AT&T
Bell Laboratories

Numerical Analysis Manuscript 91–06

Expressing Special Structures in an Algebraic Modeling Language for Mathematical Programming

Robert Fourer[*]
David M. Gay[†]

May 31, 1991

[*]Deptartment of Industrial Engineering and Management Science, Northwestern University, Evanston, IL 60201-3119, U.S.A.; `4er@iems.nwu.edu` .

[†]AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, New Jersey 07974-0636, U.S.A.; `dmg@research.att.com` .

1. Introduction

Certain special structures play a valuable role in both the formulation and solution of linear programs. When these structures are recognized explicitly, an LP model can be formulated in a simpler and more natural way. When these structures are communicated to correspondingly specialized algorithms, the resulting LP problems can also be solved faster.

The benefits of special structure are not always so easy to realize, however. It is widely recognized that, for linear programs of typical size and complexity, some kind of computer software is needed to bridge the gap between the modeler's conception and the algorithm's requirements. To take advantage of a structure, this software must be able to recognize the structure's presence in its input from the modeler, and must be able to communicate the structure in its output to the algorithm.

In the case of a particular application, the necessary software can be provided by writing a "matrix generation" program that is also specialized to the structure at hand; but the difficulties of debugging and maintaining such a program are well known. Alternatively, one can use software that has been designed to read LP formulations and to convert them automatically for solution. Most such systems are intended for general-purpose linear programming, however, and so lack any facilities for dealing with structures. Other systems that do recognize structure are highly specialized to particular structures or applications.

We are thus led to explore the possibility of a general-purpose linear programming system that offers the *option* of specifying certain structures. The structure features of such a system would be implemented as extensions to the basic facilities for linear programming. Ideally, these extensions would be independent, in the sense that a user not concerned with a certain structure would not need to know anything about the extensions for that structure. At the same time, the structure extensions would be tightly integrated with the rest of the system, so that they would not burden the user with too much in the way of new syntax or concepts.

As an example of how general-purpose LP software can accommodate special structures, this paper describes the design of extensions to the AMPL mathematical programming language and translator, for the purpose of handling network, piecewise-linear and related structures. The emphasis is on the design considerations that motivate these extensions, and on the issues that arise in integrating the extensions with the general-purpose features of the language. Close attention to design is seen to be essential in providing a syntax that is natural to use, yet that achieves the goals of independence and integration stated above.

The remainder of this introduction presents a brief self-contained summary of the essential features of AMPL, and outlines the rest of the paper.

1.1 Essentials of AMPL

AMPL is an *algebraic modeling language:* a computer-readable language for describing objective functions and constraints in the kind of algebraic notation that many modelers use. Languages of this kind, such as GAMS [1, 2] and MGG [15], were first developed in the 1970s, and have seen increasingly wide use in recent

$\mathcal{R}$, a set of raw materials $\mathcal{P}$, a set of products	`set R;  # raw materials` `set P;  # products`
a_{ij}, $i \in \mathcal{R}$, $j \in \mathcal{P}$: input-output coefficients	`param a {R,P} >= 0;` `# input-output coefficients`
b_i, $i \in \mathcal{R}$: units available	`param b {R} > 0;  # units available`
c_j, $j \in \mathcal{P}$: profit per unit u_j, $j \in \mathcal{P}$: production limit	`param c {P} > 0;  # profit per unit` `param u {P} > 0;  # production limit`
x_j, $j \in \mathcal{P}$, with $0 \leq x_j \leq u_j$: units of j produced	`var x {j in P} >= 0, <= u[j];` `# units of j produced`
Maximize $\sum_{j \in \mathcal{P}} c_j x_j$: total profit	`maximize tot: sum {j in P} c[j] * x[j];` `# total profit`

Subject to $\sum_{j \in \mathcal{P}} a_{ij} x_j \leq b_i$, $i \in \mathcal{R}$:
limited availability of material
	`subj to supply {i in R}:`
	`sum {j in P} a[i,j] * x[j] <= b[i];`
	`# limited availability of material`

Figure 1–1. *A simple linear program in algebraic notation.* At left is a traditional informal algebraic description; at right is an equivalent in the AMPL modeling language.

years. AMPL is notable for its particularly natural and general syntax, and for its variety of set and indexing expressions.

The AMPL language employs a standard computer character set, and permits none of the ambiguity that is tolerated in informal algebraic notation. Thus a term such as $\sum_{j \in \mathcal{P}} a_{ij} x_j$ becomes `sum {j in P} a[i,j] * x[j]`, in which the subscripts of `a` must be separated by a comma, and the multiplication is indicated by an explicit operator. Figure 1–1 displays a linear programming model side-by-side with its equivalent in AMPL. Even in this simple case one can see the essential features of the five basic kinds of AMPL declarations: `set` (index set), `param` (parameter), `var` (variable), `maximize` or `minimize` (objective), and `subject to` (constraint).

The design of AMPL enforces a distinction between an optimization *model*, such as is seen in Figure 1–1, and the *instances* of that model that correspond to particular choices of set and parameter data. To employ AMPL in solving a particular instance, one must supply both a model and a data description, which are processed by an AMPL translator to produce the information required by a solver. Since the extensions described in this paper affect only the way that the model is formulated, we do not show the data description in our examples.

Although the AMPL translator can process nonlinear expressions in the variables, this paper is almost exclusively concerned with linear (or piecewise-linear) objectives and linear constraints. Thus our presentation views AMPL as a linear programming language.

An earlier paper [8] describes the features of AMPL in detail. Much of the same material, together with four extended examples, is presented in a technical report [7] available from the authors.

1.2 Outline

Section 2 considers the problem of specifying objective function terms that are piecewise-linear in individual variables. Under reasonable conditions, piecewise-linear terms can be reduced to linear ones by substituting special structures of auxiliary variables or constraints. We propose instead a new syntax and conventions that permit piecewise-linearities to be described and processed directly.

Section 3 deals with the specification of network flow linear programs, which have a particularly distinctive constraint structure that relates directly to the arrangement of network nodes and arcs. We examine the drawbacks of specifying network constraints algebraically, and propose an alternative in which `node` and `arc` declarations specify the network directly.

Section 4 extends our analysis to more general kinds of network structures in LPs. We show that slight further extensions of AMPL suffice to permit a natural description of the network structure in three common cases: weighted flows, side constraints, and side variables.

The approach of the previous two sections is applied in Section 5 to another special structure, that of set covering problems. We then observe that the AMPL extensions for both networks and set covering can be regarded as cases in which the constraint coefficient matrix is specified "by column" rather than "by row". This leads to our final extension, which permits column-wise specification of any constraint structure.

Our concluding comments in Section 6 address some of the broader issues raised by this paper. Our extensions allow the language to specify a model in more than one way, by use of one or another extension; we argue that this sort of redundancy is a virtue of the design. Our column-wise extensions go against the row-wise orientation that is generally thought to be an inherent aspect of algebraic languages; we point out the many of the strengths of an algebraic modeling language are realized regardless of how the constraints are specified.

All of the extensions described in this paper have been implemented in an AMPL translator. We present statistics to show that the translator often works faster when these extensions are used. We also argue that, when structures are specified explicitly in AMPL, certain preprocessing steps of structure-exploiting algorithms can be simplified or eliminated.

2. Piecewise-Linear Programming Models

A continuous function of one variable is piecewise-linear (P-L) if its domain can be partitioned into intervals on which the function is linear. Figure 2–1(a) shows an arbitrary example. Numerous applications of optimization involve P-L functions in some guise.

We consider here a special case of piecewise-linearity, wherein the objective is to minimize a sum of convex P-L functions of individual variables, or equivalently to maximize a sum of concave P-L functions, and the constraints are linear. Even under these restrictions, the variety of applications remains considerable [6]. Yet the difficulty of solution is not significantly greater than for a purely linear program in the same number of variables and constraints [11].

For convenience, we refer to the special P-L optimization problems of interest as piecewise-linear programs, or P-LPs. In principle, any P-LP can be converted to an equivalent linear program, by any of four transformations described in [6]. The linear program can be expressed and solved by use of an algebraic modeling language such as AMPL. There are two serious drawbacks to this kind of transformation approach, however:

- People conceive of the model in its original piecewise-linear formulation, and would prefer to work with it in that form.
- Equivalent linear formulations have considerably more variables or constraints; for this and other reasons, as shown in [6, 11], they tend to be quite a bit more expensive to solve.

