MODELING TOOLS

T. Hürlimann

Working Paper No. 200

Talk at the 2nd ISDSS Conference at the FAW Ulm
June 22-25, 1992

June 1992

INSTITUTE FOR AUTOMATION AND OPERATIONS RESEARCH

University of Fribourg

CH-1700 Fribourg / Switzerland

Bitnet: HURLIMANN@CFRUNI51

phone: (41) 37 21 95 60 fax: 37 21 96 70

Modeling Tools

Tony Hürlimann, Dr. lic. rer. pol.

Key-words: Model Building, Expert Systems.

Abstract:
During the last decades, Operations Research has concentrated considerable effort on designing techniques for solving linear and integer programming models. Now large and complex LP's can be solved. Another topic in decision support is model management: tools for model formulation, model transformation, model analysis, model documentation, and result generating.
We describe a modeling language, named **LPL,** (Linear Programming Language), and other modeling tools which may be used to build, modify, and document mathematical models. The LPL language has been successfully applied to generate automatically MPS input files and solution reports of large linear and mixed integer programs. The available LPL compiler translates LPL models to the input code of any LP/MIP solver, calls the solver automatically, reads the solution back to its internal representation, and writes user defined reports in form of tables. The system can also be used simply to manipulate multi-dimensional tables, graphs or logical expressions. An input generator can read data from different format. LPL is still under development at our Institute.

"One of the problems of management science models is that managers don't use them."
O'Leary D.E. (1983)

1. INTRODUCTION

The need for computer-based decision support systems is widely recognized. But although impressive paradigms to model the decision-process have been developed during the last four decades, they are not widely used by decision makers. *Operations Research* as well as related mathematical theories in the realm of *Combinatorics* and *Graph Theory* have concentrated considerable effort on designing techniques for solving mathematical programming models, especially linear and network models, which are by far the most used in practice. Research in this field, coupled with progress in computer and software technology, now allow us to solve large and complex models on a desktop computer, which was inconceivable only ten years ago. *Artificial Intelligence* has developed rule-based expert systems to represent and process knowledge and gave an important impetus to a variety of inference techniques and resolution schemes. Different constraint programming languages and term rewriting systems have been invented to represent the knowledge in a declarative manner.

At first glance, it seems that Operations Research and Artificial Intelligence do not have much in common. OR is a mathematical discipline and its realm is quantitative models, whereas AI treats symbolic knowledge and models. It has been shown recently, however, that there are many parallels between the logical inference techniques, which is of central importance in AI, and certain solution procedures for the IP (integer programming) problems [Hooker 1988], [Jeroslow 1989]. Wittgenstein already has noted that any formula in propositional logic can be identified with a list of truth assignment tuples to its atoms (which make the formula true) [Wittgenstein 1978, p.54, Satz 4.44]. This ordered tuples can be viewed as co-ordinates of the n-dimensional unit cube, a important structure in the theory of polytope and integer programming. But only recently have these ideas been applied to compare logical and IP models [Williams 1977]. Karp [1972] proved 1972 the NP-completeness of an IP program. Blair C.E., Jeroslow R., Lowe J.K. [1985] seem to be between the first who applied mathematical programming to solve inference problems. Some work has been undertaken to explore the quantitative approach of first-order logic [Jeroslow 1985]. McKinnon K. and Williams H.P. [1989] describe an implementation of an automatic translation of logical statements into IP constraints. It is, for example, well known by now, that the logical problem of resolution and extended resolution is closely related

to the concept of cutting planes and Chvátal cuts [Hooker 1988].
Why are these parallels important for decision support systems? Because many problems can be formulated partially as logical or partially as mathematical problems. To translate the corresponding part into the other formulation paradigms would allow to solve the problems with one or the other method. Furthermore, the inference problem and the IP problem are hard enough to justify different approaches! Interesting advances have been made in this research field. A simple example show the power behind a automatic translation of models: It has been shown that Horn clauses can be translated into a 0-1 IP program and processed with an *LP* solver. *To solve the LP relaxation of such a IP program is computational equivalent to infer a Horn clause*. In other words, the inference problem of Horn clauses in AI can be processed by solving a LP problem! This is especially interesting for a large knowledge base, since big LPs can be solved very quickly.

Another topic in decision support, however, was neglected until now: model management, which denotes the whole manipulation process of the model. Only recently, different modeling tools have been developed to support the decision process.

There are three main reasons that the methods from the two mentioned disciplines are not widely used in the practical decision process. The first reason is that much work is needed to get the model set-up and the data collected and compiled into the appropriate order to solve it with the right algorithm. Once created, the model must be tested, analyzed, modified, solved, translated into different forms, etc. Often too much time and effort must be invested to manipulate and modify the model, although the model - once formulated in an appropriate form - might be solved on a fly using different algorithms on the today's desktop computer. The limiting factor, therefore, to use knowledge-based systems for many practical decision problems are no longer the solution procedures, but the model management. Therefore, the modeling manipulation process itself could and should be supported by the computer.

There is a second reason why computer-based modeling is not widely used: Most (interesting) problems are at least NP-hard and, therefore, the solution might no be found in reasonable time. At least in the integer programming, important advances have been made to *reformulate* the model such that the model is easier to solve [Jeroslow 1989]. On the other hand, heuristic techniques such as simulated annealing [Aarts 1990], genetic algorithms [Goldberg 1989] and tabu search [Glover 1989] have been developed to find maybe good solutions to big and complex problems. A great

help would be computer-based tools, which allow the decision maker to translate his problem into different forms such that the appropriate solution procedure can be applied to that problem

A third reason is that the real world problem are often uncertain and vague due to lack of information, imprecise data, or formulations. The classical approach to these problems is using probability theory. It has been supplemented, however, with other paradigms such as fuzzy logic [Zimmermann 1991], the theory of evidence [Shafer 1976], and the theory of hints [Kohlas, Monney 1990]. We are not concerned by these problems here, but will concentrate on the first reason that model management itself has been neglected.

Model management means model creation, model modification, model transformation, model survey, model views, model documentation, model analyze, model result specification, etc. There are reasons why model management has been neglected by the Operations Research as well as by the Artificial Intelligence community. First of all, most efforts are still concentrated on solution techniques which have dramatically advanced since several years. Big LP models can now be solved in a fraction of time on a PC. Faster computer and sophisticated algorithms have been developed also for the inference problems. On the other side, the business of modeling and knowledge representation is a wide range of activities. Modeling is done by people with very different backgrounds in various contexts, and it is difficult to develop modeling tools which might be used by a wide range of people. Most modeler still develop and use their own ad hoc tools to manage their models.

There are important disadvantages in doing so. Models are difficult to maintain with a changing crew. Model transparency may suffer and portability to different environments are limited. Often model or parts of it could also be used in a other context, but reusability is almost impossible. Operations Research and Artificial Intelligence journals are full of articles describing a special implementation of a model and its environment. The cost to develop its own tools should not be underestimated. That's the main reason why a decision makers tend to use rules of thumb rather than the troublesome path of model building.

It would certainly be of great use, if decision makers dispose of some 'universally usable' modeling tools and methods to do their job. Such tools are not only needed, but they are also possible.

