whether the Boolean expression

$(1) \wedge (2) \wedge (3) \wedge (4) \rightarrow (5)$

is valid. This is the same as checking whether

$(1) \wedge (2) \wedge (3) \wedge (4) \wedge \leftarrow (5)$

is a contradiction (used in resolution). If the last expression is a contradiction, the argument holds, otherwise it does not hold.

One way to solve this problem is to translate the Boolean expressions into a conjunctive of clauses (cnf-form) and then to apply the resolution procedure. The last expression (5) which is to be proved is negated and added to the list of clauses. The cnf-form of this example – with (5) negated – is then

(1a) $\leftarrow \pi \vee \theta$

(1β) $\leftarrow \pi \vee \rho$

(2) $\leftarrow \theta \vee \sigma$

(3) $\leftarrow \rho \vee \tau$

(4) $\pi \vee \upsilon$

(5α) $\leftarrow \sigma \vee \leftarrow \tau$

(5β) $\leftarrow \tau \vee \leftarrow \upsilon$

Using the resolution procedure (of PROLOG for example), it is simple to prove the contradiction of the conjunctions of the clauses, hence the argument (5) is correct.

Another way to formulate this problem is using LPL's Boolean operators. The problem can then be stated as following

```
(* LPL formulation of a simple SAT-problem (satisfiability problem) *)
MODEL sat1;
VARIABLE
 p BINARY; q BINARY; r BINARY; s BINARY; t BINARY; u BINARY;
CONSTRAINT
  s1:  p -> q AND r;
  s2:  q -> s;
  s3:  r -> t;
```

```
 s4:  ~p -> u;
PROVE s5: s AND t OR u;
WRITE : IF(s5,'The conclusion is correct','The conclusion is not correct');
END
```

The WRITE statement reports if the argument is true or not: if *s5* is true then the WRITE will report “The conclusion is correct” otherwise it reports “The conclusion is not correct”.

Another way to formulate the argument using LPL is to write one single Boolean constraint instead of five separate expressions as follows

```
PROVE s1: ((p -> q AND r) AND (q -> s) AND (r -> t) AND (~p -> u))
         ->  s AND t OR u) ;
```

This formulation suggests that a list of constraints is connected with the AND operator. This is not only true for Boolean expressions, it is true in general: since *the feasible region of a set of constraints is defined as the intersection of the feasible region of each constraint,* several constraints can by formulated as one single big constraint using the AND operator as seen in the 'concave solution space' example above.

Another way to solve the problem, using a general IP solver, is to translate each Boolean expression (manually) into a linear constraint using the translation rules in Table 3 and formulating the resulting constraints in LPL. A resulting IP model might be

$$(1a)\quad p \le \theta$$
$$(1\beta)\quad \pi \le \rho$$
$$(2)\quad \theta - \sigma \le 0$$
$$(3)\quad \rho - \tau \le 0$$
$$(4)\quad (1-\pi) - \upsilon \le 0$$
$$(5)\quad \sigma + \tau + 2\upsilon \ge 2$$

A way to test whether (5) holds in any case is to minimize the expression $s+\tau+2\upsilon$ subject to (1a)–(4). If the minimal value of the objective function is at least 2, then (5) holds and the conclusion is correct, otherwise the conclusion is not correct. The corresponding LPL model is

```
VARIABLE
 p BINARY; q BINARY; r BINARY; s BINARY; t BINARY; u BINARY;
CONSTRAINT
 s1a: p <= q;
 s1b: p <= r;
 s2:  q - s <= 0;
 s3:  r - t <= 0;
```

```
 s4: (1-p)-u <= 0;
MINIMIZE s5: s+t+2*u;
WRITE : IF(s5>=2,'The conclusion holds','The conclusion does not hold');
END
```

Since LPL translates the Boolean expression automatically into an IP model, there is no need to do the translation manually. The first formulation will do the job as well as the explicit IP formulation of the model. The PROVE statement is automatically translated into a MINIMIZE function and the corresponding objective value *s5* will be one (true) or zero (false), if and only if the expression $s \wedge \tau \vee \upsilon$ is true (false).

Example 2: another simple SAT problem

Is the following set of statements consistent? “If the bond market goes up or if interest rates decrease, either the stock market goes down or taxes are not raised. The stock market goes down when and only when the bond market goes up and taxes are raised. If interest rates decrease, then the stock market does not go down and the bond market does not go up. Either taxes are raised or the stock market goes down and interest rates decrease.”

Using the four symbols for the propositions

- *p* means “the bond market goes up”
- *q* means “interest rates decrease”
- *r* means “the stock market goes down”
- *s* means “taxes are not raised”

this problem can be formulated as

(1) $(p \vee \theta) \rightarrow (\rho \vee \sigma)$

(2) $\rho \leftrightarrow (\pi \wedge \leftarrow \sigma)$

(3) $\theta \rightarrow (\leftarrow \rho \wedge \leftarrow \pi)$

(4) $\leftarrow \sigma \vee (\rho \wedge \theta)$

The statements can be written in LPL as

```
VARIABLE
 p BINARY;  q BINARY;  r BINARY;  s BINARY;
CONSTRAINT
 s1:  (p OR q) -> (r OR s);
 s2:  r <-> (p AND ~s);
 s3:  q -> (~r AND ~p);
 s4:  ~s OR (r AND q);
PROVE s4;
WRITE : IF(s4,'The statements are consistent','The statements are not consistent');
END
```

The PROVE statement solves the satisfiability problem. If the problem is

solved to optimality (which means that the problem is feasible), the identifier s4 is true. Any constraint could be taken as objective function.

An alternative explicit IP formulation is the following LPL model

```
VARIABLE p,q,r,s BOOLEAN;
CONSTRAINT
 s1a:  p - r - s <= 0;
 s1b:  q - r - s <= 0;
 s2a:  -p - r + s <= 0;
 s2b:  -p +2*r + s <= 1;
 s3:  p + q + r <= 1;
 s4a:  -q + s <= 0;
 s4b:  -r + s <= 0;
MINIMIZE p;  (* anything can be minimized , one only needs to test feasibility *)
END
```

If the problem is feasible then the statements are consistent otherwise they are not.

Example 3: Hamiltonian Circuit formulated as a SAT-problem

The Hamiltonian Circuit (HC) problem consists of finding a tour through a directed graph that visits all nodes exactly once. Clearly, this problem is NP-complete in general, since it can be formulated as a SAT-problem.

The definition of the HC problem implies that, for any solution, each node must have exactly one incoming edge and exactly one outgoing edge to build the path. 'Exactly one' can be conveniently formulated using the logical operator XOR, which is the exclusive OR operator. This suggests the following compact LPL formulation of the HC problem

```
 (* The Hamiltonian path problem *)
SET nodes; edges{nodes,nodes};
VARIABLE IsIn{edges} BINARY; "indicates whether an edge is in the circuit OR not"
CONSTRAINT
 Cons: AND{i=nodes}
         (XOR{j=nodes|edges[i,j]} IsIn[i,j] AND
          XOR{j=nodes|edges[j,i]] IsIn[j,i]);
END
```

It should be noted that the HC problem is to find a Hamiltonian path, but not necessary an optimal one. To find any such path it suffice to satisfy the condition *Cons*. (Unfortunately, the formulation above does not eliminate subtours. Additional constraints are needed in general. However, we are not interested in this context by solving the problem, but to show how certain constraint can be written using logical formula.) In LPL, one need only add the instructions

```
PROVE Cons;
WRITE IsIn;
```

to get a Hamiltonian path, eventually. The PROVE statement solves the feasibility problem and the WRITE statement outputs the *IsIn* variables.

An instance of a graph is shown in Figure 11 [De Jong & Spears, 1989]: Five nodes are given: a, b, c, d, and e together with the following seven edges: ab, bc, cd, db, de, ea, and eb.

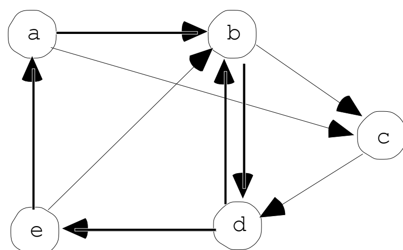


Figure 11: A Directed Graph

To see that the constraint *Cons* represents a necessary (but not sufficient) condition for a Hamiltonian Circuit, one only needs to verify the formula for several nodes:
For the node e, for example, the following condition must hold:

IsIn[de] AND (EITHER IsIn[ea] OR IsIn[eb])

For the node b, for example, the following condition must hold:

(EITHER IsIn[ab] OR IsIn[db] OR IsIn[eb]) AND IsIn[bc]

The construct EITHER ... OR is exactly what is expressed by the XOR operator. For every node a similar condition must hold.
Together with the instructions which define the instance of the graph

```
SET nodes = / a b c d e /;
   edges = / a b, a c, b c, b d, c d, d (b e), e (a b) /;
```

a complete HC problem is formulated in LPL which can be processed automatically. (Note that this formulation does not eliminate subtours).