We thus have good reason for wanting P-L objective terms to be expressible directly in AMPL.

Whereas the linear expressions of AMPL are based on common algebraic notation, there is no commonly accepted terminology for piecewise-linear functions. We have had to invent a terminology, which is described in this section. We first explain the underlying principles of the design, and then describe the specifics.

2.1 Design requirements

A piecewise-linear program, in its most general form, can be regarded as the optimization of

$$\sum_{j=1}^{n} f_j(x_j)$$

subject to linear equalities and inequalities in the variables, where f_j is a convex or concave piecewise-linear function of the variable x_j. AMPL can express the $\sum_{j=1}^{n}$ and the constraints just as for linear programs, as seen in the example of Figure 1–1, but some kind of extension is needed to accommodate the individual functions $f_j(x_j)$.

Typical convex piecewise-linear functions are depicted in Figures 2–1(b) and 2–1(c). Each f_j is characterized by increasing slopes between successive breakpoints. Indeed, any increasing series of breakpoint values together with an increasing series

Figure 2–1. *Piecewise-linear functions of one variable.* The models considered in this section minimize a sum of convex functions (b, c), or maximize a sum of concave functions (d). The finite domains of the functions may be cut off by lower bounds (c, d) or upper bounds (d).

of intervening slope values must define a convex P-L function up to an additive constant. The first finite breakpoint may lie to the left of the first finite slope, in which case the breakpoint is effectively a lower bound on the value of x_j; or the first finite slope may lie to the left of the first finite breakpoint, so that f_j is linear with that slope on the entire half-line to the left of the breakpoint. Similarly the last finite breakpoint may be an upper bound on x_j, or f_j may be linear on the entire half-line to the right of the breakpoint.

Figure 2–1(d) shows a typical concave piecewise-linear function. It is just the same as a convex example, except that its slopes are decreasing between successive breakpoints.

We wish to have an AMPL syntax for $f_j(x_j)$ that can concisely and clearly describe any piecewise-linear function. The above discussion suggests, in particular, a syntax that explicitly presents separate slope and breakpoint sequences, with conventions to deal with the fine points: whether the first (or last) breakpoint is to the left or right of the first (or last) slope; and which function is intended, out of all those differing by an additive constant. We proceed to describe our solution next, and then compare it to other possibilities.

2.2 Design specifics

The AMPL syntax for a piecewise-linear function has the following general form:

<< *breakpoints*; *slopes* >> *variable*

The components denoted *breakpoints* and *slopes* are the defining sequences, whose form is to be described shortly. They are enclosed in `<<...>>` so that they comprise an entity clearly different from any of the other algebraic constructs in AMPL. The entire entity is followed by the variable to which it applies; one can think of the entity as "multiplying" the variable by several slopes, rather than by just the one slope of a linear term.

The breakpoints and slopes are given, in the simplest case, by comma-separated lists of values. We adopt the conventions that

- there are finite slopes before the first finite breakpoint, and after the last finite breakpoint; and
- the function's value at zero is zero.

Thus, for example, the absolute value function of an AMPL variable `x[j]` is written

```
<<0; -1, 1>> x[j]
```

since the single breakpoint is at zero, and the slopes are -1 to the left and $+1$ to the right. A more complicated instance is almost as easily expressed:

```
<<-1,1,3,5; -5,-1,0,1.5,3>> x[j]
```

In an objective function, this term would tend to penalize deviation of `x[j]` from the interval between 1 and 3.

If finite breakpoints are required before the first slope or after the last one, they can be represented by simple upper or lower bounds. As an example, a convex increasing cost term can be written

```
<<3,5; 0.25,0.5,1.5>> x[j]
```

where `x[j]` has been previously declared in an AMPL statement like `var x {j in S} >= 0, <= 6`. This is a convenient arrangement for most piecewise-linear programs. The lower bound of zero is a routine non-negativity requirement, which would be found in the declarations of all variables representing physical quantities. The upper bound is also conveniently expressed as a property of the variable rather than of the P-L objective function.

Multiplication of a P-L term has the effect of multiplying all its slopes. Thus,

```
0.25 * <<0; -1,3>> x[j]
```

is the same as the convex function `<<0; -0.25,0.75>> x[j]`. On the other hand, multiplication by a negative number,

```
-1 * <<0; -1,3>> x[j]
```

gives a concave function that could just as well be written `-<<0; -1,3>> x[j]` or `<<0; 1,-3>> x[j]`. In general, any concave P-L function can be expressed as the negative of a convex one, but not always so conveniently. Consider:

```
<<1,2,4.5; 2,1,.2,0>> x[j]
```

This concave increasing function is the same as `-<<1,2,4.5; -2,-1,-.2,0>> x[j]`.

The slopes and breakpoints need not be literal numbers as in the above examples; they could just as well be any AMPL expressions. However, these examples do fix the numbers of slopes and breakpoints. This is no limitation when the terms are simple functions such as absolute values. But in other cases, such as where the terms represent penalties or convex increasing costs, it can be desirable to let the number of linear pieces be itself a function of the data. AMPL already provides for data-dependent numbers of many kinds of entities, by allowing them to be indexed over sets. Thus it is natural to allow indexed slope and breakpoint series as well.

As an example, we could let `npce` be an AMPL parameter representing the number of linear pieces in every term f_j. Then we could define parameter arrays such that `bkp[k]` and `slp[k]` are the `k`th breakpoint and slope values, respectively:

```
param npce integer > 1;
param bkp {1..npce-1};
param slp {1..npce};
```

A piecewise-linear term with these breakpoints and slopes is written as follows:

```
<<{k in 1..npce-1} bkp[k]; {k in 1..npce} slp[k]>> x[j]
```

The construct `{k in 1..npce-1}` is an AMPL indexing expression, the same as is used in declaring indexed collections of parameters, indexed sums, and the like. Here it is used to indicate that the breakpoints are `bkp[1]`, `bkp[2]`, ..., `bkp[npce-1]`; the number of breakpoints evidently depends on the value for `npce` that is supplied in the model data. Analogous comments apply to the slopes.

As a bit more elaborate example, suppose that the variables `x[j]` are indexed over some set `S`, and that we want to have a potentially different number of linear pieces for each `j` in `S`. This is readily handled in AMPL by indexing `npce` over `S`, and indexing `bkp` and `slp` over both `S` and the number of pieces:

```
set S;
var x {S} >= 0;

param npce {S} integer > 1;
param bkp {j in S, 1..npce[j]-1};
param slp {j in S, 1..npce[j]};

minimize cost:
     sum {j in S} <<{k in 1..npce[j]-1} bkp[j,k];
                    {k in 1..npce[j]} slp[j,k]>> x[j];
```

The number of pieces in the term for `x[j]` is `npce[j]`; the breakpoints in order are `bkp[j,1]`, ..., `bkp[j,npce[j]-1]`, and the corresponding slopes are `slp[j,1]`, ..., `slp[j,npce[j]]`.

The general AMPL format for a piecewise-linear term is

<< *bkpt1*, *bkpt2*, ...; *slope1*, *slope2*, ... >> *variable*

where each of the items *bkpt1*, *bkpt2*, ... and *slope1*, *slope2*, ... has one of the forms

expr
{*indexing*} *expr*

The *expr* must be an AMPL expression that evaluates to a number. The optional *indexing* must specify an ordered AMPL set: either a set entirely of numbers (as in our examples) or a set that has been declared to be ordered (with the keyword `ordered` or `circular`). Any indexed item is expanded to a sequence of numbers by taking each member of the set in order, substituting it into *expr*, and evaluating the result. After all expansions are made, the breakpoints must be nondecreasing, and the number of slopes must be one more than the number of breakpoints.

If it is desired to have the P-L function evaluate to other than zero at zero, one may simply add a constant term, observing that

constant-expression + << *bkpt1*, *bkpt2*, ...; *slope1*, *slope2*, ... >> *variable*

has the value of *constant-expression* at zero. AMPL also recognizes the more general notation

```
<< bkpt1, bkpt2, ...; slope1, slope2, ... >> (variable, constant-expression)
```

which is interpreted as before, except that *constant-expression* gives the value of *variable* at which the function is zero.

