(Deckblatt vorne)

University of Fribourg
Series in Computer Science iiUF

Volume 2

Tony Hürlimann and Jürg Kohlas (eds), Louis Cardona, Gérard Collaud, Catherine Donnet-Portenier, Pius Hättenschwiler, Harald Häuschen

Modeling Tools for Decision Support

LPL : a modeling Language,
gLPS : a Graph-Based System for Linear Problems Modeling
NetCalc : a netform editor
ACMS : a constraint programming tool

Lang

(Deckblatt hinten)

Volume 2

Tony Hürlimann and Jürg Kohlas (eds), Louis Cardona, Gérard Collaud, Catherine Donnet-Portenier, Pius Hättenschwiler, Harald Häuschen

Modeling Tools for Decision Support

LPL : a Modeling Language,
gLPS : a Graph-Based System for Linear Problems Modeling
NetCalc : a netform editor
CHRIS : a constraint programming tool

This second volume is a collection of working papers, originating from four research groups studying and creating computer-based modeling tools. LPL is a modeling language for mathematical models; gLPS is a graph-based LP modeling system; NetCalc is a speadsheet-like software, where the underlying structure is a directed, acyclic graph; and ACMS is a constraint programming language for arithmetic expressions.

Modeling Tools for Decision Support

University of Fribourg Series in Computer Science.
Cahiers informatiques de l'Université de Fribourg.
Informatik-Berichte der Universität Freiburg.

Volume 2

PETER LANG

Tony Hürlimann and Jürg Kohlas (eds), Louis Cardona, Gérard Collaud, Catherine Donnet-Portenier, Pius Hättenschwiler, Harald Häuschen

Modeling Tools for Decision Support

LPL : a modeling Language,
gLPS : a Graph-Based System for Linear Problems Modeling
NetCalc : a netform editor
CHRIS : a constraint programming tool

PETER LANG

Modeling Tools for Decision Support ....

Authors :

Louis Cardona, Gérard Collaud, Catherine Donnet-Portenier, Pius Hättenschwiler, Harald Häuschen, Tony Hürlimann, Jürg Kohlas

Institut pour l'Automation et
la Recherche Opérationnelle
University of Fribourg
Miséricorde
1700 Fribourg
Switzerland

Fax: (41) 37 219.670
E-mail: Hurlimann@cfruni51.bitnet

ISBN xxx

Table of Contents

Preface

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, have made possible the management and solving of large and complex models on a desktop computer. Such a performance would have been unrealizable only ten years ago. *Artificial Intelligence* has developed rule-based expert systems to represent and process knowledge and given 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.

There are several reasons why these methods from OR and AI are not widely used in real decision processes. One of them 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 manage the model, although - once formulated in an appropriate form - it might be solved on a fly using different algorithms on today's desktop computer. Thus, the limiting factor for using simple desktop computers, in order to assist the decision maker are no longer the solution procedures, but the model management, i.e. the creation, modification, documentation, transformation of models, survey, views, analysis of results, result specification, a.o.

A major difficulty in developing such a system is that the modeling of knowledge representation has 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 can be used by everyone. Most modelers 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 is limited. Often model or parts of it could also be used in another context, but reusability is almost impossible. OR and AI journals are full of articles describing a special implementation of a model and its environment. The cost of developing such adhoc tools should not be underestimated. This is the main reason why decision makers tend to use rules of thumb rather than the troublesome path of model building.

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

Different research projects in the realm of modeling tools have been undertaken at the Institute for Informatik (formely IAUF) of the University of Fribourg, Switzerland. In this research report, four of them are presented: LPL (Linear Programming Language), a mathematical modeling language; NetCalc, a spreadsheet software based on directed graphs and hierachical dependencies; gLPS (graphical Linear Programming System), a graph-based LP modeling software system; and ACMS (Arithmetic Constraint Modeling System), a constraint programming language for arithmetic expressions. All the presented tools have been implemented and can be used for different modeling tasks.

The seven chapters are divided into the following four groups: Chapter 1-3 present several aspects of the LPL modeling language. Chapter 1 gives a brief overview of the language. LPL is a powerful language used for formulating large LP-models in a very concise manner. Its syntax is close to the usual mathematical notation of models. The language has been used to formulate various big real-life models. Chapter 2 exposes several applications of LPL to illustrate the wide range of models LPL can handle. Chapter 3 introduces the reader into the unit concept used in LPL. Since LPL seems to be the first language introducing this concept, it is of wider interest.

Chapter 4 gives a comprehensive overview of gLPS, the graph-based modeling system. gLPS has been interfaced with LPL. The modeler creates his mathematical model (essentially an activity-constraint-graph) using a Macintosh-like direct manipulation interface. The model is then translated automatically into the textual form of LPL.

Chapter 5 and 6 deal with the NetCalc system. Chapter 5 gives the necessary theoretical background for top-down modeling, whereas Chapter 6 presents the implementation of NetCalc. NetCalc is a very useful tool for the natural top-down modeling process. Since the implementation is based on a database system, arbitrarily large models can be formulated. A bottom-up evaluator can evaluate all nodes of the graph, even cyclic, as long as the functions converge.

Chapter 7, finally, presents the ACMS system. A model in ACMS consists of a set of arithmetic equations. This declarative modeling language evaluates expressions by constraint satisfaction and local propagation of values of already known entities.

Tony Hürlimann & Jörg Kohlas

1 LPL: A Modeling Language

T. Hürlimann

Key-words: Modeling, Linear Programming, Compiler, MPS.

Summary: This paper describes the new version of the modeling language, named LPL (Linear Programming Language). It may be used to build, modify and document mathematical models. The LPL language has been successfully applied to generate automatically MPS input files and reports of large LP models. The available LPL compiler translates LPL programs to the input code of any LP/MIP solver, calls the solver automatically, reads the solution back to its internal representation, and the integrated Report Generator produces the user-defined reports of the model. Furthermore, an Input Generator can read the data from many formats.

Stichworte: Modellierung, lineare Programmierung, Compiler, MPS.

Zusammenfassung: Dieser Artikel beschreibt die neue Version der Modellierungs-sprache LPL (Linear Programming Language), die sich dazu eignet, mathematische Modelle aufzubauen, zu warten und zu dokumentieren. Die LPL-Sprache wurde zum Erstellen von MPS-Input-Dateien und Resultate-Tabellen größerer LP-Modelle erfolgreich eingesetzt. Der LPL-Compiler übersetzt ein LPL-Programm, das ein vollständiges Modell repräsentiert, in den Eingabecode eines LP/MIP-Lösungsprogramms, ruft den Lösungsalgorithmus auf, liest die Lösung, und ein integrierter Tabellengenerator gibt vom Benutzer definierte Resultate-Tabellen aus. Außerdem erlaubt ein Dateneingabe-Generator, die Daten in verschiedenen Formaten zu lesen.

Acknowledgment: The author is grateful to Prof. Dr. J. Kohlas and the two referees who have made valuable comments on the paper.

1 Introduction

This paper presents the new version of the modeling language LPL. A first version of LPL was developed and implemented some time ago. It was described in [Hürlimann, Kohlas 1988]. The new version, exposed in this paper, has been improved in several respects: A powerful Input and Report Generator has been integrated, the expression syntax has been enlarged with several new functions and operators, units can be used to measure numerical entities, restricted string manipulation is now possible, goal and multi-stage programming are also supported to some extent, and, finally, an open interface to most LP/MIP solver has been added. Many more modifications and, hopefully, improvements have taken place, but will not be mentioned here. A speciality of the old LPL version was its hierarchical indexing: Indices were nested lists which could be used as any other index within a model. To use nested lists for indexing is very intriguing, since many sets of objects can be arranged as a tree in a natural way. Practical experiences, however, have shown that they obscured the model structure - at least in the form they have been implemented in LPL. Furthermore, few modeling examples came around which used this option really. Many examples could be formulated more simply by using relations. It was, therefore, decided to drop hierarchical indexing from the LPL language. More research work should be done, before hierarchical indexing can be integrated in an advanced modeling language. A first step in the right direction is [Bisschop 1991].

Different modeling languages have recently been developed. AMPL [Fourer, Gay, and Kernighan 1990], GAMS [Brooke, Kendrick, and Meeraus 1988], LINGO [Schrage 1989], and SML [Geoffrion 1989] are closest to the LPL language. These languages, and some others, have been compared in [Steiger and Sharda 1991]. LPL has been updated since then and differs in some respects from these other languages:

- LPL has an open interface to any LP/MIP solver. This means that any linear solver can be called from within the LPL code. The user can configure the communication between LPL and the solver.
- LPL contains an Input and Report Generator to read the data from files and to write the results to files. While GAMS too includes a Report Generator, only LPL has an Input Generator. Actually, the Input/Report Generator can only read/write text files, although in a rather sophisticated way. A future version is planned to embed SQL statements to extend the language for database access.

- LPL has a flexible syntax: entities can be reassigned as in GAMS, but unlike GAMS a model structure can be parsed and compiled without the data specified.
- LPL includes a Unit statement for measurement checks.
- Furthermore, LPL allows the user to generate alias-names for variables and restrictions.

On the other hand, LPL is restricted to linear and integer models, and is not interactive. Another disadvantage which is common to all modeling languages should be addressed: the poor and restricted communication to other model functions. Since the modeling language is a closed system, it must handle the model to the solver via file transfer. For linear models, MPS is a standard. It is costly in time to make the detour via the MPS-file. It would be much simpler to generate the model within the main memory where the optimizer can directly manipulate the model data. This is only possible if the LPL object code and the optimizer code could be linked together.

A modeling language is certainly an important part of a modeling system, but various other tools should be integrated within a model management system: Tools to manage the model data [Blanning 1987, Dolk 1988], to verify and analyze the model [Greenberg 1990], to document the model [Gass 1984, Geoffrion 1989], as well as tools to support the evolution of a model.

The contribution of LPL is certainly only a first but important step towards a general model management system. 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
- open interface to most LP/MIP solver packages.

Indices

j	bonds $(j = 1,\ldots,N)$
t	time periods $(t = 1,\ldots.,T)$ with $T \leq 50$

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

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

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

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

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

Figure 2-1

The basic ideas of LPL are presented in §2 and §3 using a simple portfolio model example. §4 and §5 examine the indices and expressions, the most important constructs of LPL. Some aspects of the Report and Input Generator are discussed in §6. Appendix A lists a complete assignment model containing 2700 0-1 variables (assign 180 players to 15 teams), which covers some interesting features of LPL.

2 A Simple Example

Figure 2-1 displays the algebraic formulation of a simple portfolio model [Shapiro 1988, p.589f]. This formulation can be translated directly to an LPL model formulation as shown in Figure 2-2.

The algebraic formulation of Figure 2-1 contains different sections: Index-sets, numerical data-tables, variables (the unknowns), a minimization function, and different constraints. Note that this formulation in Figure 2-1 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.

```
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 2-2

Figure 2-2 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 hand, 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)*.

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

- *{$I...* reads the model data from file MODEL.DAT written also in LPL syntax (Figure 2-3),
- *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 (see §3.7).

```
{ 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 2-3

A specific data set for this model is shown in Figure 2-3 written in LPL syntax. The data-tables can also be in plain text. LPL includes a flexible Input Generator to read external data definition (see §6).

3 A Brief overview of the LPL language

In this section, the different components of an LPL model are briefly presented.

3.1 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 themselves of this set are not defined in this place, they are part of the model data defined in file MODEL.DAT (Figure 2-3). The modeler is, however, free to declare a set and to assign its elements to 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 (see Appendix A)

```
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 says 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) tuple-list. 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.

3.2 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 one 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 2-3 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"
```

3.3 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 (see any OR textbook for more details of the MPS code).

3.4 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) in several places within the model - to call the solver repeatedly within an 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 on whether the function is minimized or maximized. This instruction calls the solver directly.

3.5 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 is 'hardwired' within the LPL compiler, but other solver packages such as OSL, Cplex, MOPS or LINDO work too.

3.6 Reports

LPL does not only allow one 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 2-2 as

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

which prints four tables in a predefined format.

A more complex Print statement is found in the Appendix A, and is explained in §6.

3.7 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 their name. *Derived units* (as 'speed=meter/sec') are declared by their 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 follows

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

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

The different parts of an LPL model have now been described very briefly. Some aspects will be explained in greater detail in the subsequent sections. Most examples are from the model in Appendix A.

4 The Use of Indices

The indices are used to define multi-dimensional items, such as coefficients, variables, restrictions, or indexed sets. They are called *tables*.

```
    COEF a(i,j) = 1;
    VAR  x(i,j,k,l);
    MODEL r(i);
    SET s(i,j);
```

a declares a two-dimensional numerical table - a matrix - and assigns the value 1 to every table-entry, *x* is a four-dimensional variable, *r* is a restriction-vector, and *s* is a two-dimensional indexed set. In the same manner, the indices are used for the different *index-operators* (SUM, PROD,...). They are operators, which iterate over index-sets.

```
    ....SUM(i,j) a[i,j]...
    ....PROD(i,j,k,l) x[i,j,k,l]...
```

The first example returns the sum of all values in the table *a[i,j]*, the second returns the product of all *x[i,j,k,l]*.

In the last examples, the indices play an *active* part: In the expression *COEF a(i,j)=1*, for example, the indices *i* and *j* determine how many entries the table *a* contains, and the same indices decide how many terms the summation extends in the expression *SUM(i,j) a[i,j]*. One may compare this syntax with the nested loops in a classical programming language.

Indices, however, are also used in algebraic expression (within the brackets *[...]*) in the above examples, where they play a *passive* part. In the expression

```
    COEF a(i,j) = b[i,j] + 1;
```

every table-entry of *b* plus one will be copied to the corresponding table-entry of *a*. The indices in *[i,j]* play a passive part. Every passive index must be bound to an active index. In the last example, the index *i* in *[i,j]* will be bound to the index *i* in *(i,j)*, and *j* in *[i,j]* is bound to *j* in *(i,j)*. The binding guarantees a unique expression evaluation. In the assignment

```
    COEF a(i,i,j) = b[i,j];    "no error, but..."
```

the index *i* in *[i,j]* could be bound to the first or second *i* within the index-list*(i,i,j)*. Therefore, a unique binding is not possible. (By default LPL will bind it to the last, but this may be dependent on the implementation).

Binding is flexible in LPL. It is, however, necessary that the bound and the binding indices represent the same index-set. An expression such as

```
    COEF a(i,j) = b[i,k];  "bound error!"
```

produces an error. It must be replaced by

```
    COEF a(i,j) = b[i,j IN k];  "correct"
```

j IN k returns the position of a specific element of *j* within the set *k*. (In this case, *k* is regarded as an ordered set). If the specific element of *j* is not in *k*, the whole expression *b[i,j IN k]* returns zero. If the intersection of *j* and *k* is a real subset of *k,* then the table *a* will be filled up only partially by the assignment, the rest of the table *a* will remain unchanged.

In general, the LPL compiler tries to bind the passive indices automatically, identified by their names. The modeler, however, has the possibility to force any binding using *dummy-indices*. The corresponding, active indices must be headed by a dummy-index, followed by an equal sign or the IN reserved word.

```
    COEF a(d1=i,d2=j) = b[d1,d2];
    COEF a(d1 IN i,d2 IN j) = b[d1,d2];  "the same"
```

The dummy-indices *d1* and *d2* can then be used instead of the passive indices within the expressions. This forces the binding from *d1* to *i* and from *d2* to *j*. Every active index within an LPL model can be extended by a dummy. The same dummies may be used in different expressions, since they have only local effect. (They are really treated like local variables in programming languages). In some situations dummies are necessary, but in most contexts they may be dropped. It is a matter of style, whether the modeler wants to use them systematically or not. LPL even makes it possible to drop all passive indices if they can be bound uniquely. Since this relaxed use of indices within LPL models might be dangerous, LPL also offers a compiler switch, which restricts this practice [Hürlimann 1992, p 69].

Indices can also be used as terms within expressions. In this case, they return the position of an element within the index-set. The expression

```
    COEF a(i) = i;
```

assigns the values 1, 2, 3, ... , and *n* to the table *a*, if *n* is the cardinality of index-set *i*.

5 Algebraic and Logical Expressions

Algebraic and logical expressions are an important part of any LPL model. They are used in three different contexts:

- to define model restrictions
- to evaluate and to output intermediate expressions and tables

• to declare sparse, indexed tables.

Expressions are built using the usual mathematical notation with arithmetical (+ - * / ^) and logical (*and, or, not*) operators, coefficients, numbers and functions (Figure 5-2). They may also contain *index-operators* (Figure 5-3), which are iterated over index-sets. The sum-operator ∑, replaced by the reserved word SUM, is such an index-operator. LPL, like the programming language C, does not distinguish between algebraic and logical expressions. In logical expressions zero is used as FALSE and any other number is interpreted as TRUE. Figure 5-1 gives an overview of all operators in LPL in order of decreasing precedence. Every expression returns a numerical value.

Coefficients or variables within an expression return the corresponding value. If a coefficient or variable has not been assigned before in the model, a default value is taken. The default value can be entered through the reserved word DEFAULT following a declaration as in

```
COEF a(i) DEFAULT 2;
```

If no default value has been declared by the modeler, it is zero.

Examples of expressions are

```
4.6e5 + 7^9 - sin(879.8)
(((4+5)*4)^2 - 12 ) + if(a>0,3,7)
a or b and not (3*4)
sum(i) (a[i]+b[i])/2
prod(i) x[i,j]
```

The last expression contains an unbound index *j*. Therefore, it returns not one *single* numerical value, but a one-dimensional table of values over *j*. The index can be bound, if the expression is assigned to a table *b*, or if another index-operator is added to the expression:

```
COEF b(j) = prod(i) x[i,j];
sum(j) (prod(i) x[i,j]);
```

sin, log,...	functions
+ - not # IN	unary operators
^	power
* / %	product and division
sum, prod,...	index-operators
+ -	addition and subtraction
= <> < > <= >= ~	relational operators
and	logical AND
or	logical OR
, (or |)	enumeration operator

Figure 5-1 (Operators)

max(x,y)	returns x, if x>y, else returns y
min(x,y)	returns x, if x<y, else returns y
abs(x)	returns the absolute value of x
ceil(x)	returns next integer greater than x
floor(x)	returns next integer less than x
trunc(x)	returns the truncated x
sin(x)	returns the sinus of x
cos(x)	returns the cosines of x
log(x)	returns the nat. logarithm of x
sqrt(x)	returns the root of x
rnd(x,y)	returns a uniform distributed number
rndn(x,y)	returns a normal distributed number
if(x,y,z)	returns y, if x is TRUE, else returns z

Figure 5-2 (Functions)

SUM(i) a[i]	sum all a over i
PROD(i) a[i]	multiply all a over i
SMIN(i) a[i]	the smallest a[i]
SMAX(i) a[i]	the biggest a[i]
EXIST(i) a[i]	tests, whether all a[i] is FALSE
FORALL(i) a[i]	tests, whether all a[i] are TRUE
PMAX(i) a[i]	position i of the biggest a[i]
PMIN(i) a[i]	position i of the smallest a[i]
COL(i) ...	horizontal table-expander
ROW(i) ...	vertical table-expander

Figure 5-3 (Index-operators)

It is important to see, that LPL allows the evaluation and return of indexed expressions and not only single values. Insofar, LPL is more a table-manipulator language than a modeling language.

5.1 Restrictions

Expressions allow the defintion of model constraints. The syntax of an expression representing a restriction is limited. No logical operators are allowed; furthermore, the expression must be linear relative to the defined variables. The LPL compiler tests such conditions. Restrictions may also be defined as ranges. The same variable may occur several times within the expression. If the constant expression of a variable evaluates to zero, then the corresponding variable is automatically eliminated from the restriction.

```
MODEL R: x + y = sum(i) z[i];    "defines a restriction R"
MODEL B: a <= x+y <= b;          "defines a range"
MODEL L(i): x[i] + y = 5;       "defines a restriction over i"
MODEL S: x + 3*y = 4*(x+y);     "x and y occur twice"
MODEL T: x + z -y = (4-4)*x +x; "x will be eliminated"
```

The restriction *S* will be automatically simplified to *-3*x - y = 0*, and in *T* the variable *x* is eliminated.

5.2 Evaluating Tables

Expressions are also used to produce intermediate results.

```
COEF a(i,j) = b[j,i];
COEF d(i,j) = a[i,j] + b[i,j];
EQUATION e(i,k) = SUM(j) a[i,j]*c[j,k];
PRINT(i,j) : SUM(k) c[j,k] + a[i,j];
```

The first assignment copies the transposed table *b* into the table *a*. The second assigns the addition of the two matrices *a* and *b* to the table *d*. The third defines a matrix-multiplication, but the statement is not executed at this point, the evaluation is delayed instead: each time *e* is used in another expression the term at the right hand side is evaluated. The fourth calculates and outputs a two-dimensional table.

5.3 Sparse Tables

Another important application of expressions is the definition of sparse tables. This is explained by an example. Suppose we have to define a transportation model; the following model components are needed:

```
SET i;               "a list of locations"
SET links(i,i);      "a list of links between the locations"
COEF costs(i,i);     "the transportation costs of the links"
VAR x(i,i);          "the unknown transportation quantities"
```

The indexed set *links* defines which transportation links exist between the different locations *i*. Normally, this list is a (sparse) subset of all possible link-combinations. Since the links exist between some but not all locations, it is clear that the transportation costs make sense on these links only. Suppose, furthermore, that a quantity *x* is only transported on links with costs less or equal to 1000. In LPL it is possible to define such sparse tables in the following way

```
COEF costs(d1=i,d2=i | links[d1,d2]);
  { or : COEF costs(links); }
VAR x(d1=i,d2=i | links[d1,d2] and costs[d1,d2]<=1000);
  { or : VAR x(links | costs<=1000); }
```

Costs is defined over *(i,i)*, if *links(i,i)* is TRUE. If the expression - headed by a '|' - evaluates to TRUE, the tuple exists; otherwise it is not defined. The subsequent *use* of the variable *x* restricts the list automatically to the existing ones. Therefore, the expression

```
... + SUM(d1=i,d2=i) x[d1,d2] + ...
```

does not sum *all* (i,i)-tuples of *x*, but only the existent ones defined in links and restricted by the expression *costs<=1000*.

The same syntax may be used for the iteration process of the index-operators too. An example is the following expression: Sum the values of the table *a(i)*, such that *i<=5*

```
   SUM(i | i<=5) a[i];
```

The portfolio model uses this option in the *Cash_bal* constraints

```
   MODEL Cash_bal(t|t>1): SUM(j) f*x + a*s[t-1] - s = L;
```

This constraint is only defined for *t>1*, but will not be produced for the period *t=1*. Another example is the restriction *Must* in the soccer model of Appendix A

```
   Must(p,t | Must_be_in): Work = 1;
```

which would produce 2700 (=180x15) single restrictions without the condition *Must_be_in*, but actually produces only the following three constraints

```
   Must1: Work[P1,T2]=1
   Must2: Work[P2,T6]=1
   Must3: Work[P3,T7]=1
```

since *Must_be_in* is defined as

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

which is an indexed set containing three tuples.

6 The Report and Input Generator

The Print statement allows the output of tables. They are written to a file, which may be further edited with the modeler's favoured text-processor. The simplest statements are the following, where LPL uses a standard layout of the output tables.

```
   PRINT a;               "output table a"
   PRINT b:2;             "output table b with 2 positions"
   PRINT c:10:2;          "output table c with 10 positions and 2 decimals"
   PRINT a; b:2; c:10:2;  "output all three tables"
   PRINT a [10,10];       "output the table a in blocks of 10x10"
```

6.1 Print Tables of Expressions

LPL also allows us to output tables of any expression. In this case, the reserved word PRINT is separated from the expression by a colon. The layout is, once again, chosen by the LPL compiler.

```
   PRINT : a*b:12;           "output a value on 12 positions"
   PRINT(j,i) : a[i,j];      "output the transposed matrix a"
   PRINT : 'this text';      "output a text line"
   PRINT(j) : SUM(i) a[i,j]; "output the column totals of a"
```

The reserved word PRINT is followed by an index-list, if the expression is indexed.

6.2 Masks

For more complex layouts, the modeler can define a layout mask through a Mask statement. The Mask statement is headed by the reserved word MASK and a mask-name. This is followed by the contents of the mask, which extends to the reserved word ENDMASK. The contents of the mask may be any string of characters. The two characters # and *$*, however, have a special signification and are called *place-holder*. They are replaced by the numerical or string expressions of a subsequent Print statement.

Example

```
MASK m1    "'m1' is the mask-name"
  Month: $$$$$$$$  :  ####.##%   ####.#### gallons
ENDMASK
PRINT m1 ( 'April' , 17.15*2 , 2378.567321 );
```

These two statements produce the following output

```
  Month:    April  :    34.30%   2378.5673 gallons
```

The Print statement consists of the reserved word PRINT, the mask-name, and an expression-list within parentheses. A comma separates the different expressions within the list. The expressions are filled into the mask from left to right and from top to bottom at the different place-holders.

But only the two index-operators ROW and COL reveal the real power of the Report Generator. *The index-operator ROW (COL) can expand a place-holder vertically (horizontally) over index-sets.* The combination of both index-operators allows the user to produce rather complex tables. The syntax of both operators is the same as for any other index-operators. Figure 6-1 shows the result of the Mask and Print statement of the model defined in Appendix A (some lines have been cut).

```
           Results per player
         ***********************
player  Skill Age  in Team

P1         3   10    T2
P2         7   10    T6
P3         4   10    T6

... 174 lines are cut here ...

P178       4   11    T5
P179       5   11    T9
P180       7   11    T3

            Results  per team
         ***********************
team   skill  age  Av.Skill  Av.Age
-----------------------------------
T1        59  125    4.917   10.417
T2        59  124    4.917   10.333
T3        59  124    4.917   10.333
T4        59  127    4.917   10.583
T5        59  126    4.917   10.500
T6        59  124    4.917   10.333
T7        59  130    4.917   10.833
T8        59  124    4.917   10.333
T9        59  128    4.917   10.667
T10       59  127    4.917   10.583
T11       59  128    4.917   10.667
T12       59  124    4.917   10.333
T13       59  126    4.917   10.500
T14       59  126    4.917   10.500
T15       60  124    5.000   10.333


(continue...)


team | players in the team
-----------------------------------------------------------------
```

```
T1   | P9   P41  P49  P57  P70  P71  P86  P89  P118 P143 P145 P154
T2   | P1   P6   P7   P22  P23  P48  P52  P55  P92  P133 P149 P162
T3   | P38  P39  P54  P56  P91  P93  P97  P119 P141 P142 P174 P180
T4   | P20  P51  P73  P78  P80  P84  P88  P103 P107 P131 P156 P157
T5   | P21  P42  P63  P75  P82  P100 P102 P105 P134 P148 P175 P178
T6   | P2   P3   P4   P8   P35  P47  P68  P69  P81  P85  P98  P146
T7   | P24  P28  P34  P45  P58  P59  P87  P121 P126 P135 P140 P151
T8   | P10  P16  P18  P19  P64  P108 P109 P111 P147 P160 P171 P173
T9   | P30  P36  P37  P40  P62  P83  P90  P130 P132 P150 P158 P179
T10  | P15  P46  P72  P76  P96  P112 P114 P124 P127 P137 P166 P172
T11  | P5   P33  P53  P94  P120 P123 P138 P153 P155 P159 P167 P177
T12  | P12  P13  P25  P26  P29  P32  P44  P50  P77  P95  P101 P136
T13  | P11  P14  P17  P27  P65  P79  P128 P139 P144 P152 P161 P163
T14  | P31  P43  P60  P67  P74  P106 P113 P115 P122 P125 P170 P176
T15  | P61  P66  P99  P104 P110 P116 P117 P129 P164 P165 P168 P169
```

Figure 6-1

The mask contains 11 place-holder. The first four are defined on line 5 as

```
$$$$$$    ##   ##    $$$$
```

They are filled by the expression

```
(p , Skill , Age , COL(t|work) t)
```

in the PRINT statement as follows. The first place-holder is replaced by the name of player *p*, the second and third by the *Skill* and *Age* value of player *p*. The expression *COL(t|work) t* runs over all *t* for a specific *p* and outputs all team names *t* horizontally for every *work[p,t]* that is TRUE. Since a player *p* can only be in one team, this outputs only one team-name per line. Because the whole expression is also defined over *ROW(p)*, the output of this line is repeated for every player *p*, which will produce 180 lines in our case. The next five place-holders are defined on line 9 of the mask. They are filled by the expression

```
ROW(t) (t , SUM(p|work) Skill , SUM(p|work) Age ,
        (SUM(p|work) Skill)/12 , (SUM(p|work) Age)/12 ) ,
```

The first place-holder is filled by the team name t, the next by the total skill per team, calculated by the expression *SUM(p|work) Skill*. The next ones are calculated similarly. The last two place-holders in line 16 are filled by the expression

```
ROW(t) (t , COL(p|work) p)
```

Note that this simple line produces a whole two-dimensional table. The players are collected per line. The iteration of the ROW and COL index-operators can also be restricted by a condition. This allows us to select any subset of tables and to output the result in most complex layouts.

6.3 The Input Generator

The Input Generator is represented by the Read statement. It is similar to the Print statement, but instead of output data, it allows the reading of data from files. As an example, suppose the data of the portfolio model in Figure 2-3 are not organized in LPL syntax, but in a plain text file shown in Figure 6-2.

```
A data set of the portfolio model: file PORTDATA
```

```
Three tables are defined

Table 1 : bond data
.   bond name   price   min. pur-     max pur-
.                       chase quan.   chase quan.
.-------------------------------------------------
        A1        2        10          500
        A2      2.3         .          700
        A3        4         .          700
        A4        1        15          800
        A5      2.4        20          900
Endtable 1

Table 2 : period data

.     period   cash avail.  reinvestment
.--------------------------------------
        T1          -          -
        T2        12000        90 %
        T3        14000        80 %
        T4         5000        80 %
Endtable 2

Table 3 : cash flow
.           T1   T2   T3   T4
.----------------------------
      A1      .    4    4    4
      A2      .  5.5    5    4
      A3      .    5    3    6
      A4      .    6    6    .
      A5      .    4    5    6
Endtable
```

Figure 6-2

Then the data specification of the model can be replaced by the four Read statements

```
READ FROM 'portdata' BLOCK FROM 'Table' TO 'Endtable';
READ BLOCK 1 : ROW(j) ( j , c , q , Q );
READ BLOCK 2 : ROW(t) ( t , L , a );
READ BLOCK 3 : ROW(j) ( j , COL(t) f[j,t] );
```

The first Read statement indicates the filename of the input file and the *read-block-delimiters*, between which a subsequent read takes place. The second Read jumps to the first occurrence of *Table* (beginning a new line) and reads lines until a *Endtable* is encountered. On each line four tokens are read, if possible. *ROW()* has two functions: it reads lines repeatedly, and synchronizes the read. This means, if a line contains more than four tokens only the first four are read, if it contains less, less are read. In either case, a fresh read of four tokens will begin on a new line. *COL()*, too, repeats the reading of tokens on the same line, until the line ends.

It is also possible to read from different files. The three tables could have been divided into three files. In this case no block indication would be necessary. The Input Generator has several nice features. One is the following: empty lines or lines containing only separators between tokens, such as spaces or tabs, or lines beginning with a dot are skipped automatically. First experiences with the Input Generator are promising: For an LP model with 1300 constraints and 1500 variables, a Pascal program of 32 pages had been written to manipulate the data; using the new LPL Input Generator, it is now possible to code the same program in 2 pages [Hürlimann 1991].

7 Conclusion

This paper is a brief report on the new version of the LPL modeling language. The Reference Manual [Hürlimann 1992] gives a detailed description of the language. A model library of about 60 examples written in LPL is available.

The development of LPL was motivated by practical use. Different models with 1500-2000 restrictions, 2000-3500 variables, and a matrix density of 7500-12000 non-zero elements are under continuous use and development at the Institute [Hättenschwiler P, Kohlas J., 1989]. Most of them are formulated in LPL. Until very recently, these models had to be solved by a solver package on a main-frame computer. Therefore, the important task of the LPL was to produce the solver input file. The hardware of the new generation of PCs as well as the solver software packages such as XA a.o. now allow one to solve and manipulate the mentioned models locally on the PC. LPL has, therefore, been extended to a point, where different modeling tasks (model formulation - solving and report writing) are supported. Our practical experiences are, that a typical modeling-cycle (modification - resolving - reporting results) has been reduced from several hours or even days to several minutes.

The LPL compiler has been implemented with TURBO PASCAL from Borland Inc. under MS/DOS (any version). A version in ANSI C is also implemented. The PASCAL version is actually available at the Institute for Automation and Operations Research, University Fribourg, CH-1700 Fribourg, Switzerland [fax (41) 37 21 96 70].

Appendix A

The following model (SOCCER.LPL) was written in a draft form by J. Byer. I have adapted this model for use in this paper. The model is a simple assignment problem which assigns 180 players to 15 teams. The only variable is *Work* that is indexed over the players *p* and the teams *t*. It declares a total of 2700 (=180*15) single 0-1 variables. *work(p,t)* has only the values 1 or 0, depending on whether the player *p* is assigned to team *t* . The model was solved on a PC 80386 with a Coprocessor and 4 MB RAM with XA386 in 30 minutes - we were lucky, since the model is a pure 0-1 IP (integer program).

```
(* SOCCER.LPL:an assignment problem of 180 players to 15 teams (0-1 IP-
problem)
   Ref: BYER James, personal communications *)
   {--- compile this model with the LPL compiler LPLLONG.EXE ---}


PROGRAM Soccer ;   (* {$N2??} *) {$R1 (initialize random seed)}

SET
  t;                 "the list of teams"
  p;                 "the list of players"
  must_be_in(p,t);   "player p must be in team t"
  reject_from(p,t);  "player p is rejected from a team t"
  t_groups(p,p);     "groups of players which must be together in a team"
  n_groups(p,p);     "groups of players which must never be together"

COEF Skill(p);       "skill of player p"
     Age(p);         "age of player p"

VAR  work(p,t) BINARY  "work=1, if player p is in team t, else 0" ;

MODEL
  "maximize the assignment"
  obj: SUM(p,t) work;
  "player p must be only in one team"
  Bounds(p): SUM(t) work = 1;
  "total skill level of team t must be at least 59"
  Skill_level(t): SUM(p) work * Skill >= 59;
  "total player count per team t must be 12"
  Heads(t): SUM(p) work = 12;
  "total team age of team t must be at least 124"
  Team_age(t): SUM(p) work * age >= 124;
  "player p must be in team t"
  Must(i=must_be_in): work[i] = 1;
  "player p is rejected from a team t"
  Reject(i=reject_from): work[i] = 0;
  "players which must be in the same team"
  Same(t,t_groups[i,j]): work[i,t] - work[j,t] = 0;
  "players which must not be in the same team"
  Never(t,i=p |exist(j=p)n_groups[i,j]): SUM(j=p|n_groups[i,j]) work[j,t]<=1;


{------ data -----}
SET
  t  = /T1:T15/;
  p  = /P1:P180/;
  must_be_in(p,t)  = / P1 T2 , P2 T6 , P34 T7 /;
  reject_from(p,t) = / P10 T1 , P20 T2 ,
                       P166 (T1 T3 T4 T5 T6 T7 T8 T9) ,
                       P64 (T1 T12) /;
  t_groups(p,p) =    / P2 P3 , P112  P76 , P89 P9 , P34 P135 ,
                       P4 (P35 P47 P81 P98) /;
  n_groups(p,p) =    / P21 P22 , P55 P56 ,
                       P11 ( P35 P45 P56 P67 P78 P89 P90 P21 ) /;

COEF  Skill(p) = trunc(rnd(3,8));
      Age(p)   = trunc(rnd(10,12));

"check that a player p can be only in one team"
CHECK it(p): sum(t | must_be_in)1 <= 1;
{----------- end data -----------}

MAXIMIZE obj;
MASK m1
          Results per player
       ***********************
player  Skill Age  in Team

$$$$$$    ##   ##    $$$$

team   skill  age  Av.Skill  Av.Age

$$$$$  #####  ###  ###.###  ###.###

          Results  per team
       ***********************
team | players in the team

$$$$ | $$$$

ENDMASK;

PRINT  m1(ROW(p) (p , Skill , Age , COL(t|work) t) ,
          ROW(t) (t , SUM(p|work) Skill , SUM(p|work) Age ,
                  (SUM(p|work) Skill)/12 , (SUM(p|work) Age)/12 ) ,
          ROW(t) (t , COL(p|work) p) );
END
```

References

BISSCHOP J.J., KUIP C.A.C., [1991], Hierarchical Sets in Mathematical Programming Modeling Languages, Working Paper, Department of Applied Mathematics, University of Twente, The Netherlands.

BLANNING R.W. [1987], A Rational Theory of Model Management, in C.W. Holsapple and A.B. Whinston (eds.), Decision Support Systems: Theory and Application, Springer Verlag.

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

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

DOLK D.R. [1988], Model Management and Structured Modeling: The Role of an Information Resource Dictionary System, Communications ACM, 31:6 (June), pp.704-718.

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

GASS S.I., [1984], Documenting a Computer-Based Model, Interfaces, 14:3 (May-June) pp.84-93.

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

GREENBERG H.J. [1990], A Primer for ANALYSE: A Computer-Assisted Analysis System for Mathematical Programming Models and Solutions, University of Colorado, Denver, Draft, June 28.

HÄTTENSCHWILER P., KOHLAS J., [1989], Wissensbasierte Systeme auf der Grundlage linearer Modelle - Werkzeuge und Anwendungen, Output, Vol.18/12, December, Goldach, Switzerland.

HÜRLIMANN T., KOHLAS J., [1988], LPL: A Structured Language for Linear Modeling, OR Spectrum, Vol.10, p.55-63, Springer Verlag.

HÜRLIMANN T. [1991], The Input and Report Generator in LPL, Institute for Automation and Operations Research, Working Paper No. 190, September, Fribourg (updated April 1992).

HÜRLIMANN T. [1992], Reference Manual for the LPL Modeling Language, Version 3.8, Institute for Automation and Operations Research, Working Paper No. 191, September 1991, updated February 1992, Fribourg.

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, S.587-611.

STEIGER D., SHARDA R., [1991], LP Modeling Languages for Personal Computers: A Comparison, Working Paper 90-27, College of Business Administration, Oklahoma State University.

2 LPL: Applications

T. Hürlimann

Key-words: Model Building, Modeling Language.

Abstract: LPL (Linear Programming Language) is a modeling language which allows one to formulate LP models. This was the main objective when the LPL language was designed. The language, however, is general enough to represent and manipulate other models. In this paper, various applications are shown where LPL might be used in addition to the 'traditional' application of pure LP models. The general character of LPL allows one to create network models, to use as a tool for decision analysis, discrete Markov chains and game problems, to manipulate vectors and matrices conveniently, and to handle multiple-objective LP's as well as goal programming.

Stichworte: Modellierung, Modelliersprache.

Zusammenfassung: LPL (Linear Programming Language) wurde ursprünglich als Formulierungssprache für lineare Modelle konzipiert. LPL kann aber auch für verschiedene andere Modelle gewinnbringend eingesetzt werden. Es ist das Ziel dieses Artikels, verschiedene solcher Anwendungen vorzustellen. Der allgemeine Character der LPL Modellierungssprache erlaubt es Netzwerkmodelle und diskrete Markov-Ketten bequem darzustellen. LPL kann auch dazu eingesetzt werden Vektoren und Matrizen zu manipulieren. Weitere Anwendungsfelder werden vorgestellt: Entscheidungsunterstützung und Mehrziel-Optimierung.

1 Introduction

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

It is well known that OR techniques are not often used in 'real' decision making, although they were intended to be particularly apt for solving decision problems. Therefore, for many years already, some people within the OR community have found a new vocation or mission: complaining about this situation. They put the blame for the gap between theory and practice on people with a really nice, quiet job at the University (or elsewhere), far from real problem decision making. They are not quite wrong! But then they try to specify their reproaches: "You should create more user-friendly (computer-based) decision tools! The user does not understand mathematics, so, please, do not use it (or at least hide it from the user)! One must speak the language of the decision maker!" Only the problem is, that we do not know the language of the decision maker, neither do we know what user-friendliness means. An environment called 'user-friendly' for one person, is cryptic for another. It is really hard to have 'something' user-friendly for everybody!

The situation is rendered even more difficult because the decision making process was a wide range of activity. First, the decision maker constructs a 'model' of his problem, and this model may take very different forms. It may, in a first step, be a jumble of diagrams together with notes in a natural language. It may not even be explicitly formulated, but only a 'fuzzy' collection of ideas in a human's brain! (Most models are probably in this form). For most decision making processes it might not be worthwhile to make the process explicit: The shortest path from home to work can be found without building an explicit mathematical model that is 'optimized' on a computer. Human beings 'model their world' mostly without consciously constructing a problem formulation. But sometimes it might be advantageous to make the problem explicit, especially when the problem must be communicated to other human beings. In this case, we need a form in which we can express the problem. The natural language is an excellent vehicle to store, formulate and communicate a 'model'.

The computer becomes more and more an instrument to aid the decision process and we need, therefore, also tools or a language to communicate the model to the computer. Unfortunately, the computer does not (yet?) understand the natural language. Most model makers use their own ad-hoc tools to get the model into the computer. There are important disadvantages in doing so. Models are difficult to maintain and to manage, they are not re-useable on another platform and the model transparency suffers.

In the realm of 'precisely defined' problems, which can be expressed as an algorithm, many different useful tools to communicate them to the computer exist: in the first rank there are programming languages. It is interesting to observe that many programming languages exist, but almost no standard modeling language. Why? Because programs represent well-defined algorithms, whereas modeling is a much more 'fuzzy' activity. Furthermore, modeling is done by most people and with very different background, while on the contrary programming (not only the coding activity, but also the program design activity) is a more well defined job. It is, therefore, not surprising that there exist many ad-hoc, but almost no 'universally' usable tools for modeling on a computer

It would certainly be of great use, if decision makers could dispose of some 'universally usable' modeling tools and methods to do their job. I am convinced that such tools are not only needed, but they are also possible. An important step, I think, would be to have a 'universally usable' modeling language, giving a unified platform or framework, which allows one to *specify* a wide range of models. The next step should then be to build modeling tools around the language to create, maintain, and modify the model specification as well as the data for a specific model instance. The efforts of structured modeling of GEOFFRION [1989] and others tend in this direction.

LPL (Linear Programming Language) - although originally designed for LP models - is also a language in which models from different realms can be formulated. In this paper examples besides the pure linear programming models are given. Section 2 gives a short overview of the LPL language as it is defined in version 3.8. Section 3 exposes some applications in matrix calculations and manipulation. Section 4 and 5 give some examples of how LPL can be used in decision analysis problems and discrete Markov chains models. Section 6 shows that network models could also be specified more appropriately in LPL as a collection of nodes and arcs, rather than as an LP model. Finally, section 7 gives an application in multi-objective LP modeling: goal programming.

2 Overview of The LPL Language

LPL is a declarative language which allows one to formulate LP models in a way close to the mathematical notation with indexed expressions. More precisely, the main features of LPL are:

- a simple syntax of models directly applicable for documentation
- building of large LPs with optional separation of the data and the model structure
- availability of a powerful index mechanism, making model structuring very flexible
- an innovative and simple report and input generator
- intermediate indexed expression evaluation
- 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
- transparent interface to any LP/MIP solver packages
- formulation of many model types not only LP's.

All mathematical models involve *variables*, *constraints*, and *objectives*. Together with *indices* and *coefficients,* mathematical *expressions* can be constructed. An LPL model is a sequential order of declarations and assignments of such model *entities*. Each declaration contains at least the *entity type* and the *entity name* and ends with a semicolon. The entity type is defined by one of the following keywords: SET, UNIT, COEF, STRING, VAR, EQUATION, MODEL, MASK, etc. The entity name is an alphanumeric string. All entities can be indexed which means that an entity may be a list of single entities. An LPL program looks like this:

```
SET       { declare and assign here all indices }
UNIT      { declare and assign here all measurements }
COEF      { declare and assign here all data }
STRING    { declare and assign here all character data }
VAR       { declare here all model variables }
EQUATION  { declare and assign all model constraints }
MODEL     { the same as EQUATION }
MASK      { declare and assign print and report layout masks }
PRINT     { report the results }
READ      { read data from files }
DELETE    { delete some entities from the model }
CHECK     { check the consistency of model entities }
MINIMIZE (MAXIMIZE) { expression to minimize or maximize }
```

The declarations may be in any order and the same declaration may be repeated. This allows the modeler to declare an entity, and to assign it later on within the model. Note, however, that every entity must be at least declared before it can be used in a mathematical expression. This means that the model can be read from top to bottom without forward references. At any place within the model, one can add comments enclosed in quotes ("..."), in braces {....}, or within (*....*). Comments have no formal meaning for the model (except "..."), but they are useful to document it. The keyword END ends the formal part of an LPL model.

Example:

A very simple and trivial LP model is the following production model, where the profit is to be maximized with a limited production capacity

```
(* Simple production model *)
SET   products;                  "a list of products is declared"
COEF  price(products);           "each product has a price"
      capacity(products);        "each product has a capacity"
VAR   Quantity(products);        "how much to produce for each product"
MODEL     restriction(products): Quantity[products] <= Capacity[products];
MAXIMIZE  profit: sum(products)  price[products]*Quantity[products];
END.
```

The data used for this model, which are the three vectors *products*, *price*, and *capacity*, can be read from a data-file using the READ statement in LPL.

SET entities consist of a linear list of distinct members. They may also be tuple lists of members of other sets.

```
SET i = / I1 I2 I3 I4 /;                "entity i is a linear list of four
members"
SET j = / B1:B6 /;                      "entity j is a set and contains 6
members"
SET k(i,j) = / I1 B1 , I2 A2 , I4 B5 /;    "k is a tuple list"
```

Sets define indices, which can be used to subscribe other entities in much the same manner as we use them in the mathematical notation for indexed items. The indices must be listed sequentially separated by a comma and enclosed within parentheses. Such a list is called *index-list*. Certain operators, such as the SUM operator, need also index-lists. These operators are called *index-operators*.

UNIT entities define the model units such as *meter*, *kilogram* etc. They can be used to measure other numerical entities, to check their commensurability, and to scale automatically commensurable entities.

