TR/02/93 May, 1993

Sets and Indices in Linear Programming Modelling and their Integration with Relational Data Models

Bjarni Kristjansson*, Cormac Lucas**,
Gautam Mitra** and Shirley Moody**

* Maximal Software, Iceland
** Brunel University, Uxbridge, Middx, UK

w9258378

Sets and Indices in Linear Programming Modelling and their Integration with Relational Data Models

Bjarni Kristjansson*, Cormac Lucas**, Gautam Mitra** and Shirley Moody**

* Maximal Software, Iceland
** Brunel University, Uxbridge, Middx. UK

Abstract

LP models are usually constructed using index sets and data tables which are closely related to the attributes and relations of relational database (RDB) systems. We extend the syntax of MPL, an existing LP modelling language, in order to connect it to a given RDB system. This approach reuses existing modelling and database software, provides a rich modelling environment and achieves model and data independence. This integrated software enables Mathematical Programming to be widely used as a decision support tool by unlocking the data residing in corporate databases.

Keywords
Sets, Indices, Linear Programming Modelling, Relational Databases, Modelling Languages

Contents

1. Introduction

In the last four decades, Linear Programming (LP) techniques have played an important role in solving numerous problems of decision-making in a variety of industrial applications. During the last decade, rapid growth in the capability and availability of optimization software has enabled large problems to be solved in a realistic time scale on an affordable computer. Consequently, computer support for the formulation and analysis of LP problems has received increasing attention.

Many modelling systems exist today which enable the user to specify the model in a modelling language which is concise, descriptive, self documenting and comprehensible. For example, AMPL *(Fourer, Gay & Kernighan, 1987)*, GAMS *(Brooke, Kendrick & Meeraus, 1988)*, LPL *(Harlimann, 1992)*, LP-MODEL *(Dash Associates, 1992)*, MathPro *(MathPro, 1992)*, MPL *(Kristjansson, 1993)*, MODLER *(Greenberg, 1992)*, SML *(Geoffrion, 1991)* and there are many others. Thus the arduous task of representing the problem in an appropriate input form for the solver is fully automated within these systems. For a review of these see *(Sharda, 1992)*, *(Greenberg, 1991)*. The specification of a model makes use of index sets which define the dimensions of the variables, data tables and constraints. Most modelling languages employ data tables which are either constructed within the model or are created externally (separate from the model) by means of ascii 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.

In an organizational setting, the problem owner's view of his or her problem relates directly to the corporate information system (see diagram 1). The analyst's model and its solution provides the problem owner with the necessary decisions which may be operational, strategic or long term.

For the purposes of investigation or as a result of changes in the decision making environment, the data within the corporate information system may be regularly revised. It is therefore desirable that the decision making system should automatically update the model when such changes are made.

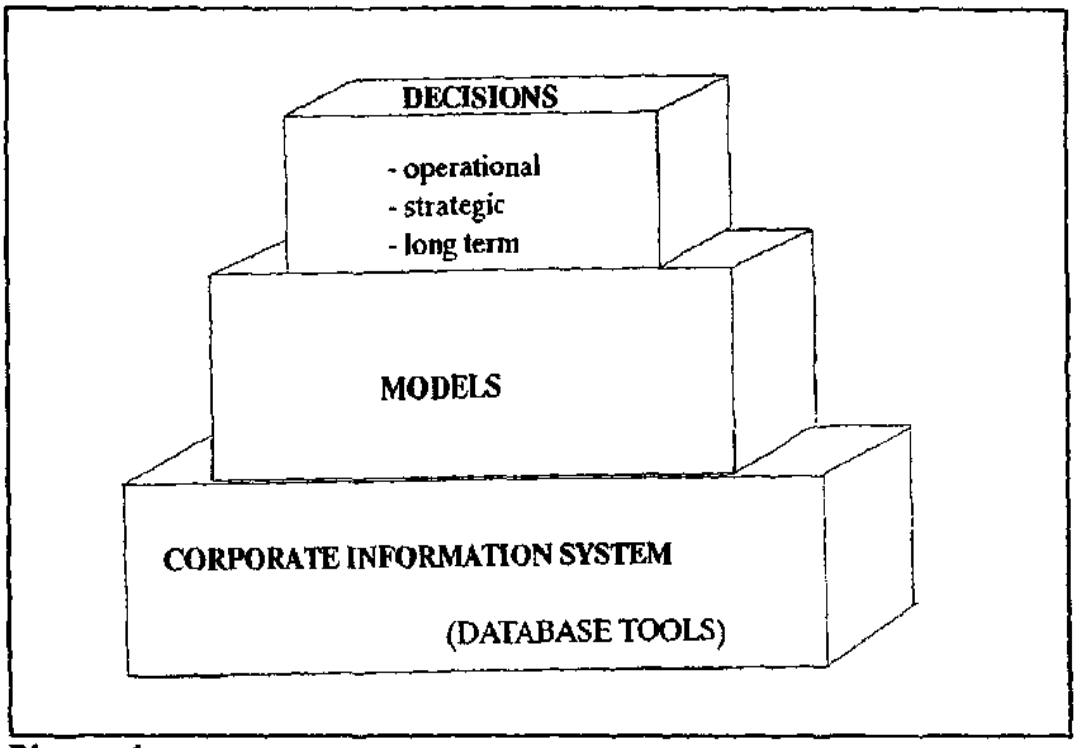

Diagram 1

Diagram 1

This perspective is consistent with the view put forward by Hurlimann who argues that despite recent developments, Mathematical Programming is still not fully exploited in practice, as much work is needed to get the model set up and data defined *(Hurlimann, 1991,*

p3). It is standard practice in industry to utilise databases for information storage and retrieval and much of the data needed for the problem is already present within the database. Thus, in order to gain better acceptance of Mathematical Programming as a modelling tool within corporate decision support studies, a unified approach which integrates an LP modelling language with a relational database is necessary. This will provide a more powerful tool for constructing models which are truly data driven.