Two complete examples are exhibited in the appendices. STRUC is a structural design model that uses simple two-piece terms (Appendix C). SCORE is an equation-fitting model for credit scoring, in which the number and value of the breakpoints and slopes depend in various ways on the problem data (Appendix B).

2.3 Other design issues

Because earlier versions of AMPL did not support ordered sets, the above-described convention concerning the order of the breakpoints and slopes could not be employed. Instead, all specified breakpoints and slopes were implicitly sorted into increasing order; as a result, any expression of the form `<<...; ...>>` *variable* specified a legal *convex* piecewise-linear function, so long as the number of slopes was one more than the number of breakpoints. The limitation to convex P-L functions had little practical effect upon the language's expressive power, since any concave function can be written as the negative of a convex one, while a P-L function that is neither convex nor concave cannot readily be optimized in any case.

Nevertheless, we prefer the current restriction to ordered sets, because it requires the modeler to more explicitly state the intended P-L function. The list of breakpoints is required to be increasing, so that there is never any doubt as to which slope goes between which breakpoints. Slopes can be increasing or decreasing, depending on whether a convex or concave function is required.

Regardless of the slope and breakpoint ordering conventions, it is necessary to deal with the possibility that a model will specify a non-convex P-L function to be minimized, or a non-concave one to be maximized. The AMPL language does not regard this as as error, since the resulting optimization problem is well defined—even through it is hard to solve. If the modeler desires, AMPL's `check` statements can be used to verify that the slopes are in the expected order; in the case of the above example, the following would be appropriate:

```
check {j in S, k in 1..npce[j]-1}: slp[j,k] <= slp[j,k+1];
```

It is not necessary to check the breakpoints in this way, since it is an AMPL error to specify them in other than an increasing order.

In the absence of an effective check, any attempt to minimize a non-convex P-L function (or to maximize a non-concave one) should be rejected unless an algorithm is available for such a purpose. When the interface between the AMPL translator and the solvers is a simple file-based one such as described in [8], then in fact the convexity test is left to the solver; while at least one solver of P-LPs, CPLP [11], does in fact make such a test, there is no guarantee that other P-L solvers hooked to AMPL will do so. Thus there is at least a little risk that an inappropriate algorithm may be applied, and a bad solution may be returned with no warning. To provide greater security, the AMPL system would have to be able to automatically check a model's attributes against a table of solver capabilities, before allowing the model to be passed on for solution.

We conclude by noting that not all P-L functions of interest are zero at zero, as the AMPL convention assumes. The function's value at zero may in fact be of only nominal interest, particularly in cases where the variable has a positive lower bound. Instead the function value may be zero at the first or last breakpoint, or at a breakpoint where the function is smallest. Our convention may require a messy constant term to adjust the function level for these cases.

Just about any convention will be awkward for some cases, however. We have adopted the "zero at zero" convention because it is simple and independent of the data. In the final analysis, the convention makes no difference to the optimization; it is important only to the reporting of optimal results. The software that presents the results could well offer the option of adjusting the reported P-L function level in any of several desired ways.

3. Network Flow Models

The linear network flow model is one of the best known and most important in linear programming. In its purest form, it is based on the sort of directed network exhibited in Figure 3–1, in which nodes are connected by "one-way" arcs. Some kind of entity is imagined to flow, literally or figuratively, from node to node along the arcs, so that the decision variables are the levels of these flows. The objective function is some linear function of the flows; the constraints are simple bounds on the flows, and balances of flow at the nodes. Details are supplied in Chvátal [3] and many other linear programming texts.

Network flow models are important because people often want to formulate them, and because they are particularly easy to solve by means of specialized algorithms. Although the original design of AMPL can express virtually any of these models, the resulting constraint expressions are often not as natural as we would like. Indeed, it can be hard to tell whether a model's algebraic constraints, taken together, represent a valid collection of flow balances on a network. As a result, when specialized network optimization software is used, additional checking of the model must be made outside of the language; Zenios [17] describes a comparable arrangement in conjunction with the GAMS algebraic modeling system [2].

To overcome the awkwardness of expressing network flow models in AMPL, we have added the option of declaring nodes and arcs more explicitly. We first explain the source of the awkwardness at greater length below, and then detail the design of the network extensions.

3.1 Design requirements

The issues faced by our design can be seen even in the simplest model of minimum-cost flows on a directed network. The structure of the network is determined by a set $\mathcal{N}$ of nodes, and a set $\mathcal{A} \subseteq \mathcal{N} \times \mathcal{N}$ of arcs, such that $(i,j) \in \mathcal{A}$ if and only if there is an arc from node i to node j. The objective is to minimize the total cost of the flows x_{ij} along all the arcs,

$$\sum_{(i,j)\in\mathcal{A}} c_{ij} x_{ij},$$

subject to bounds on the flows along the arcs,

$$l_{ij} \le x_{ij} \le u_{ij}, \quad (i,j) \in \mathcal{A},$$

Figure 3–1. *Diagram of a directed network,* in which circles represent the nodes, and arrows the arcs. For the models considered by this section, a cost and maximum flow are associated with each arc, and a required net flow is associated with each node.

and balance of flow at the nodes. The latter constraints have the form

$$\sum(\text{flows out of } i) - \sum(\text{flows into } i) = b_i, \quad i \in \mathcal{N},$$

where b_i is the amount of flow produced at node i if $b_i \geq 0$, or $-b_i$ is the amount consumed at node i if $b_i \leq 0$. (The special case of $b_i = 0$ defines a pure "transshipment" node at which flow out equals flow in.)

The flow balance constraints pose the main difficulty of this model for algebraic notation. To be explicitly correct, these should be written as

$$\sum_{j \in \mathcal{N}:(i,j)\in\mathcal{A}} x_{ij} - \sum_{j \in \mathcal{N}:(j,i)\in\mathcal{A}} x_{ji} = b_i, \quad i \in \mathcal{N},$$

but the required indexing expressions are then so long as to be awkward. Instead one can abbreviate them, writing

$$\sum_{(i,j)\in\mathcal{A}} x_{ij} - \sum_{(j,i)\in\mathcal{A}} x_{ji} = b_i, \quad i \in \mathcal{N}.$$

Since the constraint is being defined for a given i, it makes sense to interpret $(i,j) \in \mathcal{A}$ in this context as the set of all arcs out of i, and $(j,i) \in \mathcal{A}$ as the set of all arcs into i.

We will refer to the AMPL version of this model as TRANSSHIP. Its data can be declared as follows:

```
set N;
set A within N cross N;

param b {N};
param c {A} >= 0;

param l {A} >= 0;
param u {(i,j) in A} >= l[i,j];
```

The variables can then be defined, along with their bounds, by

```
var Flow {(i,j) in A} >= l[i,j], <= u[i,j];
```

We name the variables `Flow` rather than `x`, to make the model a little clearer, and to help with the subsequent exposition. We can now easily write the objective function as

```
minimize total_cost: sum {(i,j) in A} c[i,j] * Flow[i,j];
```

Finally, the flow balance constraints are expressed by

```
subject to Balance {i in N}:
     sum {j in N: (i,j) in A} Flow[i,j] -
     sum {j in N: (j,i) in A} Flow[j,i] = b[i];
```

or, more concisely,

```
subject to Balance {i in N}:
     sum {(i,j) in A} Flow[i,j] - sum {(j,i) in A} Flow[j,i] = b[i];
```

These representations of the constraints are direct transcriptions of the two alternatives formulated previously.

Whether in the language of mathematics or of AMPL, the expressions for the flow balance constraint are fundamentally unsatisfying. The idea of constraining "flow out minus flow in" is intuitively obvious, yet its algebraic equivalents are not so quickly comprehended. The problem is only worse in models of realistic complexity; as two examples, we give a planning and distribution model (DIST) in Appendix A and a scheduling model (TRAIN) in Appendix D.