In this paper, I present briefly some 'main stream' tools such as matrix generators, spreadsheets, and database management systems, since they are widely used. I try to explain why these instruments are interesting but not sufficient and why we need a 'universally usable' modeling language which allow to represent a wide range of different models. There are mathematical models (linear, integer, or not-linear) which can be represented by a formal language close to the common mathematical notation. There are also logical models which can be represented by a syntax close the predicate logic as used by Prolog or some other constraint programming languages. To use a - yet to invent - common language would be a important step in modeling techniques. Then, I present a still relatively modest modeling language LPL (linear programming language) based on mathematical and logical notation. Actually, LPL is more or less restricted to linear mathematical models extended by some logical operators.

While a modeling language could certainly be used to provide the means of a unique representation scheme of models and their structures, it is not sufficient for a computer-based decision tool. The 'end-user' - or the modeler - should even not be concerned about the modeling language syntax. He should have others - maybe graphical - tools to manipulate the model and its data. Therefore, *view* is one of the key terms in modeling and knowledge representation: while the model may have a single (internal) representation, written in a common language, it may have many views. View is used here in the same sense as in the database terminology. A view is an abstract model of a portion of the conceptual database. The database scheme and their data are stored internally in a specified format. The user of the database may then declare views which show only a portion of the data or in a different format. Data are selected, manipulated, and formatted to concentrate the user to limited aspects of the data. The same facilities are use in model management. Mostly, the modeler is only interested in partial aspects of his model at the same time and he wants to see them in different styles (graphically, as tables, mathematically, etc.).

I'll close my talk by presenting some graphical tools which represent linear model structures, which are under development at our Institute.
Our goal is to create a integrated multi-view architecture modeling tool which supports a wide range of user to create, manipulate, and describe their models. There is still a long way to go and our goals are ambitious.

2. APPROACHES IN MODELING

Matrix Generator have long been the standard tool to create larger models. A matrix

generator is generally a data driven environment, which produces the Matrix by collecting the data from different tables or by generating a computer program (in FORTRAN) which will generate the matrix. But matrix generators have often been cumbersome to use and might, therefore, introduce errors which were difficult to detect. To write a matrix generator program is more a job of a programmer than of a modeler.

A more natural way to do model management actually - since they are in the main stream of commercial data processing - are *spreadsheets*. The matrix A, as well as the vector b and c or some sub-tables can be entered easily into spreadsheet tables. EXCEL III includes even a linear solver for small LPs. One of the main advantage is that spreadsheets not only allow to store numeric and string constants but also formula. Any cell, or whole blocks of cell, can be specified by formulas. Model results produced by a solver can be written back to the spreadsheets and all kinds of pre- or post-calculations are possible. Although the operations with different spreadsheet tables at the same time can be quite complicated, spreadsheets are widely and increasingly used to manipulate models. An important disadvantage is that spreadsheets are not generic - one cannot build indexed objects. Each object must be entered explicitly into a cell. Tables may not be 'blown up' by simple instructions. With large models - they are more common in practice than smaller ones - the modeler would be in trouble.

Another approach to model management is to use the now popular *relational databases*. Typical tasks in model management are sub-table selection, table join or some other tables manipulations. Relational database systems are just designed to do that kind of operations very well. An important advantage of this approach is that the user (the modeler) must only design the structure of his tables (the scheme) by a database definition language and fill them up with the corresponding tables. Any manipulation can be done by a powerful database manipulation language. This approach is also very flexible and transparent: the table contents as well as the table structure may be modified any time. All we need to do is to define our data-tables, to fill them up with the right data, and to write some code indicating how the tables has to be linked together to produce a whole instance of our problem. Once solved, the solution can be write back to the database and the solution reports come almost for free. (A example was given in [Hürlimann 1991]). Of course, one needs to know a database language. But they are now very common and can be used in many application fields. Using this kind of approach for model management is very promising.

The database approach might be so convincing that it is hard to see why modeler need still other, more specialized and appropriate tools for model management. There are many reasons. Non-linear or logical (rule based) models are more difficult to be defined and manipulated by the database approach. Another reason is that the model *structure* is often lost in the database tables or within the code. But the model structure is an important outstanding part of most models, that the question arise how to formulate the structure as a separated standing-alone part of the model. The data of a large model should clearly be stored in a database, but I doubt whether the model structure can be stored within a database system in a transparent manner. For every model new code to manipulate the data must be written. The maintenance of this code must be guaranteed.

A third approach in modeling is to design a *modeling language*, which can handle a wide range of different model types. The main advantage of this approach is that the whole model (structure) is in a unique, concise, and readable format - readable by the modeler *and* the machine- and can be processed from that form. The language should be declarative and probably close to some mathematical and first-order logic syntax. Models are created and used by people with different cognitive skills. Many people, for example, use the mathematical language to build their models. It would be natural for them to have a language close to the mathematical notation to create the model.
This approach is the focus of this paper, and will be exposed in the next sections.

The main advantage of a 'universal' modeling language is that it is close to a common notation and known to many modeler in different domains. The expense to learn such a language is limited. Another benefit of this approach is that it is extensible. Non-linear models, logical models, and even fuzzy models [Zimmermann 1991] can be integrated by adding more features to the modeling language. A third and often overlooked advantage of a common modeling language for logical and mathematical models is that logical models such as knowledge bases can be (automatically) translated to integer programs or vice versa to exploit efficient solvers and algorithms. It is also well known that different algebraic representations of the same integer program may behave very differently in computation. While one formulation might be intractable, the other might be easy to solve. So, while the modeler formulates his model, the modeling management system could translate the model to an easier form. It is obvious that a common modeling language facilitates such translations. A forth advantage is that a model - stored in the form of text stream (say language) - needs only a parser to translate the model into different *views* and can be easily ported to other environments. This could be compared with a page description language like

PostScript which allows to store a page in a text stream which can be accepted by different environments.

3. FEATURES OF A (HYPOTHETICAL) MODELING LANGUAGE

A universally usable modeling language should have at least the following features

1 representing a wide range of different models (mathematical and logical)
2 representing the model structure separated from the data
3 representing model-parts as objects (modules)
4 reading data from a wide range of format (input generator)
5 writing data in various formats (report generator)
6 manipulating data in a similar manner as databases do
7 verifying the model and its data with different tools
8 reformulating the models in a semi-automatic manner
9 linking the model to a variety of solvers and inference machines

The modeling language having these features is still to be invented. Different modeling languages in the realm of mathematical modeling have been implemented recently. AMPL [Fourer, Gay, and Kernighan 1990], GAMS [Brooke, Kendrick, and Meeraus 1988], LINGO [Cunningham, Schrage 1989], and SML [Geoffrion 1989] are closest to the LPL language [Hürlimann 1992]. For logical model, the best known language is Prolog. Although most people will say that Prolog is a programming language, one may argue that it is also a modeling language, since it represents the problem in a declarative way, like the mentioned mathematical modeling languages. There exist many parallels between mathematical modeling languages and logic constraint languages, like there exist many parallels between OR and AI. The hypothetical modeling language(s) will be probably something between the mathematical and logic constraint languages. It is notable that Prolog III [Colmenauer 1987] contains an integrated Simplex algorithm - not to replace its inference engine, but to solve linear systems! The two research domains are coming closer together.