By our approach it is possible to connect corporate databases with a modelling system, MPL *(Kristjansson,1993)*, which possesses a rich environment for model creation and documentation. This is achieved by incorporating RDB structures into the syntax of MPL, thus reusing the existing database and modelling software.

This paper is intended for the LP modelling and information systems specialists and therefore provides overviews of both modelling and relational database systems. Thus the paper is organised as follows: the basic structure and syntax of MPL is introduced in section two and some well known but necessary background to relational data models is described in section three. An illustrative example is provided in section four demonstrating the need for new constructs to be introduced in modelling languages such as MPL to permit the integration of relational and LP models. MPL is then extended in section five to incorporate such constructs and some brief discussion draws this work to a conclusion.

2. The Structure of an LP Modelling Language: MPL

The modelling of LP problems involves 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 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 items. *(Lucas & Mitra, 1988, p365).*

MPL (or Mathematical Programming Language) is a modelling system which enables problems to be formulated in a declarative form and an input file for an optimiser (eg MPS file) to be generated. The key features of MPL are described in the MPL Users Guide *(Kristjansson, 1991, p1.2 -1.4).*

The structure of MPL reflects this modelling methodology and 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.
END		

Apart from the objective function (distinguished by the keyword MAX or MIN) and the

constraints (identified by the keywords subject to), all sections are optional.

To illustrate the basic syntax of the language consider this small planning example. A company which manufactures three products (product) needs to decide on a monthly basis what quantities of each product it should produce (Production [product, month]), what quantities it should store (inventory[product,month]) and what quantities it should sell (sales [product, month]) in order to maximize profit. Profit is simply considered to be the Revenues from sales minus the Totalcost (total production cost + total storage cost). In addition there are various restrictions: there is a limit to the amount that can be stored each month; the amount produced each month cannot exceed the production capacity; the quantities sold should not exceed the demand for these products (shown in bounds section). Finally, a balancing constraint specifies that the amount in storage in any particular month must be equal to the amount that was in storage the previous month plus the amount produced minus the amount Sold, that is (InventoryBalance[product ,month]). This planning model is stated in MPL as follows:

```
        {       Planning. mpl       }
        {       Aggregate production planning for 12 months       }
TITLE
        Production_Planning;
INDEX
        Product   =  1. .3;
        month     = (January, February, March, April, May, June, July,
                     August, September, October, November, December);
DATA
        price[product]                          : =   (105.09, 234.00, 800.00);
        Demand[month, product]                  : =   DATAFILE(Demand.dat) ;
        ProductionCapacity[product)             : =   1000 (10, 42, 14) ;
        ProductionCost [product]                : =   (64.30, 188.10, 653.20);
        InventoryCost                           : =   ?;
```

```
DECISION
        Inventory [product, month]              ->    Invt
        Production [product, month]             ->    Prod
        Sales [product, month]                  ->    Sale

MACRO
        Revenues        : =   SUM (product, month: price * Sales);
        Total Cost      : =   SUM (product, month: Production Cost * Production
                                        + Inventory Cost * Inventory);
MODEL

        MAX   Profit  =  Revenues – TotalCost;

SUBJECT TO
        Inventory Balance [product, month]              ->  IBal :
                Inventory   =  Inventory[month-1] + Production - Sales;

BOUNDS
        Sales    <   Demand ;
        Production       <   Production Capacity  ;
        Inventory [month<December]    <    90000 ;
        Inventory [month=December]    =    20000 ;
END
```

(op. cit., p 6-3)

Throughout the MPL model meaningful names of any length may be used to aid documentation. There are two index sets in this model, namely product (which may take values 1,2 or 3) and month. All subsequent vectors are defined in terms of these index sets. The data for this problem may be entered in the model definition file, as in the data table price [product], or may be stored in an external file and referred to by the keyword DATAFILE, as in the two-dimensional data table Demand [month, product]. Data constants may also be entered at runtime, hence the question mark in the scalar Inventory Cost.

The two-dimensional decision variables are provided with stubs (for example "-> Invt") which are abbreviations to enable distinct variable names as required by the MPS naming convention: an input file format necessary for the optimiser. The macro section defines two quantities which are used in the objective function. This enhances the readability of the model and encourages meaningful documentation. The keyword sum defines a summation to be carried out over the indices specified in the ensuing brackets. The four mathematical

operators, addition, subtraction, multiplication and division are represented by the usual symbols: + .-.*, /, respectively.

The constraint inventory Balance [product, month] defines 36 constraints (one for each combination of product and month). The index month-1 is used to reference the previous month. Thus this constraint states that for any particular product and month, the inventory must be equal to the previous month's inventory, plus the amount produced this month, minus the amount sold this month of this product.

Having defined the model in this manner using MPL's editor, the analyst selects a 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

In a relational data model there is only one construct visible to the user: the relation or table. These relations (or tables) represent the basic entities of a system. The basic properties or characteristics of these entities are known as attributes and these form the columns of the table. For example, consider a power station. This entity may have several attributes such as the station name, the type of fuel it requires, the (heat) energy required, minimum or maximum accepted levels for sulphur, chlorine and ash, an indicator flue gas desulphurisation (FGD) unit. The table PowStat provides this information for several power stations.

PowStat ***(Power Station)***

Station Name	Fuel Type Required	Heat Requirement	FGD Presence	Min Sulphur	Max Chlorine	Min Ash
Stat1	1	270	1	0.2	0.2	0.3
Stat2	2	240	0.5	0.1	0.09	0.2
Stat3	1	350	0	0.1	0.1	0.1
.	.	.	.	.	.	.
.	.	.	.	.	.	.
.	.	.	.	.	.	.
Stat9	2	100	0.5	0.2	0.08	0.3
Stat10	1	600	1	0.1	0.1	0.25

This table provides information about power stations. The columns of this table are the attributes of this relation and the rows are known as tuples. Thus, each tuple represents an entity, namely a power station and the attributes of this table represent a relationship between these entities (hence "relation" being synonymous with "table" in this context). Accordingly, a relation may be defined as a set of tuples.

