AN EFFICIENT LOGIC-TO-IP TRANSLATION PROCEDURE

T. Hürlimann

Talk at the APMOD98 International Conference, Applied Mathematical Programming and Modeling, Limassol, Cyprus, March 11–13, 1998

March 1998

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 2845 fax: 26 300 9726

This research is supported by the Swiss National Science Foundation and financed by the project no. 1217-45922.95.

An Efficient Logic-to-IP Translation Procedure

Tony Hürlimann

Abstract: In this paper, a procedure is presented which translates logical constraints into mathematical constraints containing 0–1 variables. The procedure has been carefully designed in order to generate compact IP models. It has been fully integrated into the modeling language LPL (Linear Programming Language). Hence, this language can be used to formulate models which contain mathematical *and* logical constraints at the same time. The translation procedure can be used to generate mixed integer models in a complete automatic way, or it can be used as a supporting tool for the modeler in order to translate logical constraints manually.

Keywords: modeling, integer programming, modeling language, logical constraints.

1 INTRODUCTION

It is well known that propositional logical statements can be translated into linear constraints containing 0–1 variables [Williams 1990, 1987, 1977]. A simple general procedure for doing this is, in a first step, to reformulate the Boolean expressions in a *conjunctive normal form*, and in a second step, to replace each clause by a linear constraint. Unfortunately, such a general method can produce, in the worst case, exponential many clauses in the term of the propositions. An alternative is to reformulate the Boolean expressions in a *disjunctive normal form* and then to translate it into linear constraints by introducing additional (continuous) variables. Unfortunately, this second method may introduce exponential many variables in the worst case. (This second method is more interesting if the original logical statement is already a disjunctive set of linear inequalities, for example – and not a purely Boolean expressions.)

Example 1:
The first method is shown using the expression

$$((x \wedge y) \vee z) \veebar w \quad \text{(where } \veebar \text{ is the XOR-operator)} \qquad (1)$$

Replacing $a \veebar b$ by $(a \vee b) \wedge (\bar{a} \vee \bar{b})$ gives

$$((x \wedge y) \vee z \vee w) \wedge (\overline{(x \wedge y) \vee z} \vee \bar{w})$$

Moving the NOT-operator inwards gives

$$((x \wedge y) \vee z \vee w) \wedge (((\bar{x} \vee \bar{y}) \wedge \bar{z}) \vee \bar{w})$$

Moving the OR-operators inwards gives the following conjunctive normal form

$$(x \vee y \vee z) \wedge (x \vee y \vee w) \wedge (\bar{x} \vee \bar{y} \vee \bar{w}) \wedge (\bar{z} \vee \bar{w})$$

Translating the clauses into linear constraints, is now straightforward. The resulting 0–1 IP is (where *x*, *y*, *z*, and *w* are binary 0–1 variables):

$$x + y + z \geq 1,\ x + y + w \geq 1,\ 1 - x + 1 - y + 1 - w \geq 1,\ 1 - z + 1 - w \geq 1. \qquad (2)$$

Example 2:
The second method is shown using a disjunction of linear constraints:

$$A_1 x \geq b_1 \vee A_2 x \geq b_2 \vee \ldots \vee A_n x \geq b_n \qquad (3)$$

By splitting the vector **x** into *n* components ($\mathbf{x}^{(i)}$, $i = \{1,\ldots,n\}$), this disjunction can be formulated as [Williams 1994]

$$\begin{aligned} &\mathbf{A}_i\mathbf{x}^{(i)} \geq \mathbf{b}_i\delta_i \\ &\mathbf{x} = \sum_i \mathbf{x}^{(i)} \\ &\sum_i \delta_i = 1 \\ &\delta_i \in \{0,1\},\ \mathbf{x}^{(i)} \geq 0,\ \ i = \{1,\ldots,n\} \end{aligned} \tag{4}$$

The reformulation (4) has been developed by Jeroslow [1989] based on the work of [Balas 1979] and [Meyer 1981] and others.

The traditional way to model (3) as an MIP problem is to introduce 0–1 variables δ_i , $i = \{1,\ldots,n\}$ together with the constraints

$$\begin{aligned} &\mathbf{A}_i\mathbf{x} - \mathbf{b}_i \geq \mathbf{L}_i(1 - \delta_i) \\ &\sum_i \delta_i \geq 1 \\ &\delta_i \in \{0,1\},\ \ i = \{1,\ldots,n\} \end{aligned} \tag{5}$$

where $\mathbf{L}_i$ is a lower bound for the rows of $\mathbf{A}_i\mathbf{x} - \mathbf{b}_i$.
The advantage of formulation (4) is that it is sharp (a proof using duality can be found in [Williams 1994]), meaning that the LP relaxation $0 \leq \delta_i \leq 1$ generates integer values for all δ_i. A disadvantage is that exponential many variables are needed in the worst case. The disadvantage of (5), however, is that it is not a sharp formulation and many additional cuts are needed in the worst case to make it sharp. A further drawback is that the formulation (5) needs a lower bound for each row.
The two approaches are “dual” to each other: while (4) needed additional variables, (5) needed additional cuts to be solved. Which method is better depends on the model at hand.

In many cases, we are forced to have either a sharp but large model on the one hand, or an not-sharp but small and compact model on the other hand [Nemhauser 1988], [Williams 1978], [Williams 1974]. Therefore, every automatic procedure must make some compromise between these two extremes. Another related difficulty is that such a procedure, if not carefully designed, generates many redundant 0-1 variables and constraints which do not

sharpen the formulation; on the contrary, the model formulation is deteriorated.
A few attempts have been presented in the literature to implement such a procedure [Yeom al. 1996], [McKinnon al. 1989], [Meier al. 1992], [Mitra al. 1994]. Unfortunately, none of them is fully integrated into an existing modeling language. Yeom/Lee [Yeom al. 1996], the most recent reference, only proposes a list of already well-known rules. It is not clear from the paper whether these translation rules have been implemented. Mitra/Lucas/Moody [Mitra al. 1994] proposes a procedure that is part of a modeling system, but the generated IP model frequently generates many redundant 0–1 variables and constraints. The same is true for the procedure presented by Meier/Düsing [Meier al. 1992]. The only efficient procedure which produces a compact IP formulation seems to have been proposed by McKinnon and Williams [McKinnon al. 1989]. Unfortunately, it was implemented in Prolog and is not integrated into a modeling system.
In this paper, a similar but more general translation procedure is proposed which is already an integral part of the LPL modeling language. The translation is entirely symbolic and can be traced step by step, in order to be useful also for manual reformulation.

This paper is organized as follows. Section 2 gives a simple but instructive example which illustrates different translations options. Section 3 defines all logical operators which are eliminated by the procedure. Section 4 presents the translation procedure step by step and is the core of the paper. The last section gives several examples to show the benefits of such a procedure.

2 AN ILLUSTRATIVE EXAMPLE: FLEXIBLE STORAGE

A small example is presented to show how to translate logical constraints into a MIP problem [MEIER G al., 1992, p 150 (see model *b1*)].
Suppose a company produces two liquids *A* and *B* in unknown quantities *x* and *y*. The liquids are stored in containers. The company owns two containers with capacities *a* and *b* The liquids cannot be mixed in the same container. Normally, liquid *A* is stored in the first and liquid *B* in the second container. Hence, the company can produce only the maximum quantity of *a* of liquid *A* and a maximum quantity of *b* of liquid *B*, because of the storage constraint. Occasionally, only one liquid is produced and *both* containers could be used to store it. In this case, the company may produce the liquid at a maximum quantity of $a+b$. Hence, *either* the company produces both liquids at quantities less than or equal to *a* and *b or* it produces only one liquid with quantity less

than or equal to $a+b$

How can we model such a situation? Figure 1 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.

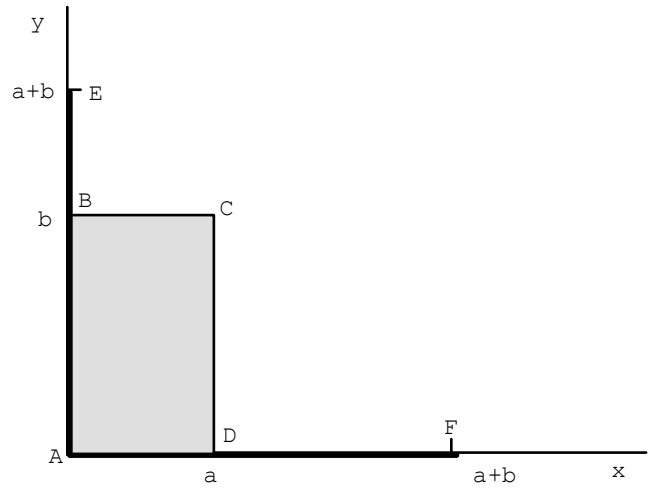

Figure 1: Disjunctive Set

The feasible space is a *disjunctive set*. Therefore, the situation cannot be formulated as a single LP model. However, it can be divide into three convex spaces and each can be formulated as a LP:

LP1: $x \le a\,,\ y \le b$

LP2: $x \le a+b\,,\ y \le 0$

LP3: $x \le 0\,,\ y \le a+b$

The overall non-convex set can be expressed as:

$LP1 \vee LP2 \vee LP3\,.$

Hence, the problem can be formulated as a single constraint containing logical and mathematical operators in the same expression as follows:

$$(x \le a \ \wedge\ y \le b) \ \vee\ (x \le a+b \ \wedge\ y \le 0) \ \vee\ (y \le a+b \ \wedge\ x \le 0) \qquad (6)$$

(Note that the comma in the LP formulation is in fact an AND-operator).