The nine features are not exhaustive, but certainly necessary to define a modeling language. Let's go through these points briefly.

Representing a wide range of different models is essential. The mathematical-logical notation is especially useful, because it allows to formulate a wide variety of problems. The notation can even be extended to integrate new approaches in logic (default reasoning), or to express uncertainty in many forms (fuzzy logic). To represent large models, the indexing capability is elementary.

A point often overlooked is the difference between *model structure* and *model*

instance. A model structure defines a problem type. It can describe a infinity of single problems, whereas an instance defines a single model. We may adopt the definition: *model structure* + *data* = *model instance*. One of the main disadvantage of Prolog is that it amalgamate the model structure with the data. *Conceptually* at least, the structure should clearly be separated from the data, which does not mean that we should have two languages: one for the structure and one for the data. A single language should be capable to manipulate both. This separation is essential if the modeler works with big models containing a lot of data. In DBMS (database management systems), there are often two different languages: the data definition language (DDL) to manipulate the scheme (the structure) and the data manipulation language (DML) to manipulate the data stored in the databases.

A modeling language should also be capable to encapsulate model parts, which we may call *modules*. This joins the idea of an object oriented modeling language. Some research projects are already on its way in this domain, but it is too early to see concrete results.

Reading data from databases or from some weakly structured data formats is an important task. High-level as well as low-level reading procedures should be integrated within every modeling language. This procedure may be called Input Generator.

The inverse procedure is to write data back to databases or to user-formatted tables. This is better known as Report Generator.

We take for granted that a modeling language should be able to manipulate the model entities in a convenient and concise way. Tables and other multi-dimensional structures should be manageable in a single instruction. *Instant evaluation* of any expression should be possible at any time. In this sense, a modeling language has much in common with a database management system or some other programming languages such as APL or SETL [Schwartz et al., 1986].

Checking and verifying the structure as well the data should be an integrated part of a modeling language. There are different part which should be tested automatically: range checking, unit checking and others. Furthermore, user defined checks and verifications are a useful tool for the modeler.

A recent research field is the reformulation techniques of the models. As is generally known by now, a logical model can be translated into a IP model to be solved by a branch and bound heuristic; or a MIP model can be translated into another MIP problem, which is - hopefully - easier to solve; or an (special) LP model can be transform into a (bigger) transportation model [Williams 1991]. Generally, however, it does not make much sense to translate a model *mechanically* into another form without the intervention of the user. Therefore, the modeling language must be

equipped with features apt to indicate manually how the transformation is to be achieved (see section 4 for some concrete examples).
There is a wide range of different solvers for special models, and it is unlikely that a solver will ever be found which can solve every mathematical or logical model in a reasonable time. Thousands of methods have been implemented for special models. It is, therefore, essential that a modeling language is flexible enough to be linked with a variety of solvers. This is another limitation of Prolog. Prolog - at least in its present form - will not be used in practical decision support, because its integrated inference engine is not - cannot be - powerful enough to solve any problem in a reasonable time. It is fundamental for every modeling language to have an open interface to external callable solver or inference machines to process a wide range of models.

The question may arise, why a universal modeling language should be designed, since different models require different solvers anyway. Would it no be more pragmatic to develop several modeling languages for the different model paradigms. The answer is that a small modification within the model structure or the data makes it necessary to use a different solver to process the model. It would be much more practical to have a unique representation scheme for a wide range of models and to interface this with as many solvers as possible, than to have many languages for which the model must be newly designed. There is another point: In fact, we can formulate much more complex problems using a declarative language than we can solve, that's true. But sometimes 'the' solution of a model is not important to the decision maker. What he or she often wants is a 'good' or acceptable solution. The representation has some legitimacy by itself. Simply to organize or order the ideas and represent them in a unique scheme may 'solve' the problem of the decision maker. Furthermore, a unique modeling language encourages and favours the production of standard *viewing* tools.

The different features of a hypothetical modeling language have now been briefly commented. The next section give some examples of feature 8 (reformulating models) in an informal way; it highlights some new aspects of modeling, not found in the modeling research community.

4. MODEL TRANSFORMATION

Model reformulation can concern the whole model or only a small part. Some languages already integrate such facilities, such as GAMS or LPL which change a sub-expression *x+x* into *2*x*. within a linear constraint automatically. But such modifications are more of cosmetic nature.
A less trivial modification is the following example. Suppose the language has a

function *max()* with a variable list of parameters and returns the largest of them. So the expression $\max_{i=1}^{n}(x_i)$ will return the largest x_i with $i = \{1\text{K } n\}$. In a numerical expression (where all x_i are known), there is no need to modify anything. But suppose all x_i are model variables which are assigned under the control of a (external) solver, than this single expression $\max_{i=1}^{n}(x_i)$ makes the whole model non-linear! There is a formulation technique, however, to transform this sub-expression into a form which makes the model linear, such that the model becomes easier to be solved. The procedure is as following:

1. replace the expression $\max_{i=1}^{n}(x_i)$ by a new model variable, say z
2. add n linear constraints $x_i \le z$ where $i = \{1\text{K } n\}$ to the model

Maybe the modeler wants to do this transformation of his model, maybe not. But the modeling language should offer some facilities to do the job if required.

Another example is the fixed charge problem. Fixed charge problems arise if the existence of some unknown is dependent on other unknowns. Suppose the cost C is dependent of the fact whether a product is manufactured at all, such that

if $x = 0$ then $cost = 0$
if $x > 0$ then $cost = C$

A natural way to formulate such a situation would be to write: $cost = \text{if}(x > 0, c, 0)$, (supposing x is non-negative) where *if()* is a function returning the second or third argument depending whether the first argument is true or false. Here again, there is no need to reformulate this expression if no variable occurs within the first argument. But if there *is* a model variable within the first argument, then the model must be reformulated to be linear. To handle such an expression using a MIP solver, we need to write this expression in the following form as two linear restrictions: $cost = C\delta,\ x - M\delta \le 0$, where δ is a 0-1 variable, and M is an upper bound on x.

The same techniques may be used to reformulated whole models. As an example, take a simple model formulated in the propositional logic. (example is from [Hooker 1988]).

$\neg p_1 \rightarrow p_2 \vee p_3$
$\neg p_1 \vee ((p_2 \vee \neg p_4) \wedge p_3 \wedge (\neg p_3 \vee p_4))$
$\neg(\neg p_1 \wedge p_3)$
PROVE: p_2

(where $p_1, p_2, p_3, p_4 \in \{false, true\}$ are propositions)

This is a typical inference problem: given the first three formulas, prove p_2. First we translate these formulas into a list of clauses to get the conjunctive normal form. This can be done by applying the following transformation rules repeatedly

$(p \rightarrow q) \leftrightarrow (\neg p \vee q)$
$\neg(p \vee q) \leftrightarrow \neg p \wedge \neg q$
$\neg(p \wedge q) \leftrightarrow \neg p \vee \neg q$

to get the form containing the six clauses

$p_1 \vee p_2 \vee p_3$
$\neg p_1 \vee p_2 \vee \neg p_4$
$\neg p_1 \vee p_3$
$\neg p_1 \vee \neg p_3 \vee p_4$
$p_1 \vee \neg p_3$
PROVE: p_2

It is easy now to formulate a IP problem from which we can decide whether p_2 is deducible from the clauses or not.