Each attribute has an associated domain from which values may be drawn for this attribute. In the above example, suppose that there are only four types of fuel available (1,2,3 and 4). Then the attribute Fuel Type Required may only take values 1, 2, 3 or 4. Thus a domain is a valid set of values for an attribute in a table. Further, "each occurrence of an attribute must

contain a single value, and cannot contain repeating groups or any other nested data structures." (Butler et al, 1990, p120)

Within a relational table there is often one attribute (or a combination of attributes) which can be used to uniquely identify the tuples (rows) of that table. These are called keys. These reflect the functional dependencies of the relation: that is, each attribute value depends on the value of the key attribute. As there may be more than one possible candidate or candidates for the key of a relation, it is common to designate one key or combination of keys as the primary key. If a key consists of more than one attribute it is known as a composite key.

In addition to these basic data structures, the relational data model has three associated rules for maintaining data integrity. Firstly the entity integrity rule requires that there exists a unique method of identifying each tuple. This is normally achieved by the definition of a primary key where no component may be null *(Date, 1981, p89).* The second rule is concerned with domain integrity. This stipulates that all attributes may only take legitimate values from the declared domain. Thirdly, the rule of referential integrity which ensures that if a relation references another via a particular set of attributes then all values of this attribute must exist in the relation being referred to. An attribute which is the primary key of another relation is known as a foreign key. Such keys represent references which enable tables to be combined with each other usually by joins (discussed later).

Software that enables a user to use and modify information stored in a database is called the database management system. This system allows the user to view the data in abstract terms

rather than look at how the data is actually stored. The end user has a language at his or her disposal which includes a data sublanguage. This subset of the language is concerned with both database objects and operations and therefore can be viewed as two languages: a data definition language which enables the construction of the database objects (tables) and a query language or data manipulation language which supports die manipulation and processing of these objects enabling queries to be formulated. One of the most widely accepted of these query languages is SQL (Structured Query Language) which has become the industrial standard query language for many database systems. SQL is both a data definition language and a query language. It combines both the algebraic type of query language, where queries are expressed by applying specialized operators to relations, and the predicate calculus type of language where queries describe a desired set of tuples by specifying a predicate that the tuples must satisfy *(Ullman, 1982, pl51).*

As there are many RDB systems which are implemented to acceptable industrial standards, there is no advantage in recreating another query language. In order to integrate LP modelling languages with such relational databases it is necessary to use some of the basic operations defined in relational algebra, that is, the mathematical set theory operators and the relational operators. The traditional mathematical operators (union, intersection, difference, cartesian product) are well known and apply to the relational model because the relation is defined to be a set of tuples. Consequently, set operators are used whenever relations are processed. The relational operators (selection, projection and join) are illustrated below.

The select operator "selects" a horizontal subset of the tuples in a relation. For example, in order to obtain all power stations which required fuel type 1 from the relation PowStat, a

selection of all tuples (rows) for which the attribute Fuel Type Required has value 1 must be carried out. This results in the table StFuell.

StFuell ***(Stations with Fuel Type 1)***

Station Name	Fuel Type Required	Heat Requirement	FGD Presence	Min Sulphur	Max Chlorine	Min Ash
Stat1	1	300	1	0.2	0.1	0.3
Stat3	1	350	0	0.1	0.1	0.1
.	.	.	.	.	.	.
.	.	.	.	.	.	.
.	.	.	.	.	.	.
Stat10	1	600	1	0.1	0.1	0.25

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 consider the table PoStOwn. This has the attributes company and station name and represents the power stations that are owned by particular companies.

PoStOwn *(Power Stations owned by companies)*

Station Name	Company
Stat1	IndepCo
Stat2	IndepCo
Stat3	IndepCo
.	.
.	.
.	.
Stat10	PublicCo

Suppose it is required to obtain a set of all the companies that own a station, then a project operation over the attribute company would achieve this resulting in the table Comp in which all duplicates have been eliminated.

Comp *(Company)*

Company
IndepCo
PublicCo

This ensures that the result of a project operation is also a relation (ie. a set of tuples). 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, consider the table FeasRout.

FeasRout *(Feasible Routes)*

Route Number	Fuel Site Name	Station Name	Route Cost
1	1	start1	100
2	1	start3	150
.	.	.	.
.	.	.	.
.	.	.	.
18	7	start2	120
19	7	start5	250
20	7	Start1	350

This table shows the feasible routes from fuel sites to power stations and the unit cost of transporting fuel along these routes. If it were necessary to know which companies owned which routes, a join of FeasRout with PoStOwn would be required such that the Station Name in the FeasRout relation matched the Station Name in the PowStat table. This would result in the wider table CoFeRo (company feasible routes).

CoFeRo *(Company Feasible Routes)*

Route Number	Fuel Site Name	Station Name	Route Cost	Station Name	Company
1	1	start1	100	start1	IndepCo
2	1	start3	150	start3	IndepCo
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
18	7	start2	120	start2	IndepCo
19	7	start5	250	start5	IndepCo
20	7	Start10	350	Start10	PublicCo

Thus this join operation is equivalent to performing a cartesian product of the tuples of the relation FeasRout with the tuples of the relation PoStOwn 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 repetition of the attributes station name as these were the join attributes. As one of these is superfluous, a join operator which removes the second attribute in an equijoin condition is used. This 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. There are other operators such as

insert, delete and update. These are necessary for maintaining a database, but are operations which should be performed by the user who has overall control of the database system and such operations are not required for the LP modeller. The structure of the database however is important. In particular the normalisation of the database which ensures that attributes are associated with the correct table *(Date, 1981, p237f).* Throughout the examples provided in this paper it is assumed that the data is normalised.