Example 4: The graph clique problem

A clique C of a graph G is a subgraph of G such that all vertices in C are connected to all other vertices in C. The graph clique problem is to find the clique in a graph with the largest number of vertices. It is easy to formulated this problem with LPL. A predicate *IsIn* for each vertex is introduced which is true if the corresponding vertex is in the clique and false if it is not in the clique. Vertices and edges are given by two sets *nodes* and *edges* as in the Hamiltonian path problem:

```
SET nodes; edges{nodes,nodes};
```

```
VARIABLE IsIn{nodes} BINARY;
```

The objective is to maximize the number of vertices (note that the value of a logical variable is 0 or 1):

```
MAXIMIZE obj: SUM{nodes} IsIn;
```

subject to the constraint of every pair i and j of vertices which are not adjacent to each other that if the vertex i is in the clique then vertex j is not in the clique. This can be formulated as

```
CONSTRAINT CliqueConstraint{i,j| ~edges[i,j]}: IsIn[i] -> ~IsIn[j];
```

A more economical way to formulate the constraint is to introduce the complement of edges

```
SET compl{i in nodes,j in nodes} = ~edges[i,j];
```

and to write the constraint as

```
CONSTRAINT CliqueConstraint{compl[i,j]}: IsIn[i] -> ~IsIn[j];
```

The constraint will be translated into the linear constraints:

```
CONSTRAINT CliqueConstraint{compl[i,j]}: IsIn[i] - IsIn[j] <= 1;
```

Is there a more concise way to formulate the clique problem?

5.6 Mathematical problems with logical conditions

Purely logical problems are not seen very often in commercial applications. But mathematical models extended with logical conditions, are quite common. Several examples illustrate these ideas.

Example 1: a blending problem

In a multi-period food blending problem [Williams 1978] the modeler wants – in addition to the common mathematical balancing and capacity constraints – to impose the following constraints

1) The food may never be made up of more than three oils (ingredients).
2) If an oil is used, at least 20 tons and at most 200 tons must be used.
3) if either Veg1 or Veg2 is used then Oil3 must also be used

The second condition is quite common in many blending problems to limit the quantity of an ingredient or to rule out small quantities used in a blend. Traditionally, this is imposed by introducing a 0–1 variable d_i for every ingredient i with the following meaning in our example

$d_i = 1$, if the ingredient (oil) i is used in the food

$d_i = 0$, otherwise

and by 'linking' this variable to the unknown quantity x_i within the blend: "d_i =1, if x_i is between the lower and upper bound, otherwise d_i =0".
The three conditions can now be imposed by introducing the following constraints, where x_i is the quantity of ingredient i in the mixture, L_i and U_i are the corresponding lower and upper bounds.

$$
\begin{array}{ll}
(1) & \sum_{i=1}^{\nu} d_i \leq 3 \\
(2\alpha) & \xi_\iota - \Lambda_\iota \delta_\iota \geq 0 \\
(2\beta) & \xi_\iota - Y_\iota \delta_\iota \leq 0 \\
(3\alpha) & \delta_{\varsigma\varepsilon\gamma 1} - \delta_{O\iota\lambda 3} \leq 0 \\
(3\beta) & \delta_{\varsigma\varepsilon\gamma 2} - \delta_{O\iota\lambda 3} \leq 0
\end{array}
$$

Using LPL this model fragment is formulated as

```
SET i = /Veg1 Veg2 Oil1 Oil2 Oil3/;    (* ingredients *)
PARAMETER L{i}; U{i};                  (* lower and upper bounds *)
VARIABLE x{i};            (* quantity of ingredient used in the mixture *)
   d{i} BINARY;          (* =1, if i is used in the mixture *)
CONSTRAINT
 Cond1:      SUM{i} d <= 3;
 Cond2a{i}: x - L*d >=0;
 Cond2b{i}: x - U*d <=0;
 Cond3a:    d['Veg1'] - d['Oil3'] <= 0;
 Cond3b:    d['Veg2'] - d['Oil3'] <= 0;
```

The condition *Cond3* (*Cond3a* and *Cond3b* in the model above) could also have been formulated as

```
 Cond3:     d['Veg1'] + d['Veg2'] - 2*d['Oil3'] <= 0;
```

However, this formulation generates a less tight feasible set for the LP relaxation. The solution runtime of the whole model with both formulations of the third constraint *Cond3* was compared in [WILLIAMS 1978] and he found a dramatic difference. The first formulation was by far better and the solution was found in a fraction of time compared with the second formulation, although the first formulation contains more constraints.

Another way to formulate the three conditions in LPL makes use of logical operators. First, a 0–1 variable d_i for every ingredient i is introduced as in the first formulation. But this time, it may be interpreted differently. Instead of interpreting it as a 0–1 variable, it is interpreted as a predicate and declared in LPL as following

```
VARIABLE d{i} BINARY : x>=L AND x<=U;
```

This declaration means that d_i is a predicate or a logical proposition with the meaning

"if ingredient i is used with quantity x_i (i.e. d_i is true), then $L_i \leq \xi_\iota \leq Y_\iota$ is true"

LPL generates automatically the two following constraints from the predicate declaration:

$$x_i - Y_\iota \delta_\iota \leq 0$$

$$\xi_\iota - \Lambda_\iota \delta_\iota \geq 0$$

The three condition can now be formulated as

1) At most 3 out of all d_i are true
2) Simply link d_i to the variable x_i and impose lower and upper bounds on x_i.
3) d_{Veg1} or d_{Veg2} imply d_{Oil3}.

Using LPL syntax the three conditions can be formulated as

```
VARIABLE d{i} BINARY : x>=L AND x<=U;
CONSTRAINT Cond1: ATMOST(3){i} d[i];
CONSTRAINT Cond3: d['Veg1'] OR d['Veg2'] -> d['Oil3'];
```

Note that it is not necessary to formulate the second condition explicitly as a constraint since it was imposed by the definition of the predicate d_i itself.

Still a different way to formulate this problem would be to impose the bounds on the x variable and to declare the predicate d as follows:

```
VARIABLE x{i} [L,U];
VARIABLE d{i} BINARY ~: x=0;
```

The second line imposes the constraints $(x_i = 0) \rightarrow \leftarrow \delta_\iota$, which is the same as $d_i \rightarrow (\xi_\iota \neq 0)$. In our context – since x_i is bounded – this is equivalent to $d_i \rightarrow ((\xi_\iota \geq \Lambda_\iota) \wedge (\xi_\iota \leq Y_\iota))$, the predicate defined above. Hence the two formulations are equivalent. However, the declaration

```
VARIABLE x{i} [L,U];
```

imposes hard lower bounds on x_i , hence x_i cannot become zero (unless L_i is less or equal zero). This means that all predicates d_i all always true. What we need in this context is a mechanism to declare real variable as *semi-continuous*, which means that a variable can be zero or between a lower and upper bound. A possibility would be to declare the variable x as:

```
VARIABLE x{i} [L,U] SEMICONTINOUS;   (* not LPL conform syntax *)
```

(XA is the first MIP solver that supports semi-continuous variables directly.)

Another way, using a more general notation, would be

```
VARIABLE x{i} [0,L..U];   (* not LPL conform syntax *)
```

This last formulation would allow the modeler to declare variables that are defined only in a finite number of ranges:

```
VARIABLE x{i} [L1..U1,L2..U2,L3..U3];   (* not LPL conform syntax *)
```

Semi-continuous variables are not yet implemented in LPL.

Example 2: a logical statement

Another example [McKinnon & Williams 1989] is the following statement: "If there are at most 2 kg of ingredients A or at most 3 kg of ingredients B in a mixture then there must be at most 8 kg of ingredients E or at least 7 kg of ingredients F and vice versa."
Supposing the (unknown) number of kilograms of each ingredient is *a*, *b*, *e*, and *f*, then this statement can be formulated using LPL simply as

```
CONSTRAINT S: a<=2 OR b<=3 <-> e<=8 OR f>=7;
```

To translate the logical constraint into pure mathematical constraints, LPL produces four predicates as follows:

```
VARIABLE p1 BINARY : a<=2;
VARIABLE p2 BINARY : b<=3;
VARIABLE p3 BINARY : e<=8;
VARIABLE p4 BINARY : f>=7;
```

The original constraint *S* is now replaced by

```
CONSTRAINT S: p1 OR p2 <-> p3 OR p4;
```

Next, *S* is translated (by LPL) into a CNF-form as follows:

```
CONSTRAINT S: (p1 OR p2 OR ~p3) AND (p1 OR p2 OR ~p4) AND
              (~p1 OR p3 OR p4) AND (~p2 OR p3 OR p4);
```

The four linear constraints are now easy to derive. The four predicates are also translates in a straightforward way using the rules 70 and 71. However it should be noted that lower and upper bounds are necessary for the translation. (If no bounds are imposed explicitly on the variables, LPL take the lower bounds of zero and sets the upper bounds to the arbitrary value of 1E6.)

An alternative to the translation procedure just exposed would be to decompose the constraint

```
CONSTRAINT S: p1 OR p2 <-> p3 OR p4;
```

further and to introduce two additional predicates as follows

```
VARIABLE p5 BINARY : p1 OR p2;
VARIABLE p6 BINARY : p3 OR p4;
```

The constraint *S* is now simplified to:

```
CONSTRAINT S : p5 <-> p6;
```

However, this introduces two (redundant) 0-1 variables which do not tighten the space. On the contrary, the resulting MIP-model would be worse. Hence, such a translation must be avoided in any case. For the example above, the effect is not dramatic, but if we imagine that the original variables can be indexed, this would introduces many 0-1 variables (together with additional constraints) having a disastrous impact on the solution process!
We see again how important it is to apply the rules in a well defined order. The success of the approach proposed in this paper stands and falls with these considerations. LPL – as it is now – is certainly far from an ideal translator, but it has the merits to avoids *many* of such not so trivial pitfalls. (I struggled long enough to get things right with this respect.)