The corresponding IP model is (where $x_1, x_2, x_3, x_4 \in \{0,1\}$)

$x_1 + x_2 + x_3 \geq 1$
$-x_1 + x_2 - x_4 \geq 1 - 2$
$-x_1 + x_3 \geq 1 - 1$
$-x_1 - x_3 + x_4 \geq 1 - 2$
$x_1 - x_3 \geq 1 - 1$
MINIMIZE: x_2

If $x_2 \geq 1$ then p_2 is proven, otherwise it cannot be inferred from the clauses. The translation rules are simple:

1. replace every atom p_i by a new 0-1 variable x_i.
2. replace every $\vee$ (or operator) by a + (addition sign)
3. replace every $\neg$ by 1- (such that $\neg p_i$ becomes $(1 - x_i)$
4. add ≥ 1 at the right of every formula

Different translation methods exist. Supporting such transformations by a semi-automated procedure might be a useful method for the necessary model reformulations.

5. LPL: A MODELING LANGUAGE FOR LINEAR MODELS

LPL (Linear Programming Language) is a modeling language which supports various stages in modeling: preparing and manipulating the data, formulating the model structure, calling the external solver, and printing the user-defined solution reports. Actually, LPL supports only linear models, but its extension to non-linear or logical models is straightforward. The main difficulty with non-linear models is, how to link the model with a (or several) solver(s), since there does not exist any standard representation of a model instance. For logical models, the syntax must even not be extended, since AND, OR, NOT and other operators make part of LPL already. To summarize, the main features of LPL are:

- a simple syntax of models with indexed expressions close to the mathematical notation, and directly applicable for documentation
- 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 and other output
- open interface to most LP/MIP solver packages
- support goal, multi-stage, multi-objective programming as well as data checking.

By the means of a simple portfolio model [Shapiro 1988], some concepts of the modeling language LPL are presented. Figure 5-1 displays the model structure in an algebraic form. This formulation can be translated directly to an LPL model formulation as shown in Figure 5-2.

The algebraic formulation of Figure 5-1 contains different sections: Index-sets, numerical data-tables, variables (the unknowns), a minimization function, and different constraints. Note that this formulation in Figure 5-1 represents only a model *structure* not a specific, *instantiated* model, since no data are defined. To produce a specific model, it must be supplemented by the values of all index-sets and data-tables.

Indices

j	bonds $(j = 1,\text{K}\ , N)$
t	time periods $(t = 1,\ldots., T)$ with $T \le 50$

Data Tables

c_j	current market price for bond j
f_{jt}	cash flow produced by bond j at the end of period t
L_t	cash requirement at the end of period t
q_j	conditional minimum purchase quantity of bond j
Q_j	maximum allowable purchase of bond j
a_t	re-investment rate for period t

Variables

x_j	quantity of bond j to be purchased here and now
s_t	cash surplus to be accumulated at the end of period t
d_j	is 1, if bond j is selected for portfolio, else it is 0

Minimize

$$\sum_{j=1}^{N} c_j x_j + s_0$$

subject to

$$\sum_{j=1}^{N} f_{jt}x_j + a_t s_{t-1} - s_t = L_t \quad \text{for} \quad t = 2,\text{K}\ ,T$$

$$x_j - q_j d_j \geq 0 \quad \text{for} \quad j = 1,\text{K}\ , N$$

$$x_j - Q_j d_j \leq 0 \quad \text{for} \quad j = 1,\ldots, N$$

$$x_j \geq 0, s_t \geq 0 \quad \text{and} \quad d_j \in \{0,1\}$$

Figure 5-1

Figure 5-2 shows the entire model structure in LPL syntax. The different sections of index-sets, data, variables, minimizing function, and constraints are headed by the reserved words SET, COEF, VAR, MINIMIZE, and MODEL. Units can be defined using the UNIT statement. Comments can be added anywhere between quotes or {...}. Comments surrounded by the curly brackets can be used to document a model, but they do not belong to the formal part of the model. Quoted comments, on the other side, are part of the formal documentation used in the reports.

Since an LPL model can be processed directly by the LPL compiler, its formulation has some particularities compared to the algebraic notation: The sigma sign ∑ is replaced by the reserved word SUM; subscript indices are listed between parentheses; a semicolon must end every declaration. It is also possible to use multi-letter names in place of single-letter names. Hence, *c(j)* might be replaced by *marketprice(bond)*.

```
PROGRAM portfolio; { syntax in LPL } {$C}
SET   {Indices}
  j;                    " list of bonds "
  t;                    " time periods "

UNIT  {measurement units}
  dollar;               " unit of cash "
  percent = 1/100;      " unit of percent (%) "
  pieces;
  unitPrix = dollar/pieces;

COEF  {data tables}
  c(j) UNIT 100*unitPrix; " current market price for bond j (in $100) "
  f(j,t) UNIT unitPrix;   " cash flow produced by bond j in period t "
  q(j) UNIT pieces;       " conditional minimum purchase quantity of bond
j "
  Q(j) UNIT pieces;       " maximum allowable purchase of bond j "
  a(t) UNIT percent;      " reinvestment rate for period t "
  L(t) UNIT dollar;       " cash available in period t "

VAR   {variables}
  x(j) UNIT pieces;       " quantity of bond purchased "
  s(t) UNIT dollar;       " cash surplus "
  d(j) BINARY;            " =1 if bond is selected, else =0 "

MODEL  {constraints}
  invest UNIT dollar: SUM(j) c*x + s[1];
  Cash_bal(t|t>1) UNIT dollar: SUM(j) f*x + a*s[t-1] - s = L;
  C(j) UNIT pieces: x - q*d >= 0[pieces];
  D(j) UNIT pieces: x - Q*d <= 0[pieces];
  initS UNIT dollar: s[1] = 0[dollar];

{$I model.dat }          {read the model data from file MODEL.DAT}
```

```
MINIMIZE invest;          {call the solver, and read the solution}
PRINT invest; x; s; d;    {print the solution tables to a file}
END                     {--end of model formulation}
```

Figure 5-2

Four additional instructions are found in the LPL model in Figure 5-2:

- *{$I...* reads the model data from file MODEL.DAT written also in LPL syntax (Figure 5-3),
- *MINIMIZE...* calls the solver and reads the solution back to the LPL internal representation,
- *PRINT...* prints the required tables to the output file,
- the *UNIT* statements used within the model.