4. Illustrative Example

4.1 Problem Definition

Consider the problem faced by power companies allocating fuel to power stations where the objective is to minimize the total cost of fuel allocation subject to certain restrictions imposed on the power companies and the power stations. For example, suppose there are two companies (called IndepCo and PublicCo) each owning several power stations which require various types of fuel from seven accessible fuel sites. Diagram 4.1 depicts the fuel sites, power stations, companies which own the power stations and the viable routes between sites and stations.

The problem is to decide how much fuel must be shipped along each viable route so as to minimize the total cost. There are of course constraints which must also be satisfied: firstly, each power station has certain energy (heat) requirements and restrictions for sulphur, chlorine and ash levels; secondly there are environmental regulations concerning SO_2 levels which restrict the maximum output levels for SO_2 for each company. In addition each power company is restricted to take (if any) a minimum quantity of each fuel type (these are measures imposed by the various fuel companies); there are also capacity limits on the

available fuel supply at each site.

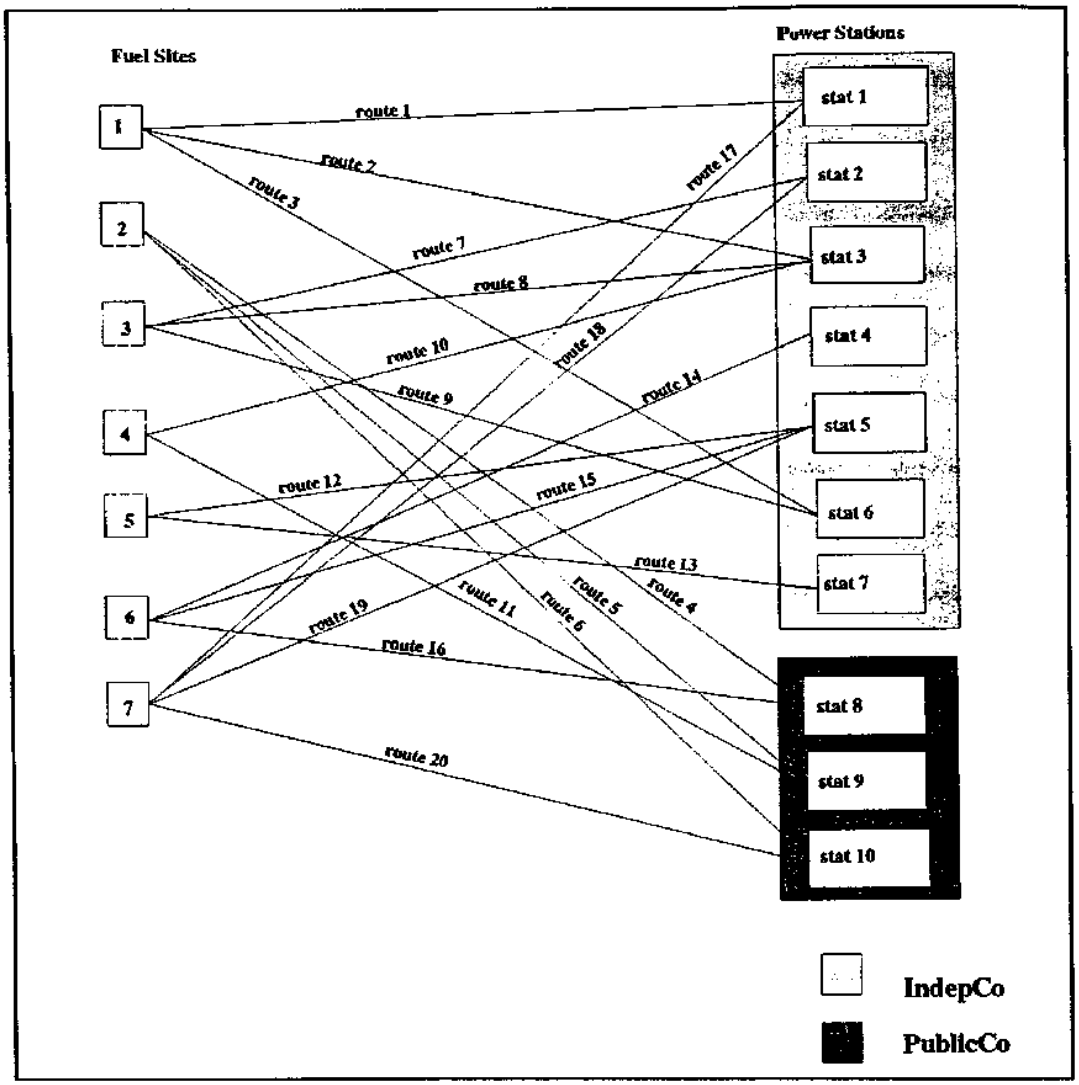

Diagram 4.1

4.2 Data Model

The information required for this model is stored in an existing database which consists of six relations. These relations may be divided up into four main groups: concerning the power companies, the power stations, the fuel sources (sites) and the routes. This is reflected in the respective relation (or table) names "PoCo..", "PoSt..", "FuelSrc" and "Routes". The

following tables illustrate the relations and provide typical attribute values. The attributes underlined are the primary keys. The domains of all the attributes are real numbers or integers and in order to keep the description of this example simple, units have been omitted.

The relation PoCoSO2 contains the attributes Company and Max SO2. Max SO2 is dependent on the company which is the key for this table. The tuples represent the maximum SO_2 output level permitted for the company. The second power company relation PoCoTak has attributes Company, Fuel Type and Min Take. MinTake is dependent on Company and Fuel Type so these attributes form a composite key for this relation. The tuples here represent the minimum order quantity of each company for the different fuel types.

The relation PoStOwn contains the attributes Company and Station Name where tuples indicate the owner (company) of a particular power stations. In this relation Company is dependent on the attribute Station Name and thus the attribute Station Name is the key. The remaining information regarding the power stations are contained in the relation PowStat. This has attributes: Station Name, Fuel Type Required, Heat Requirement, FGD Measure, Min Sulphur, Max Chlorine, Min Ash. Each power station has a requirement for a particular fuel type (or types). In addition there is a heat requirement for each power station as well as minimum or maximum accepted levels for sulphur, chlorine and ash. All these attributes are dependent upon the Station Name which provides the key. The FGD Presence indicates the reduction in SO_2 output due to the existence of a flue gas desulphurisation unit.