This disjunctive formulation is an instance of (3) and can be translated into a sharp MIP using the reformulation (4) as follows:

$$
\begin{aligned}
& x^{(1)} \le a \cdot \delta_1 , \quad y^{(1)} \le b \cdot \delta_1 \\
& x^{(2)} \le (a+b) \cdot \delta_2 , \quad y^{(2)} = 0 \\
& x^{(3)} = 0 , \quad y^{(3)} \le (a+b) \cdot \delta_3 \\
& x = x^{(1)} + x^{(2)} + x^{(3)} \\
& y = y^{(1)} + y^{(2)} + y^{(3)} \\
& \delta_1 + \delta_2 + \delta_3 = 1 \\
& \quad x^{(i)}, y^{(i)} , \; i = \{1,2,3\} \; , \text{ are continuous variables} \\
& \quad \delta_i , \; i = \{1,2,3\} \; , \text{ are binary variables}
\end{aligned}
\tag{7}
$$

Some simplifications are possible: Since $x^{(3)}$ and $y^{(2)}$ are zero, they can be eliminated; one of the three binaries also can be replaced by the two others.

Using a variant of the reformulation (5), one gets the following (non-sharp) model, (where the lower bound most be replaced by the upper bound, since $Ax - b \ge 0$ is $-Ax + b \le 0$ and the lower bound of $-Ax + b \le 0$ is the upper bound of $Ax - b \ge 0$):

$$
\begin{aligned}
& x - a \le b \cdot (1 - \delta_1) \\
& y - b \le a \cdot (1 - \delta_1) \\
& y \le (a+b) \cdot (1 - \delta_2) \\
& x \le (a+b) \cdot (1 - \delta_3) \\
& \delta_1 + \delta_2 + \delta_3 \ge 1 \\
& \quad \delta_i , \; i = \{1,2,3\} \; , \text{ are binary variables}
\end{aligned}
\tag{8}
$$

Note that the lower and upper bounds on the constraints are known and calculated from the lower and upper bounds of the involved variables. These bounds are:

$$
0 \le x, y \le a + b \tag{9}
$$

Formulation (8) can be simplified and one of the binaries can be eliminated, but this does not make the formulation any sharper.

Another way to model the disjunctive feasible space is to say, that if *x* exceeds *a* then *y* must be zero and if *y* exceeds *b* then *x* must be zero:

$$
(x > a \rightarrow y \le 0) \; \wedge \; (y > b \rightarrow x \le 0)
\tag{10}
$$

which is

$$(x \le a \vee y \le 0) \ \wedge \ (y \le b \vee x \le 0) \tag{10a}$$

This formulation makes an implicit use of the bounds (9). It is essentially a conjunctive normal formulation.
Applying the transformation (5) to the two clauses separately, generates the following model:

$$\begin{aligned} & x - a \le b(1 - \delta_1) \\ & y - b \le a(1 - \delta_3) \\ & y \le (a + b)(1 - \delta_2) \\ & x \le (a + b)(1 - \delta_4) \\ & \delta_1 + \delta_2 \ge 1 \\ & \delta_3 + \delta_4 \ge 1 \end{aligned} \tag{11}$$

This model can also be simplified by eliminating two binary variables, giving:

$$\begin{aligned} & x - a \le b(1 - \delta_1) \\ & y - b \le a(1 - \delta_2) \\ & y \le (a + b)\delta_1 \\ & x \le (a + b)\delta_2 \end{aligned} \tag{12}$$

Again (12) is not sharper than (11).

To check that (12) models the mixed (logico-mathematical) expression (10), consider the following four cases:

1) $\delta_1 = \delta_2 = 0$, then (12) reduces to

$$\begin{aligned} & x - a \le b \\ & y - b \le a \\ & y \le 0 \\ & x \le 0 \end{aligned}$$

since all four constraints must be true, this means that $x = y = 0$.

2) $\delta_1 = 1$, $\delta_2 = 0$, then (12) reduces to

$$x \le a$$
$$y \le a + b$$
$$y \le a + b$$
$$x \le 0$$

since all four constraints must be true, this means that $x = 0$, $y \le a + b$.

3) $\delta_1 = 0$, $\delta_2 = 1$, then (12) reduces to

$$x \le a + b$$
$$y \le b$$
$$y \le 0$$
$$x \le a + b$$

since all four constraints must be true, this means that $x = a + b$, $y = 0$.

4) $\delta_1 = \delta_2 = 1$, then (12) reduces to

$$x \le a$$
$$y \le b$$
$$y \le a + b$$
$$x \le a + b$$

since all four constraints must be true, this means that $x \le a$, $y \le b$.

By the way, one could note that (12) is the most compact form to model the feasible space: the model formulation in (4) uses two 0–1 variables to map a space containing three disjunctive subspaces. It is not possible to use less than two 0–1 variables to model this space.
Of course, (12) is logical not equivalent to (10), but (12) implies (10). That is whenever (12) is true than also (10) must be true, if however (12) is false nothing can be said about the truth of (10) – it could be true or false.

3 OVERVIEW AND INTERPRETATION OF LOGICAL OPERATORS

In this section, a overview of all logical operators involved in the translation procedure is given, then several definitions are presented to transform these operators into each other, finally we deal with mixed (logico-mathematical) expressions and introduce the concept of *proposition definition*. The modeling language LPL [Hürlimann 1997] is used to illustrate a concrete syntax.

THE LOGICAL OPERATORS

Table 1 summarizes all logical operators which are defined in the modeling language LPL and that can be used in the formulation of logical model constraints.

Operator	Alternative formulation	Interpretation
	(x and y are any logical sub-expression containing variables)	
unary operators		
~x		x is false (NOT)
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 OR ~y	y implies x (reverse 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

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]<>0)};
```

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.

REDUCTIONS AND DEFINITIONS

The operators ~ (NOT), AND, OR have the usual meaning (Table 2). It is possible to eliminate one of the two binary operators AND or OR, using the identities:

$$x \wedge y \equiv \neg(\neg x \vee \neg y) \quad \text{and} \quad x \vee y \equiv \neg(\neg x \wedge \neg y)$$

or it is even possible to reduce all of them to either the binary NAND or the binary NOR operator, but these reductions will not be used in the procedure described below. The following formulas, however, are important.

The operator -> is the implication; it is defined as (note that ¬ is also the NOT-operator)

$$x \to y \underset{def}{=} \neg x \vee y \tag{13}$$

The operator <- is the reverse implication; it is defined as

$$x \leftarrow y \underset{def}{=} x \vee \neg y \tag{14}$$

The operator <-> is the equivalence; its definition is:

$$x \leftrightarrow y \underset{def}{=} (x \to y) \wedge (x \leftarrow y) = (x \vee \neg y) \wedge (\neg x \vee y) = (x \wedge y) \vee (\neg x \wedge \neg y) \tag{15}$$

The operator XOR is the exclusive OR (either ... or), it is defined as

$$x \dot{\vee} y \underset{def}{=} \neg(x \leftrightarrow y) = (x \vee y) \wedge (\neg x \vee \neg y) = (x \wedge \neg y) \vee (\neg x \wedge y) \tag{16}$$

Two other operators are the NOR (neither ... nor) and the NAND, which are the negations of the OR and the AND operators. They are defined as

$$x \text{ NAND } y \underset{def}{=} \neg(x \wedge y)$$

$$x \text{ NOR } y \underset{def}{=} \neg(x \vee y)$$

(17)

Table 2 gives an overview of the interpretation ("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	NOT y
1	1	1	1	0	1	1	0	0	0
1	0	0	1	1	0	0	1	0	1
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 are used as *binary* as well as *index operators* in LPL. As an example, the expression

```
AND{i in I} a[i]
```

is the same as

```
a[1] and a[2] and ... and a[n]
```

supposing that the set *I* is defined as $I = \{1,\ldots,n\}$.
Furthermore

```
x AND y
```

can also be written as

```
AND(x,y)
```

For a detailed syntax of logical expressions see the Reference Manual of LPL 4.25 [Hürlimann 1997].
Note that XOR, NOR, and NAND are not associative which means 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” to make the whole expression true (a[i] being a Boolean expression).