A specific data set for this model is shown in Figure 5-3 written in LPL syntax. The data-tables can also be in plain text. LPL includes a flexible Input Generator to read external data definition.

```
{ The file MODEL.DAT is }
SET
  j         |   c |  q |  Q  = /
       {--------------------}
        A1 |   2 | 10 | 500
        A2 | 2.3 |  . | 700
        A3 |   4 |  . | 700
        A4 |   1 | 15 | 800
        A5 | 2.4 | 20 | 900   /;
COEF
  TMAX INTEGER [0,100] = 50;  { a data not declared up to here }
  L UNIT 1000*dollar;         { redefine the unit (L is measured in $
1000) }
SET
  t         |    L  |   a = /
       {-------------------}
        T2 |    12 |  90
        T3 |    14 |  80
        T4 |     5 |  80   /;

COEF
  f =
       |   T2   T3   T4
           A1      4    4    4
           A2    5.5    5    4
           A3      5    3    6
           A4      6    6    .
           A5      4    5    6 | ;

"some data consistency tests"
CHECK This(t): q < Q ;     "every q must be smaller than Q"
CHECK This: #t <= TMAX;    "the cardinality of set t must be smaller than
TMAX"
```

Figure 5-3

The different model section are now briefly commented

5.1 INDEX-SETS

An index-set is an unordered or ordered collection of objects. They are called *elements*. The set of bonds *j* in our example is such a set. A set is declared in LPL as

```
SET j;  "bonds"
```

The elements itself of this set are not defined at this place, they are part of the model data defined in file MODEL.DAT (Figure 5-3). The modeler is, however, free to declare a set and to assign its elements at the same place within the model. The set *j* could have been defined as

```
SET j = /A1:A5/;  "j is a set of five bonds"
```

The two elements *A1* and *A5* separated by a colon define a range of elements, which is the same as

```
SET j = /A1 A2 A3 A4 A5/;
```

where every element is mentioned explicitly. The element-names *A1,...,A5* might also be replaced by more meaningful names as in

```
SET j = /World_bank88 Treasury_bills Sandoz_baby EFF ATT87/;
```

Index-sets may be indexed, in which case they define a tuple-list of its indices. If the sets *p* and *t* are defined in the following way

```
SET p = / P1:P180 /; "a list of 180 players"
SET t = / T1:T15 /;  "a list of 15 teams"
```

then an indexed set *Must_be_in* can be specified as a list of three elements

```
SET Must_be_in(p,t)  = /  P1 T2 ,  P2 T6 , P3 T7 /;
```

This tuple-list defines a relation *Must_be_in* between the players *p* and the teams *t*. It tells, that player *P1* belongs to team *T2*, player *P2* to team *T6*, and player *P3* to team *T7*. It is important to note, that this relation assigns only a sparse sublist of the whole (Cartesian) tuplelist. Another example is to define sublists of players as

```
SET pla1(p) = / P1 P45 P56 P67 P78 P122 /;
SET pla2(p) = / P2 P67 P123 P145 P12 P178 /;
```

Indexed sets may also be assigned by logical expressions. The union, the intersection and the difference of set *pla1* and *pla2* may be defined as

```
SET union(p)  = pla1 or pla2;
SET intersection(p) = pla1 and pla2;
SET difference(p) = pla1 and not pla2;
```

Index-set is one of the most important components in any large-scale LP model. LPL offers a broad variety of different set types.

5.2 NUMERICAL DATA

Numerical data within the model are collected in *coefficients*. Its declaration is headed by the reserved word COEF. The simplest coefficient consists of only one value:

```
COEF TMAX  INTEGER  [0,100];
```

TMAX is declared as a coefficient, but its value is not yet known. The declaration,

however, tells that TMAX must be an integer value within the range [0,100]. The LPL compiler tests these conditions and reports an error, if they are violated. LPL offers also the CHECK statement which can check the data consistency using more complex conditions. The instruction

```
CHECK This(j): q < Q ;     "q must be smaller than Q for every j",
```

checks for every *q* over *j*, if *q* is smaller than *Q*.

Coefficients can also be indexed, and their values are collected in tables. *c(j)* is such a coefficient in our example. It declares a marketprice *c* for every bond *j*. The conditional minimum purchase quantity *q* and the allowable purchase *Q* are also indexed over *j*. (Note that LPL does not distinguish lower and upper-case letter by default, but the directive *{$C}* within a model instructs the compiler to do so, therefore, *q* and *Q* are distinct in this model).

The first table of Figure 5-3 is a convenient way to define the set *j* together with the three coefficients *c*, *q*, and *Q*. This is possible, because all three coefficients run over the index *j* only. This is, however, not the only way to enter the data. The modeler is free to declare and define the data within the model structure as

```
SET  j = /A1:A5/;
COEF c(j) = [ 200  230  400  100  240 ];
COEF q(j) = [  10    .   20   15   20 ];
COEF Q(j) = [  50   70   70   80   90 ];
```

LPL offers different table formats for the data. But the most flexible way to get the data from external formats is to use the Input Generator.

Indexed coefficients are not restricted to only one-dimensional (with only one index) items. They can be two- three- or higher-dimensional. *f(j,t)*, for example, declares a two-dimensional coefficient, where the cash flow *f* is defined for every bond *j* within every time period *t*. Data tables are reassignable.

```
COEF a = 10;  "10 is assigned to the coefficient a"
COEF b = a;   "the value of a is copied to b (10)"
COEF a = 20;  "a gets a new value 20, the old one is erased, but b is still 10"
```

5.3 VARIABLES

Variables have the same properties as coefficients. They can also be defined as multi-dimensional objects and numerical values can be assigned to them. The only differences are, that their declarations are headed by the reserved word VAR and their values are usually assigned under the solver's control. A typical variable declaration is

```
VAR  x(j);   "purchased quantity x of bond j".
```

The variable *x* is declared over *j* and may be interpreted as a numerical one-dimensional table of unknown values. But the modeler may also assign values to them. It is also possible to restrict the values of the variables. Lower and upper bounds on variables are often used. Sometimes variables must be integers. These options can be added to any variable declaration as in

```
VAR d(j) INTEGER;
```

The declaration of *d* declares a variable over set *j*. If the reserved word INTEGER is

replaced by BOOLEAN or LOGICAL, its values can only be the integers 0 or 1. These restrictions are automatically translated by the LPL compiler as BOUNDS and INT-MARKERS in the BOUND- and COLUMN-Section of the MPS input-code for the LP/MIP solver.

5.4 THE MODEL

The model restrictions are declared in the MODEL statement. Each restriction begins with a name. The optimizing function can be included within the list. The restriction name is followed by a colon and a linear expression. Restrictions can also be indexed. The restriction

```
MODEL B(j): x[j] - q[j]*d[j]  >=  0;
```

is defined for every element of the set *j*. This generates as many single restrictions as *j* has elements. Any algebraic expression is allowed as long as the expression is linear in the variables. Summations begin with the reserved word SUM. The term

```
... SUM(j) c[j]*x[j] ...
```

sums the product *c*x* over the set *j*. This is close to the mathematical notation. The indices of *c[j]* and *x[j]* can also be dropped, if no ambiguities arise from the expression. This simplifies the term to

```
... SUM(j) c*x ...
```

The declaration of restrictions can also be separated from their definition. The declaration

```
MODEL
  Invest;            "total bond purchase and initial cash investment"
  Balance(t | t>1);  "cash balance equation in all periods t>1"
  C(j); D(j);        "logical conditions"
```

is perfectly correct. The assignment of the restriction may take place in a different model part. The reserved word MODEL can also be replaced by the reserved word EQUATION. But only the restrictions collected within the MODEL statement make part of a model that is passed to the solver. This makes it possible - using the solver statement (MINIMIZE or MAXIMIZE) at several places within the model - to call the solver repeatedly within a LPL model with different model variants. Another useful application of this option is multi-goal programming.