The unit heat generated from a certain fuel is calculated from the calorific percent values of each fuel type which are held in the table FuelSrc. Similarly the percentages of sulphur,

chlorine and ash for fuel taken from a given site are contained in FuelSrc. This relation has the attributes Fuel Site Name, Fuel Type Available, Price, Calorific Percent, Capacity, Sulphur Percent, Chlorine Percent and Ash Percent representing: the fuel site names, the type of fuel available at these sites, the unit price for the fuel, the various levels previously mentioned and the capacity of each site. This capacity is a maximum limit on the amount that can be shipped from this site. The key therefore in this relation is the Fuel Site Name as all attributes depend on the site.

The final relation Route indicate routes that are available from a fuel site to a power station. It shows the unit cost (Route Cost) of transporting fuel along each route as well as the maximum and minimum capacities of these routes (Minimum Capacity on Route and Maximum Capacity on Route). These attributes are all dependent upon the particular route, ie., Route Number. There are also some columns contained in this relation whose data is refreshed after optimisation. These are for the amount shipped along each route and consequently the cost of that shipment. The values of these attributes are at present unknown, but will become available after the optimization takes place.

The Company Database

PoCoSO2 ***(Power Company SO_2)***

Company Name	Max SO2
IndepCo	1 000
PublicCo	2 000

PoCoTak ***(Power Company Take)***

Company Name	Fuel type	Min Take
IndepCo	1	50 000
IndepCo	2	100 000
IndepCo	3	20 000
IndepCo	4	10 000
PublicCo	1	75 000
PublicCo	2	200 000
PublicCo	3	20 000
PublicCo	4	100 000

PoStOwn ***(Power Station Owners-)***

Company Name	Station Name
IndepCo	start1
IndepCo	start2
IndepCo	start3
.	.
.	.
.	.
PublicCo	start8
PublicCo	start9
PublicCo	Start1

PowStat *(Power Station)*

Station Name	Fuel Type Required	Heat Requirement	FGD Presence	Min Sulpher	Max Chlorine	Min Ash
start1	1	270	1	0.2	0.2	0.3
start2	2	240	0.5	0.1	0.09	0.2
start3	1	350	0	0.1	0.1	0.1
.	.	.	.	.	.	.
.	.	.	.	.	.	.
.	.	.	.	.	.	.
start9	2	100	0.5	0.2	0.08	0.3
start10	1	600	1	0.1	0.1	0.25

FuelSrc ***(Fuel Source)***

Fuel Site Name	Fuel Type Available	Price	Calorific Percent	Capacity	Sulphur Percent	Chlorine Percent	Ash Percent
1	2	20	0.67	1 000 00	0.1	0.32	0.1
2	2	30	0.89	250 000	0.2	0.45	0.5
3	1	35	0.75	350 000	0.25	0.6	0.23
4	3	25	0.50	500 000	0.4	0.26	0.5
5	4	20	0.70	600 000	0.62	0.32	0.61
6	3	35	0.60	1 000 000	0.35	0.45	0.41
7	1	30	0.85	1 250 000	0.05	0.25	0.21

Routes

Route Number	Fuel Site Name	Station Name	Route Cost	Minimum Capacity on route	Minimum Capacity on route	Amount Shipped	Total Route cost
1	1	start1	100	1 000	5 000		
2	1	star3	150	2 000	10 000		
.	.	.	.	.	.		
.	.	.	.	.	.		
.	.	.	.	.	.		
18	7	start2	120	3 500	7 500		
19	7	star5	250	1 000	5 000		
20	7	star10	350	2 800	10 000		

4.3 LP Model

The algebraic representation of this LP model first involves defining sets and subsets. For example, let company denote the set of power companies in this example and station the set of power stations and let i $\in$ company and j $\in$ station be the indices that are used to reference the elements of these sets. A subset J; may be defined to represent the stations that belong to a particular company i. The union of all these sets defines a set, J which represents all the owned power stations. This is represented in MPL by a two-dimensional set stat ionowners [company, stat ion]. This structure is discussed further in section five. This

definition allows us to use the corresponding relational data table PoStOwn. Having defined all such sets and subsets the required data for this model is declared as tables. Decision variables are defined in terms of these sets and subsets. The objective function and the constraints are in turn stated as summations using the data tables and decision variables: this follows the progressive (logical) steps for constructing the model (see section two).

The model objective is to minimise the total cost. The cost along a particular route is derived from the cost of the fuel on this route plus the route cost, which is then multiplied by the amount shipped along the route. Therefore the total cost is the summation of costs for the feasible routes, that is,

$$\textbf{Minimise} \sum_{\ell \in \text{feasrout}} \textbf{(fuelprice(}\ell\textbf{) + routecost(}\ell\textbf{)) * amntship(}\ell\textbf{)}$$

where feasrout is the set of all feasible routes, amntship(ℓ) are the decision variables denoting the amount of fuel to be shipped on each feasible route, fuelprice(ℓ) and routecost(ℓ) are data table entries representing the costs.

A typical constraint in this model is the limit on the maximum emissions of SO_2 for each company. This may be written as:

$$\sum_{j \in \text{PostOwn}} \sum_{k \in \text{StRoute}} \textbf{Sulphurconst (1 - station-fgd-percent(j)) * fuel-sul-percent(k) * amntship(k)} \leq \textbf{Max SO2(i)} \quad \textbf{i} \in \textbf{company}$$

This defines sulphur constraints for each power company. Sulphurconst is a standard conversion constant that applies to all power stations; station_fgd_percent(j) indicates the presence of flue gas desulphurisation units for each power station j (j $\in$ PoStOwn - the set

of power stations owned by company i); fuel_sul_percent(k) is the sulphur percentage of fuel shipped on route k(k ∈ StRoute which is the subset of all routes which end at power station j); the Max SO2(i) is the maximum SO_2 output imposed on company i. Therefore, these constraints are constructed by summing over the routes ending at all the power stations which are owned by that particular company.