```
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”.

In the following part, several identities are presented which allow one to transform different logical operators into each other.
The ATLEAST operator can be expressed as a conjunctive normal form:

$$\mathrm{ATLEAST}(k)_{i\in I}\, X_i \equiv \bigwedge_{S\subseteq I,|S|=n-k+1} \left(\bigvee_{i\in S} X_i\right) \tag{18}$$

which is the conjunction of $\binom{n}{n-k+1}$ clauses consisting each of exactly *k* positive propositions.

The indexed AND, FORALL and ATLEAST(all) are the same (If all expressions must be true then at least all are true). Furthermore, the indexed OR, EXIST and ATLEAST(1) are the same. These facts are expressed by the formula:

$$\begin{aligned} &AND_{i=1}^{n} x_i \equiv \forall_{i=1}^{n} x_i \equiv ATLEAST(n)_{i=1}^{n} x_i \\ &OR_{i=1}^{n} x_i \equiv \exists_{i=1}^{n} x_i \equiv ATLEAST(1)_{i=1}^{n} x_i \end{aligned} \tag{19}$$

The indexed NAND means “not all”, which is the same as “at least one is false” or “at most all but one are true”. The indexed NOR means “none is true”, which is the same as “all are false” or “at least all are false” or at most zero is true”. The indexed XOR means “exactly 1 out of *n* is true”. This is expressed by the formula:

$$\begin{aligned} &NAND_{i=1}^{n} x_i \equiv \neg(\forall_{i=1}^{n} x_i) \equiv \exists_{i=1}^{n}(\neg x_i) \equiv ATLEAST(1)_{i=1}^{n}(\neg x_i) \equiv ATMOST(n-1)_{i=1}^{n} x_i \\ &NOR_{i=1}^{n} x_i \equiv \neg(\exists_{i=1}^{n} x_i) \equiv \forall_{i=1}^{n}(\neg x_i) \equiv ATLEAST(n)_{i=1}^{n}(\neg x_i) \equiv ATMOST(0)_{i=1}^{n} x_i \\ &XOR_{i=1}^{n} x_i \equiv EXACTLY(1)_{i=1}^{n} x_i \end{aligned} \tag{20}$$

The ATMOST-operator can be expressed by the ATLEAST-operator and vice versa. “At least *k* out of *n* are true” means that “at most *n–k* out of *n* are false” and likewise “at most *k* out of *n* are true” means that “at least *n–k* out of *n* are false”. Furthermore, if it is false that “at least *k* out of *n* are true” then it is true that “at most *k*–1 out of *n* are true”, and likewise, if it is false that “at most *k* out *n* are true” then it is true that “at least *k*+1 out of *n* are true”. These facts are expressed by the formula (21).

$$
\begin{aligned}
&ATLEAST(k)_{i=1}^{n}(x_i) \equiv ATMOST(n-k)_{i=1}^{n}(\neg x_i)\\
&ATMOST(k)_{i=1}^{n}(x_i) \equiv ATLEAST(n-k)_{i=1}^{n}(\neg x_i)\\
&\neg(ATLEAST(k)_{i=1}^{n}(x_i)) \equiv ATMOST(k-1)_{i=1}^{n}(x_i)\\
&\neg(ATMOST(k)_{i=1}^{n}(x_i)) \equiv ATLEAST(k+1)_{i=1}^{n}(x_i)\\
&\quad \text{where } 0 \le k \le n
\end{aligned}
\tag{21}
$$

Note also that $ATLEAST(k)$ with $k > n$ and $ATMOST(k)$ with $k < 0$ is defined to be false; likewise $ATMOST(k)$ with $k \ge n$ and $ATLEAST(k)$ with $k \le 0$ is defined to be true.

Table 3 gives a complete overview of the relationship between the ATMOST and the ATLEAST operator and there relation to the indexed AND and OR operator.

k	$E \equiv ATLEAST(k)_{i=1}^{n}(x_i)$ $(S = \{1 \ldots n\})$ $\equiv ATMOST(n-k)_{i=1}^{n}(\neg x_i)$	$\neg E \equiv ATMOST(k-1)_{i=1}^{n}(x_i)$ $\equiv ATLEAST(n-k+1)_{i=1}^{n}(\neg x_i)$
$k \le 0$	TRUE	FALSE
$k = 1$	$\vee_{i=1}^{n}(x_i)$	$\wedge_{i=1}^{n}(\neg x_i)$
$k = 2$	$\bigvee_{S:\lvert S\rvert=2}(\wedge_{i\in S}x_i) \equiv \bigwedge_{S:\lvert S\rvert=n-1}(\vee_{i\in S}x_i)$	$\bigwedge_{S:\lvert S\rvert=2}(\vee_{i\in S}\neg x_i) \equiv \bigvee_{S:\lvert S\rvert=n-1}(\wedge_{i\in S}\neg x_i)$
...	...	...
$k = j$	$\bigvee_{S:\lvert S\rvert=j}(\wedge_{i\in S}x_i) \equiv \bigwedge_{S:\lvert S\rvert=n-j+1}(\vee_{i\in S}x_i)$	$\bigwedge_{S:\lvert S\rvert=j}(\vee_{i\in S}\neg x_i) \equiv \bigvee_{S:\lvert S\rvert=n-j+1}(\wedge_{i\in S}\neg x_i)$
...	...	...
$k = n-1$	$\bigvee_{S:\lvert S\rvert=n-1}(\wedge_{i\in S}x_i) \equiv \bigwedge_{S:\lvert S\rvert=2}(\vee_{i\in S}x_i)$	$\bigwedge_{S:\lvert S\rvert=n-1}(\vee_{i\in S}\neg x_i) \equiv \bigvee_{S:\lvert S\rvert=2}(\wedge_{i\in S}\neg x_i)$
$k = n$	$\wedge_{i=1}^{n}(x_i)$	$\vee_{i=1}^{n}(\neg x_i)$
$k > n$	FALSE	TRUE

Table 3: ATLEAST versus ATMOST

The operator EXACTLY can be expressed by the following formula:

$$EXACTLY(k)_{i=1}^{n} x_i \equiv ATMOST(k)_{i=1}^{n} x_i \wedge ATLEAST(k)_{i=1}^{n} x_i \tag{22}$$

MIXING LOGICAL AND MATHEMATICAL OPERATORS

The mathematical operators consists of the six relational operators <, <=, >, >=, =, and <>. The binary arithmetic operators are +, –, *, / (and others which are not important here). The unique indexed arithmetic operator used here is SUM (summation). It is written as follows:

```
SUM{i in I} a[i]
```

which is interpreted as

```
a[1] + a[2] + ... + a[n]
```

supposing that the set *I* is defined as $I = \{1,\ldots,n\}$.

Arithmetic operators have higher precedence than relational operators, and relational operators have higher precedence than logical ones.
Using these three types of operators, one can form mixed constraints such as:

$$\begin{aligned} &(x \le a \vee y = 0) \rightarrow (p \wedge \neg q) \\ &(r \leftarrow q) \wedge (x = 10) \veebar (z = x + y) \end{aligned} \tag{23}$$

supposing that *x*, *y*, and *z* are continuous variables, *a*, *b*, and *c* are numerical parameters, and *p*, *q*, and *r* are Boolean propositions.
The first constraint says that if *x* is less or equal to *a* or if *y* is zero then *p* and not-*q* cannot be true at the same time. The second means that either *q* implies *r* and *x* is ten or *z* is *x*+*y*. Note that the four subexpressions $x \le a, y = 0, x = 10, z = x + y$ are of type Boolean, so the constraints in (23) make sense.
By adopting the convention, that the Boolean is a subtype of the numerical type by interpreting TRUE as “any number different than zero” and FALSE as “zero”, any combination of operators is not only syntactically correct but also semantically meaningful. This convention is handy in the context of the presented translation procedure, since *propositions* can be interpreted directly as *0–1 variables*. In fact, within the modeling language LPL no distinction is made between these two concepts.

In practical modeling, it generally does not make sense to have continuous variables forming mathematical expressions and propositions forming Boolean subexpressions which both are independent from each other as in (23). Often the modeler wants to introduce *propositions* which substitutes a complicated Boolean expressions. An example is: “Let *p* be true if and only if the quantity *x* of a product manufactured is strictly positive”. This could be formulated as:

$$p \leftrightarrow x \neq 0$$

The general form is: “Let us introduce a proposition *p* which is true if and only if a Boolean expression *E* is true”. The formulation is

$$p \leftrightarrow E \tag{24}$$

An other common form is: “Let *p* be true only if[1] a Boolean expression *E* is true”. This can be formulated as:

$$p \rightarrow E \tag{25}$$

A third often used form, especially in fixed charge problems, is: “Let the proposition *p* be false if an expression *E* is true”. This can be formulated as:

$$\neg p \rightarrow E \tag{26}$$

Note that the contraposition $p \leftarrow \neg E$ is equivalent to (26).

