TR/11/93 October 1993

INTRODUCING NEW CONSTRUCTS FOR DATA MODELLING AND COLUMN GENERATION IN LP MODELLING LANGUAGES

C. Lucas, G. Mitra and S. Moody
and B. Kristjansson

INTRODUCING NEW CONSTRUCTS FOR

DATA MODELLING AND COLUMN GENERATION

IN LP MODELLING LANGUAGES

by

C. Lucas, G. Mitra and S. Moody
Brunel University, Uxbridge, Middlesex, UK
U.K.

and

Bjarni Kristjansson
Maximal Software, Iceland

October 1993

Keywords

Linear Programming Modelling, Linear Programming Modelling Languages, Language Syntax, Linear Programs in Declarative Form, Databases, Relational Database Constructs, Column Generation, Objects, Methods, Procedures

Introducing New Constructs for Data Modelling and Column Generation in LP Modelling Languages

Contents

0. Abstract

Through popular implementation of structured query language (SQL) and query-by-example (QBE) relational databases have become the de-facto industry standard for data modelling. We consider the indices, sets, and the declarative form of Linear Programming (LP) modelling languages and introduce new constructs which provide direct link to the database systems. The models constructed in this way are data driven and display a dynamic structure. We then show how this approach can be naturally extended to include column generation features stated in procedural forms within an otherwise declarative modelling paradigm.

1. Background & Introduction

The software support for the solution of optimization problems has advanced rapidly in the last two decades and in order to make use of this software, models representing an optimization problem must be presented in a machine readable form to the optimizer. For an LP model the declarative statement, known as the "modeler's form" *(Fourer, 1983)*, is suitable for model description, documentation, easy modification and is comprehensible. Unfortunately, until recently the dominance of sparse simplex (SSX) *(Levkovitz & Mitra, 1993)* solvers and their requirement to have the constraint matrix specified in column order meant that the "algorithms form" *(Fourer, 1983)* has to be taken into account. This form depends on the data structure used in the implementation of algorithms and typically the constraints had to be presented in a column-wise representation of the non-zero elements of the matrix. To obtain this, a suitable input format had to be defined and today the most well established and the defacto standard is the MPS format *(IBM, 1976)*. For the purpose of modelling, the sparse and column oriented data definition of MPS is most inadequate. It is now acknowledged that algebraic and specially equational forms provide an easier and more understandable model definition. A few algebraic languages have been designed to support these equational forms. In addition to this essential feature LP Modelling Languages today have other important and well established roles in Mathematical Programming such as rapid application development and model investigation. Today there are many systems in existence which provide essential modelling support. For example, AMPL *(Fourer,Gay & Kernighan, 1987)*, GAMS *(Bisschop & Meeraus, 1982)*, AIMMS (Bisschop, 1993), LPL *(Hurlimann, 1987)*, LP-MODEL *(Ashford & Daniel, 1987)*, MPL *(Kristjansson, 1993)*, MODLER *(Greenberg, 1991b)*, SML *(Geoffrion, 1992a & 1992b)*, for a review of such systems see (Steiger & Sharda, 1993), (Greenberg, 1991a).

Most modelling systems provide a language in which the user can specify models in a declarative algebraic form and the system automatically transcribes this model and creates a matrix definition for the optimizer. Although this provides the modeller with a powerful computer based tool we can identify at least two main deficiencies of these systems. Firstly, the algebraic statement of the model and the actual data for the problem are usually kept separate so that the definitive problem may be distinguished from particular instances of the

problem. Most modelling languages employ data tables which are created externally and communicated as (ascii) text files. Usually within a modelling language, altering the dimensions of a problem invalidates the given data and model instance and requires changes to be made manually to the data. This is not altogether desirable and a more flexible approach to the data definition is required. Since it is standard practice to use databases for information storage and retrieval in industry there is a clear need for the integration of modelling languages with relational databases (RDBs). Some modelling systems have recently introduced such connections *(Baker, 1993)* but this is relatively new and much work is needed to maintain a consistent approach and user-friendly modelling environment. By extending the syntax of the modelling language MPL *(Kristjansson, 1991)* with new constructs which makes use of the existing well known RDB structures, we provide a method of connecting corporate databases with a modelling system. Secondly, whilst most LP models are stated declaratively, there are well known optimization problem classes, for example, crew scheduling *(Darby-Dowman & Mitra, 1985)*, vehicle scheduling *(Christofides, Mingozzi & Toth, 1979), (Fischer & Jaikumar, 1981), (Mitra, Lucas, Darby-Dowman & Smith, 1994), cutting stock (Gilmore & Gomory, 1965),(Chambers & Dyson, 1976)* contract selection *(Maros, Mitra & Moody, 1993)* which require a procedural method to generate the matrix columns, often called activities and typically representing legal duties, permitted schedules, appropriate cutting patterns, alternative contract specifications. Currently no LP modelling language is able to provide adequate support for constructing this class of models. Traditionally column generation is carried out in a high level programming language and cannot be connected to the sets, indices and constraint groups specified in declarative LP modelling languages. We have introduced new language constructs which use the modern programming concept of an object and provide the facility to define procedures and methods connecting the indexing structure and constraints of the LP model. Through these syntactic and operational extensions we have designed a modelling tool with much greater potential and richer functionality, thereby extending the applicability of LP modelling systems to more complex problem domains.

The rest of this paper is organized in the following way. In section 2 the declarative structure of an LP modelling language is explained and its syntax is illustrated with an example of a distribution problem with transshipment. In section 3 new constructs are introduced to

enable the modelling language to be connected to a database. The distribution example is used to demonstrate the new syntax. Column generation is discussed in section 4 and new constructs are introduced in the modelling language to permit the specification of procedural rules. The distribution problem is extended to include multiple commodities and this example is used to illustrate the new syntax and method of operation. Section 5 contains a discussion and some concluding remarks.

2. The Declarative Structure of an LP Modelling Language: MPL

The modelling of LP problems can be formally achieved through three logical steps. In *step one*, the subscripts and their ranges are specified: these are essentially the sets and dimensions of the model. In *step two*, the data tables and model decision variables are defined in terms of these subscripts. In *step three*, the model constraints are specified in a row-wise fashion connecting the previously defined data table entries and decision variables. *(Lucas & Ultra, 1988, p365)*.

MPL (or Mathematical Programming Language) is a modelling system which enables problems to be formulated in a declarative form and after executing the model statements it generates an input file (eg MPS file) for an optimizer. The key features of MPL are described in the MPL Users Guide *(Kristjansson, 1991, pl.2 -1.4)*.