COEF entities define the model data. They are condensed in tables. LPL offers several standard data format. Data may also be results of calculations or imported from files using the powerful Read Statement (import generator). COEF entities can be subscripted as any other entities.

```
COEF a;           "declare an entity a to be of type COEF"
COEF b:=1;        "declare b and assign 1 to it"
COEF c(i,j);      "declare an indexed entity c over set i and j (a matrix)"
COEF d := a*b;    "d as the result of a calculation"
```

VAR entities define the model variables. In contrast to the COEF entities, they are assigned under the solver's control, although the modeler may also assign values to them as for COEF entities. Several options may follow the VAR and COEF entity declaration. The modeler can define a range or restrict them to integer values.

```
VAR x(i)  INTEGER  [100,10];     "integer variable x bounded to [10,100]"
COEF c(i,j)  [.,2000];           "COEF entity c with upper bound of 2000"
```

EQUATION entities define the model constraints and the objectives. Objectives begin with the keyword MINIMIZE or MAXIMIZE, depending on whether the function has to be minimized or maximized. The constraint itself is a linear expression.

```
EQUATION r;                     "declare r to be a constraint entity"
EQUATION r : a*x + b*y <= 2;    "assign a simple expression to r"
EQUATION s(i,j);                "declare an indexed constraint entity s"
```

MODEL entities are the same as EQUATION entities, except that they are handed over to a solver. This allows us to solve different subsets of EQUATION entities; this is also useful for multi-goal programming.

MASK entities define the layout and the format of the output of the report generator. The mask contents is a string extending over several lines and ends with the keyword ENDMASK. The two characters # and *$* have a special meaning within the mask contents. They define numeric and alphanumeric *place-holders* which are replaced by numeric and string expressions when printing the mask to an output device.

```
MASK m
  This is the mask contents
  Second line with two place-holders: $$$$  ####
  That the last line of the mask contents
ENDMASK;
```

The PRINT statement writes entities of the model to an output device. Any entities or any expression can be printed. The layout of the output is defined by some internal formats or by previously defined MASK entities.

```
PRINT a;                   "output an entity a (which could be indexed)"
PRINT : a*b;               "output the result of an expression"
PRINT(j,i) : a[i,j];       "output the transposed matrix of a"
PRINT m ( 'John' , 234 );  "fill the mask m (see above) with two expressions"
```

The READ statement allows one to read data from an input device in a complex way. A read session cuts the input stream into tokens and the Read expression indicates how the tokens are to be translated into model entities.

```
READ : COL(j) j , ROW(i) ( i , COL(j) a[i,j] ;  "read a two-dimensional
table"
```

The DELETE and CHECK statement manipulate the entities. Entities can be deleted dynamically from the model. Any reference in the subsequent part of the model will produce an error, since these entities are no longer available. The CHECK statement can be used to test data consistency of the model. This helps the modeler to detect more errors.

An important element of any LPL models is the *expression*. The various items, such as numbers, sets, coefficients, variables, and results of other expressions, can be combined in algebraic or logical expressions. The usual algebraic and logical operators are available (Figure 2-1).

+ - not # IN	unary operators
^	power
* / %	product, division, modulo
+ -	addition and subtraction
= <> < > <= >=	relational operators
and	logical AND
or -->	logical OR, and implication
, (or |)	enumeration operator

Figure 2-1

Every expression returns one or several numerical values depending on whether the expression is indexed or not.

Examples:

```
COEF a := 4;                    "an assignment of a constant, a contains now
4"
COEF b := 2*a;                  "an expression returns a single value to b"
COEF c(i) := b*i; "c contains a list of values, assigns in this single
statement"
COEF d(i,j) := if(i<>j, i*j , 0);   "assigns values to a matrix d, the
diagonal
                                     is filled with 0, all other with i*j"
COEF e :=  sin(34) , log(1000) , max(34,67) , 45/67*976 , 1<2 , b , b/56 , y;
              "assigns a list of expressions to e, e is now a vector of
values"
COEF f(i,j|i<>j) := i*j;    "defines a matrix where all elements are i*j
                             except the diagonal elements which are non-
existent"
```

The same is true for logical expressions which return 1 for TRUE and 0 for FALSE. Furthermore, several functions and index-operators are integrated in LPL (Figure 2.2).

the functions are:						
max	min	abs	ceil	floor	trunc	sin
cos	log	sqrt	rnd	rndn	if	exp
the index-operators are:						
sum	prod	smin	smax	pmax	pmin	
exist	forall	col	row			

Figure 2-2

Complex and powerful expressions can be built with LPL. Let's give a non-trivial example: Define two sets *s* and *t*, the first of 11 elements and the second of 5 elements out of the first one. The CHECK statement tests whether all elements in *t* are also in *s* (which is not the case in this example).

```
SET s    := / 1:11 /;
    t    := / 2:5,12 /;
CHECK This(t) : t IN s ;
```

Now let's construct two two-dimensional tables x and y, such that the first is a sparse subtable of the Cartesian product *s*x*s*, and the second is defined over the Cartesian product of *s* and *t*. The sparsity of the first one is defined by the formula:

i<7 and (j=i+1 or j=i+2 or j=i+5).

The LPL syntax to define such tables is:

```
COEF x(i=s,j=s | i<7 and (j=i+1 or j=i+2 or j=i+5));
COEF y(s,i=t | i IN s);
```

We fill both tables just with the value 1:

```
x(s,s) := 1;
y(s,t) := 1;
```

and finally, the tables are printed on the screen

```
PRINT TO '' x:1 [11,11]; y:1;
```

The two tables you will see on the screen are

```
X(S,S)                                              Y(S,T)
     1  2  3  4  5  6  7  8  9 10 11                     2  3  4  5 12
1    -  1  1  -  -  1  -  -  -  -  -                1    1  1  1  1  -
2    -  -  1  1  -  -  1  -  -  -  -                2    1  1  1  1  -
3    -  -  -  1  1  -  -  1  -  -  -                3    1  1  1  1  -
4    -  -  -  -  1  1  -  -  1  -  -                4    1  1  1  1  -
5    -  -  -  -  -  1  1  -  -  1  -                5    1  1  1  1  -
6    -  -  -  -  -  -  1  1  -  -  1                6    1  1  1  1  -
7    -  -  -  -  -  -  -  -  -  -  -                7    1  1  1  1  -
8    -  -  -  -  -  -  -  -  -  -  -                8    1  1  1  1  -
9    -  -  -  -  -  -  -  -  -  -  -                9    1  1  1  1  -
10   -  -  -  -  -  -  -  -  -  -  -                10   1  1  1  1  -
11   -  -  -  -  -  -  -  -  -  -  -                11   1  1  1  1  -
```

Note that both tables are really stored as sparse tables. Many more useful examples will be shown in this paper.

A MAXIMIZE and MINIMIZE statement writes the objective function and all MODEL entities into a file (the RES-file), then the MPS-file is produced automatically, an external solver is called and the variables are assigned with the solution (if any).

An LPL model also may contain various *compiler directives*. They are special comments beginning with the two characters *{$*. So, e.g., the string *{$I filename}* within the LPL model causes the indicated file to be interpolated. *{$I* directives may be nested. *{$R}* causes the resetting of the computer's random generator to some initial value. *{$C}* at the beginning of the model turns the case-sensitivity switch on, so that lower and upper case letter are distinct by the LPL compiler. Several other directives are defined which are of minor importance in this context. More information and the entire syntax of LPL is found in the Reference Manual (Hürlimann 1992).

LPL was developed to formulate LP models close to a mathematical notation. The general LP model containing *i* rows and *j* columns, the *A* coefficient matrix, the *b* right hand side, the cost vector *c*, the *X* variable vector, and *R* rows can be formulated as

```
PROGRAM  General_LP_Model_in_LPL_syntax;
SET  i; j;                                      { ------------}
COEF A(i,j);  b(i);  c(j);                      { This is the }
VAR  X(j);                                      { whole model }
MODEL    R(i):  sum(j)  A[i,j]*X[j] <= b[i];    { structure!  }
MAXIMIZE obj_function:  sum(j)  c[j]*X[j];      { ------------}
```

The data again would be read from a data-file or they may also be produced by the LPL internal random generator (if one needs to produce random LPs). Note that the structure above declares a whole LP model in LPL. In most practical cases the model is not available in this form. But it shows that any LP model can be formulated using LPL. All one needs - to get a model instance - is to fill the three data tables *A*, *b*, and *c*.

However, it turns out that the LPL language may be used for many other model-types. The objective of the next sections is to show several applications.

3 Matrix calculation using LPL

Matrix calculation and manipulation with LPL is very convenient and easy, because LPL allows one to define multi-dimensional tables in a sparse manner. The declaration

```
SET i; j;
COEF a(i,j);
```

declares two indices and a two-dimensional numerical table. Different manipulations and calculations are now explored using LPL.

1. The following expressions give some basic examples of how matrixes can be manipulated:

```
COEF A(i1=i,i2=i | i1=i2) := 1          " this defines a unity matrix " ;
COEF B(i,j | i>=j)                      " this declares an upper triangular
matrix";
COEF D(i,j) := A(i,j) + B(i,j)          " this defines a matrix addition " ;
COEF E(i,k) := SUM(j) A(i,j) * C(j,k);  " matrix multiplication "
COEF F(i,j) := RND(10,20) ;             " a matrix filled with random numbers
                                          uniformly distributed between 10
and 20 "
```

2. Transpose a matrix (or a table): The task of transposing a table seems to be easy: just interchange rows and columns. As soon as a user needs to do that job manually, he or she might realize that it is not so easy at all. A quite complex program must be written to do the job. Even in EXCEL or another spreadsheet package, one needs to write a macro to do the job, supposing the data are in the spreadsheet format. But suppose the data are on a file 'IN.TXT' with the following layout

```
               GK          ZK          EI          AK          MB
P110       8.0000      8.0000      8.0000    100.0000    100.0000
P240       8.0000     10.0000      5.0000    100.0000    100.0000
P260       8.0000      8.0000      8.0000    100.0000    100.0000
P280       8.0000      8.0000      5.0000    100.0000    100.0000
P300       4.0000      8.0000      5.0000    100.0000      4.0000
P310       8.0000      4.0000      4.0000    100.0000      8.0000
P320       4.0000      4.0000      4.0000    100.0000      8.0000
P350       8.0000     10.0000      5.0000    100.0000    100.0000
P360       8.0000     10.0000      5.0000    100.0000    100.0000
P380       8.0000      8.0000      8.0000    100.0000      8.0000
P400       8.0000      8.0000      8.0000    100.0000    100.0000
P430       8.0000      8.0000      8.0000    100.0000    100.0000
```

To produce another file which contains the transposed table using LPL, simply write the following LPL program!

```
SET i; j;
COEF a(i,j);
READ FROM 'IN.TXT' : COL(j) j , ROW(i) ( i , COL(j)  a[i,j] ) ;
COEF b(j,i) := a[i,j];
PRINT TO 'OUT.TXT'  b;
END
```

Call the LPL compiler and the solution in the 'OUT.TXT' file is then

```
             P110        P240        P260        P280        P300        P310
P320
GK         8.0000      8.0000      8.0000      8.0000      4.0000      8.0000
4.0000
ZK         8.0000     10.0000      8.0000      8.0000      8.0000      4.0000
4.0000
EI         8.0000      5.0000      8.0000      5.0000      5.0000      4.0000
4.0000
AK       100.0000    100.0000    100.0000    100.0000    100.0000    100.0000
100.0000
MB       100.0000    100.0000    100.0000    100.0000      4.0000      8.0000
8.0000
```

```
          P350       P360       P380       P400       P430
GK      8.0000     8.0000     8.0000     8.0000     8.0000
ZK     10.0000    10.0000     8.0000     8.0000     8.0000
EI      5.0000     5.0000     8.0000     8.0000     8.0000
AK    100.0000   100.0000   100.0000   100.0000   100.0000
MB    100.0000   100.0000     8.0000   100.0000   100.0000
```

The PRINT statement breaks the block automatically into subblocks of seven columns. If the subblocks should contain more columns, simply indicate it within the Print statement as follows:

```
PRINT TO 'OUT.TXT'  b [50,20];  "in blocks of 50 rows and 20 columns"
```

3. Solving a linear set of equations: Suppose we get the following linear system of equations:

$$Ax = b$$

The system can be solved, if the matrix *A* is non-singular, using *LU* decomposition: Suppose we are able to write the matrix A as a product of two matrices

$$LU = A$$

where *L* is lower triangular and U is upper triangular. There is an example with a 4x4 matrix:

$$\begin{pmatrix} l_{11} & 0 & 0 & 0 \\ l_{21} & l_{22} & 0 & 0 \\ l_{31} & l_{32} & l_{33} & 0 \\ l_{41} & l_{42} & l_{43} & l_{44} \end{pmatrix} \cdot \begin{pmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$$

Then we may use the decomposition to solve the linear set

$$Ax = (LU)x = L(Ux) = b$$

by first solving the vector y such that using the trivial procedure of *forward substitution*

$$Ly = b \qquad (1)$$

and then solving using *backward substitution* (PRESS al. [1989], p.39ff)

$$Ux = y \qquad (2)$$

One single instruction in LPL is needed to do the *LU* decomposition (supposing A is nonsingular and has no zeroes on the diagonal). Here it is:

```
COEF  a(i,j) := if( i<=j, a[i,j] - SUM(k=i|k<i) a[i,k]*a[k,j] ,
                         (a[i,j] - SUM(k=j|k<j) a[i,k]*a[k,j])/a[j,j] );
```

a(i,j) contains now both triangular matrices, where l_{ii}=1 in the following format:

$$\begin{pmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ l_{21} & u_{22} & u_{23} & u_{24} \\ l_{31} & l_{32} & u_{33} & u_{34} \\ l_{41} & l_{42} & l_{43} & u_{44} \end{pmatrix}$$

Example: Given the following 4x4 matrix *A(i,j)*, the next LPL program produces the two matrices *L* and *U*.

```
SET i := /1:4/;  j := /1:4/;
COEF  a(i,j) := [ 5 3 -1 0 , 0 6 -2 3 , 2 0 4 1 , -3 3 -3 5];
  c(i,j) := a;  " make a copy of the original matrix "
  a(i,j) := if( i<=j, a- SUM(k=i|k<i) a[i,k]*a[k,j] ,
                     (a- SUM(k=j|k<j) a[i,k]*a[k,j])/a[j,j] );
  L(i,j|j<=i) := if(i=j,1,a);
  U(i,j|j>=i) := a;
  mul(i,j) := sum(k=j) L[i,k]*U[k,j];   "check the result by multiplying"
PRINT  c; a; L; U; mul;
```

The print outputs *U* and *L* as following

```
COEF U(I,J)                                  COEF L(I,J)
         1        2        3        4              1        2        3
4
1      5.00     3.00    -1.00     0.00      1   1.00        -        -
-
2         -     6.00    -2.00     3.00      2   0.00     1.00        -
-
3         -        -     4.00     1.60      3   0.40    -0.20     1.00
-
4         -        -        -     3.40      4  -0.60     0.80    -0.50
1.00
```

Of course, no diagonal element within the matrix must be zero, otherwise a pivoting must be executed. This is not done by the procedure above!

4. Calculate the determinant of the matrix: A single statement is needed to calculate the determinant if the upper triangular matrix *U* is given:

```
COEF  det := PROD(i) U[i,i];
```

If over- or underflow in the calculation occurs, the formula using logarithms might be used as follows

```
COEF  det_Log := SUM(i) log(U[i,i]);
```

The *LU* decomposition, as defined above, supposes a non-singular matrix and no zeroes on the diagonal. If neither condition is fulfilled, the LPL program will abort with an error message. Of course a 'real' decomposition of a big matrix needs some further manipulation as scaling and implicit pivoting, which is not included in the simple program above. Scaling is needed to minimize the accumulated rounding errors and implicit pivoting is needed to produce row permutation on the matrix. (see also CHVATAL 1983, p. 84ff).

5. Forward and backward substitution can be executed using the following instructions

```
COEF y(i) := b[i] - SUM(j | j<i) a[i,j]*y[j] ;  "forward substitution"

COEF x(!i) := y[i]/a[i,i] - SUM(j | j>i) a[i,j]*x[j] ;  {i:= N...1} "no LPL
syntax!"
```

Backward substitution needs to run sequentially through *i* from *N* down to 1. This is not yet implemented within the LPL compiler. But the syntax *(!i)* may indicate a possible syntax of how this might be done.

6. A simple LPL program shows how the language might be used to calculate recursive structure and to store the results in a table. The first 50 Fibonacci numbers can be computed with the instructions

```
SET l := / L1:L50 /;
COEF  h(l) := / L1 1 , L2 2 /;
      h(l) := if(l>2 , h[l-1]+h[l-2] , h[l]);
```

7. Solving a system of linear equation using an external solver.

The problem is to find the coefficients of a polynom of degree *i* supposing *i* points of the polygon in the Euclidean space are known. This problem can be formulated as a system of linear equations. The problem can be solved using an LP solver package by minimizing or maximizing any expression containing at least one variable.

```
SET   i;    "degree" of polygon"
VAR  a(i);  "unknown coefficients of the polynom"
COEF x(i);  "known x-coordinates of given points"
     y(i);  "known y-coordinates of given points"

MODEL
 r(k=i): sum(j=i) a[j]*x[k]^j = y;     "system of equations"
 rb(i): a > -100000;                   "allow negative coefficients"

"a model instance is"
SET   i = /1:4/;                    "polynom of degree 4"
COEF x(i) = [  3,  5, 8, 10];     "x-coordinates of given points"
     y(i) = [100, 70, 80, 0];     "y-coordinates of given points"

MAXIMIZE r_obj: a[1]; "optimize anything" "call the external solver"
PRINT TO '' a:10:5;             "print the results to the screen"
```

The following model defines exactly the same model instance with all equations explicitly mentioned.

```
VAR a; b; c; d;              "four unknown parameters"
COEF x1=  3; x2= 5; x3= 8; x4=10;
     y1=100; y2=70; y3=80; y4= 0;
MODEL
  r1: a*x1^4 +  b*x1^3 + c*x1^2 + d*x1  = y1;
  r2: a*x2^4 +  b*x2^3 + c*x2^2 + d*x2  = y2;
  r3: a*x3^4 +  b*x3^3 + c*x3^2 + d*x3  = y3;
  r4: a*x4^4 +  b*x4^3 + c*x4^2 + d*x4  = y4;
  rb1: a >-100000; rb2: b >-100000; rb3: c >-100000; rb4: d >-100000;
MAXIMIZE r_obj: a;
PRINT TO '' a:10:5; b:10:5; c:10:5; d:10:5;
```

8. A final example shows how to calculate the median of an unsorted array using LPL. The program was found in SCHWARTZ u.a [1986], Introduction in SETL. SETL is a powerful programming language which allows one to manipulate sets. The following program is written in SETL and returns the median of the set *s*.

```
program median;              $ This is SETL code
  read(s);
  if exists x in s | #{ y in s | y<x } = #{ y in s | y>x } then
    print('The median is:',x);
  else
    print('No median, the set has an even number of elements');
  end;
end program;
```

In LPL the same might be implemented using the following program:

```
PROGRAM median;               "This is LPL code"
  SET s; COEF a(s);
  READ : COL(s) (s , a);
    { or defined within the code :
      SET s= /1:9/; COEF a(s) = [ 13 11 45 0 -16 21 85 46 80 ];  }
  COEF m := EXIST(i IN s | SUM(j IN s | a[j]<a[i])1=SUM(j IN s|a[j]>a[i])1)
a;
  PRINT : if(m , 'The median is: ', a[m] , 'No median exists');
END.
```

This section has shown that LPL has many interesting applications for numerical manipulations besides LP modeling. Whether the implemented version of LPL is efficient enough to do this kind of calculations is another question. LPL gives a notational framework for formulating these problems. Many problems which must manipulate tables and multi-dimensional structures are candidates for LPL. Therefore, it is not surprising that LPL is, to some extents, a database management system; it allows one to select, to join, to project, or to combine them in an easy way. LPL is particularly apt for sparse table manipulation. The syntax is straightforward and easy to learn, since it copies the usual mathematical notation.

Let's explore what further can be done with LPL...

4 Decision Analysis using LPL

Decision analysis is a formal procedure to aid the decision maker in solving certain types of decision problems. Two types of such problems are formulated in this section: *multi-criteria decision problems* and *decision problems with uncertainty*.

First, we define a standard multi-criteria decision problem as follows. The decision maker is faced with

- a set of *m* alternatives a = { a_1, ... , a_m }
- a set of *n* criteria c = { c_1 ,... , c_n } together with a weight *w* for each criterion
- a set *r* of *m*x*n* outcomes or weights for every alternative/criterion pair

He or she has now to find the 'best' alternative using a certain decision rule.

LPL can be used to formulate such problems in an easy and convenient way.

To illustrate this problem situation, we use a simple model example that can be found in KOHLAS J. [1990], Heft 4:

> "The location of a school complex must be decided. There are five alternative locations. Different criteria enter the decision such as social environment, access (busses etc.), costs, and others. Suppose an analysis reveals that seven criteria are relevant to the decision. The weights of an alternative/criteria pair is given in the r(a,c) table. Furthermore, each criteria has an overall weight given the table w(c). Which alternative has to be chosen?"

The structure of the (general) decision model can be formulated as

```
SET  a;        "the set of alternatives"
     c;        "the set of criteria"
COEF r(a,c);   "the tuple list of outcomes"
     w(c);     "the overall weights of the criteria"
```

Note, there is no need to define the number of alternatives and criteria in advance; neither is it necessary to define the matrix. We call such a model part *model structure* in contrast to a *model instance* which contains all necessary data to proceed with a concrete, specific model. In the above model fragment, only the model structure is declared. LPL manages dynamically any model instance at a given time when the sets and the data table are filled. For convenience, however, we give a specific model instance for the mentioned example as follows

```
SET a := /a1:a5/                          "five alternatives" ;
    c := /c1:c7/                          "seven criteria" ;
COEF r(a,c) := [ 2  0  6  1 10 10  3      "weights"
                 8  5  4  3  0  7  8
                10  6  8 10  6  6 10
                 4 10  2  5  8  2  2
                 8  8 10  8  4  1  5 ] ;
     w(c) :=   [ 1  1  1  1  2  2  1 ]    "overall weight" ;
```

Different measurements can be defined using LPL syntax. Their names in boldface are extended by a short description and the LPL instruction. Note that all the following measurements are again data independent formulation, which means that they are equally valid for any model instance.

The **dominance matrix** returns zero or one for every alternative/alternative pair depending on whether an alternative dominates another one in all criteria. It can be defined as

```
COEF Domi(i=a,j=a|i<>j) := forall(k=a) (r[i,k] > r[j,k]) ;
```

The **dominated alternatives** are found using

```
COEF Dominated(i=a) := FORALL(j=a) (Domi[i,j]=0);
```

The **best alternative** using weighted evaluation is given by

```
COEF best_Alternative := PMAX(a) (SUM(c) r*w);
```

(The LPL syntax is not powerful enough to rank the alternatives using weighted evaluation, because no sort-algorithm is integrated.)

The **concordance matrix** compares the alternatives mutually and gives another indication of how to rank the alternatives. The matrix elements (concordance indices) vary in the range [0,1]. A high number near one means that the 'likelihood' of an alternative outdoing another alternative is high. It is defined as

```
COEF Conc(h=a,k=a|h<>k) := SUM(j=c | r[h,j] >= r[k,j]) w / (SUM(c) w) ;
```

The **discordance matrix** measures the 'likelihood' that an alternative is not outdone by another alternative. It is defined as

```
COEF Disc(h=a,k=a|h<>k) := SMAX(j=c | r[k,j] >= r[h,j])
                 (r[k,j]-r[h,j]) / (SMAX(a,c) r - SMIN(a,c) r);
```

The **threshold** and the **veto value** (Schwellenwert, Vetowert, see KOHLAS p.44) define limits for the concordance and the discordance. They must be defined by the decision maker.

```
COEF TV := 0.65 ;  VV := 0.8;
```

To decide which alternative outdoes which other ones using the corresponding threshold and veto value, one may build the following matrix

```
COEF OutDo(h=a,k=a|h<>k) := (Conc[h,k] >= TV) AND (Disc[h,k] <= VV) ;
```

The matrix OutDo returns zero or one for every alternative pair. One is returned, if the first alternative outdoes the second one.

The *decision problem with uncertainty* can be defined in a similar way as the multi-criteria problem: instead of criteria we have a set of states or events; each event occurs with a (given and known) probability; the outcome matrix is replaced by a pay-off matrix.

In the decision problem under uncertainty the decision maker is faced with

- a set of *m* alternatives a={a_1, ... , a_m}
- a set of *n* events e={e_1 ,... , e_n} , an event occurs with a known probability *p*.
- a set *r* of *m*x*n* pay-offs for every alternative/event pair

He or she has now to find the 'best' alternative using a certain decision rule.

The general model structure is defined as

```
SET o;          "a set of alternatives o"
    e;          "a set of events or states"
COEF po(o,e);   "a pay-off matrix" ;
     p(e);      "probability that an event occurs"
```

An illustrative example is chosen from RAVINDRAN, Chapter 4

> "The owner of a tennis shop must decide how many tennis shirts to order. He must order in batches of 100 pieces. An order of 100 costs \$1000, 200 cost \$900 per batch, and 300 cost \$850 per batch. The selling price is \$12 per piece. Left shirts at the end of the season are sold at \$6. The owner expects to sell 100, 150, or 200 shirts. If he understocks, there is a goodwill loss of \$0.5 per shirt.
>
> The owner has 3 alternatives: to buy 100, 200, or 300 shirts. Three events may occur: to sell 100, 150, or 200 shirts. The pay-off for every alternative/event pair is the net income calculated as:
>
> - demand*net_revenue-(cost-6)*(supply-demand), if supply>=demand
> - supply*net_revenue - 0.5*(demand-supply, if demand>supply
>
> The probabilities, that event demand=100, demand=150, or demand=200 occurs, is estimated as 0.5, 0.3, and 0.2 respectively."

The model instance is defined in LPL as

```
SET o := / o100, o200, o300 /   "alternatives" ; {orders}
    e := / d100 d150 d200 /     "events" ;       {demands}
COEF po(o,e) := [200 175 150 , 0 300 600 , -150 150 450] "pay-off" ;
     p(e) := [0.5  0.3  0.2]    "probability" ;
```

For each alternative we calculate the **expected value** as

```
COEF EV(o) := SUM(e) po*p ;
```

The highest expected value and the alternative with the highest expected value are now

```
COEF Max_EV   := SMAX(o) EV ;
COEF Max_EV_A := PMAX(o) EV ;
```

The **MaxiMin** and the **MaxiMax value** are calculated by

```
COEF MaxiMin := SMAX(e) (SMIN(o) po);
     MaxiMax := SMAX(o,e) po;
```

The alternatives containing the MaxiMin and MaxiMax values are

```
COEF MaxiMinA := trunc((EXIST(o,e) (po=MaxiMin)-1) / #e)+1;
     MaxiMaxA := trunc((EXIST(o,e) (po=MaxiMax)-1) / #e)+1;
```

Using the **Hurwicz rule** with the optimizing index Hw instead, we get the expected value for every alternative with the following formula

```
COEF Hw := 0.7;                      "must be defined by the decision maker"
     Exp_Hur(o) := Hw*SMIN(e) po + (1-Hw)*SMAX(e) po;
```

The **regret** (or opportunity loss) **matrix** represents the difference in value between what we obtain for a given alternative/event pair and what we could obtain, if we knew beforehand that the given event was in fact the true event. It is defined as

```
COEF RM(o,e) := (SMAX(o) po) - po ;
```

The **expected opportunity losses** for all alternatives are then

```
COEF EOL(o) := SUM(e) p*RM;
```

The alternative with the **lowest expected opportunity loss** is then

```
COEF Regret_A := PMIN(o) EOL;
```

Under the assumption of perfect information the expected value is

```
COEF EVPI := SMIN(o) EOL;
```

This is, in fact, the price a (rational) decision maker would pay to get perfect information.

Other applications may be implemented in LPL. The *decision tree problem* not shown in this paper might be an interesting model application.

5 Markov Chains using LPL

This section compiles techniques of modeling random processes - processes that evolve through time in an unpredictable manner. Discrete Markov processes can model many random processes. Excellent introductions to the theory of Markov chains can be found in KOHLAS J. [1977] Chapter 3 and in RAVINDRAN A. al. [1987] Chapter 6. Both examples presented are copied from the second reference.

LPL is particularly apt to model discrete Markov chains, because they need to manipulate matrices as well as to find the solution of linear equation systems. The second can be done by modeling a linear model system which is handed over to a simplex solver. LPL then reads the result back and goes on in its calculation.

The general model structure is simple, we need the number of states and the transition matrix as follows

```
SET s;       "the states"
COEF p(s,s); "the transition matrix"
```

A simple model instance may be described as

> "The owner of a limousine service operates between the airport, downtown, and the main bus station. After discharging a customer, the driver will stay at the same location to await another. He or she wants a model that tracks the movement of the car. Typical questions he or she wants to answer are:
>
> - Supposing the car is actually at the airport. What's the probability it will be back at the airport after three fares?
>
> - Over a long run, what fraction of trips are to the airport? etc."

Using LPL the model instance can be formulated as

```
"the states"
 SET s := / airport downtown station /;
"the transition matrix is"
 COEF p(s,s) := [0  0.5  0.5 , 0.333  0.333  0.334 , 0.667 0.333 0] ;
```

The transition diagram is shown in Figure 5.1.

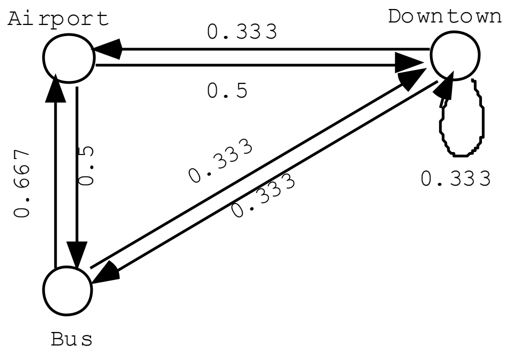


Figure 5-1

The sum of the row-transition probabilities must be one (or near one). The next statement checks this and returns an error if any one is far from one.

```
CHECK(i=s) : abs(SUM(j=s) p[i,j] - 1)<0.00001 ;
```

The **two-step transition** defines the probability of being in a certain state supposing of being in a certain state two-time-intervals before. It can be found by

```
COEF p2(i=s,j=s) := SUM(k=s) p[i,k]*p[k,j];
```

The **three-step transition** is now

```
COEF p3(i=s,j=s) := SUM(k=s) p2[i,k]*p[k,j];
```

The **four-step transition** is now

```
COEF p4(i=s,j=s) := SUM(k=s) p3[i,k]*p[k,j];
```

LPL does not (yet) have a LOOP statement as to calculate the n-th transition where n is given by a certain convergency criterion on the transitions. So the modeler must - in advance - define a number of transitions in order to get all transitions up to that number.

```
SET n := /1:10/;  "we define 10 transitions"
```

Now all transition probabilities can be found by

```
COEF pp(n,i=s,j=s) := if(n=1,p[i,j],SUM(k=s) pp[n-1,i,k]*p[k,j]);
```

The **zero-state probabilities** are given by (entered by the decision maker)

```
COEF sp(s) := [0.333  0.333  0.334];
```

Then the **n-state probability** returns the probability of being in a certain state after n-time-intervals. It can be calculated by

```
COEF spp(n,j=s) := SUM(i=s) sp[i]*pp[n,i,j];
```

The **first-passage** (or return) **probability** gives the probability of being in a certain state at time n *and not before*. It is given by

```
COEF f(l=n,i=s,j=s) := pp[l,i,j] - SUM(k=n|k<l) f[k,i,j]*pp[l-k,j,j];
```

It might be interesting to know the transition probability after n steps for n --> ∝. This is called the **steady-state probability**. Again, LPL cannot properly implement this by applying the matrix power method, since the break condition cannot be implemented. As before, one may define a specified number of states and hope the transition will converge within this number, a rather arbitrary assumption

```
COEF steady_state(i=s,j=s) := pp[10,i,j];
```

Fortunately, the steady state probability can also be found by the solution of a system of linear equations, which can be formulated in LPL as

```
VAR x(s);
MODEL R1(i=s): x[i] = SUM(j=s) p[j,i]*x[j];
      R2: SUM(s) x = 1;
MAXIMIZE R3: x[1];           "anything" "hand the problem over to a LP
solver"
COEF steady_state1(s) := x;   "copy the solution"
DELETE x; R1; R2; R3;         "get rid of all the model stuff again"
```

The **mean first passage time** might be calculated with the formula

```
COEF m_bad(i=s,j=s) := SUM(n) n*f[n,i,j]; "a bad approximation"
```

supposing that n --> ∝. This, however, is a bad approximation since n should grow infinitely to calculate the exact values. Using again a system of equations we get

```
VAR x(s,s);
MODEL R1(i=s,j=s) :  x[i,j] = 1+SUM(k=s|k<>j) p[i,k]*x[k,j];
MAXIMIZE R2: x(1,1);    "maximize anything" "solve the system by an external
solver"
COEF m1(i=s,j=s) := x;  "retain the result"
DELETE x; R1; R2;       "get rid of the rest"
```

The **mean recurrence time** (which is the same as m_{ii}) can also be obtained by

```
COEF m2(s) := 1/steady_state1;
```

Some results of the *Markov chains with absorbing states* may be added next. The last example had no absorbing states, therefore we define a new model instance with two transient states and two absorbing states as follows

```
SET s := /1:4/;
COEF p(s,s) := [ 0.25  0.25  0.5  0     ,
                 0.333 0.333 0    0.334 ,
                 0     0     1    0     ,
                 0     0     0    1       ];
```

The transition diagram (Ravindran [1987] p. 273 is illustrated by Figure 5.2

To find all **transient** and **absorbing** states use the following construct within LPL

```
SET absorbing(i=s) := EXIST(j=s) (p[i,j]=1);      " or := p[i,i]=1 "
    transient(s) := NOT absorbing;
```

They are simply defined as subsets of the set of all states.

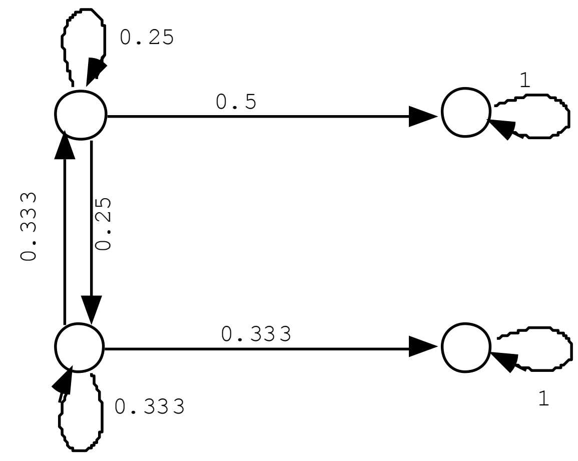


Figure 5-2

The **absorbing probability** for absorbing states is found by the following system of linear equations

```
VAR x(i=transient,j=absorbing);
MODEL R1(i=transient,j=absorbing): x[i,j] = p[i,j] + SUM(k=transient)
p[i,k]*x[k,j];
MAXIMIZE R2: x[exist(transient)1,exist(absorbing)1];
COEF ab(i=transient,j=absorbing) := x[i,j];
DELETE x; R1; R2;
```

Note that x is defined over the subsets of states. Therefore we cannot simply maximize the variable x[1,1] or any other fixed variable. Which variables are defined depends on which states are transient and which are absorbing. So maximizing anything (containing at least one variable) means that we must construct an existing variable. The expression *exist(transient)1* in LPL returns the first transient state.

The **mean number of times a transient state is occupied** is found by the linear model system

```
VAR x(i=transient,j=transient);
MODEL R1(i=transient,j=transient|i=j) : x[i,j] = 1+SUM(k=transient)
p[i,k]*x[k,i];
      R2(i=transient,j=transient|i<>j): x[i,j] =   SUM(k=transient)
p[i,k]*x[k,j];
MAXIMIZE R3: x[exist(transient)1,exist(transient)1];
COEF mn(i=transient,j=transient) := x;
DELETE x; R1; R2; R3;
```

Note that the last formulation uses two equations (R1 and R2). This is somewhat more readable than the equivalent formulation of the next equation R4

```
MODEL R4(i=transient,j=transient) :
  x[i,j] = if(i=j , 1+SUM(k=transient) p[i,k]*x[k,i] , SUM(k=transient)
p[i,k]*x[k,j]);
```

The **mean duration** (mean total number of transitions until absorption) is found by

```
COEF md(i=transient) := SUM(j=transient) mn[i,j];
```

LPL is useful for modeling discrete Markov chains, as the examples above show; especially because the modeler does not need to define a specific model instance beforehand. Any model instance can be treated with the mentioned formulas.

6 Network Modeling using LPL

It is well known that any transportation problem and many other network problems can be formulated as an LP model. The model could be coded in LPL and LPL could translate automatically to an MPS solver input file. The simplex would then be applied to solve the model. Realistic applications, however, might produce models too big to be handled with MPS and a standard simplex solver. Furthermore, an LP formulation implies some unnecessary problem transformations on an otherwise natural model formulation. It is easier to think about 'nodes' and 'arcs' than about 'variables' and 'restrictions' in network problems. Generally, specialized model generator tools and much more efficient solvers (as PNET or GENNET) are used to handle these models.

Forster [1988] presents a specialized modeling language (GNGEN) which allows the modeler to generate a network. GNGEN is entirely declarative. A simple example is presented to show this generator: Consider the problem of producing three goods in two plants shipped to three distribution centers. Factory 1 produces only products 1 and 2, factory 2 produces products 1 and 3. The third center can only be supplied from factory 2. Excess production in a period t may be stored at the factory. In GNGEN this problem can be formulated as (see Forster 1988)

```
NETWORK Production_Distribution;
NODESETS
  Supply        = ( Factory , Time ),
  Production  = ( Factory , Commodity [ Factory ] , Time),
  Distribution  = ( Center , Commodity [ Center ] , Time );

ARCS FROM s □ Supply TO p □ Production :
          s.Factory = p.Factory AND
          s.Time = p.Time
LABEL CoP [ p.Factory ] , 0 , INFINITE;

ARCS FROM p □ Production TO p' □ Production :
          p.Factory = p'.Factory  AND
          p.Commodity = p'.Commodity  AND
          s.Time+1 = p'.Time
LABEL 0 , 0 , INFINITE;

ARCS FROM p □ Production TO d □ Distribution :
          d.Center □ C [ p.Factory ]  AND
          p.Commodity = d.Commodity  AND
          p.Time = d.Time
```

```
LABEL f * Dist [ p.Factory , d.Center ] , 0 , INFINITE;

NODES  s  Supply  LABEL S [ s.Factory ]
NODES  d  Distribution
LABEL  - D [ d.Center , d.Commodity , d.Time ];
ENDNET;
```

The data are stored in the following notation

```
Time := 1..4;  Factory := 1,2;  Center := 1..3;

Commodity [ Factory ] := 1,2;  1,3;
Commodity [ Center ]  := 1;  2,3;  1;

CoP [ Factory ]  :=  150; 200;
C   [ Factory ]  :=  1,2;  1,2,3;

Dist [ Factory , Center ]  := 1200  1100   900
                               850  1050   520  ...
```

The language allows one to declare the network model in terms of nodes and arcs and their attributes. We may try to 'simulate' this form of model using LPL. In the SET statement one can define the nodes and in the COEF statement we specify the arcs as well as all the attributes. Hence, the LPL formulation of the last example would take the following form

```
SET
  Time ; Factory ; Center ; Product ;
  CommF ( Product , Factory ) ;
  CommD ( Product , Center ) ;
  Supply ( Factory , Time ) ;
  Production ( Factory , Product , Time | CommF) ;
  Distribution ( Center , Product , Time | CommD) ;

COEF
  Sup_Prod_Arc ( Supply , Production ) ;
  Prod_Prod_Arc ( Production , Production ) ;
  Prod_Dist_Arc ( Production , Distribution ) ;

  S ( Factory ) ;
  D ( Center , Product , Time | CommC) ;
  CoP ( Factory ) ;
  Dist ( Factory , Center ) ;
  f ;

(* data : a model instance is *)

SET
  Time := / 1:4 /; Factory := / 1,2 /; Center := / 1:3 /; Product := / 1:3 /;
  CommF ( Product , Factory ) := / 1 ( 1 2) , 2 (1 3) /;
  CommD ( Product , Center )  := / 1 1 , 2 (2 3) , 3 1 /;
  C ( Center , Factory )      := / 1 ( 1 2 ) , 2 ( 1 2 3 ) /;
  Supply ( Factory , Time ) := 1;
  Production ( Factory , Product , Time | CommF ) := 1;
  Distribution ( Center , Product , Time | CommD ) := 1;

COEF
  Sup_Prod_Arc ( Supply[f,t] , Production[f1,c,t1] ) := f=f1 AND t=t1;
  Prod_Prod_Arc ( Production[f,c,t] , Production[f1,c1,t1] ) :=
         f=f1 AND c=c1 AND t+1=t1;
  Prod_Dist_Arc ( Production[f,c,t] , Distribution[f1,c1,t1] ) :=
         f1 IN f AND c=c1 AND t=t1;

(* data : the model instance is *)
  S ( Factory ) := [ 180 210 ];  { ??? }
```

```
    D ( Center , Product , Time ) :=
               [ 100 100 100 100 , 110 110 110 110 , 110 110 110 110
                 100 100 100 100 , 110 110 110 110 , 120 120 120 120
                 100 100 100 100 , 100 100 100 100 , 100 100 100 100 ]; { ???
  }
    CoP ( Factory ) := [ 150 200 ];
    Dist ( Factory,Center ) := [  1200  1100   900
                                   850  1050   520 ];
    f := 1; { ??? }
```

A PRINT statement in LPL can then be used to produce the necessary SHARE input file (Barr al. [1973]) for the network solver. (The SHARE format is an industrial standard for network problems like the MPS format for LP models).

Another general representation of network models is the GN (Generalized Network) graph (Glover al. [1978]. It is defined as a directed graph G(V,E). Every vertex V_i {i=1..v} contains two attributes: a name n_i and a capacity b_i. Every vertex has in-degree of at most two. Every edge E_j contains four attributes: a capacity c_j, an upper u_j and lower bound l_j and a multiplier a_j. The general network may be formulated in LPL as

```
  SET nodes;                         "the vertices"
  SET edges(nodes,nodes);            "the edge list"
  STRING n(nodes);                   "the node name"
  COEF b(nodes);                     "the node capacity"
  COEF c(edges);                     "the edge capacity"
  COEF upper(edges);                 "the upper bounds of the edge"
  COEF lower(edges);                 "the lower bounds of the edge"
  COEF a(edges) [0,.];               "the multiplier (always positive)"
  VAR  flows(edges);                 "the unknown flows"

  "the model can then be formulated as"
  MODEL
    subject_to(i=nodes) : sum(j=nodes) a[i,j]*flows[i,j] = b[i];
    bounds(edges): lower <= flows <= upper;
  MINIMIZE flow: sum(edges) c*flows;
```

The restriction *subject_to* balances the flows at each node. Lower and upper bound restrictions are added by the restrictions *bounds*, the node capacity is an additional net inflow on the node. In pure networks only the source has a net positive inflow, whereas the sink has a net outflow.

A model instance was given by Glover [1978], which can be formulated in mathematical notation as follows

$$\begin{array}{lllllllll}
\text{Min :} & 1x_{12} & +5x_{13} & +3x_{23} & +1x_{24} & -4x_{32} & -9x_{34} & & \\
\text{Subject\_to:} & & & & & & & & \\
 & -1x_{12} & -1x_{13} & & & & & = & -5 \\
 & +2x_{12} & & -1x_{23} & -1x_{24} & +1/3x_{32} & & = & 0 \\
 & & 1/2x_{13} & +1x_{23} & & -1x_{32} & -1x_{34} & = & 0 \\
 & & & & -1/5x_{24} & & +3x_{34} & = & 10
\end{array}$$

Represented as a GN graph, the model has the form of Figure 6-1.

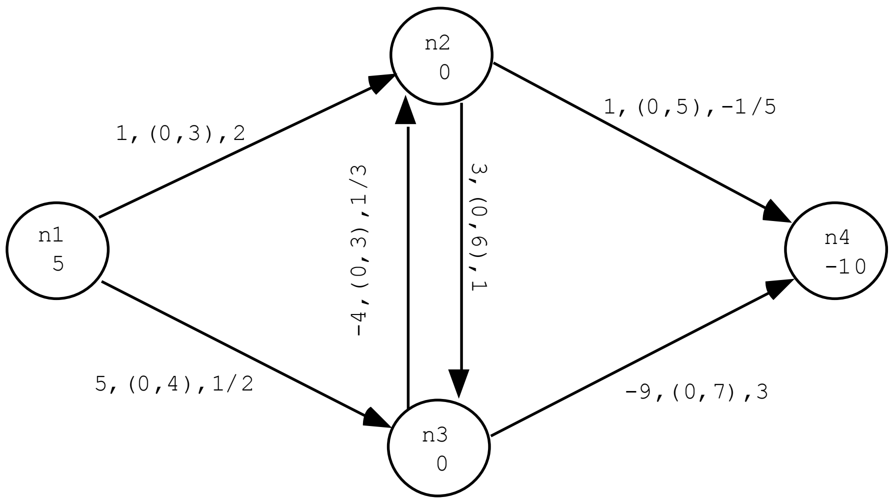


Figure 6-1

Implementing this model example into LPL syntax gives the following model instantiation

```
SET nodes  := /1:4/;
SET edges(nodes,nodes) := / 1 (2 3) , 2 (3 4) , 3 (2 4) /;
STRING n(nodes)  := [ 'n1' , 'n2' , 'n3' , 'n4' ];
COEF b(nodes) := [ 5 , 0 , 0 , 10 ];
COEF c,lower,upper,a (edges):=
   /1 2     1    0    3    2
    1 3     5    0    4    0.5
    2 3     3    0    6    1
    2 4     1    0    5    -0.2
    3 2    -4    0    3    0.333
    3 4    -9    0    7    3     /;
```

This example shows that any network model represented as GN graph can be directly translated into an LPL model - only the data change the model instance, the model structure is invariant and independent from the instance.