The full algebraic representation of this model is expressed in MPL and presented in the Appendix together with the new constructs introduced in the following section.

5. Sets and New Database Constructs within MPL

Set operations such as union, intersection and difference have been introduced in MPL. For example, the following index sets may be defined explicitly in MPL:

```
INDEX
    Plants                  :=  (London, Paris, NewYork, Chicago, Madrid);
    Open Plants                 := (London, NewYork, Madrid);
    European Plants [plants]    := (London, Paris Madrid);
```

Further indices may be specified as follows:

```
ClosedPlants [plants]       := plants - OpenPlants;
EuropeanOrOpen [plants]     := EuropeanPlants OR Open Plants;
EuropeanAndOpen[plants]     := EuropeanPlants AND Open Plants;
```

The set "ClosedPlants" uses the difference operator to obtain the resulting set containing Paris and Chicago. The index "EuropeanOrOpentplants]" utilizes the operator OR and performs the union of the two sets "EuropeanPlants" and "OpenPlants". Thus, this represents the set {London,NewYork,Madrid,Paris}. The final index "EuropeanAndOpen[plants]" employs the AND operator to carry out the intersection of two sets to obtain the smaller set containing London and Madrid.

In order to make use of the RDB structure, MPL is extended to incorporate the other operators described in section three. Thus new constructs are introduced to enable:

(i) the specification of 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 demonstrated by considering the Power Model example. For the rest of this section all references to MPL denote "extended MPL".

Index sets may be also derived from the keys of a database table and the syntax for defining index sets incorporates the keyword "Database":

```
INDEX
    Set_name  := Database (" database_table_name",  "key");
```

For example, the following defines the index set "company" in MPL which is the key to the table "PoCoSO2" in the database:

```
INDEX
    company : = Database ("PoCoSO2",     "Company Name");
```

The index section, defines the index set "company" which corresponds to the key "Company Name" taken from the relation "PoCoSO2". Subsequently, "company", rather than "Company 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, as stated in section 4.3, are represented in MPL by two (or more) dimensional sets, that is sets of sets. The syntax for these multi-dimensional sets is as follows:

INDEX

set_name[*setl,set2*...] := Database (*"database_table_name", setl, set2*...); where setl, set2,... are previously defined index sets. For example, the subset J described earlier is defined as follows:

```
INDEX
    StationOwners[company,station]:=
        Database("PoStOwn", company, station);
```

This defines a projection of "Company Name" and "Station Name" taken from the relation "PoStOwn". This index is a multi-dimensional set or, in relational terms, a multi-attribute (or composite) key for the data tables and constraints which are as yet to be defined.

StationOwners

Company	Station
IndepCo	stat1
IndepCo	stat2
IndepCo	stat3
IndepCo	stat4
IndepCo	stat5
IndepCo	stat6
IndepCo	stat7
PublicCo	stat8
PublicCo	stat9
PublicCo	stat10

It is easier to think of this subset as the relational table above rather than as a set of power stations which are owned by company i, say. Another example is as follows:

```
INDEX
    Station         := Database("PowStat", "Station Name");
    route           := Database("Routes", "Route Number");

    StationRoutes[station,route]   :=
                        Database ("Routes", station, route);
```

This multi-dimensional set StationRoutes is defined as a projection of "Station Name" and

"Route Number" taken from "Routes".

Multi-dimensional sets can also be derived from more than one table or subset. This is achieved intoducing the keywords WHERE and IN:

INDEX

set_name[*setl,set2...*] WHERE FORSOME(*set3* IN *subset1*[*set1,set3...*])
= (*set3* IN *subset2*[*set3,set2...*]);

(As set1, set2, set3,..., subset1 and subset2 are previously defined index sets, it is not necessary to repeat the index sets of the subsets as shown above. They are, however, included here for clarity.) For example,

```
INDEX
    CompanyRoutes[company,route] WHERE
        FORSOME(station IN StationOwners[company,station] =
                station IN StationRoutes [station, route]);
```

This is equivalent to performing a (natural) join followed by a projection. The resulting multi-dimensional table is CompanyRoutes. The keyword **"WHERE FORSOME"** indicates that a join condition is about to be specified and the IN operator checks for entries in a relation. For example "station IN StationOwners" selects all values of the attribute station which occur in the relation StationOwners. This "IN" operator is again utilised in the data and constraint sections of the model.

CompanyRoutes

company	route
IndepCo	1
.	.
.	.
.	.
PublicCo	16
PublicCo	5
PublicCo	11
PublicCo	6
PublicCo	20

Data may be imported from a database by using the keyword "IMPORT". That is,

```
IMPORT
    Database ("database_table_name" INDEX set1 , set2 , . .
        DATA data_table[set1,set2...] = "attribute_name"
            data_table[set1 ,set2,..] = "attribute_name"
                                .
                                .
                                .                    );
```

Again setl,set2,.. are previously defined index sets. For example, a one-dimensional data table extracted from a database may be defined as follows

```
    IMPORT
      Database("PowStat" INDEX station
        DATA Fgdpercent[station] = "FGD Presence");
```

This is equivalent to a projection from the table "PowStat" of the attribute "FGD Presence".