Imposing the bounds (note that *e* has no bounds)

```
VARIABLE a [1,20]; b [0,100]; e; f[3,20];
```

LPL generates the following linear constraints:

```
-p3 +p1 +p2 >= 0;
+p1 +p2 -p4 >= 0;
-p1 +p3 +p4 >= 0;
+p3 +p4 -p2 >= 0;
+a +18*p1 <= 20;
+b +97*p2 <= 100;
+e +999992*p3 <= 1000000;
+f -4*p4 >= 3;
a >= 1;
a <= 20;
b <= 100;
f >= 3;
f <= 20;
```

The alternative would generate the following constraints:

```
p5 = p6;
p1+p2 ≥ -p5;
p3+p4 ≥ -p6;
-p3 +p1 +p2 >= 0;
+p1 +p2 -p4 >= 0;
-p1 +p3 +p4 >= 0;
+p3 +p4 -p2 >= 0;
```

```
+a +18*p1 <= 20;
+b +97*p2 <= 100;
+e +999992*p3 <= 1000000;
+f -4*p4 >= 3;
a >= 1;
a <= 20;
b <= 100;
f >= 3;
f <= 20;
```

Since *p5* and *p6* are equal, one of them could be eliminated. This will normally take place in a preprocessing step of the solver. Nevertheless, this formulation is worse.

McKinnon & Williams [1990] propose another translation procedure which generate the following linear constraints from this example:

```
a+(La-2)*d1 >= La;
b+(Lb-3)*d1 >= Lb;
e+(Ue-8)*d2 <= Ue;
f-(Lf-7)*(d1+d2) >= 7;
e+(Le-8)*d4 >= Le;
f+(Uf-7)*d4 <= Uf;
a+(Ua-2)*d5 <= Ua;
b-(Ub-3)*(d4+d5) <= 3;
```

where *La* (*Ua*), *Lb* (*Ub*), *Le* (*Ue*), and *Lf* (*Uf*) are the lower (upper) bounds of the four variables *a*, *b*, *e*, and *f*. *d1–d5* are newly introduced 0-1 variables. It is interesting to see that McKinnon & William's procedure produces the same number of linear constraints, but uses a additional 0-1 variable. I do not know which of the two formulation is tighter, but one can note that the first four constraints in the LPL translation generate a convex hull in the (p1,p2,p3,p4)-space. Therefore, there cannot be a tighter formulation in this subspace.

Example 3: [McKinnon & Williams 1989]

Another interesting constraint is als from [McKinnon & Williams 1989]. Consider the following expression:

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

To formulate this expression in LPL, we introduce a predicate d_i with the meaning:

d_i : "product i is made"

This predicate is linked with the rest of the model in the usual way, that means

d_i is true if $L_i \leq x_i \leq U_i$, where x_i is the (unknown) quantity of product to produce and L_i and U_i are their lower, respectively upper bounds. Hence, we would introduce the predicate into LPL as

```
VARIABLE d{i} BINARY : L <= x <= U;
```

(Note that "*L <= x <= U*" is interpreted by LPL as "*L <= x AND x <= U*". Since such an expression has already been considered in previous examples, we remove it and use only the predicate variable.)

The logical statement can now be reformulated using the predicate d_i as follows:

"At least 3 of d_i $\{i=1..5\}$ are true or at most 3 of d_i $\{i=3..6,8,9\}$ are true and not none of d_i $\{i=5..7\}$ is true implies that at least 2 of d_i $\{i=7..9\}$ are true."

It is easy to code this formulation using LPL syntax. It is:

```
SET i     = /1:9/;          (* list of all products *)
   i1{i} = /1:5/;          (* different subsets *)
   i2{i} = /3:6,8,9/;
   i3{i} = /5:7/;
   i4{i} = /7:9/;

VARIABLE d{i} BINARY;

CONSTRAINT Cond : (ATLEAST(3){i1} d[i1] OR ATMOST(3){i2} d[i2]) AND ~(NOR{i3} d[i3])
            -> ATLEAST(2) {i4} d[i4];
```

First the NOR- and ->-operators are eliminated, giving:

```
CONSTRAINT Cond : ~((ATLEAST(3){i1} d[i1] OR ATMOST(3){i2} d[i2]) AND
                  ~(ATMOST(0){i3} d[i3])) OR ATLEAST(2) {i4} d[i4];
```

Next the ~operators are pushed inwards, giving:

```
CONSTRAINT Cond : (ATMOST(2){i1} d[i1] AND ATLEAST(4){i2} d[i2]) OR
                 ATLEAST(3){i3} d[i3] OR ATLEAST(2) {i4} d[i4];
```

The expression is decomposed by introducing three 0-1 variables as follows:

```
VARIABLE p1 BINARY : ATMOST(2){i1} d[i1] AND ATLEAST(4){i2} d[i2];
VARIABLE p2 BINARY : ATMOST(0){i3} d[i3];
VARIABLE p3 BINARY : ATLEAST(2) {i4} d[i4];
CONSTRAINT Cond : p1 OR p2 OR p3;
```

Next all logical operators are eliminated, giving the following constraints:

```
VARIABLE p1 BINARY : SUM{i1} d[i1] <= 2  AND  SUM(4){i2} d[i2] >= 4;
VARIABLE p2 BINARY : AND{i3} (d[i3] <= 0);
VARIABLE p3 BINARY : SUM{i4} d[i4] >= 2;
CONSTRAINT Cond : p1 + p2 + p3 >= 1;
```

The resulting linear constraints generated by LPL (by applying the rules 51,

70, and 71) are:

```
+p1 +p2 +p3 >=1;
+d1 +d2 +d3 +d4 +d5 +3*p1 <= 5;
+d3 +d4 +d5 +d6 +d8 +d9 -4*p1 >= 0;
+d5 +p2 <= 1;
+d6 +p2 <= 1;
+d7 +p2 <= 1;
+d7 +d8 +d9 -2*p3 >= 0;
```

Several points are noteworthy about this example. First of all, it must be emphasized that this example is far from trivial. While it is relatively easy to do it by hand, a lot of fine-tuned and subtle implementation machination are needed to it automatically. This has to do with the fact that – while the rules itself are quite straightforward – their implementation is quite tricky, especially in the context of indexed terms.

The first notable point is how the AND-operator is treated. Sometimes it is preferable to detach a subtree (subexpression) having a root node labelled with AND, sometimes it is not. LPL takes the decision to detach the subtree if and only if there exists a node within the subtree which is *not* labelled by AND, OR, <->, XOR, ~, or *x* (where *x* is a binary variable). This is the case in our example for the predicate p1.
The alternative would have been to detach the four subexpression with roots labelled ATMOST and ATLEAST. This would have generated the following four predicates:

```
VARIABLE p1 BINARY : ATMOST(2){i1} d[i1]
VARIABLE p1a BINARY: ATLEAST(4){i2} d[i2];
VARIABLE p2 BINARY : ATMOST(0){i3} d[i3];
VARIABLE p3 BINARY : ATLEAST(2) {i4} d[i4];
```

together with the constraint:

```
CONSTRAINT Cond : (p1 AND p1a) OR p2 OR p3;
```

Supposing that the last constraint is translated into a CNF, pushing AND outwards this will generate two constraints as follows

```
CONSTRAINT Cond  : p1 OR p2 OR p3;
CONSTRAINT Conda : p1a OR p2 OR p3;
```

We cannot do any better in this case. It would have been even worse, if the AND were not push outwards as much as possible. This is of concern since pushing the AND outwards as much as possible (and generating a CNF-form) can produce an expression that has an exponential length in the number of propositions which should be avoided, of course. Therefore, as already noted, LPL applies the strategy to push the AND only one level outwards, which

would generate the following constraint in our example:

```
CONSTRAINT Cond : ((p1 OR p2) AND (p1 OR p1a)) OR p3;
```

The next step would then be to detach the subexpression with the root having the AND label, giving

```
VARIABLE p4 BINARY : (p1 OR p2) AND (p1 OR p1a)
CONSTRAINT Cond : p4 OR p3;
```

We see that the strategy on how the AND is handled is critical in the transformation procedure.

A second notable point is how the expression

```
ATMOST(0){i3} d[i3]
```

attached to p2 is handled. It is not translated into the summation

```
SUM{i3} d[i3] <= 0
```

although this is also correct. But there is another tighter form. If the sum of all 0-1 variables are zero then each single variable most be zero. Therefore, LPL generates the constraint:

```
AND{i3} (d[i3] <= 0)
```

The procedure described by McKinnon & Williams [1990] generates the following linear constraints:

```
c1: SUM{i1} d[i1]+3*p1<=5;
c2: SUM{i2} d[i2]-4*p1>=0;
c3{i3}: d[i3]+p2<=1;
c4: SUM{i4} d[i4]-2*p3>=0;
cs: p1+p2+p3>=1;
```

which are exactly the same as LPL produces also.
(The procedure described by McKinnon & Williams [1990] also produces the smaller (but not tighter) formulation:

```
c1 : SUM{i1} d[i1]+3*p1<=5;
c2 : SUM{i2} d[i2]-4*p1>=0;
c3{i3}: d[i3]+p2<=1;
c4 : SUM{i4} d[i4]+2*(p1+p2)>=2;
```

This might be important if many of the cs-type constraints are involved in the formulation. LPL does not take this step presently.)