The three common forms to introduce Boolean propositions into an otherwise mathematical model are very similar and in practical modeling building it is sometimes not trivial to chose the right one. But the differences between them are capital. Since these forms are so convenient and often used in practical modeling, in LPL the modeler can declare and *define propositions* in the form of 0–1 variables declarations as follows (where *E* is any Boolean expression containing continuous variables and other propositions):

```
VARIABLE
  p BINARY : E;        (* which means: p → E  *)
  q BINRAY ~: E;       (* which means: ¬p → E  *)
  r BINARY <->: E;     (* which means: p ↔ E  *)
```
(27)

These definitions of propositions behave as if they were model constraints of the form:

```
CONSTRAINT
```

[1] The term “only if ... is true” in English often means “implies”.

```
Cp : p --> E;
Cq : ~p -> E;                                                    (27a)
Cr : p <-> E;
```

The three forms in (27) are also essential in the general translation procedure presented in the next section. Often complex mixed expressions are broken into smaller pieces by introducing additional propositions as defined in (27). The way and the order, in which this is done, is absolutely fundamental in generating good MIP formulations. We now present the translation procedure implemented in LPL (version 4.28) step by step.

4 THE TRANSLATION PROCEDURE

The translation of mixed constraints, containing logical and mathematical operators, is done in eight sequential steps. Within each step a determined number of symbolic transformation rules are applied to each node of the syntax tree of a constraint. The overall structure of the procedure is given in Table 4.

Starting point: A constraint in LPL using the operators in Table 1.
Step 1: Several logical operators are eliminated at parse time. This step is applied to *every* expression.
Step 2: Several operators are eliminated after parse-time. This step and the following steps are only applied to model constraints and proposition definitions as defined in Section 3.
Step 3: Some special constructs are replaced.
Step 4: The NOT-operator is pushed inwards within the expression.
Step 5: Complex constraints are broken into parts and new propositions are introduced in one of the form of (27).
Step 6: XOR and <-> are eliminated. The OR-operator is pushed inwards over the AND-operator (to generate a conjunctive normal form.
Step 7: Further special constructs are replaced.
Step 8: All remaining logical operators are eliminated and linear constraints are generated.

Table 4: Overall Structure of the Algorithm

We now explain the single steps. In the subsequent tables, the rules are listed in the form

```
Nr   Expression1  ::=  Expression2
```

which means that *Expression1* is replaced by *Expression2* symbolically. *Nr* identifies the rule by a number. How exactly this is done internally, depends on the implementation of the symbol table and the syntax tree and can be quite involving. In this paper, we are not concerned about a particular implementation and try to expose the rules in an implementation neural way. For all rules, however, we use LPL syntax as defined in [Hürlimann 1997].

STEP 1: OPERATOR REDUCTIONS

Table 5 collects all rules which are applied to *every* expression (not only constraints) at parse-time.

```
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]

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

Table 5: Elimination of Operators at Parse-Time

(There are other rules implemented in LPL which do not apply to logical expressions, therefore, they have been left out here.)
Rule 2 and 3 removes the binary operators NOR and NAND using the definition (17). Rules 4 to 8 replaces the indexed operators AND, OR, XOR, NOR, and NAND using the definitions (19) and (20). Rule 10 implements the convention that several relational operators in sequence must hold individually. An example shows this practical rule. The expression:

$$x \le y \le z \le w$$

for example, is directly translated into the following expression:

$$(x \le y) \wedge (y \le z) \wedge (z \le w).$$

STEP 2: FURTHER REDUCTIONS

Table 6 collects a number of rules which are applied to constraints only after the model has been parsed completely. The number of logical operators is further reduced by this step.

```
1   X -> Y                    ::=   ~X OR Y
1a  X <- Y                    ::=   X OR ~Y
11  FORALL{i} X[i]            ::=   ATLEAST(#i){i} X[i]
12  EXIST{i} X[i]             ::=   ATLEAST(1){i} X[i]
13  ATMOST(k){i} X[i]         ::=   ATLEAST(n-k){i} ~X[i]
```

```
14  EXACTLY(0){i} X[i]          ::=   ATMOST(0){i}  X[i]
15  EXACTLY(#i){i} X[i]         ::=   ATLEAST(#i){i} X[i]
```

Table 6: Eliminate Operators after Parse-Time

The rules 1 and 1a replace the operators -> and <- using the definitions (13) and (14). The rules 11 and 12 replaces the indexed operators FORALL and EXIST using again the definition (19). One may ask why these rules have not been applied in step 1. This would have been possible, but for several technical reasons of little interest in this context, it was decided to separate these rules. The rule 13 replaces the operator ATMOST using the definition (21). The rules 14 and 15 treat two special cases. Such rules are typical and a great number of similar ones could be applied to make the translation more powerful. Since the operator EXACTLY can produce many unnecessary linear constraints, it makes sense to eliminate it as early as possible in the translation process. (One should note that *#i* means "the cardinality of *i*.)

STEP 3: REPLACE SOME SPECIAL CONSTRUCTS

This step could include a large number of special constructs that are difficult to handle later on. In LPL, actually, only two of them are implemented (Table 7).

```
17  x=a                 ::=   if(a,x,~x)
18  if(w,y,z)           ::=   (~w OR y) AND (w OR z)
     (a is a 0-1 parameter, x is a binary variable, w is an
expression
      containing variables, y and z are any expressions.)
```

Table 7: Reduction of Further Constructs

The syntactical construct *if(a,b,c)* within an expression means that if *a* is true (different from zero) then *b* must be returned else *c* must be returned. In a constraint, if *a* in *if(a,b,c)* is a parameter then this can be interpreted as either *b* or *c* depending on the value of *a*. Hence a constraint

```
3*x + if(a, 2*y, 4*z) - 5*w
```

simply means

```
3*x + 2*y - 5*w
```

or

```
3*x + 4*z - 5*w
```

depending on whether *a* is true (different from zero) or not.
However, if the condition *a* itself contains variables as in rule 18, then the if()

part cannot be removed in this way. In this case, it must be interpreted as:

$$(w \to x) \wedge (\neg w \to y) \tag{28}$$

Now (28) is nothing else then $(\neg w \vee x) \wedge (w \vee y)$ by applying the definition (13) which verifies the rule 18. Rule 17 is very convenient in indexed expressions, because it avoids to generate a large number of redundant linear constraints (an example is the *car.lpl* model which is found in the LPL model library [Hürlimann 1998]).

STEP 4: PUSH THE NOT-OPERATOR

In this step, the NOT-operator (~) is pushed as far inwards in the expression as possible. The rules are listed in Table 8.

Rules 20 to 26 apply trivial Boolean laws. Rules 27 to 29 apply the definitions (21) and (22). The rules 32 and 33 are somewhat arbitrary. Rule 32 simply means that if a continuous variable is negated then it should be zero (since their lower bound is normally zero. However, if a expression *Z* containing continuous variables (rule 33), then it is transformed to be non-zero. One should note that the NOT-operator is eliminated altogether in this step except before a binary variable and before a constant expression (containing no variables).