Here again, an appropriate PRINT statement within LPL would write the needed SHARE format and a modified solver interface instruction - the {$P directive - call a network solver instead of the standard simplex solver. Hence, there is no need to translate this model into an MPS file to solve it.

7 Game Theory and Linear Programming

The finite, two-person, zero-sum, one-move game may be defined as follows (Brickman [89], p. 98):

There are two players called One and Two. Player One (Two) has m (n) many choices of play, each makes a choice in turn, and the combination of two choices determines a payoff a_{ij}. All the m×n payoffs are known to each player. What one player wins, the other loses, which means that the payoff for player One is a_{ij} and for player Two it is $-a_{ij}$.

If player One chooses her move i with frequencies x_i, and player two makes his move j with frequencies y_j, then the average payoff for player One is given by

E(One) = $\sum_{i=1}^{m}(\sum_{j=1}^{n} a_{ij} x_i y_j)$This is also the expected payoff of player One. The expected payoff of player Two is therefore -E(one). Note that

$\sum_{i=1}^{m} x_i = 1$ and $\sum_{j=1}^{n} y_j = 1$must hold, since a player must make some move in turn. The problem for player One (Two) is now to find all x_i (y_j), such that his expected payoff is maximized (minimized). This is called the *optimal strategy* for player One (Two).

Von Neuman [1928] proved the famous *MiniMax Theorem*:

> There exist a x-vector and a y-vector such that the expected payoff of both players is equal. They are obtained, if both players choose their optimal strategies.

The problem of the two-person, zero-sum game can easily be translated into a specific LP problem. LPL is used as a notational framework.

The basic objects of the game can be defined using LPL:

```
SET
  i;        "a set of choices for player One"
  j;        "a set of choices for player Two"
COEF
  A(i,j);  "the payoff matrix"
VAR
  x(i);     "the (unkown) frequencies of a choice for player One"
  y(j);     "the (unkown) frequencies of a choice for player Two"
```

The expected payoff per round (EV) can be calculated with

```
COEF
  EV := SUM(i,j) A*x*y;
```

The optimal strategy for player One can be found by solving the following LP

```
MODEL SUM(i) x = 1;
MAXIMIZE ExpectedValue : MIN{j} (SUM(i) A*x);   ! This is LPL3.9 Syntax !
```

Note that the maximizing function is translated by LPL (3.9) automatically into the following piece of model

```
VAR lambda;
MODEL subject_to(j): lambda <= SUM(i) A*x;
MAXIMIZE ExpectedValue : lambda;
```

The optimal strategy for player Two can be found by solving the following LP

```
MODEL SUM(j) y = 1;
MINIMIZE ExpectedValue : MAX{i} (SUM(j) A*y);  // LPL 3.9
```

Note that the minimizing function is translated by LPL automatically into the following piece of model

```
VAR lambda;
MODEL subject_to(i): lambda >= SUM(j) A*x;
MAXIMIZE ExpectedValue : lambda;
```

One can easily see that the two LP models are dual to each other. Therefore, the optimal value must be the same in both LPs. Thus, the MiniMax Theorem follows easily from the Duality Theorem. The optimal value *ExpectedValue* is called the *value of the game*. A game is only fair if this value is zero, or if - at each round - player One pays *ExpectedValue* units to player Two. (It may be noted that for arbitrary a_{ij} payoff values the primal LP may be unbounded, to avoid this, simply add a constant to any a_{ij} such that all $a_{ij}>=0$. This does only change the optimal value. See Brickman [89] p.108).

A condensed overview of the game theory can be found in Brickman [89] chap 6 or Chvatal [83] chap15. Many game examples are exposed in Thie [1988].

8 Multi-Objective LP's (goal programming)

Multiple Criteria Decision Making (MCDM) refers to making decisions in the presence of multiple, usually conflicting, objectives or goals. We are looking for an optimal solution. But what is meant by an optimal solution in the case of conflicting goals? Since it is - in general - impossible to maximize all objectives simultaneously, we need a definition of what is meant by 'optimal'. One definition of 'optimal' may be to search all non-dominated solutions (for a definition see Zionts p.139, the non-dominated solutions in Figure 7-1 are on the line segments A-B-C-D). Unfortunately, for many problems the number of non-dominated alternatives is still too large.

Another way to define an optimal decision is *one that maximizes a decision maker's utility* (or satisfaction) [Zionts 1988]. Different methods exist, which try to 'find' the utility function.

The general MCDM problem - sometimes also called the vector optimization problem - may be formulated using mathematical notation as

$$\begin{array}{ll} \text{Maximize} & F(x) \\ \text{subject to} & G(x) \le 0 \end{array}$$

where x is the vector of decision variables, *F(x)* is the vector of objectives to be maximized, and *G(x)* are the constraints that define the feasible space. The multiple objective linear programming problem (MOLP) is defined as the

MCDM problem, where *F* and *G* are all linear functions and the problem becomes

$$\begin{array}{l}\text{Maximize } Cx \\ \text{subject to } Ax \le b\end{array}$$

where *C* is a transposed numerical vector, *b* is a numerical vector, and *A* is a numerical matrix. It is rather easy to formulate this problem in LPL: the objective functions can be put in a list of single objective functions separated by a comma as in

```
EQUATION F : -x1 + 2*x2  ,  2*x1 - x2 ;
```

The comma operator makes *F* indexed. The objective function may now be defined as

```
MAXIMIZE MyMax : F[1] + F[2] ;    "no weights"
```

or using weights for the different single objectives

```
MAXIMIZE MyMax : w1*F[1] + w2*F[2] ;    "weights are w(i)"
```

This is the simplest case where we compound the different single objectives together into a final weighted single overall objective function. Another possibility is to use multi-stage LPs, which means, that the LP is solved several times with different objective functions. Example 3 below exposes this choice. Three examples are given below.

Example: suppose the model has the following two maximizing functions where x_1 and x_2 are the variables, F_1 and F_2 are the objective-names (problem of Figure 7-1)

$$\text{maximize } F_1\text{: } -x_1 + 2x_2 \quad \text{maximize } F_2\text{: } 2x_1 - \ x_2$$

The different methods to formulate this with LPL using no weights are

```
{ 1.}     MAXIMIZE F1_and_F2 : -x1 + 2*x2  +  2*x1 - x2 ;

{ 2.}     EQUATION F1 :  -x1 + 2*x2 ;  F2 : 2*x1 -   x2 ;
          MAXIMIZE  F1_and_F2 : F1 + F2 ;

{ 3.}     EQUATION  F :  -x1 + 2*x2 , 2*x1 -   x2 ;
          MAXIMIZE  F1_and_F2 : F[1] + F[2] ;
```

Any of the formulations above will produce one single objective function: they are simply added together. If the modeler wants to weight them, he or she has only to add a weight factor to any of the functions, otherwise every objective has weight one. This solution - as can be shown - yields non-dominated solutions.

The method just described is the simplest way to treat the MOLP with LPL: the different objectives are just added together with weight 1 and the resulting single objective will be optimized.

There are other methods, however, and they are explained now. (For the notation we suppose, that *F* is declared in LPL as *EQUATION F(i)*. Note that *F* can be used within other restrictions as identifier; in this case it is simply expanded.)

1. A naive method is to set levels on all objectives: The modeler sets levels *d* on each objective and looks then for a feasible solution of the problem. LPL has no difficulty in formulating this kind of problem, it just formulates all objectives as restrictions *obj* and adds any objective function

```
COEF d(i);
EQUATION F(i);
MODEL
   obj(i) : F[i] = d[i];
   subject_to :           " .. all other restrictions .. "
   maximize MyObj :       " .. anything .. "
```

The result may be a non-feasible solution, a dominated solution (a point inside the hull), or a non-dominated solution (a point on the hull).

2. Setting minimal levels on all but one objective: *One* objective is maximized and the others must be fulfilled at a certain degree *d*. In LPL this is formulated as

```
MODEL
   maximize MyObj : F[1];
   obj(i|i>1) : F[i] >= d[i];   "all F except the first one"
   subject_to : ..... all other restrictions ....
```

The result will be a non-dominated solution.

3. Finding all efficient extreme-point (non-dominated) solutions. Except for two objectives, which can be solved with parametric programming, this is not workable, because there are far too many solutions to analyze. Except for very few objectives this is not workable either in practice, since the decision maker must choose between all the solutions.

4. Using weights to combine objective functions (as already mentioned): The idea is that every objective gets a weight to build one single composite objective

to be maximized. The difficulty from the modelers point of view lies in specifying the weights.

5. Goal programming: Lower *L* and upper *U* bounds are specified on each objective as well as a positive and a negative deviation variable *posSlack* and *negSlack* together with a weight. The sum of all weighted deviations is minimized. This can be formulated in LPL explicitly as (variant 1)

```
COEF L(i); U(i);                "lower and upper bounds of the objectives"
     posW(i); negW(i);          "weights for the positive and negative slack"
VAR posSlack(i); negSlack(i);   "i is the number of objectives"
MODEL
   posObj(i) : F[i] - posSlack[i] <= U[i];
   negObj(i) : F[i] + negSlack[i] <= L[i];
   subject_to : .... all other restrictions .... ;
MINIMIZE MyOby : sum(i) (posW*posSlack + negW*negSlack) ;
```

Instead of minimizing the sum of the weighted deviations one may minimize the maximum deviation. We remove the objective MyObj from variant 1 and replace it by the following definitions (variant 2)

```
MINIMIZE MyNewObj:  max{i} (posW*posSlack,negW*negSlack)  // LPL 3.9
```

Note again that LPL (3.9) automatically translates this minimizing function into the following piece of model

```
VAR z;        "a new variable is added"
MODEL
   posMax(i) : posW*posSlack <= z;
   negMax(i) : negW*negSlack <= z;
MINIMIZE MyNewOby : z;
```

If the decision maker gives an aspiration level *Asp* instead of lower and upper bound on each objective, then we might use still another formulation, which minimizes the weighted sum of all deviations (variant 3)

```
COEF Asp(i);                    "aspiration levels of the objectives"
     posW(i); negW(i);          "weights for the positive and negative slack"
VAR posSlack(i); negSlack(i);   "i is the number of objectives"
MODEL
   posObj(i) : F[i] - posSlack[i] + negSlack[i] = Asp[i];
   subject_to : .... all other restrictions .... ;
MINIMIZE MyOby : sum(i) (posW*posSlack + negW*negSlack) ;
```

Another variant is to employ preemptive priorities instead of the numerical weights (see example 3).

Goal programming does not have to say much about the choice of weights. The specification of weights is left to the user. Different mostly interactive methods have been developed to support the modeler in choosing the right weights. I do not treat them here. Despite this drawback, goal programming is widely used, because of its simple and transparent approach. Unfortunately, all three variants above - written in LPL syntax - obscure the model structure by introducing explicitly new slack variables as well as new restrictions, which have nothing to do with the model structure itself. Fortunately, all three variants can be formulated in a much easier way than the formulations above. LPL supports goal programming using the operator ~. An objective with an aspiration level (variant 3) can be formulated as:

```
MODEL MyGoal          :  F =~ Asp;
      MyManyGoals(i)  :  Fx[i] =~ Aspx[i];
```

where *F* and *Fx* are objective functions; *Asp* and *Aspx* are the corresponding aspiration levels. The operator =~ means *approximately equal*. The first defines one goal, whereas the second defines i goals. LPL produces automatically two slack variables with the same name as the goal-name preceded by the letter 'p' for the positive slack and a letter 'n' for negative slacks. The last two goal definitions will produce the same model as

```
VAR pMygoal; nMyGoal;
    pMyManyGoal(i); nMyManiGoal(i);
MODEL
    MyGoal         :  F - pMyGoal + nMyGoal = Asp;
    MyManyGoal(i)  :  Fx[i] - pMyManyGoal[i] + nMyManyGoal[i] = Asp;
```

Variant 1 (lower and upper bound instead of an aspiration level) can be written in LPL using the range syntax. The model fragment

```
MODEL
     MyGoal          :  L <=~ F <=~ U;
     MyManyGoals(i)  :  Lx[i] <=~ Fx[i] <=~ Ux[i];
```

will produce the same model piece as

```
VAR pMygoal; nMyGoal;
    pMyManyGoal(i); nMyManiGoal(i);
MODEL
    MyGoalP : F - pMyGoal <= U;
    MyGoalN : F + nMyGoal >= L;
    MyManyGoalP(i) : Fx[i] - pMyManyGoal[i] <= Ux[i];
    MyManyGoalN(i) : Fx[i] + nMyManyGoal[i] >= Lx[i];
```

Variant 2 cannot be formulated otherwise than indicated above.

8.1 Examples

Example 1 [Zionts 88] : two objectives (without aspiration level, either upper or lower bound). Formulate this model in LPL as (model represented by Figure 7-1)

```
VAR x1; x2;
EQUATION
   F1: -x1 + 2*x2;
   F2: 2*x1 - x2;
MODEL
   subject_to1 : x1 <= 4;
```

```
      subject_to2 : x2 <= 4;
      subject_to3 : x1 + x2 <= 7;
      subject_to4 : -x1 + x2 <= 3;
      subject_to5 : x1 - x2 <= 3;
   MAXIMIZE Obj : F1 + F2;
```

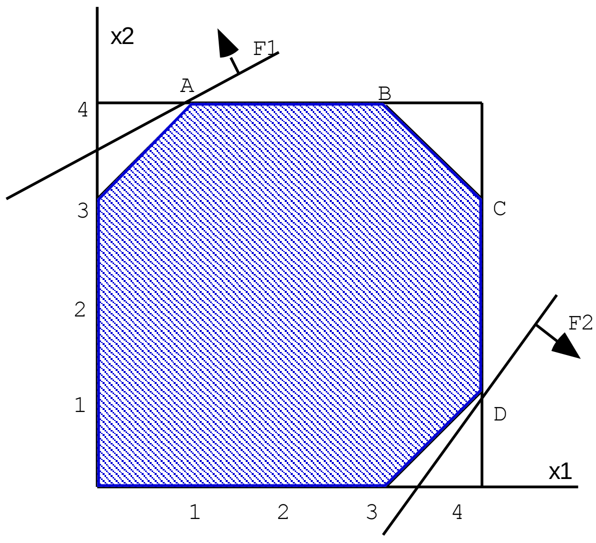


Figure 7-1

The resulting maximizing function will be

```
   MAXIMIZE obj: x1 + x2 ;  { same as : F1 + F2 }
```

The optimal solution is on the line segment BC.

Example 2 [Zionts 88] : three objective functions together with lower bounds but no upper bounds are given together with two restrictions. The bounds are (66, 80, 75).

```
   VAR x1; x2; x3; x4;
   MODEL
     F1 : 3*x1 + x2 + 2*x3 + x4 >=~ 66;
     F2 :   x1 - x2 + 2*x3 + 4x4 >=~ 80;
     F3 :  -x1 + 5*x2 + x3 + 2*x4 >=~ 75;
     subject_to1 : 2*x1 + x2 + 4*x3 + 3*x4 <= 60
     subject_to2 : 3*x1 + 4*x2 + x3 + 2*x4 <= 60
   MINIMIZE obj : nF1 + nF2 + nF3;   "minimize the (negative) goal deviations"
```

Note that LPL generates only negative slack variables, because only the lower bounds are given. The resulting model is therefore the same as

```
   VAR x1; x2; x3; x4;
       nF1; nF2; nF3;
   MODEL
      F1 : 3*x1 + x2 + 2*x3 + x4 + nF1 >= 66;
      F2 :   x1 - x2 + 2*x3 + 4x4 + nF2 >= 80;
      F3 :  -x1 + 5*x2 + x3 + 2*x4 + nF3 >= 75;
      subject_to1 : 2*x1 + x2 + 4*x3 + 3*x4 <= 60
      subject_to2 : 3*x1 + 4*x2 + x3 + 2*x4 <= 60
   MINIMIZE obj : nF1 + nF2 + nF3;   "minimize the goal deviations"
```

Using variant 2 of goal programming for the same model, it would be natural to replace simply the minimizing function by

```
MINIMIZE obj : max(nF1 , nF2 , nF3);  "minimize the maximum goal deviation"
//LPL 3.9
```

LPL produces automatically the following piece of model

```
VAR z;
MODEL
   MINIMIZE : z;   "minimize the maximum goal deviation"
   F1N : nF1 <= z;
   F2N : nF2 <= z;
   F3N : nF3 <= z;
```

Example 3 [Beilby al. 1983] : An academic library acquisitions allocation: The library staff is charged with allocating academic and research library budgets for books and periodicals. Their task is to satisfy the needs and interests of groups which are, ipso facto, competing for library resources. Goal programming was successfully used for this kind of problems by Beilby al.[1983]. The model contains only goals and no further restrictions. The minimizing function is a preemptive priority. The model can be formulated in LPL as follows. Unfortunately, I was not able to reproduce exactly the same solution as reported in the reference.

Note that this formulation has several nice features:

1. The slack variables are produced by the LPL compiler automatically as soon as a relational operator together with the symbol ~ is used, otherwise the modeler would have to declare 18 slack variables explicitly.

2. The solver may be called several times (each time the MINIMIZE keyword is used.

3. The variables may be transformed into simple data entities between the calls of the solvers. The first optimization function is A[1], which is minimized to make the (slack)variables *nG1* and *pG1a* as small as possible (here zero). Then they are fixed simply by transforming them into COEFs (they are no longer variables for the next optimizations). The whole model is produced and a new minimizing function *A[2]* is used.

4. A weighted sum of the different goals may be used instead, but this is not recommended for this problem. LPL allows the use of both of them. Up to the modeler to choose!

```
(* LIBRARY.LPL : book acquisition in a library (GOAL programming)
   REF:. BEILBY M.H., MOTT T.H. [1983], Academic Library Acquisitions
   Allocation based on Multiple Collection Development Goals,
   Comput. & Ops. Res. Vol. 10, No. 4, pp 335-343. *)

SET
  i  "types"  :=   / Books , Periodicals / ;
  j  "book and periodocal types"  :=
    / Humanities , Social_Science , Sciences , Education , Interdisciplinary
/;
COEF
  avrCost(i,j)  :=   "average cost of a title"
   [ 13.01 , 12.55 , 19.32 , 10.53 , 13.10 , 37.64 , 23.12 , 114.0 , 24.18 ,
35.0 ];
  CostPp(j)  :=  "projected average cost for periodicals for five years"
   [ 75.71 , 37.92 , 229.29 , 55.32 , 58.98 ] ;
  cir(i,j)  :=   "circulation data"
   [ 0.028 , 0.028 , 0.033 , 0.066 , 0.004 , 0.033 , 0.099 , 0.075 , 0.209 ,
0.037];
  e(i,j) :=   "enrollment percentage of total book and periods enrollment
titles"
   [ 0.278 , 0.273 , 0.186 , 0.263 , 0.0    ,  0.278 , 0.273 , 0.186 , 0.263
, 0 ];
  pr(j) :=  "research productivity"
   [ 0.175 , 0.450 , 0.600 , 0.210 , 0.000 ] ;
  ret(j) :=  "retrospective data"
   [ 0.252 , 0.475 , 0.042 , 0.185 , 0.046 ] ;
  low(i,j)  :=   "minimum percent level of acquisition"
   [ 0.15 , 0.25 , 0.05 , 0.15 , 0.05  ,  0.03 , 0.10 , 0.03 , 0.05 , 0.10 ];
  up(i,j)  :=   "maximum percent level of acquisition"
   [ 0.25 , 0.35 , 0.10 , 0.25 , 0.10  ,  0.08 , 0.15 , 0.05 , 0.10 , 0.15 ];

VAR x(i,j);

MODEL
  "Goal 1: Acquire at least 7500 titles but no more than 10500 titles"
  G1 :  SUM(i,j)  x  =~ 7500;
  G1a:  SUM(i,j)  x  =~ 10500;

  "Goal 2: Do not exceed total acquisitions budget of 300000"
  G2:  SUM(i,j)  avrCost*x  =~ 300000;

  "Goal 3: Limit periodical expenditures to 60% of the total acquisitions
   expenditures"
  G3:  SUM(j) avrCost[2,j]*x[2,j] =~ 0.6*( SUM(j) avrCost[2,j]*x[2,j] );

  "Goal 4: Limit periodical acquisitions to the level which can be supported
for a
   five-year period"
  G4:  SUM(j) CostPp*x[2,j]  =~ 193261 ;

  "Goal 5: Allocate titles by subject according to circulation data (550 is
an
   arbitrary unit)"
  G5:  SUM(i,j) cir*x =~ 550;

  "Goal 6: Allocate titles by subject according to enrollment data"
  G6(i,j | e):  x[i,j] - e*(SUM(j | e) x[i,j]) =~ 0;

  "Goal 7: Limit research acquisitions to 15% of the total acquisitions and
allocate
   on the basis of departmental research productivity"
  G7:  SUM(j | pr) pr*x[1,j] - 0.15*(SUM(i,j) x) =~ 0 ;

  "Goal 8: Limit retrospective acquisitions to 5% of total acquisitions and
allocate
   on the basis of retrospective subject needs"
  G8 :  SUM(j) ret*x[1,j] - 0.05*(SUM(i,j) x)  =~ 0;

  "Goal 9: Meet desired subject acquisition ranges"
  G9(i,j) : x - low*(SUM(i,j) x)  =~ 0;
  G9a(i,j) : x - up*(SUM(i,j) x)  =~ 0;

"preemptive priorities A[1]-A[8]"
EQUATION A :  nG1+pG1a, pG2, SUM(i,j) (nG9+pG9a), nG5,
              SUM(i,j|e) if(j<=2,pG6,nG6), pG7+pG8, pG3+pG4, nG2+nG3+nG4;

                        {--- preemptive priorities as multi-stage minimizing }
MINIMIZE  M: A[1];  COEF nG1; pG1a;
MINIMIZE  M: A[2];  COEF pG2;
MINIMIZE  M: A[3];  COEF nG9; pG9a;
MINIMIZE  M: A[4];  COEF nG5;
MINIMIZE  M: A[5];  COEF pG6; nG6;
MINIMIZE  M: A[6];  COEF pG7; pG8;
MINIMIZE  M: A[7];  COEF pG3; pG4;
MINIMIZE  M: A[8];

PRINT /VAR/;
END

{--- or minimize a weighted sum  !! is not recommended !! }
MINIMIZE M :  10000000*A[1]
              +1000000*A[2]
               +100000*A[3]
                +10000*A[4]
                 +1000*A[5]
                  +100*A[6]
                   +10*A[7]
                    +1*A[8] ;
END.
```

In this section some applications of multi-objective linear programming have been shown and how they can be implemented using LPL. LPL handles more than one objective function in the following way

- if several minimizing or maximizing functions are used within an LPL program, write them down as EQUATION entities and define a weight.

- if goal are used, add a ~ symbol and consider the goal as simple MODEL entity. LPL introduces automatically positive and negative slacks for every goal depending on whether upper bounds, lower bounds or aspiration levels are defined: a lower bound introduces a positive slack variable, an upper bound adds a negative slack variable, and an aspiration level adds both of them. This simple construct helps the modeler to concentrate on his model structure, rather than on the different modeling techniques.

9 Conclusion

In this paper several applications of the LPL language have been exposed, which have not been reported until now. It shows, that LPL can already be used for many model structures besides the original purpose of modeling LP models.

The LPL compiler has been implemented using TURBO PASCAL from Borland Inc. under MS/DOS. A version in ANSI C is also implemented (but not available). The PASCAL version is actually available from the author for a small reward. [bitnet: HURLIMANN@FRUNI51, fax: (41) 37 21 96 70].

References

General

GEOFFRION A.M. [1989], Computer-Based Modeling Environments, European Journal of Operational Research, 41(1989), pp. 33-45.

HÄTTENSCHWILER P, [1992], Konzepte & Instrumente für die mathematische Modellierung, Institute for Automation and Operations Research, University of Fribourg, Switzerland, to appear.

HÄTTENSCHWILER P., KOHLAS J., [1989], Wissensbasierte Systeme auf der Grundlage linearer Modelle - Werkzeuge und Anwendungen, Output, Vol.18/12, December, Goldach, Switzerland.

HÜRLIMANN T. [1991], Linear Modeling Tools (Talk at the 11th European Congress on Operational Research RWTH Aachen, July 16-19), 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, September 1991 (updated February 1992), Fribourg.

Matrix Calculation

PRESS W.H., FLANNERY B.P., TEUKOLSKY S.A., VETTERLING W.T. [1989], Numerical Recipes in PASCAL, Cambridge University Press, Cambridge.

CHVATAL V. [1983], Linear Programming, W.H. Freeman and Comp, New York.

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

Decision Analysis, Markov Chains and Game Theory

BRICKMAN L., [1989, Mathematical Introduction to Linear Programming and Game Theory, Springer-Verlag.

HÜRLIMANN T. [1990], 6 Begleithefte mit Programmen zu den 6 Lehrheften (Kohlas [1989]) in Operations Research, AKAD Verlag, Zürich.

KOHLAS J., [1977], Stochastische Methoden des Operations Research, Teubner, Stuttgart.

KOHLAS J., [1989], 6 Lehrhefte in Operations Research, AKAD Verlag, Zürich.

RAVINDRAN A., PHILLIPS D.T., SOLBERG J.J., [1987], Operations Research Principles and Practice, 2nd edition, Wiley, New York, (Chapter 4)

THIE P.R., [1988], An Introduction to Linear Programming and Game Theory, John Wiley & Sons, New York.

von Neumann J., [1928], Zur Theorie der Gesellschaftsspiele, Mathematische Annalen 100, p.295-320.

Goal programming & Multi-objectives

BEILBY M.H., MOTT T.H. [1983], Academic Library Acquisitions Allocation based on Multiple Collection Development Goals, Comput. & Ops. Res. Vol. 10, No. 4, pp 335-343.

CHANKONG V., HAIMES Y.Y. [1983], Multiobjective Decision Making, Theory and Methodology, North-Holland,New York.

ZIONTS S. [1988], Multiple Criteria Mathematical Programming: An Updated Overview and Several Approaches, in: Mathematical Models for Decision Support, Mitra G. (ed.), NATO ASI Series F, Vol 48, Springer Verlag.

Networks and Graphs

BARR R., GLOVER F., KLINGMAN D., STUTZ J. [1973], Technical Documentation NETGEN, March 1973.

FORSTER M. [1988], A General Network Generator, in:Mathematical Models for Decision Support, ed. G. Mitra, NATO ASI Series F. Vol:48, p. 207-215.

GLOVER F, HULTZ J., KLINGMAN D., STUTZ J. [1987], Generalized Networks: A Fundamental Computer-Based Planning Tool, Management Science Vol 24(12), August 1978, p.1209-1220.

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

3 LPL: The Concept of Units

T. Hürlimann

Key-words: Model Building, Modeling Language design, units.

Abstract: A method is proposed to incorporate a system of measurement units into the modeling language LPL. Benefits of keeping track of units in a modeling system can be to trap more errors, to enhance reliability and readability of the model, or to scale the data automatically. A short introduction on manipulation rules with measurement units is given; the full syntax of this LPL extension and several examples are presented; and finally some implementation aspects are rehashed.

Stichworte: Modellierung, Modelliersprache, MIP Programmierung.

Zusammenfassung: Es wird eine Methode vorgestellt, die erlaubt Masseinheiten in the Modelliersprache LPL einzubinden. Die Vorteile sind, dass mehr syntaktische Fehler automatisch entdeckt werden können, dass das Modell u.U. an Lesbarkeit gewinnt, und dass Daten automatisch skaliert werden. Es wird eine kurze Einführung in die Algebra der Einheitenrechnung gegeben; sodann breiten wir die volle Syntax in LPL aus und geben einige Beispiele, schliesslich werden noch einige Implementationsaspekte berührt.

1 Introduction

"...but at the system level I don't think you'll ever be fully confident that somebody somewhere hasn't punched in feet instead of miles in the computer program." (John Pike).

The crew of the space shuttle Discovery in 1985 fed the number 10023 into the onboard guidance system. The number was correct, but it was supplied in feet to the crew, whereas the system expected a unit in nautical miles. Thus, the space shuttle flew upside-down over Maui.

Most quantities in models are measured in units (dollar, hour, meter etc.). As the introductionary example shows, there may be a big difference between a quantity being given in one or another unit. Experiences in Operations Research teaching in our Institute revealed that one of the most frequent errors of students in modeling was the inconsistency of measurement units.

In physics and other scientific, technical and commercial applications, using units of measure has a long tradition. It increases the reliability and the readability of calculations. Thus, it is not surprising that there have been different proposals to include this concept into the programming languages design [Karr 1978, House 1983, Männer 1986, Dreiheller 1986, Baldwin 1987]. The concept of unit seems to fit quite well into the much broader concept of data type used in different, strong-typed programming languages such as PASCAL and ADA. One may think that units can simply be included by a strict form of name equivalence, whereby two types should be considered different, even if they are based on the same basic type. For example the two types TIME and DISTANCE in a PASCAL type declaration

```
TYPE
  TIME = REAL;
  DISTANCE = REAL;
```

should be different types (in ADA, they are indeed different types). Unfortunately, this does not solve our problem. In unit calculations, we need derived units such as speed which is defined as distance per time. Combining different units (types) in the same expression would produce a type error. Hence, the type concept must be somewhat extended to allow full integration of unit calculations.

Our concern here is not programming languages design but modeling language design. As far as I can see, there has been one proposal to incorporate a unit system into an executable modeling language. Bradley [1987] specifies a type calculus for an extended dimensional system for modeling languages. Each model entity as well as input and output are assigned a type that consists of its concepts, quantities and units of measurement.

A modeling language should in some way keep track of physical and other units, e.g. volts, horsepower, hertz, dollar etc. Explicit mention of units can enhance readability of model, can give the compiler additional checking power which may reduce the number of syntax errors, and can let the compiler do the job of automatic unit conversion and scaling.

We do not consider how units are related to the real world. The question of which units are elementary will lead to interminable discussions unless the following resolution is taken: The model language should not commit itself to any particular set of units. The modeler must be free to define his own unit system. It is true, however, that there exist international standards and conventions. In physics, e.g., all quantities seem to be reducible to the following seven basic units:

length	[m]	meter
mass	[g]	gram
time	[s]	second
electric current	[A]	Ampere
thermodynamic temperature	[K]	Kelvin
amount of substance	[mol]	mole
luminous intensity	[cd]	Candela.

All other physical units are derived from these seven basic units. For example, Joule [J] is defined as [$1000m^2gs^{-2}$]. But in other domains these units are different. A modeling language should be independent of any specific application domain. The modeler must even be free to make use of the unit system or not. It should be entirely optional. He or she alone should be responsible a good (or bad) model formulation. The modeling language designer, on the other hand, must offer powerful checking possibilities. Whether they are used or not to formulate a concrete model should be decided by the modeler. Sometimes it may be a mere matter of taste whether units should be incorporated into a model. In a model, for example, where we know that all time quantities are given in weeks and every money quantity is given in dollars, there is no need to add this information explicitly to the model.

2 Definitions

Units are something which is carried along in calculations. They act like numeric identifiers in numerical expressions, obeying commutativity and associativity laws. In the expression

$$g = 9.81*(m/sec)/sec = 32*feet/sec^2$$

the '9.81' can be regarded as being multiplied by the expression 'm/sec^2'. Thus, when we want to calculate how far an object falls in 3 seconds using $d=1/2gt^2$, we may write

$$d = 1/2 * 9.81 * (m/sec^2) * (3\ sec)^2 = \frac{9.81*m*9*sec^2}{2*sec^2} = 44.145\ m$$

or using feet instead of meters we get

$$d = 1/2 * 32 * (m/sec^2) * (3\ sec)^2 = \frac{32*m*9*sec^2}{2*sec^2} = 144\ feet$$

Ordinary algebra rules can be used to calculate expressions using units. Units are useful because we may define relationships among them; for example we may write expressions like

1 m = 3.261 feet

12 inches = 1 foot

watts = volts amps

Meters can be converted into feet using such expressions and vice versa. Units are also useful, because they disallow operations which do not make sense; for example, '2 m + 3 watts' is an addition which does not make sense, because m and watts are not commensurable. But the calculation '1 m + 1 foot' is something we may accept; it is 4.261 feet or 1.306m, depending on whether we express the result in feet or meters. I do not give a complete overview here of unit calculus, which can be found in any physics course; only some basic concepts on unit calculus are introduced.

Since we are only interested in how units are manipulated, we may abstract those properties of interest to us. We start with a finite set of (user-defined) *elementary units*, which we think of simply as symbols with suggestive names: feet, dollar, week. A *derived unit* A is *commensurable* to another unit B, if and only if A/B is a dimensionless quantity; in this case there must exist a well defined *commensurateness relationship* between the units A and B. A quantity is *dimensionless*, if the division A/B can be reduced to a simple number. A commensurateness relationship consists of the derived unit name, an assignment symbol, and a *unit expression*:

```
Derived_unit_name = Unit_expression
```

Example

feet	feet is an elementary unit
inch = feet/12	inch is a derived unit, the

Inch is commensurable with feet, since $\frac{\text{inch}}{\text{feet}} = \frac{\text{feet/12}}{\text{feet}} = 1/12$ and no unit is left in the last expression. Also *compound unit* definition may be useful, which are derived units, compound of more than one unit. An example is

watts = volts*amps

Note that the units 'watts' and the units 'volts/amps*amps^2' are commensurable using the relationship above since

$$\frac{\text{watts}}{\text{volts/amps*amps}^2} = \frac{\text{volts*amps}}{\text{volts/amps*amps}^2} = 1$$

can be reduced to the number 1.

A *unit expression* has a very limited syntax: Only other unit names - elementary or derived ones - together with numeric literals and the two operators * and / are allowed. Principally, the power operator ^ might be allowed providing the exponent is rational (see syntax rules in [House 1983]). We will disallow the power operator as well.

Units calculations depend on the operators; they must be handled differently. We present now a short summary on rules of expression checking

Assignment: The units of the left hand side must be commensurable to the unit of the right hand side. If they are not commensurable, an error occurs. The left hand side must be multiplied by the corresponding scale factor.

Addition, Subtraction, and relational operators: The units of the operandi must be commensurable. If they are not commensurable, an error occurs. Each side must be scaled before the comparison or operation can take place. For boolean operators, this can give rise to real arithmetic precision errors like

```
IF  ( inch = 1*Feet/12 )  THEN ...
```

Will the test be true or false? Unfortunately, this may depend on the real arithmetic calculation of the specific computer. Unit calculation must be carefully applied in these cases. The resulting unit of all operators is commensurable to one of its operandi.

Multiplication, division: Any units may be multiplied or divided. The resulting unit is the multiplication or division of the units of its operandi.

Exponentiation: The exponent must be rational and its unit must be commensurable to 1. Any other unit calculation is nonsense. The resulting unit is obtained by exponentiating the unit of the basis by the exponent.

Build-in functions: Each function belongs to one of the three following groups: 1. The argument(s) must be dimensionless (sin, log, and all other transcendental functions); 2. The argument(s) may have any units, the function does not care about units (ceil, floor, rnd, rndn, etc.); 3. Functions which change the resulting units (sqrt, sqr).

Index operators: They have their corresponding meaning of their dual operators, e.g. SUM is treated in the same way as the addition operator.

Parameter passing in user defined functions: If the parameters are passed by value, then this can be treated in the same way as assignment: formal and actual parameters must be commensurable and the scaling is executed each time. The same is true for the return value(s). If the parameters are passed by reference, then, of course, formal and actual parameters must also be commensurable, but the conversion is more complicated. The following solution can be found in Karr [1978].

We consider a function F with the formal parameter X with units u. Suppose the i-th call on F is done with the actual parameters Z_i with unit u_i. The conversion factor of u_i to u is c_i. Then the following transformations are made:

- change the declaration of F(X:u) by F(X,C) - both parameters with unit 1.
- replace the i-th call on $F(Z_i:u_i)$ by $F(ZZ_i,c_i)$, where ZZ_i is Z_i but dimensionless.
- replace all occurrences of X within F by X*C.
- replace all assignments X = ... within F by X = (...)/C
- handle all calls within F of a function G(X) - supposing G was declared to have a reference parameter - as follows: replace the call G(X) by $G(X,c*c_0)$

where c_0 is the conversion factor between the unit of the formal parameter of G and u. This works even in recursive function calls, and in particular when G=F.

The transformation given above works in any case. Of course, there are better solutions in specific cases; if we know that Z_i is not used as global with the function call, then the Z may be multiplied by C just after entering F and divided by C just before exiting F.. If Z is not modified within F, then C*Z may be copied to a temporary variable and Z itself never be used thereafter.

There are several 'pathological units' which cannot be manipulated in the way we explained above. A linear conversion,e.g., $u_2 = a*u_1 + b$ of two units u_1 and u_2 are not allowed, when $b \neq 0$. The conversion between Celsius and Fahrenheit is of this form. Other examples include measurements with an arbitrary reference point such as AD. The above rules cannot be applied to this 'unit'; for example, we cannot write 1990AD - 1986AD = 4AD, but the subtraction makes sense, since we may interpret it as 4 YEARS. Even a more pathological case is decibel. "The justification for designing constructs especially for them is doubtful". [Karr], because they are rarely used in calculations.

3 Syntax and Applications in LPL

LPL needs to be extended by a UNIT statement, where the elementary units are declared as unique identifiers, and the derived units are defined through their commensurateness relationship. Furthermore, the declaration of any numeric entity (COEFs, VARs, and EQUATIONs) may be extended by a unit declaration. Numeric literals within expressions must also be extensible by indication of the units. Finally, the input and output statements must be extended by an optional unit declaration.

The entire unit concept in LPL is based on the following syntax elements:

First, any unit used within the model must be defined by the modeler through a (new) UNIT statement which starts with the reserved word UNIT followed by a unit declaration. The complete syntax of the UNIT statement is

```
UnitStatement ::= UNIT unit ';'
unit ::= ElementaryUnit  |  DerivedUnit
ElementaryUnit ::= UnitName
DerivedUnit ::= UnitName '=' UnitExpression
UnitExpression ::= UnitFactor { UnitOperator UnitFactor }
UnitFactor ::= number  |  '(' UnitExpression ')'  |  UnitName
UnitOperator := '*'  |  '/'
UnitName ::= identifier
```

Note that this syntax allows also dimensionless units as in

```
UNIT
  gram;             "a elementary unit gram"
  Mile; inch; year; "three other elementary units"
  kilo = 1000;      "derived and dimensionless"
  kg = kilo*gram;   "derived and compound, but commensurable to gram"
  SquareMile = Mile*Mile;         "a compound unit"
  acceleration = inch/year/year;  "another one"
```

Other examples can be found in the Model *EnergyLoss* below. In contrast to other identifiers, unit names cannot be redefined within the entire model. Derived units must be declared <u>and</u> assigned at the same place; they cannot be declared, and assigned later on. This excludes cyclic unit declarations (e.g. meter is defined in feet or vice versa but not both). This excludes also *inconsistent unit* declarations. But this does not exclude redundant unit declarations. A *redundant unit* is a unit which can be derived by another commensurateness relationship. As an example, we have

```
UNIT
  inches;             "an elementary unit"
  feet = 12*inches;   "a derived unit"
  Fuss = 12*inches;   "'Fuss' is the same as feet and redundant"
```

Redundant units are useful as any other 'redundant' identifier: the same entity has different names. There is nothing wrong with this.

Units might be predefined in separate files as any other part of the model, and they can then be included in any model just by adding the include statement. The modeler does not need to define them each time he uses the same units. Commensurable units are linked by a commensurateness relationship in the UNIT statement as explained above.

Secondly, any numeric entity, such as data, variables, or constraints, may be extended with a unit option simply by adding the reserved word IN, UNIT, or both just after the declaration together with a unit name or a unit expression. An numeric entity without this option is dimensionless. The syntax is

```
Options ::= INTEGER | DEFAULT Number | '[' Range ']' | UnitOption
UnitOption ::= UnitOp UnitSpec
UnitOp ::= IN [ UNIT ] | UNIT [ IN ]
UnitSpec ::= UnitName | UnitExpression
```

Examples are

```
COEF  weight IN kg;                      "unit of weight is kg"
VAR   cars INTEGER [.,100]  IN 1000;     "unit of cars is in 1000"
EQUATION r UNIT 12*kg;                   "r is in dozen of kilogram"
```

Third, unit expressions are allowed in four different parts of the LPL program:

1. in the UNIT statement as left hand side of an assignment.

```
   UNIT derived_unitname = <UnitExpression>;
```

2. in the COEF, VAR, and EQUATION (MODEL) statement, to declare the identifier of a specific unit. (Note that SETs do not have units.)

```
   COEF MyData IN <UnitExpression>;
   VAR MyVar UNIT <UnitExpression>;
   EQUATION MyCons IN UNIT <UnitExpression>;
```

3. In a regular expression when using a numeric literal. The numeric literal is extended by a bracket unit expression as in

```
   ....+  600[hour/day] - ....
```

which means that the numeric literal 600 is to be read in hour/day.

4. in the input and output statements (READ and PRINT statement). The unit option must be indicated just before the semicolon. The read or printed data are automatically converted to the specified unit

```
   { .... see model example below .... }
   PRINT profit IN DailyIncome; Robots IN piece/day; HC;
   READ HC IN hour/day;
```

Normally, inputs and outputs are measured in units declared by the entity. Thus, if HC is declared in hour/week, then *READ HC;* or *PRINT HC;* are automatically supposed to be given in hour/week. But in the input/output statement this measure can be overwritten, providing the units are commensurable, otherwise an error occurs.

We give now an entire model example to illustrate the use of units. To see the contrast, first an LPL formulation without any use of unit is given.

A firm produces i={1..10} different types of robots. Three production steps must be carried out: a) Production of the components, which takes HC_i hours for each robot i, with a total capacity of 3500 hours per week; b) Mounting (capacity=920 hours per day - say a week has 5 days) taking HM_i hours for each robot i and c) Testing (capacity=3000 hours per week) taking HT_i hours for robot i. The selling prices for each robot i is $price_i$. There are already some robots of each type ordered. How many robots of each type can be produced per week, if the firm wants to maximize the selling profit?

The model may be formulated using LPL as

```
   (***********  LPL formulation without using units  ***********)
```

```
SET  i = / 1:10 /;   { ten robots types }
VAR  Robots(i);
COEF HC(i) = [ 5 5 4 5 6 5 7 8 4 7 ];
     HM(i) = [ 4 8 5 6 4 8 7 6 5 3 ];
     HT(i) = [ 6 2 4 6 3 4 5 2 5 3 ];
     Ordered(i)  = [ 20 15 7 6 5 8 9 8 7 5 ];
     Price(i)  = [ 300 200 100 50 50 100 200 100 400 200 ];
MODEL
  Components: SUM(i) HC(i)*Robots(i) <=  3500;
  Mounting:   SUM(i) HM(i)*Robots(i) <=  4800;
  Testing:    SUM(i) HT(i)*Robots(i) <=  3000;
  Order(i):   Robots(i)              >=  Ordered(i);
MAXIMIZE profit: SUM(i) Price(i)*Robots(i);
PRINT profit; Robots; HC;
END
```

Note that the modeler must be careful to translate all capacity measures to the same unit. So 920 hours per day must be translated manually to 4800 hours per week. If, by neglect, the manual translation was not carried out, the solution of this model is quite different from the 'true' solution (see below):

```
{ ---------------- faulty solution with 920 as right hand side --------------
- }
PROFIT = 49770.0000

ROBOTS(I)
      1       2       3       4       5       6       7       8       9
10
   20.0    15.0     7.0     6.0     5.0     8.0     9.0     8.0    87.8
5.0
```

The same model is now formulated in LPL using units for all entities. Note that no manual translation of units is necessary for any expression.

```
(***********  LPL formulation with units  ***********)
UNIT
  piece;                     "quantity unit (numbers)"
  dollar;                    "money unit"
  d100 = 100*dollar;         "another compatible money unit in $100"
  hour;                      "time unit"
  day=8*hour;                "another time unit"
  week=5*day;                "still another time unit"
  DailyIncome = dollar/day;  "a compound unit"

SET  i = / 1:10 /;           "10 different robots to produce"

VAR  Robots(i) IN piece/week;  "number of robots per week"

COEF HC(i) IN hour/piece = [ 5 5 4 5 6 5 7 8 4 7 ];
     HM(i) IN hour/piece = [ 4 8 5 6 4 8 7 6 5 3 ];
     HT(i) IN hour/piece = [ 6 2 4 6 3 4 5 2 5 3 ];
     Ordered(i) IN piece/week  = [ 20 15 7 6 5 8 9 8 7 5 ];
     Price(i)  IN d100/piece = [ 3 2 1 0.5 0.5 1 2 1 4 2 ];

EQUATION
  Components IN hour/week;
  Mounting IN hour/week;
  Testing IN hour/week;
  Order(i) IN piece/week;
  profit IN dollar/week;

MODEL
  Components: SUM(i) HC(i)*Robots(i) <=  3500[hour/week];
  Mounting:   SUM(i) HM(i)*Robots(i) <=  920[hour/day];
  Testing:    SUM(i) HT(i)*Robots(i) <=  3000[hour/week];
  Order(i):   Robots(i)              >=  Ordered(i);
```

```
MAXIMIZE profit: SUM(i) Price(i)*Robots(i);
PRINT profit; profit IN DailyIncome; Robots; Robots IN piece/day; HC;
END
```

The PRINT statement produces the following output (after the model has been correctly solved)

```
PROFIT IN DOLLAR/WEEK = 238332.3529

PROFIT IN DailyIncome = 47666.4704

ROBOTS(I) IN PIECE/WEEK
      1       2       3       4       5       6       7       8       9
10
   20.0   281.0     7.0     6.0     5.0     8.0     9.0     8.0   426.1
5.0

ROBOTS(I) IN PIECE/DAY
      1       2       3       4       5       6       7       8       9
10
    4.0    56.2     1.4     1.2     1.0     1.6     1.8     1.6    85.2
1.0

HC(I) IN HOUR/PIECE
      1       2       3       4       5       6       7       8       9
10
    5.0     5.0     4.0     5.0     6.0     5.0     7.0     8.0     4.0
7.0
```

Another example is the following LPL model which computes the energy distribution of monoenergetic particles after they have passed a thin foil. [Männer 1986]

```
PROGRAM EnergyLoss;
UNIT
  g;         "grams"             "basic units"
  cm;        "centimeters"
  sec;       "seconds"
  C;         "coulombs"

  k      = 1000;                 "dimensionless scale factor"
  M      = k*k;                  "still dimensionless"

  kg     = k*g;                  "kilograms"
  m      = 100*cm;               "meters"
  mu     = m/M;                  "microns"
  N      = kg*m/(sec*sec);       "newtons (force)"
  erg    = g*cm*cm/(sec*sec);    "ergs (work)"
  eV     = 1.602e-12*erg;        "electron volts"
  MeV    = M*eV;                 "mega electron volts"
  g_cm3  = g/(cm*cm*cm);         "volume mass density"
  C2_Nm2 = C*C/(N*m*m);          "dielectricity"

COEF
  { parameters }
  FinalEnergy, MeanFinalEnergy IN MeV  [0,1e3];
  ChargeParticle, ChargeTarget  [0,105];
  AtNrTarget  [0,260];
  DensityTarget IN g_cm3  [0,10];
  ThicknessTarget IN mu  [0,1e6];

  { result }
  EnergyDistribution;

  { constants }
```

```
    Pi                    = 3.14159;
    Epsilon0  IN C2_Nm2   = 8.85419e-19;
    e IN C                = 1.6021e-19;
    AMU IN g              = 1.660431e-24;
    k1                    = 1.33;

    { intermediate results }
    Alpha, I   IN MeV;
    n1  IN 1/(cm*cm*cm);

    { calculations }
    I     = 11.5 [eV] * ChargeTarget;
    n1    = DensityTarget/(AtNrTarget*AMU);
    Alpha = sqrt(1/(4*Pi*Epsilon0^2) * n1 * ChargeParticle^2 * e^4 *
            ChargeTarget * ThicknessTarget * (1+k*I/2*MeanFinalEnergy) *
            log(4*MeanFinalEnergy/I)));
    EnergyDistribution = 1[MeV] / (Alpha * sqrt(Pi)) *
                         EXP(-((FinalEnergy-MeanFinalEnergy)/Alpha)^2;
  END
```

Two errors have been detected in Männer's formulation of this problem using the LPL compiler. In his PASCAL source code presentation, he wrote the last two formula as

```
    Alpha = sqrt(1/(4*Pi*Epsilon0^2) * ChargeParticle^2 * e^4 *
            ChargeTarget * ThicknessTarget * (1+k*I/2*MeanFinalEnergy) *
            log(4*MeanFinalEnergy/I)));
    EnergyDistribution = 1 / (Alpha * sqrt(Pi)) *
                         EXP(-((FinalEnergy-MeanFinalEnergy)/Alpha)^2;
```

In the first line of this piece of code a factor 'n1' is missing and in the fourth line [MeV] is missing. Two errors which can easily be detected using units within the model!