Example 4: an energy model [Hadjiconstantinou al. 1992]:

The following example is a complete energy import model with mathematical

balancing and capacity constraints coupled with some logical conditions. It was presented in [Hadjiconstantinou al. 1992] and [Mitra/Lucas 1994].
Coal, gas and nuclear fuel can be imported in order to satisfy the energy demands of a country. Three grades of gas and coal (low, medium, high) and one grade of nuclear fuel may be imported. The costs are known for every sort of energy. Furthermore, there are upper and lower limits in the imported quantities from a country. What quantities should be imported if the import costs have to be minimized? In addition, three further conditions must be fulfilled:

1 A supply condition: "Each country can supply either up to three (non-nuclear) low or medium grade fuels or nuclear fuel and one high grade fuel."
2 Environmental constraint: "Environmental regulations require that nuclear fuel can be used only if medium and low grades of gas and coal are excluded."
3 Energy mixing condition: "If gas is imported then either the amount of gas energy imported must lie between 40 – 50% and the amount of coal energy must be between 20 – 30% of the total energy imported or the quantity of gas energy must lie between 50 – 60% and coal is not imported."

To formulate this model in LPL, one can first state the data tables and the variables:

c_{ijk} is the cost of a (non-nuclear) fuel j of grade i from country k.
l_{ijk} and u_{ijk} are the lower and upper imported quantities
x_{ijk} is the unknown quantity of non-nuclear fuel imported
nc_k is the cost of the nuclear fuel imported from country k
nl_k and nu_k are their lower and upper bound
y_k is the unknown quantity of nuclear power to import

To formulate the three logical conditions, six predicates with the following meaning are introduced:

P_{ijk} : "non-nuclear fuel j with grade i is imported from country k"
N_k : "nuclear fuel is imported from country k"
$Qlow_j$: "non-nuclear fuel j imported is at least $MinR_j$% of the total imports"
$Qhigh_j$: "non-nuclear fuel j imported is at most $MaxR_j$% of the total imports"
Rlow : "gas imports is at least MinA% of the total imports"

Rhigh : “gas imports is at most MaxA% of the total imports”
where $MinR_j$ = {20,40}, $MaxR_j$ = {30,50},
MinA = 50, and MaxA=60

The three logical requirements can now be formulated using the predicates as following:

1. For each country k the following condition holds: “Either at most 3 out of all P_{ijk} (with i≠high) are true or N_k is true and exactly one out of $P_{high,j,k}$ is true.”
2. “If at least one of N_k is true then none of P_{ijk} (with i≠high) is true.”
3. “If any of $P_{i,gas,k}$ is true then either ($Qlow_j$ is true and $Qhigh_j$ is true) or (*Rlow* is true and *Rhigh* is true and none of $P_{i,coal,k}$ is true).”

This intermediate formulation in a natural language, using the predicate symbols, is the most important step in formalizing the conditions correctly. The coding in a formal language such as LPL is now straightforward. The whole model written in LPL follows:

```
(* An energy import model *)
SET
 i = /low med high/;            (* different quality grades *)
 j = /gas, coal/;               (* non nuclear energy sources *)
 k = /GB FR IC/;                (* countries exporting energy *)
PARAMETER
 c{i,j,k}; l{i,j,k}; u{i,j,k};  (* non-nuclear energy cost, lower and upper bound *)
 nc{k}; nl{k}; nu{k};           (* nuclear energy cost, lower and upper bound *)
 e;                             (* desired energy amount to import *)
 MinR{j}; MaxR{j};  (* minimum and maximum energy take if coal and gas are imported *)
 MinA; MaxA;        (* minimum and maximum gas take if gas energy only is imported *)

VARIABLE
 x{i,j,k} [0,u];          (* quantity of non-nuclear energy imported *)
 y{k} [0,nu];             (* quantity of nuclear energy imported *)

 P{i,j,k} BINARY : l <= x <= u;
 N{k} BINARY : nl <= y <= nu;
 Q{j} BINARY : e*MinR <= SUM{i,k} x <= e*MaxR;
 R BINARY : e*MinA <= SUM{i,k} x[i,'gas',k] <= e*MaxA;

CONSTRAINT
 Cost: SUM{i,j,k} c*x + SUM{k} nc*y;
 ImportReq: SUM{i,j,k} x + SUM{k} y = e;    (* import requirement *)
 SuplCond{k} : ATMOST(3){i,j|i<>'high'} P XOR (N AND XOR{j} P['high',j,k]);
 Environ : OR{k} N  ->  NOR{i,j,k|i<>'high'} P;
 AltMix : OR{i,k} P[i,'gas',k]  ->  AND{j}Q XOR R AND NOR{i,k} P[i,'coal',k];
MINIMIZE Cost;
END
```

There is no new element in this model that has not yet discussed already. One

can only say that the procedure used in LPL generate fewer 0-1 variables and constraints than the procedure described in Mitra and Lucas [1994] does

.

Example 5: The fancy dress party problem [Suhl]:

A problem that mix mathematical and logical (Boolean) knowledge.

The problem is the following: Mr. Greenfan wants to give a dress party where the male guests must wear green dresses. The following rules are given:

1 If someone wears a green tie he has to wear a green shirt.
2 A guest may only wear green socks and a green shirt if he wears a green tie or a green hat.
3 A guest wearing a green shirt or a green hat or who does not wear green socks must wear a green tie.
4 A guest who is not dressed according to rules 1–3 must pay a DM 11 entrance fee.

Mr Simson wants to participate but owns only a green shirt. He could buy a green tie for DM 10, a green hat (used) for DM 2 and green socks for DM 12. What is the cheapest solution for Mr Simson to participate?

One can formulate this problem using Boolean variables and constraints as follows: Let us introduce the propositions with the meaning:

k “Mr Simson wears a green tie”
b “Mr Simson wears a green hat”
h “Mr Simson wears a green shirt”
s “Mr Simson wears a green socks”
n “Mr Simson is not costumed according the three rules 1–3”

The four rules can be formulated using Boolean expressions as follows:

$$(k \rightarrow \eta) \vee \nu$$

$$((\sigma \wedge \eta) \rightarrow (\kappa \vee \beta)) \vee \nu \qquad (1)$$

$$((\eta \vee \beta \vee \leftarrow\sigma) \rightarrow \kappa) \vee \nu$$

The four rules 1–4 are true if and only if the three Boolean expressions in (1) are true simultaneously.

Another way to formulated would be to connect the first three rules by an AND operator and to add the fourth rule by an OR operator. To resulting expression would be

$$((k \rightarrow \eta) \wedge ((\sigma \wedge \eta) \rightarrow (\kappa \vee \beta)) \wedge ((\eta \vee \beta \vee \leftarrow\sigma) \rightarrow \kappa)) \vee \nu \qquad (2)$$

It is easy to see that both (1) and (2) have the same feasible space.

The minimizing function of our model is a mathematical function at which Mr Simson must minimize his costs (where we interpret the Boolean propositions as “1” if the it is “TRUE” and as “0” otherwise:
Cost: $10k + 2\beta + 12\sigma + 11\nu$

In LPL two solutions exists to formulate the problem:

1. Translate the Boolean constraints into 0–1 constraints (by hand) to get a complete mathematical IP problem. Then to apply an IP-solver,
2. Use a modeling language LPL to formulate the problem in partially mathematical and partially logical form. Given the formulation above this straightforward.

The first solution is simple too because the corresponding Boolean constraints can easily be translated into 0–1 constraints by hand.
Step 1: eliminate the implication: the resulting constraints are

$\neg k \vee h \vee n$

$\neg s \vee \neg h \vee k \vee b \vee n$

$(\neg h \wedge \neg b \wedge s) \vee k \vee n$

Step 2: apply some straightforward transformation rules to get 0–1 constraints: a resulting model would be:

$(1-\kappa) + \eta + \nu \geq 1$

$(1-\sigma) + (1-\eta) + \kappa + \beta + \nu \geq 1$

$((1-\eta) + (1-\beta) + \sigma) + 3\kappa + 3\nu \geq 3$

The complete 0–1 model would now be:

Minimize *Cost*: $10k + 2\beta + 12\sigma + 11\nu$

Subject to:

$$-k + h + n \geq 0$$
$$-s - h + k + b + n \geq -1$$
$$-h - b + s + 3k + 3n \geq 1$$

The translation by hand is easy for small problem with simple Boolean constraints like the rules in our fancy party problem. But it can quickly become very difficult. Experiences have shown that the translation by hand is a substantial source of error. If the problem becomes large, the translation by hand becomes impracticable.