The structure of MPL reflects the above mentioned modelling strategy. A problem expressed in MPL is divided into sections by the use of keywords and has the following format:

TITLE	The problem name.
INDEX	Index sets which define the dimensions of the problem.
DATA	Scalars, datavectors, datafiles.
DECISION	Vector decision variables.
MACRO	Macros for repetitive parts.
MODEL	
MAX (or MIN)	The objective function.
SUBJECT TO	The constraints.
BOUNDS	Simple upper and lower bounds.
FREE	Free variables.
INTEGER	Integer variables.

BINARY Binary (0/1) variables.
SOS1 Special Ordered Set (Type 1) variables
SOS2 Special Ordered Set (Type 2) variables
END

Apart from the objective function (distinguished by the keyword MAX or MIN) and the constraints (identified by the keywords SUBJECTED TO), all sections are optional.

Distribution Problem

To illustrate the basic syntax of the language consider a single product distribution problem with transshipment.

Subscripts, Ranges

$i \in I$ denotes factories

$j \in J$ denotes depots

$k \in K$ denotes customers

Decision Variables

x_{ij} denotes the quantity sent from factory i to depot j

x'_{ik} denotes the quantity sent from factory i to customer k

x''_{jk} denotes the quantity sent from depot j to customer k

Coefficients

c_{ij} denotes the unit cost for distribution between factory i and depot j

c'_{ik} denotes the unit cost for distribution between factory i and customer k

c''_{jk} denotes the unit cost for distribution between depot j and customer k

f_i denotes the capacity of factory i

d_j denotes the maximum monthly throughput for depot j

r_k denotes the monthly requirement for customer k

Linear Constraint Relations

$$\text{Minimize cost} = \sum_{i\in I}\sum_{j\in J} c_{ij}x_{ij} + \sum_{i\in I}\sum_{k\in K} c'_{ik}x'_{ik} + \sum_{j\in J}\sum_{k\in K} c''_{jk}x''_{jk}$$

subject to

$$\sum_{j\in J} x_{ij} + \sum_{k\in K} x'_{ik} \le f_i, \qquad i \in I$$

$$\sum_{i\in I} x_{ij} \le d_i, \qquad j \in J$$

$$\sum_{i\in I} x'_{ik} + \sum_{j\in J} x''_{jk} \ge r_k, \qquad k \in K$$

$$\sum_{i\in I} x_{ij} - \sum_{k\in K} x''_{jk} = 0, \qquad j \in J$$

$$x_{ij}, x'_{ik}, x''_{jk} \ge 0, \ \forall i \in I, j \in J, k \in K$$

Consider a simple instance of this model taken from *Williams (1990)* where a company has two factories and four depots. It sells its product to six customers each of whom may be supplied either from a depot or direct from the factory. Not all the routes between factories, depots and customers exist; the cost data for these routes are not supplied and the corresponding decision variables are not defined. The company has to pay distribution costs (in £s per ton) for the deliveries (factory_dept_cost, factory_customer_cost and depot_customer_cost). In a simple model routes which are not possible are allocated a high cost in the data tables (9000) and the corresponding decision variables are excluded using except where statements. Each factory has a monthly capacity (factory_capacities) and each depot has a maximum monthly throughput which cannot be exceeded (depot_capacities). In addition each customer has a monthly requirement which must be met (customer_requirements). The company wishes to find a distribution pattern which minimizes cost. The model is stated in MPL in figure 1.

```
TITLE Distribution;

INDEX
    factories   := (Liverpool, Brighton);
    depots      := (Newcastle, Birmingham, London, Exeter);
    customers := (cl,c2,c3,c4,c5,c6);
DATA
    FactoryCap[factories]  :=  [Liverpool, 150000,
                                Brighton,   200000];

    DepotCap[depots]       :=  [Newcastle,    70000,
                                Birmingham, 50000,
                                London,     100000,
```

```
                                    Exeter,          40000];
CustomerReq[customers]   :=  [c1,  50000,
                              c2,  10000,
                              c3,  40000,
                              c4,  35000,
                              c5,  60000,
                              c6,  20000];
FactoryDepotCost[factories,depots]  :=
   [Liverpool,   Newcastle,       0.5,
    Liverpool,   Birmingham,      0.5,
    Liverpool,   London,          1.0,
    Liverpool,   Exeter,          0.2,
    Brighton,    Birmingham,      0.3,
    Brighton,    London,          0.5,
    Brighton,    Exeter,          0.2];

FactoryCustCost[factories,customers]  :=
   [Liverpool,   c1,   1.0,
    Liverpool,   c3,   1.5,
    Liverpool,   c4,   2.0,
    Liverpool,   c6,   1.0,
    Brighton,    c1,   2.0] ;

DepotCustCost[depots,customers] :=
    [Newcastle,    c2, 1.5,
     Newcastle,    c3, 0.5,
     Newcastle,    c4, 1.5,
     Newcastle,    c6, 1.0,
     Birmingham,   c1, 1.0,
     Birmingham,   c2, 0.5,
     Birmingham,   c3, 0.5,
     Birmingham,   c4, 1.0,
     Birmingham,   c5, 0.5,
     London,       c2, 1.5,
     London,       c3, 2.0,
     London,       c5, 0.5,
     London,       c6, 1.5,
     Exeter,       c3, 0.2,
     Exeter,       c4, 1.5,
     Exeter,       c5, 0.5,
     Exeter,       c6, 1.5];

DECISION VARIABLES
     FactoryDepotQty[factories,depots]     → FD WHERE (FactoryDepotCost);
     FactoryCustQty[factories,customers]   → FC WHERE (FactoryCustCost);
     DepotCustQty[depots,customers]        → DC WHERE (DepotCustCost);

MODEL
     MIN cost = SUM(factories,depots: FactoryDepotCost * FactoryDepotQty) +
                SUM(factories,customers: FactoryCustCost * FactoryCustQty) +
                SUM(depots,customers: DepotCustCost * DepotCustQty);
SUBJECT TO

     FactoryCapacity[factories]  → FCAP :
     SUM(depots: FactoryDepotQty) + SUM(customers: FactoryCustQty)
```

```
    <= FactoryCap;
    DepotCapacity[depots] → DCAP :
    SUM(factories: FactoryDepotQty)  <=  DepotCap;

    DepotBalance[depots] → DBAL :
    SUM(customers: DepotCustQty)  =  SUM(factories: FactoryDepotQty);
    CustomerRequirement[customers] → CREQ :
    SUM(factories: FactoryCustQty)   +   SUM(depots: DepotCustQty)
    = CustomerReq;

END
```

Figure 2.1 *Distribution example in MPL*

Throughout the MPL model, meaningful names of any length may be used to aid documentation. There are three index sets in this model, namely factories, depots and customers. All subsequent vectors are defined in terms of these index sets. The data for this problem has been embedded in the model definition, but it is also possible (and preferable) to store these externally and refer to them in the MPL model with the keyword DATAFILE. The data is in sparse format; dense format is also possible. Decision variables are provided with stubs (for example "→ FD") which are abbreviations which enable the user to have control of the MPS names generated by MPL. (Stubs are also used in the constraint section.) The WHERE clauses enclosed in brackets limit the number of variables being defined: variables are only created when the appropriate values exist in the data tables defined previously. The objective is to minimize cost which is the total distribution cost. The keyword SUM defines a summation to be carried out over the indices specified in the ensuing brackets and delimited by a colon. The four mathematical operators: addition, subtraction, multiplication and division, are represented by the usual symbols: + ,-,*, /, respectively.