An example, where the unit system in LPL cannot be used, is in temperature conversion between Celsius, Fahrenheit, and Kelvin as in the following code. The translation must be done by expressions.

```
  { faulty model : 'pathological units' are not allowed }
  UNIT
    Kelvin;
    Celsius=Kelvin-273.2;          { faulty definition }
    Fahrenheit=9/5*Kelvin-460;     { faulty definition }

  COEF temp IN Celsius = 100;
  PRINT temp IN Fahrenheit;        { would be nice ! }
  END
```

4 Implementation

It turns out that the implementation of units into the modeling language is quite easy. Say the number of elementary units is n and the number of derived units is m. Then we reserve space for the unit symbol table as follows:

- declare a matrix M of integer values with m rows and n columns
- declare a vector V of m reals

- initialize both data structures to zeroes
- for every commensurateness relationship (derived unit) fill a row with the corresponding exponents of the elementary units. Enter the conversion factor in the corresponding position of V.

Example

The first model above declares the following units

```
UNIT
  piece;                     "quantity unit (numbers)"
  dollar;                    "money unit"
  d100 = 100*dollar;         "another compatible money unit in $100"
  hour;                      "time unit"
  day=8*hour;                "another time unit"
  week=5*day;                "still another time unit"
  DailyIncome = dollar/day;  "a compound unit"
```

The corresponding unit symbol table looks like this

M[i,j]	piece	dollar	hour	V[i]
d100	0	1	0	100
day	0	0	1	5
week	0	0	1	40
DailyIncome	0	1	-1	1/8

To test if two derived units are commensurable is now very easy: all entries in the corresponding row must be equal and the conversion factor is the multiplication of the two corresponding V entries. Thus, the all manipulations with units in expressions can be reduced to completely mechanical procedures via linear algebra. Of course, the physical symbol table may be stored by a sparse matrix data structure to save space. This is also advantageous, because the maximal dimension of the table is not known to the compiler in advance.

5 Conclusion

In this paper, an extension of LPL has been proposed that allows one to incorporate units into the modeling language. Some basic ideas behind the unit calculus was discussed, a full syntax for LPL has been given. Two model examples illustrate all important points concerning units; and an implementation was suggested.

The extensions lead to a model where more errors can be detected by the compiler. This is the most important issue. A convenient effect is the automatic conversion of commensurable units; but, as I mentioned, this automatic conversion can lead to errors which may be difficult to detect because of real arithmetic round-offs. The advantages, however, are overwhelming, and unit should be used as much as possible in model creation. It enhances the readability and extends the documentation of a model.

Acknowledgements: This paper was mostly influenced and stimulated by the profound paper of Karr 1978.

References

BALDWIN G., Implementation of Physical Units, Sigplan Notices, Vol.22, No.8, August 87, p.45-50

BRADLEY G.H., CLEMENCE R.D., A Type Calculus for Executable Modeling Languages, IMA Journal of Mathematics in Management, 1(1987) p.177-191.

DREIHELLER A., MOERSCHBACHER M., MOHR B., Programming Pascal with Physical Units, Sigplan Notices, Vol.21, No.12, December 1986, p.114-123.

HOUSE R.T., A Proposal for an Extended Form of Type Checking of Expressions, The Computer Journal, Vol.26, No.4, 1983, p.366-374.

HÜRLIMANN T. [1990], Reference Manual for the LPL Modeling Language, Version 3.5, Institute for Automation and Operations Research, Working Paper No. 175, June, Fribourg.

KARR M, LOVEMAN D.B., Incorporation of Units into Programming Languages, Comm. of the ACM, May 87, Vol.21, No.5, pp.385-391.

MANKIN R., (Letter), Sigplan Notices, Vol.22, No.3, March 1987, p.13.

MÄNNER R., Strong Typing and Physical Units, Sigplan Notices, Vol.21, No.3, March 1986, p.11-20.

4 gLPS: a Graph-Based System for Linear Problems Modeling

Gérald COLLAUD *and Jacques* PASQUIER-BOLTUCK

Institute for Automation and Operations Research (IAUF),
University of Fribourg, CH - 1700 Fribourg, Switzerland
E-mail : COLLAUDG@CFRUNI51.BITNET

Abstract

The use of suitable tools is one of the cornerstones of every aspect of human activity. In mathematical modeling, graphical representation of complex systems - like circles and arrows to depict states and transitions in Markov Chains - has been used for many years. Increased computing power has further stimulated this tendency by offering direct manipulation interfaces that allow users to experiment with the aid of graphical representation of their models.

In the area of linear programming, gLPS (graphical Linear Programming System) describes classical linear problems in terms of graphical objects (circles for restrictions, squares for variables, etc.) networked according to specific rules to form a model. The system ensures the consistency of the model, allows for its manipulation and modification through a Macintosh-like interface and activates a solver to compute the solution. The major strength of gLPS resides in the ability to apply the same simple set of components and rules in modeling any kind of linear system (transshipment, product-mix, etc.).

gLPS is not only a modeling language, but also an integrated software tool for the construction, modification, documentation and calculation of linear problems.

Keywords

Linear programming, graph-based LP modeling

1 Introduction

The emergence of the latest generations of powerful graphics-based microcomputers and workstations has significantly influenced endusers perception of interactions with the machine. WYSIWYG word processors and the definition of sophisticated, direct manipulation user interfaces in the desktop publishing domain lead the way in this trend towards graphics-based user interfaces. With a few years' delay, the same trend now appears in software tools dedicated to mathematical modeling. Examples of such tools are Stella (HPS, 1990) and Extend (Imagine, 1987) for system dynamics and discrete event simulation.

With linear programming (LP) models, this evolution occurred as shown in Figure 1. At first, the only executable form was the cumbersome MPS format (IBM, 1976). Eventually systems known as "matrix generators" like APEX-II (CDC, 1974), OMNI (Haverly, 1977) or Modeler (Burroughs, 1980) were developed to better manipulate the MPS format. Within the last decade, as a result of increased computing power and the development of extremely fast algorithms (Karmarkar, 1984), the needs of OR specialists have shifted from mere solution software to integrated modeling environments. A first step in this direction was the creation of executable, declarative, modeling languages like GAMS (Bisschop and Meeraus, 1982), LINDO (Schrage, 1984) and AMPL (Fourer et al., 1987), or of structured modeling language (Geoffrion, 1989). The latest stage in the development of such systems concentrates on defining modeling languages based on graphic syntax like And-Or graphs (Raghunathan, 1987), NETFORMS (Glover et al., 1990) or GIN (Steiger et al., 1991).

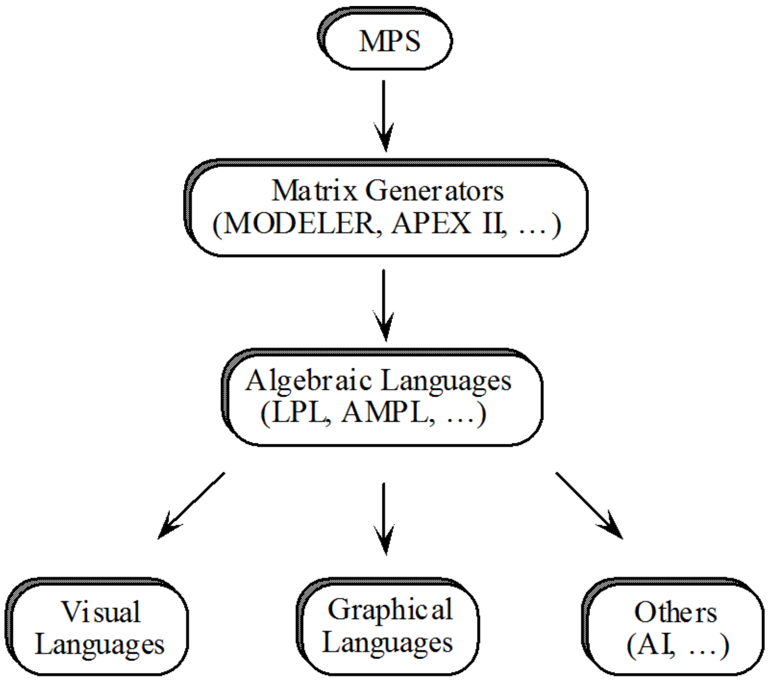

MPS
Matrix Generators (MODELER, APEX II, …)
Algebraic Languages (LPL, AMPL, …)
Visual Languages
Graphical Languages
Others (AI, …)

Figure 1 : Generations of LP software tools

At the IAUF, there exists a long line of tradition in the development of software tools that deal with large strategic planning LP models (Egli and Kohlas, 1981; Pasquier et al., 1986). A graph-based symbolism, in particular, has been designed for documentation purposes (Egli, 1980), as has an executable modeling language which greatly simplifies the definition and modification processes of large LP models (Hürlimann and Kohlas, 1988). By integrating both the convenience of Egli's symbolism and the power of Hürliman's linear programming language (LPL) within a unique direct manipulation software environment, gLPS (graphical Linear Programming System) represents a steady continuation of these research projects.

The purpose of this paper is not to present the implementation details of gLPS, but rather to provide a functional description of the system. In order to best achieve this goal, the paper is structured as follows : Section 2 describes the symbolism of gLPS graph-based modeling language; Section 3 concentrates on the capabilities of gLPS direct manipulation user interface; and Section 4 summarizes the main benefits of the system.

2 Symbolism

The objects of gLPS are divided into three categories :

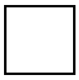
– basic elements, which represent the components of a general LP model, e.g. variables, restrictions, etc.,

– connecting elements, which serve to build a graph representation of a general LP model,

– submodel element, which allows the visualization of any sub-part of a model as a unique component.

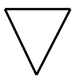
2.1 Basic elements

Every LP consists of at least one variable, one restriction and one objective function. Within gLPS, these elements are represented by combinations of the following symbols :



The square designates a variable (e.g. the quantity of robots of a given type to be produced).

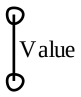
The circle, associated with the square(s), refers to the left hand side (LHS) of a restriction or to an objective function. It can be labeled in different ways, depending on its type : '≥', '≤', '=', '`Min`' or '`Max`'.



The triangle represents the right hand side (RHS) of a restriction and contains the associated data.

2..2 Connecting elements

The basic elements alone do not constitute a model if they are not connected to form a suitable network with the help of the following objects :



The coefficient line always relates a square and a circle and, therefore, is used to define a part of either a restriction LHS or an objective function. It contains the associated data.

The RHS line always relates a triangle and a circle, thus defining a restriction RHS.

Figure 2 illustrates a typical constraint (from Example 1 below), using the above symbolism.

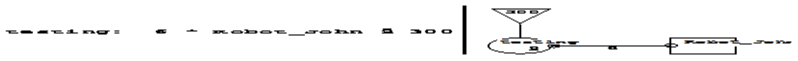


Figure 2 : A typical restriction

2.3 Sub-model element

The hexagon represents a submodel, i.e. any subgraph of the complete model network. Submodels are useful for two reasons. First, they can be stored in special libraries to simplify any further construction of related models. Secondly, an appropriate compaction of elements into submodels eases the comprehension of complex models whose graphical representation requires too much space or which become confusing to the user.

3 User Interface

Unlike the classical mathematical notation, the symbolism presented in the previous section allows for the use of a software tool based on a modern direct manipulation interface : gLPS. A session with such a tool presents two main activities : (1) The user completely defines the structure of the model. (2) Then, it is possible to work with it effectively in checking its data, exploring its structure at various levels of detail, and computing the optimal solution.

Creation of the model

Consider the following problem :

Example 1 : A production problem

A company produces three types of robots called 'John', 'Kevin' and 'Jane'. Three production steps must be carried out :

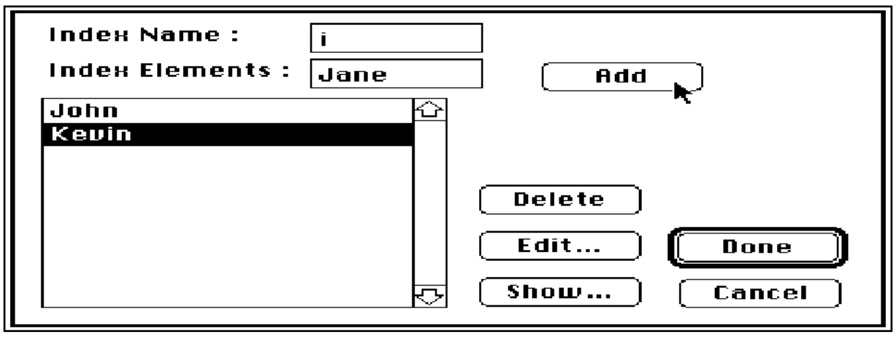
- Production of the components takes 5 hours for each robot John and Kevin and 4 hours for Jane, with a total capacity of 350 hours per week;
- Mounting (capacity <= 480) takes 4 hours for John and 5 hours for Kevin;
- Testing (capacity <= 300) takes 6 hours for John, 2 for Kevin and 4 for Jane.

Profit is $300 for John, $200 for Kevin and $100 for Jane. Of type John robots, 20 have already been ordered. Of type Kevin 15, and of type Jane

```
7.  How many robots of each type must be produced in order to maximize
the sales profit ?
```

Figure 2 illustrates the testing constraint for robot John. In order to define a complete graph representation of the model, it would be necessary to add two squares (one for each missing robot), two circle/triangle combinations (one for each missing production step), one circle for the objective function, as well as appropriate coefficient and RHS lines. There is, however, a better way to proceed, this being through the use of the powerful concept of indices. The final model produced by this new approach is presented in figure 5. Its creation process includes the following operations :

– First, two indices, `i` with range (`John, Kevin, Jane`) and `j` with range (`prod, mount, test`), must be created. Specialized dialog boxes and predefined libraries, either in gLPS format or ASCII, are among the several facilities offered by gLPS for specifying such indices. This step is illustrated by Figure 3 : the user adds the index element `Jane` to the already existing range for index `i`.


Index Name : i
Index Elements : Jane

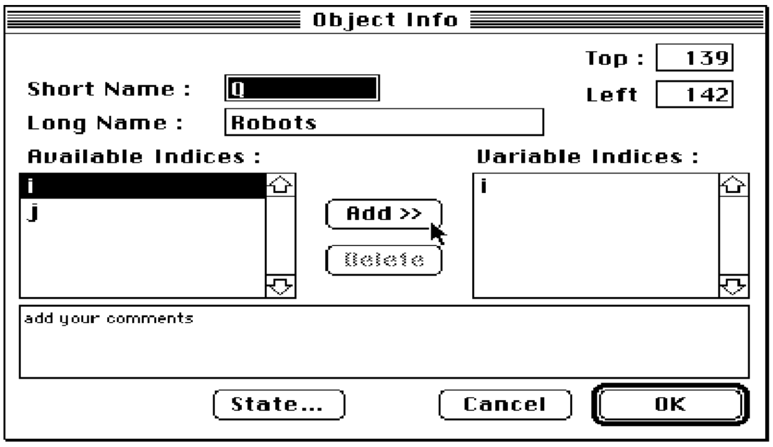
John
Kevin

Figure 3 : A screen shot of indices creation

– Secondly, the basic elements are created and positioned on the view.

– Third, the structure of the graph is defined with the help of the coefficient and the RHS lines. gLPS helps the consistency of these connections to ensure.

– Fourth, the appropriate indices are associated with the different elements of the model (see Fig. 4). gLPS automatically generates the RHS indices and restricts the choice of coefficient lines indices to consistent ones.



Object Info

Top : 139

Left 142

Short Name : Q

Long Name : Robots

Available Indices : i, j

Variable Indices : i

add your comments

Figure 4 : A screen shot of the 'Object Info' dialog

– Finally, the RHS and coefficient data are introduced either manually or from external files.

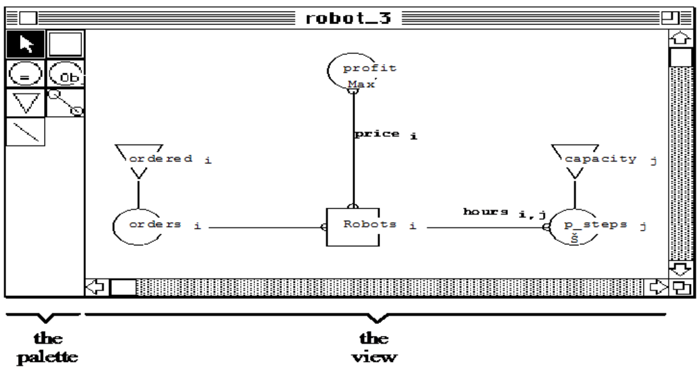


Figure 5 : A production problem

Note that the order of the above operations is not strictly defined (e.g. the structure may be specified before the indices) and that they are all carried out with the mouse, the palette and/or the menus according to the Apple human interface guidelines (Apple, 1987).

The graph of figure 5 can be edited and its comprehension and legibility enhanced with the help of the gLPS menus (see Fig. 6, VIEW and DATA menus). These self-explanatory menu items allow for the manipulation of submodels, simplification of displays by hiding classes of elements and for edition of elements properties (INFO… items).

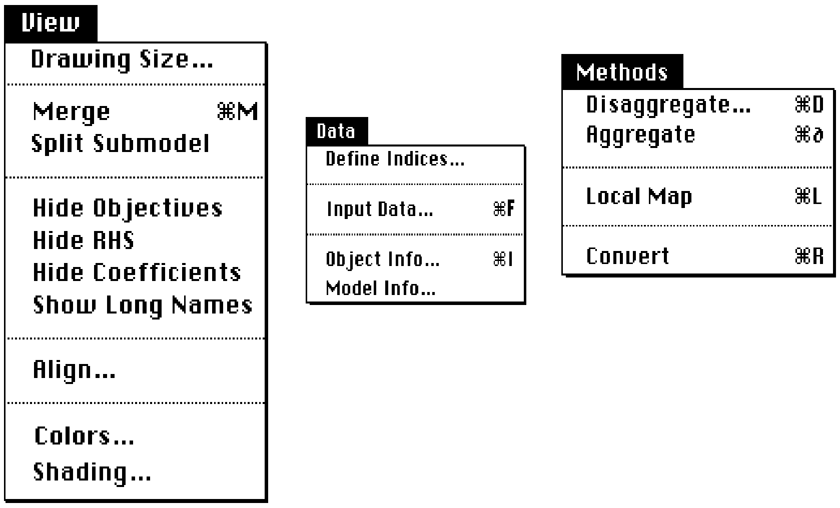


Figure 6 : gLPS menus

Working with the model

The METHODS menu of Figure 6 allows for both computation and exploration of the model. A model evaluation is initiated by the CONVERT item, while its inspection is performed through the use of the LOCAL MAP, DISAGGREGATE and AGGREGATE items.

The calculation process consists of three steps, the first being the conversion of the graph into an LPL notation, which is saved in a text file (see Table 1). Secondly, the LPL compiler tests the accuracy of the textual file and transforms it into the format required for the available solver (e.g. MPS). Then, the solver computes the optimal solution. This three-step process is based on the idea of dividing the work into small and coherent modules, each responsible for a specific group of tasks. gLPS integrates these modules and manages the user interface. LPL provides the full power of a general modeling language. Finally the solver takes care of calculations.

```
SET
  i = (John Kevin Jane);
  j = (comp mount test);
COEF
  capacity(j) =
     [  comp 350
        mount 480
        test 300 ] ;
  ordered(i) =
     [  John 20
        Kevin 15
        Jane7 ] ;
  hours(i,j) =
     [  John  comp 5
        John  mount 4
        John  test 6
        Kevin comp 5
        Kevin test 2
        Jane comp 4
        Jane mount 5
        Jane test 4 ] ;
  price(i) =
     [  John  300
        Kevin 200
        Jane 100 ] ;
VAR
  Robots(i);
MODEL
  p_steps[j] :
    SUM(i) hours * Robots(i)  <=  capacity;
  orders[i] :
    SUM(i) Robots(i)  >=  ordered;
  MAXIMIZE profit :
    SUM(i) price * Robots(i) ;
{$Solve}
PRINT
Robots; profit;
END
```

Table 1 : A production problem : LPL code

Once a model has been defined, it is possible to inspect and modify its structure and data by disaggregating selected basic elements. The decomposition of an element involves the expansion of one or many of its indices. A possible disaggregation of the production model is illustrated by the graph in Figure 7. This is a complete one, thus allowing for verification and modification of all the coefficients of the model. Such a complete decomposition is only convenient for rather small models. For larger ones, it is possible (and even recommended) to work with several partial decompositions by selecting subsets of basic elements and expanding only subranges of their indices.

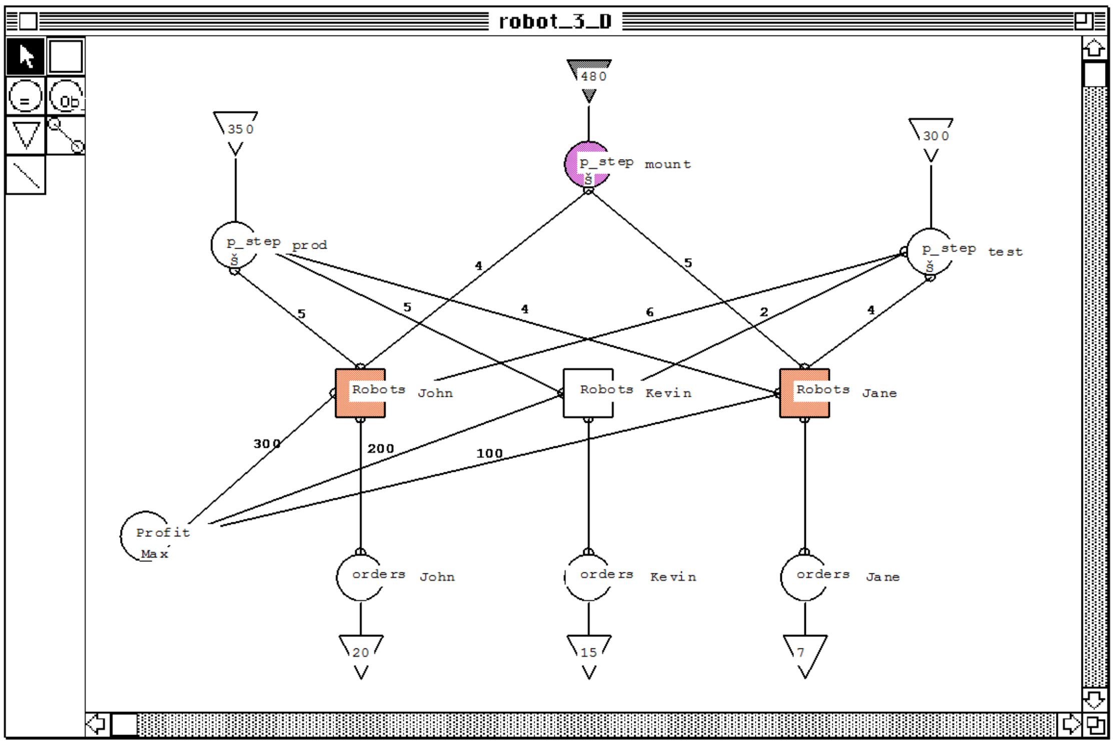


Figure 7 : A production problem : complete disaggregated view

This feature, along with the generation of local maps (Utting and Yankelovich, 1989) (see Fig. 8 for a local map on robot `Kevin`), provide the user with a powerful tool in dealing with complex models. The underlying concept behind the disaggregation process, as well as its main benefits, will be discussed further in Section 4.

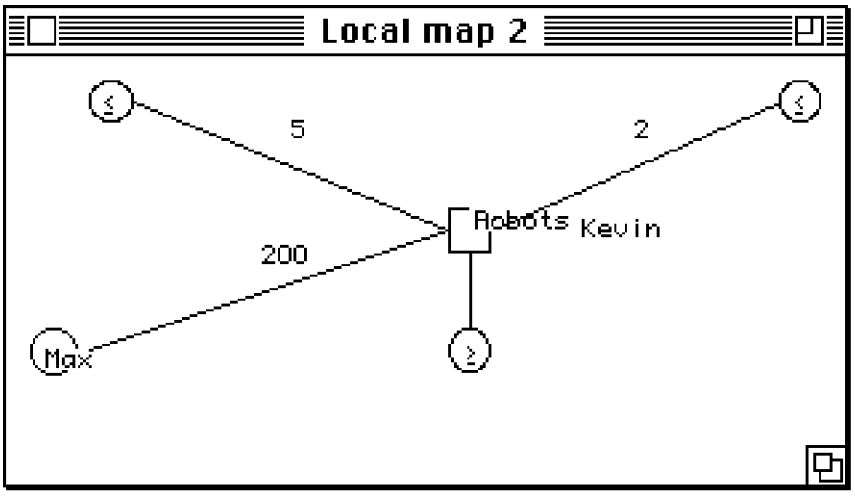


Figure 8 : Local map for robot `Kevin`

4 Benefits

In addition to the convenience of its user interface, gLPS offers the following benefits : (1) A general symbolism which allows for the definition of any LP model and for the integration of all the views necessary for its modeling and evaluation and; (2) The flexibility of its disaggregation mechanism.

4.1 Generality

As demonstrated in Figure 9, the symbolism of gLPS is a direct graphical translation of the classical mathematical LP notation. Thus, unlike other graphical tools like for example generalized networks (Glover et al., 1978), gLPS is not restricted to specialized subclasses of linear problems.

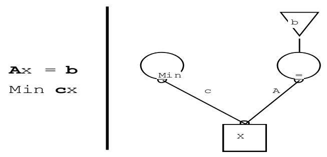


Figure 9 : Standard formulation of a general LP model

4.2 Views integration

The modeling and evaluation of an LP problem normally require at least three representations (Fourer, 1983). OR specialists first build a symbolic representation of the linear program (the modeler's form). Then, the model is translated into an algorithmic form for calculation by the solver. Finally, a third representation must be created to present the results in a concise and legible form (see Fig. 10). With gLPS, all these tasks are accomplished within the system without having to switch between formalisms. To build the model, the user works directly with the system (thanks to gLPS direct manipulation user interface, this process is incremental and reflects true trial-and-error modeling). The model is then automatically sent to the solver, so the user is not obligated to know anything about the necessary format. Finally, the results are either listed in a classical way or returned to the gLPS graph. The latter is clearly more powerful, allowing for inspection of not only the input but also the output data with the tools described in Section 2.

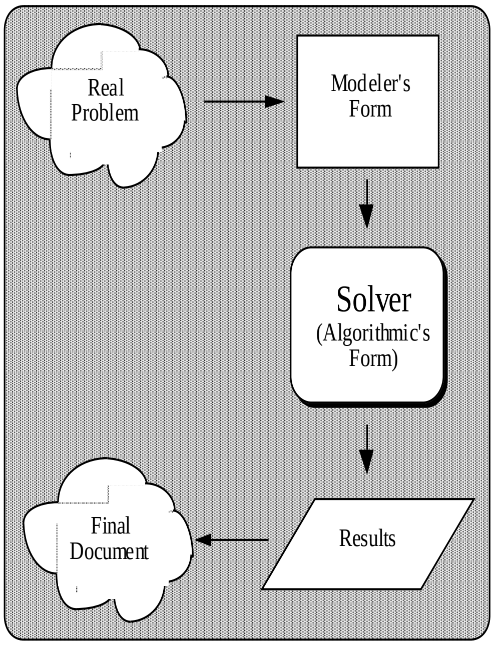


Figure 10 : Different representations of an LP model

4.3 Disaggregation

Figure 11 summarizes the process of exploring the model's structure with the help of the tools proposed by the METHODS menu (see Fig. 6). The user defines the model at a given level, analyzes its deeper structure with the DISAGGREGATE item and may return to the original level thanks to the AGGREGATE item. The usefulness and power of this inspection mechanism are better highlighted in the problem of Example 2 below.

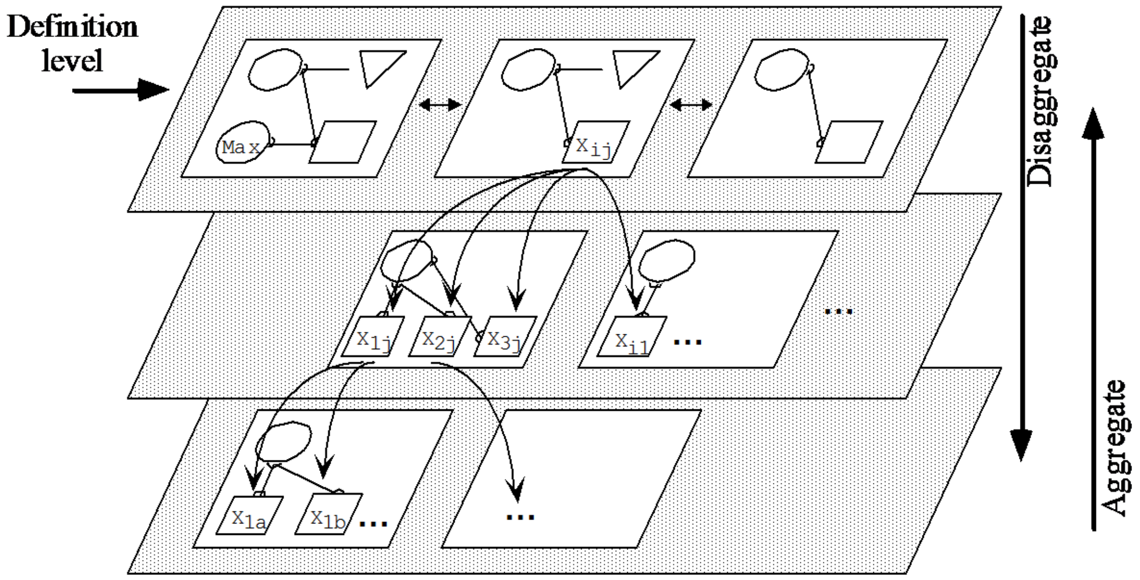


Figure 11 : Different levels of an LP model

Example 2 : A transshipment problem with capacity and stocks

Goods must be shared between the following Swiss towns, depending on the supply and demand of each : Geneva, Neuchatel, Sion and Fribourg.

Transportation costs, which rely on distance, cannot exceed a certain limit. Goods may also be stocked at a certain cost and distributed at a later date (a possible data set is presented in Figure 12). The objective is to determine, over four periods, the optimal transportation quantities which will both satisfy supply and demand and minimize overall costs.

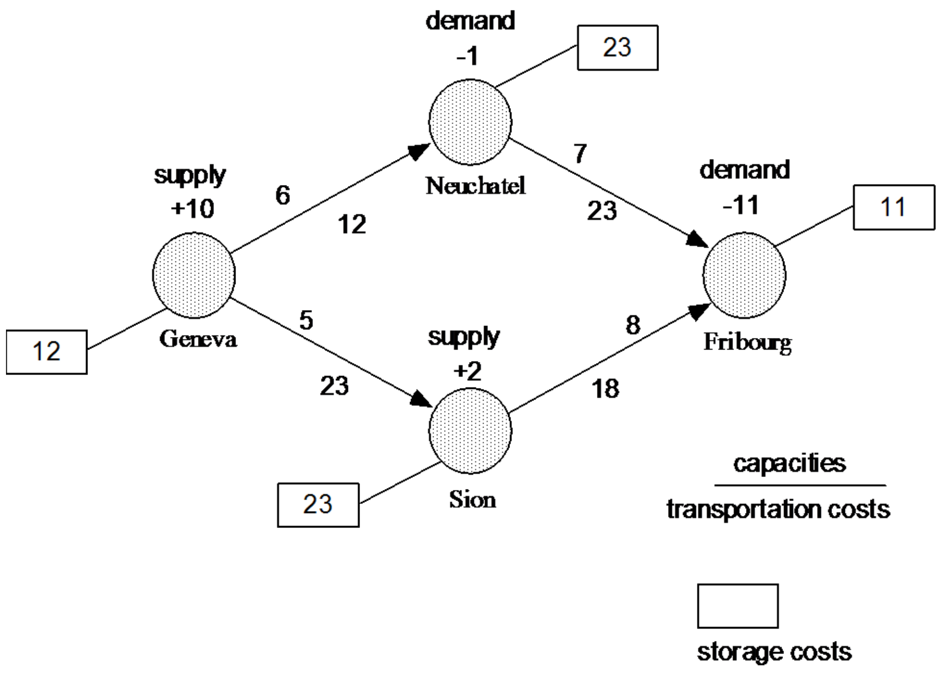


Figure 12 : Data definition of the transshipment problem

Note that despite its apparent convenience, the symbolism of Figure 12 is limited to transshipment problems and, even then, is incomplete because it does not allow for precise representation of the time component.

Figure 13a proposes a gLPS representation of the problem. It consists of :

– two indexed variables (squares) : one representing the stock at each town `k` over all periods `t` and the second designating the quantities of goods transported on each arc (`i,j`) over all periods `t`. The additional horizontal line on the right square is an optional short cut for representing an upper-bound restriction on this variable.

– an objective function (lower circle) which minimizes the sum of storage and transportation costs over all periods.

– an indexed constraint (upper circle) which expresses the well-known balance equations for each town over all periods, i.e. at a given period, the goods leaving the town minus those entering must be equal to the town's supply or demand.

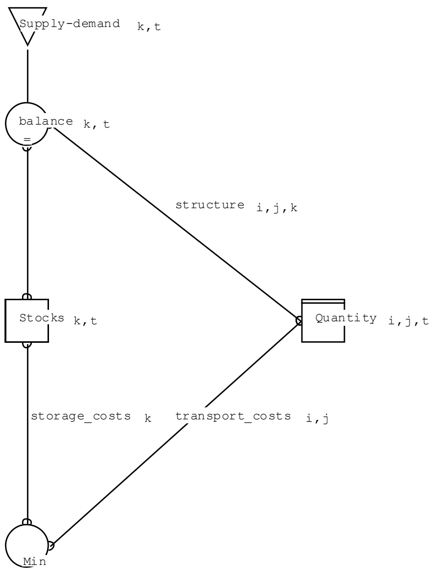


Figure 13a : gLPS graph of the transshipment problem

Deeper insights into the structure of these equations are shown in Figures 13b, 13c and 13d. Figure 13b illustrates the consideration of stock in the balance equation of a given town `k`. Basically, in order to balance its flow of goods, town `k` at period `t` demands its stock of period `t-1` and supplies that of period `t`. Figure 13c shows that the balance equation at each town depends on the quantites leaving the town and those entering. In order to produce this representation, it was necessary to disaggregate the quantities over all arcs `i,j` and the balance equations over all towns `k`. Figure 13d presents a complete disaggregation (over all arcs for quantities and over all towns for stocks) of the objective function variables for a given period `t`. This latter representation provides for convenience when checking and eventually modifying the transportation and storage costs.

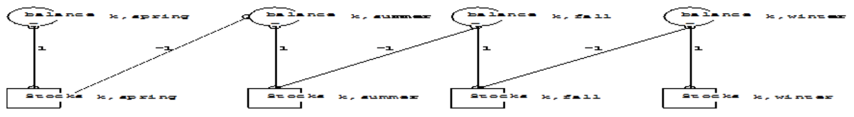


Figure 13b : Disaggregation over the periods for the carrying of stocks in a given town `k`

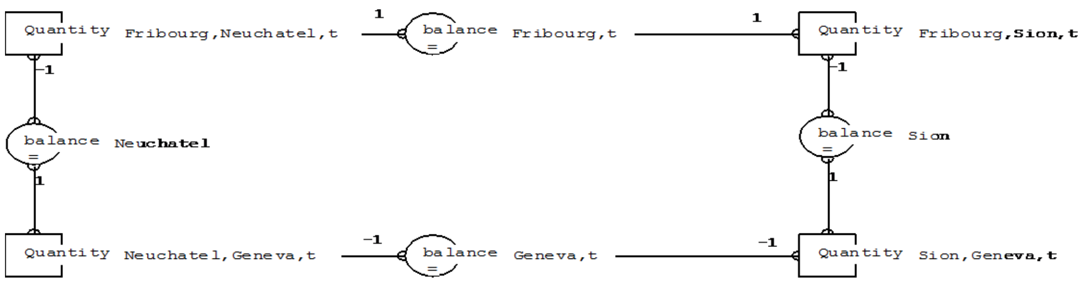


Figure 13c : Balance equation for each town

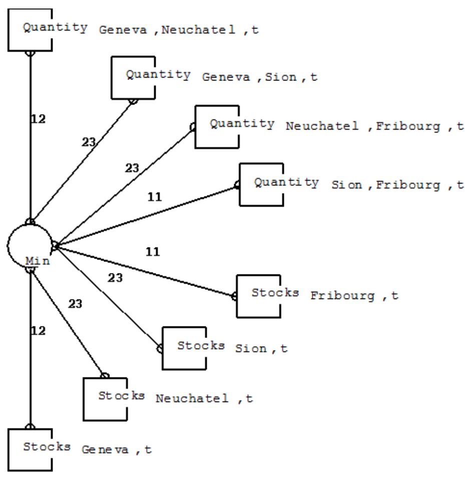


Figure 13d : A closer look at the objective function

5 Conclusion

We have provided, in this paper, both the definition of a new graph-based language used to depict general LP models, and a functional description of its software realization. It must not be forgotten, however, that the final goal of the gLPS project is not only to implement a new modeling language but also to propose an integrated software tool to provide the user the capability to :

- define, store and recall libraries of models, submodels and indices,
- shift back and forth between the graphics-based symbolism of gLPS and the text-based symbolism of LPL,
- compute the optimal solution and obtain the results either through LPL report generating facilities or directly on the original gLPS graph,
- link the components of the gLPS graph within a hypertext database.

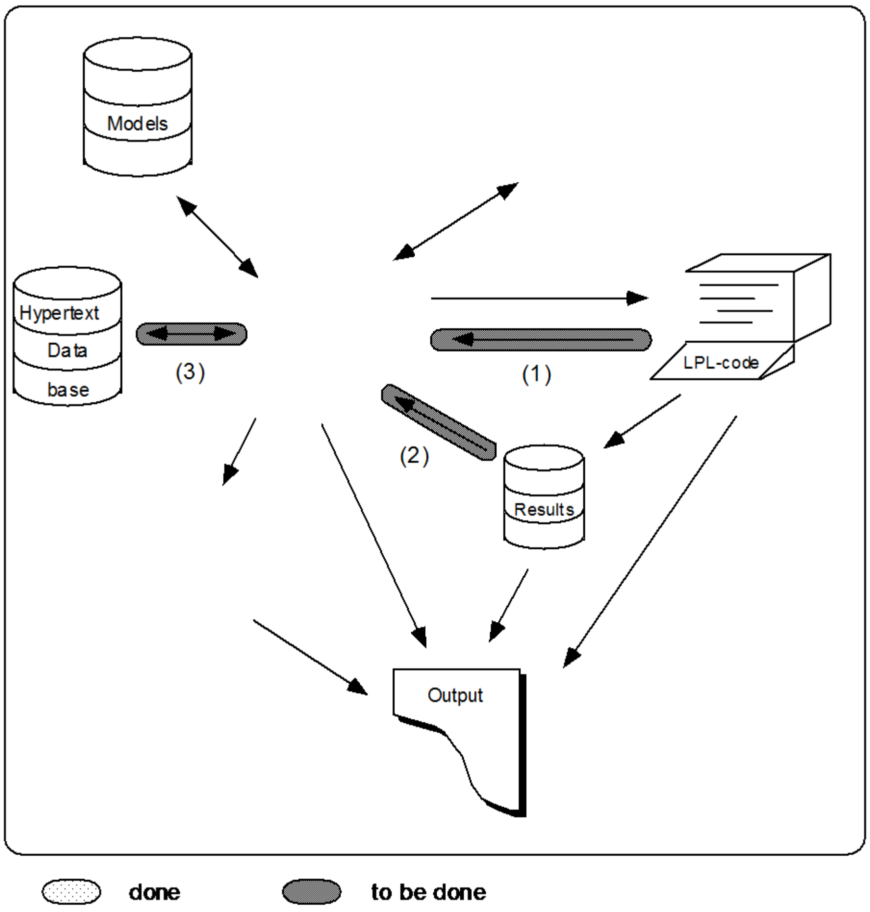


Figure 14 : The complete gLPS modeling environment

Figure 14 provides an overview of the modeling environment described above. Highlighted are the areas yet to be completed : (1) Investigation and implementation of efficient methods of translating LPL text files into gLPS graphs; (2) Visualization of optimal solution values directly on the gLPS graph (a fairly straightforward task which can be accomplished directly or through LPL; not yet programmed); (3) Integration of gLPS and WEBS, the hypertext engine developed at the IAUF (Pasquier et al., 1992). The latter has been anticipated from the very beginning, being that the hypertext capabilities of WEBS are contained entirely within a layer of software objects, which directly extends the basic object oriented library (MacApp from Apple) used to program gLPS.

References

Apple Computer (1987), *Human Interface Guidelines: The Apple Desktop Interface*, Addison-Wesley, New York.

Bisschop J. and Meeraus A. (1982), "On the Development of a General Algebraic Modeling System in a Strategic Planning Environment", *Mathematical Programming Study* 20, 1-29.

Burroughs Corporation (1980), "Model Development Language and Report Writer (MODELER)", User’s Manual 1094950, Detroit, MI, USA.

Control Data Corporation (1974), "APEX-II Reference Manual", 59158100, Rev C, Minneapolis, MN, USA.

Egli G. (1980), *Ein Multiperiodenmodell der Linearen Optimierung für die Schweizerische Ernährungsplanung in Krisenzeiten*, Thesis, Fribourg, Switzerland.

Egli G., Kohlas J. (1981), "A policy model for planning alimentary self-sufficiency in Switzerland", in : JP Brans (ed.), *Operational Research '81*, North-Holland, Amsterdam, 311-322.

Fourer R. (1983) "Modeling Languages versus Matrix Generators for Linear Programming". *ACM Transactions on Mathematical Software* 8, 2, 143-183.

Fourer R., Gay D.M. and Kernighan B.W. (1987), "AMPL : A Mathematical Programming Language", AT & T Bell Laboratories, Murray Hill, NJ, USA.

Geoffrion A. M.(1989), "Computer-based modeling environments", *European Journal of Operational Research* 41, 33-43.

Glover F., Hultz J., Klingnan D. and Stutz J. (1978), "Generalized Networks : a Fundamental Computer-Bases Planning Tool", *Management Science* Vol. 24 No 12, 1209-1220.

Glover F., Klingnan D. and Phillips N. (1990), "Netform Modeling and Applications", *Interfaces* Vol. 20:4, 7-27.

Haverly Systems Inc. (1977), "OMNI Linear Programming System : User Manual and Operating Manual", Denville, N. J., USA.

High Performance Systems (1990), "Stella II : User manual", Hanover, NH, USA.

Hürlimann T., Kohlas J. (1988), "LPL : A Structured for Language Linear Programming Modeling", *OR Spectrum* 10, 55-63.

IBM Corporation (1976), "IBM Mathematical Programming System Extended/370 (MPSX/370) Program Reference Manual", SH19-1095-1, New York and Paris.

ImagineThat ! (1987), Extend : "Performance Modeling for Decision Support", San Jose, CA, USA.

Karmarkar N. (1984), "A New Polynomial - Time Algorithm for Linear Programming", *Combinatorica* 4, 375-395.

Pasquier J, Hättenschwiler P., Hürlimann T., Sudan B. (1986), "A convenient technique for constructing your own MPSX generator using DBASE II", *Angewandte Informatik* 7, 295-300.

Pasquier J., Collaud G., Monnard J. (1992), "An Object-oriented approach to conceptualizing and programming an interactive system for the creation and consultation of electronic books", in J. Pasquier (ed.), *Electronic Books and their Tools*, Lang, New York, 23-38.