Fortunately, LPL let us formulate a mixed model as follows:

```
MODEL Fancy_Party_Problem;
VARIABLE k BINARY; b BINARY; h BINARY; s BINARY; n BINARY;
CONSTRAINT
       Rule1: (k -> H) OR n;
       Rule2: ((s AND h) -> (k OR b)) OR n;
       Rule3: ((h OR b OR ~s) -> k) OR n;
```

```
MINIMIZE
       Cost: 10*k+2*b+12*s+11*n;
END
```

Of course, the first formulation is also possible in LPL:

```
MODEL Fancy_Party_Problem;
VARIABLE k BINARY; b BINARY; h BINARY; s BINARY; n BINARY;
CONSTRAINT
       Rule1: -k+h+n >= 0;
       Rule2: -s-h+k+b+n >= -1;
       Rule3: -h-b+s+3*k+3*n >= 1;
MINIMIZE
       Cost: 10*k+2*b+12*s+11*n;
END
```

An extended problem

The fancy dress problem as stated above is easy to handle, since there are only 5 variables and 3 constraints. Suppose now that there are 50 persons that may participate at our fancy dress party and everybody must be dressed up according the rules 1–3. If a guest is not dress according to these rules he must pay an entrance fee that is proportional his monthly income (say 0.1%). What is now the cheapest overall cost (we suppose that the 50 guests are members of a society which pays the cost for them)?

In LPL the formulation of the whole problem is very easy:

```
MODEL Fancy_Party_Problem;
SET i=/1:50/;      (* fifty guests *)
PARAMETER Income{i};    (* there income (here 50 numbers follow) *)
VARIABLE k{i}BINARY; b{i}BINARY; h{i}BINARY; s{i}BINARY; n{i} BINARY;
CONSTRAINT
Rules{i}: (k -> H) AND ((s AND h) -> (k OR b)) AND (h OR b OR ~s) -> k) OR n;
MINIMIZE
Cost: SUM{i} (10*k+2*b+12*s+Income*0.001*n);
END
```

The formulation does only slightly change when there are 50 instead of a single guest! Indexing is a major benefit also in logical programming. It comes for free.

5.7 FIXED CHARGE PROBLEMS

Fixed charge problems arise if the value of some unknown depend on a condition which is not known to be true or false. A typical case is the *plant location problem*: The problem is to decide where to build plants in order to minimize overall transportation costs to the customers and fixed plant building costs. The transportation costs depend generally on the distances and the amount of goods shipped between two places or some other variable quantities. The plant building and certain maintenance costs are fixed costs. They occur only if the plant was built at a particular location. If no goods

were shipped from a plant, one would not build it. Problems like this often occur in situations where one has to decide whether to invest a large amount to build a factory or a warehouse or not. These problems, called fixed charge problems, can be formulated using MIP techniques. Fixed charge models are used in a variety of design problems such as water supply systems, heating systems, or route networks. Three examples are chosen to show the modeling process.

Example 1: The Capacitated Facility Location Problem

In the simplest case, a number of plants at locations i={1..m} each with a maximum capacity M_i are planned to produce a single commodity for a number of customer j={1..n} each with a demand of d_j units. The shipping cost from a (planned) plant i to the customer j is g_{ij} and the unknown amount to ship is z_{ij}. Building the plant at location i gives rise to a large (known) fixed cost f_i. The problem is to determine the locations where the new plants are to be built with minimal overall costs. The traditional way to formulate the problem is to introduce a 0–1 variable x_i for every potential location i to differentiate the two states: "plant will be built at location i" and "plant will not be built at location i". x_i is forced to be one if plant at location i is built, otherwise x_i is zero. Then the minimizing cost function can be formulated as

```
MINIMIZE costs: SUM{i,j} g[i,j]*z[i,j] + SUM{i} f[i]*x[i];
```

The fixed cost f_i will not be added if the plant is not built (x_i=0), otherwise the fixed costs are added. To make the link of the x_i variables with the rest of the model the condition that x_i is 1 whenever anything is shipped from the potential plant at location i must be imposed. The condition that "anything is shipped from plant i" can be expressed by the term

$$\sum_{j=1}^{\nu} z_{ij} > 0 \qquad (1)$$

The constraint "if (1) is true for a particular i then x_i = 1" can now be formulated as:

$$\sum_{j=1}^{\nu} z_{ij} > 0 \rightarrow \xi_{\iota} = 1 \qquad (2)$$

Traditionally, the constraint (2) is formulated explicitly by the linear constraint (3) as follows:

$$\sum_{j=1}^{\nu} z_{ij} - M_{\iota}\xi_{\iota} \leq 0 \qquad (3)$$

It is easy to verify it: If $x_i = 0$ then constraint (3) implies $\sum_{j=1}^{\nu} z_{ij} \leq 0$, which expresses exactly the same as (2) since $x_i = 0 \rightarrow \sum_{\varphi=1}^{\nu} \zeta_{\iota\varphi} = 0$ is the contraposition of (2).

The whole problem can now be formulated in LPL as follows:

```
MODEL CFL "The Capacitated Facility Location Problem";
SET
 i      "potential plant locations";
 j      "customers";
PARAMETER
 g{i,j} "shipping cost from a plant i to a customer j";
 f{i}   "fixed cost if plant i is built";
 d{j}   "demand of customer j";
 M{i}   "maximum (planned) capacity at plant i";

VARIABLE
 z{i,j} [0,30]  "amount shipped from plant i to customer j";
 x{i} BINARY    "indicates whether the plant i is built";
CONSTRAINT
 Capa{i}: SUM{j} z[i,j] <= M[i]*x[i];
 Demand{j}: SUM{i} z = d;
MINIMIZE costs: SUM{i,j} g[i,j]*z[i,j] + SUM{i} f[i]*x[i];
END
```

In this formulation no special logical construct has been used. In LPL, however we can formulate the constraint directly. LPL will generate automatically the constraint (3). Again we attach an expression to the predicate variable d. The whole formulation now is:

```
MODEL CFL "The Capacitated Facility Location Problem";
SET
 i      "potential plant locations";
 j      "customers";
PARAMETER
 g{i,j} "shipping cost from a plant i to a customer j";
 f{i}   "fixed cost if plant i is built";
 d{j}   "demand of customer j";
 M{i}   "maximum (planned) capacity at plant i";

VARIABLE
 z{i,j} [0,30]  "amount shipped from plant i to customer j";
 x{i} BINARY ~:= SUM{j} z<=0 ;
CONSTRAINT
 Demand{j}: SUM{i} z = d;
MINIMIZE costs: SUM{i,j} g[i,j]*z[i,j] + SUM{i} f[i]*x[i];
WRITE costs; z;
END
```

The explicit constraint *Capa* was replaced by a predicate declaration. Note that the ~ operator after BINARY is important. It imposes the constraint $x_i = 0 \rightarrow \sum_{\varphi=1}^{\nu} \zeta_{\iota\varphi} = 0$ instead of the constraint $x_i > 0 \rightarrow \sum_{\varphi=1}^{\nu} \zeta_{\iota\varphi} = 0$. That is a subtle difference. The former condition imposes the implication "If any positive amount is shipped from a factory then the factory exists (therefore it has been built)". The later condition, on the other hand, says "if the factory does not exist then nothing can be shipped from it". The difference seems small at first sight. However, the second condition does not forbid to build a factory from which nothing is shipped, while the first does. *This is exactly the situation we want to avoid: to build a factory that is not used!*

The next formulation in LPL goes even an step further: We can avoid the declare any binary variable altogether. In LPL it is possible to multiply a Boolean expression by a constant expression within a constraint. As an example, the constraint

```
CONSTRAINT C :  a*x + b*(y>0) = 0;
```

is perfectly legal. Supposing that a and b are parameters and x as well as y are (continuous) variables. Then the constraint says that:

```
a*x + b = 0   if y>0
a*x = 0       otherwise (y=0, since lower bounds are zero)
```

Using this option, the capacitated facility location problem can be formulated as follows:

```
MODEL CFL "The Capacitated Facility Location Problem";
SET
 i      "potential plant locations";
 j      "customers";
PARAMETER
 g{i,j} "shipping cost from a plant i to a customer j";
 f{i}   "fixed cost if plant i is built";
 d{j}   "demand of customer j";
 M{i}   "maximum (planned) capacity at plant i";

VARIABLE
 z{i,j} [0,30]  "amount shipped from plant i to customer j";
CONSTRAINT
 Demand{j}: SUM{i} z = d;
MINIMIZE costs: SUM{i,j} g[i,j]*z[i,j] + SUM{i} f[i]*(SUM{j}z>0);
WRITE costs; z;
END
```

Is there a more concise way to formulate this problem? I doubt. It should be noted that *in all three formulations LPL generates exactly the same model*. However, the last is clearly to be preferred, since the modeler does not need to be concerned about additional binary variables at all.