The constraint FactoryCapacity defines two constraints: one for each factory. This constraint states that the total quantities shipped out from the factories to the depots or customers cannot exceed the factories' capacities. There is a similar constraint for the depots' capacities. The DepotBalance constraint ensures that at each depot, the quantities that arrive from the factories are all shipped out to customers: ie. there is no storage, loss or gain. The

final constraint requires that the customers' requirements are satisfied.

Having defined the model in this manner using MPL's editor, the user selects the pulldown menu to optimize the model. This provides only a brief overview of the structure and syntax of MPL. For more detailed information see the user guide *(Kristjansson, 1991).*

3. Relational Data Models

3.1 Constructs connecting LP Modelling Languages with Relational Databases

The structures of RDBs are well known and understood: for a review see *Date (1981), Ullman (1982), Butler, Bloor & Bleach (1990).* In *Lucas, Mitra, Moody & Kristjansson (1993)* we discuss connection of MPL with an RDB system. The constructs which connect RDBs with< LP modelling languages may be divided into three groups: constructs for performing set operations (such as union, intersection and difference), constructs for performing database operations (such as projection, selection and join) and constructs for importing and exporting sets and data.

Set operations such as union, intersection and difference are already well defined constructs in MPL: see *(Lucas,Mitra, Moody & Kristjansson, 1993).* Relational operators act on the tuples (rows) or attributes (columns) of database tables. For example, consider the following database table.

FC Route ID	Factory Name	Customer ID	Cost
FC1	Liverpool	C1	1.0
FC2	Liverpool	C3	1.5
FC3	Liverpool	C4	2.0
FC4	Liverpool	C6	1.0
FC5	Brighton	C1	2.0

Table *fcrout*

This table provides information about the routes from factories to customers. The primary key "FC Route ID" is shown underlined.

The selection operator "selects" (horizontally) a subset of the tuples in a table. For example, to obtain all customers supplied by the Liverpool factory directly, a selection of all tuples

(rows) of table *fcrout* for which the attribute Factory Name has value Liverpool is carried out resulting in the following table:

FC Route ID	Factory Name	Customer ID	Cost
FC1	Liverpool	C1	1.0
FC2	Liverpool	C3	1.5
FC3	Liverpool	C4	2.0
FC4	Liverpool	C6	1.0

Liverpool's customers

The select operator is unary and therefore cannot be used to choose tuples from more than one relation. The degree of the relation resulting from a select operation is the same as that of the original relation since it has the same attributes. Selections are commutative so a sequence of selections may be carried out in any order.

The projection operator, on the other hand, makes a vertical selection of a relation, choosing some attributes (columns) and eliminating others. If it were necessary to use only certain attributes of a relation, the project operator is used to "project" the relation over these attributes. For example, suppose it was required to obtain a set of all customers that may be supplied directly by factories. Then a project operation from the table *fcrout* over the attribute Customer ID would achieve this, resulting in the following table. Customer C1 is supplied by both factories but this value is not duplicated in the resulting table. This ensures that the result of a project operation is also a relation.

Customer ID
C1
C3
C4
C6

Customers supplied directly by factories

The project operator is also unary and has degree equal to the number of attributes specified in the projection list. If some attributes projected are non-key attributes then it is possible that some duplication will occur and will therefore have to be eliminated. The number of tuples in a relation resulting from a projection is less than or equal to the number of tuples in the original relation. If the projection list includes the key of the relation, then the resulting table will have the same number of tuples as the original. Projections are not commutative.

The join operator is used to combine related tuples from two relations into one relation. For example, the tables *dcrout* and *depcap* below provide information about the routes from depots to customers, and the capacity of depots, respectively.

DC Route ID	Depot Name	Customer ID	Cost
DC1	Newcastle	C2	1.5
DC2	Newcastle	C3	0.5
DC3	Newcastle	C4	1.5
DC4	Newcastle	C6	1.0
DC5	Birmingham	C1	1.0
DC6	Birmingham	C2	0.5
DC7	Birmingham	C3	0.5
DC8	Birmingham	C4	1.0
DC9	Birmingham	C5	0.5
DC10	London	C2	1.5
DC11	London	C3	2.0
DC12	London	C5	0.5
DC13	London	C6	1.5
DC14	Exeter	C3	0.2
DC15	Exeter	C4	1.5
DC16	Exeter	C5	0.5
DC17	Exeter	C6	1.5

Table *dcrout*

Depot Name	Max Thoughput
Newcastle	70 000
Birmingham	50 000
London	100 000
Exeter	40 000

Table *depcap*

Suppose it is necessary to have a limit on the amount a customer may receive from a depot (assuming it receives the depot's total stock and the depot is operating at full capacity). Then a join of these two tables is required such that the depot name in both tables are matched. This results in a wider table, as shown below.

DC Route ID	Depot Name	Customer ID	Cost	Max Throughput
DC1	Newcastle	C2	1.5	70 000
DC2	Newcastle	C3	0.5	70 000
DC3	Newcastle	C4	1.5	70 000
DC4	Newcastle	C6	1.0	70 000
DC5	Birmingham	C1	1.0	50 000
DC6	Birmingham	C2	0.5	50 000
DC7	Birmingham	C3	0.5	50 000
DC8	Birmingham	C4	1.0	50 000
DC9	Birmingham	C5	0.5	50 000
DC10	London	C2	1.5	100 000
DC11	London	C3	2.0	100 000
DC12	London	C5	0.5	100 000
DC13	London	C6	1.5	100 000
DC14	Exeter	C3	0.2	40 000
DC15	Exeter	C4	1.5	40 000
DC16	Exeter	C5	0.5	40 000
DC17	Exeter	C6	1.5	40 000

Limit on the amount a customer may receive from a depot

Thus this join operation is equivalent to performing a cartesian product of the tuples of the tables dcrout with the tuples of the relation *depcap* but only when the combination satisfies the join condition. When the join condition involves, as in this case, an equality comparison the join operator is known as an equijoin. The attributes used in the join are known as join attributes. Note that there is no repetition of the join attribute depot name as this would be superfluous. A join operator which removes the second attribute in an equijoin condition (as in this example) is called a natural join. To perform a natural join it is required that the join attributes have the same name.

These three relational operators (selection,projection and natural join) play a fundamental role in the manipulation of information stored in a database. MPL and indeed many other modelling language syntaxes require new constructs in order to deal with these operations.