The objective function begins with the reserved word MINIMIZE or MAXIMIZE, depending whether the function is minimized or maximized. This instruction calls directly the solver.

5.5 THE SOLVER

LPL has no integrated solver, but can call an external solver automatically. The interface between most available LP/MIP solvers and LPL can even be specified by the modeler. The command *MINIMIZE* or *MAXIMIZE* instructs the LPL compiler to produce the MPS file, the standard solver input file, then to call the solver itself with the right parameters, and to read the solution back to the LPL internal data structure.

The interface is explained in detail in the reference manual [Hürlimann 1992]. By default, the interface to the XA solver [SUNSET 1990] is 'hardwired' within the LPL compiler, but other solver packages work too.

5.6 REPORTS

LPL does not only allow to formulate a complete model, but can also produce model reports. This is an integrated part of LPL. The reserved word PRINT is used to generate even most complex reports. The simplest reports are generated using the syntax shown in Figure 5-2 as

```
PRINT Invest; x; s; d;
```

which prints four tables in a predefined format. Reports can be much more complex. The user can specify the output format by the means of a mask.

5.7 UNITS

Every declaration of numerical entities can be extended by indicating the units. This increases the reliability and the readability. Furthermore, explicit declaration of units gives the compiler additional checking power that may reduce the number of syntax errors. Every expression is checked of unit commensurateness, before it is evaluated. Automatic unit conversion takes place without the intervention of the user.

Units must be declared using the Unit Statement beginning with the keyword UNIT. *Basic units* (as 'meter') are just declared by its name. *Derived units* (as 'speed=meter/sec') are declared by its unit expression

```
UNIT
  meter;              " a basic length measure"
  kilo = 1000;        " a derived unit commensurable to type number "
  km = kilo*meter;    " a derived measure, commensurable to 'meter'"
  cm = m/100;         " a derived measure, also commensurable to 'meter'"
  speed = meter/sec; " another derived unit, not commensurable to 'meter' "
```

The use of units within the model structure as well as within the data tables (defined in LPL) is simple: just extend the declaration with the reserved word UNIT and add a unit expression. A number within an expression must be extended by *[<unit expression>]*. The Report Generator too accepts units. The table *invest* might be written in 1000$ instead of $ to the output device. To do this one may add the unit expression as following

```
PRINT invest UNIT 1000*dollar;
```

But note that units are optional. The modeler may or may not use them within his model.

6. GRAPHICAL TOOLS

LPN (Linear Programming Network) is a graphical representation of a LP model, which can be build automatically from a mathematical representation such as LPL. Each (indexed) variable is represented by a square and each (indexed) restriction as a

circle. Squares can be connected to circles, but squares or circles cannot be connected to each other by edges. The resulting graph is a bipartite graph [Hürlimann 1987]. A circle is connected to a square, if and only if the variable shows up in the restriction. If the variables are not indexed, the graph corresponds to the fundamental graph of Greenberg [1981]. A similar graph was used by Egli [1980] to document the EP-model, a multi-period, agricultural model, which is a major tool for agricultural policy in Switzerland. LPN is a very convenient tool for moving within the model and for analyzing the model structure. Sub-structures such as netforms or flow structures can be revealed quickly. Interesting operations such as filtering and dis-aggregating (unfolding the indexed objects!) can be implemented. Appendix A shows the EGYPT model in an LPN representation, which has been produced automatically from the LPL representation.
Another graphical representation is the *Genus Graph (GG)* defined by Geoffrion [1987]. The GG shows a 'bottom-up' structure of the whole model and can be - like the LPN graph - produced automatically from a mathematical presentation . The same operations such as filtering and dis-aggregating are possible.

A *block structure* (see Appendix B for model EGYPT) is still another representation of the model structure. The whole model structure is exposed as a condensed matrix block. The PAM modeling environment is entirely based on this approach [KETRON 1986].

All three mentioned representations are tools which reveal different aspects of a *model structure*, independently of a specified *model instance*. The same tools may be used to show a model instance by dis-aggregating the whole model. Dis-aggregating means to unfold the model using some or all index-sets. The corresponding representation, if all index-sets are unfolded, are the *fundamental graph* (defined by Greenberg [1981]), the *element graph* (defined by Geoffrion [1987]), and the *matrix picture*, which shows the whole matrix on textual or on bitmap level (see Appendix B for a bitmapped picture of the EGYPT model).
The power of these tools comes from the idea of (partially) unfolding. To 'visit' the model, one may use first the folded model, then unfold some specific model parts to see the interested details. 'Visiting' and querying the model is one aspect, modifying the other. All mentioned tools can also be used to edit the model. A research group at our Institute is developing a tool which allow this manipulation. So to be concrete: We create modeling tools which allow the translation LPL --> LPN, but also the translation LPN --> LPL. The modeler may edit the model in LPL form using a word processor or in LPN using a graphical editor or some other forms and the translation

to the other forms is done automatically by the - yet to implement - model manager.

APPENDIX A

LP-Model EGYPT: a specific instance has 145 rows, 350 columns, 1268 non-zeroes, matrix density: 2.5%

This is a one-period model of the Egyptian fertilizer industry developed at the World Bank (Choksi, Meeraus, and Stoutjesdijk 1980). Fertilizer is produced at different plants or imported, and consumed in the different regions of the country. The amount consumed in each region of all final products (the fertilizer) is given. The production uses several raw materials and intermediate products bought domestically or aboard. The total utilization by the processes cannot exceed a certain capacity at any plant. The production is also limited by the given input-output coefficients of the processes. Transport costs, distances, and the prices of the commodities (raw materials, intermediate products) are known. What is the level of processes at each plant; what is the shipped amount of all commodities between the plants and the regions; how much fertilizer must be imported, if we want to minimize the overall purchase and transportation costs?

This model is a relative simple model which combines a production, a transportation, and a consumption model of a whole industry. First, we give the entire formulation of the model in LPL form. A data set and model results are added.