Example 2: The Uncapacitated Facility Location Problem

As in the first example, suppose there are i={1..m} facility locations, j={1..n} customers, variable costs (g_{ij}) and fixed (f_i) costs. z_{ij} is now the fraction of the demand of client j that is satisfied from the facility at location i and the upper bound M_i is normalized at 1. The constraint that each client must be satisfied is

```
D{j}: SUM{i} z[i,j] = 1;
```

The condition that a client j cannot be served from i unless a facility is placed at location i can be modeled by introducing a 0–1 variable x_i and by imposing the linear constraints

```
C{i,j}: z[i,j] – x[i] <= 0;
```

(Note, that these (i*j) constraints may be formulated by fewer constraints as described in the preceding section of this paper which has a less tighter solution space.) The whole model structure can be formulated in MIP using LPL as:

```
MODEL UCFL "The Uncapacitated Facility Location Problem";
SET
 i      "a number of facility locations producing a commodity";
 j      "customers";
PARAMETER
 g{i,j} "shipping cost from a facility at i to a customer j";
 f{i}   "fixed cost of an operating facility at i";

VARIABLE
 z{i,j}       "amount shipped from a facility at i to customer j";
 x{i} BINARY  "indicator variable";
CONSTRAINT
 Capa{i,j}: z[i,j] - x[i] <= 0;
 Demand{j}: SUM{i} z[i,j] = 1;
MINIMIZE costs: SUM{i,j} g[i,j]*z[i,j] + SUM{i} f[i]*x[i];
END
```

In the same way as in the capacitated facility location problem we can replace the constraint Capa by a predicate declaration as follows:

```
MODEL UCFL "The Uncapacitated Facility Location Problem";
SET
 i      "a number of facility locations producing a commodity";
 j      "customers";
PARAMETER
 g{i,j} "shipping cost from a facility at i to a customer j";
 f{i}   "fixed cost of an operating facility at i";

VARIABLE
 z{i,j} [0,1]   "amount shipped from a facility at i to customer j";
 x{i} BINARY ~:= AND{j}(z<=0) ;
CONSTRAINT
 Demand{j}: SUM{i} z[i,j] = 1;
MINIMIZE costs: SUM{i,j} g[i,j]*z[i,j] + SUM{i} f[i]*x[i];
END
```

Again the third formulation of this problem is:

```
MODEL UCFL "The Uncapacitated Facility Location Problem";
SET
 i      "a number of facility locations producing a commodity";
 j      "customers";
PARAMETER
 g{i,j} "shipping cost from a facility at i to a customer j";
 f{i}   "fixed cost of an operating facility at i";

VARIABLE
 z{i,j} [0,1]   "amount shipped from a facility at i to customer j";
CONSTRAINT
 Demand{j}: SUM{i} z[i,j] = 1;
MINIMIZE costs: SUM{i,j} g[i,j]*z[i,j] + SUM{i} f*(OR{j}(z>0));
WRITE costs; z; x;
END
```

In is important to note that the third formulation uses the negation of the term "AND{j}(z<=0)" which is "OR{j}(z>0)".

Example 3: The fixed charge network flow problem

The network flow problem is well known: Suppose to have a network with a set of nodes i={1...n} and a set of arcs (i,j) which point from a node i to a node j (indicating a direct shipping route from node i to node j). Associated with each node i, there is a number b_i. Depending whether b_i is positive, zero, or negative, the node i is called a demand node, a transit node, or a supply node. Each arc has a flow capacity u_{ij}. and a flow cost h_{ij}. Let y_{ij} be the unknown flow on arc (i,j). A flow is feasible if and only if

```
C{i,j}: y[i,j] <= u[i,j];    "flow capacity"
F{i}: SUM{j in arcs[j,i]} y[j,i] - SUM{j in arcs[i,j]} y[i,j] = b[i];
```

The objective is now to minimize the costs:

```
MINIMIZE Costs: SUM{arcs[i,j]} h[i,j]*y[i,j];
```

The whole *network flow problem* can then be formulated as

```
MODEL NetFlow "the network flow problem";

SET i ALIAS j      "nodes";
   arcs{i,j}      "arcs";
PARAMETER
 b{i}             "balance amount at node i";
 h{arcs}          "shipping costs";
 u{arcs}          "capacity";
 c{arcs}          "fixed costs";

VARIABLE
 y{arcs} [0,u]    "amount to be shipped";
CONSTRAINT
 F{i}:  SUM{j|arcs[i,j]} y[i,j] - SUM{j|arcs[j,i]} y[j,i] = b[i];

MINIMIZE costs: SUM{arcs} h*y;
END
```

The *fixed charge network flow problem* is obtained by imposing a fixed cost c_{ij} if there is a positive flow y_{ij} on arc (i,j). Hence, one has to introduce a binary variable x_{ij} to indicate whether arc (i,j) is used or not.
The whole model can then defined as follows:

```
MODEL FixChargeFN "the fixed charge network flow problem";

SET i ALIAS j      "nodes";
   arcs{i,j}      "arcs";
PARAMETER
 b{i}; h{arcs};  u{arcs};  c{arcs}

VARIABLE
 y{arcs} [0,u]    "amount to be shipped";
CONSTRAINT
 F{i}:  SUM{j|arcs[i,j]} y[i,j] - SUM{j|arcs[j,i]} y[j,i] = b[i];

MINIMIZE costs: SUM{arcs} (h*y + c*(y>0));
END
```

Example 4: Multi-commodity distribution and facility location

This model is a generalization of the facility location problem with several commodities, and shipment from a plant to a customer occurs through a distribution centre [Geoffrion & Graves 1974]. The problem can be formulated as

```
MODEL MDFL "Multicommodity distribution and facility location";
SET
 i          "index for commodities";
 j          "index for plants";
 k          "index for possible distribution center sites";
 l          "index for customer demand zones";
PARAMETER
 S{i,j}     "production capacity for commodity i at plant j";
 D{i,l}     "demand for commodity i by customer l";
 L{k};U{k}  "minimum and maximum allowable total annual flow through
              the distribution center at site k";
 f{k}       "fixed annual operating cost of the distribution center";
 v{k}       "per unit cost of flow through the distribution center";
 c{i,j,k,l} "per unit cost of producing and shipping commodity i
             from plant j to customer l via distribution center k";
VARIABLE
 x{i,j,k,l}     "the amount of commodity i shipped from plant j to
                 customer l via distribution center k";
 y{k,l} BINARY  "a 0-1 variable, where y(k,l)=1 if distribution center k
                 serves customer l, and y(k,l)=0 otherwise";
 z{k} BINARY    "a 0-1 variable, where z(k)=1 if distribution center k
                 is opened , and 0 otherwise";
CONSTRAINT
 Capa{i,j}: SUM{k,l} x <= S;
 Dem{i,k,l}: SUM{j} x = D*y;
 Serve{l}: SUM{k} y = 1;
 Flow{k}: L*z <= SUM{i,l} D*y <= U*z;
MINIMIZE Cost: SUM{i,j,k,l} c*x + SUM{k} f*z + SUM{k,i,l} v*D*y;
END
```

This model was used by a major food firm with 17 commodities, 14 plants, 45 possible distribution centre sites, and 121 customer zones. A full discussion of this model, as well as the application of Benders decomposition algorithm to solve it, are given in the paper cited above.
Again using LPL's logical extension the model can be formulated more concisely in the same way (see model UFL.lpl).

5.8 Modeling discontinuous functions

Sometimes non-linear, discontinuous functions, such as ABS(x), MAX(x,y), MIN(x,y), IF(x,y,z) and others, must be used in the model. There is no problem, if the expressions x, y, and z do not contain any model variables. If, on the other hand, these expressions contain variables, the model becomes non-linear and the linear solver can no longer be applied. In some cases, such functions can be replaced by well known constructs, eventually by introducing new variables or constraints. A well known example comes from game theory: In a finite two-person zero-sum one-move game, the optimal strategies x_i of the

first player can be found by solving the following optimizing problem where a_{ij} is the known pay-off matrix

$$\text{MAXIMIZE} \quad \min_{j=1}^{\nu} \sum_{\iota=1}^{\mu} \alpha_{\iota\varphi}\xi_{\iota}$$
$$\text{συβφεχτ τ} \sum_{\iota=1}^{\mu} \xi_{\iota} = 1 \qquad (1)$$

Unfortunately, this model formulation (1) contains a non-linear **min-function** with variables as parameters.

There exists a reformulating technique, however, to put the model in a equivalent form (2) which is linear and has the same solution space, where λ is introduced as a new model variable (not necessary integer).

$$\text{MAXIMIZE} \quad \lambda$$
$$\text{συβφεχτ τ}\vartheta \le \sum_{\iota=1}^{\mu} \alpha_{\iota\varphi}\xi_{\iota} \quad \text{ωιτη} \quad \varphi = \{1\text{K } \nu\} \qquad (2)$$
$$\sum_{\iota=1}^{\mu} \xi_{\iota} = 1$$

In LPL, both formulations are possible: the first formulation is:

```
CONSTRAINT R: SUM{i} x[i] = 1;
MAXIMIZE MinLoss: MIN{j} (SUM{i} a[i,j]*x[i];                    (3)
```

and the second formulation in LPL is

```
VARIABLE lambda;
CONSTRAINT R: SUM{i} x[i] = 1;
CONSTRAINT Q{j}: lambda <= SUM{i} a[i,j]*x[i]                    (4)
MAXIMIZE MinLoss: lambda ;
```

Formulation (3) is more natural and shorter, and – what is more important – the LPL compiler takes care to transform it to the formulation (4) without any intervention of the modeler.

LPL implements the following rule for the MIN operator:

Any expression $\ldots + \min_{i=1}^{n}(x_i) + \text{K}$ containing the sub-expressions x_i within a model constraint will be replaced by a newly created variable...+z+... . Furthermore, n constraints $z \le \xi_{\iota}$ ωηερε $\iota = \{1\text{K } \nu\}$ are added to the model.