3.2 MPL extended syntax - database connection

As stated earlier in most modelling languages it is usual to store data tables externally in flat files. In order to connect the modelling language to a database it is necessary to import and export database tables in the modelling language. This means that it should be possible to define index sets which are keys of a database table and to specify subsets (or

multidimensional sets). Some modelling languages handle subsets, but few deal with the specification of keys as index sets in the model.

In order to make use of the RDB structure, MPL is extended to incorporate the relation operators described in section 3.1 and new constructs are introduced to enable:

(i) the specification of RDB keys as index sets in the model;

(ii) the specification of subsets (multi-dimensional sets);

(iii) data to be imported from a database into MPL;

(iv) the specification of selection, projection and joins in defining index sets, data tables and constraints.

The syntax and use of these constructs are explained in this section by considering the distribution example introduced previously. Assume the data for this model is stored in the following database tables. The keys for each table are underlined.

Database Tables

Factory Name	Capacity
Liverpool	150 000
Brighton	200 000

Table *factcap*

Depot Name	Max Throughput
Newcastle	70 000
Birmingham	50 000
London	100 000
Exeter	40 000

Table *depcap*

Customer ID	Monthly Requirement
C1	50 000
C2	10 000
C3	40 000
C4	35 000
C5	60 000
C6	20 000

Table *custreq*

FD Route ID	Factory Name	Depot Name	Cost
FD1	Liverpool	Newcastle	0.5
FD2	Liverpool	Birmingham	0.5
FD3	Liverpool	London	1.0
FD4	Liverpool	Exeter	0.2
FD5	Brighton	Birmingham	0.3
FD6	Brighton	London	0.5
FD7	Brighton	Exeter	0.2

Table *fdrout*

FC Route ID	Factory Name	Customer ID	Cost
FC1	Liverpool	C1	1.0
FC2	Liverpool	C3	1.5
FC3	Liverpool	C4	2.0
FC4	Liverpool	C6	1.0
FC5	Brighton	C1	2.0

Table *fcrout*

DC Route ID	Depot Name	Customer ID	Cost
DC1	Newcastle	C2	1.5
DC2	Newcastle	C3	0.5
DC3	Newcastle	C4	1.5
DC4	Newcastle	C6	1.0
DC5	Birmingham	C1	1.0
DC6	Birmingham	C2	0.5
DC7	Birmingham	C3	0.5
DC8	Birmingham	C4	1.0
DC9	Birmingham	C5	0.5
DC10	London	C2	1.5
DC11	London	C3	2.0
DC12	London	C5	0.5
DC13	London	C6	1.5
DC14	Exeter	C3	0.2
DC15	Exeter	C4	1.5
DC16	Exeter	C5	0.5
DC17	Exeter	C6	1.5

Table *dcrout*

The MPL model for this example is provided in figure 3.2.1.

```
TITLE
    Distribution;

INDEX
    factories                          := DATABASE("factcap", "Factory Name");
    depots                             := DATABASE("depcap",  "Depot Name");
    customers                          := DATABASE("custreq", "Customer ID");
    FDRoutes[factories,depots]         := DATABASE("fdrout");
    FCRoutes[factories,cutomers]       := DATABASE("fcrout");
    DCRoutes[depots,customers]         := DATABASE("dcrout");

DATA
    FactoryCap[factories]       := DATABASE("factcap","Capacity");
    DepotCap[depots]            := DATABASE("depcap", "Max Throughput");
    CustomerReq[customers]  := DATABASE("custreq","Monthly Requirements");

    FactoryDepotCost[FDRoutes]         :=  DATABASE("fdrout","Cost");
    FactoryCustCost[FCRoutes]          :=  DATABASE("fcrout","Cost");
    DepotCustCost[DCRoutes]            :=  DATABASE("dcrout","Cost");

DECISION VARIABLES
    FactoryDepotQty[FDRoutes]          →  FD WHERE  (FactoryDepotCost);
    FactoryCustQty[FCRoutes]           →  FC WHERE  (FactoryCustCost);
    DepotCustQty[DCRoutes]             →  DC WHERE  (DepotCustCost);

MODEL
    MIN cost = SUM(factories,depots:        FactoryDepotCost * FactoryDepotQty) +
               SUM(factories,customers:   FactoryCustCost * FactoryCustQty) +
               SUM(depots,customers:      DepotCustCost * DepotCustQty);
SUBJECT TO
     FactoryCapacity[factories]    →  FCAP :

     SUM(FDRoutes.depots: FactoryDepotQty)
   + SUM(FCRoutes.customers: FactoryCustQty)

     <=   FactoryCap;

      DepotCapacity[depots]  → DCAP :
      SUM(FDRoutes.factories:  FactoryDepotQty)  <= DepotCap;

      DepotBalance[depots]   → DBAL :

      SUM(DCRoutes.customers: DepotCustQty)
    = SUM(FDRoutes.factories   : FactoryDepotQty);

      CustomerRequirement[customers]   → CREQ :

      SUM(FCRoutes.factories: FactoryCustQty)
    + SUM(DCRoutes.depots: DepotCustQty)
      =  CustomerReq;
END
```

Figure 3.2.1 *Distribution example in MPL with database connection.*

The algebraic representation of this LP model first involves defining sets. For example, let factories, depots, and customers denote the sets of factories, depots and customers respectively in this example. In general index sets are derived from the keys of a database table. Thus the syntax for defining index sets incorporates the keyword "DATABASE":

INDEX

set_name := DATABASE ("*database_table_name*", *"key"*);

For example, the index factories in figure 2 is defined to be the key attribute "Factory Name" from the database table "FactCap". Subsequently, "factories", rather than "Factory Name" can be used in the model, even in defining new indices. This avoids any unnecessary repetition of long names and provides good documentation on how the index sets of the model correspond to the database keys.

Subsets are also defined, for example FDRoutes represents the routes from factories to epots. Subsets like these are represented in MPL by two (or more) dimensional sets, that is sets of sets. The syntax is as follows:

INDEX

set_name[*set1*,*set2*...] := DATABASE("*database_table_name*", *set_name* = "*key*") *or*
set_name[*set1*,*set2*...] := DATABASE("*database_table_name*", "*key*");

where set1, set2,... are previously defined index sets. FDRoutes is defined from the previously defined sets factories and depots and links between factories and depots are only created for the entries in the database table fdrout. As this table has the same column names as tables factcap and depcap (from which the index sets factories and depots were created) there is no need to specify the column names again. FDRoutes is defined as a projection of "Factory Name" and "Depot Name" taken from the database table "FDRout". This index is a multi-dimensional set or, in relational terms, a multi-attribute (or composite) key for the data tables and/or constraints which are as yet to be defined. Thus this subset may be viewed as a relational table rather than as a set of routes from factories to depots. Subsets are thus defined in figure 3.2.1 for all the routes. Multidimensional sets can also be

derived from several tables or subsets using the keywords WHERE FORSOME and IN see *(Lucas, Mitra, Moody & Kristjansson, 1993)*.