The difficulty arises from the fact that standard algebra can directly describe only variables and constraints, while the existence of nodes and arcs is an implicit property of the formulation. People tend to approach network flow problems in just the opposite way. They imagine giving an explicit definition of nodes and arcs, from which flow variables and balance constraints implicitly arise. To deal with this situation, we have designed an extension to AMPL that permits the language to express the relevant network concepts directly.

3.2 Design specifics: `node` declarations

The network extensions to AMPL comprise two new kinds of declaration, `node` and `arc`, which take the place of the `var` and `subject to` declarations in an algebraic constraint formulation.

AMPL's `node` declarations name the nodes of a network, and characterize the flow balance constraints at the nodes. The `arc` declarations then name and define the arcs, by specifying the nodes that arcs connect, and by providing a variety of optional information associated with arcs. Since at least some of the nodes must be declared before any of the arcs, we begin by describing our design for the `node` declarations, and then introduce the `arc` declarations. A completed statement of the TRANSSHIP example using `node` and `arc` is exhibited in Figure 3–2, with the equivalent example using `var` and `subject to` placed alongside for comparison.

A `node` declaration names and defines a network node, or an indexed collection of nodes. Thus the TRANSSHIP example declares

```
node Balance {i in N}: net_out = b[i];
```

to define a node `Balance[i]` for each member `i` of the set `N`. The optional part of the declaration after the colon is a description of the flow balance constraint, as we will explain shortly.

One might informally think of `N` above as actually being the set of nodes; but in AMPL it is just a set of objects over which the collection `Balance` of nodes happens to be defined. Such a distinction permits a collection of nodes to be indexed over any set that can be described in AMPL, such as a set of pairs in the TRAIN model of Appendix D:

```
node N {cities,times};
```

This declaration specifies one node for each city at each time in the planning period.

More complicated network models have several kinds of nodes, each defined by

```
set N;
set A within N cross N;

param b {N};
param c {A} >= 0;

param l {A} >= 0;
param u {(i,j) in A} >= l[i,j];

minimize total_cost;

node Balance {i in N}:
  net_out = b[i];

arc Flow {(i,j) in A}:
  from Balance[i], to Balance[j],
  >= l[i,j], <= u[i,j],
  obj total_cost c[i,j];
```

```
set N;
set A within N cross N;

param b {N};
param c {A} >= 0;

param l {A} >= 0;
param u {(i,j) in A} >= l[i,j];

var Flow {(i,j) in A}
  >= l[i,j], <= u[i,j];

minimize total_cost:
  sum {(i,j) in A} c[i,j] * Flow[i,j];

subject to Balance {i in N}:
  sum {(i,j) in A} Flow[i,j] -
  sum {(j,i) in A} Flow[j,i] = b[i];
```

Figure 3–2. *Two AMPL descriptions of the* TRANSSHIP *model described in the text.* At the left, the `node` and `arc` extensions are employed; at right, the constraints are given in the customary algebraic way.

a separate `node` declaration. In the DIST example of Appendix A there are in fact 7 such declarations, which begin as follows:

```
node RT: rtmin <= net_out <= rt_max;   # source of regular crews
node OT: otmin <= net_out <= ot_max;   # source of overtime crews
node P_RT {fact};   # regular crews at factories
node P_OT {fact};   # overtime crews at factories
node M {prd,fact};   # manufacturing locations
node D {prd,dctr};   # distribution centers
node W {p in prd, w in whse}: net_in = dem[p,w];   # warehouses
```

The first two of these define individual nodes, while the others are declarations of indexed collections. Dummy indices, such as `p in prd`, are not specified except (as for `W`) where needed in the remainder of the declaration—which describes the balance condition at the nodes.

In the case of TRANSSHIP, the balance condition is described by the phrase `net_out = b[i]`. Intuitively, this says that the net flow out of node `Balance[i]`—that is, flow out minus flow in—must equal `b[i]`. This is precisely the condition that was previously specified by the algebraic constraint `Balance[i]` in the network model. Here, however, the outflow is represented by just the keyword `net_out`, rather than by the long expression `sum {(i,j) in A} Flow[i,j] - sum {(j,i) in A} Flow[j,i]`.

The DIST model shows some other possibilities. The balance condition may involve inequalities, as in the case of nodes `RT` and `OT`. When a node is a destination of flow, such as a warehouse, then it is more natural to specify the condition in terms of the net flow in; thus the condition for node `W[p,w]` is `net_in = dem[p,w]`. A pure transshipment node can be specified by either `net_in = 0` or `net_out = 0`; such a node is the default in AMPL, so it can be defined by giving no condition at all, as in several of the examples above.

In general, the balance condition may take any of the following forms:

> *net-expression* `=` *arith-expression*
> *net-expression* `<=` *arith-expression*
> *net-expression* `>=` *arith-expression*
>
> *arith-expression* `=` *net-expression*
> *arith-expression* `<=` *net-expression*
> *arith-expression* `>=` *net-expression*
>
> *arith-expression* `<=` *net-expression* `<=` *arith-expression*
> *arith-expression* `>=` *net-expression* `>=` *arith-expression*

The *arith-expression* may be any AMPL arithmetic expression in previously declared sets and parameters, while the *net-expression* is restricted to one of the following:

`net_in`	`net_out`
`net_in +` *arith-expression*	`net_out +` *arith-expression*
arith-expression `+ net_in`	*arith-expression* `+ net_out`

Any balance condition written in these ways is guaranteed to be a constraint on net flow of the kind that must appear in a network linear program.

As an alternative, we could allow the balance condition to be any AMPL expression that uses `net_in` or `net_out`. The keywords `flow_in` and `flow_out` might be provided as well, to represent just the flows in or flows out. This approach does have the appeal of introducing no new syntactic rules; but it allows the modeler to specify a balance condition that is not a proper restriction on flow out minus flow in, or indeed that does not represent any kind of "balance" at all. Such an alternative would go well beyond our immediate goal, of allowing network flow linear programs to be specified in an obvious way. Hence we have chosen to proceed more cautiously, by permitting only the forms of *net-expression* specified above.

3.3 Design specifics: `arc` declarations

An `arc` declaration names and defines a network arc, or an indexed collection of arcs. The arcs of the TRANSSHIP example are declared as follows:

```
arc Flow {(i,j) in A}:
     from Balance[i], to Balance[j],
     >= l[i,j], <= u[i,j], obj total_cost c[i,j];
```

There is one arc `Flow[i,j]` for each member `(i,j)` of the set `A`. This arc takes the place of the like-named variable in the algebraic model. Much as in the case of the nodes, one might informally think of `A` as actually being the set of arcs, but in AMPL it is a set of pairs of objects over which the collection `Flow` of arcs happens to be defined. More complex models have an arc declaration for each different kind of arc; in the TRAIN model, for example, two of the declarations begin:

```
arc U {c in cities, t in times} ...
arc X {(c1,t1,c2,t2) in schedule} ...
```

An arc `U[c,t]` represents train cars held over in city `c` at time `t`, while an arc `X[c1,t1,c2,t2]` stands for cars traveling in a train from city `c1` (at time `t1`) to city `c2` (at time `t2`).

The body of an `arc` declaration consists of a series of phrases, in any convenient order. (Commas separating the phrases are optional; their only use is to make the boundaries between phrases a little more obvious.) At the least, there must be `from` and `to` phrases to show where the arc goes in the network; their general form is

```
from node-name
to node-name
```

In TRANSSHIP, the arc `Flow[i,j]` is naturally declared to be `from Balance[i]` and `to Balance[j]`. More elaborate models use these phrases to establish the structure of the network. Thus within the TRAIN example, the `U[c,t]` arcs are `from N[c,t]` and `to N[c, if t < last then t+1 else 1]`, while the `X[c1,t1,c2,t2]` arcs are `from N[c1,t1]` and `to N[c2,t2]`.

Other phrases specify bounds on the flow along the arc. They have the general forms

```
>= arith-expression
<= arith-expression
```

In TRANSSHIP the bounds for arc `Flow[i,j]` are `>= l[i,j]` and `<= u[i,j]`; in DIST most of the bounds are `>= 0`. AMPL recognizes similar phrases with the operator being `=` to fix certain flows, or `:=` to suggest initial flows for algorithmic purposes.