The LPL representation of the whole model structure is shown in figure A-1 (The declaration of the sets and the data are omitted).

```
VAR
  Z(plant,process | p_pos);              "level of possible process at plant"
  Xf(c_final,plant,region | cp_pos);     "amount of final product shipped from plant to
region"
  Xi(c_ship,i=plant,j=plant | cp_pos(,i) AND cc_pos(,j)); "amount of intermed. shipped
between plants"
  Vf(c_final,region,port);               "amount of the final product imported by a region
from a port"
  Vr(c_raw,plant | cc_pos AND p_imp>0); "amount of raw material imported use at a plant"
  U(c_raw,plant | cc_pos AND p_dom>0);  "amount of raw material at a plant purchased
domestically"
  Psip;                                  "domestic recurrent cost"
  Psil;                                  "transport cost"
  Psii;                                  "import cost"

MODEL
  MINIMIZE Psi:  Psip + Psil + Psii;  "minimize the overall costs"
  mbd(nutr,region):                   "total nutrients supplied to a region for each final
product"
           sum(c_final) fn*(sum(port) Vf + sum(plant) Xf) >= sum(c_final) fn*cf75;
  mbdb(c_final,region | cf75>0):      "total of each final product supplied to a region"
           sum(port) Vf + sum(plant) Xf  >= cf75;
  mb(c=commod,pl=plant):              "production and consumption at a plant, plus the inter-
plant
                                       shipments, plus imports and domestic purchases must
exceed
                                       the total shipped for final products for each
commodity"
           sum(process) io*Z + sum(p2=plant) Xi[c,pl,p2] + sum(p3=plant) Xi[c,p3,pl]
           + Vr + U  >=  sum(region) Xf;
  cc(plant,unit | m_pos):           "total utilization by all processes at each plant and for
each unit"
         sum(process) util*Z  <=  util_pct*icap;
  ap:  Psip = sum(c_raw,plant) p_dom*U;
```

```
  al:  Psil = sum(c_final,plant,region)  tran_final*Xf + sum(c_final,port,region)
tran_import*Vf
             + sum(c_ship,pl=plant,p2=plant) tran_inter*Xi[,pl,p2] + sum(c_raw,plant)
tran_raw*Vr;
  ai:  Psii/exch = sum(c_final,region,port)  p_imp*Vf + sum(c_raw,plant) p_imp*Vr;
```

Figure A-1

APPENDIX B

Two aggregated LPN representation of the EGYPT model can be seen in the graphs of figure B-1 and B-2. The variables show up as squares and the restrictions as circles.

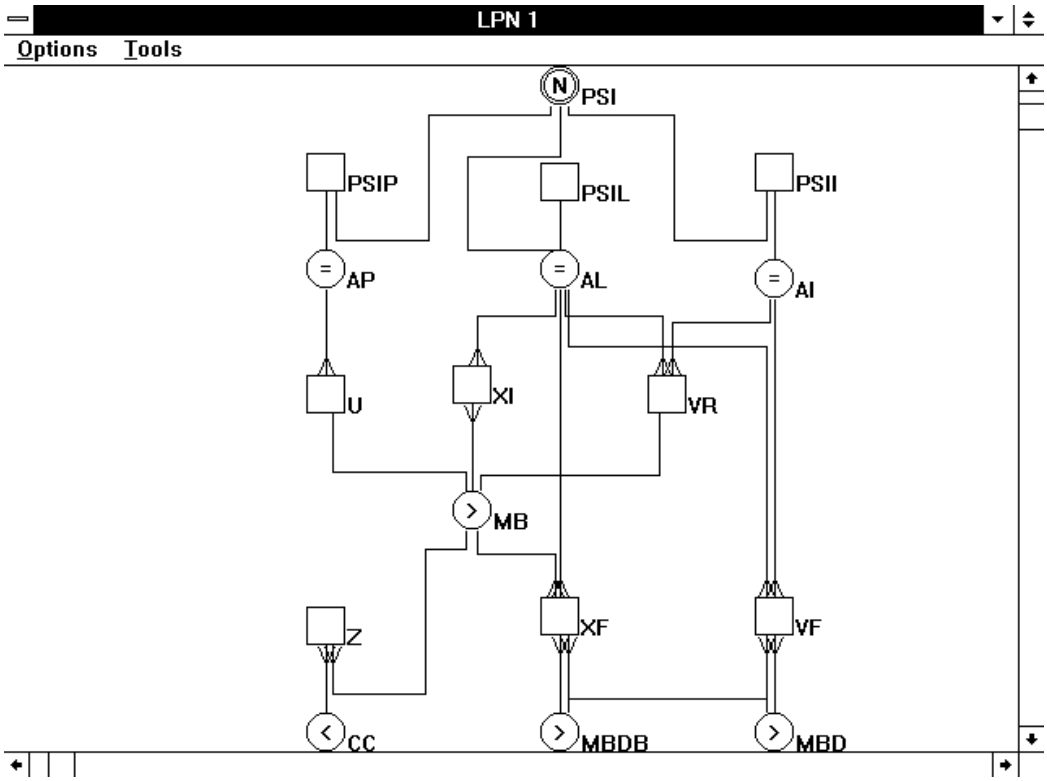

Figure B-1

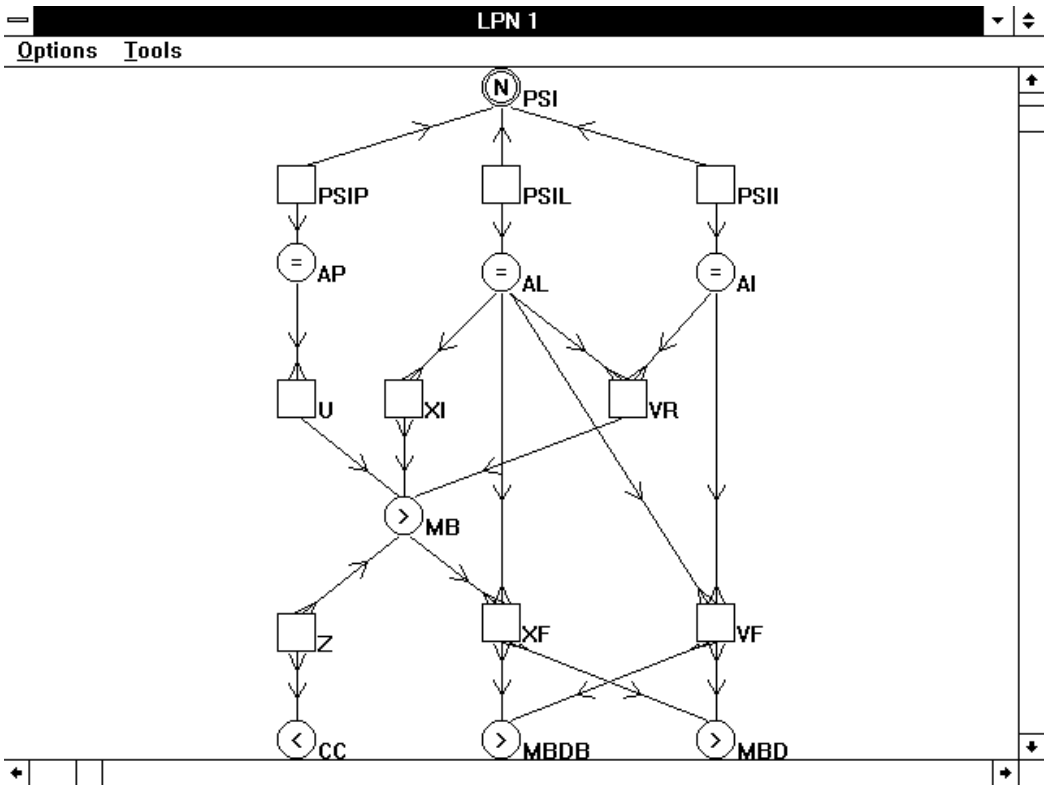

Figure B-2

APPENDIX C

A condensed block representation of the EGYPT model is shown in figure C-1.