Data is imported from a database in a similar way. That is,

data_table_name[set1,set2…] :=

DATABASE("*database_table_name", set_name = "attribute"*)

where *set1,set2,..* are previously defined index sets. Data tables may also be derived by manipulating pre-defined data tables, for example, the relational operation join may be performed using the "IN" operator.

The values of the decision variables obtained from the optimization may be exported back into the database using the keyword EXPORT. For example, the FactoryDepotQty values may be exported back to the database table "fdrout" and become the attribute "Qty" as follows:

FactoryDepotQty[FDRoutes] → FD WHERE (FactoryDepotCost)

```
EXPORT TO DATABASE("fdrout", "Qty");
```

For a more detailed description of these new constructs, in particular in specifying the relational operations when defining index sets, data tables and constraints see *(Lucas, Mitra, Moody & Kristjansson, 1993)*.

4. Procedural Forms and Column Generation

4.1 Column Generation in LP Modelling

Many LP models have an otherwise static structure reflecting generally the declarative nature of such models. For instance in the distribution example considered earlier, or many other resource allocation models such as assignment, transportation and production models, the sources, the rates of production and so forth, determine the coefficients of the model. They are usually specified within the model statement. There is, however, a large class of other models in which the coefficients of the constraint matrix cannot be specified immediately by the raw problem data. In these cases the model data needs to be processed by some rules or procedures in order to derive the coefficient and usually the activities or columns of the LP (or IP) constraint matrix.

4.2 Multi-commodity Network Flow: An Example of Column Oriented Modelling

Consider the extension of the distribution example introduced earlier to the case of multiple commodities (products). Assume that there are various transportation options along each possible route. For example on a route between a particular factory and depot there are several choices concerning how a given vehicle may be loaded with different combinations of products. Then a particular route with one product load combination needs to be distinguished from the same route with a different combination, as seen in the index sets ℓ_{ij}, m_{ik} and n_{jk} defined in the following model.

Subscripts, Ranges

$i = 1,2,..,I$	denotes factories
$j = 1,2,..,J$	denotes depots
$k = 1,2,..,K$	denotes customers
$p = 1,2,..,P$	denotes products
$\ell_{ij} = 1,2,..,L_{ij}$	denotes product load combinations for a route from factory i to depot j $(i=1,2,..,I;\ j = 1,2,..,J)$
$m_{ik} = 1,2,..,M_{ik}$	denotes product load combinations for a route from factory i to customer k $(i=1,2,..,I;\ k=1,2,..,K)$
$n_{jk} = 1,2,..,N_{jk}$	denotes product load combinations for a route from depot j to customer k $(j =1,2,..,J;\ k=1,2,..,K)$

Decision Variables

$X_{\ell_{ij}}$	denotes the number of vehicles with product load combination ℓ_{ij}
$X'_{m_{ik}}$	denotes the number of vehicles with product load combination m_{ik}
$X''_{n_{jk}}$	denotes the number of vehicles with product load combination n_{jk}
S_{jp}	denotes the quantity of product p stored at depot j

Coefficients

$C_{\ell_{ij}}$	unit cost for the product load combination ℓ_{ij}
$C'_{m_{ik}}$	unit cost for the product load combination m_{ik}

$c''_{n_{jk}}$ unit cost for the product load combination n_{jk}

$_{p\ell_{ij}}$ amount of product p in the product load combination ℓ_{ij}

$'_{pm_{ik}}$ amount of product p in the product load combination m_{ik}

$''_{pn_{jk}}$ amount of product p in the product load combination n_{jk}

h_{jp} cost of storage of product p at depot j

A_{ip} Maximum amount of product p that can be produced at factory i.

B_{jp} Maximum handling capacity at depot j for product p.

D_{kp} Customer k's demand for product p.

Linear Constraint Relations

Minimize cost =

$$\sum_i \sum_j \sum_{l_{ij}} c_{\ell_{ij}} x_{\ell_{ij}} + \sum_i \sum_k \sum_{m_{ik}} c'_{m_{ik}} x'_{m_{ik}} + \sum_j \sum_k \sum_{n_{jk}} c''_{n_{jk}} x''_{n_{jk}} + \sum_j \sum_p h_{jp} s_{jp} \qquad (4.2.1)$$

subject to

Factory availability

$$\sum_j \sum_{\ell_{ij}} R_{p\ell_{ij}} x_{\ell_{ij}} + \sum_k \sum_{m_{ik}} R'_{pm_{ik}} x'_{m_{ik}} \le A_{ip} \quad \forall i, \forall p \qquad (4.2.2)$$

Depot Capacity

$$\sum_i \sum_{\ell_{ij}} R_{p\ell_{ij}} x_{\ell_{ij}} \le B_{jp} \quad \forall j, \forall p \qquad (4.2.3)$$

Depot Balance

$$\sum_i \sum_{\ell_{ij}} R_{p\ell_{ij}} x_{\ell_{ij}} - \sum_j \sum_{n_{jk}} R''_{pn_{jk}} x''_{n_{ij}} = s_{jp} \quad \forall p, \forall j \qquad (4.2.4)$$

Demand

$$\sum_i \sum_{m_{ik}} R'_{pm_{ik}} x'_{m_{ik}} + \sum_j \sum_{n_{jk}} R''_{pn_{jk}} x''_{n_{jk}} = D_{kp} \quad \forall k, \forall p \qquad (4.2.5)$$

4.3 Column Structure - Procedural Rules for Generation

For the problem introduced in the previous section consider an instance defined by the data sets given in Tables 4.3.1 to Table 4.3.5. Table 4.3.1 contains the capacities of factories for

the four product ranges; Table 4.3.2 shows the throughput capacities of the depots for each of the four products; the customer demands for the four products are presented in Table 4.3.3 and Table 4.3.4 displays the dimensions of the product containers and vehicles. We have simplified the vehicle loading problem by assuming that all containers are trolleys which are placed in the vehicles and cannot be stacked.

Following permissible product combinations and the size restrictions, different vehicle loading strategies can be computed and are summarized in Table 4.3.5.