```
 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])              ::=   ATLEAST(n-k+1) {i} ~X[i]
 29  ~(EXACTLY(k){i} X[i])              ::=   ATLEAST(n-k+1){i} ~X[i]
                                                 OR ATLEAST(k+1){i}
X[i]
 29a ~IF(c,x,y)                         ::=   IF(a,~x,~y)
 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
 34  X<>Y                               ::=   X<Y OR X>Y
```

Table 8: Push the NOT Operator

STEP 5: BREAK THE CONSTRAINT INTO PARTS

At this point the following seven logical operators remain:

OR, XOR <->, ~ (NOT), AND, ATLEAST, and EXACTLY.

This fifth step does not reduce their number, but complex constraint are broken into parts and additional propositions are introduced in the form of (24), (25), or (26). It is very important to do this intelligently, otherwise many redundant propositions will be introduced, *which do not generate a sharper formulation*.

Example: An example illustrates this crucial point. Suppose the formula (6) is given. This constraint could be broken into parts in different ways. One way is to introduce a proposition for every of the three conjunctions as follows:

$$\begin{aligned} &\delta_1 \leftrightarrow x \le a \ \wedge \ y \le b \\ &\delta_2 \leftrightarrow x \le a + b \ \wedge \ y \le 0 \\ &\delta_3 \leftrightarrow y \le a + b \ \wedge \ x \le 0 \end{aligned} \tag{29}$$

the main expression (6) being substituted by

$\delta_1 \vee \delta_2 \vee \delta_3$.

By observing that each subexpression of the conjunctions in (29) can be applied independently, this could be further broken into:

$$\begin{aligned} &\delta_1 \leftrightarrow x \le a \\ &\delta_1 \leftrightarrow y \le b \\ &\delta_2 \leftrightarrow x \le a + b \\ &\delta_2 \leftrightarrow y \le 0 \\ &\delta_3 \leftrightarrow y \le a + b \\ &\delta_3 \leftrightarrow x \le 0 \end{aligned}$$

Note that no further propositions need to been introduced.

An naive procedure, however, could have broken the expressions in (29) further into:

$$\begin{aligned} &\delta_4 \leftrightarrow x \le a \\ &\delta_5 \leftrightarrow y \le b \\ &\delta_6 \leftrightarrow x \le a + b \\ &\delta_7 \leftrightarrow y \le 0 \\ &\delta_8 \leftrightarrow y \le a + b \\ &\delta_9 \leftrightarrow x \le 0 \end{aligned}$$

expression (29) being substituted by

$$\delta_1 \leftrightarrow \delta_4 \wedge \delta_5$$

$$\delta_2 \leftrightarrow \delta_6 \wedge \delta_7$$

$$\delta_3 \leftrightarrow \delta_8 \wedge \delta_9$$

€

This simple example shows that it is extremely important how to break the expression into parts and to introduce new propositions. It is, therefore, not surprising that the condition, on how this is done in LPL, is quite complicated. To understand it, let us imagine that the constraint is represented as a syntax-tree. The syntax-tree is traversed in a preorder way, visiting the node before traversing its subtrees. *Visiting the node* means that a complex condition is checked and if it is true the corresponding subtree is detached and replaced by a single node representing a newly introduced proposition. This is done in a way as to introduce the least possible additional propositions.

Basically, the visiting procedure of node *n* compares its label with the label of its parent node *pn*. Table 9 lists all combinations and displays a "Y" if a subtree must be split for a particular (*n*,*pn*)-tuple at a node *n*. (A "y" means that the subtree is detached only if it contains a mathematical operator.)

n \ *pn*	ATL	Or{}	And{}	Xor{}	Iff	Xor	Or	And	xb	else
ATL	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Or{}	Y			Y					Y	Y
And{}	Y	Y		Y	y	y	y		Y	Y
Xor{}	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Iff	Y			Y					Y	Y
XOR	Y			Y					Y	Y
Or	Y			Y					Y	Y
And	Y			Y					Y	Y
xb										
else	Y	Y	Y	Y	Y	Y	Y			

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

The operators and operands in Table 9 are grouped into classes as follows:

```
ATL   = ATLEAST(k) or EXACTLY(k)   (k not in {#i,1})
Or{}  = ATLEAST(1)
And{} = ATLEAST(#i)
Xor{} = EXACTLY(1)
Iff   = <->
Xor   = XOR
```

```
Or    = OR
And   = AND or <comma-operator>
xb    = <binary variable>, <NOT-operator>
else  = <all other operators and operands>
```

Example: An example may illustrate this manipulation. The expression (see (10a)):

$$(x \le a \vee y \le 0) \ \wedge \ (y \le b \vee x \le 0)$$

can be represented by a syntax-tree as shown in Figure 2, each node representing an operator or an operand.

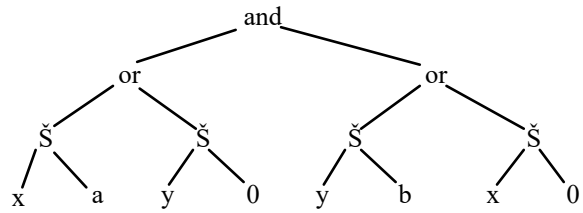

Figure 2: a Syntax-tree

Figure 3 shown the result, in which a subtree of Figure 2 is detached and a new 0–1 variable δ has been introduced.

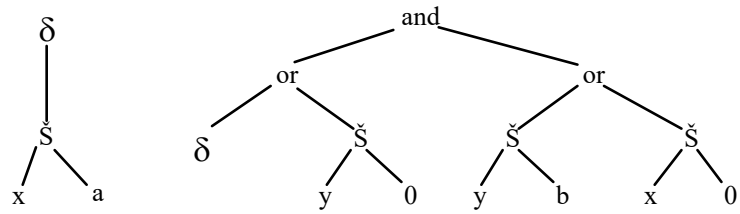

Figure 3: Detached Subtree

The detached part generates one of the formula (24) to (26). The proposition δ is attached as an operand to the original syntax tree at the point where the subtree was removed. The result of this operation consists of a constraint representing the pruned tree and a *proposition definition* of the form (24), (25), or (26) representing the cut tree.

$(\delta \vee y \le 0) \ \wedge \ (y \le b \vee x \le 0)$ (the pruned constraint)

$\delta : x \le a$ (the proposition definition)

€

The procedure is recursively applied to the proposition definition, thus breaking it into further subtrees eventually.

STEP 6: GENERATE A CNF

Step 6 eliminates the XOR and the <-> operator and generates a conjunctive form of each constraint. The rules of Table 10 are applied.

```
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)

45   x OR AND{i} y[i]             ::= AND{i} (x OR y[i])
46   AND{i} y[i] OR x             ::= AND{i} (y[i] OR x
47   OR{i} (x[i] AND y[i]})       ::= OR{i} x[i] AND OR{i} y[i]}
```

Table 10: Generate a CNF

The rules 40 and 41 apply the definitions (15) and (16). It is important to note that these two rules introduce NOT-operators, which must be pushed inwards again by applying the step 4 and 5 to the corresponding subexpressions. The rules 42 and 43 push the OR-operator inwards in order to generate a conjunctive normal form.
ATLEAST(1) is the same as the indexed OR-operator and for technical reasons ATLEAST(0) is interpreted as indexed AND. Hence, the rules 45 and 46 push the binary OR inwards even over a indexed AND, whereas rule 48 redistribute a conjunction over an indexed OR. These rules are extremely useful in avoiding to introduce additional redundant propositions.

STEP 7: FURTHER REPLACEMENTS

In this step, the strictly-less- and strictly-greater-operators are replaced (Table 11). This requires a small number *e* to be introduced. If both *x* and *y* are integer expressions the number *e* can be chosen to be one.

```
91  x < y                ::=   x + e <= y
92  x > y                ::=   x >= y + e
95  MAX{i} X[i]          ::= y  (y is a new continuous variable)
                     and add constraint: y >= X[i]
96  MIN{i} X[i]          ::= y  (y is a new continuous variable)
                     and add constraint: y <= X[i]
```

Table 11: Further Replacements

For completion, we mention that the indexed MIN and MAX-operators within constraints are also replaced in this step, although this has nothing to do with

the logical to IP translation.

STEP 8: ELIMINATES ALL REMAINING LOGICAL OPERATORS

In this last step of the translation procedure, all logical operators are replaced, constraints are further split and linear constraints are generated. It is important to distinguish constraints of the form (24)–(26) and ordinary constraints. First, Table 12 gives the rules for ordinary constraints.

```
50   X AND Y AND ...                 ::=  X, Y, ...
51   x OR y OR ...                   ::=  x + y + ... >= 1
54   ATLEAST(k){i} x[i]              ::=  SUM{i} x[i] >= k
56   EXACTLY(k){i} x[i]              ::=  SUM{i} x[i]  = k
58   ~x    (x is a binary var)       ::=  x<=0  (complete constraint)
58a  ~x   (x is a binary var)        ::=  1-x   (in a subexpression)
59   x     (x is a binary variable)  ::=  x>=1  (complete expression)
```

Table 12: Remove the remaining Operators

Rule 50 says that constraints containing the AND operator can simply be split into several constraints *X*, *Y*, etc. Note that the AND-operator can only occur at the top of the syntax-tree at this point, since step 6 generated a conjunctive normal form. Rule 51 replaces the OR operator by the addition operator. This assumes that all *x*, *y*, etc. are proposition variables. The splitting operation and step 6 with pushes the OR-operator inwards guarantee that this is indeed the case. The same holds for rule 54 and 56. The rules 58 and 59 apply, if the constraints only consists of the proposition *x* or its negation; if, however, a negated proposition occurs within a constraints containing other operators and operands then rule 58a applies.

Table 13 lists the rules which apply for proposition definitions in the form of (24)–(26) which have been generated in step 4 or which have been declared and defined by the modeler.