Raghunathan S. (1987) "An Intelligent DSS for Model Formulation", Working paper, University of Pittsburgh, KS, USA.

Schrage L. (1986), *Linear, Integer, and Quadratic Programming with LINDO*, Scientific Press, Palo Alto, CA, USA

Steiger D., Sharda R. and LeClaire B. (1989), "Functional Description of a Graph-Based Interface for Network Modeling (GIN)", Working paper, College of Business Administration, Oklahoma State University, OK, USA.

Utting K, Yankelovich N. (1989), "Context and Orientation in Hypermedia Networks", *ACM Transactions on Information Systems* Vol 7 No 1 (January), 58-84.

5 Computer Assisted Top Down Modeling

Pius Hättenschwiler

*Institute for Automation and Operations Research (IAUF),
University of Fribourg, CH - 1700 Fribourg, Switzerland
E-mail : HAETTEN@CFRUNI51.BITNET*

1 Introduction: Characteristics of Modeling and Challenges for a successful Modeling System

Computer assisted modeling for everybody

Human thinking is concerned all the time with abstracting some part of the real or imagined world. This is an activity that is very close to modeling. Human awareness of his own environment consists mainly of abstract images of the surrounding world, kept in the mind. They could be called implicit models. From this kind of view, modeling is a familiar task of information processing. On the contrary, explicit modeling, which is concerned with formalised explicit models[1], is considered as a difficult task of any intellectual work.

Combining the two facts, natural need of modeling and extreme difficulty of explicit modeling, it is plausible to conclude that there is an enormous market potential for computer based modeling systems, comparable to the market volume of systems for text processing, calculation or information retrieval. Actually computer assisted modeling systems for everybody are far from being commercialised profitably. What are the reasons for this?

Missing tradition for formal modeling

Computer assisted modeling may seem a much more intelligent application of the computer than text processing. This is probably wrong. The arguments for this will be given in the following paragraphs. It is true that explicit modeling is done by fewer people than writing is. In addition, writing is learnt and trained

[1] Explicit models are materialised objects. They can be managed, i.e. kept or destroyed, visualised, altered, duplicated, applied, and so on.

during ten to fifteen years and has a long tradition. Formal modeling, however, is trained very little and late in the educational course. Education on and application of formal modeling could become a common activity as soon as modeling gets standardised and is facilitated by the computer.

Missing standards and basic concepts for modeling

At present, modeling techniques are not sufficiently standardised. What is still missing in this field are the basic concepts that are accepted as standard. As a contrasting (positive) example, the basic concepts for document processing are well established: letter, word, sentence, paragraph, intendation, page, table, image, title, section, document, footnote, index. While these elements are sufficient for describing any sequential document and also its processing, we do not yet dispose of a similar set of basic concepts for all kinds of formal models. This is an important field of research, that will be discussed in a future paper.

Missing user-friendly systems for modeling

What about the system needed for computer assisted modeling? Must it be much smarter than a word processing package? This opinion is widespread but wrong. A successful (computer based) modeling system will not consist essentially of a sophisticated expert system. Text processing systems can help a lot to create good papers even though the system does not understand the text at all (no intelligence). The creative part of work has to be done by persons. This is true for writing as well as for modeling. The computer can just raise the productivity of creative work - and by this - allow one to evaluate more and better alternatives in a creative process.

Modeling is (must be) an evolutionary process

Models offer - as documents or photographs do - a possible description of a certain part of the real or conceived world. Note that within this comparison, models are definitely closer to text documents than to photographs: both text and models can evolve their contents starting from a primitive or summary description, getting more and more accurate. This might be the crucial characteristic of modeling: a formal model is seldom a final product that is first created and then applied. It much resembles an organism. Similar to that, a formal model is close to death as soon as it stops evolving (developing, maturing). Moreover, the evolving model wants to be used (applied, tested), otherwise its evolution stops because of the missing feed-back from its real world counterpart. So, we can summarise: a model must be developed and applied continuously.

A modeling system must facilitate the evolution of a model

If it does so, a modeler can start building simple and shallow models and try to improve them continuously. In this way, formal modeling could be accessed by everybody. Such a modeling system would sell like successful text processing systems. What would be the main tasks of such a modeling system?

A modeling system has to provide for:

- Computer Assistance for proper conceptual modeling
- Model Views
- Re-use of model code
- Knowledge representation scheme
- Knowledge transfer vehicle

Computer Assistance for proper conceptual modeling

There is no doubt that models have their basic components and a common architecture as well as documents do have them. These basic concepts have to be found and defined. Probably the basic concepts will be defined within the object oriented paradigm. For the developer of a modeling system, the main challenge will be to develop a graphic management system of these conceptual elements that is intuitively understandable for the end-user modeler. The objective should be to completely discharge the end-user modeler of syntactical and grammatical concerns, offering him or her a dialogue that guarantees proper conceptual modeling. In an early phase of the model design the modeler shouldn't even have to care about visualisation of the created objects and their relations. The modeling system should offer some "Standard Views" (see below) that are sufficient for checking the correctness of the conceptual model (see also 4.3).

Model Views - Standard Views and application oriented Views

Model Views are different ways of looking at a model. A model View shows all or part of the elements of a model, all or part of their properties, in a specific order of placement of the elements.

The most important help a modeling system can offer is its power to produce, i.e. calculate and visualise specific Views of a model within seconds or even milliseconds. By this, the modeler can - for different kinds of questions - switch quickly between different virtual Views[1] and can get familiar with the model. An important role can be assigned to so-called Standard Views. First, they liberate the modeler during the model designing phase from the laborious task of arranging elements on the screen. Secondly, Standard Views allow the viewing of completely different models in always the same familiar manner and facilitate by this the re-use and integration of models. For the application of the model, the user should have the opportunity to define easily convenient Views, i.e. application oriented Views; this by applying all the progress recently made in graphical user interfaces.

Models for different purposes, re-use of models

A model at different levels of evolution, presented in different Views, and describing a more or less complex context, can have several uses (purposes). Often models can be re-used as modules within more complex models. The re-use of models in a kind of modular modeling might be decisive for an economic break-through to "everybody's modeling".

Models as knowledge representation

In the future, the most important roles of models will be knowledge representation, knowledge handling, and knowledge application. In the world of classical Operations Research, models were just a means to apply solving techniques (algorithms) to modeled problems. However, this is only the application aspect of a model. As a consequence, most of the OR-models were black box models for most people involved. This situation is subject to change since computers and powerful software[2] allow the storing and handling of more than just simple data (i.e. facts). The era of knowledge processing and knowledge representation is just about starting. It will also become the era of comprehensible models.

Mathematical, semantic or logical models are a very useful and - to rule based systems - complementary form of knowledge representation. While models concentrate on describing the context[3] of a problem, rule bases focus on how to proceed in the problem solving process. The procedural character of production rules is given by the option of executable actions on the right hand side of the rule: if some conditions are true then another condition becomes true and one or more actions are started. The trivial action do nothing is allowed. If, however, a

[1] A virtual View is a view that is not saved permanently but exists only at run time.

[2] especially the invention of abstract data types

[3] the scope of the world where a problem has been localised.

rule base contains only trivial actions, it becomes a logical model (of predicates).

Models are complementary to rule based systems for a second reason: modeling has a more normative character, i.e. in general, models reflect the theoretical knowledge of a context. Rule bases, however, allow the description of positive knowledge (experience) that has never been proved or even cannot be understood. Expert systems can be composed of rules of thumb.

This may be explained by an example: In general, a good medical practitioner has studied profoundly the anatomy and the physiology of healthy persons. He has acquired an overall picture or model of the well functioning of people. He will be able to detect and understand any sickness, even a new one he has never collected experience on. The other, so-called positive approach consists in learning from pathology: What kinds of symptoms indicate which type of sickness? Diagnosis by this second approach is more direct. The rules connecting symptoms with hypothesis and perhaps actions (further analysis for example) can work well even if they are not understood. A very good physician, of course, will combine both types of knowledge, theory and experience.

The most important differences between theory or model on one hand and experience or rule base on the other hand are:

theory / model	experience / rule base
large problem context	specific problem oriented
consistent building of related knowledge	independent chunks of knowledge
is derived by reasoning	is derived by observation
good for education	quick for application

Expressive knowledge representation - knowledge transfer

Using an elaborated viewing technique (many Views, explicit naming, iconising of objects, parallel inspection by different Views in several windows) models can become extremely expressive. This can lead to a completely new role of computer based models: as vehicles for communication about knowledge upon a specific context (knowledge exchange, knowledge transfer, education, fellowship and expertise in a company).

Since computer based models can be stored, transformed, applied and transmitted (by telecommunication) very efficiently using modern computer technology, they could achieve a key function within future knowledge-handling. Explicit (computer based) models could open a new era of knowledge-management, as happened when the book was invented.

2 Model Representation and Model Views

Similar to data base systems, a computer based modeling system should provide three levels of model representation[1]: a conceptual scheme which is independent of a specific implementation, an internal scheme for an efficient implementation, and finally an external scheme allowing the definition of standard Views and user defined Views.

2.1 Conceptual scheme (logical scheme)

A computer based modeling system has to provide a powerful model representation scheme that allows the capturing of data and structures of a wide variety of models. The model representation scheme should be conceptual. In other words, it should be independent of a specific hard- and software platform. The conceptual scheme has the following functions:

(a) It is used as a unique representation scheme for a comprehensive class of models. It should offer a generic representation that allows one to reduce redundancy to a minimum. (I.e. separation of the generic model structure from data of the model detail).

(b) It gives a lexicographic description, i.e. it can be read (analysed) sequentially and published in a printed media.

(c) It figures as a external interchangeable format between different modeling systems. Models can be exported to and imported from the conceptual scheme.

There does not yet exist an generally accepted conceptual representation scheme for modeling. Most of the published schemes have been defined for algebraic or mathematical modeling purposes (GAMS[2], LPL[1], AMPL[2], LAMP[3], MODLER[4],

[1] Elmasri, R. and Navathe S.B., Fundamentals of Database Systems, The Benjamin/Cummings Publishing Company, Inc. Reedwood City ,California, 1989.

[2] Brooke A., Kendrick D., Meeraus A., GAMS: A Users Guide, The Scientific Press, Redwood City, CA, 1988.

SML[5]). They do - with a partial exception of SML - not allow the modeling of a database system or an organisation of a company for instance.

The conceptual representation can be used to work on the model, but this is not the primary purpose of it. Yet the systems mentioned above do so. A promising modeling system, however, should produce the conceptual representation starting from the internal scheme or the vice versa, translate the "poor sequential View" of the conceptual scheme into its powerful internal representation. Based on the internal scheme, it should offer the modeler a variety of external Views, i.e. standard Views and application oriented Views. Normally the standard Views should suffice for model design, specification and testing (see below 4.3).

The main research objectives in the field of conceptual model representation are:

1) powerful model representation independent of problem context and model solver technique that could be proposed as a standard representation scheme of models.
2) maximum model insight, even on this level of the conceptual scheme.
3) modularity of model representation: composing more complex models from elementary sub-models (modules).

These aspects will be discussed in a future paper.

2.2 Internal scheme (physical realisation: Implementation)

1 Hürlimann T., Reference Manual for the LPL Modeling Language, Version 3.8, June 1990, Working Paper No. 191, Institute for Automation and Operations Research, University of Fribourg / Switzerland, 1992.

2 Fourer, Gay, Kernighan, A Modeling Language for Mathematical Programming AMPL, WP, Technical Report No. 133., published in Management Science, Vol. 36, No. 5, May 1990.

3 Singh I.S., A Support System for Optimization Modelling, North-Holland, Decision Support Systems 3 (1987) 165-178.

4 Greenberg H.J., A Primer for MODLER: Modeling by Object-Driven Linear Elemental Relations, University of Colorado at Denver, Mathematics Department, Denver, CO, 1990.

5 Geoffrion A.M., The Formal Aspects of Structured Modeling, Operations Research, Vol. 37, No. 1, January-February 1989.

The internal scheme serves as "active" internal representation of a model in a computer for a specific modeling system. It is optimised for quick data access, for efficient data storing and recovering, and for high performance of data processing during an interactive modeling session.

The internal scheme results from importing models from the conceptual scheme and vice versa, it can be translated (exported) to the conceptual scheme.

The creation of models as well as their modification should be executed on the internal scheme, under the strict control of the interactive modeling system. For this task the system should dispose of one or several graph grammars[1]. The last could be extended and refined in parallel to evolving models and the evolving model base.

The main research objective in this field is the definition of a standard kernel for modeling system environments. Such a kernel should constitute the common platform for data resources[2] based on DDL[3] and for modeling functions (primitives) based on DLL[4]. It should be easily portable to any soft- and hardware platform at the price of renouncing any user interface functions.

2.3 External scheme - standard and application-oriented Views

The external scheme is not really an equivalent object to the conceptual and the internal schemes, which are two different (independently stored) forms of a model's representation. In contrast to them the external scheme is a collection of special Views of the model, Views that are produced[5] at the moment, by consulting the internal scheme. For this reason they are also called virtual Views or virtual representations.

The virtual Views are probably the most exciting feature and the best kind of support a computer can offer to the modeler, first by facilitating his job and secondly by stimulating his creativity. Pre-defined Views that show us a subset of elements in always the same stereotype manner are called standard Views. Examples are discussed in section 4.3. They relieve the modeler of the task of organising elements in some hierarchy or in their position on the screen. Active

1 Jones C., An Introduction to Graph-Based Modeling Systems, ORSA Journal on Computing, Vol. 2, No. 2, Spring 1990.

2 model structure data, model detail data of model instances, graph grammar data etc.

3 DDL: Dynamic Data Linking.

4 DLL: Dynamic Link Library.

5 i.e. calculated and displayed to the screen.

standard Views allow the modeler to edit[1] the displayed material. Passive Views give just a better insight[2], and eventually they allow the user to navigate through the model.

A good modeling system provides a rich variety of active standard Views that allow all kinds of modeling tasks to be carried out interactively, controlled by the system's graph grammar.

For the application of a model to solve a specific problem or just for answering a question, the standard Views might not be sufficient, or at least, not very comfortable. A promising modeling system, therefore, has to provide a special tool that allows the user to define completely new Views[3]. Inside the prototype modeling system NetCalc, such user defined Views can be edited by Framework or by EXCEL (see Häuschen[4] in this edition). NetCalc produces the interface between the selected elements of the model and the externally defined View.

The main research objectives in this field are (1) to define excellent standard Views that support the interactive modeling process, and (2) to develop powerful tools that enable the end-user to define appropriate Views for his problem. The application oriented Views are needed to input the user's queries and to output the answers of the model.

3 Standard Views and Functions for Conceptual Modeling

3.1 Conceptual Modeling

What is conceptual modeling?

First we have to define what we mean by "conceptual modeling". It does not simply mean "producing the conceptual representation scheme of a model". The Oxford English Dictionary explains a concept as "something that exists in the mind as the product of careful mental activity". A concept is always something that is well defined but is not yet existing in the final stage of realisation. Although a concept is similar to a plan, it's not the same. A plan is executable and leads by its execution straightforward to the realisation of the planned

[1] editor: change, delete, add elements.

[2] browser: find, visit, show, zoom on elements.

[3] It is called sometimes "editor-editor".

[4] Häuschen H., NetCalc, Modellier- und Kalkulationssystem für vernetzte Modelle, working paper No. 189, Institute of Automation and Operations Research, University of Fribourg (CH), 1992.

object. To emphasise the difference: (1) there can exist a concept of a plan[1], (2) a concept is not guaranteed to be realisable, (3) a concept does not determine exactly the final outcome, but only the contours (main structure) of the next stage of realisation of something.

Now, that we have got the feeling for the notion "concept", we will describe what "conceptual modeling" is intended to be.

Conceptual modeling leads to a concept of a model. Such a model concept is not ready for application. Nevertheless, it might explain a lot about the future model or about the subject to be modeled respectively. Such a concept might evolve into a better concept or into an applicable model. An applicable model might reveal again to be a good concept for another model, or for a more sophisticated model of the same subject, or for a model with a different purpose.

Conceptual modeling leads to knowledge representation

Conceptual modeling is an evolutionary process of knowledge representation. Both concepts and applicable models represent formalised knowledge about the subject to be modeled. This formalised knowledge has a value of its own (Eigenwert).

Conceptual modeling is a necessary phase in every modeling process. Unfortunately, most people (and most of the modeling systems) do not save materialised concepts for later re-use because they concentrate on the use of the "final" model.

Conceptual modeling means capturing and materialising all kind of significant knowledge that was applied on the way to get to a useful (applicable) model. Conceptual modeling leads automatically to self explaining models. Documentation is part of the model development, in a very natural way.

Conceptual modeling is not identical with top down modeling, but it comprises top down modeling as the procedure of stepwise refinement from shallow to deep models.

Conceptual modeling explained by an example

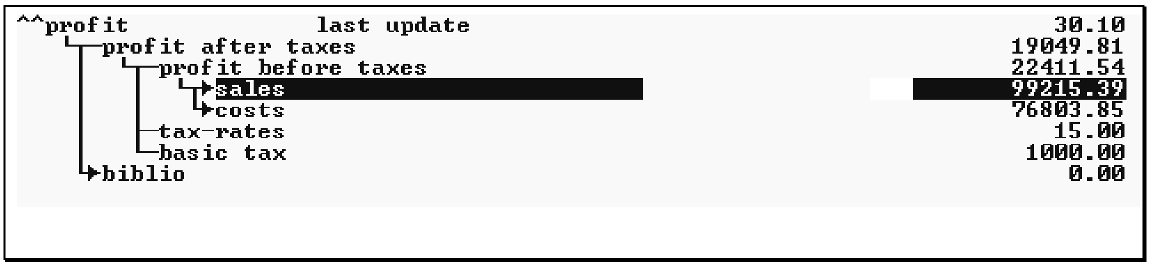
Assume we want to develop a profit model of a company. Contrasting with the classical model-understanding of mathematicians, conceptual modeling starts from an early stage. Modeling the profit of a company conceptually means first to describe what is understood in this company by "profit". How can it be disposed of? What is it influenced by? How could the profit be calculated? Where could the relative data be found? How could the profit be raised? How

[1] but hardly a plan of a plan.

could it be maximised? If maximisation is an objective, are there other (competing) objectives?

The mathematician starts modeling only (a) after having collected all the necessary information in his mind or on some sheet of paper, and (b) after having defined exactly, what the problem to be solved is. Finally he starts "building" the model by transforming some of his conceptual knowledge into a formal symbolic representation. Some other part of his conceptual knowledge is used to document the mathematical model by legends (declarations of used symbols). The rest of his conceptual knowledge - which is not relevant for this specific problem-solving - gets lost. It is not saved in a materialised form.

The formalised knowledge about the profit of a company - captured by a model concept - is very precious. It answers (for the benefit of newcomers) all the above questions. In addition it is on the way to a model (in a classical sense) that can calculate, simulate, compare or optimise the profit.



```
^^profit                last update                          30.10
  └┬profit after taxes                                    19049.81
   │  └┬profit before taxes                               22411.54
   │   │  └┬►sales                                        99215.39
   │   │   └►costs                                        76803.85
   │   ├tax-rates                                            15.00
   │   └basic tax                                          1000.00
   └►biblio                                                   0.00
```

Figure 1 Explaining tree View

The above View (fig.1) shows a simple explaining tree of elements (notions). It reveals at the first glance that the company's profit can be discussed before and after taxes have been applied. Taxes seem to depend on the amount of the gross profit, on applied tax rates and on a basic tax. The gross profit depends somehow on costs and yields. The arrows at the elements cost and yields indicate that these elements are explained by further elements. Most of the elements need some more comment, have specific properties, are in a quantitative relationship with the successor elements, contain references to other data resources, have many instances, know when they have been created and by whom.

Need of various Views

A model concept of the profit model cannot be inspected by a single (super) View. On the contrary, the more Views a modeling system can provide, the easier complex models can be handled. It is an important field of research to develop a powerful scheme of standard model representation and a rich set of standard Views. Some of the proposed standard Views in what follows have been experimented with on the prototype system NetCalc (see the contribution of Häuschen) or are on the way of being implemented in the succeeding system NetMod.

3.2 Standard Views

Standard Views are the Views of a modeling system that are applicable to every or almost every model that has been created by a modeling system, independent of the structure and purpose of the model. Or by a negative definition: Standard Views are not user-defined Views. So, "standard" does not qualify the general serviceability to different modeling systems but to different models inside a system. They could be called the system Views. Nevertheless we prefer to maintain the notion "standard" because most of them are so fundamental for conceptual modeling that they could become standard Views in the classical sense.

In what follows we will characterise some important standard Views that have been applied with success.

They are:

- explaining tree and outlining feature,
- network representation,
- sorted lists,
- path-View of propagation (of effects),
- modular representation,
- representation of generic dependence (Genus-Graph of Geoffrion),
- representation as attributed graph (Jones[1])
- iconised representation,
- generic Views and Views of instances.

This list is open ended. Proposals to complete it are welcome.

explaining tree

[1] Jones C., An Introduction to Graph-Based Modeling Systems, ORSA Journal on Computing, Vol. 2, No. 2, Spring 1990.

Before defining the notion "explaining tree" the more generally known terms "tree" and "directed network" may be remembered:

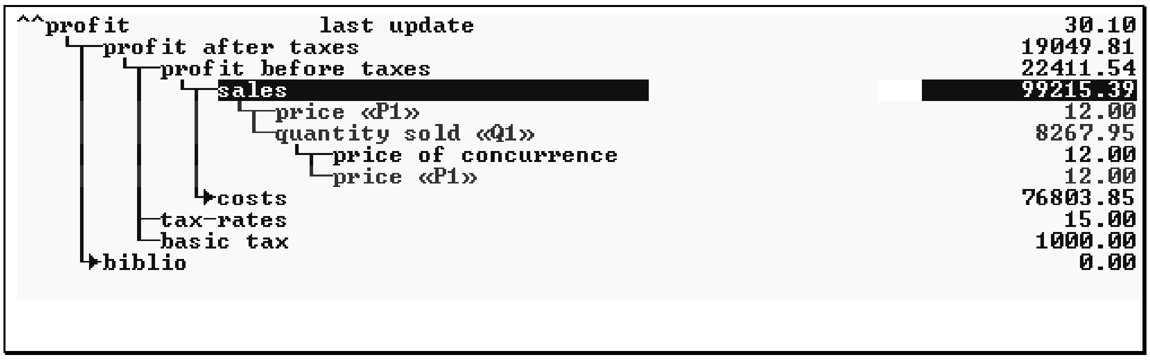
A tree is a hierarchical structure of the relationship parent - child, that is of the cardinal association one to many. Every node can have (but need not) one or more child nodes (direct successors) but at most one parent (direct predecessor). The only node that has no predecessor is called root of the tree. Nodes that have no successors (children) are called the leafs of the tree. A directed network is a generalisation of the tree allowing that every node can have several parents. It can also have several roots. It is called directed because the neighbours of a node can be differentiated either as parent nodes or as child nodes.

An explaining tree is a hierarchical tree-like View of a directed network of nodes. The explaining tree is a crossing-free representation of the directed network. Nodes having n parents must appear n times, once as a child of each of the n parents.

Convention: In what follows the notion element is used frequently in the place of the notion node, which is a term within the graph theory.

The explaining tree is an excellent View for the top-down development of a model. In the top-down approach, more general or aggregated objects are progressively explained by more special or disaggregated things. The parent-child relationship fits well to the setting-up of mathematical models that are composed of explicit mathematical functions yr=f(x1,x2,x3, ... ,xn), i.e. one endogenous and one-to-many exogenous variables. Exogenous variables appearing in different functions cause the network structure of an explaining tree. Exogenous variables that are - in a subsequent step - explained themselves by other exogenous-ones (engendered) increase the depth of the model.

The explaining tree is an excellent View for conceptual modeling if it is combined with the outlining feature, i.e. the possibility to hide the successors of an element. By this option the user can selectively choose the depth of his view to a part of the model.



```
^^profit                last update                          30.10
   └┬profit after taxes                                   19049.81
    │ └┬profit before taxes                               22411.54
    │  │ └┬sales                                          99215.39
    │  │  │ └┬price «P1»                                     12.00
    │  │  │  └quantity sold «Q1»                           8267.95
    │  │  │     └┬price of concurrence                       12.00
    │  │  │      └price «P1»                                 12.00
    │  │  └►costs                                         76803.85
    │  ├tax-rates                                            15.00
    │  └basic tax                                          1000.00
    └►biblio                                                  0.00
```

Figure 2 Explaining tree View - selective opening

An important advantage of the explaining tree View is the fact that any element - except leaf elements - represents all its successors, i.e. a part of the model. By simply manipulating[1] the representing element, the complete sub-ordered part of the model is manoeuvred in the same sense.

Another important aspect of the explaining tree is the possibility of a hierarchical context search: assume you are searching inside a farm model of about 5000 elements the particular element that represents the fertiliser costs. However, you can't find it any more, either by deep-search or by sound-like-search. If the model has a maximal depth of 15 levels and you have an idea of how a farm business model could be structured, there is an excellent chance of finding the wanted element by less than 15 steps of search, starting at the top root of the completely closed outlining-tree of the model. Every step consists of opening the actual node, choosing a favourite child element and move to that element. Business school students that did not know the model were able to find any proposed element in less than 30 seconds, just by benefiting from their context knowledge (economics).

outlining feature

Outlining is a supplementary feature of certain Views allowing to hide the successors of an element or let them appear again. It is an important feature of modern document processing systems. Outlining becomes very interesting for conceptual modeling if all parent elements of a tree can "remember" whether they are closed or opened. If they can so, a sub-tree of a model can be opened (or closed) selectively moving down the hierarchy. In this way, a special view into the depth of the model will result. Since all parents remember their opening status, the whole sub-tree can be closed intermediately and reopened exactly the same way. A specific virtual View has been defined.

A more primitive type of outlining allows only the opening or closing of either all successors (total) or the direct successors of an element (stepwise outlining). Within text processing systems, outlining to a desired level of hierarchy is also provided, since titles are managed within a strong hierarchy. In the prototype system NetCalc all kinds of outlining - except opening by hierarchy levels - have been implemented in combination with the explaining tree View.

[1] deleting, moving, copying, colouring etc.

selective opening --> intermediate closure --> remembered selective opening

--> definitive closure --> stepwise opening

--> total opening

--> opening up to hierarchy level

Figure 3 Outlining Functions

Outlining could somehow be combined with the following Views: network representation, path-View of propagation, modular representation, generic View, attributed graph View. More detail on the implementation of outlining for these Views will be given in a forthcoming working paper.

network representation

Network representation is a somewhat local view of a directed network. It shows a model from the point of view of a selected element being actually the element focused on. From this element a glance at the successors as well as one at the predecessors is given. The parent and the child elements are well defined.

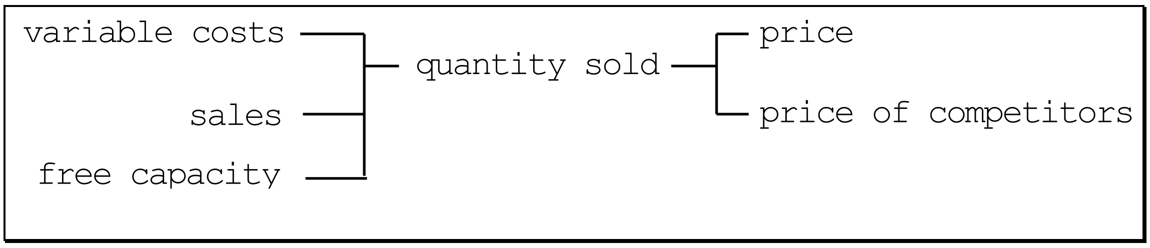


Figure 4 Network View

If two more levels have to be shown, one on the predecessor and one on the successor side, the parent and the child level list must focus each on a single element again.

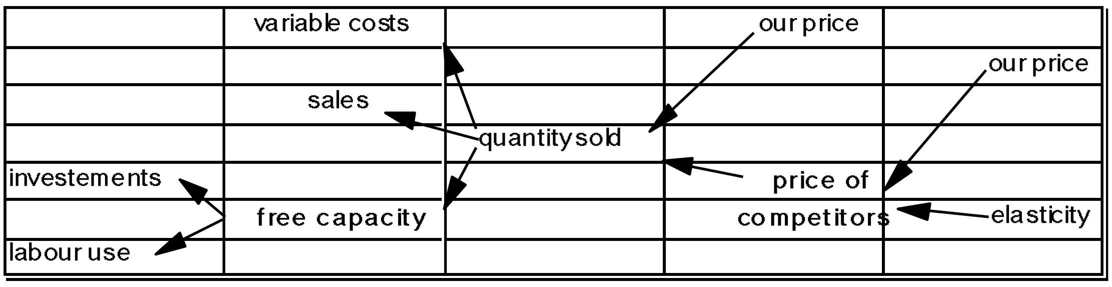


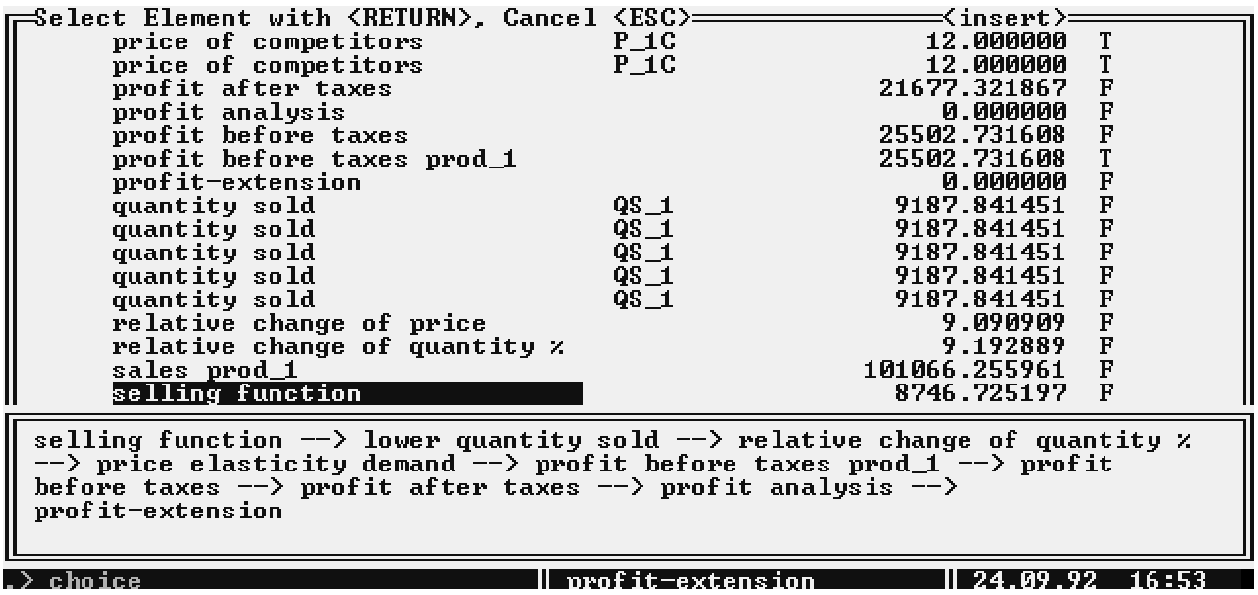
Figure 5 parents of parents and children of children

The network representation gives the best view of the web structure of a model. Furthermore, it is useful for navigating through a model.

sorted lists - filtered lists

A sorted list of the elements of a model - sorted by any of their attribute(s) - is required as soon as the number of elements exceeds one hundred. It shows the association of elements that are similar due to their properties. Along with sorting, a list can be filtered. In this case, only elements satisfying a test will appear in the sorted list. It is very convenient to use poly- and isomorphism's[1] for modular modeling. This may lead to numerous elements of identical name but different meanings in different contexts (areas) of the model. A sorted list can show us all appearances of an element "quantity" in completely different parts of the model (polymorphism), or it can display all instances of a generic element (isomorphism). The index structures that are used to induce sorted lists can also be applied to offer quick search features.

[1] Polymorphism: different elements being placed in different parts of a model (eventually having different meanings) have got an identical name.
Isomorphism: Elements of equivalent (but not identical) meaning being placed in different (but identically structured) parts of a model have got different names.



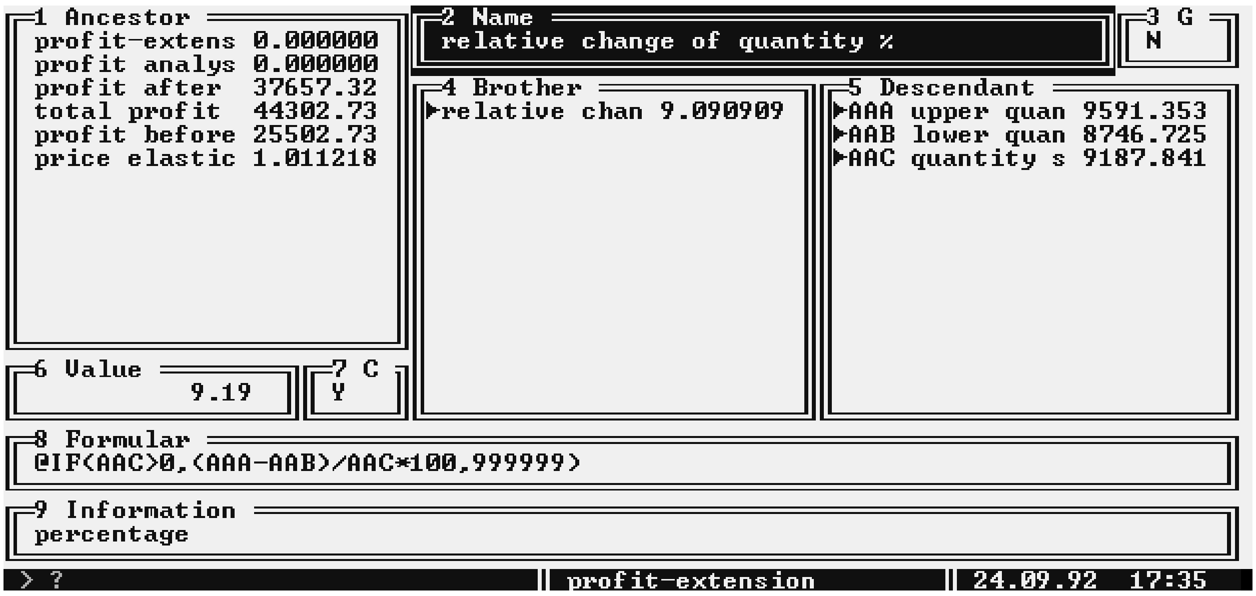
```
Select Element with <RETURN>, Cancel <ESC>                <insert>
     price of competitors          P_1C           12.000000  T
     price of competitors          P_1C           12.000000  T
     profit after taxes                        21677.321867  F
     profit analysis                               0.000000  F
     profit before taxes                       25502.731608  F
     profit before taxes prod_1                25502.731608  T
     profit-extension                              0.000000  F
     quantity sold                 QS_1         9187.841451  F
     quantity sold                 QS_1         9187.841451  F
     quantity sold                 QS_1         9187.841451  F
     quantity sold                 QS_1         9187.841451  F
     quantity sold                 QS_1         9187.841451  F
     relative change of price                      9.090909  F
     relative change of quantity %                 9.192889  F
     sales prod_1                             101066.255961  F
     selling function                           8746.725197  F

selling function --> lower quantity sold --> relative change of quantity %
--> price elasticity demand --> profit before taxes prod_1 --> profit
before taxes --> profit after taxes --> profit analysis -->
profit-extension

.> choice                        || profit-extension        || 24.09.92  16:53
```

Figure 6 Sorted list and path-View of propagation of the selected element

path-View of propagation (of effects)

The structure of a directed network can support both, hierarchical ordering of elements and the propagation of effects (messages) along affected paths in one or the other direction. The visualisation of affected paths by a standard View is very helpful (a) for the identification of polymorphic elements by their hierarchic path notation and (b) for analysing the propagation of effects. Examples of propagation are: forwarding of results of a network calculator, logical reasoning by predicates, blowing up generic structures due to their hierarchical generic dependence, (multiple) inheritance.

In the prototype system NetCalc a path-View of propagation has been integrated within the sorted list View (Figure 6) and within the attribute View (Figure 7, ancestor pane 1).



```
1 Ancestor                  2 Name                                    3 G
profit-extens 0.000000      relative change of quantity %             N
profit analys 0.000000
profit after  37657.32      4 Brother                   5 Descendant
total profit  44302.73      ►relative chan 9.090909     ►AAA upper quan 9591.353
profit before 25502.73                                  ►AAB lower quan 8746.725
price elastic 1.011218                                  ►AAC quantity s 9187.841

6 Value          7 C
          9.19   Y
8 Formular
@IF(AAC>0,(AAA-AAB)/AAC*100,999999)
9 Information
percentage
> ?                          profit-extension          24.09.92  17:35
```

Figure 7 Attribute View

modular presentation

Modular presentation is not only a question of viewing but does also depend on the concept of model representation (see section 4.5). Modules have three main functions:

1. Re-use of model code for faster modeling using model components,
2. Management of model prototyps for coping with complexity,
3. Reduction of redundancy for easier model maintenance.

Furthermore, modules have the following properties: They are self-contained, i.e. their functionality does not depend on the rest of the model structure. From the user's point of view a module is a kind of model that is not determined in its final purpose; or in other words, a model that can be (re-)used for different purposes can figure as a module. Finally, a module has a well-defined interface which is alone responsible for the communication with the environment, i.e. with the rest of the model.

For modular presentation, several standard Views and features are needed if the discussed characteristics should be emphasised. One of them should show any module as a whole (as a unit). For this task, the explaining tree View combined with the outlining feature can be used: one element can represent the rest of the module. Another View should indicate the interface of a module, i.e. the common elements of a module and its host model. Another special feature (marking or colouring) should allow one to distinguish the elements of a module from other elements, and this within most of the standard Views.

If a module is generic (i.e. indexed or instantiable) and is presented as a generator unit, it should be displayable, both as generator element and as a list of instatiated elements (see also generic Views and Views of instances).

If modules are allowed to contain other modules, a model becomes a hierarchical structure of modular dependence. A hierarchy browser should allow the display of a monotone ordered tree of modules, i.e. an outlining structure that guarantees the absence of forward references [Geoffrion[1]].

presentation of generic dependence

Generic structures of models are induced by mathematical indexing or simply by re-use of identical model substructures for a number of cases (instancing). See also 4.4. The embedding of generic structures within other generic structures matches with multiple indexing. Geoffrion[2] has worked out systematically the generic dependence of elements of generic models: Due to Geoffrion the elements of a generic model can be classified into:

1. primitive elements pe, corresponding to Entities in the database world,
2. compound elements ce, corresponding to Relationships in the database world,
3. attribute elements ae, corresponding to Attributes in the database world,
4. function elements fe, corresponding to a function that returns a value, as used in programming,
5. test element te, corresponding to a controlling structure, as used in programming.

The generic dependence of an element must follow the above hierarchy: a primitive element does generically not depend on other elements, a compound element can only depend on primitive elements whereas a test element can depend on any of the element types 1 to 4.

[1] Geoffrion A.M., The Formal Aspects of Structured Modeling, Operations Research, Vol. 37, No. 1, January-February 1989.

[2] Geoffrion A.M., An Introduction to Structured Modeling, Management Science, Vol. 33, No. 5, May 1987.

The generic dependence can be visualised by the Genus-Graph proposed by Geoffrion. The Genus Graph orders all genera[1] of elements of a model into 5 columns according to their element type and shows their dependence by the arrows of a directed network. The arrows start from the generators and point to the generated genera. Figure 8 shows the Genus Graph of the transport problem.

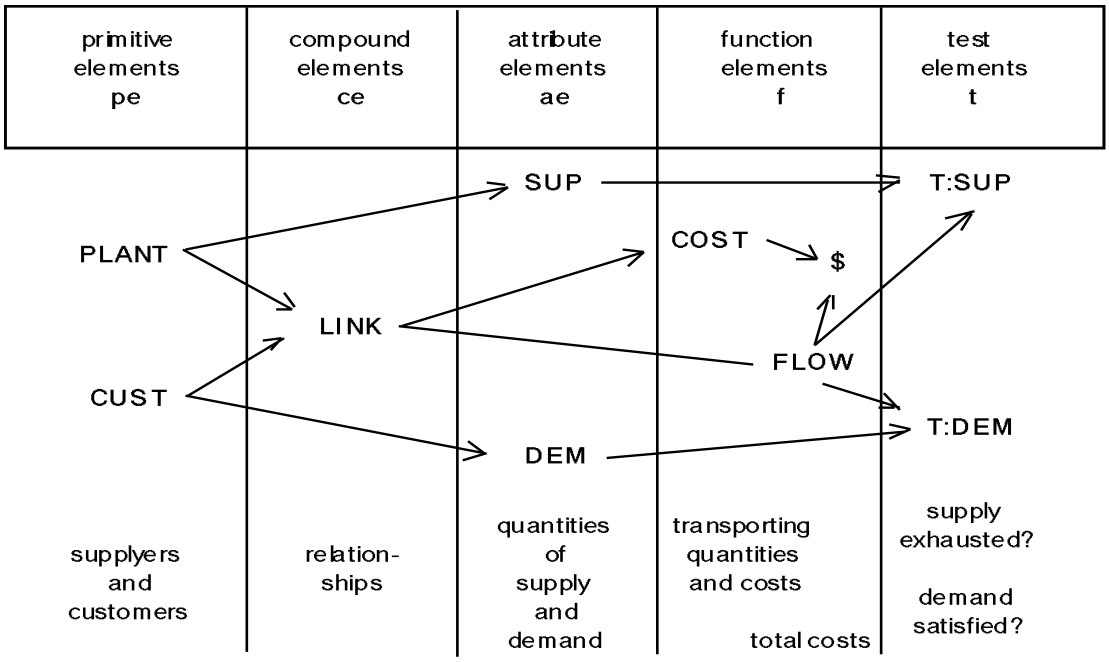


Figure 8 Genus Graph of the transport model

presentation as attributed graph

Attributed graphs [Jones[2]] use the two basic types of elements of graphs, edges and arrows, for the representation of the two basic data structures Entities (or primitive elements) and Relationships (or compound elements). Each of them can hold a list of attributes. An attributed graph allows the coexistence of several edge-types and several arrow-types defined by their specific attribute pattern. In the graphical presentation, edge-type elements must alternate with arrow-type elements. The attribute list of each type can also hold standard or user-defined meta-information defining the graphical appearance and placement of the elements. This is an advantage and a handicap at the same time. The potential for problem tailored graphical presentation can not be exhausted by a general standard View. On the contrary, a default placement of edges and arrows of an attributed graph by a standard View generally turns out to be worthless. As a consequence an attributed graph has to be considered as a user-

1 sets of elements, classes of objects.

2 Jones C., An Introduction to Graph-Based Modeling Systems, ORSA Journal on Computing, Vol. 2, No. 2, Spring 1990.

defined View belonging to the external scheme, and it must be offered as a graphic View that is easily editable by mouse dragging. The attributed graph View is like the spread-sheet View a kind of standard scheme for user-defined Views. Such standard schemes can offer a lot of standard functions for the viewing but not a fully automated generation of a convenient View.

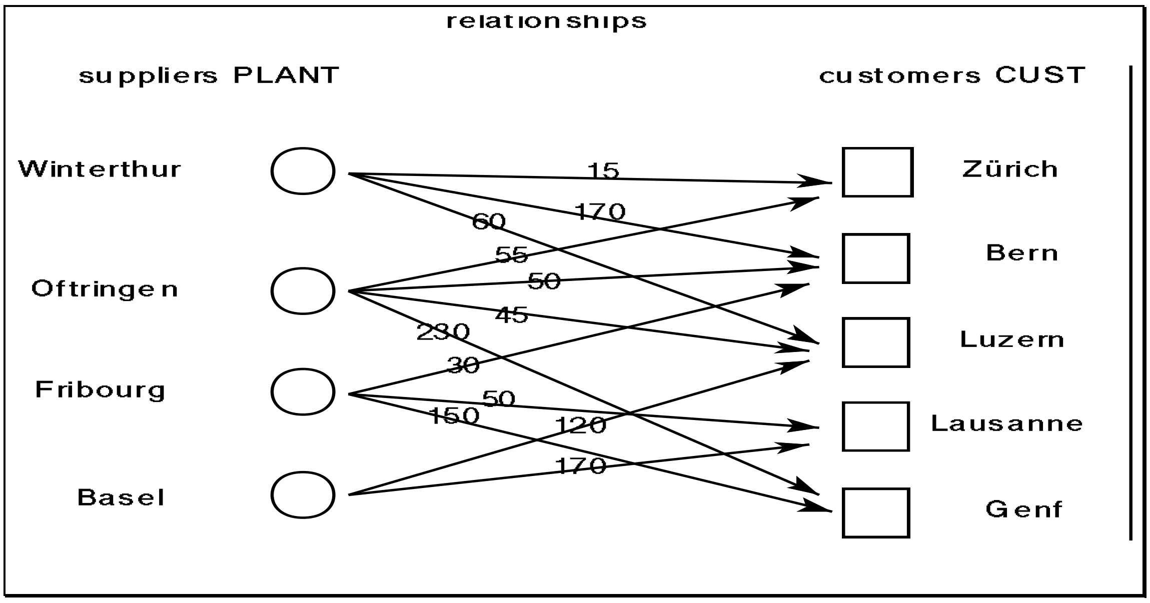


Figure 9 attributed graph of the transporting model

The LPN-graph which has been applied within the graphical modeling tool gLPS (see relative contribution in this edition)[1] is a special form of an attributed graph that suits well for representing linear programming models.

iconised representation

NetCalc has been used for modeling within the Decision Support System in Agricultural Policy. Real world applications led to models of respectable size (>1000 elements). Although the access to any element of large models is fully guaranteed by explaining-tree View, sorted lists, quick and deep search features, a practice of placing so called label elements close to cardinal elements has come about. As soon as the number of such "labels" became important, the modelers started building a kind of hierarchic survey model using those entry-point-labels.

[1] Collaud G., Pasquier-Boltuck J., gLPS: a Graphical Tool for the Definition and Manipulation of Linear Problems, working paper No. 197, Institut pour l'Automation et la Recherche Opérationnelle, Université de Fribourg (CH)., 1992.