As shown in the syntax above, several attributes may be imported at a time in the following way:

```
IMPORT
  Database("FuelSrc" INDEX site,
      DATA   FuelType      = "Fuel Type Available";
             FuelPrice     = "Price";
             FuelSulphur   = "Sulphur Percent";
             FuelCal       = "Calorific Percent";
             FuelChl       = "Chlorine Percent";
             FuelAsh       = "Ash Percent";
             FuelCap       = "Capacity");
```

This defines the one-dimensional data tables FuelType, FuelPrice,FuelSulphur etc taken from the database relation "FuelSrc". All of these tables are dimensioned by the index set site indicated by the key word INDEX. Thus the new keyword "IMPORT" enables the definition of several data tables at once. A two or more dimensional table is similarly defined. For example:

```
IMPORT
  Database("PoCoTak", INDEX company, fueltype
          DATA Mintake [company, fueltype] = "Min Take");
```

Data tables may also be derived by manipulating pre-defined data tables. This is performed using the "IN" operator described earlier. For example, if SiteRoutes is defined to be a

multidimensional set representing the routes which terminate at a particular fuel site, that is,

```
INDEX
    SiteRoutes[site,route] := Database("Routes",site,route);
```

and FuelPrice is defined to be a one-dimensional data table:

```
IMPORT
      Database("FuelSrc" INDEX site,
            DATA FuelPrice := "Price") ;
```

Then a new data table may be defined as:

```
DATA
      FuelPriceRoute [route] := FuelPrice(site IN SiteRoutes [site, route]);
```

For each route, the fuelprice for the route is obtained by selecting the site at which this route begins (from siteRoutest site,route)) and then selecting the fuel price for this site (from FuelPrice [site]). As before, it is not obligatory to specify the indices
[site, route] as SiteRoutes is previously defined, but this practice aids clarity.

The "IN" operator may be also used in the constraint section of the MPL model: in defining summations. For example:

```
MinimumFuel [company,fueltype] :
  SUM(route IN CompanyRoutes [company,route]
      WHERE (fueltype = fueltype IN FuelRoute[fueltype,route]:
          AmountShipped [route]) >=CompanyMinTake [company,fueltype];
```

This defines constraints called MinimumFuel: one for each combination of company and fuel type. For each company and fueltype, the summation is performed over the index route selected from the table (or subset) CompanyRoutes, where the route carries that particular fueltype. This latter information is contained in the table FuelRoute so the selection of routes which transport this fuel type is defined by the where condition which only chooses routes which have the same fueltype as the constraint index. For example, company 2 has routes 4,5,6,11,16 and 20 (see diagram 4.1) but only routes 11 and 16 accommodate fueltype 3, so this constraint for company 2 and fuel type 3 would be:

AmountShipped[routes 11 & 16] ≥ CompanyMinTake[company 2, fueltype 3]

A full (extended) MPL version of this example is provided in the Appendix.

6. Conclusion

LP models are usually constructed using index sets and data tables. These index sets and data tables are closely related to the attributes and relations of RDB constructs. The extensions of MPL to connect the modelling language with the RDB system are presented in this paper. The obvious advantage of such an approach is that, within an organisational context, the data needed for the model is held in a database system which is updated according to the organisation's requirements. Consequently, models which are constructed with database connections are automatically modified and revised. This model and data independence is extremely valuable from an operational point of view. The version of extended MPL in operation at BruneL University is connected to Paradox.

Early modelling systems (matrix generators) such as DATAFORM *(Ketron, 1975)* introduced some database features within the modelling language itself. In our view this replication of relational constructs which already exist within database systems is not desirable. More recently, Choobineh showed how the Structured Query Language (SQL) may be extended with additional constructs for the definition of objective functions and constraints to form SQLMP (SQL for Mathematical Programming) *(Choobineh, 1991)*. Our approach adopts the modeller’s viewpoint and therefore focuses more on the description and documentation of the model in a natural algebraic form.

The MIMI Modelling system *(Baker, 1992)* has its own internal database and provides

external links with an SQL server. Our approach, however, is not to develop another database system, but to provide the connection between the modelling language and the database; thus reusing the existing database and modelling software. This also allows many well established database features, for example, data security and unit checking, to be exploited. This also has the advantage of providing safeguards such as data integrity by the normalisation of the database.

The database connections introduced in this paper are easily extended to provide highly desirable features for model investigation, namely model management, solution and report analysis and case management. Thus the link between the modelling language and the database system is a two way relationship enabling data to be updated as a result of optimising the LP model as well as within the database environment. For example, once values for the decision variables have been obtained, it may be desirable to store these values in a database table.

7. References

Baker, T., 1992, MIMI/LP User Manual, Chesapeake Decision Science Inc., NJ, USA

Brooke, A., D. Kendrick & A. Meeraus, 1988, GAMS A User's Guide, The Scientific Press

Butler M., R. Bloor & P. Beach, 1990, Database:an Evaluation and Comparison, Butler Bloor

Choobineh J., 1991, SQLMP: A Data Sublanguage for Representation and Formulation of Linear Mathematical Models, ORSA Journal on Computing, Vol 3, No 4, Fall 1991

Dash Associates, 1991, LP-Model, Northants, UK

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

Fourer, R. D.M. Gay & B.W. Kernighan, 1993, AMPL: A Modeling Language for Mathematical Programming, The Scientific Press

Geoffrion, A.M., 1991, The SML Language fro Structured Modeling: Levels 1 and 2, Western Management Science Institute, University of California, LA, USA

Greenberg H.& F. Murphy , 1991, A Comparison of Mathematical Programming Modeling Systems, March 1991, University of Colorado at Denver, CO. USA

Greenberg, H. 1991, A Primer for MODLER: Modeling by Object Driven Linear Elemental Relations, December 1991, Mathematics Department, University of Colorado at Denver, CO. USA

Hürlimann, T., 1990, Reference Manual for the LPL Modeling Language (Version 3.5), Working Paper No. 175, Institute for Automation and Operations Research, University of Fribourg, Switzerland

Hürlimann, T., 1991, Linear Modeling Tools, Working Paper No. 187, Institute for Automation and Operations Research, University of Fribourg, Switzerland

Ketron Inc. MPS III DATAFORM:User Manual, 1975

Kristjansson, B., 1991, MPL Modelling System User Manual, Maximal Software Inc., Iceland

Kristjansson, B., 1993, MPL Version 2.8, Maximal Software Inc., Arlington, VA, USA