The transformation, therefore, is

$\ldots + \min_{i=1}^{n}(x_i) + \text{K}$ gives $\ldots + z + \text{K}$ adding the constraints $z \le x_i$ where $i = \{1\text{K } n\}$. Of course, the requirement that z must be exactly at the minimum is not automatically fulfilled, there z is added to the maximizing (or –z to the minimizing) function.

A similar rule can be given for the **max-function**:

Any expression $\ldots + \max_{i=1}^{n}(x_i) + \text{K}$ containing the sub-expressions x_i within a model constraint will be replaced by a newly created variable...+z+... . Furthermore, n constraints $z \ge \xi_{\iota}$ ωηερε $\iota = \{1\text{K } \nu\}$ are added to the model.

The transformation, therefore, is

$$\ldots+\max_{i=1}^{n}(x_i)+\ldots \quad \text{gives} \quad \ldots+z+\ldots \quad \text{adding the constraints } z \ge x_i \text{ where } i=\{1\ldots n\}$$

It should be noted that LPL does the translation automatically, even in an unbounded case such as $\ldots+\max_{i=1}^{n}(x_i)+\ldots \ \ge \ \ldots$. The modeler should use these functions very carefully.

The **absolute value function** $|x|$ can be replaced by $\max(x,-x)$ and the translation rules of the MAX operators might be applied. A sum of absolute values is more involving. Different translations are possible. The first is stated by

$$\sum_{i=1}^{n}|x_i| \le 1 \qquad \text{gives} \qquad \sum_{i\in S}x_i - \sum_{j\in\{1\ldots n\}-S}x_j \le 1 \quad \text{for every } S\subset\{1\ldots n\}. \qquad (5)$$

A disadvantage of this translation rule (5) is that the number of added constraints grows exponentially with n. For n=2 this gives

$$\begin{aligned} x_1 + x_2 &\le 1 \\ -x_1 + x_2 &\le 1 \\ x_1 - x_2 &\le 1 \\ -x_1 - x_2 &\le 1 \end{aligned} \qquad . \qquad (5a)$$

An alternative formulation of the absolute sum function is

$$\begin{aligned} x_i &\le u_i \quad \text{with } i=\{1\ldots n\} \\ -x_i &\le u_i \quad \text{with } i=\{1\ldots n\} \\ \sum_{i=1}^{n} u_i &\le 1 \end{aligned} \qquad (6)$$

still another formulation is

$$\begin{aligned} x_i &= u_i - w_i \quad \text{with } i=\{1\ldots n\} \\ \sum_{i=1}^{n}(u_i + w_i) &\le 1 \end{aligned} \qquad (7)$$

In the first formulation (5) techniques, 2^n constraints are added but no variables; the second formulation (6) adds 2n+1 constraints and n variables; and formulation (7) uses 2n new variables and n+1 new constraints (see Jeroslow 1989, p.8). In LPL, the absolute function is not allowed within model constraints and if needed it must be reformulated manually using one of the mentioned methods.

And generalisation of the fixed charge problem could be formulated using the the **function if(x,y,z)**. This function is fully allowed within constraints and causes no particular problem even if the sub-expressions *y* or *z* contain model variables. However, the sub-expression *x* should not contain any variables. If the expression *x* is true then expression *y* is evaluated, otherwise expression *z* is evaluated.
Example:

```
SET i=/1:10/;
VARIABLE u{i}; v{i};
```

```
MODEL R: SUM{i} if(i<5,u,v) = 0;
```

produces the model constraint:

```
U1+U2+U3+U4+V5+V6+V7+V8+V9+V10 = 0
```

In some models situations arise where the sub-expression *x* may contain variables and in this case the expression if(x,y,z) cannot be evaluated, since *x* has an unknown value. One might even think that it is impossible to obtain some meaningful interpretation of such an expression. But the situation arises frequently in discontinuous cost functions and some other contexts. Suppose that the total cost, which has to be minimized, contain variable and fixed costs. The variable costs depend (linearly) on the (unknown) produced quantity *x* of a product and the fixed costs are zero if nothing is produced (if *x*=0), but they are at least *c* if something is produced (if *x*>0). Figure 13 shows the cost function depending on the quantity *x* produced. This problem is called the *fixed charge problem*. The discontinuous cost function can be defined as

total costs = \B\LC\{(\S(ax+c if x>0,0 if x=0))

A convenient way to formulated this function is to use the discontinuous function *if* in LPL and to model the total costs which contain fixed and variable costs as follows:

```
Total_costs = a*x + if(x,c);
```

Again, this expression is not linear. A reformulating technique exists, however, which produces a model with the same solution space and where the model becomes linear, using integer variables.

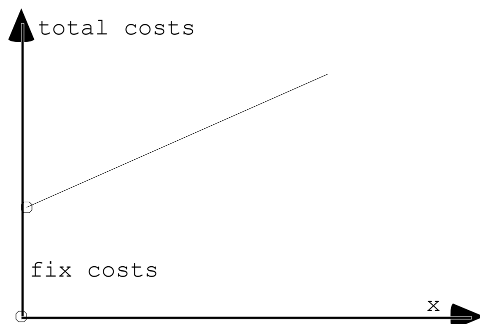


Figure 13: Fixed Charge Costs

The translation rule is as follows:

Add a new 0–1 variable *z* to the model whose values (0 and 1) map the dichotomy between *x*=0 and *x*>0; replace the expression *if(x>0,c)* by *c*z*; and add a new constraint *x<=M*z*, where *M* is an upper bound of *x*. Another

section gives an alternative formulation technique for the fixed charge problem. LPL implements this transformation rule.

6 IS TERM REWRITING USEFUL FOR MODEL TRANSFORMATION?

This section gives some general ideas about (automated) model reformulation. Several applications have been presented in the last section, where the formulation uses logical operators, like *and*, *or* etc. to define the constraints of a problem. The formulations were given partly in a subset of predicate logic and partly in mathematical notations as linear constraints. This is convenient for many problems, but it cannot be processed directly by a solver. All examples have shown that they can be converted to a pure MIP form. Several rules have to be applied to these formulations to eliminate all Boolean operators (except the semicolon, which can be interpreted as the *and* operator).

Of course, the transformation to a pure MIP model is only needed if the modeler wants to solve the model with an MIP solver. There is no reason, otherwise, to carry out the transformation. For purely logical models, for example, a more specialized solver may be applied to the model directly without any transformation to an MIP model. For different solvers, different transformations rules must be implemented. *The modeling language itself should make it possible for the* ***modeler*** *to specify its own transformation rules in order to interface the model with the right solver.* Actually, the transformation rules described above are hardwired within the LPL compiler. There should be a possibility to specify the rules themselves as language statements.

Let's give a hypothetical extension of LPL which allows one to specify such rules: A new Rule statement is added to the modeling language LPL. The Rule statement begins (like any other statement in LPL) with a reserved word (use 'RULE'). The reserved word is followed by a rule head and a rule tail, separated by an assignment operator. The rule head contains a mask of the part to be transformed and the rule tail contains the result of the transformation. Rule head and rule tail must be legal language constructs eventually containing place holder.

```
RULE <rule-head>  ::=  <rule-tail> ;
```

Place holders are identifiers (symbols), such as X or Y. The content of a place holder may be any language construct. Let's give an example

The rule which replaces the -> operator, might be written as

```
RULE (X) -> (Y)  ::=  (~(X) OR (Y)) ;
```

Note that '*x -> y*' is the same as '*~x OR y*' in Boolean algebra. The example is a very simple one and could be implemented using the same construct as the programming language C uses for its macro definition.

By the C pre-compiler instruction

```
#define max(x,y)  ((x) > (y) ? (x) : (y)
```

the C preprocessor replaces every occurrence of 'max(x,y)' within the program by the second part of the definition.
A similar and somewhat more sophisticated preprocessor can be found in the professional CLIPPER database system produced by Nantucket.

These capabilities may be subsumed under the notion of *term rewriting*. Term rewriting is also a fundamental concept in constraint programming languages [Leler 1988] and is an active field of research in the artificial intelligence community [Book 1991]. Leler defines term rewriting as "the application of a set of rewrite-rules to an expression to transform it into another expression". It has drawn much attention as a possible paradigm for automating mathematical theorem proving and symbolic manipulation. For example, the knowledge of symbolic differentiation could be condensed into a set of rules such as

1) $\frac{dC}{dx} = 0$

2) $\frac{\delta \xi \nu}{\delta \xi} = \nu \xi^{\nu - 1}$

3) $\frac{\delta \phi(\xi) + \delta \gamma(\xi)}{\delta \xi} = \frac{\delta \phi(\xi)}{\delta \xi} + \frac{\delta \gamma(\xi)}{\delta \xi}$

4) K

A system that processes these rules can do symbolic differentiation. If more rules are to be integrated into the system, one has only to add the rules to the list.