This work could be done automatically by a standard View that collects such labels, puts them in an iconised hierarchic outlining View. The appearance of such icons could be user-defined. It could show the value, the relative change, the modification date, or other information needed for model controlling. Such a standard View could give a kind of pictographic overview of what is considered important by the modeler or the model user.

generic Views and Views of instances

NetCalc facilitates the re-use of model-code (parts of the model) by many features, but it is not generic. This very soon leads to structural redundancy, i.e. to many sub-models of identical structure but different properties. As soon as such a physically copied sub-model has to be refined, a laborious and error-prone job of model maintenance must be done. The search of each appearance of the substructure is painful and the adaptation work is repetitious. Therefor it is necessary to pass to generic modeling (re-use of identical structures through instancing).

In a modeling system that supports generic modeling it is very important that the modeler can switch at any moment between the generic View and the instance View of an actually selected standard View. A conclusion for the developpers is this: Generic and instance Views are not really Views on their own, but they are a supplementary dimension to implement with every (or almost every) standard View provided by a modeling system.

3.3 Standard Functions

Standard functions are those functions a modeling system can provide for any model, regardless of its structure and purpose. They could be standardised (in a strong sense) among different modeling systems once a standard for model representation has been found.

The standard functions can be grouped into the following collections:

Model management: list models, describe.. , select.., create .., delete.., copy.., rename.., export.., import model.

Micro-modeling: (defining new structures) select a standard View, list element classes, create.., modify element class, create element of class, copy.., edit.., move.., delete.., arrange element.

Macro-modeling (i.e. using pre-defined structures by module management): create.., re-use (exemplify).., specify.., power-use.., delete module, show.., select.., inspect.., specify.., delete.., arrange instance.

Most of the functions for model management and for micro-modeling are well known from spread-sheet systems. The management of element classes is known from object oriented programming or from abstract data types of classical programming or from database design.

The macro-modeling functions are more interesting to describe because they are innovative for interactive modeling systems. They are based on mathematical indexing and the management of index sets, but they are intended to discharge the modeler from formal indexing by transforming this painful task into a more natural job of re-using existing parts of the model either identically or just in an analogous way (homomorphisme[1]). The macro-modeling functions will be implemented

in the successor system of NetCalc called NetMod. A more detailed description of both concept and implementation of these functions will be given in a following paper.

The macro-modeling functions are central for introducing generic features into interactive modeling systems. In addition, this combination of interactivity and generic representation is crucial for really getting support in the modeling process. This is the subject of the following section.

4 Generic Modeling without Mathematical Formalism

4.1 Need for generic modeling:

There is a need for generic modeling both for technical reasons and for reasons of efficient (economic) use of modeling system. The technical need comes from the objective to separate specific data from the general structure of a model. This again is necessary to avoid redundancy which in turn leads to inconsistency and to anomalies of models, precisely as in data base systems. The economic need arises from the necessity to re-use model-code (modules, parts of models) as much as possible, because modeling, when starting from zero, is a difficult, time consuming and error-prone job.

However, there is another, very important reason for laying stress on generic modeling: There is a natural need for it because of people's example-based

[1] Homomorphism: Elements of equivalent (but not identical) meaning being placed in different (but similar structured) parts of a model have got different names.

reasoning.[1] Most people first try to recognise whether the problem to solve is just a new case (example, version) of a known problem. In the positive case, it's easy as well as economic to create a new instance of an existing model and to specify it (adapt it to the particular case). Most of the actually occurring problems are of this first category !

If the problem is not of this first category people will try, as a second strategy, to find out whether the encountered problem is similar to a known problem, or in other words, the problem turns out to have a slightly altered structure of a known problem. In this case, people will start from the model structure of a similar problem and modify it until they think that it fits well the problem faced with. Within daily affairs, problems of the two categories mentioned will cover the overwhelming part of all problems encountered.

Problems of the third category are finally those that are completely new. They have to be analysed, represented, formalised and synthesised as new models, and they have to be tested. This third category is the domain of analysts or research specialists. These people are experienced in modeling. Most of them regularly apply mathematical methods.

Since it is our vision to open the world of computer assisted modeling to a large public of PC users, the re-use of models and modules is of central interest. It's a fact that generic structures managed by mathematical indexing techniques are an efficient means to cope with the problems of model-code recycling. Yet the users addressed cannot be expected to apply mathematical indexing. For this reason another, more natural (intuitive) way of generic modeling, which hides an automated mathematical background indexing, had to be found.

4.2 A natural way of generic modeling

Generic modeling can be derived from the natural need to create a new example of an existing model or object (i.e. data structure) or model part. This case might arise if a modeler has discovered that the structure of a part of an existing model fits well the new problem context. Now there are two possibilities: Either (a) the selected sub-structure fits exactly or (b) it has to be altered slightly. In the first case the selected substructure has only to be interpreted or evaluated using new data. Technically this can be arranged by a background process of

1 Angehrn, A.A., Modelling by example; neue Wege zur Entscheidungsunterstützung mittels interaktiver Modellierung, Diss. Math. ETH Zürich Nr. 8864, 1989.

indexing and instancing. Within this case, there are two sub-cases to distinguish:

(a1) It's the first time the selected substructure is re-used. In other words, there are actually two cases for which the substructure should be used. The modeler is asked by the system to give a general notion that contains both cases: for example, product for butter and milk, or location for Bern and Zürich. The substructure will be indexed by the system using this notion, but under control of the modeler: That is, the modeler communicates which of the elements are case-sensitive, i.e. will systematically change for different cases. Those elements are indexed by the system. The others, which do not change for different cases, are not indexed.

A new instance is generated by the system. It is named, using the notion that characterises the second case (milk or Zürich for the examples above mentioned) and the modeler is invited to edit data of the indexed elements (i.e. to adapt them to the new case). Non-edited attributes keep the values of the first case figuring thus as default case.

(a2) The selected substructure was used several times before hand and is consequently indexed already. A new instance is generated and the data of the indexed elements can be edited (adapted to the new case).

(b) The substructure being re-used has to be altered slightly. In this case, a simple instancing of an indexed structure is not sufficient. First a new editable "copy" of the substructure in discussion must be generated. Yet this act is not a simple copying, but a differentiated generation of a re-usable module. During its generation it has to be decided whether the not-indexed elements (i.e. the elements that are not case sensitive) are identical with or different from the original ones. In other words, they are declared to be either of completely global character or just of global character within the new context. Concerning the indexed elements the modeler is invited to decide whether they shall "obey" the default index set[1] or a new index set which has to be named thus. A new index set can be decided either to be empty or to contain the same items as the default index set. In both cases the new index set can be altered adding or deleting items. The modifications, however, will only influence the new module and its instances.

Summarising this interactive process of creating a module: The new module has got from the original model part:

[1] The first index set was created because of the existence of two different cases (instances). It's a unnamed index set that will always figure as the default index set.

1. the generic structure, and the relational structure to be used as the initial (but editable) structure,

2. possibly some identical (global) elements or more precisely, referencing parameters of global elements of the donator model.

3. possibly referencing parameters of the default index sets[1] or, if new index sets have been agreed, perhaps the items of some default index sets, being used as starting items.

In the succeeding modeling process the module can be changed as follows:

(1) in its relational structure, that is: new elements can be associated (introduced), and elements can be deleted, but the generic structure of existing elements may not be changed,

(2) items of the (named) new index sets can be added or deleted.

Both, indexing and creating modules result from a system-supported re-use of a model part inside a model at a new place. The re-use of a not-indexed module that has no referencing parameters (of global elements), returns the trivial case of a simple (physical) copy of a substructure. Figure 10 shows the re-using of a sub-tree by physical copy, induced instancing, induced indexing, explicit indexing, and explicit instancing. Finally it shows the creation of a module assigning a new index set to it.

Modules, after having been altered, can be re-used (a) by importation into a model unifying referencing parameters and (b) through indexing of the imported (or linked) module.

1 The plural is used because multiple indexing is possible.

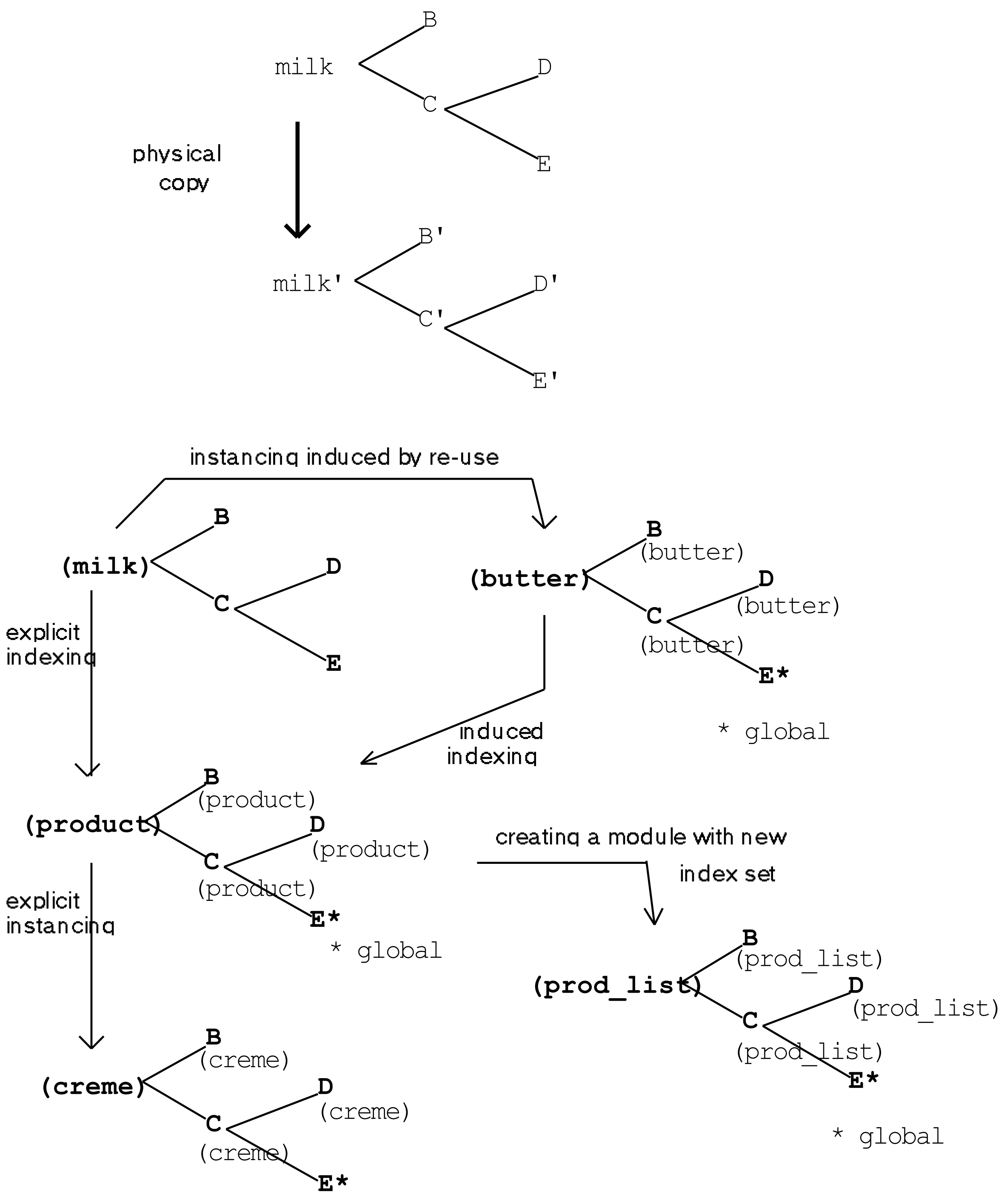


Figure 10 re-use of structures by copying, instancing, indexing, and exporting to modules

5 Modular Modeling

First it is necessary to define the central notions object, module and model.

Objects: In the modeling context, an object is defined as in an object-oriented language: it is an example (instance) of a class of Entities of identical data structure and method collection. Objects are members of an object class hierarchy and inherit data structure as well as methods from upper classes within the hierarchy.

Modules: In the modeling context a module is defined as a collection of objects and binary relationships between those objects. The objects contained in a module may be instances of different classes (i.e. different element types) and also subclasses of indexed collections (i.e. class objects).

Models: Models are composed of objects and modules. They have been assigned a role in a problem context.

Modules are very similar to models. There is no difference in their architecture. Modules, however, tend to be smaller units, explicitly designed for reuse. They have the character of components that are supposed to be used for composing more complex models. Models however are built to solve problems, to answer questions, to be applied. Models must, therefore, be accessible through application-oriented Views. Modules need not. Isolated modules have no practical significance. Modules, however, are more highly organised units than objects are. They resemble much to procedures in the world of procedural programming.

Modules have an important role regarding the re-use of model parts within other models of perhaps other modeling systems. They are the means for porting model code for reuse, independent of application oriented external Views. For this purpose a good standard for model representation is needed, as has been stated earlier in this paper.

The difference between modules and models becomes obvious at the moment a module has to be extracted from the host model and deposited in the model library. A sub-structure, as long as being part of a model, may point to global elements of the model. The exported module receives a set of copies of the global elements of the host model. It has become a complete model. The copied global elements of the module, however, will serve as referencing parameters during importation into a model. Once the module is re-imported into a model, the modeler must decide whether the global elements of the module or those of the receiver model have to be kept (applied)[1]. During importation, the modeler also has to choose between available index sets, unless he wishes the default index set to be used.

Another fairly subtle difference can be noted if a module is generic. Inside a host model a module can be instanced by the (unnamed) standard set and moreover by one or several named sets. Outside a model, i.e. as part of the model library, a module is not accompanied by sets of instances. It appears in the generic (indexed) form. As a consequence, an isolated module can be instanced only by importing it into a model. Once imported, it can get more instances either by interactive module re-use, or data-driven by named index sets.

Finally, the main role of modules is to break down complexity of a model. A module is supposed to be an encapsulated tested unit with a well-defined interface[2] that can be handled like a black box. A model can then be viewed by a limited number of modules instead of a collection of a mass of elements. In order to favour of these advantages some standard Views should emphasise this modular representation (see 4.3).

6 Conclusion

During the last decade, mathematical modeling tools have become available on the PC software market in parallel to the successful spreadsheet packages. These modeling tools have significantly increased the productivity of academic modelers, but have not yet facilitated modeling in a way such as to render modeling accessible or even profitable for a larger public. Most of the modeling systems use a representation scheme that is close to mathematical notation. They have the advantage of a short, concise, and generic syntax, but they allow neither getting easy insights into a model nor do they really assist the modeling process. Most of the knowledge applied by the modeler during the transformation and abstraction of a real world problem into a formal model gets lost. The application of this kind of formal models is extremely limited because it depends on a continuous support by a modeler, in general even by the author himself.

---

[1] N.B.: A global element can represent a global (sub-)structure. The global structure of both the module version and the version of the receiver model can differ because of their separate development.

[2] The list of global objects constitutes the interface.

The mathematics-oriented modeling systems ignore the increased potential of modern computer systems for knowledge representation. Models built and managed by computer systems should represent as much as possible of the knowledge of a problem or its context. Moreover, they should give easy access to this knowledge by offering various expressive Views of it. Each of these Views may facilitate the "use" of the modeled knowledge. Standard Views can help accessing, analysing, modifying the modelled knowledge itself, whereas application oriented Views can help applying models (for problem solving) and interpreting results. Mathematics-oriented presentations of some modeled knowledge are very useful as Standard Views for analysing a model, perhaps they could serve as a basic representation scheme, but they are not a promising View for modeling (model development).

In this paper, we have discussed essential concepts of a modeling system that could be used by a large public:

1. the role of a triple representation scheme (conceptual, internal, external) and the related research activities,
2. proposals of standard Views and functions for conceptual modeling,
3. the possibility to support generic modeling interactively in such a way that users need not know how to cope with mathematical formalisms,
4. role of modules within modeling and related macro modeling functions.

These concepts are on the way of being implemented in the successor system of the prototype NetCalc.

References

Angehrn, A.A., Modelling by example; neue Wege zur Entschei-dungs-unterstützung mittels interaktiver Modellierung, Diss. Math. ETH Zürich Nr. 8864, 1989.

Brooke A., Kendrick D., Meeraus A., GAMS: A Users Guide, The Scientific Press, Redwood City, CA, 1988.

Collaud G., Pasquier-Boltuck J., gLPS: a Graphical Tool for the Definition and Manipulation of Linear Problems, working paper No. 197, Institut pour l'Automation et la Recherche Opérationnelle, Université de Fribourg (CH), 1992.

Elmasri, R. and Navathe S.B., Fundamentals of Database Systems, The Benjamin / Cummings Publishing Company, Inc. Reedwood City ,California, 1989.

Fourer, Gay, Kernighan, A Modeling Language for Mathematical Programming AMPL, WP, Technical Report No. 133., published in Management Science, Vol. 36, No. 5, May 1990.

Geoffrion A.M., An Introduction to Structured Modeling, Management Science, Vol. 33, No. 5, May 1987.

Geoffrion A.M., The Formal Aspects of Structured Modeling, Operations Research, Vol. 37, No. 1, January-February 1989.

Greenberg H.J., A Primer for MODLER: Modeling by Object-Driven Linear Elemental Relations, University of Colorado at Denver, Mathematics Department, Denver, CO, 1990.

Häuschen H., NetCalc, Modellier- und Kalkulationssystem für vernetzte Modelle, working paper No. 189, Institute of Automation and Operations Research, University of Fribourg (CH), 1992.

Hürlimann T., Kohlas J., A Structured Language for Linear Modeling, OR-Spectrum, Vol.10, p.55-63 , Springer Verlag,1988.

Hürlimann T., Reference Manual for the LPL Modeling Language, Version 3.8, June 1990, Working Paper No. 191, Institute for Automation and Operations Research, University of Fribourg / Switzerland, 1992.

Jones C., An Introduction to Graph-Based Modeling Systems, ORSA Journal on Computing, Vol. 2, No. 2, Spring 1990.

Singh I.S., A Support System for Optimization Modelling, North-Holland, Decision Support Systems 3 (1987) 165-178.

6 NetCalc

H. Häuschen

1 Überblick

NetCalc ist ein Software Produkt, das für PC's implementiert und bereits in Gebrauch[1] ist. Es ist ein integriertes System, das eine rasche, einfache Modellierung und Auswertung verschiedenster Modelltypen erlaubt. Es kann als eine wichtige Ergänzung zu Tabellenkalkulationen wie Framework III[2] oder Excel[3] eingesetzt werden. NetCalc legt das Gewicht auf die Modellierung (Erfassung und Erstauswertung) von Problemzusammenhängen. Diese ist weitgehend einheitlich für alle Benutzer und Probleme. Die Auswertung ist ebenfalls einheitlich und kann bei Bedarf in Verbindung mit einer Tabellenkalkulation nach eigenen Wünschen gestaltet werden.

NetCalc zeichnet sich durch folgende Merkmale aus:

- rasche Erstellung eines Modell-Prototyps,
- computerunterstütztes strukturiertes Modellieren,
- interaktive Sofortauswertungen auf beliebigen Aggregationsstufen,
- Arbeiten mit Modulen (Austausch von Modulen zwischen den einzelnen Modellen und den verschiedenen Benutzern),
- Datenorganisation in Datenbanken,
- offene Programmarchitektur (Erweiterung des Funktionsumfanges im Programm durch den Benutzer) und eine benutzerfreundliche Oberfläche.

[1] NetCalc wird im Bundesamt für Landwirtschaft zur Modellierung aktueller agrarpolitischer Fragen eingesetzt.

[2] Framework III ist ein Produkt der Ashton Tate GmbH.

[3] Excel ist ein Produkt der Microsoft GmbH.

Die Grundlage der Modellierung in NetCalc ist ein standardisiertes (einheitliches) Modelldarstellungskonzept. Diese Standardisierung garantiert, dass verschiedenartige Probleme von unterschiedlichen Benutzern gleich dargestellt werden, was das Arbeiten in Gruppen wesentlich erleichtert. Dem Aussenstehenden ermöglicht es des weiteren, die Beziehungen zwischen den Grössen im Modell zu erkennen und mit ihnen zu arbeiten. Die Entwicklung der Modelle geschieht vor allem *'top down'*. Bei der *'top down'* Entwicklung wird eine aggregierte Grösse in mehrere Komponenten aufgespalten. Es entstehen dann, ausgehend von jeder Schlüsselgrösse, sogenannte Erklärungsbäume bzw. -netze. Die Modellierung des Problems muss nicht vollständig sein, sondern kann etappenweise formuliert und Berechnungen oder Auswertungen auf jeder Entwicklungsstufe vorgenommen werden. So ist dafür gesorgt, dass die Modelle jederzeit ohne grossen Aufwand ergänzt, verfeinert bzw. verändert werden können.

Die Modellierung und Auswertung eines Problems geschieht derart, dass der Benutzer in einer ersten Phase beginnt die ihn interessierenden Grössen durch andere Grössen zu erklären. Er muss sich dabei nicht um eine auswertungsbezogene Anordnung der Elemente kümmern, wie dies z.B. bei Tabellenkalkulationen der Fall ist. Die einzelnen Elemente können durch algebraische Ausdrücke zueinander in Beziehung gebracht werden. Die interaktive Sofortauswertung erlaubt dabei eine erste Kalkulation und Simulation. In einer nächsten Phase werden die gewünschten problemorientierten Sichten erstellt, die je nach Fragestellung nur die relevanten Elemente ins Blickfeld rücken und diese zusätzlich durch Text oder Graphiken beschreiben. Hier können nur die Werte der Elemente verändert werden, die Beziehungen unter den Elementen bleiben unverändert. Die Sichten dienen zur Dokumentation, bzw. als Eingabemedium für die Werte (im Sinne eines Formulars). In den nächsten Phasen kann das Modell erweitert bzw. verfeinert werden. Die Sichten bleiben hierbei gleich gestaltet oder können verfeinert werden.

NetCalc möchte die erfolgreiche Tabellenkalkulation um die Funktionalität der strukturierten Modellierung ergänzen, um deren Schwächen in diesem Bereich aufzuheben. Die Probleme von Tabellenkalkulationen liegen vor allem in folgenden Punkten:

- das Auswertungsschema ist zugleich das Modell;
- treten neue Fragestellungen beim bestehenden Modell auf, muss meist die bestehende Tabelle überarbeitet und neu gestaltet werden;

- überschreiten die zu bearbeitenden Probleme eine gewisse Komplexität bzw. sind die Zellen untereinander stark vernetzt, geht leicht der Überblick verloren;
- die Wartung, Erweiterung und Überprüfung eines grossen Modells kann meistens nur mit Mühe gewährleistet werden.

Um die Bedienung und das Erscheinungsbild an bestehende Software anzugleichen und neuen Anforderungen gerecht zu werden, wird NetCalc zur Zeit unter Windows 3.1[1] reimplementiert. Das zukünftige NetCalc will vermehrt die Modellstruktur, Instanzen von Modellen und Sichten auf Modelle und deren Instanzen voneinander trennen. Das Ergebnis ist eine Modellstruktur für beliebig viele Instanzen mit differenzierten Sichten. Diese Richtung wird auch von verschiedenen anderen Autoren (Greenberg [4], Geoffrion [2]) verfolgt.

Dieser Artikel, soll einen Überblick über das bestehende Modellierwerkzeug NetCalc geben und die Möglichkeiten und Einsatzgebiete aufzeigen (Abschnitt 2). Ferner sollen Ausblicke auf das neue NetCalc unter Windows 3.1 eröffnet werden, welches zur Zeit in Entwicklung ist, und sich vor allem als abgerundetes Modelliersystem präsentieren wird (Abschnitt 3).

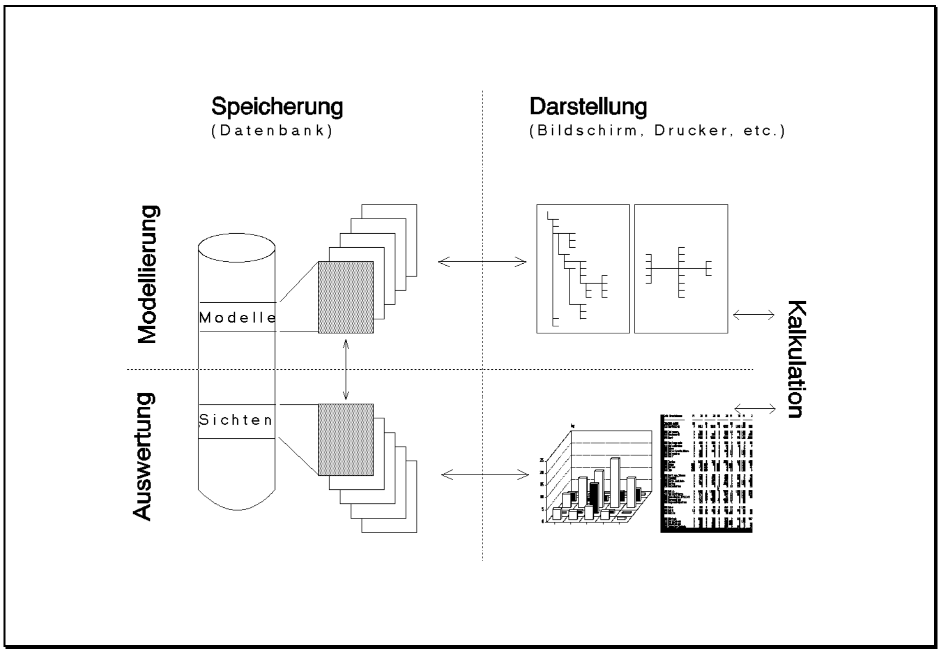


Abbildung 1

1 Windows 3.1 ist eine Benutzeroberfläche der Microsoft GmbH. Diese Benutzeroberfläche setzt sich zunehmend auf PC's durch und kann daher als Standard betrachtet werden.

2 Einführung in NetCalc

2.1 Ein Modell in NetCalc

Generell kann ein Modell als Abbildung eines Teils der Realität betrachtet werden, das einem Anwender hilft *'sein Problem'* darzustellen und zulösen. In diesem Falle mit Hilfe eines Computers. Warum er sein Problem mit Hilfe eines Modells löst, soll an dieser Stelle nicht weiter erörtert werden. Der Prozess des Modellierens ist nie ganz abgeschlossen und stellt oft ein intuitives Vorgehen dar. Er besteht darin, dass der Anwender mit einem groben Modell seines Problems beginnt, und dieses, nachdem ihm die Zusammenhänge klarer werden, schrittweise verfeinert. Dabei muss er das Modell aus verschiedenen Blickwinkeln betrachten können. Diese unterschiedlichen Betrachtungsweisen sollten automatisch generiert werden können, sowie auch manuell nach Wünschen des Benutzers gestaltbar sein.

NetCalc versucht nun diesen Modelliervorgang optimal zu unterstützen, indem es dem Benutzer die Umgebungen für die Modellierung und die Erstellung der Sichten mit den entsprechenden Werkzeugen und Methoden zur Verfügung stellt.

In NetCalc stehen dem Anwender zur Modellierung Elemente zur Verfügung, die er untereinander vernetzt d.h. zu einander in Beziehung bringt. Diese Vernetzung in NetCalc ist hierarchisch. Funktionale Beziehungen können zwischen einem Element und seinen direkten Nachfolgern definiert werden. Die Auswertung erfolgt grundsätzlich in Richtung der Wurzel. (Abbildung 2).

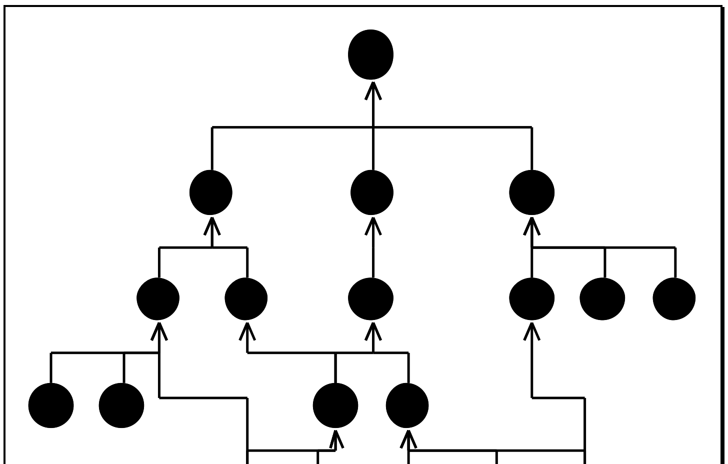


Abbildung 2

Das Element kann Träger folgender Informationen, die im weiteren auch Attribute genannt werden, sein:

- Name
- Abkürzung
- Kommentar
- Wert
- Formel zur Berechnung des Wertes

Jedes Element (mit Ausnahme der Wurzel) ist mindestens einem Element untergeordnet.

Zur Darstellung der Modelle kann der Anwender während der Modellierung zwischen vier standardisierten Sichten hin- und herschalten, die ihm unterschiedliche Einblicke in das Modell und die einzelnen Elemente erlauben.

- Attributsicht
- Baumsicht
- Listensicht
- Netzsicht

In diesen verschiedenen Sichten wird modelliert und es können erste Auswertungen gemacht werden. Zur Dokumentation, Simulation und Eingabe von Werten kann der Benutzer zusätzlich mit Hilfe der Tabellenkalkulation weitere Sichten erzeugen, die nur noch die Elemente enthalten, die ihn wirklich interessieren.

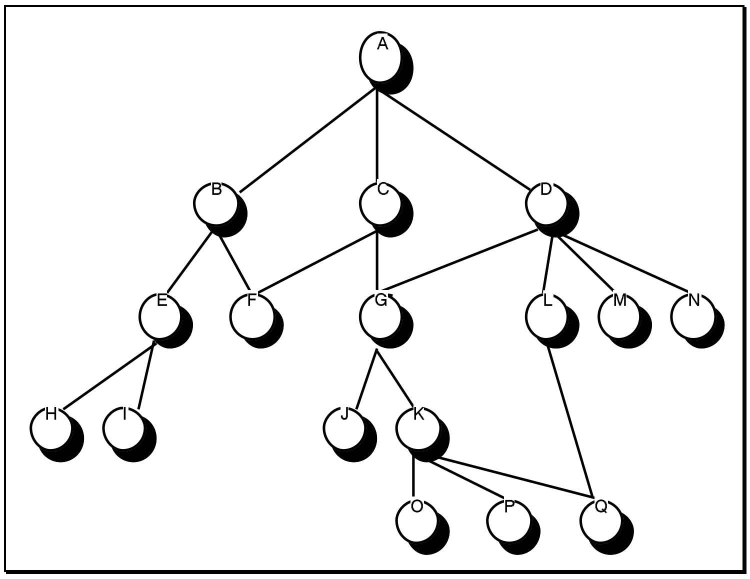


Abbildung 3

2.2. Begriffe

Im weiteren werden folgende definierte Begriffe verwendet. Sie sind in Abbildung 3 verdeutlicht.

Wurzel:	Ausgangspunkt des Modells (A).
Vorgänger:	alle Elemente, die auf der Verbindung zwischen Element und Wurzel liegen (z.B.: Vorgänger von E sind B und A).
Nachfolger:	alle Elemente, die auf ein Element folgen (z.B.: Nachfolger von B ist E, F, H und I).
Exogene:	Elemente, die nicht weiter erklärt werden, also keine weiteren Nachfolger haben (z.B.: H, I, F, M, ...).
Endogene:	auch aggregierte Grössen genannt; Elemente, die durch andere umschrieben werden (z.B.: A, B, C, D, E, ...).

2.3. Modellierung in NetCalc

2.3.1. Allgemein

Die Modellformulierung bzw. -erstellung ist im Gegensatz zu einer Tabellenkalkulation klar von der nachgelagerten Modellauswertung und Resultatdarstellung getrennt. Auf diese Weise kann eine weitgehende Automatisierung im Bereich der strukturierten Modellierung gewährleistet werden, und eine effiziente Unterstützung in der späteren problemorientierten Modellauswertung erfolgen.

Im jetzigen NetCalc kommt das strukturierte Modellieren vor allem in der Baum- und Netzsicht zum Ausdruck. Das Strukturieren ist als Gliederung bzw. Einordnug der Elemente in ein Modell und nicht im Sinne einer formalen Sprache, wie bei Geoffrion [2] [3], zu verstehen, denn es gibt nur einen Elementtyp. Erst in der neuen Version ist eine Ausweitung auf Element- und Modellklassen vorgesehen. Zur Zeit ist lediglich die Darstellung der Modelle strukturiert. Diese ist einfach und derart ausgelegt, dass der Anwender sie intuitiv erfassen kann. Er soll sich nicht mit dem Ballast strenger Formalismen zur Erstellung von Graphen (eine graphenbasierte Modellierumgebung wird z.B. bei Jones [6] beschrieben) plagen müssen.

2.3.2. Vorgehensweise

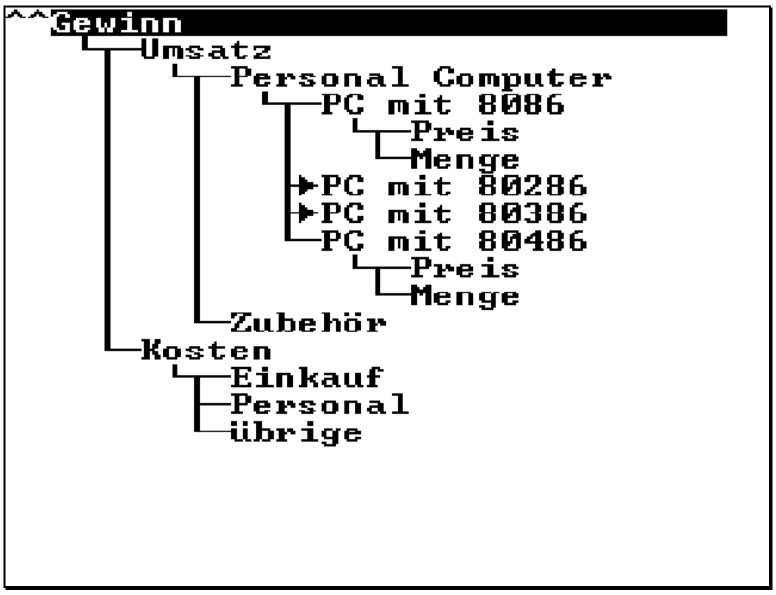
Die Modellierung ist standardisiert und geschieht in erster Linie *'top down'*. Man beginnt mit der Definition der interessierenden Grösse und ordnet dieser die erklärenden Grössen als Nachfolger zu. Dabei können die Elemente mit selbsterklärenden Namen versehen werden.

z.B. erklärt sich *Gewinn* aus *Umsatz* und *Kosten* (Abbildung 4)

Wie sich der Gewinn aus seinen Komponenten dann konkret errechnet, muss an dieser Stelle noch nicht definiert werden. Die Endogenisierung wird solange fortgesetzt, bis die gewünschte Genauigkeit im Modell erreicht ist (vergl. mit den übrigen Elementen in der Abbildung 4, wie z.B. *'Umsatz'*).

Die Zuordnung einzelner Elemente zueinander erfolgt somit bei der Modellierung explizit durch Einordnung in den Erklärungsbaum (-netz). Diese Vorgehensweise ermöglicht es, dass die Elemente gleiche Namen haben können (beachte das Element *'Preis'*, welches in der Abbildung 4 mehrmals auftritt). Um welches Element es sich handelt ist aus dem Kontext ersichtlich (Kontext = Weg, der zu diesem Element führt)[1]. Eine Beziehung entsteht bei der Erstellung eines neuen und durch Zuordnung eines bereits bestehenden Elements. Beziehungen können ebenso durch Verschieben sowie durch Kopieren eines Elements mit all seinen Nachfolgern definiert werden. Jedes Element kann beliebig viele Nachfolger und Vorgänger besitzen. Es besteht zudem die Möglichkeit, zyklische Abhängigkeiten (Feedbacks, simultane Gleichungssysteme) zu definieren.

1 Diese Eigenschaft (dass Elemente gleich heissen können) ist vor allem in grossen Modellen und beim Kopieren von Teilmodellen von grosser Bedeutung. Man hat nicht immer den Zwang für ähnliche Sachverhalte neue Namen finden zu müssen.



```
^^Gewinn
  └┬─Umsatz
   │ └┬─Personal Computer
   │  │ └┬─PC mit 8086
   │  │  │ └┬─Preis
   │  │  │  └─Menge
   │  │  ├►PC mit 80286
   │  │  ├►PC mit 80386
   │  │  └─PC mit 80486
   │  │    └┬─Preis
   │  │     └─Menge
   │  └─Zubehör
   └─Kosten
     └┬─Einkauf
      ├─Personal
      └─übrige
```

Abbildung 4

2.3.3. Sichten

Die Standardisierung im Modellierprozess wird nicht nur funktional durch die *'top down'* Methode unterstützt, sondern vor allem durch die vier gleichbleibenden Sichten auf das Modell, zwischen denen der Benutzer jederzeit hin und herschalten kann (*'Switching'*). Diese vermitteln ihm unterschiedliche Informationen über die Modellstruktur und deren Elemente (*'Zooming'* und *'Condensing')*. Die Funktionen 'Switch', 'Zoom' und 'Condens' sind grundlegend in einer Modellierumgebung *(*Greenberg [4]).

Die Erklärungsbaumsicht (Abbildung 4) stellt das Modell graphisch als Erklärungsbaum dar und dient der besseren Übersicht. Die einzelnen Äste können wie in einer Gliederung beliebig auf- bzw. zugeklappt werden, damit das Modelle mehr oder weniger detailliert darstellt werden kann (in der Abbildung 4 sind die zugeklappten Elemente mit dem Symbol '>' gekennzeichnet).

Die *Attributsicht* (Abbildung 5) zeigt in neun verschiedenen Fenstern die komplette Information eines Elements (Attribute, Vorgänger, Nachfolger, Struktur). Normalerweise werden über diese Sicht die Informationen eingegeben, verändert oder gelöscht.

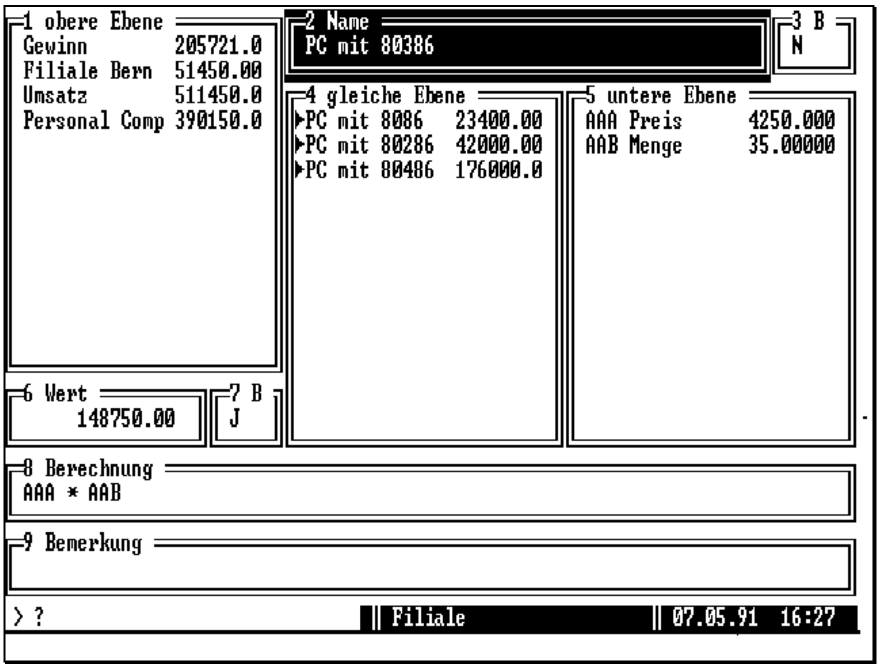
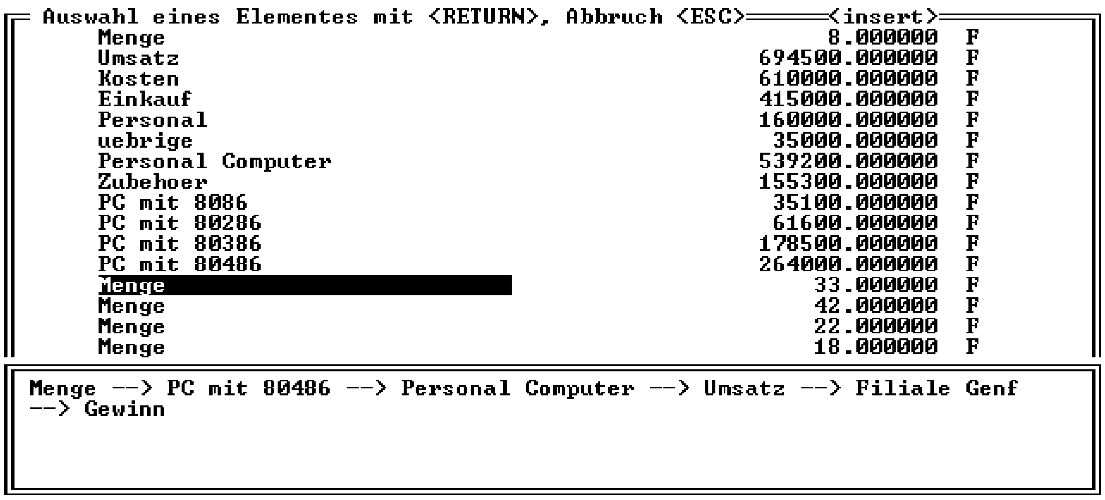

```
1 obere Ebene                 2 Name                                3 B
Gewinn          205721.0      PC mit 80386                          N
Filiale Bern    51450.00
Umsatz          511450.0      4 gleiche Ebene            5 untere Ebene
Personal Comp   390150.0      ▸PC mit 8086   23400.00    AAA Preis     4250.000
                              ▸PC mit 80286  42000.00    AAB Menge     35.00000
                              ▸PC mit 80486  176000.0

6 Wert             7 B
    148750.00      J

8 Berechnung
AAA * AAB

9 Bemerkung

> ?                              || Filiale            || 07.05.91  16:27
```

Abbildung 5

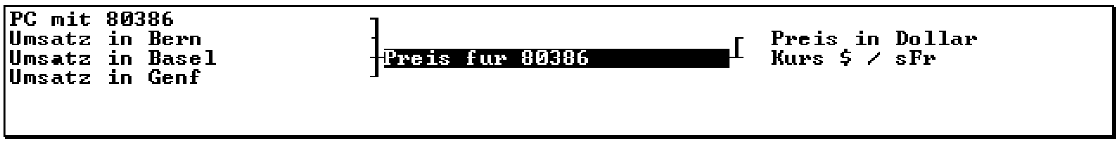
Die _Listensicht_ (Abbildung 6) zeigt die Elemente sortiert nach verschiedenen Kriterien, und es kann nach bestimmten Attributinhalten gesucht werden. Sie dient dem schnellen Zugriff auf ein bestimmtes Element. Zudem zeigt die Listensicht auch einen möglichen Weg vom gewünschten Element zur Wurzel (unteres Teilfenster).

Die _Netzsicht_ (Abbildung 7) zeigt die Vernetzung eines Elements mit den Vorgänger- und Nachfolgerelementen. Es ist eine sehr lokale Sicht, die nur die nächstgelegenen Elemente berücksichtigt.



```
Auswahl eines Elementes mit <RETURN>, Abbruch <ESC>=====<insert>
      Menge                                          8.000000  F
      Umsatz                                    694500.000000  F
      Kosten                                    610000.000000  F
      Einkauf                                   415000.000000  F
      Personal                                  160000.000000  F
      uebrige                                    35000.000000  F
      Personal Computer                         539200.000000  F
      Zubehoer                                  155300.000000  F
      PC mit 8086                                35100.000000  F
      PC mit 80286                               61600.000000  F
      PC mit 80386                              178500.000000  F
      PC mit 80486                              264000.000000  F
      Menge                                         33.000000  F
      Menge                                         42.000000  F
      Menge                                         22.000000  F
      Menge                                         18.000000  F

  Menge --> PC mit 80486 --> Personal Computer --> Umsatz --> Filiale Genf
  --> Gewinn
```

Abbildung 6



```
PC mit 80386
Umsatz in Bern                                  Preis in Dollar
Umsatz in Basel        Preis fur 80386          Kurs $ / sFr
Umsatz in Genf
```

Abbildung 7

2.3.4. Modularisierung

NetCalc unterstützt mit verschiedenen Funktionen die Aufteilung der Modelle in Module. Diese Art der Modellentwicklung erhöht die Produktivität und verbessert die Transparenz der Modelle, da jedes Modul für sich entwickelt, getestet und dokumentiert werden kann. Bei der Aufspaltung des Modells in kleinere Einheiten entstehen oft Teile, die in anderen Anwendungen ebenfalls Verwendung finden können. Mit der Zeit kann somit eine Modulbibliothek erstellt werden.

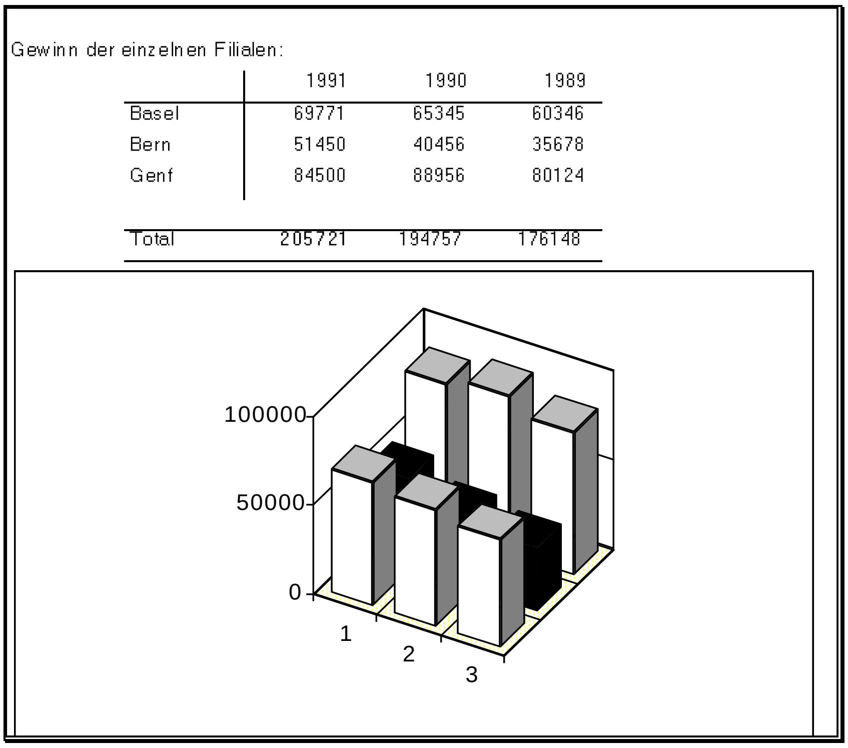
Dank der weitgehenden Standardisierung haben alle Modelle dasselbe Aussehen und können daher leichter modularisiert und von unterschiedlichen Anwendern entwickelt und anschliessend zusammengefügt werden. Das Denken und Arbeiten mit Modulen in NetCalc vermindert somit den Koordinationsaufwand, wenn mehrere Modellierer gleichzeitig auf unterschiedlichen Geräten an verschiedenen Teilen des Modells arbeiten.