Manufacturing capacity of factory				
Factory	Product 1	Product 2	Product 3	Product 4
Liverpool	50,000	70,000	10,000	20,000
Brighton	50,000	70,000	80,000	0

Table 4.3.1

handling capacity of depots				
Depot	Product 1	Product 2	Product 3	Product 4
Newcastle	20,000	20,000	10,000	20,000
Birmingham	10,000	20,000	10,000	10,000
London	25,000	25,000	25,000	25,000
Exeter	10,000	0	30,000	0

Table 4.3.2

Demand for products				
Customer	Product 1	Product 2	Product 3	Product 4
C1	10,000	20,000	10,000	10,000
C2	10,000	0	0	0
C3	0	15,000	0	20,000
C4	0	0	35,000	0
C5	30,000	20,000	5,000	5,000
C6	5,000	5,000	10,000	0

Table 4.3.3

	product containers and vehicle sizes	
	Width	Length
Product 1	1	1
Product 2	1	1½
Product 3	1½	2
Product 4	2	2½
Vehicle	3	10

Table 4.3.4

product load for each route				
Route Factory-Depot	**Product 1**	**Product 2**	**Product 3**	**Product 4**
Liverpool-Newcastle	30	0	0	0
Liverpool-Newcastle	0	18	0	0
Liverpool-Newcastle	3	18	0	0
Liverpool-Newcastle	0	0	10	0
Liverpool-Newcastle	0	0	0	8
Liverpool-Newcastle	12	12	0	0
Liverpool-Newcastle	6	2	2	4
Liverpool-Newcastle	0	2	6	2
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
Brighton-Exeter	0	2	6	2

Table 4.3.5

The procedure for generating feasible loading schedules is shown in pseudocode in figure 4.3.6. This pseudo-procedure computes for all three types of routes (factories - customers, factories - depots and depots - customers) the corresponding loading schedules. After initialization, a routine (feasible products) is called which establishes permissible products in this route. Then the recursive routine (generate load combination) is called which computes all possible load combinations (vehicle loading strategies) for this given route. These vehicle loading strategies are then used in turn to construct the column of the matrix.

```
product_load_combination

do for all factories
      do for all depots
          feasible_routes
          generate_load_combinations(starting product)
      end for all
end for all
    .
    .
    .
generate_load_combinations
do for all possible products
      see_if_product_fits(possible_products, productmix,widths,lengths)
      if (fits) then
          update (productmix, route_costs)
          generate_routes_for_feasible_products (this product)
      endif
end for all
```

Figure 4.3.6 *Pseudocode for generating feasible loading schedules*

Consider factory 1 and depot 3 and the index ℓ_{13} representing a particular load combination. The decision variable $x_{\ell_{13}}$ denotes the number of vehicles assigned with this load combination. Consider the coefficients of this variable appearing in the objective function (4.2.1) and the restrictions (4.2.2) - (4.2.5). The coefficients of the column $x_{\ell_{13}}$, (ℓ_{13}=7) are computed using loading strategy given in row 7 of Table 4.3.5. The actual structure of this column is set out in figure 4.2.7.

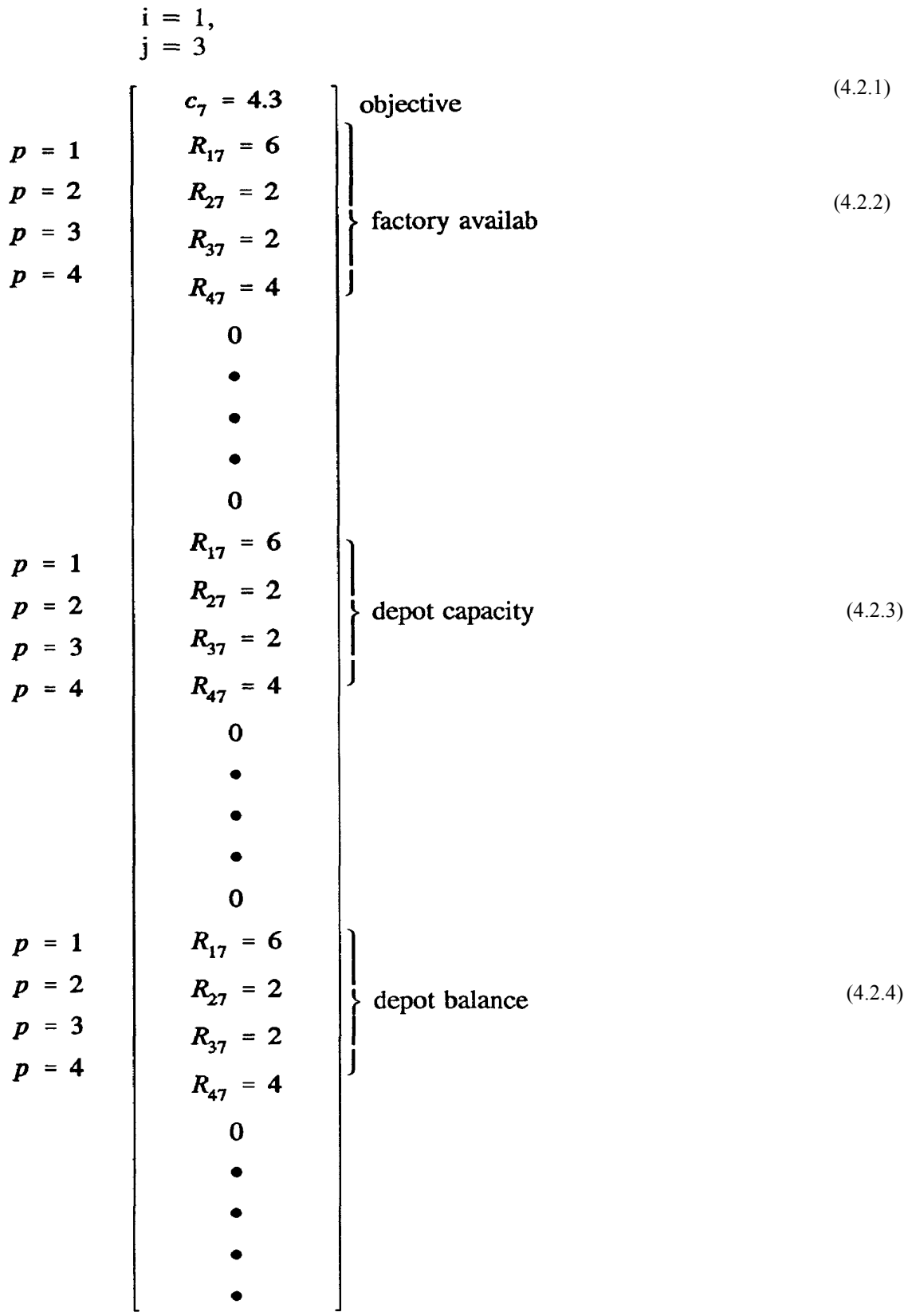

Figure 4.3.7 column structure for $x_{\ell 13} = x_7$

4.4 MPL extended syntax for column generation

Most modelling languages are declarative and are unable to handle procedural features such as the column generation described in the previous section. In many real applications the coefficients are of a dynamic nature and are only known when the data values in the data tables are supplied and the rule base is interpreted using the rule parameters.