Finally, there is a phrase for a linear objective function coefficient:

```
obj objective-name arith-expression
```

The *arith-expression* is evaluated to give the arc's contribution, per unit of flow, to the named objective's value. As in other AMPL models, the objective must be previously declared. For example, in TRANSSHIP, prior to the use of the objective function phrase `obj total_cost c[i,j]`, there is a declaration

```
minimize total_cost;
```

Unlike other `minimize` or `maximize` declarations, this one requires no expression for the objective function. The objective coefficients are fully defined by the `obj` phrase in the subsequent `arc` declaration.

As another example, the TRAIN model declares two objectives,

```
minimize cars;
minimize miles;
```

and the subsequent `arc X` declaration is seen to have two corresponding `obj` phrases:

```
obj cars (if t2 < t1 then 1 else 0),
obj miles distance[c1,c2];
```

The `arc U` declaration has only a phrase for the `cars` objective; by default, these arcs have coefficients of zero in the `miles` objective.

4. Generalizations of the Network Model Features

Many network applications diverge from the "pure" network flow formulation considered in the previous section. With only modest further extensions, however, our AMPL `node` and `arc` declarations can be used to specify several more general kinds of network.

This section describes extensions for network flow models with "multipliers" on the flows, and with extra ("side") constraints or variables.

4.1 Network flows with multipliers

The network balance-of-flow constraints may be generalized by allowing arbitrary coefficients on the flows in and out:

$$\sum_{(i,j)\in\mathcal{A}} v_{ij}x_{ij} - \sum_{(j,i)\in\mathcal{A}} w_{ji}x_{ji} = b_i, \quad i \in \mathcal{N}.$$

In terms of the network, one can imagine that the flow x_{ij} is from node i with multiplier v_{ij}, and to node j with multiplier w_{ij}. A pure network flow problem (as in Section 3) is just the default case in which all multipliers are 1.

AMPL supports this generalization by allowing an optional expression for the multipliers in a `from` or `to` phrase:

> `from` *node-name mult-expression*
> `to` *node-name mult-expression*

The *mult-expression* can be any AMPL expression in the previously declared sets and parameters. It defaults to 1 if absent.

An example of this feature is found in the DIST model's declaration of the arcs `Manu_OT`. Each arc `Manu_OT[p,f]` carries a "flow" of overtime labor—measured in crew-hours—from the overtime pool at factory `f` (node `P_OT[f]`) to the manufacturing facilities for product `p` at factory `f` (node `M[p,f]`). The balance constraint at the "to" node is not in terms of crew-hours, however, but rather in 1000s of cases produced. To correct for this difference, the flow must be multiplied by `1/pt[p,f]`, where `pt[p,f]` is the previously defined parameter that represents crew-hours needed to produce 1000 cases.

The declaration of `Manu_OT` thus contains the phrases

```
from P_OT[f]
to M[p,f] (1/pt[p,f])
```

The multiplier at the "from" node is left at its default of 1, since that node does have a balance constraint in terms of crew-hours.

Besides changes in units, this generalization is often used to model gains or (more often) losses in transportation between nodes.

4.2 Side constraints

Some network problems are constrained by more than just balance of flow at the nodes and bounded flows on the arcs. The additional "side constraints" come

```
set N;
set A within N cross N;

set P;  # products

param b {N,P};
param c {A,P} >= 0;

param l {A,P} >= 0;
param u {(i,j) in A, p in P} >= l[i,j,p];

param u_mult {A} >= 0;

minimize total_cost;

node Balance {i in N, p in P}: net_out = b[i,p];

arc Flow {(i,j) in A, p in P}:
     from Balance[i,p], to Balance[j,p],
     >= l[i,j,p], <= u[i,j,p], obj total_cost c[i,j,p];

subject to Mult {(i,j) in A}:
     sum {p in P} Flow[i,j,p] <= u_mult[i,j];
```

Figure 4–1. *A multicommodity flow model with side constraints.*

in many forms, but in general they do not correspond to individual nodes or arcs of the network. Thus it is desirable to let an AMPL model specify arbitrary algebraic constraints in addition to the constraints implied by `node` and `arc` declarations.

The problem of multicommodity flows provides a simple example. In the network model of Section 3, we can imagine the nodes as representing locations, and the flows along the arcs as being the transportation of some product. But what if there are many products? We can simply replicate the model so that there are separate network data, nodes and arcs for each product in some set. Instead of an arc `Flow[i,j]` for each `(i,j)` in `A`, for example, we have an arc `Flow[i,j,p]` for each `(i,j)` in `A` and `p` in `P`. If we carry out this replication and nothing more, we have a model that decomposes into separate pure network linear programs, one for each product.

In a more likely scenario, however, total shipments of all products from location `i` to location `j` are bounded by some overall upper limit. This additional multicommodity constraint ties the different product networks together; since it simultaneously involves many disjoint arcs `Flow[i,j,p]` for different products `p`, it cannot be modeled by merely adding more nodes and arcs. Rather, we recall from Section 3 that arcs correspond to variables in the associated network linear program. We can thus use the names `Flow[i,j,p]` to stand for the flow variables in an algebraic description of the multicommodity restriction.

As seen in Figure 4–1, the AMPL model defines overall shipment bounds for all `(i,j)` in `A`:

```
param u_mult {A} >= 0;
```

```
set N;
set A within N cross N;

set F;  # feedstocks
set P;  # products

param a {P,F} >= 0;
param u_feed {F,N} >= 0;

param b {N,P};
param c {A,P} >= 0;

param l {A,P} >= 0;
param u {(i,j) in A, p in P} >= l[i,j,p];

minimize total_cost;

var Feed {f in F, i in N} >= 0, <= u_feed[f,i];

node Balance {i in N, p in P}:
     net_out = b[i,p] + sum {f in F} a[p,f] * Feed[f,i];

arc Flow {(i,j) in A, p in P}:
     from Balance[i,p], to Balance[j,p],
     >= l[i,j,p], <= u[i,j,p], obj total_cost c[i,j,p];
```

Figure 4–2. *A multicommodity flow model with side variables.*

Then the desired constraints can be written as

```
subject to Mult {(i,j) in A}:
     sum {p in P} Flow[i,j,p] <= u_mult[i,j];
```

The entire model now clearly appears as a network (defined by `node` and `arc`) with side constraints (defined by `subject to`).

This example generalizes in the obvious way. Once an entity has been defined in an `arc` declaration, it represents a model variable, and may be used as such in any subsequent `subject to` declaration.

4.3 Side variables

Once we have defined how `subject to` declarations can be employed in conjunction with `node` and `arc` declarations, we can allow supplementary `var` declarations as well. A `var` declaration defines "side variables" that can be used, along with variables implicitly defined by `arc` declarations, in any `subject to` constraint.

The side variables of some network models appear not in the side constraints, however, but as complicating terms in the balance-of-flow constraints at the nodes. Figure 4–2 shows how these variables can be handled in an obvious way within the `node` declarations of AMPL. This model is based on a replication of Section 3's network model, just like the previous one. The separate product networks are held together not by additional constraints, however, but by additional variables representing feedstocks that can be used at the nodes.

The new declarations in Figure 4–2 are

```
param a {P,F} >= 0;
param u_feed {F,N} >= 0;
```

which define the amount `a[p,f]` of product `p` that can be derived from one ton of feedstock `f`, and the limit `u_feed[f,i]` on feedstock `f` available at location `i`. The connecting variables are declared as

```
var Feed {f in F, i in N} >= 0, <= u_feed[f,i];
```

where `Feed[f,i]` is the amount of feed `f` used at location `i`. The amount of product `p` derived from feedstock `f` at location `i` is given by `a[p,f] * Feed[f,i]`.

At the node `Balance[i,p]`, we want to specify that flow out minus flow in is equal to `b[i,p]` (as before) plus the total of product `p` that is derived from feedstocks at location `i`. Thus the AMPL node declaration must be:

```
node Balance {i in N, p in P}:
     net_out = b[i,p] + sum {f in F} a[p,f] * Feed[f,i];
```

To permit this formulation in AMPL, we need only slightly expand our previous definition of the `node` declaration, to allow the use of variables in an *arith-expression* within the balance condition.

The arcs of this model are defined as before, using an `arc` declaration. In this way, the distinction between network and side variables is emphasized by the AMPL formulation.