2.3.5. Formeln

Sobald ein Element Nachfolger besitzt (durch weitere Grössen erklärt ist), kann eine Formel zur Berechnung des Wertes eingegeben werden Ein Beispiel einer solche Formel ist in der Abbildung 5 im Fenster *'8 Berechnung'* zu sehen. NetCalc bietet eine breite Palette an mathematischen Funktionen (Trigonometrie-, Exponential-, Finanzfunktionen etc.) und Steuerfunktionen (Bedingungen, Testfunktionen, etc.). Eine Sofortauswertung dieser Ausdrücke sorgt dafür, dass das Modell auf jeder Aggregationsstufe immer auf dem neuesten Stand ist. Neue Werte können jederzeit eingegeben und ihre Auswirkungen auf andere Elemente beobachtet werden (Simulationen, Sensitivitätsanalysen etc.). Ebenfalls besteht die Möglichkeit vorübergehend endogene Grössen als exogene zu behandeln, indem ohne Verlust der ursprünglichen Formel der errechneten Wert übersteuert (überschrieben) und zu einem späteren Zeitpunkt wieder zurückgesetzt wird.

2.4. Auswertungssichten

Ist ein Problem ganz oder teilweise formuliert, so kann der Wunsch enstehen, dasselbe Modell aus verschiedenen Blickwinkeln zu betrachten. Reichen die in NetCalc angebotenen Modellansichten nicht aus, bzw. wird ein Modell zu gross, so können mit Hilfe der Tabellenkalkulation weitere Sichten erstellt werden. Hierzu werden die interessierenden Elemente im Modell markiert. Das Programm erstellt anschliessend automatisch eine Datei, die als Schnittstelle zu der gewählten Tabellenkalkulation dient. Mit Hilfe von unterstützenden Makros, welche die Verbindung zwischen Schnittstelle und Tabellenkalkulation sicherstellen, können nun die Sichten bearbeitet werden. Als Beispiele seien die Gestaltung einer Eingabemaske, eine graphische oder eine statistische Auswertung erwähnt (Abbildung 8). In der Auswertungssicht können die quantitativen Beziehungen zwischen den Elementen nicht verändert werden. Alle Änderungen der Werte, die in der Tabellenkalkulation vorgenommen werden, übernimmt NetCalc auf Wunsch automatisch in das ursprüngliche Modell. Umgekehrt werden alle Modelländerungen, die in NetCalc gemacht werden, automatisch in der Schnittstelle und somit auch in der Tabellenkalkulation nachgeführt. Eine Änderungen hat daher nicht zur Folge, dass der Benutzer bestehende Sichten überarbeiten muss. Neue referenzierte Elemente können an beliebiger Stelle in der Auswertungssicht eingefügt werden.


Gewinn der einzelnen Filialen:

	1991	1990	1989
Basel	69771	65345	60346
Bern	51450	40456	35678
Genf	84500	88956	80124
Total	205721	194757	176148

Abbildung 8

2.5. Datenverwaltung in NetCalc

Die gesamte Information eines Modells, d.h. die Elemente mit ihren Attribut- und Strukturinformationen, wird in einer *dBASE III*[1] Datenbank gespeichert. Die Datenbank ermöglicht den Datenaustausch und garantiert eine leistungsfähige Datenschnittstelle zu praktisch jeder anderen Software. Netalc unterstützt auch den Zugriff auf bereits bestehende externe Datentabellen, so dass nicht alle Daten vom Benutzer eingetippt werden müssen. Zusätzlich erlaubt die Datenverwaltung dem geübten *dBASE III* Benutzer die Daten interaktiv zu bearbeiten. Es ermöglicht dem Anwender mehr Attributfelder anzufügen, die Grösse der Felder zu verändern, oder Anfragen an die Datenbank zu stellen, die in *NetCalc* nicht oder nur mit Mühe formuliert werden können. Derartige Arbeiten werden mit *dBASE III* oder einem Programm, das *dBase III* Dateien lesen kann (z.B. Excel oder Framework), problemlos erledigt.

Die Benutzung externer Speicher (Verwaltung aller Daten in einer Datenbank auf dem Plattenspeicher anstatt im RAM) erhöht nicht nur die Transparenz, sondern gleichermassen die Sicherheit vor einem eventuellen Datenverlust durch ein technisches Versagen. Alle Änderungen werden sofort in der Datenbank vermerkt. Gelöschte oder teilweise verlorene Informationen können rekonstruiert werden. Die Datenbank bietet zudem den Vorteil, dass die Modellgrösse von der Speicherkapazität der Platte und nicht von der Grösse des vorhandenen RAM-Speichers abhängt.

2.6. Weitere Funktionalitäten

Alle NetCalc-Funktionen zu beschreiben würde den Rahmen dieses Artikels sprengen[2]. Ein wichtiger Aspekt sei an dieser Stelle jedoch noch vermerkt: das Funktionsangebot kann durch den Benutzer jederzeit erweitert werden. Diese Weiterentwicklung und Anpassung des Systems an die Bedürfnisse des Benutzers werden durch Makros, frei gestaltbare Menüs und durch in *dBase III* Dateien abgelegte Texte zur Oberfläche des Programms ermöglicht.

---

[1] dBase III ist ein Produkt der Ashton Tate GmbH.

[2] Der interessierte Leser kann eine Demoversion und eine ausführliche Beschreibung [5] über die Bedienung und Funktionalität von NetCalc beim Autor selbst oder bei der Gruppe für Ernährungssicherung am Institut für Automation der Universität Fribourg anfordern.

Makros erlauben immer wiederkehrende Abläufe einmal aufzuzeichnen und einer Taste oder einem Menüpunkt zuzuordnen. Sie können aber ebenfalls dazu dienen das Leistungsangebot zu erweitern. Der erfahrene *dBASE III* Anwender hat die Möglichkeit das bestehende Programm durch eigene Programme zu erweitern, indem er eine Funktion in *dBASE III* oder in einer anderen Programmiersprache (wie zum Beispiel C) programmiert und einem Makro zuordnet.

Die im Programm verwendeten Menüs[1] und Texte für die Benutzerschnittstelle können beliebig editiert werden. Daher ist es zum Beispiel möglich, dass verschiedene Sprachen von verschiedenen Benutzern gleichzeitig nebeneinander verwendet werden können. Die Sprachen Französisch, Englisch und Deutsch stehen bereits jetzt dem Benutzer per Option zur Verfügung.

[1] Menüs können beliebig erweitert, verkürzt oder umgestellt werden.

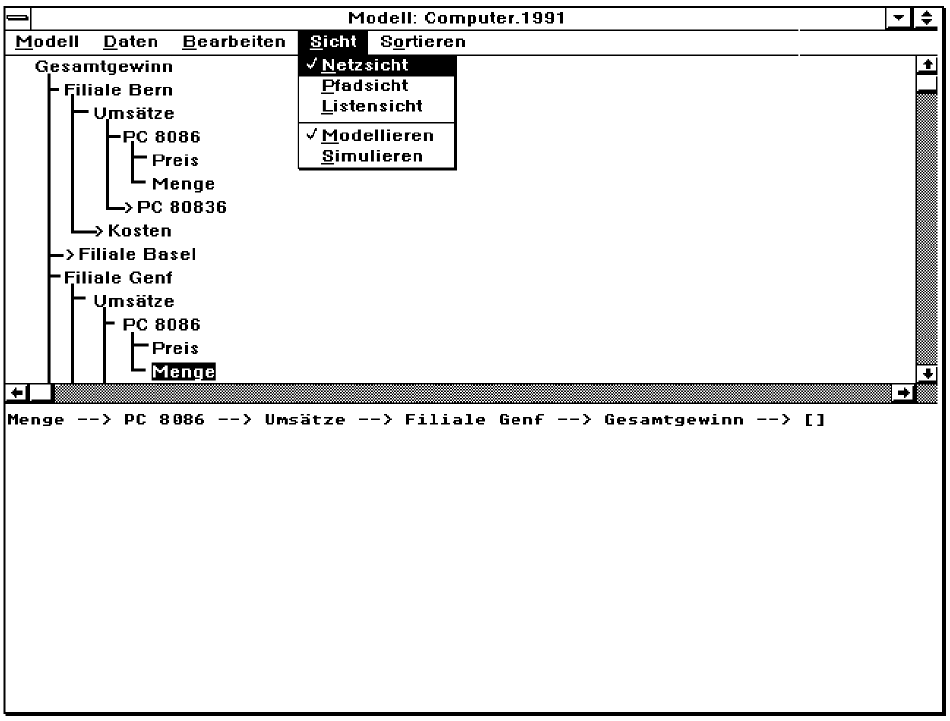
3 Ausichten auf NetCalc für Windows 3.1

Zur Zeit wird NetCalc umgeschrieben. Es werden hierbei verschiedene Verbesserungen mit dem Ziel eines abgerundeten Modelliersystem angestrebt. Diese werden im folgenden kurz erläutert :

- o Implementation unter Windows 3.1,
- o Verbindung von objektorientiertem Ansatz und Datenbank,
- o Trennung von Modellstruktur und Daten,
- o Erweiterung der einfachen Formelauswertung zu einer Programmiersprache.

3.1. Implementation unter Windows 3.1

NetCalc wird mit Hilfe der Programme Smalltalk und Microsoft C[1] unter Windows 3.1 implementiert. Windows gestattet eine verbesserte graphische Benutzeroberfläche und -führung. Dies hat nicht nur ein attraktiveres Programm zur Folge, sondern verkürzt ebenfalls die Einarbeitungszeit, da die Oberfläche standardisiert ist (Abbildung 9).



```
Gesamtgewinn
 ├ Filiale Bern
 │  ├ Umsätze
 │  │  ├ PC 8086
 │  │  │  ├ Preis
 │  │  │  └ Menge
 │  │  └> PC 80836
 │  └> Kosten
 ├> Filiale Basel
 ├ Filiale Genf
 │  ├ Umsätze
 │  │  ├ PC 8086
 │  │  │  ├ Preis
 │  │  │  └ Menge
```

```
Menge --> PC 8086 --> Umsätze --> Filiale Genf --> Gesamtgewinn --> []
```

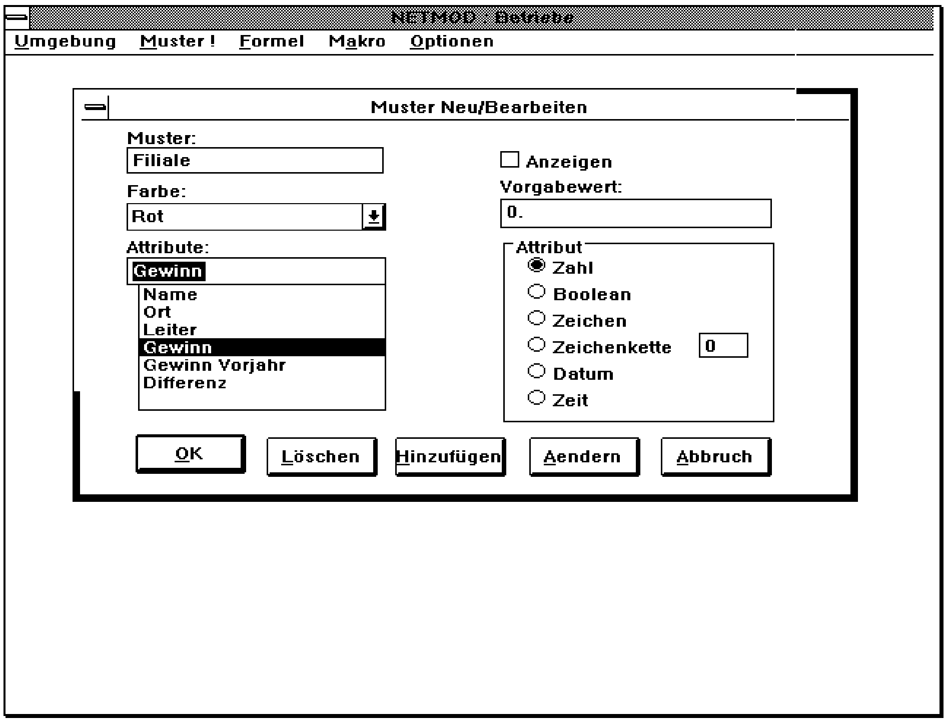
Abbildung 9

[1] Microsoft C ist ein Produkt von Microsoft GmbH.

Des weiteren erhöht sich die Portabilität des Programmes für verschiedene Computer mit unterschiedlichen Eingabe-[1] und Ausgabemedien[2]. Die Verbindung zu anderen Programmen unter Windows 3.1 (zum Beispiel Excel) kann durch konsequente Ausnutzung von DDE[3] (gemeinsamer Datenpool), DLL[4] (gemeinsamer Funktionspool) und OLE[5] (Verknüpfung beliebiger Windowsobjekte und Applikationen) stark verbessert werden. Damit steigt der Gesamtnutzen der einzelnen Programme durch Datenverbund und grössere Funktionsvielfalt.

3.2. Verbindung des objektorientierten Ansatzes mit einer Datenbank

Neu ist die Möglichkeit, dass die Elemente nicht nur in den Attributinhalten sondern auch in ihrer Datenstruktur verschieden sein können.



Muster Neu/Bearbeiten

Muster: Filiale

Farbe: Rot

Attribute: Gewinn
- Name
- Ort
- Leiter
- Gewinn
- Gewinn Vorjahr
- Differenz

☐ Anzeigen

Vorgabewert: 0.

Attribut
- ● Zahl
- ○ Boolean
- ○ Zeichen
- ○ Zeichenkette 0
- ○ Datum
- ○ Zeit

[1] Maus, Tablett, Tastatur etc.

[2] Drucker, Bildschirm.

[3] DDE = Dynamic Date Exchange: Windows Applikationen können beliebige Daten miteinander austauschen.

[4] DLL = Dynamic Link Library: Funktionen aus anderen Programmen können ausgeführt werden.

[5] OLE = Object linking and embedding: Einbettung von Objekten in andere Anwendungen.

Abbildung 10

Dazu kann der Benutzer Elementklassen definieren, welche frei wählbare Attribute und Methoden[1] beinhalten (Abbildung 10). Jedes Element in NetCalc muss als Instanz einer solchen Klasse definiert werden (Elementinstanz). Dies gewährleistet die Verwaltung verschiedener Elementtypen in einem Modell und die Vererbung der Inhalte bei Elementinstanzen einer Klasse (Abbildung 11). Die Klasse kann für die einzelnen Attribute sogenannte 'Defaults' vorsehen. Die Formel für die Berechnung der Werte eines Attributes kann von Instanz zu Instanz verschieden sein (vergl. hierzu den Abschnitt 'Spracherweiterung').

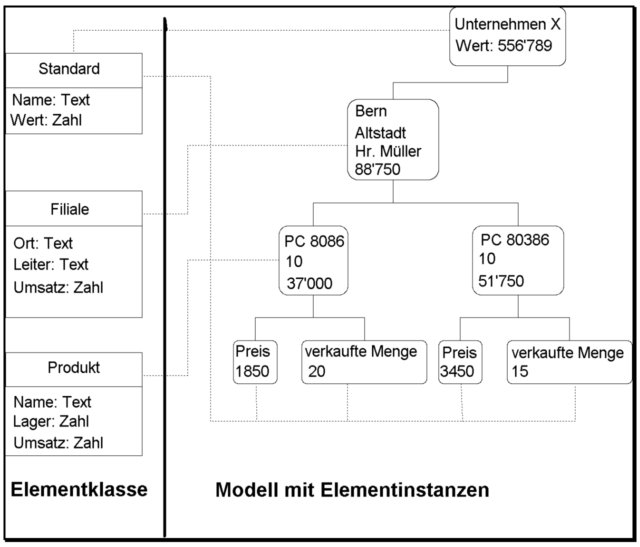


Abbildung 11

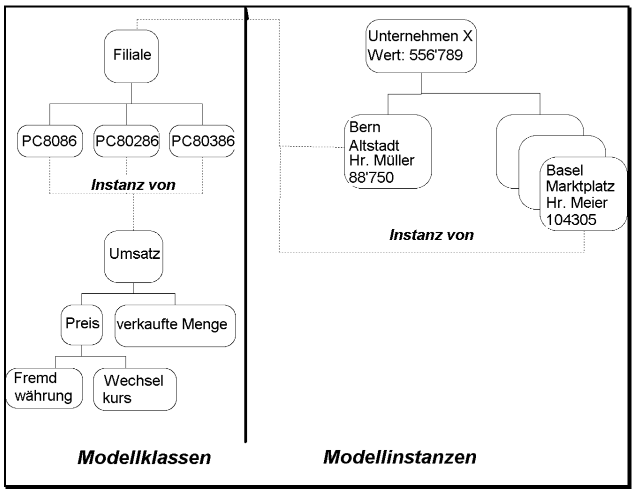


Abbildung 12

1 vergleiche hierzu den Abschnitt 'Spracherweiterung'.

Da die Erfahrung zeigt, dass viele grosse Modelle aus Strukturen bestehen, die sich ähnlich sind oder gar wiederholen, wird der Begriff der Instanzierung erweitert. Die einzelnen Elementinstanzen bilden untereinander ein Beziehungsgeflecht, welches ein ganzes oder ein Teilmodell darstellt. Diese Modelle (Modellteile) können zugleich als Einheit (Modellklasse) betrachtet werden und somit instanzierbar sein. D.h. ein Objekt kann somit ein ganzes Modell beinhalten (die Elemente *'Bern' und 'Basel'* in Abbildung 12 sind solche Instanzen). Eine solche Instanz wird im folgenden Modellinstanz genannt.

Bei einer Modellinstanz kann nur der Inhalt und nicht die Struktur verändert werden. Die Elemente werden hierbei zu Attributen, die selber wieder Element- oder Modelleninstanzen sind. Die Modellinstanzierung in NetCalc ist vergleichbar mit der mathematischen Modellierung mit Indizes, was auf der folgenden Seite an einem sehr einfachen Beispiel und anhand der Abbildung 13 gezeigt wird.

Der Nutzen der Instanzierbarkeit besteht darin, dass sehr grosse Modelle aus einer kleinen und überschaubaren Anzahl von Modellklassen (Grundstrukturen) formuliert werden können. Die Grösse eines Modells entsteht erst durch den Einsatz konkreter Daten ('das Modell wird praktisch durch die Daten aufgeblasen'). Weitere Effekte werden im Abschnitt 'Trennung von Struktur und Daten' beschrieben.

3.3. Beispiel einer Modellinstanzierung

Ein Unternehmen besitzt Filialen in Bern, Basel und Genf. Diese verkaufen Personalcomputer von 8086 - 80386. Es sollen die Gewinne pro Filiale und der Gesamtgewinn berechnet werden.

Zuerst wird eine Filiale modelliert. Der Gewinn einer Filiale lautet:

Gewinn = Umsatz - Kosten

Für ein Produkt gilt: Umsatz = Preis * Menge

Die Filiale besitzt mehrere Produkte (PC 8086, 80286, 80386). Der Umsatz ist folglich die Summe aus den Einzelumsätzen:

$$\text{GUmsatz} = \sum_i \text{Umsatz}_i$$

$$\text{Umsatz}_i = \text{Preis}_i * \text{Menge}_i$$

wobei 'i' in der Abbildung 13 den Produkten (PC 8086, PC 80286, PC 80386) entspricht.

In NetCalc wird diesem Umstand in der Weise Rechnung getragen, dass ein Modell für die Berechnung des Umsatzes (Abbildung 13) gemacht wird und bei den einzelnen Elementen zusätzlich erklärt wird, welche instanzierbar sind und welche nicht.

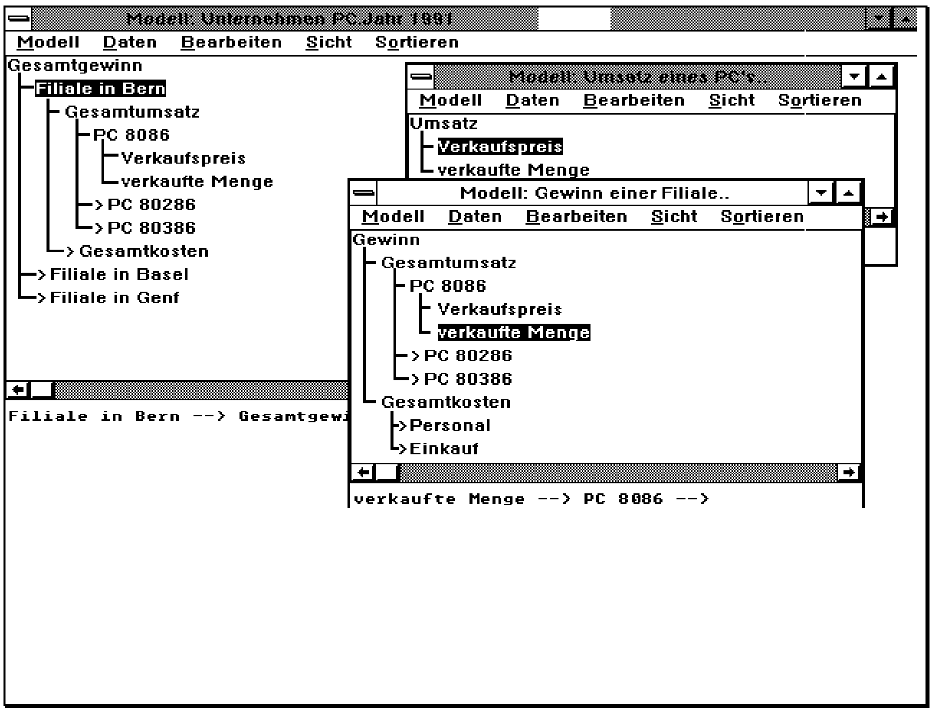
Modelle können auch verschachtelt werden und somit mehrere Indizes haben. Dies wird bei der Berechnung des Gewinns des Unternehmens ersichtlich. Dementsprechend kann die oben modellierte Filiale als Modellklasse betrachtet werden und der Gewinn der Unternehmung ist die Summe aller Einzelgewinne der Filialen.

$$\text{GGewinn} = \sum_k \text{Gewinn}_k$$

$$\text{Gewinn}_k = \sum_i \text{Preis}_i * \text{Menge}_{ki} - \text{Kosten}_k$$

wobei 'k' in Abbildung 13 den Filialen Basel, Bern und Genf und 'i' den Produkten PC 8086, PC 80286 und PC 80386 entspricht.

Wichtig: ändert sich die Struktur des Modells, so ändern sich alle Instanzen dieses Modells mit. Desselben variiert der Verkaufspreis eines Produkts, so variieren alle Produktpreise in den Instanzen. Die Verkaufsmengen hingegen können in allen Instanzen unterschiedlich sein.


Modell: Unternehmen PC Jahr 1991
Modell Daten Bearbeiten Sicht Sortieren
Modell: Umsatz eines PC's..
Modell Daten Bearbeiten Sicht Sortieren
Modell: Gewinn einer Filiale..
Modell Daten Bearbeiten Sicht Sortieren

```
Gesamtgewinn
├Filiale in Bern
│ ├ Gesamtumsatz
│ │ ├PC 8086
│ │ │ ├Verkaufspreis
│ │ │ └verkaufte Menge
│ │ ├>PC 80286
│ │ └>PC 80386
│ └> Gesamtkosten
├>Filiale in Basel
└>Filiale in Genf

Filiale in Bern --> Gesamtgewi
```

```
Umsatz
├ Verkaufspreis
└ verkaufte Menge
```

```
Gewinn
├ Gesamtumsatz
│ ├PC 8086
│ │ ├ Verkaufspreis
│ │ └ verkaufte Menge
│ ├>PC 80286
│ └>PC 80386
└ Gesamtkosten
  ├>Personal
  └>Einkauf

verkaufte Menge --> PC 8086 -->
```

Abbildung 13

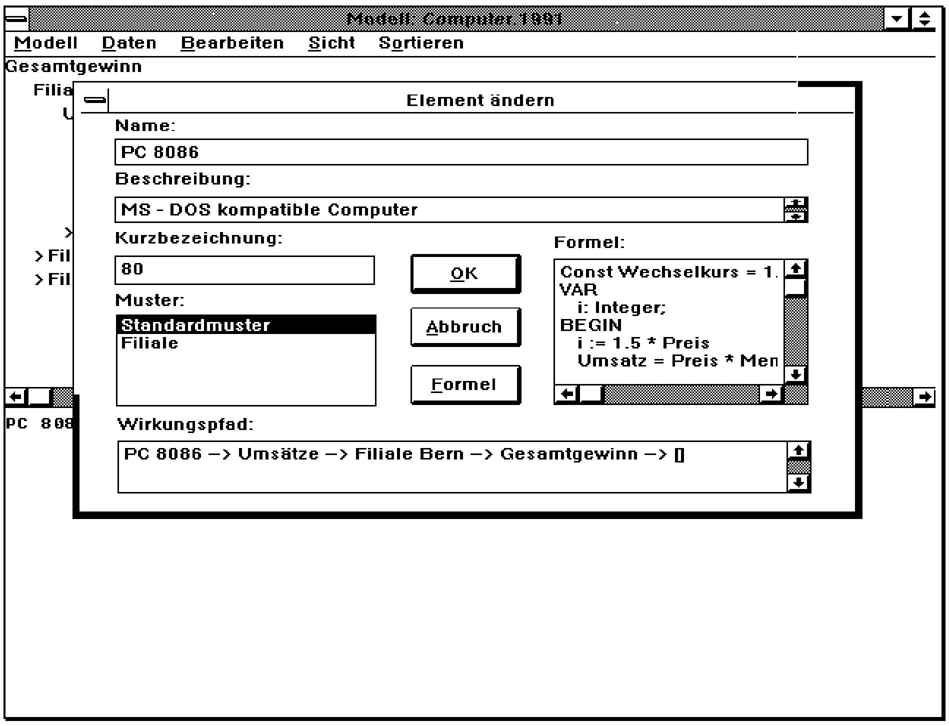
3.4. Trennung von Struktur und Daten

Bis anhin wurden die Daten eines Elements beim Element selbst gespeichert. Vergleiche von Modellaussagen zu verschiedenen Datensätzen waren dadurch nicht einfach zu erstellen. Nun werden die Daten getrennt vom Modell abgespeichert. Zu jeder Modellstruktur kann es neu mehrere Datensätze geben. Dies ermöglicht nun die Durchführung von Simulationen und das Abspeichern aller Ergebnisse der verschiedenen Simulationen, um sie zu einem späteren Zeitpunkt zu verwenden oder miteinander zu vergleichen.

Die Vorteile dieser Neuerungen zeigen sich in einem Netzwerk, wobei das Modell (das Modellskelett) auf dem Server liegt und die Benutzer ihre Modelldaten auf dem eigenen Rechner oder im eigenen Benutzerverzeichnis haben. Änderungen der Modellstruktur auf dem Server sind sogleich für alle Benutzer ohne zusätzlichen Aufwand verfügbar.

3.5. Spracherweiterung

Durch den Einsatz von Modell- und Elementklassen mit beliebigen Attributen und Attributtypen (String, Boolean, Zahlen, Datum etc.), ist nun die Formel/Formelsprache, welche bis jetzt nur die Berechnung eines Attributes (mit dem Namen 'Wert') zuliess, erweitert worden.



Name: PC 8086

Beschreibung: MS - DOS kompatible Computer

Kurzbezeichnung: 80

Muster: Standardmuster, Filiale

Formel:

```
Const Wechselkurs = 1.
VAR
  i: Integer;
BEGIN
  i := 1.5 * Preis
  Umsatz = Preis * Men
```

Wirkungspfad: PC 8086 –> Umsätze –> Filiale Bern –> Gesamtgewinn –> []

Abbildung 14

Neu kann in einer Berechnung jedes Attribut verwendet und mit anderen verknüpft werden. Diese Definition wird in einem eigenen Feld 'Formel' (Abbildung 14) abgelegt, und wie ein Programm (=Skript) formuliert. Es können Funktionen, Schlaufen, Variablen etc. in den Skripten Verwendung finden. Der Aufbau eines solchen Skripts entspricht der Sprache Pascal (Abbildung 14).

Im Einsatz der ersten Versionen von NetCalc wurde zudem festgestellt, dass der Funktionsumfang oft nicht ausreicht[1] und gewisse Berechnungen nicht mit einem einzigen Berechnungsschritt[2] erledigt werden können. Dies führt dazu, dass der Benutzer jederzeit selber neue Funktionen aus einem Grundset von Operationen und Funktionen definieren und auf Wunsch allen übrigen Modellen und Benutzern zur Verfügung stellen kann (Funktionspool).

[1] Die Benutzer wollten vor allem vermehrt datenbankspezifische Manipulationen ausführen können.

[2] Gewisse Berechnungen brauchen Schlaufen, temporäre Variablen, Rekursionen und anderes mehr.

4 Bibliographie

1: S. Bodily, 1986. *'Spreadsheet Modeling as Stepping Stone'*, Interfaces, 16.5 (Sept.-Oct.), S. 34-52.

2: A.M. Geoffrion, 1987. *'An Introduction to Structured Modeling'*, Management Science, Band 33, Nr. 5, S. 547 - 588.

3: A.M. Geoffrion, 1989. *'The Formal Aspects of Structured Modeling'*, Operations Research, 37, 1, S. 30 - 50.

4 H.J. Greenberg, F.H. Murphy, 1991. *'Views of Mathematical Programing Models and Their Instances'*, Working Paper, University of Colorado at Denver and Temple University.

5: H. Häuschen, 1988. *'NetCalc: Modellier-, Kalkulations- und Auswertungssystem für hierarchisch vernetzte Modelle'*, Diplomarbeit, Wirtschaftswissenschaftliche Fakultät Freiburg, Schweiz.

6: C. Jones, 1990. *'An Introduction to Graph-Based Modeling Systems'*, ORSA Journal on Computing, Band 2, Nr. 2.

7 ACMS: An Arithmetic Constraint Modeling System

Catherine Donnet-Portenier, Louis Cardona, Jürg Kohlas

Abstract

The traditional way to solve arithmetic problems consists in using imperative computer languages like C, Pascal or the well-known spreadsheets. On the other hand, the constraint programming approach uses a model as a knowledge base which can be queried in many different ways, thus avoiding some drawbacks of the imperative programming approach, like redundancy, mixing of the declarative and procedural knowledge, lack of security, consistency and integrity.

In this paper the constraint programming language ACMS (Arithmetic Constraint Modeling System) will be described. An economic example taken from a real application will then be computed as an illustration by means of ACMS.

While this language is well suited for the management of pure arithmetic models, it would have to be extended in order to be able to handle more general models.

1 Preface

1.1 Arithmetic Models

Most commercial decisions are based on quantitative criteria, but with problems of any complexity it becomes difficult to see how the different factors interact with each other. As a consequence, quantitative modeling techniques are necessary in order to understand and analyse real commercial systems.

A *quantitative model* is a set of relations (constraints) between quantitative units, which may change (variables) or not (constants).

As a subset of quantitative models, *arithmetic models* are composed of a set of relations expressable as a set of *arithmetic equations*, i.e. equations whose terms are either constants, variables or *arithmetic expressions*. In the last case the variables as well as the constants are connected by arithmetic operators, i.e. addition, subtraction, multiplication and division (cf. diagram 1.1.1).

```
arithmetic-model ::= {equation}*
equation ::= expression "=" expression
expression ::= constant | variable | arithmetic-expression
constant ::= real | integer
variable ::= identifier
arithmetic-expression ::= expression { a-operator expression }*
a-operator ::= "+" | "-" | "*" | "/"
identifier ::= character {character | digit }*
```

Diagram 1.1.1: arithmetic models (EBNF-syntax)

Arithmetic models are widely used to express the most common laws and definitions of economics, and can thus be considered as an important tool for decision support tasks. The following diagram gives an example of a very simple arithmetic model.

```
costs = variable_costs + fix_costs
profit = revenue - costs
revenue = unit_price * sales
```

Diagram 1.1.2: a very simple arithmetic model

1.2 Imperative versus Constraint Programming: the language ACMS

The traditional way to solve arithmetic problems consists in using imperative computer languages, such as C, Pascal or the well-known spreadsheets. There the programmer has to anticipate the different questions to be answered in order to encode the expressions correctly, because, in such languages, the following expressions are considered to be different and serve to solve different problems:

$$\text{revenue} := \text{unit\_price} * \text{sales} \quad (1.1)$$

$$\text{unit\_price} := \text{revenue} / \text{sales} \quad (1.2)$$

$$\text{sales} := \text{revenue} / \text{unit\_price} \quad (1.3)$$

This approach is first of all quite unnatural, because these equations are both from a human and a mathematical point of view, essentially equivalent. Moreover, the imperative programming approach is in some respect very inefficient, as the number of equations will grow exponentially when new variables and relations are introduced into the model.

On the other hand, the constraint programming approach considers a model as a description or definition of relations between variables. This means that the model can be used as a knowledge base, independent of the questions one might ask. Thus programming becomes a declarative task: the programmer states a set of *relations* (called constraints) between a set of *objects* (variables and/or constants), and it is the job of the constraint-satisfaction system to find a solution that satisfies these relations [LELER 1988].

As an example of the constraint programming approach, a new constraint programming language called ACMS (Arithmetic Constraint Modelling System) will be described in this paper. This language is constructed upon the constraint-propagation system originally designed at the MIT by Abelson and Sussman [ABELSON, SUSSMAN 1985].

In this system the constraints are implemented by procedural objects with local state. This makes ACMS similar to object-oriented languages, the main difference being that in ACMS the computation is *bidirectional* in nature (cf. [NAKAZAWA, ISODA, MIYAZAKI, AISO 1986]).

ACMS provides *logical data independence* between applications and the conceptual knowledge by encapsulating application-oriented behaviour into *abstract operation types* (e.g. conditional expressions) which are themselves objects. Thus, both declarative and procedural aspects of the system can be handled uniformly (cf. [SHEPHERD, KERSCHBERG 1986]).

A knowledge base built by means of ACMS consists of a set of constraint equations (cf. [MORGENSTERN 1986]).

For instance, the equations (1.1), (1.2) and (1.3) will be expressed declaratively as a single constraint equation, namely:

$$\text{revenue} = \text{unit\_price} * \text{sales} \quad (1.4)$$

This equation defines only a relation between the three variables *revenue*, *unit_price*, and *sales*. Whenever two variables are known, the third will be automatically computed. A simple, but complete knowledge base could be given by the following set of equations:

$$\text{costs} = \text{variable\_costs} + \text{fix\_costs} \tag{1.5}$$

$$\text{profit} = \text{revenue} - \text{costs} \tag{1.6}$$

$$\text{revenue} = \text{unit\_price} * \text{sales} \tag{1.7}$$

Among other consequences, the reader will already have noticed that in this approach there is no need for any kind of assignment. In fact, the equality A = B stands for both A := B and B:= A simultaneously. As a consequence, each constraint needs to be written only once, in one of its equivalent forms.

In the light of these considerations, it becomes clear that for example traditional decision-making models, which needed to distinguish rigidly between decision variables, intermediate variables and output variables, may now be expressed in a more concise and flexible way. Moreover, this way of programming avoids some drawbacks of the imperative programming approach, like redundancy, mixing of the declarative and procedural knowledge, lack of security, consistency and integrity (cf. [LAFUE, SMITH 1986]).

ACMS is currently implemented in two different environments: ACMS-Explorer [PORTENIER 1990] runs under CLOS [STEELE 1990] on an Explorer machine, whereas MacNet [CARDONA, KOHLAS, SEELIGER 1992] runs under Scheme [LIGHTSHIP 1989] on a Macintosh computer.

In the next chapter, the ACMS language will be specified. An economic model taken from a real application will then be formulated and discussed in chapter 3.

2 Language Specification

The purpose of this chapter is to give a brief description of the language ACMS. In the sections 2.1, 2.2, and 2.3 the basic version of the language running on the Explorer and Macintosh environments will be presented. Some extensions to this basic version are discussed in section 2.4. The section 2.5 gives an overview of the user-interface for arithmetic models implemented in MacNet.

2.1 Relational Structures and Modules

The whole power of expression of the ACMS language can be described in terms of relational structures and modules (cf. diagram 2.1.1).

```
module ::= "(DEFMODULE" m-name " '( " {c-name}* ")"
                                          {generalized-constraint}* ")"
generalized-constraint ::= "(DEFCONSTRAINT"
                                    c-name "(" {formal-parameter}* ")"
                                                  {g-equation}* ")"
g-equation ::= equation | generalized-constraint
equation ::= "(" expression "=" expression ")"
m-name ::= quoted-identifier
quoted-identifier ::= " ' " identifier
c-name ::= identifier
formal-parameter ::= identifier
```

Diagram 2.1.1: relational structures and modules (EBNF-syntax)

A *relational structure* is a way to express a constraint between some objects (variables and/or numerical constants in our case). In ACMS there are two different relational structures: numerical equations and generalized constraints.

An equation *explicitly* states an equality relation between two arithmetic expressions.

Generalized constraints go a step further establishing a relationship between units in terms of a system of equations, which may itself be composed of previously defined generalized constraints. By means of generalized constraints one has therefore the possibility of introducing powerful abstraction mechanisms. Generalized constraints in fact can be considered as procedures defining equations *implicitly*, i.e. as the result of the encapsulation of a subset of equations. For this reason a generalized constraint may be seen as a kind of user-defined constraint linking two or more variables.

```
(defconstraint elasticity
               (elast price1 price2 quantity1 quantity2)
                (elast = (mqr / mpr))
                (mpr = (dp / (sp / 2)))
                (mqr = (dq / (sq / 2)))
                (dp = (price1 - price2))
               (dq = (quantity1 - quantity1))
                (sp = (price1 + price2))
                (sq = (quantity1 + quantity2)))

(make-model '((p0 = (p01 + p02))
                (q0 = (q01 + q02))
                (elasticity e0 p0 pp1 q0 q1)
                (elasticity e1 pp1 p2 q1 q2)))
```

Diagram 2.1.2: definition and use (gc-call) of a generalized constraint

This kind of constraint is defined in the constraint section of an ACMS program (cf. section 2.2 below) and used in the model section by means of the so-called *generalized constraint calls* (also gc-call). Their behaviour is similar as for ordinary procedures in functional languages like LISP (cf. diagram 2.1.2).

Modules finally improve the flexibility of the language in a modular way: submodels, expressed as a set of generalized constraints, may be defined as modules, compiled separately, and used in different models. Their interface is given by the list of names of the generalized constraints that they encapsulate. Diagram 2.1.3 gives an example of the definition of a module.

```
(defmodule 'testmodule '(test2 test3)  --------> interface
      (defconstraint test1 (a b c)     --------> not exported
          (a = (b + d))
          (d = (2 * c)))
      (defconstraint test2 (a b c)    --------> exported
          (test1 a b d)
          (d = (2 * c)))
      (defconstraint test3 (a b f g)  --------> exported
          (test2 a b c)
          (c = (3 * b))
          (test1 a g f)))
```

Diagram 2.1.3: the definition of a module

Because modules are saved in their own files, they have to be loaded separately before they can be used as part of a model, according to the following syntax: "(load " <module-name> ".xld)".

2.2 Specification of Models: the ACMS Program

An arithmetic model can be fully specified by means of an ACMS *program*, which can be broken down into three main sections: the *module* section, the *constraint* section, and the *model* section (cf. diagram 2.2.1).

```
ACMS-program ::= [module-section] [constraint-section] model-section
module-section ::= {load-module}*
load-module ::=  "(LOAD" module-name ".xld)"
constraint-section ::= {generalized-constraint}*
model-section  ::= "(MAKE-MODEL" quoted-model ")"
quoted-model ::=  " '(" model  ")"
model ::=  {equation}*
equation ::= "(" expression "=" expression ")" | gc-call
gc-call ::=  "("  c-name {actual-parameter}* ")"
module-name ::= identifier
c-name ::= identifier
actual-parameter ::= identifier
```

Diagram 2.2.1: the specification of a model (EBNF-syntax)

In the module section all the modules needed to run the program are loaded in turn. They must have been defined and compiled previously.

In the constraint section all the other generalized constraints needed and still not contained in the loaded modules are defined.

In the model section finally, a model is described in terms of relational structures. The specification of a model in ACMS is done by sending the message **make-model** to the system and passing the described model as parameter.

An example of an ACMS-program is given in the following diagram.

```
(load module1.xld)
(load module2.xld)
(load module3.xld)

(defconstraint test1 (a b c)
          (a = (b + d))
          (d = (2 * c)))
(defconstraint test2 (a b c)
          (test1 a b d)
          (d = (2 * c)))

(make-model
      ' ((test2 h i j)
         (k = l - (test1 m d a))
         (h + i = k + l)))
```

Diagram 2.2.2: a program written in ACMS

2.3 Using Models

The model's manipulation consists mainly in performing the following operations one or several times: setting values (**set-value!**), forgetting values (**forget-value!**) and getting values (**get-value!**) (cf. diagram 2.3.1).

```
set-value ::= "(SET-VALUE!" variable value quoted-source ")"
forget-value ::= "(FORGET-VALUE!" variable quoted-source ")"
get-value ::= "(GET-VALUE!" variable ")"
quoted-souce ::= "'" identifier
value ::= real | integer | interval
interval ::= "(" lower-bound upper-bound ")"
lower-bound ::= real | integer
upper-bound ::= real | integer
```

Diagram 2.3.1: manipulation commands (EBNF-syntax)

SET-VALUE!: this function is used whenever the user wants to assign a value to a variable. This kind of assignment has as a side-effect the propagation of the value through the whole net. If a contradiction is found an error message is returned and the assignment removed.

FORGET-VALUE!: this function is used whenever the user needs to reset a variable. The message forget-value! has as a side-effect the propagation of delete commands which may in turn delete the values of other variables.

GET-VALUE!: this function is used whenever the user wants to know the current status of a variable.

Diagram 2.3.2. shows an example of a model's manipulation. Note that it is possible to set intervals of values for a variable such that the results are also intervals.

```
(make-model '( (costs = (variable_costs + fix_costs)
                        (profit = revenue - costs)
                        (revenue = unit_price * sales)))
(set-value! fix_costs 3000 'user)
(set-value! variable_costs (2500 2700) 'user)
(get-value! costs)   ; the answer would be --> (5500 5700)
(forget-value! variable_costs 'user)
```

Diagram 2.3.2: an example of model's manipulation

2.4 Extending ACMS

The elements of the language previously defined are actually not powerful enough to solve all the problems one would like to express by means of arithmetic models. In particular there are two extensions that would be very welcome: the *conditional expressions* and the *functional constraints* (cf. diagram 2.4.1).

```
conditional- expression ::= "@IF" condition "THEN" expression
                                                  "ELSE" expression
condition ::= expression ( ">" | "<" | "=" ) expression
functional-constraint ::= procedural-constraint |  operational-constraint
procedural-constraint ::= mean-integral
mean-integral ::= "@KUMD" ( variable | constant )
                              "(" group1 [group2 ... group5] expression ")"
operational-constraint ::= function "(" variable | constant ")"
function ::= sin | cos | tan | arcsin | arccos | arctan | square | sqrt
groupi ::= expression limit
limit ::= expression
```

Diagram 2.4.1: extensions to the basic ACMS

Conditional expressions allow one to describe alternative expressions. The selected expression for execution depends on the value of the given condition. An example of a conditional expression might be:

@IF demand < supply THEN (demand - supply) ELSE 0

Functional constraints on the other hand allow one to define functional relations between variables other than the simple arithmetic functions. The main problem here is to define functions in terms of bidirectional constraints (for functions of one variable). This means that the inverse of the function, questions of domain and selection of values (if the inverse is not unique) must be addressed. An example of such functional constraints is given by the *mean-integral* .

These two extensions are now available in the ACMS-version running on the Explorer machine. The MacNet version contains only a subset of them (cf. diagram 2.5.1 below).

2.5 The MacNet Interface

The construction and use of arithmetic models can be very difficult for non-specialists. It would be interesting to have an interface allowing the introduction of models in a more natural way. The key idea consists of allowing the user to introduce models according to the classical mathematical notation.

The MacNet interface provides a first solution, well suited to solve small arithmetic models in which neither modules nor generalized constraints need to be used. The syntax is depicted in diagram 2.5.1. Models are defined and edited according to this syntax in an editor window. The model therefore becomes a normal text document that is passed through the compiler, which is responsible for the efficient construction of the model. Compiled models are then ready for computation.

```
model ::= {equation}*
equation ::= expression "=" expression
expression ::= [+ | -] term { (+ | -) term}
term ::= exp-factor {(* | /) exp-factor }
exp-factor ::= factor [^factor]
factor ::= variable | constant | functional-constraint | "(" expression ")"
functional-constraint ::= function "(" variable | constant ")"
function ::= sin | cos | tan | arcsin | arccos | arctan | square | sqrt
```

Diagram 2.5.1: the interface for small arithmetic models (EBNF-syntax)

The commands for the manipulation of such models are put together in a menu (*command*) (cf. diagram 2.5.2).

This interface has been implemented using an improved version of the Lightship Scheme's Toolsmith [CARDONA 1991], well suited to build up high-level interface objects like menus as well as editors (text windows). An example of the use of this interface is given in the next chapter.