```
60   p :  X AND Y AND ...            ::=  p : X , p : Y , ...
61   p :  x OR y OR ...              ::=  x + y + ... >= p
64   p :  ATLEAST(k){i} x[i]         ::=  SUM{i} x[i] >= k*p
66   p :  EXACTLY(k){i} x[i]         ::=  SUM{i} x[i]  = k*p
68   p :  x                          ::=  x >= p
69   p :  ~x                         ::=  1-x >= p

70   p :  ∑ ax <= b            ::=  ∑ ax-b <= U(∑ ax-b)*(1-p)
71   p :  ∑ ax >= b            ::=  ∑ ax-b >= L(∑ ax-b)*(1-p)
72   p :  ∑ ax = b             ::=  p : ∑ax >= b  ,  p : ∑ax <= b

80   p ~:  ∑ ax <= b           ::=  ∑ ax-b <= U(∑ ax-b)*p
81   p ~:  ∑ ax >= b           ::=  ∑ ax-b >= L(∑ ax-b)*p
```

82	p ~: ∑ ax = b	::= p ~: ∑ax >= b , p ~: ∑ax <= b
83	p <->: X	::= p : X , p ~: ~X

Table 13: Remove the Operators from Proposition Definitions

Rule 60 is similar to rule 50 and splits a conjunction into several parts getting rid of the AND-operator. The rules 61 implements the equivalence of the Boolean expression with the propositions *p*, *x*, *y*, ...

$$p \rightarrow (x \vee y \vee \ldots) \tag{30}$$

and the linear constraint with the 0–1 variables *p*, *x*, *y*, ...

$$x + y + \ldots \geq p \tag{31}$$

To prove the equivalence between (30) and (31), one should note that (30) is false only if all propositions *x*, *y*, ... are false and *p* is true. Likewise (31) is false only if all variables *x*, *y*, ... are zero and *p* is one. Similar arguments holds for the rules 64, 66, 68, and 69.

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

$$L \leq \sum_{j=1}^{n} a_j x_j - b \leq U$$

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 the LPL modeling language both methods are possible.) In the second case, these bounds are calculated using the following formula [Brearley/Mitra/Williams 1975]:

$$L = \sum_{j \in \{j: a_j > 0\}} a_j l_j + \sum_{j \in \{j: a_j < 0\}} a_j u_j - b$$

$$U = \sum_{j \in \{j: a_j > 0\}} a_j u_j + \sum_{j \in \{j: a_j < 0\}} a_j l_j - b$$

where $l_j \leq x_j \leq u_j$.

Rule 70 establishes the equivalence between the Boolean expression

$$p \rightarrow \sum_{j=1}^{n} a_j x_j \leq b \tag{32}$$

and the linear, mixed integer constraint

$$\sum_{j=1}^{n} a_j x_j - b \leq U\left(\sum_{j=1}^{n} a_j x_j - b\right)(1-p) \tag{33}$$

To prove the equivalence between (32) and (33), we first suppose that p is true (p=1). Then (32) is true only if $\sum_{j=1}^{n} a_j x_j \leq b$. In this case ($p$=1), the formula (33) reduces to $\sum_{j=1}^{n} a_j x_j - b \leq 0$ which is the same as $\sum_{j=1}^{n} a_j x_j \leq b$, which proves the first part.
Suppose now that p is false (p=0). In this case, (32) is true trivially and (33) reduces to $\sum_{j=1}^{n} a_j x_j - b \leq U\left(\sum_{j=1}^{n} a_j x_j - b\right)$ which is also true trivially, since $U(X)$ is defined to be an upper bound on X. In any case, (32) and (33) are equivalent. A similar argument hold for the rule 71.
Rule 72 simply reduces the expression to the rules 70 and 71.

Rule 80 establishes the equivalence between the Boolean expression

$$\neg p \rightarrow \sum_{j=1}^{n} a_j x_j \leq b \tag{34}$$

and the linear, mixed integer constraint

$$\sum_{j=1}^{n} a_j x_j - b \leq U\left(\sum_{j=1}^{n} a_j x_j - b\right) p \tag{35}$$

To prove the equivalence between (34) and (35), we only need to replace p by ~p (p by 1–p) and their equivalence is reduced to the equivalence of (32) and (33). A similar argument hold for the rule 81.
Rule 82 again reduces to rule 70 and 71. Finally, rule 83 reduces

$$p \leftrightarrow X$$

to

$$(p \rightarrow X) \wedge (\neg p \rightarrow \neg X).$$

The procedure now is complete. In the next section, several applications are given which illustrates the translation procedure.

5 APPLICATIONS

Four examples are presented to illustrate the translation procedure. The first two examples are simple examples, the last two are complete models. All examples are coded in the modeling language LPL 4.28 and the output of intermediate steps was generated directly by the procedure implemented in LPL.

EXAMPLE 1: A BOOLEAN EXPRESSION

For arbitrary Boolean expressions which only contain binary connectors together with the NOT-operator, it is normally best to generate a complete conjunctive normal form without introduce additional propositions to break it in part. This can be seen in our first example which corresponds to the formula (1). In LPL, this constraint can be coded as follows:

```
VARIABLE  x, y, z, w BINARY;
CONSTRAINT  r : ((x AND y) OR z) XOR w;
```

(The parentheses are even not necessary.) In step 6 of the procedure, LPL generates the following constraint:

```
CONSTRAINT r : (x OR z OR w) AND (y OR z OR w) AND (~x OR ~y OR ~w) AND (~z OR
~w)
```

Finally in step 8 this is translated into the four constraints:

```
CONSTRAINT  r : x+z+w>=1
CONSTRAINT  X14 : 1-x+1-y+1-w>=1
CONSTRAINT  X15 : 1-z+1-w>=1
CONSTRAINT  X16 : y+z+w>=1
```

which corresponds to the linear system (2).

EXAMPLE 2: FLEXIBLE STORAGE

In section 2, an example was presented which mixes logical and mathematical operators. Here we see how the procedure transforms it. In LPL, the constraint (10) can be formulated as follows:

```
PARAMETER   a; b;             (* container capacities *)
VARIABLE    x,y [0,a+b];      (* lower and upper bounds of x and y *)
CONSTRAINT C:  (x>a -> y<=0)  AND  (y>b -> x<=0);
```

The translation procedure of LPL modifies the constraint in step 2 to:

```
CONSTRAINT  r : (~(x>a) OR y<=0) AND (~(y>b) OR x<=0)
```

Step 4 pushes the NOT-operator inwards giving:

```
CONSTRAINT  r : (x<=a OR y<=0) AND (y<=b OR x<=0)
```

Step 5 detaches four subtrees which introduces each a proposition as follows:

```
CONSTRAINT  r : (X16 OR X17) AND (X18 OR X19)
VARIABLE  X16 BINARY  : x<=a
VARIABLE  X17 BINARY  : y<=0
VARIABLE  X18 BINARY  : y<=b
VARIABLE  X19 BINARY  : x<=0
```

In step 8 the AND- and the OR-operators are removed giving:

```
CONSTRAINT  r  : X16+X17>=1
CONSTRAINT  X20  : X18+X19>=1
VARIABLE  X16 BINARY  : x<=a
VARIABLE  X17 BINARY  : y<=0
VARIABLE  X18 BINARY  : y<=b
VARIABLE  X19 BINARY  : x<=0
```

This corresponds to the linear system (11). LPL makes a further reduction on the number of 0–1 variables which finally generates the linear system (12).

EXAMPLE 3: A ENERGY IMPORT MODEL

This very instructive example is from [Mitra al. 1994]. The problem can be formulated as follows:

> 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. Furthermore, there are upper and lower limits in the imported quantities from a country. What quantities should be imported to meet the demand, if the import costs have to be minimized?

Three sets must be declared

I = {low, medium, high}	the quality grades of energy
J = {gas, coal}	non-nuclear energy sources
K = {GB FR IC}	countries from which energy is imported

With $i \in I, j \in J, k \in K$, the parameters are:

c_{ijk}	(non-nuclear) energy unit costs imported,
l_{ijk}, u_{ijk}	minimum and maximum amount of non-nuclear energy,
nc_k	nuclear energy unit costs,
nl_k, nu_k	minimum and maximum amount of nuclear energy,
e	desired energy amount to import (the demand),
lR_j , uR_j	minimum and maximum percentage of non-nuclear energy if coal and gas are imported (40–50, 20–30%),
lA, uA	minimum and maximum gas take if gas energy only is imported (50–60%).

The numerical variables are:

X_{ijk}	quantity of non-nuclear energy imported,
Y_k	quantity of nuclear energy imported.

To formulate the logical constraints, the following propositions are defined:

P_{ijk}	is defined as : $l_{ijk} \le X_{ijk} \le u_{ijk}$
N_k	is defined as : $nl_k, \le Y_k \le nu_k$

Q_j	is defined as :	$e \cdot lR_j \le \sum_{i \in I} \sum_{k \in K} X_{ijk} \le e \cdot uR_j$
R	is defined as :	$e \cdot lA \le \sum_{i \in I} \sum_{k \in K} X_{i,gas,k} \le e \cdot uA$

The constraint to attain the desired amount of energy is:

$$\sum_{i \in I} \sum_{j \in J} \sum_{k \in K} X_{ijk} = e$$

The three logical constraints can be formulated as follows:

– The supply condition: "Either at most 3 of P_{ijk} are true (where i≠high) or N_k and exactly one $P_{high,j,k}$ is true for all k":

$$\text{ATMOST (3)}_{i \in I, j \in J | i <> high} P_{ijk} \text{ XOR } (N_j \text{ AND XOR}_j P_{high,j,k}) \quad \text{for all } k \in K$$

– The environmental restriction: "if at least one N_k is true then none of P_{ijk} is true (where i≠high)":

$$\text{OR}_k N_j \rightarrow \text{NOR}_{i,j,k | i <> high} P_{ijk}$$

– The energy mixing condition: "If any of $P_{i,gas.k}$ is true then either Q_j is true or R and none of $P_{i,coal,k}$ is true":

$$\text{OR}_{ik} P_{i,gas,k} \rightarrow \text{AND}_j Q_j \text{ XOR } R \text{ AND NOR}_{ik} P_{i,coal,k}$$

The objective is to minimize overall energy import costs:

$$\text{MIN: } \sum_{i \in I} \sum_{j \in J} \sum_{k \in K} c_{ijk} \cdot X_{ijk} + \sum_{k \in K} nc_k \cdot Y_k$$

Table 14: The Energy Import Model

In addition, three further conditions must be fulfilled:

1 The 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 The environmental restriction: Environmental regulations require that nuclear fuel can be used only if medium and low grades of gas and coal are excluded.

3 The 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.

Using mathematical and logical operators as displayed in Table 1, the model can be formulated as shown in Table 14.

The model shown in Table 14 can be coded directly in LPL as follows:

```
MODEL EnergyImport;
SET
  i = /low med high/;
  j = /gas, coal/;
  k = /GB FR IC/;

PARAMETER
  c{i,j,k}; l{i,j,k}; u{i,j,k};  (* data to be specified *)
```

```
  nc{k}; nl{k}; nu{k};
  e;
  lR{j}; uR{j};
  lA; uA;

VARIABLE
  x{i,j,k};
  y{k};
  P{i,j,k} BINARY : l <= x <= u;
  N{k} BINARY : nl <= y <= nu;
  Q{j} BINARY : e*lR <= SUM{i,k} x <= e*uR;
  R BINARY : e*lA <= SUM{i,k} x[i,'gas',k] <= e*uA;

CONSTRAINT
  Cost: SUM{i,j,k} c*x + SUM{k} nc*y;
  ImportReq: SUM{i,j,k} x + SUM{k} y = e;

  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
```

Together with the data tables of all parameters, this formulation is a *complete* instanciated model. It can be processed by the LPL compiler and solved by a mixed integer solver quite efficiently.

Let us now take a look at how LPL processes these logical constraints step by step. The source code for the 4 proposition definitions and the 3 constraints is as follows:

```
VARIABLE
  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
  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];
```

Step 1 and 2 replaces several logical operators. (Note that all subsequent code fragments are generated and written by LPL itself. LPL transforms the statements *symbolically* and can output intermediate steps in symbolic form.) Hence, the resulting code after step 1 and 2 is as follows [note also as already mentioned that for technical reasons ATLEAST(0) is interpreted as ATLEAST(<all>)]:

```
 VARIABLE P{i,j,k} BINARY := l<=x AND x<=u
 VARIABLE N{k} BINARY := nl<=y AND y<=nu
 VARIABLE Q{j} BINARY := e*MinR<= SUM {i,k}x AND  SUM {i,k}x<=e*MaxR
 VARIABLE R BINARY := e*MinA<= SUM {i,k}x[i,1,k] AND  SUM {i,k}x[i,1,k]<=e*MaxA
 CONSTRAINT SuplCond{k} := ATLEAST(1){i,j|i<>'high'} ~P XOR N
                           AND EXACTLY(1){j} P[3,j,k]
 CONSTRAINT Environ:= ~( ATLEAST(1){k} N) OR  ATLEAST(0){i,j,k|i<>'high'} ~P
 CONSTRAINT AltMix:= ~(ATLEAST(1){i,k} P[i,1,k]) OR  ATLEAST(0){j} Q
                     XOR R AND  ATLEAST(0){i,k} ~P[i,2,k]
```

The following rules have been applied:

(1) “X ≤ Y ≤ Z” was replaced by “X ≤ Y and Y ≤ Z” according to Rule 10,
(2) “Indexed XOR” was replaced by “EXACTLY(1)” according to Rule 6,
(3) Implication was replaced according to Rule 1; and, finally
(4) “indexed NOR” was replaced by “ATMOST(0)” according to Rule 7,
(5) “ATMOST(k)” was replaced by “ATLEAST(n-k)” according to Rule 13.

Step 3 does not modify anything. Step 4 is to push the NOT-operator inwards as far as possible. The result (again automatically generated by LPL) is (only modified constraints are repeated):

```
CONSTRAINT Environ:= ATLEAST(0){k} ~N OR  ATLEAST(0){i,j,k|i<>'high'} ~P
CONSTRAINT AltMix:= ATLEAST(0){i,k} ~P[i,1,k] OR  ATLEAST (0){j}Q
                    XOR R AND  ATLEAST(0){i,k} ~P[i,2,k]
```

The unique rule applied was:

(1) “~ATLEAST(k)” is replace by “ATLEAST(n-k+1)”, (Rule 27).

Step 5 does not modify anything. Step 5 only modifies the constraint *SuplCond*. It is broken into a constraint and two proposition definitions according to Table 9 as follows:

```
CONSTRAINT SuplCond{k} : X34[k] XOR (N AND X35[k]);
VARIABLE X34{k} : ATLEAST(1){i,j|i<>'high'} ~P;
VARIABLE X35{k} : EXACTLY(1){j} P[3,j,k];
```

The two proposition variables X34 and X35 are introduced. Note that they are automatically indexed by *k*.

The next step removes the XOR-operator and generates a conjunctive normal form. The resulting modifications are:

```
CONSTRAINT  SuplCond{k} : (X34[k] OR N) AND (X34[k] OR X35[k])
                          AND (~X34[k] OR ~N OR ~X35[k])
CONSTRAINT  Environ :  ATLEAST(0){i,j,k|i<>'high'} ATLEAST(0){k}(~N OR ~P)
CONSTRAINT  AltMix :  ATLEAST(0){j} (ATLEAST(0){i,k} (~P[i,1,k] OR Q OR R))
  AND  ATLEAST(0){i,k} (ATLEAST(0){j} (ATLEAST(0){i,k}(~P[i,1,k] OR Q OR
~P[i,2,k])))
  AND (ATLEAST(1){i,k}P[i,1,k] OR ~R OR  ATLEAST(1){i,k}P[i,2,k])
  AND (ATLEAST(1){j}~Q OR ~R OR ATLEAST(1){i,k}P[i,2,k])
```

In constraint *SuplCond*, the rule 40 was applied, in the constraint *Environ*, rule 45 is used, while in *AltMix* the rules 42, 43, and 45 are combined.
This example shows the advantages of the rules 45–47. Without them, it would have been necessary to break the constraint *AltMix* further into parts by introducing additional propositions as follows:

```
CONSTRAINT AltMix:=(X33 OR X34 OR X35) AND (~X33 OR ~X35) AND (~X35 OR ~X34)
VARIABLE X33 BINARY : ATLEAST(0){i,k} ~P[i,1,k]
```

```
VARIABLE X34 BINARY : ATLEAST(0){j} Q
VARIABLE X35 BINARY : R AND ATLEAST(0){i,k} ~P[i,2,k]
```

In this particular case, this would not have been disastrous, since “only” three additional 0–1 variables are introduced. However, if the variables introduced were indexed, many redundant variables would have entered the formulation.

Step 8 removes all AND-operators. The resulting model is:

```
VARIABLE    P{i,j,k} BINARY  : l<=x
VARIABLE    P*{i,j,k} : x<=u
VARIABLE    N{k} BINARY  : nl<=y
VARIABLE    N*{k} : y<=nu
VARIABLE    Q{j} BINARY  : e*MinR <= SUM{i,k} x
VARIABLE    Q*{j} :  SUM{i,k} x <= e*MaxR
VARIABLE    R BINARY  : e*MinA<= SUM{i,k}x[i,1,k]
VARIABLE    R* :  SUM{i,k} x[i,1,k] <= e*MaxA

CONSTRAINT  SuplCond{k} : X34[k]+N>=1
CONSTRAINT  X42{k} : X34[k]+X35[k]>=1
CONSTRAINT  X40{k} : 1-X34[k]+1-N+1-X35[k]>=1
VARIABLE    X34{k} BINARY  :  SUM{i,j|i<>'high'} (1-P)>=3
VARIABLE    X35{k} BINARY  :  SUM{j} P[3,j,k]<=1
VARIABLE    X35*{k} ~:  SUM{j} P[3,j,k]>=1
CONSTRAINT  Environ{i,j,k,k1 in k|i<>'high'} : 1-N[k]+1-P[i,j,k1]>=1
CONSTRAINT  AltMix{j,i,k} : 1-P[i,1,k]+Q[j]+R>=1
CONSTRAINT  X45 :  SUM{i,k} P[i,1,k]+1-R+ SUM{i,k} P[i,2,k]>=1
CONSTRAINT  X46 :  SUM{j} (1-Q)+1-R+ SUM{i,k} P[i,2,k]>=1
CONSTRAINT  X47{i,k,j,i1 in i,k1 in k} : 1-P[i1,1,k1]+Q[j]+1-P[i,2,k]>=1
```

Finally still in step 8, the rules 70 and 71 apply to all proposition definitions to generate proper linear constraints. The complete MIP-formulation that LPL outputs from the initial 4 propositions and 3 constraints is as follows (here edited by hand for better reading):