	Z	Xf	Xi	Vf	Vr	U	Psip	Psil	Psii		RHS
mbd		1		fn						>	fn*cf
mbdb		1		1						>	cf75
mb	io	-1	1	1		1				>	0
cc	util									<	util*
ap					-p_dom		1			=	0
al		-tr	-tr	-tr	-tr			1		=	0
ai				p_imp	p_imp				1	=	0
OBJ							1	1	1		

Figure C-1

Figure C-2 is a (bitmapped) picture of the matrix of model EGYPT. Each non-zero within the matrix is shown as a black pixel.

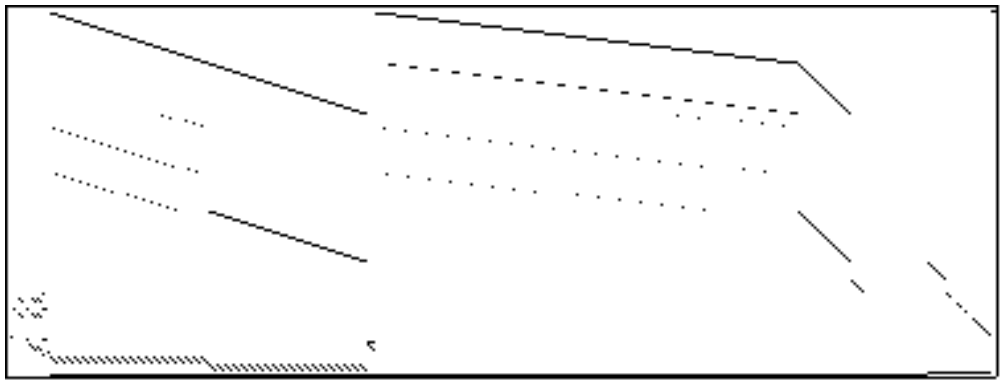

Figure C-2

REFERENCES

AARTS E., KORST J., [1990], Simulated Annealing and Boltzmann Machines, A Stochastic Approach to Combinatorial Optimization and Neural Computing, John Wiley-] Sons, Chichester.

BLAIR C.E., JEROSLOW R., LOWE J.K., [1985], Some Results and Experiments in Programming Techniques for Propositional Logic, to appear in Computers and Operations Research.

BROOKE A., KENDRICK D., and MEERAUS A., [1988], GAMS, A User's Guide, The Scientific Press.

COLMERAUER A., [1987], Opening the Prolog III Universe, Byte Magazine, August 1987, p.177-182.

CUNNINGHAM K., SCHRAGE L., [1989], The LINGO Modeling Language, University of Chicago, Preliminary, 27 February.

DOLK D.R., [1987], Relational Data Models and Relational Data Base Systems, NATO ASI on Mathematical Models for Decision Support, Springer.

EGLI G., [1980], Ein Multiperiodenmodell der linearen Optimierung für die schweizerische Ernährungsplannung in Krisenzeiten, Dissertation, University of Fribourg (Switzerland).

FOURER R., GAY D.M.*, KERNIGHAN B.W.* [1990], A Modeling Language for Mathematical Programming, Management Science 36:5 (May).

GEOFFRION A.M., [1987], An Introduction to Structured Modeling, Management Science Vol.33, p.547-588.

GEOFFRION A.M., [1989], SML: A Model Definition Language for Structured Modeling, Working Paper No. 360, Western Management Science Institute, University of California, Los Angeles, revised November 1989.

GLOVER F., [1989], Tabu Search - Part I, ORSA Journal on Computing Vol 1 No. 3, p.190-206, and Part II ORSA Journal on Computing Vol 2, No. 1 p.4-32.

GOLDBERG D.E., [1989], Genetic Algorithms in Search, Optimization, and Machine Learning, Addison-Wesley Publ.

GREENBERG H.J., [1981], "Graph-Theoretic Foundations of Computer-Assisted Analysis", Greenberg H.J. and Maybee J.S., (eds.), Computer Assisted Analysis and Model Simplification, Academic Press, New York, p. 481-495.

GREENBERG H.J., MURPHY F.H., [1991], Views of Mathematical Programming Models and Their Instances, Working Paper.

HOOKER J.N., [1988], A Quantitative Approach to Logical Inference, Decision Support Systems 4, North-Holland, p.45-69.

HÜRLIMANN T., [1987], LPL: A Structured Language For Modeling Linear Programs, Dissertation, Peter Lang Verlag, Bern.

HÜRLIMANN T., [1991], Linear Modeling Tools, Institute for Automation and Operations Research, Working Paper No. 187, July, Fribourg.

HÜRLIMANN T., [1992], Reference Manual for the LPL Modeling Language, Version 3.8, Institute for Automation and Operations Research, Working Paper No. 191, February, Fribourg.

JOHNSON E.L., [1989], Modeling and Strong Linear Programs for Mixed Integer Programming, in: Algorithms and Model Formulations in Mathematical Programming, ed. Stein W.W., NATO ASI Series F, Vol.51, Springer, p.1-43.

JEROSLOW R.G., [1985], Computation-oriented Reductions of Predicate to Propositional Logic, Working Paper, Georgia Institute of Technology, Atlanta, GA.

JEROSLOW R.G., [1989], Logic-based Decision Support, Mixed Integer Model Formulation, North-Holland, Amsterdam.

KARP R.M., [1972], Reducibility among Combinatorial Problems, in: R.E. Miller and J.W. Thatcher, eds., Complexity of Computer Computations, Plenum Press (1972), p.85-103.

KETRON [1986], PAM, a Practitioner's Approach to Modeling, Part 1: Primer, Ketron Management Science, Inc, Arlington, Virginia, revised September 1986.

KOHLAS J., MONNEY P.A., [1990], Modeling and Reasoning with Hints, Working Paper No.174, Institute for Automation and Operations Research, University of Fribourg.

LASSEZ C., [1987], Constraint Logic Programming, Byte Magazine, August 1987, p.171-176.

LENARD M. [1986], Representing Models as Data, J. Management Information Systems, Vol. 2, p.36-48.

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.

SCHWARTZ J.T., DEWAR R.B.K., DUBINSKY E., SCHONBERG E., [1986], Programming with Sets, an Introduction to SETL, Springer Verlag, New York.

SHAFER G., [1976], A Mathematical Theory of Evidence. Princeton University Press.

SHAPIRO J.F., [1988], Stochastic Programming Models for Dedicated Portfolio Selection, NATO ASI Series, Vol. F48,Mathematical Models for Decision Support, Edited by G. Mitra, Springer Verlag, Berlin, p.587-611.

SUNSET SOFTWARE [1990], XA, A Professional Linear and Mixed 0/1 Integer Programming System, 1613 Chelsea Road, Suite 153, San Marino, Ca 91108.

WILLIAMS H.P., [1974], Experiments in the formulation of integer programming problems, Math. Programming Study 2 (1974), p.180-197.

WILLIAMS H.P., [1977], Logical problems and integer programming, Bull. Inst. Math. Appl., 13 (1977), p.18-20.

WILLIAMS H.P., [1991?], Mathematical Programming Modelling, Working Paper,

Faculty of Mathematical Studies, University of Southampton, U.K.

WITTGENSTEIN W., [1920], Tractatus logico-philosophicus, suhrkamp Verlag, 13. Auflage, 1978.

ZIMMERMANN H.J., [1991], Fuzzy Set Theory and its Applications, Second Edition, Kluwer Academic Publ., Boston.