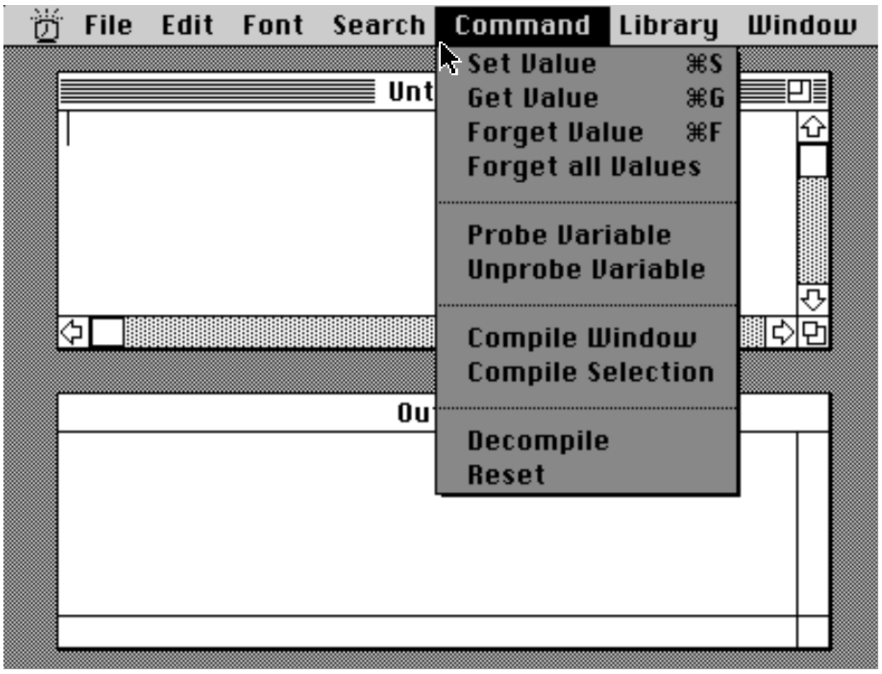


Diagram 2.5.2: the MacNet user-interface

2.6 Limitations of ACMS

According to [LELER 1988], the main limitations of existing constraint languages can be summarized as follows:

1) General problem-solving techniques are weak, so constraint-satisfaction systems must use application-dependent techniques. The few constraint languages that can be adapted to new applications are adaptable only by dropping down into their implementation language.

2) The data types operated on by typical constraint languages are fixed. There is no way to build up new data types (such as using *records* or *arrays* as in conventional languages).

3) Some constraint languages allow the definition of new data types, but new constraints that utilize these new data types cannot be added. New constraints correspond to procedures in conventional languages.

4) Many existing constraint languages do not allow any computations to be expressed beyond what can be expressed by a conjunction of primitive constraints.

5) Even in constraint languages that do allow computation, few are computationally complete. This is a consequence of the difficulty of adding control structures (such as conditionals, iteration, or recursion) to a non procedural language, such as a constraint language, without adding any procedural semantics.

6) Even if we do not require computational completeness, if our language does not have conditionals, then constraints that depend on other constraints cannot be expressed.

7) Many constraint-satisfaction systems use iterative numeric techniques such as *relaxation*. These techniques can have numeric-stability problems; a system using these techniques might fail to terminate even when the constraints have a solution, or might find one solution arbitrarily for constraints with more than one solution.

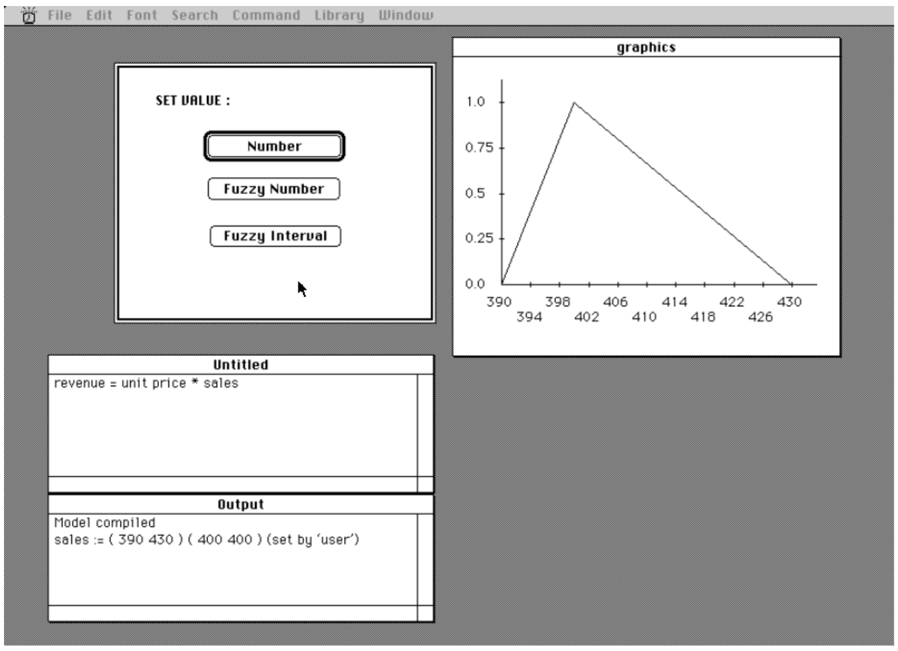
ACMS avoids some of these limitations by means of some powerful language constructs defined above, like generalized constraints, modules and conditional expressions (which are implemented as a **higher-order constraint**).

The last problem does not arise in ACMS, since these kinds of iterative numeric techniques are not present. In fact, for problem-solving ACMS only uses local propagation, which is an intuitive, well known method (cf. [ABELSON, SUSSMAN 1985]) and fast. This allows one to separate in ACMS the knowledge representation from the control mechanisms, which makes it easy to add and to eliminate constraints in given models. Furthermore it allows one to extend the language by new defined generalized constraints. However, local propagation is limited because constraint nets containing cycles, such as those arising from simultaneous equations, cannot be solved at all, unless the user supplies a redundant view.

A first way to overcome this drawback consists in the introduction of certain kinds of symbolic computations, e.g. the so-called **term-rewriting**, where rewrite-rules are used to transform the nets arising from the model's compilation into other, simpler nets.

However, term-rewriting can only solve a few trivial cases, because it is still limited to looking at locally connected subgraphs. Therefore some additional features, like the so-called **augmented term-rewriting**, should be introduced into ACMS (cf. [LELER 1988]).

Another class of limitations of ACMS is given by the inability to manage new types of numerical data (e.g. fuzzy numbers, cf. [ZADEH 1975] and [DUBOIS, PRADE 1988], arrays, probability distributions, etc.). In fact, in some realistic cases one needs to modelize expressions like "about 5" or "near to x". To do this, an extension of MacNet, called MacNet$^+$ [CARDONA, GAUCH, KOHLAS, ZAHNO 1992] has been developed. In this system one is allowed to insert fuzzy numbers as well as fuzzy intervals into the given model (cf. diagram 2.6.1).



```
revenue = unit price * sales
```

```
Model compiled
sales := ( 390 430 ) ( 400 400 ) (set by 'user')
```

Diagram2.6.1: the interface of Mac Net^{+}

3 Example: The ROI Economic Model

3.1 The Current Rate of Return on Investment - ROI

Most companies recognize the importance of relating current absolute profits (or cash flow) to the relevant investment base to determine how efficiently they are investing their funds. This is important for the company as a whole, as well as for various operating divisions within the company. Obviously, top management cannot learn much from information on the absolute profit in the different divisions because each division has a different amount of resources committed by the firm to its operation. The logical basis is to consider the *return on investment* (popularly called ROI) for each division. The model dramatizes the need for the firm to interrelate its capital, margin, and financial plans effectively (cf. [KOTLER 1971]).

Diagram 3.1.1 shows an extended ROI model (DuPont schema) , slightly modified from [IBM 1979].

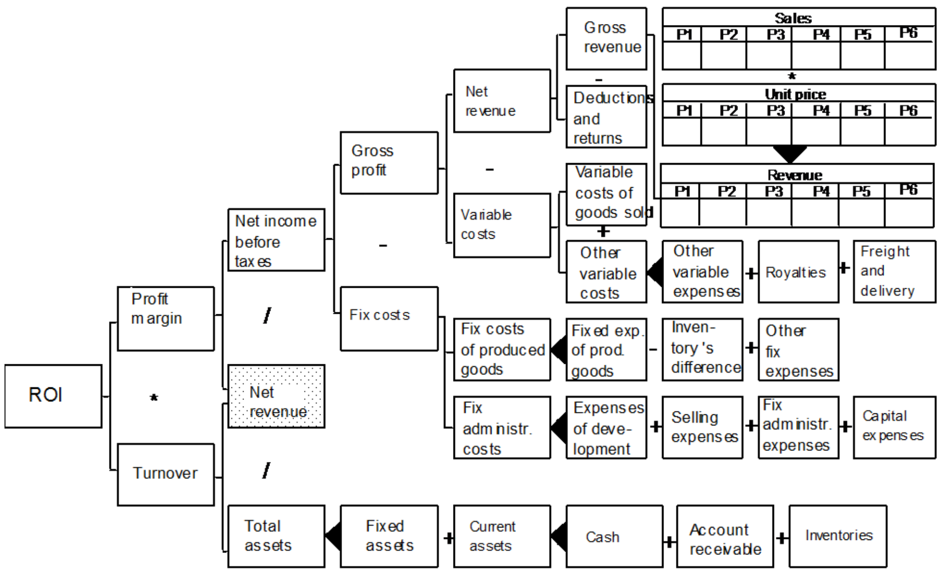


Diagram 3.1.1: the ROI model

The ROI model as depicted above can be considered a typical example of an arithmetic model.

Consider now an imaginary example: division A from the company XYZ produces and sells 6 different types of goods in East Europe. The expected figures for 1992 are the following:

unit prices (already fixed)

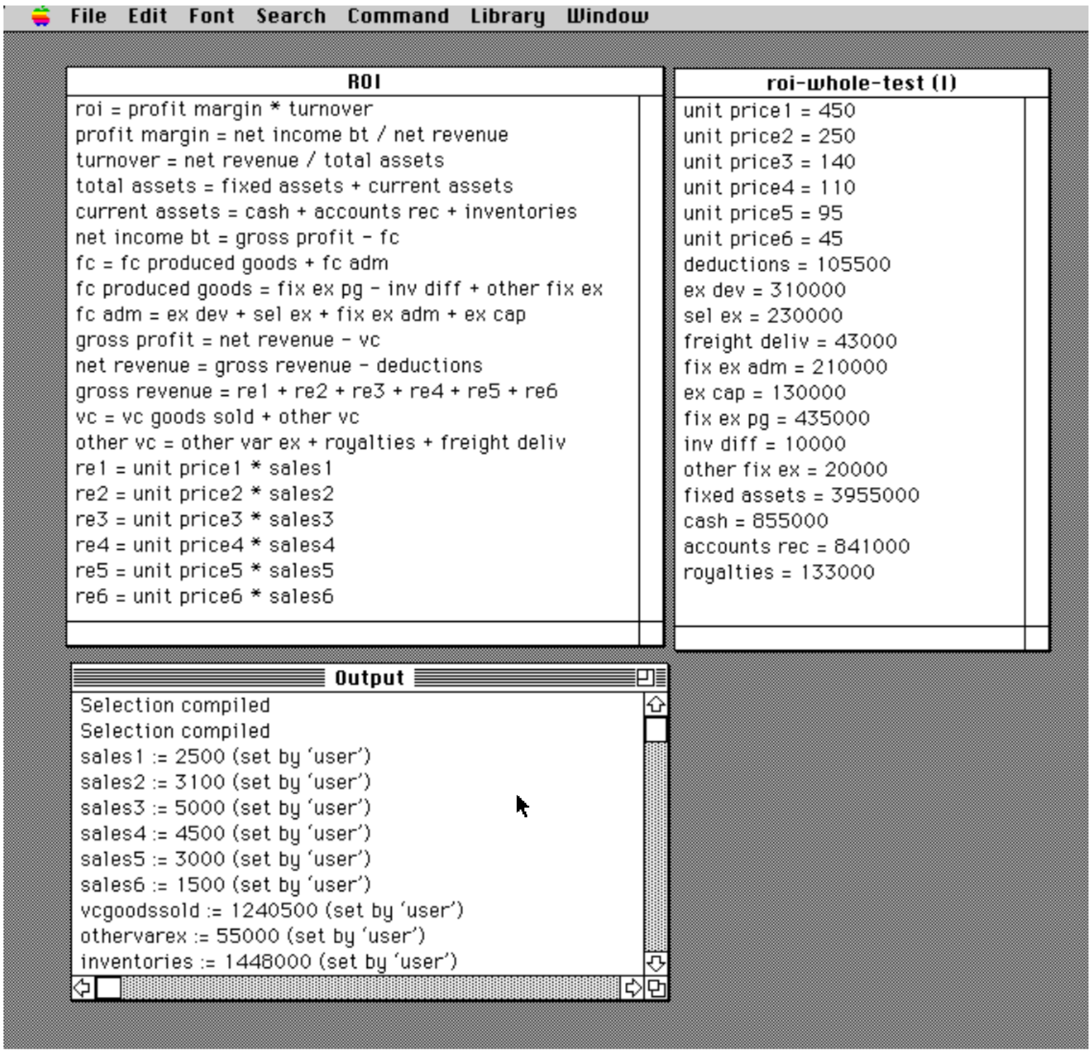
good type 1	450.-
good type 2	250.-
good type 3	140.-
good type 4	110.-
good type 5	95.-
good type 6	45.-

expected goods sold (in units)

good type 1	2,500
good type 2	3,100
good type 3	5,000
good type 4	4,500
good type 5	3,000
good type 6	1,500

other expenses and costs (budgeted)

deductions and returns	105,000.-
variable-costs goods sold	1,240,500.-
other variable expenses	550,000.-
expenses of development	310,000.-
selling expenses	230,000.-
freight and delivery	43,000.-
fix administrative expenses	210,000.-
expenses of the capital	130,000.-
fix expenses produced goods	435,000.-
inventories difference	10,000.-
other fix expenses	20,000.-
royalties	133,000.-



```
ROI
roi = profit margin * turnover
profit margin = net income bt / net revenue
turnover = net revenue / total assets
total assets = fixed assets + current assets
current assets = cash + accounts rec + inventories
net income bt = gross profit - fc
fc = fc produced goods + fc adm
fc produced goods = fix ex pg - inv diff + other fix ex
fc adm = ex dev + sel ex + fix ex adm + ex cap
gross profit = net revenue - vc
net revenue = gross revenue - deductions
gross revenue = re1 + re2 + re3 + re4 + re5 + re6
vc = vc goods sold + other vc
other vc = other var ex + royalties + freight deliv
re1 = unit price1 * sales1
re2 = unit price2 * sales2
re3 = unit price3 * sales3
re4 = unit price4 * sales4
re5 = unit price5 * sales5
re6 = unit price6 * sales6
```

```
roi-whole-test (I)
unit price1 = 450
unit price2 = 250
unit price3 = 140
unit price4 = 110
unit price5 = 95
unit price6 = 45
deductions = 105500
ex dev = 310000
sel ex = 230000
freight deliv = 43000
fix ex adm = 210000
ex cap = 130000
fix ex pg = 435000
inv diff = 10000
other fix ex = 20000
fixed assets = 3955000
cash = 855000
accounts rec = 841000
royalties = 133000
```

```
Output
Selection compiled
Selection compiled
sales1 := 2500 (set by 'user')
sales2 := 3100 (set by 'user')
sales3 := 5000 (set by 'user')
sales4 := 4500 (set by 'user')
sales5 := 3000 (set by 'user')
sales6 := 1500 (set by 'user')
vcgoodssold := 1240500 (set by 'user')
othervarex := 55000 (set by 'user')
inventories := 1448000 (set by 'user')
```

Diagram 3.1.2: ROI as arithmetic model in MacNet

other estimated figures (from the balance)

fixed assets	3,955,000.-
cash	855,000.-
accounts receivable	841,000.-
inventories	1,448,000.-

All the steps needing to be done, in order to compute some values, e.g. the ROI value for the division A of the company XYZ, are depicted in diagram 3.1.2.

The first task consists in the construction of the model (cf. window *ROI*). With the help of ACMS this can be done in a straightforward manner, since the order of the different equations can be arbitrary. The automatically constructed constraint-net after compilation is depicted in diagram 3.1.3. The compiler introduces auxiliary variables (represented by small circles on diagram 3.1.3.) when needed to link different parts of the model.

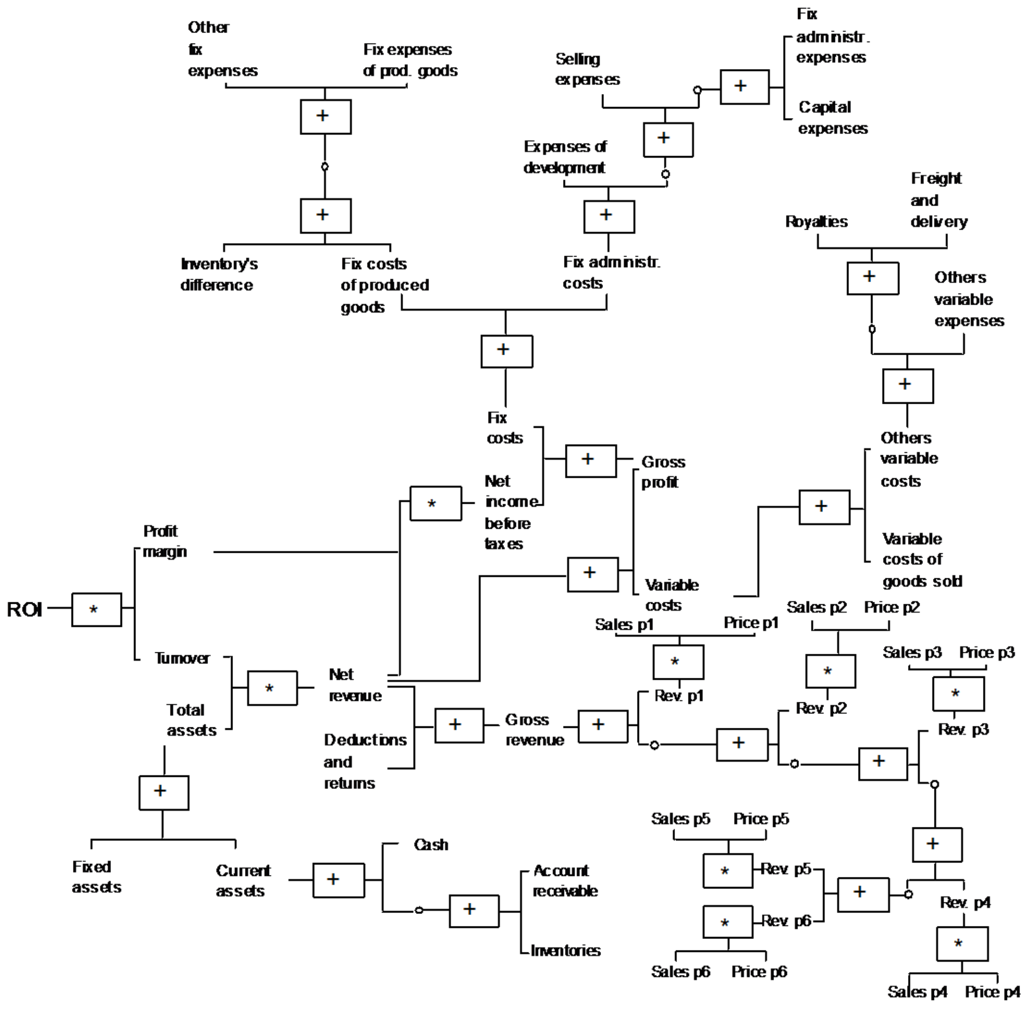
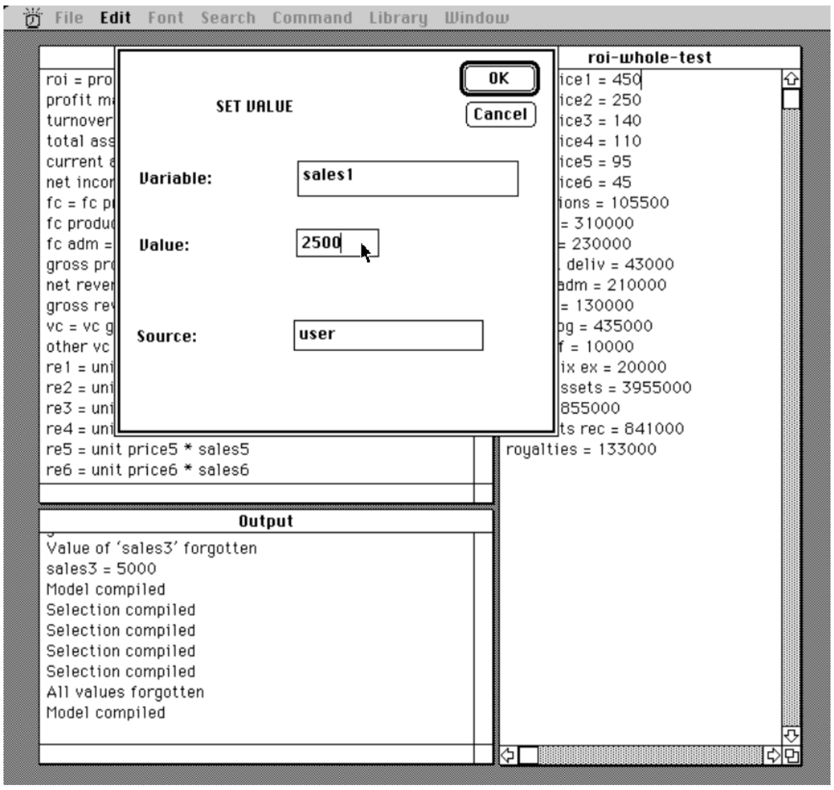


Diagram 3.1.3: ROI as constraint-propagation net

Subsequently, as some values are introduced into the model (either as constants, cf. window *roi-whole-test(I)*, or as variables by means of the **set-value dialog** as shown in diagram 3.1.4), the system propagates them to the other variables till the whole net can be fully determined.

Whenever the user wants to know the value of a single variable, he needs only to ask the system for it (**get-value dialog**, cf. diagram 3.1.5). The answers brought by the system are depicted in the window *output*.

Sometimes, e.g. when performing economic analysis, it will be necessary to reset some variables' values. This will be achieved by means of the **forget-value dialog** (cf. diagram 3.1.6).



```
roi = pro
profit m
turnover
total ass
current a
net incor
fc = fc p
fc produ
fc adm =
gross pr
net rever
gross re
vc = vc g
other vc
re1 = uni
re2 = uni
re3 = uni
re4 = uni
re5 = unit price5 * sales5
re6 = unit price6 * sales6
```

Variable: sales1

Value: 2500

Source: user

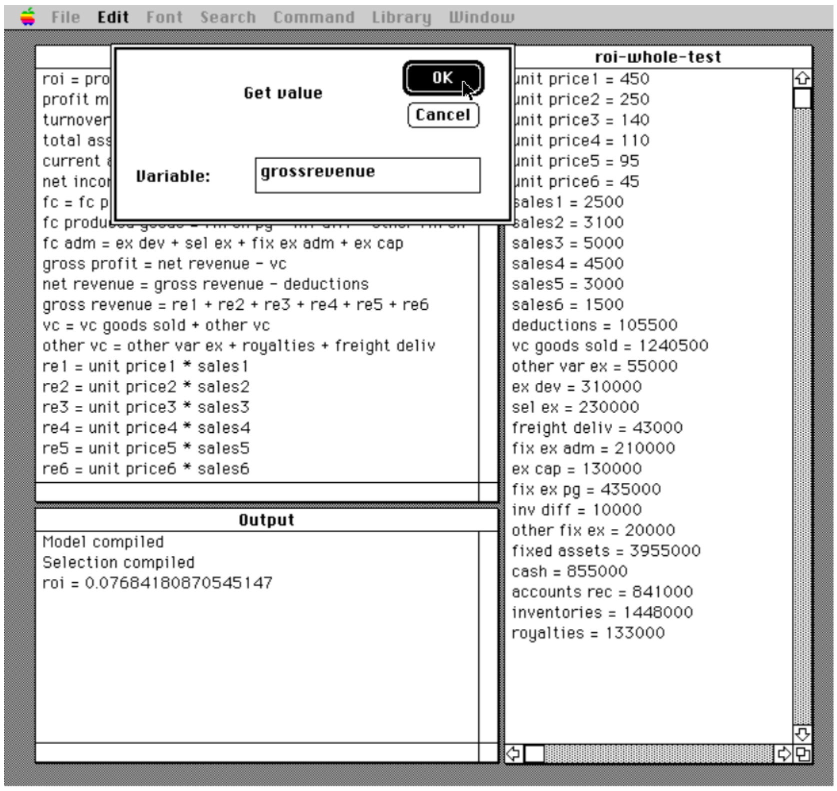
```
roi-whole-test
ice1 = 450
ice2 = 250
ice3 = 140
ice4 = 110
ice5 = 95
ice6 = 45
ions = 105500
= 310000
= 230000
 deliv = 43000
adm = 210000
= 130000
pg = 435000
f = 10000
ix ex = 20000
ssets = 3955000
855000
ts rec = 841000
royalties = 133000
```

```
Output
Value of 'sales3' forgotten
sales3 = 5000
Model compiled
Selection compiled
Selection compiled
Selection compiled
Selection compiled
All values forgotten
Model compiled
```

Diagram 3.1.4: dialog window for the command (set-value! sales1 2500 'user)

3.2 Performing Economic Analysis

One of the strengths of the ACMS approach lies in the flexibility reached by the bi-directionality of the computation. In fact, since the model formulated and compiled in the preceding section can be detached from the querying process, it can be used directly to perform economic analysis in both directions: deductive and inductive (e.g. sensitivity analysis).



```
grossrevenue
```

```
roi = pro
profit m
turnover
total ass
current
net incor
fc = fc p
fc produ
fc adm = ex dev + sel ex + fix ex adm + ex cap
gross profit = net revenue - vc
net revenue = gross revenue - deductions
gross revenue = re1 + re2 + re3 + re4 + re5 + re6
vc = vc goods sold + other vc
other vc = other var ex + royalties + freight deliv
re1 = unit price1 * sales1
re2 = unit price2 * sales2
re3 = unit price3 * sales3
re4 = unit price4 * sales4
re5 = unit price5 * sales5
re6 = unit price6 * sales6
```

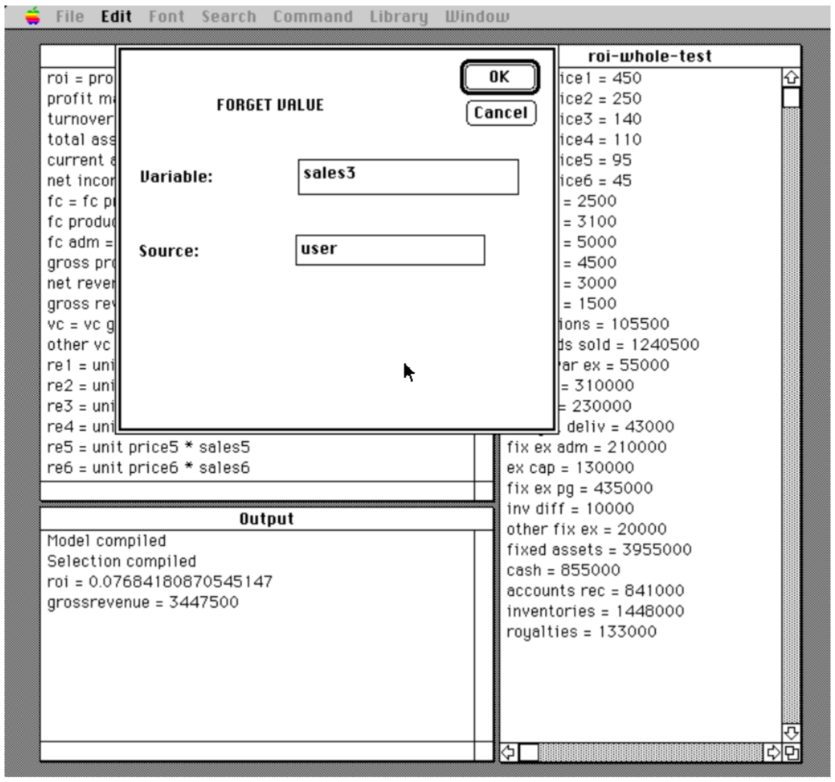
Output

```
Model compiled
Selection compiled
roi = 0.07684180870545147
```

roi-whole-test

```
unit price1 = 450
unit price2 = 250
unit price3 = 140
unit price4 = 110
unit price5 = 95
unit price6 = 45
sales1 = 2500
sales2 = 3100
sales3 = 5000
sales4 = 4500
sales5 = 3000
sales6 = 1500
deductions = 105500
vc goods sold = 1240500
other var ex = 55000
ex dev = 310000
sel ex = 230000
freight deliv = 43000
fix ex adm = 210000
ex cap = 130000
fix ex pg = 435000
inv diff = 10000
other fix ex = 20000
fixed assets = 3955000
cash = 855000
accounts rec = 841000
inventories = 1448000
royalties = 133000
```

Diagram 3.1.5: dialog window for the command (get-value! grossrevenue)



```
sales3
user
```

```
roi = pro
profit m
turnover
total ass
current
net incor
fc = fc p
fc produ
fc adm =
gross pr
net rever
gross re
vc = vc g
other vc
re1 = uni
re2 = uni
re3 = uni
re4 = uni
re5 = unit price5 * sales5
re6 = unit price6 * sales6
```

Output

```
Model compiled
Selection compiled
roi = 0.07684180870545147
grossrevenue = 3447500
```

roi-whole-test

```
ice1 = 450
ice2 = 250
ice3 = 140
ice4 = 110
ice5 = 95
ice6 = 45
= 2500
= 3100
= 5000
= 4500
= 3000
= 1500
ions = 105500
ds sold = 1240500
ar ex = 55000
= 310000
= 230000
deliv = 43000
fix ex adm = 210000
ex cap = 130000
fix ex pg = 435000
inv diff = 10000
other fix ex = 20000
fixed assets = 3955000
cash = 855000
accounts rec = 841000
inventories = 1448000
royalties = 133000
```

Diagram 3.1.6: dialog window for the command (forget-value! sales3 'user)

To illustrate this, let us consider the following examples:

a) Mr. Pearson, the chief-executive of the division A, is confident about the possibility of increasing the number of units of type 3 sold from 5,000 to 7,500. He wants to know the new values for the following economic variables: ROI, profit margin and turnover (diagram 3.2.1). In order to do that, Mr. Pearson has to perform the following operations (cf. window Output in diagram 3.2.1):

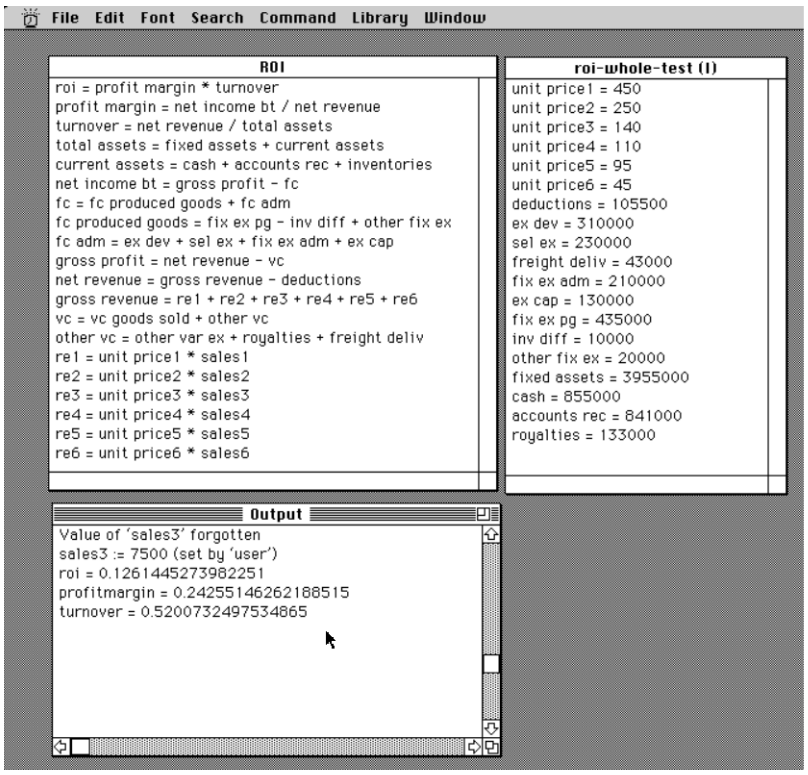
(1) (forget-value! sales3)

(2) (set-value! sales3 7500 'user)

(3) (get-value! roi)

(4) (get-value! profitmargin)

(5) (get-value! turnover)



```
roi = profit margin * turnover
profit margin = net income bt / net revenue
turnover = net revenue / total assets
total assets = fixed assets + current assets
current assets = cash + accounts rec + inventories
net income bt = gross profit - fc
fc = fc produced goods + fc adm
fc produced goods = fix ex pg - inv diff + other fix ex
fc adm = ex dev + sel ex + fix ex adm + ex cap
gross profit = net revenue - vc
net revenue = gross revenue - deductions
gross revenue = re1 + re2 + re3 + re4 + re5 + re6
vc = vc goods sold + other vc
other vc = other var ex + royalties + freight deliv
re1 = unit price1 * sales1
re2 = unit price2 * sales2
re3 = unit price3 * sales3
re4 = unit price4 * sales4
re5 = unit price5 * sales5
re6 = unit price6 * sales6
```

```
unit price1 = 450
unit price2 = 250
unit price3 = 140
unit price4 = 110
unit price5 = 95
unit price6 = 45
deductions = 105500
ex dev = 310000
sel ex = 230000
freight deliv = 43000
fix ex adm = 210000
ex cap = 130000
fix ex pg = 435000
inv diff = 10000
other fix ex = 20000
fixed assets = 3955000
cash = 855000
accounts rec = 841000
royalties = 133000
```

```
Value of 'sales3' forgotten
sales3 := 7500 (set by 'user')
roi = 0.1261445273982251
profitmargin = 0.24255146262188515
turnover = 0.5200732497534865
```

Diagram 3.2.1: economic analysis - case a)

b) Mr. Pearson is usually very prudent in his expectations. Therefore, he wants to know if a reduction of the inventories from 1,448,000.- to 600,000.- will be sufficient to obtain the same ROI, in the case where it would not be possible to increase the number of units sold of type 3 product (diagram 3.2.2). In order to do that, he has to perform the following operations (cf. window Output in diagram 3.2.2):

(1) (forget-value! inventories 'user)

(2) (set-value! inventories 600000 'user)

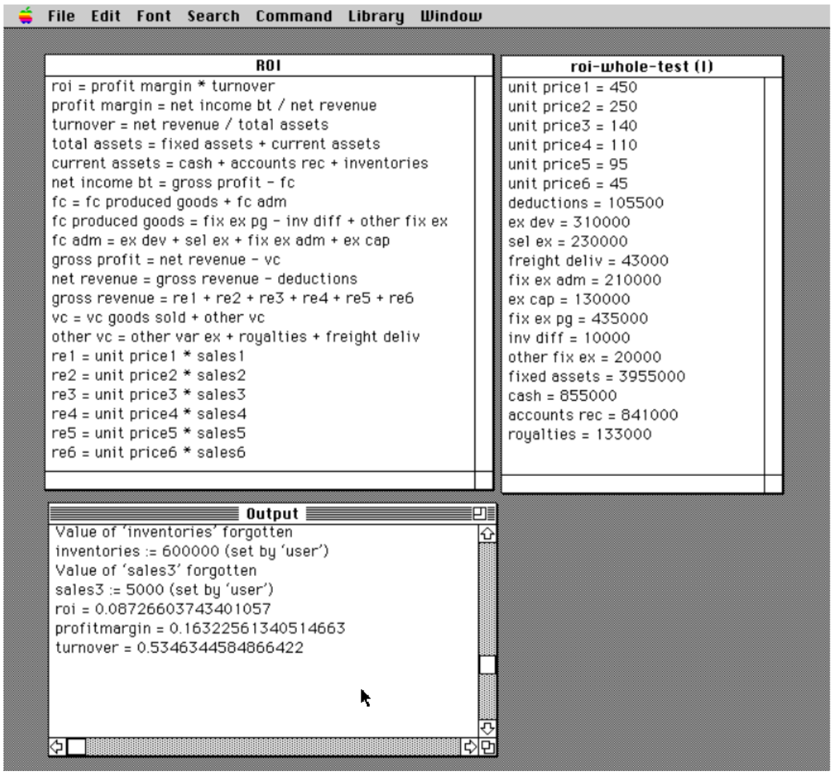
(3) (forget-value! sales3)

(4) (set-value! sales3 5000 'user)

(5) (get-value! roi)

(6) (get-value! profitmargin)

(7) (get-value! turnover)



```
ROI
roi = profit margin * turnover
profit margin = net income bt / net revenue
turnover = net revenue / total assets
total assets = fixed assets + current assets
current assets = cash + accounts rec + inventories
net income bt = gross profit - fc
fc = fc produced goods + fc adm
fc produced goods = fix ex pg - inv diff + other fix ex
fc adm = ex dev + sel ex + fix ex adm + ex cap
gross profit = net revenue - vc
net revenue = gross revenue - deductions
gross revenue = re1 + re2 + re3 + re4 + re5 + re6
vc = vc goods sold + other vc
other vc = other var ex + royalties + freight deliv
re1 = unit price1 * sales1
re2 = unit price2 * sales2
re3 = unit price3 * sales3
re4 = unit price4 * sales4
re5 = unit price5 * sales5
re6 = unit price6 * sales6
```

```
roi-whole-test (1)
unit price1 = 450
unit price2 = 250
unit price3 = 140
unit price4 = 110
unit price5 = 95
unit price6 = 45
deductions = 105500
ex dev = 310000
sel ex = 230000
freight deliv = 43000
fix ex adm = 210000
ex cap = 130000
fix ex pg = 435000
inv diff = 10000
other fix ex = 20000
fixed assets = 3955000
cash = 855000
accounts rec = 841000
royalties = 133000
```

```
Output
Value of 'inventories' forgotten
inventories := 600000 (set by 'user')
Value of 'sales3' forgotten
sales3 := 5000 (set by 'user')
roi = 0.08726603743401057
profitmargin = 0.16322561340514663
turnover = 0.5346344584866422
```

Diagram 3.2.2: economic analysis - case b)

c) Another alternative occurs to Mr. Pearson: to make product 2 more attractive. In fact, the inventories' value is not flexible enough due in part to the irregular demand in the branch. This means that a great safety stock is needed in order to avoid stockouts. Therefore, the reduction of the inventories from 1,448,000.- to 600,000.- is too optimistic after all. On the other hand, product 2 is still young and innovative in its fashion. In some sense it can truly be considered as a "cash cow". He is asking himself about the units of product 2 that should be sold, in order to obtain a net revenue of 4,000,00.-(diagram 3.2.3). To do that, he has to perform the following operations (cf. window Output in diagram 3.2.3):

(1) (forget-value! inventories 'user)

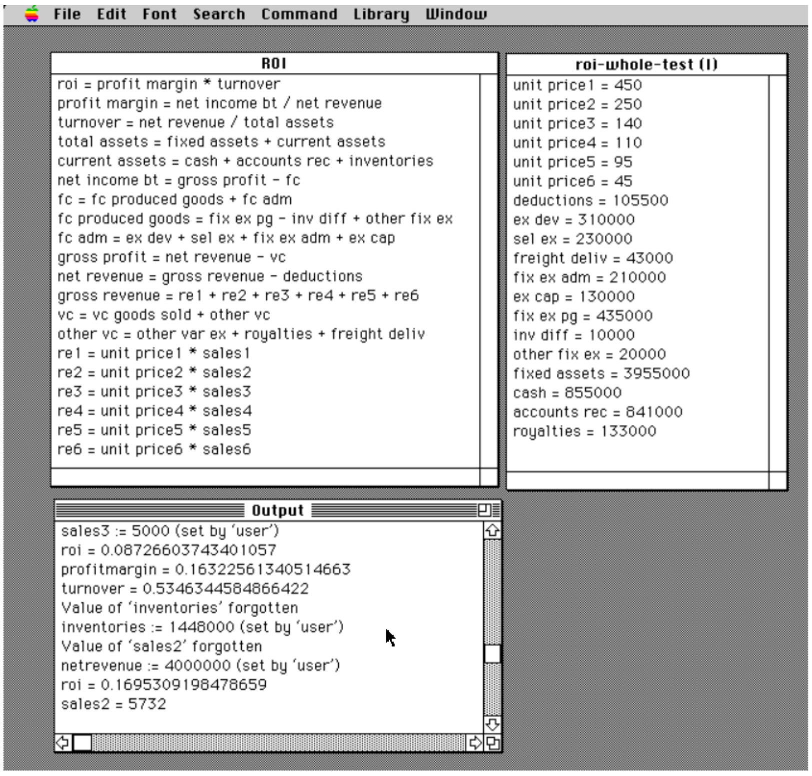
(2) (set-value! inventories 1448000 'user)

(3) (forget-value! sales2)

(4) (set-value! netrevenue 4000000 'user)

(5) (get-value! roi)

(6) (get-value! sales2)



```
roi = profit margin * turnover
profit margin = net income bt / net revenue
turnover = net revenue / total assets
total assets = fixed assets + current assets
current assets = cash + accounts rec + inventories
net income bt = gross profit - fc
fc = fc produced goods + fc adm
fc produced goods = fix ex pg - inv diff + other fix ex
fc adm = ex dev + sel ex + fix ex adm + ex cap
gross profit = net revenue - vc
net revenue = gross revenue - deductions
gross revenue = re1 + re2 + re3 + re4 + re5 + re6
vc = vc goods sold + other vc
other vc = other var ex + royalties + freight deliv
re1 = unit price1 * sales1
re2 = unit price2 * sales2
re3 = unit price3 * sales3
re4 = unit price4 * sales4
re5 = unit price5 * sales5
re6 = unit price6 * sales6
```

```
unit price1 = 450
unit price2 = 250
unit price3 = 140
unit price4 = 110
unit price5 = 95
unit price6 = 45
deductions = 105500
ex dev = 310000
sel ex = 230000
freight deliv = 43000
fix ex adm = 210000
ex cap = 130000
fix ex pg = 435000
inv diff = 10000
other fix ex = 20000
fixed assets = 3955000
cash = 855000
accounts rec = 841000
royalties = 133000
```

```
sales3 := 5000 (set by 'user')
roi = 0.08726603743401057
profitmargin = 0.16322561340514663
turnover = 0.5346344584866422
Value of 'inventories' forgotten
inventories := 1448000 (set by 'user')
Value of 'sales2' forgotten
netrevenue := 4000000 (set by 'user')
roi = 0.1695309198478659
sales2 = 5732
```

Diagram 3.2.3: economic analysis - case c)

4 Outlook

In the present paper we have discussed the benefits of the constraint propagation approach as a modeling tool for decision support in the context of arithmetic models, and a new constraint propagation language, called ACMS, has been presented. While this language is well suited for the management of this kind of model, extensions in several directions are possible. It would be interesting for instance to have some extensions to solve models containing cycles (i.e. simultaneous equations), to define new types of numerical data (e.g. fuzzy numbers) and statistical distributions, to perform certain kinds of symbolic computations (e.g. term rewriting), to combine arithmetic elements with logical constraints , etc. Some of these extensions are actually studied and implemented.

References

[ABELSON, SUSSMAN 1985]: Abelson H., Sussman, G. J. - Structure and Interpretation of Computer Programs. The MIT Press, Mass. 1985.

[CARDONA 1991]: Cardona L. - Einführung in den Macsheme Toolsmith. *Institute for Automation and Operations Research, University of Fribourg 184*.

[CARDONA, KOHLAS, SEELIGER 1992]: Cardona L., Kohlas J., Seeliger O. - Einführung in MacNet. *Institute for Automation and Operations Research, University of Fribourg 202*.

[CARDONA, GAUCH, KOHLAS, ZAHNO 1992]: Cardona L., Gauch D., Kohlas J., Zahno B. - Einführung in MacNet$^+$. *Institute for Automation and Operations Research, University of Fribourg 203*.

[DUBOIS, PRADE 1988]: Dubois D., Prade H. - Possibility Theory, Plenum, New York 1988.

[IBM 1979]: IBM - Planungshandbuch. IBM Form GE12-1511-1. IBM Deutschland GmbH. Oktober 1979.

[KERSCHBERG 1986]: Kerschberg L. - Expert Database Systems. *Proceedings From the First International Workshop*. The Benjamin/Cummings Publishing Company, Inc., Menlo Park, Califonia 1986.

[KOTLER 1971]: Kotler P. - Marketing Decision Making. Holt, Rinehart and Winston Marketing Series. New York 1971.

[LAFUE, SMITH 1986]: Lafue G. M. E., Smith R. G. - Implementation Of A Semantic Integrity Manager With A Knowledge Representation System. In [KERSCHBERG 1986].

[LELER 1988]: Leler Wm. - Constraint Programming Languages. Their Specification and Generation. Addison-Wesley, Reading, Mass. 1988.

[LIGHTSHIP 1989]: Lightship Scheme, MacScheme + Toolsmith, a Scheme Development System, Lightship Software 1989.

[MORGENSTERN 1986]: Morgenstern M. - The Role of Constraints in Databases, Expert Systems, and Knowledge Representations. In [KERSCHBERG 1986].

[NAKAZAWA, ISODA, MIYAZAKI, AISO 1986]: Nakazawa M., Isoda M., Miyazaki J., Aiso H. - MILK: Multi Level Interface Logic Simulator at Keio University. Experience in Using the CONSTRAINTS Language. In [KERSCHBERG 1986].

[PORTENIER 1990]: Portenier C. - Traitement de modèles quantitatifs au moyen de la propagation de contraintes. *Institute for Automation and Operations Research, University of Fribourg 178.*

[SHEPHERD, KERSCHBERG 1986]: Shepherd A., Kerschberg L. - Constraint Management in Expert Database Systems. In [KERSCHBERG 1986].

[STEELE 1990]: Steele G. L. - Common Lisp. Digital Press 1990.

[ZADEH 1975]: Zadeh L.A. - Fuzzy Logic and Approxiamtive Reasoning, Synthese, 30, pp 407-428.