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

MathPro Inc., 1992, MathPro Software, MathPro Inc., Washington DC, USA

Sharda, R., 1992, Linear Programming Software for Personal Computers: 1992 Survey in

OR/MS Today June 1992
Ullman, J.D., 1982, Principles of Database Systems, Computer Science Press

Appendix - Power Model in MPL

```
{  Power. mpl  }
{  Fuel allocation model for a power industry.  }

TITLE Power;

CONST
      Total Fuel Types =   4;
INDEX
      company    := Database("PoCoS02", "Company Name");
      station          := Database("PowStat", "Station Name");
      site         := Database("FuelSrc", "Fuel Site Name");
      route        := Database("Routes", "Route Number");
      fueltype      := 1..Total Fuel Types;

!     Definitions of multidimensional sets
!     ----------------------------------------
!     Power <station> owned by <company>.     (station –> company)
!     ------------------------------------------------------------------------------
      StationOwners [company, station] :=
         Database("PoStOwn", company, station);

!     <fuel type> for power <station>.      (fueltype –> station)
!     ------------------------------------------------------------------
      FuelStation [station, fuel type] :=
          Database("PowStat", station, "Fuel Type Required");

!     <routes> that have power <station> as a source.  (route -> station)
!     ------------------------------------------------------------------------------
      StationRoutes[station, route] :=
          Database ("Routes", station, route);

!     <routes> that have fuel <site> as a sink.    (route -> site)
!     ------------------------------------------------------------------
      SiteRoutes [site, route] :=
           Database ("Routes", site, route);
!     <fueltype> of a <site>.  (site -> fuel type)
!     ---------------------------------------------------
      FuelType Site [site, fuel type] :=
          Database("FuelSrc", site, "Fuel Type Available");
```

```
!       <routes> to a power <station> owned by a <company>.
!       (routes –> station –> company)
!       ------------------------------------------------------------------
        CompanyRoutes [company, route] WHERE
        FORSOME (station IN Station Owners [company, station] =
                                station IN StationRoutes [station, route]));
!       <routes> from a <site> that produces <fuel type>.
!       (routes –> site –> fuel type)
!       ----------------------------------------------------------
        FuelRoute [fuel type, route] WHERE
            FORSOME (site IN Fuel Type Site [site, fuel type]  =
                               site IN SiteRoutes [site, route]);
DATA
        Sulphur Constant                := DATAFILE(suIconst.dat);

IMPORT
        Database ("PoCoS02", INDEX company
                                DATA MaxS02 =  "Max S02");
        Database ("PoCoTak", INDEX company, fuel type
                                 DATA MinTake = "Min Take");
        Database ("PowStat",   INDEX  station
                               DATA   HeatReq          = "Heat Requirement",
                                      FgdPercent       = "FGD Measure",
                                      MinSulphur       = "Minimum Sulphur",
                                      MaxChlorine      = "Maximum Chlorine",
                                      MinAsh           = "Minimum Ash");

        Database("Fuel Src", INDEX site,
                               DATA FuelPrice       = "Price",
                                    FuelCal         = "Calorific percent",
                                    FuelCap         = "Capacity");
                                    FuelSulphur     = "Sulphur percent",
                                    FuelChl         = "Chlorine percent",
                                    FuelAsh         = "Ash percent",

        Database ("Routes", INDEX route,
                          DATA RouteCost    = "Route Cost");

        FuelPriceRoute[route]    := FuelPrice[site IN Site Routes [site, route]];
        FgdRoute [route]         := Sulphurconst*(1 – FgdPercent [station IN
                                              StationRoutes [station, route]]);
        Fuel SulRoute[route]     := FuelSulphur[site IN SiteRoutes [site, route]];

DECISION VARIABLES
        Amount Shipped [route];

MODEL
        MIN  costs  =  SUM (route: (FuelPriceRoute[route] + RouteCost[route])
                               * AmountShipped[route]);
```

SUBJECT TO

```
!       Maximum emissions of S02 for each <company>.
!       ------------------------------------------------------------
        MaximumS02[company] :
            SUM(route IN CompanyRoutes[company, route]:
                 FgdRoute[route] * FuelSulRoute[route] * AmountShipped[route])
            <= Max S02 [company];

!       Minimum order quantities of a <fuel> for a <company>.
!       ------------------------------------------------------------------
      MimimumFuel[company, fue1 type] :
          SUM(route IN Company Routes[company, route]
              WHERE (fuel = fuel IN FuelRoute[fueltype, route]):
                   AmountShipped[route])
          >= CompanyMinTake [company, fuel);

!       Individual power <station> heat requirements.
!       ---------------------------------------------------------
        HeatRequirement [station] :
            SUM(route IN StationRoutes[station, route] :
                 FuelCal[site IN SiteRoutes[site, route]] * Amount Shipped [route])
            = HeatReq [station];
!       Individual power <station> minimum sulphur level.
!       ------------------------------------------------------------
        MinSulhpurLevel[station] :
            SUM(route IN StationRoutes[station, route] :
                 FuelSulphur[site IN SiteRoutes[site, route]] * mountShipped[route]
            -   Min Sulphur [station] * AmountShipped[route])
            >=  0;

!       Individual power <station> maximum chlorine level.
!       ------------------------------------------------------------
        MaxChlorine Level[station] :
            SUM(route IN Station Routes[station, route] :
                 FuelChlorine[site IN SiteRoutes[site, route]] * AmountShipped [route]
```

```
        - MaxChlorine[station] * AmountShipped[route])
        <=  0;


Individual power <station> minimum ash level.
-------------------------------------------------------------
MinAshLevel[station] :
        SUM(route IN Station Routes [station, route] :
              FuelAsh[site IN SiteRoutes[site, route]] * AmountShipped[route]
        -  Min Ash [station] * Amount Shipped [route])
        >= 0;
 Capacity of individual fuel <site>.
-----------------------------------------
FuelSiteCapacity[site] :
    SUM(route IN SiteRoutes[site, route] : AmountShipped[route])
    <=   FuelCap[site] ;
END
```