5. Other Column-Wise Structures

Although the AMPL network extensions can be motivated entirely by a desire to more naturally represent network flow models, they can also be viewed as a special case of a "column-wise" alternative to the specification of linear programs. As a result, some of the ideas of the `node` and `arc` declarations can reasonably be introduced into the standard `var` and `subject to` declarations as well.

This section first presents an extension for another special structure, that of set covering problems. It then shows how all of the preceding ideas can be adapted to permit column-wise specification of arbitrary linear programs.

5.1 Set covering

When we tried to represent various set covering models in AMPL, we encountered much the same kinds of problems that we had with networks. Thus we have used much the same approach in devising a solution, though the some important details are necessarily different.

In a simple kind of set covering problem, we are given subsets $\mathcal{S}_j \subset \mathcal{U}$ and costs c_j, for $j = 1, \ldots, n$. Our goal is to find the cheapest collection of subsets whose union contains all of $\mathcal{U}$. This problem is properly formulated as a zero-one integer program, in which the variable x_j is 1 if and only if the subset $\mathcal{S}_j$ is in the desired collection; but for purposes of illustration we can confine the present discussion to the linear programming relaxation:

$$\begin{array}{lll} \text{Minimize} & \sum_{1 \le j \le n} c_j x_j & \\ \text{Subject to} & \sum_{1 \le j \le n \,:\, i \in \mathcal{S}_j} x_j \ge 1, & \text{for each } i \in \mathcal{U} \\ & 0 \le x_j \le 1, & \text{for each } j = 1, \ldots, n \end{array}$$

As in the network case, the summation within the main constraint seems awkward, particularly compared to the very simple original description of the problem.

The awkwardness persists when the common algebraic notation is transcribed to AMPL, as shown in Figure 5–1. Moreover, the processing of this formulation is likely to be inefficient. When it comes to evaluating the summation in the constraints,

```
subject to complete {i in U}:
     sum {j in 1..n: i in S[j]} x[j] >= 1;
```

the condition `i in S[j]` must be tested for every one of the `n` subsets `S[j]`. In our implementation of an AMPL translator, these constraints are generated individually for each instance of `i in U`; as a result, the number of tests of the form `i in S[j]` that the translator must make is equal to `n` times the cardinality of `U`. Since the number of possible subsets of `U` grows exponentially, the size of `n` can easily be 100,000 or more, while values in the millions are not unknown for crew scheduling applications. As a result, this kind of AMPL model can become very expensive to translate.

There is a simpler (and more satisfying) way to describe this model's coefficients. A variable `x[j]` has nonzero coefficients of 1 in the constraints `complete[i]` for each `i in S[j]`, as well as a coefficient `c[j]` in the objective. The following AMPL alternative says this directly:

```
param n > 0;
param c {1..n} >= 0;

set U;
set S {1..n} within U;


minimize cost;


subject to complete {i in U}:
  to_come >= 1;

var x {j in 1..n} >= 0, <= 1,
  cover {i in S[j]} complete[i],
  obj cost c[j];
```

```
param n > 0;
param c {1..n} >= 0;

set U;
set S {1..n} within U;

var x {1..n} >= 0, <= 1;

minimize cost:
  sum {j in 1..n} c[j] * x[j];

subject to complete {i in U}:
  sum {j in 1..n: i in S[j]} x[j]>=1;
```

Figure 5–1. *AMPL formulations of the set-covering problem.* At left, the description uses the `cover` extension; at right, the covering constraints are specified in the usual algebraic way.

```
var x {j in 1..n} >= 0, <= 1,
     cover {i in S[j]} complete[i], obj cost c[j];
```

The full model is given alongside its algebraic equivalent in Figure 5–1.

As in the network models that use `node` and `arc`, the alternative set-covering model begins by declaring the objective and constraints:

```
minimize cost;
subject to complete {i in U}: to_come >= 1;
```

In the latter, the keyword `to_come` is a placeholder that works much like `net_in` or `net_out`; it shows where the linear expression in the variables is to be placed.

In the subsequent `var` declaration, the `cover` phrase specifies the nonzero coefficients. It has the forms

`cover` *constraint-name*
`cover {`*indexing*`}` *constraint-name*

The first form places a coefficient of 1 in the named constraint; it is the analogue of the previously discussed `to` and `from` phrases. The second form uses `{`*indexing*`}` to specify an entire indexed collection of constraints in which a coefficient of 1 is to be placed. Our example requires this feature, because the number of coefficients of `x[j]` is determined by the cardinality of `S[j]`, which varies according to the data supplied. (By contrast, a network model has just two coefficients per variable.)

To process this alternative AMPL formulation, the translator need only scan each subset `S[j]` once. This can represent a particularly great savings when `n` is large and each subset is just a small part of `U`.

5.2 Arbitrary column-wise formulations

The most significant common feature of our network examples (using `node` and `arc`) and our set-covering examples (using `cover`) is the specification of the coefficients within the declarations of the variables. In the argot of linear programming, these are column-wise formulations, since a variable's coefficients lie in one column of the constraint matrix. By contrast, a fully algebraic formulation is row-wise, specifying the coefficients within the declarations of the constraints.

Most efficient implementations of linear programming algorithms have used a data structure in which the coefficients are grouped by column. As a result, early "matrix generators" tended to encourage column-wise formulations. This view has persisted, out of habit or convenience, so that in certain applications it is still customary to view linear programs "by activity" rather than by constraint [14, 16].

Once the previously-described syntactic forms for networks and set covering have been defined, it is only a small step to provide for column-wise specification of coefficients in general. By analogy with the use of `net_in` and `net_out` in `node` declarations, we allow `subject to` declarations to employ the placeholder `to_come` within expressions of the form

> *variable-expression* `=` *arith-expression*
> *variable-expression* `<=` *arith-expression*
> *variable-expression* `>=` *arith-expression*
>
> *arith-expression* `=` *variable-expression*
> *arith-expression* `<=` *variable-expression*
> *arith-expression* `>=` *variable-expression*
>
> *arith-expression* `<=` *variable-expression* `<=` *arith-expression*
> *arith-expression* `>=` *variable-expression* `>=` *arith-expression*

where the *variable-expression* is any of

> `to_come`
> `to_come +` *arith-expression*
> *arith-expression* `+ to_come`

As in the simple case of the set covering constraint, the keyword `to_come` indicates where the linear terms defined column-wise will be added in.

In subsequent `var` declarations, we need only allow a `coef` phrase that combines the features of the `from/to` and `cover` phrases:

> `coef` *constraint-name value*
> `coef {`*indexing*`}` *constraint-name value*

The first form says that the variable has a coefficient with the given value in the named constraint. The second form is similar except that it defines a collection of coefficients, one for each member of the set specified by `{`*indexing*`}`.

Objectives are handled in an analogous way. A `minimize` or `maximize` declaration may use any *variable-expression* as above to specify an objective function, though the function consisting of `to_come` alone is most common and is assumed by default. In subsequent `var` declarations, `obj` phrases are used to specify variables'

```
set R;
set P;

param a {R,P} > 0;
param b {R} > 0;
param c {P} > 0;
param u {P} > 0;


maximize tot;


subject to supply {i in R}:
  to_come >= b[i];

var x {j in P} >= 0, <= u[j],
  coef {i in R} supply[i] a[i,j],
  obj tot c[j];
```

```
set R;
set P;

param a {R,P} > 0;
param b {R} > 0;
param c {P} > 0;
param u {P} > 0;

var x {j in P} >= 0, <= u[j];

maximize tot:
  sum {j in P} c[j] * x[j];

subject to supply {i in R}:
  sum {j in P} a[i,j] * x[j] >= b[i];
```

Figure 5–2. *Column-wise (left) and row-wise forms for a linear program in AMPL. This simple production model was introduced in Figure 1–1.*

```
set N;
set A within N cross N;
set P;

param b {N,P};
param c {A,P} >= 0;

param l {A,P} >= 0;
param u {(i,j) in A, p in P} >= l[i,j,p];
param u_mult {A} >= 0;

minimize total_cost;

node Balance {i in N, p in P}: net_out = b[i,p];

subject to Mult {(i,j) in A}: to_come <= u_mult[i,j];

arc Flow {(i,j) in A, p in P}:
     from Balance[i,p], to Balance[j,p],
     >= l[i,j,p], <= u[i,j,p],
     coef Mult[i,j] 1.0, obj total_cost c[i,j,p];
```

Figure 5–3. *An entirely column-wise formulation for the multicommodity flow model introduced in Figure 4–1.*

coefficients in the objectives. The syntax is exactly the same as for `coef` above, except for the initial keyword `obj`.