This paradigm of term rewriting could be used for different model transformation. The rule of transforming a general MIP model into a 0–1-MIP model, for example, might be formulated as

```
RULE
      (X:"X is a general integer variable")  ::=  (SUM{i=.lg2} new[i] )
                                               ADD (VARIABLE new{.lg2(X).up} BINARY ;);
```

'*(X:"X is a general integer variable)"*' is the rule head which means that X must be a general integer variable.
'*(SUM{i=.lg2} new[i]) ADD (VARIABLE new{.lg2(X).up} BINARY ;)*' is the rule tail which says that every occurrence of a general integer variable X must be replaced by the expression '*SUM{i=.lg2} new[i]*' and at the first occurrence a new variable definition *new* must be added to the model.
The rule to eliminate the min- or max-operator can be formulated as

```
RULE min{<i>} (<X>)  ::=  (<%1>)
                         ADD (VARIABLE <%1>; ADD CONSTRAINT <%2>{<i>} : %1 <= <X>);
RULE max{<i>} (<X>)  ::=  (<%1>)
                         ADD (VARIABLE <%1>; ADD CONSTRAINT <%2>{<i>} : %1 >= <X>);
```

But the general MIN-rule is more complex: suppose we have the following constraint:

```
CONSTRAINT R{i}: SUM{j} (MAX{k} x[i,j,k];
```

This must be translated into the following two constraints and a new variable z

```
VARIABLE z{i,j};
CONSTRAINT R{i}: SUM{j} z[i,j];
CONSTRAINT R1{i,j,k}: z[i,j] >= x[i,j,k];
```

It is not clear how to formulated the RULE to get the result.

New operators may also be introduced using the RULE statement. Suppose that the XOR{} operator did not exist within the modeling language; in this case, it may be introduced by a rule

```
RULE
 OPERATOR XOR{<i>} <X>    ::=  (EXACTLY(1){<i>} <X>
```

Up to now, it is not clear how to implement such a rule statement in general, but it would make the model formulation much more flexible.

7 CONCLUSION

This paper has introduced some ideas of a new generation of modeling languages for mathematical and logical models. The possibility to mix logical conditions and mathematical constraints is a powerful and flexible method to formulate complex problem situation. Since a general *solver* for such problems does not exist, it may be beneficial to have a general *formulation language* for all these problems and some mechanism to transform the formulation at hand into an appropriate form for a specified solver. A new version of the LPL modeling language was presented which allows to mix logical and mathematical statements. By default, the LPL compiler translates the logical part of the model into a list of mathematical constraints in order to solve the whole model, using a general MIP solver.

***Acknowledgements**:* This paper was influenced and stimulated most of all by the book of Williams 1990 as well as the paper of McKinnon & Williams 1989.

REFERENCES

AHO A.V., SETHI R., ULLMAN J.D., [1986], Compilers, Principles, Techniques, and Tools, Addison Wesley, Menlo Park.

BARTH P., [1996], Logic-Based 0–1 Constraint Programming, Kluver Academic Publ., Boston.

BOOK R.V., [1991], Rewriting Techniques and Applications, 4th International Conference, RTA–91, Como, Italy, April 1991, Proceedings, Springer, Berlin.

BRADLEY G.H., HAMMER P.L., WOLSEY L., [1974], Coefficient reduction for inequalities in 0–1 variables, Math. Programming 7 (3) December. p.263–282.

BREARLEY A.L., MITRA G., WILLIAMS H.P., [1975], Analysis of Mathematical Programming Problems Prior to Applying the Simplex Algorithm, in: Mathematical Programming 8 (1975), p. 54–83.

BÜNING K., LETTMANN T., [1994], Aussagenlogik: Deduktion und Algorithmen, Teubner Verlag, Stuttgart.

CHANDRU V., HOOKER J.N., [1988], Logical Inference: a Mathematical Programming Perspective, in: Workshop "Mathematics and AI", Vol II, FAW Ulm, 19th–22nd December 1988, p.315–375.

DARBY-DOWMAN K., LUCAS C., MITRA G., YADEGAR J., [1988], Linear, Integer, Separable and Fuzzy Programming Problems: A Unified Approach Towards Reformulation, in: J. Opl Res. Soc., Vol. 39, No. 2, pp. 161–171.

De JONG K.A., SPEARS W.M., [1989], Using Genetic Algorithms to Solve NP–Complete Problems, in: Proceedings of the 3th International Conference on Genetic Algorithms, Schaffer J.D. (ed.), George Mason University, June 4–7.

FOURER R. [1983], Modeling Languages Versus Matrix Generators for Linear Programming., ACM Transactions on Mathematical Software, Vol. 9, No. 2, June 1983, pp. 143–183.

GEOFFRION A., GRAVES G.W., [1974], Multicommodity Distribution System Design by Benders Decomposition, Management Science 20(5) p. 822–844.

GRIZE F., STROHMEIER A., [1983], SARTEX: Manuel de reference du language, Version 2.0, juin, département de mathématiques, Ecole Polytechnique Fédérale de Lausanne, Suisse.

HADJICONSTANTINOU E., LUCAS C., MITRA G., MOODY S., [1992], Tools for Reformulating Logical Forms into Zero–One Mixed Integer Programs

(MIPS), Working Paper (to appear in EJOR).

HAMMER P.L. (IVANESCU), RUDENU S., [1968], Boolean Methods in Operations Research and Related Areas, Springer, New York.

HOOKER J.N., [1993], Logic-Based Methods for Optimization, a Tutorial, GSIA Working Paper #1994-05.

HOOKER J.N., [1988], Generalized Resolution and Cutting Planes, in: Annals of Operations Research 12, p.217–239.

HOOKER J.N., [1988a], A Quantitative Approach to Logical Inference, in: Workshop "Mathematics and AI", Vol II, FAW Ulm, 19th–22nd December 1988, p.289–314. (reprinted in: DSS 4 (1988), p.45–69).

HÜRLIMANN T. [1996], Reference Manual for the LPL Modeling Language, Version 4.0, Institute of Informatics, Working Paper, November 1996, Fribourg.

HÜRLIMANN T. [1996a], LPL: A mathematical programming language, Institute of Informatics, Working Paper, November 1996, Fribourg. (An older version of this paper is in: OR Spektrum, 15:43–55(1993), Springer.)

HÜRLIMANN T. [1991], The 20 LP Models written in LPL from Williams H.P. 1990, Institute for Automation and Operations Research, Working Paper No. 181, January, Fribourg.

JEROSLOW R.G., LOWE J.K., [1984], Modelling with Integer Variables, Mathematical Programming Study, Vol 22, 1984, p.167–184.

JEROSLOW R.G., [1989], Logic-Based Decision Support, Mixed Integer Model Formulation, Annals of Discrete Mathematics Vol 40, North-Holland, Amsterdam.

LELER W., [1988], Constraint Programming Languages, Their Specification and Generation, Addison-Wesley Publ. Comp., New York.

LUCAS C., MITRA G., [1988], Computer-assisted mathematical programming (modeling) system: CAMPS, The Computer Journal, Vol 31(4), pp 364–375.

MARTELLO S., TOTH P., [1990], Knapsack Problems, Algorithms and Computer Implementation, John Wiley & Sons, Chichester.

McKINNON K.I.M., WILLIAMS H.P. [1989], Constructing Integer Programming Models by the Predicate Calculus, Annals of Operations Research, Vol 21, p.227–246.

MEIER G., DÜSING R., [1992], Zur Modellierung logischer Aussagen ergänzend zu Linearen Programmen, Grundlagen und Entwurfsüberlegungen für einen Modellgenerator, OR-Spektrum, (1992) 14:149–160.

MENDELSON E., [1965], Introduction to Mathematical Logic, Leicester University Press.

MITRA G., LUCAS C., MOODY S., [1994], Tools for reformulation logical forms into zero–one mixed integer programs, in: European Journal of Operational Research, Vol. 72,2, pp 262–276, North-Holland.

NEMHAUSER G.L., WOLSEY L.A. [1988], Integer and combinatorial

Optimization, Wiley.

NEMHAUSER G.L., WOLSEY L.A. [1989], Integer Programming, Chap VI in: Handbooks in Operations Research & Management Science, Vol. 1, eds. Nemhauser G.L., Rinnooy Kan A.H.G., Todd M.J., North Holland.

QUINE W.v.O., [1974], Grundzüge der Logik, suhrkamp taschenbuch.

SALKIN H.M., MATHUR K., [1989], Foundations of Integer Programming, North-Holland, New York.

WILLIAMS H.P., [1991], Computational Logic and Integer Programming: Connections between the Methods of Logic, AI and OR, Working Paper, University of Southampton, U.K.

WILLIAMS H.P. [1990], Model building in mathematical programming, Third Edition, Wiley, Chichester.

WILLIAMS H.P. [1987], Linear and integer programming applied to the propositional calculus, Int. J. Syst. Res. Inform. Sci.., 2 (1987), pp.81–100.

WILLIAMS H.P. [1978], The Reformulation of two mixed integer programming problems, Math. Programming 14 (1978) pp.325–331.

WILLIAMS H.P. [1977], Logical problems and integer programming, Bull. Inst. Math. Appl., 13 (1977), pp.18–20.

WILLIAMS H.P. [1974], Experiments in the formulation of integer programming problems, Math. Programming Study 2 (1974) pp.180–197.

YAGER R.R. [1987], A mathematical programming approach to inference with the capability of implementing default rules, in: Gaines B.R., Boose J.H. (eds), Machine Learning and Uncertain Reasoning, Knowledge-Based Systems Vol 3, Academic Press, London, 1990, pp.261–290.