```
VARIABLE x{i,j,k}; y{k};   (*continuous *)
  P{i,j,k}; N{k}; Q{j}; R; X34{k}; X35{k}; (*binary*)
CONSTRAINT
  Pa{i,j,k} : x[i,j,k] <= l[i,j,k]*P[i,j,k];
  Pb{i,j,k} : x[i,j,k] <= u[i,j,k];
  Na{k}     : y[k] <= nl[k]*N[k];
  Nb{k}     : y[k] <= nu[k];
  Qa{j}     : SUM{i,k} x[i,j,k] >= e*MinR[j];
  Qb{j}     : SUM{i,k} x[i,j,k] <= e*MaxR[j];
  Ra        : SUM{i,k} x[i,1,k] >= e*MinA;
  Rb        : SUM{i,k} x[i,1,k] <= e*MaxA;

  SuplCondA{k}: X34[k] + N[k] >= 1;
  SuplCondB{k}: X34[k] + X35[k] >= 1;
  SuplCondC{k}: -X34[k] - N[k] - X35[k] >= -2;
  CX34{k}     : SUM{i,j|i<>'high'}P + 3*X34[k] <= 4;
  CX35a{k}    : SUM{j} P[3,j,k] - X35[k] >= 0;
  CX35b{k}    : SUM{j} P[3,j,k] + X35[k] <= 2;
  Environ{i,j,k,k1 in k|i<>'high'}  : 1-N[k] + 1-P[i,j,k1] >= 1
  AltMix{j,i,k}  : 1-P[i,1,k] + Q[j] + R >= 1
  CX45  :  SUM{i,k} P[i,1,k] + 1-R + SUM{i,k} P[i,2,k] >= 1
  CX46  :  SUM{j} (1-Q) + 1-R + SUM{i,k} P[i,2,k] >= 1
  CX47{i,k,j,i1 in i,k1 in k}  : 1-P[i1,1,k1] + Q[j] + 1-P[i,2,k] >= 1
```

It is difficult to imagine how this model formulation could have been compiled

without any automatic generator!

EXAMPLE 4: THE CAPACITATED FACILITY LOCATION PROBLEM

The capacitated facility location problem is an example of a fixed charge problem. Such problems typically arise, if the value of some unknown depends on a condition which is not known to be true or false. A typical case is the warehouse location problem: Where should warehouses be built in order to minimize variable transportation costs to customers and fixed warehouse building costs? Transportation costs depend upon where the warehouse are built. Traditionally, such problems are formulated by introducing a 0–1 variable for each potential location.
The capacitated facility location problem can be stated formally as given in Table 10-8.

Introducing the binary variables x_i has the following consequence. If a plant at location *e* is built (meaning that $x_e = 1$) the fixed costs f_e are added in the minimizing function and the capacity constraint is trivially satisfied. If a plant at location *e* is not built (meaning that $x_e = 0$) the fixed costs f_e are not added in the minimizing function and the capacity constraint says that nothing can be shipped from the plant at location *e*. This is exactly what we need to model.

The sets are:

I	potential plant locations
J	customers

The parameters with $i \in I, j \in J$ are:

$g_{i,j}$	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

The variables are:

$z_{i,j}$	amount shipped from plant i to customer j
x_i	indicates whether the plant i is built (Boolean)

The constraints are:

capacity is limited: $\sum_j z_{i,j} \le M_i \cdot x_i \;, \quad \forall \{i \in I\}$

demands must be satisfied: $\sum_i z_{i,j} = d_j \;, \quad \forall \{j \in J\}$

The objective is to minimize variable and fixed costs:

MINIMIZE: $\sum_{i,j} g_{i.j} \cdot z_{i,j} + \sum_i f_i \cdot x_i$

Table 15: The Capacitated Facility Location Model

The formulation in Table 15 can be directly coded 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];
WRITE costs; z;
END
```

A more straightforward formulation would be to use logic. What we want is to force x_e to be one if the plant at location *e* is built. The condition "the plant at location *e* is built" can be expressed by the Boolean term $\sum_{j=1}^{n} z_{ij} > 0$, since we can only ship something from location *e* if the plant is built.
The constraint "if the plant at location *e* is built then x_i = 1" can now be formulated as:

$$\sum_{j=1}^{n} z_{ij} > 0 \rightarrow x_i = 1 \text{ or as } x_i = 0 \rightarrow \sum_{j=1}^{n} z_{ij} = 0$$

In LPL, this condition can be implemented by writing the following proposition declaration:

```
VARIABLE x{i} BINARY  ~:  SUM{j} z<=0 ;
```

x is declared to be a 0–1 variable (a proposition) and the expression is the implied condition. The ~ is needed, otherwise the declaration would model the condition $x_i = 1 \rightarrow \sum_{j=1}^{n} z_{ij} = 0$, which is not exactly the same. The former constraint says that if a plant is not built then nothing can be shipped from it; the latter says, that if nothing is shipped from the plant, it is not built. We need the first condition. Hence, a second formulation in LPL is:

```
...
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
```

It is important to note that the parameter *M*, is no longer needed, since LPL calculates it from the upper bound of $\sum_{j=1}^{n} z_{ij}$.

We can still do better. It is not needed altogether to impose any logical constraint. What we want – as a modeler – is simply to say that: "if the plant is built, we have to consider additional fixed costs". Could we do better than formulate the model as follows?

```
...
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
```

The expression *f[i]*(SUM{j}z>0)* simply says that the costs *f[i]* must be added if *SUM{j}z>0* is true. We see that the declaration of the binary variable (the proposition) has gone altogether. Nothing is left, except this simple construct. Needless to say, that LPL – whichever of the three formulations the modeler chooses – *generates exactly the same model for every variant*.

(For purists in type-checking, one could give a slightly different formulation of the minimizing function:

```
MINIMIZE costs: SUM{i,j} g[i,j]*z[i,j] + SUM{i} IF(SUM{j}z>0,f[i]);
```

but this is only a minor modification.)

6 CONCLUSION

A procedure was presented to translate mixed constraints, which contain mathematical and logical operators, into mixed integer constraints. Such a procedure can be very useful, especially for large mathematical models which are enriched and extended by a few logical constraints. It is also helpful for symbolic manipulation of such constraints. However, there is certainly no claim here that such a translation procedure is practical in all circumstances. Generating a MIP model is often not the right way to attack a problem that can be formulated using logical and mathematical operators.

Furthermore, we do not claim either that the procedure presented in this paper is optimal in all practical cases. The reformulation (4), which generates a sharp model, is not integrated in the procedure, several special cases could be added, and many extensions could be incorporated.

Nevertheless, I think the procedure presented here is a good starting point for

further improvements. Fundamentally, this is not difficult, technically, that is from the implementation point of view, however, this is an entirely different matter.

(LPL and a complete documentation is free and available at the LPL site ftp://ftp-iiuf.unifr.ch/pub/lpl/.)

REFERENCES

BALAS E., [1979], Disjunctive Programming, in: Discrete Optimization II, ed. Hammer P.L., Johnson E.L., Korte B.J., North Holland, Amsterdam.

BREARLEY A.L. & MITRA G. & WILLIAMS H.P., [1975], Analysis of mathematical programming problems prior to applying the simplex algorithm, in: Mathematical Programming 8, p. 54–83.

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: [Mitra al. 1994]).

HÜRLIMANN T., [1998], LPL site on the Internet: LPL code, documentation, and model library: ftp://ftp-iiuf.unifr.ch/pub/lpl/.

HÜRLIMANN T., [1997], Reference Manual for the LPL Modeling Language, Working Paper, Version 4.25, November 1997, Institute of Informatics, University of Fribourg, (newest version is always at the LPL-site: ftp://ftp-iiuf.unifr.ch/pub/lpl/Manuals, file Manual.ps).

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.

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.

MEYER R.R., [1981], A Theoretical and Computational Comparison of Equivalent Mixed Integer Formulations, in: Naval Research Logistics Quarterly 28, p. 115–131.

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.

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., [1994], An alternative explanation of disjunctive formulations, EJOR, 72, p. 200–203.

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.

YEOM K., LEE J.K., [1996], Logical representation of integer programming models, in : Decision Suppoprt Systems, Vol. 18(3&4), November.