Figure 5–2 shows how the simple production model of Figure 1–1 can be expressed either row-wise or column-wise through different uses of `var`, `maximize`, and `subject to`. The `coef` phrase can also be used within an `arc` declaration, to specify the coefficients in side constraints; Figure 5–3 shows how this possibility may be used so as to rewrite the model of Figure 4–1 in an entirely column-wise manner.

6. Reflections and Conclusions

To this point, our presentation has concentrated on specific design decisions motivated by the extension of AMPL to several model structures. This concluding section considers some of the broader issues raised by our designs.

We argue that the structure extensions add valuable "redundancies" to the language, and permit the benefits of algebraic notation to be enjoyed even when the objective and constraints are not specified in a conventional algebraic way. We then summarize the various efficiencies that the extensions make possible in translation and solution of models.

6.1 Redundancy in the language design

In the preceding discussion of column-wise coefficient specification, we observed that an `obj` phrase places coefficients within objective functions in just the same way that a `coef` phrase places coefficients within constraints, using the same syntax except for the initial keyword. Evidently the `obj` phrase is redundant, in that the `coef` phrase could just as well serve the same purpose. AMPL enforces a distinction, however, for two reasons: to promote the readability of `var` declarations, and to allow for some additional error-checking. This approach is consistent with an earlier and more fundamental AMPL design decision, to syntactically distinguish the declarations of objectives from the declarations of constraints. It is possible to design a language that uses the same syntax for both, as in the case of GAMS [2].

The `cover` phrase is a similar case of redundancy. Its role is to provide a convenient, suggestive and reliable way of specifying coefficients equal to 1.

The special network syntaxes of Section 3 are also redundant in a sense. The `node` and `arc` declarations could be subsumed by `subject to` and `var`; the effect of the `from` and `to` phrases could be accomplished by `coef`; and the `net_in` and `net_out` keywords could be replaced by `to_come`. In this situation, however, the special syntax is particularly advantageous. It permits the AMPL model to more closely correspond to people's common notions of a network linear programming formulation. And it enables the translator to verify that every flow variable has at most two coefficients, observing the proper sign conventions.

As this discussion suggests, there are many redundancies built into AMPL, only some of which are introduced by the extensions described in this paper. Examples of others have been cited in our analysis of fundamental language features [8]. In each case, some degree of naturalness and error-checking has been achieved, at some loss of simplicity and generality. We have sought to limit redundancies to cases in which this tradeoff is reasonably favorable.

6.2 Benefits of an algebraic language

Algebraic notation and modeling languages are most often associated with a row-wise specification of the constraint coefficient matrix, simply because they are most often used to express a constraint-by-constraint formulation. The discussions in [14] and [16] are representative of this view. As our examples have shown, however, many of the features of an algebraic language are just as valuable in a column-wise specification.

AMPL's apparatus for describing data—in `set` and `parameter` declarations—is seen in Figures 3–2, 5–1 and 5–2 to provide equally good support for row-wise and column-wise constraint specifications. AMPL indexing expressions also prove to be equally useful whether they are indexing sums in an algebraic declaration, or constraint names in a `coef` phrase.

Within a `from`, `to`, `coef` or `obj` phrase, the coefficient value may be specified by a nontrivial algebraic expression, as in this example from the DIST model of Appendix A:

```
arc Manu_RT {p in prd, f in fact: rpc[p,f] <> 0} >= 0
     from P_RT[f] to M[p,f] (dp[f] * hd[f] / pt[p,f])
     obj cost (rpc[p,f] * dp[f] * hd[f] / pt[p,f]);
```

Column-wise specifications can make extensive use of an algebraic language in this way, even though they do not describe any complete objective or constraint algebraically.

Finally, an algebraic modeling language can accommodate convenient hybrids of column-wise and row-wise specifications. Figure 4–1 offers an example in a model of a network with side constraints. As another possibility, a model could have a nonlinear objective function expressed algebraically in a `minimize` or `maximize` declaration, but subject to network constraints specified by means of `node` and `arc` declarations.

6.3 Efficiency of translation

We previously remarked that the AMPL language translator should be able to process certain set covering models much faster when they are specified column-wise, by use of the `cover` phrase. Table 6–1 presents evidence of this advantage; in a randomly constructed 100 by 25000 example, the speedup is by a factor of about 9.

Other tests reported in Table 6–1 show that the special syntaxes for network and piecewise-linear structures also permit a speedup, though to a much more modest degree. In the network examples, both alternatives generate the same linear program, but the translator can process the `node` and `arc` declarations somewhat more efficiently than the corresponding `var` and `subject to` declarations. In the piecewise-linear examples, the advantage is due to the P-L formulation's smaller number of variables and coefficients than the equivalent LP formulation.

6.4 Efficiency of solution

Previous investigations have established that network and piecewise-linear models can be solved faster by methods that take advantage of their special structures. When AMPL is used to formulate models for these methods, its special structure syntax can make possible some additional efficiencies.

For an algorithm to take advantage of structure in a linear program, it must receive input indicating where the structure is present. If piecewise-linear functions

	COVER	DIST	SCORE	STRUC	TRANS
Without extensions	92.20	1.38	11.23	0.87	14.35
With extensions	10.37	1.18	8.80	0.77	7.90

Table 6–1. *Efficiencies afforded by the AMPL structure extensions.* Timings are for the "genmod" phase of the AMPL translator, on a Sun SPARCstation 2; other phases (as described in [8]) were much more nearly the same. The test problems are as follows:

COVER: The set covering model of Figure 5–1 with randomly generated data. There are 100 members in the set `U`, and 25000 subsets of one to five members each.

DIST: The network model of Appendix A with a realistic data file; similar to test problem SHIP12L in the netlib set [12]. There are 1268 nodes and 2295 arcs.

SCORE: The piecewise-linear model of Appendix B with a realistic data file; equivalent to test problem FIT2P in the netlib set [12]. The piecewise-linear version has 3000 constraints in 3025 variables, with 4 or 5 linear pieces per variable.

STRUC: The piecewise-linear model of Appendix C with a realistic data file; equivalent to test problem SCSD8 in the netlib set [12]. The piecewise-linear version has 397 constraints in 1375 variables.

TRANS: The network model of Figure 3–2 with randomly generated data. There are 5000 nodes and 50455 arcs.

or network nodes and arcs are declared explicitly, then an AMPL translator can automatically generate the input that a specialized algorithm requires. If ordinary algebraic declarations are employed, however, then the algorithm must incorporate a pre-processing stage in order to make the structure explicit for its purposes.

Previous studies have described pre-processors both for network constraints [17] and for piecewise-linear objective terms [11]. These routines necessarily involve some extra cost, but they are usually cheap compared to the overall cost of translation and solution. A greater source of difficulty lies in the possibility that, due to some error in the formulation, the generated model may fail to exhibit the expected structure. An intended network flow model might accidentally be written, for example, in a way that causes some variables to have three nonzero coefficients rather than the required two. A pre-processor must be relied upon to detect any such error and to communicate it in a useful way to the modeler. By constrast, no comparable error handling is necessary when the structure is already explicit in the AMPL model.

There are special algorithms for other kinds of structures in linear programming. In what cases might similar benefits be realized? Our observations suggest that a structure is a likely candidate if it is well known to modelers, and if it tends to simplify a formulation when its presence is made explicit. Introduction of a new syntax for such a structure can offer benefits in the formulation and translation of a model as well as in its solution.