The MPL syntax is therefore extended to include dynamic structures by introducing objects. Objects typically have attributes such as input and output data as well as methods for returning the requested information. In figure 4.4.1 the example described in the previous section is shown in MPL using this extended syntax. It is assumed as in the earlier example that all the data is held in a database. The extra index set products is introduced as this is a multi-commodity problem.

A new section 'OBJECT' is introduced in MPL. This has the effect of (i) declaring an object to the MPL system; and (ii) allowing subsequent communication with the declared object via appropriate methods. An object is made up of 3 sections: namely *data* which connects the object to data tables in the database; *methods* which provides a mechanism for communicating required items of information to and from the object; and *procedures* which provide a set of interpretative rules and instructions for computing the method values.

In the illlustrative MPL example shown in figure 4.4.1, annotation (1), the object PRODCOMBO, is declared and is followed by calls to initialization and generation methods (PRODCOMBO.init, PRODCOMBO.generate).

An additional index section, annotation (2), defines dynamic index sets. These sets are dynamic in the sense that their members are unknown until the necessary procedures have been executed. The syntax for defining a dynamic index by calling a method within an object is as follows:

INDEX

setname [set 1, set 2….] FROM *object name.method name;*

where *set 1, set2*,…are previously defined index sets and *object_name* and *method_name* refer to a previously defined object and one of its methods. In the statement

fdloads FROM PRODCOMBO.fd_pos_ind;

the index set fdloads represents the set of all permissible load combinations that could occur on the routes between factories and depots and at this stage is independent of the routes. fcloads and dcloads are similar sets of load combinations for factory-customer and depot-customer routes. fdloadroutes is a two-dimensional index set which connects a particular load combination with the routes it applies to. This index set is also dynamic since the necessary information is determined by the object PRODCOMBO.

In the DATA section dynamic tables are created using a similar syntax and the same keyword FROM. The entries for these dynamic tables are also determined by the object PRODCOMBO. In annotation (3), the statement
"FDprodload [fdloadroutes,products]FROM PRODCOMBO.fd_prodload_table"
determines the entries for the data table FDprodload which represents the amount of a given product in a particular load combination on a specified route between a factory and a depot. The tables FDcost, FCcost and DCcost are also dynamic as their dimensions are defined by the dynamic sets previously introduced.

Annotation (4), defines decision variables which use these previously discussed dynamic sets. The SUM operator is extended to cycle over these dynamic index sets. For example, annotation (5a) and (5b) illustrate the objective function (4.2.1) and the factory availability constraint (4.2.2) specified in MPL.

```
TITLE
    distribution

INDEX
    factories                       := DATABASE("factcap", "Factory Name");
    depots                          := DATABASE("depcap",  "Depot Name");
    customers                       := DATABASE("custreq", "Customer ID");
    FDRoutes[factories,depots]      := DATABASE("fdrout");
    FCRoutes[factories,cutomers]    := DATABASE("fcrout");
    DCRoutes[depots,customers]      := DATABASE("dcrout");
    Products                        := DATABASE("factcap", "Products");

OBJECT                                                          |
    PRODCOMBO;                                                  |      (1)
    PRODCOMBO.init;                                             |
    PRODCOMBO.generate;                                         |
```

Figure 4.4.1 *Distribution example in MPL with column generation*

```
INDEX
!    dynamic index sets (Lij, Mik, Njk, ℓij, mik and njk)
!    ---------------------
  fdloads FROM PRODCOMBO.fd_pos_ind;                                    (2)
  fcloads FROM PRODCOMBO.fc_pos_ind;
  dcloads FROM PRODCOMBO.dc_pos_ind;
  fdloadroutes[fdloads,fdroutes]  FROM PRODCOMBO.fd_load_r_ind;
  fcloadroutes[fcloads,fcroutes]  FROM PRODCOMBO.fc_load_r_ind;
  dcloadroutes[dcloads,dcroutes] FROM PRODCOMBO.dc_load_r_ind;

DATA
   FactoryCap[factories,products]   :=DATABASE("factcap","Capacity");
   DepotCap[depots,products]        :=DATABASE("depcap","Max Throughput");
   CustomerReq[customers,products]:=DATABASE("custreq","Monthly Requirements");
   StorageCost[depots,products]     :=DATABASE("depcap", "Storage Cost");

!    dynamic data tables  (cℓij , c'mik , c''njk , Rplij , R'pmik , R''pnjk )
! ------------------------
 FDprodload[products,fdloadroutes] FROM RODCOMBO.fd_prodload_table
 FCprodload[products,fcloadroutes] FROM PRODCOMBO.fc_prodload_table
 DCprodload[products,dcloadroutes] FROM RODCOMBO.dc_prodload_table
                                                                       (3)
 FDCost[fdloadroutes] FROM PRODCOMBO.fd-cost-table;
 FCCost[fcloadroutes] FROM PRODCOMBO.fc_cost-table;
 DCCost[dcloadroutes] FROM PRODCOMBO.fc_cost-table;;

DECISION VARIABLES
 FactoryDepotQty[products,fdloadroutes]→ FD WHERE (FDCost);
 FactoryCustQty[products,fcloadroutes]  → FC WHERE (FCCost);          (4)
 DepotCustQty[products,dcloadroutes]    → DC WHERE  (DCCost);
 DepotStore[products,depots]            → DSt;

MODEL
      MIN cost =
      SUM(fdloadroutes.fdroutes.factories,fdloadroutes.fdroutes.depots:
                FDCost * FactoryDepotQty) +
      SUM(fcloadroutes.fcroutes.factories,fcloadroutes.fcroutes.customers:   (5a)
      FCCost * FactoryCustQty) +
      SUM(dcloadroutes.dcroutes.depots, dcloadroutes.dcroutes.customers:
                DCCost * DepotCustQty);
SUBJECT TO
      FactoryCapacity[products,factories]  → FCAP :                    (5b)
      SUM(fdloadroutes.fdroutes.depots: FactoryDepotQty)
     +SUM(fcloadroutes.fcroutes.customers:FactoryCustQty)

    <= FactoryCap;

    DepotCapacity[products,depots]  →  DCAP :
    SUM(fdloadroutes.fdroutes.factories: FactoryDepotQty)   <=  DepotCap;

    DepotBalance[products,depots] → DEAL :

    SUM(dcloadroutes.dcroutes.customers: DepotCustQty)
 =SUM(fdloadroutes.fdroutes.factories: FactoryDepotQty);
```

Figure 4.4.1 continued Distribution *example in MPL with column generation*

```
        CustomerRequirement[products,customers] → CREQ :

        SUM(fcloadroutes.fcroutes.factories: FactoryCustQty)
       +SUM(dcloadroutes.dcroutes.depots: DepotCustQty)

        = CustomerReq;
END
```

Figure 4.4.1 continued *Distribution example in MPL with column generation*

	PRODCOMBO
OBJECT	
DATA	factories depots customers products costs lengths widths . . .
METHODS	init ! initializes generate ! executes the procedure fd_pos_ind ! returns the factory_depot position index fc_pos_ind . dc_pos_ind . fd_load_r_ind . fc_load_r_ind dc_load_r_ind . . .
PROCEDURES	Product_load_combination do for all factories do for all depots feasible_products generate_load_combinations end for all end for all . . .

Figure 4.4.2 *Object PRODCOMBO*

5. Discussion

Connecting a relational database to a modelling language such as MPL is important for creating a rich LP modelling environment in which a generic model can be made truly data driven. This is achieved by importing the keys and table entries of relational tables as dynamic data items. We have extended the syntax of MPL and shown how this connection is made.

For a long time column generation has been a neglected aspect within LP modelling systems. This is because it is difficult to integrate procedural knowledge within an otherwise declarative knowledge representation paradigm. Traditionally column generation was carried out within the solver environment, but this came at a cost of sacrificing model and solver independence.

Creating optimization applications by exploiting robust commercial optimizers such as FortMP, CPLEX, OSL, see (*Sharda, 1993*), has now become established industrial practice: so the introduction of column generation functionality within the modelling language is much needed. VLSI routing, cutting stock, vehicle and crew scheduling, contract selection are a few examples of a wide range of optimization applications which call for modelling by column generation. Indeed in these examples the most challenging task of conceptualization and knowledge representation consists of identifying the rules and procedures governing the creation of the activities or the columns.

In the simplest form these column generation procedures can be carried out within the database language and the range of indices and tables generated in this way can then be communicated to the modelling language. The key element in this context is to understand the dynamic nature of the column that we are generating by interpreting the procedural rules. By introducing objects, procedures and methods and providing a syntactic structure for connecting them to LP modelling languages we have taken this concept one step further and we are able to bring this genre of models to the modeller who is already familiar with declarative modelling languages such as MPL.

6. Acknowledgements

The support of the UK Science and Engineering Research Council (SERC) is gratefully acknowledged, who together with Numerical Algorithms Group (NAG) have supported Ms. S. Moody's CASE studentship. We also thank Mr. B. Kristjansson of Maximal Software Ltd for his advice and very enthusiastic collaboration with our research group.

7. References

Ashford, R.W. & Daniel, R.C. (1987) LP-MODEL: XPRESS_LP's Model Builder, *IMA Journal of Mathematics in Management* 1, 163-176

Baker, T.E. (1993), Graph based modelling with MIMI/G presented at APMOD93, Budapest, Hungary

Bisschop, J.J. (1993), AIMMS Modelling System User Guide, prepared by Paragon Decision Technology B.V., Haarlem, Netherlands.

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

Butler M., R. Bloor & P. Beach, (1990), *Database: An Evaluation and Comparison,* Butler Bloor

Chambers, M.L. & Dyson, R.G. (1976), The Cutting Stock Problem in the flat glass industry - selection of stock sizes *Operational Research Quarterly*, Vol.27 pp 949-957

Christofides N., Mingozzi, A. & Toth, P. (1979) The Vehicle Routing Problem in *Combinatorial Optimization*, Wiley, London.

Darby_Dowman, K. & G. MItra, (1985) An Extension of Set Partitioning with Application to Scheduling Problems, *European Journal of Operational Research* 21 1985 200-205

Date, C.J., (1981), *An Introduction to Database Systems*, 3rd edition, Addison-Wesley

Fischer, M. & Jaikumar, R. (1981), A Generalized Heuristic for Vehicle Routing, *Networks*, 11, p109.

Fourer, R. (1983), Modelling Languages versus Matrix Generators for Linear Programming *ACM Transactions on Mathematical Software*, Vol 9 No. 2 June 1983 pp143-183

Fourer,R. D.M Gay,& B.W. Kernighan (1987) AMPL: A Mathematical Programming Language Computing Science Technical Report No. 133, January 1987 AT&T Bell Laboratories

Geoffrion A.M. (1992a), The SML Language for Structured Modeling: Levels 1 and 2 *Operations Research* 40:1 38-57

Geoffrion A.M.(1992b), The SML Language for Structured Modeling: Levels 3 and 4 *Operations Research* 40:1 58-75

Gilmore, P.C. & Gomory, R.E. (1965), Multi-stage Cutting Stock Problems of Two or More Dimensions Operations Research, Vol.13 pp 94-120.

Greenberg, H.J.(1991a) A Comparison of Mathematical Programming Modelling Systems March 1991 University of Colorado at Denver

Greenberg, H.J.(1991b) A Primer for MODLER : Modeling by Object Driven Linear Elemental Relations December 1991 University of Colorado at Denver

Hurlimann, T.& Kohlas, J. (1987), LPL Structured Language for Linear Programming Modelling, 10 55-63.

IBM World Trade Corporation (1976) IBM Mathematical Programming System Extended/370 (MPSX/370) Program Reference Manual, 2nd edn, *IBM Publication No. SH19-1095-1* New York and Paris

Kristjansson, B. (1991) MPL Modelling System User Guide, Maximal Software Ltd, Iceland

Kristjansson, B. (1993) MPL Modelling System Release 3.0, Maximal Software Ltd., Iceland

Levkovitz,R.& Mitra, G. (1993) Solution of Large Scale Linear Programs: *A Review of Hardware, Software and Algorithmic Issues in Optimization in Industry*, edited by T.A. Ciriani & R.C. Leachman, John Wiley & Sons, UK.

Lucas C. & G. Mitra (1988) Computer-Assisted Mathematical Programming (Modelling) System: CAMPS *The Computer Journal* Vol 31. No. 4.

Lucas, C., Mitra, G., Moody, S. & Kristjansson, B. (1993), Sets and Indices in Linear Programming Modelling and their Integration with Relational Data Models TR/02/93, Brunel University, Dept of Mathematics and Statistics, Uxbridge, UK.

Maros,I., Mitra, G. & Moody, S. (1993), Contract Portfolio Selection for an Electricity Company, Technical Report in preparation at Brunel University, Uxbridge, UK.

Mitra, G. C.Lucas, K. Darby-Dowman & J.Smith (1994), Maritime Scheduling using Discrete Optimization and Artificial Intelligence Techniques in *Practical Applications of Optimization,* edited by A. Sciomachen, John Wiley & Sons

Sharda, R. (1993), Linear & Discrete Optimization and Modeling Software, Unicom (UK) & Lionheart Publishing Inc. (USA)

Steiger D. & R. Sharda (1993) LP Modeling Languages for Personal Computers: A Comparison, Applied Mathematical Programming and Modelling, edited by G. Mitra & I. Maros *Annals* of Operations Research, 43 1993, 195-216

Ullman, J.D., 1982, Principles of Database Systems, Computer Science Press

Williams, H.P. Model Building in Mathematical Programming, 3rd edition, John Wiley 1990