Appendix A. DIST, a generalized network distribution model

This generalized network model determines a production and distribution plan to meet given demands for a set of goods. An equivalent model, employing a somewhat different formulation, is described by use of algebraic constraints in [7].

```
###  SHIPPING SETS AND PARAMETERS  ###

set whse 'warehouses';  # Locations from which demand is satisfied

set dctr 'distribution centers' within whse;
                        # Locations from which product may be shipped

param sc 'shipping cost' {dctr,whse} >= 0;
                        # Shipping costs, to whse from dctr, in $ / 100 lb

param huge 'largest shipping cost' > 0;
                        # Largest cost allowed for a usable shipping route

param msr 'minimum size restriction' {dctr,whse} logical;
                        # True indicates a minimum-size restriction on
                        # direct shipments using this dctr --> whse route

param dsr 'direct shipment requirement' {dctr} >= 0;
                        # Minimum total demand, in pallets, needed to
                        # allow shipment on routes subject to the
                        # minimum size restriction

###  PLANT SETS AND PARAMETERS  ###

set fact 'factories' within dctr;
                        # Locations where product is manufactured

param rtmin 'regular-time total minimum' >= 0;
                        # Lower limit on (average) total regular-time
                        # crews employed at all factories

param rtmax 'regular-time total maximum' >= rtmin;
                        # Upper limit on (average) total regular-time
                        # crews employed at all factories

param otmin 'overtime total minimum' >= 0;
                        # Lower limit on total overtime hours at all factories

param otmax 'overtime total maximum' >= otmin;
                        # Upper limit on total overtime hours at all factories

param rmin 'regular-time minimums' {fact} >= 0;
                        # Lower limits on (average) regular-time crews

param rmax 'regular-time maximums' {f in fact} >= rmin[f];
                        # Upper limits on (average) regular-time crews

param omin 'overtime minimums' {fact} >= 0;
                        # Lower limits on overtime hours

param omax 'overtime maximums' {f in fact} >= omin[f];
                        # Upper limits on overtime hours

param hd 'hours per day' {fact} >= 0;
                        # Regular-time hours per working day

param dp 'days in period' {fact} > 0;
                        # Working days in the current planning period
```

```
###  PRODUCT SETS AND PARAMETERS  ###

set prd 'products';     # Elements of the product group

param wt 'weight' {prd} > 0;
                        # Weight in 100 lb / 1000 cases

param cpp 'cases per pallet' {prd} > 0;
                        # Cases of product per shipping pallet

param tc 'transshipment cost' {prd} >= 0;
                        # Transshipment cost in $ / 1000 cases

param pt 'production time' {prd,fact} >= 0;
                        # Crew-hours to produce 1000 cases

param rpc 'regular-time production cost' {prd,fact} >= 0;
                        # Cost of production on regular time,
                        # in $ / 1000 cases

param opc 'overtime production cost' {prd,fact} >= 0;
                        # Cost of production on overtime, in $ / 1000 cases

###  DEMAND SETS AND PARAMETERS  ###

param dt 'total demand' {prd} >= 0;
                        # Total demands for products, in 1000s

param ds 'demand shares' {prd,whse} >= 0.0, <= 1.0;
                        # Historical demand data, from which each
                        # warehouse's share of total demand is deduced

param dstot {p in prd} := sum {w in whse} ds[p,w];
                        # Total of demand shares; should be 1, but often isn't

param dem 'demand' {p in prd, w in whse} := dt[p] * ds[p,w] / dstot[p];
                        # Projected demands to be satisfied, in 1000s

set rt 'shipping routes available' :=
 {d in dctr, w in whse:
        d <> w  and  sc[d,w] < huge  and
        (w in dctr or sum {p in prd} dem[p,w] > 0)  and
        not (msr[d,w] and sum {p in prd} 1000*dem[p,w]/cpp[p] < dsr[d]) };
                        # List of ordered pairs that represent routes
                        # on which shipments are allowed

###  OBJECTIVE  ###

minimize cost;          # Total cost:  regular production, overtime
                        # production, shipping, and transshipment

###  NODES  ###

node RT:  rtmin <= net_out <= rtmax;
                        # Source of all regular-time crews allocated

node OT:  otmin <= net_out <= otmax;
                        # Source of all overtime hours allocated

node P_RT {fact};       # Sources of regular-time crews at factories

node P_OT {fact};       # Sources of overtime hours at factories
```

```
node M {prd,fact};      # Sources of manufacturing:
                        # send to factory's W node for local demand;
                        # send to factory's D node for distribution

node D {prd,dctr};      # Sources of distribution:
                        # receive transshipped goods from center's W node;
                        # receive manufactured goods from center's M node;
                        # send to W nodes elsewhere

node W {p in prd, w in whse}:  net_in = dem[p,w];
                        # Locations of warehousing:
                        # receive from D nodes and local M node (if any),
                        # to satisfy local demand;
                        # send to local D node (if any) for transshipment

###  ARCS  ###

arc Work_RT {f in fact}
      from RT  to P_RT[f]  >= rmin[f],  <= rmax[f];
                        # Regular-time crews allocated to each factory

arc Work_OT {f in fact}
      from OT  to P_OT[f]  >= omin[f],  <= omax[f];
                        # Overtime hours allocated to each factory

arc Manu_RT {p in prd, f in fact: rpc[p,f] <> 0} >= 0
      from P_RT[f]  to M[p,f] (dp[f] * hd[f] / pt[p,f])
      obj cost (rpc[p,f] * dp[f] * hd[f] / pt[p,f]);
                        # Regular-time crews allocated to
                        # manufacture of each product at each factory

arc Manu_OT {p in prd, f in fact: opc[p,f] <> 0} >= 0
      from P_OT[f]  to M[p,f] (1 / pt[p,f])  obj cost (opc[p,f] / pt[p,f]);
                        # Overtime hours allocated to
                        # manufacture of each product at each factory

arc Prod_L {p in prd, f in fact} >= 0
      from M[p,f]  to W[p,f];
                        # Manufacture of each product at each factory
                        # to satisfy local demand, in 1000s of units

arc Prod_D {p in prd, f in fact} >= 0
      from M[p,f]  to D[p,f];
                        # Manufacture of each product at each factory,
                        # for distribution elsewhere, in 1000s of units

arc Ship {p in prd, (d,w) in rt} >= 0
      from D[p,d]  to W[p,w] obj cost (sc[d,w] * wt[p]);
                        # Shipments of each product on each allowed route

arc Trans {p in prd, d in dctr} >= 0
      from W[p,d]  to D[p,d]  obj cost (tc[p]);
                        # Transshipments of each product at each
                        # distribution center
```

Appendix B. SCORE, a piecewise-linear data fitting model

This model fits linear inequalities to data, by minimizing a sum of convex piecewise-linear penalties on the deviations from an acceptable fit. The motivation comes from a problem of weighting certain customer attributes so as to distinguish "good" from "bad" credit risks.

```
###  DATA ON PEOPLE  ###

set Good;                              # Good risks
set Bad;                               # Bad risks

set people := Good union Bad;          # Everyone is either good or bad

param app_amt > 0;                     # General credit-approval amount
param bal_amt {people} >= app_amt;     # Maximum-balance amounts of individuals

###  DATA ON FACTORS  ###

set factors;                           # Questions posed to individuals

set wt_types := {'pos','neg','free'};  # Required signs of weights
param wttyp {factors} symbolic within wt_types;

param answer {people,factors} >= 0;    # Numerical responses to all questions

###  DATA DEFINING THE PENALTY FUNCTION  ###
# Parameters starting with G (or B) are for the good (or bad) risks

param Gpce > 1;
param Bpce > 1;                        # Linear pieces in penalty term

param Gslope {1..Gpce};  check {k in 1..Gpce-1}: Gslope[k] < Gslope[k+1];
param Bslope {1..Bpce};  check {k in 1..Bpce-1}: Bslope[k] < Bslope[k+1];
                                       # Increasing slopes in penalty term

set bkpt_types := {'A','B'};
param Gbktyp {1..Gpce-1} symbolic within bkpt_types;
param Bbktyp {1..Bpce-1} symbolic within bkpt_types;
param Gbkfac {1..Gpce-1};  check {k in 1..Gpce-2}: Gbkfac[k] <= Gbkfac[k+1];
param Bbkfac {1..Bpce-1};  check {k in 1..Bpce-2}: Bbkfac[k] <= Bbkfac[k+1];
                                       # Information to define the
                                       # increasing breakpoints in penalty
                                       # terms (see objective function)

param Gprop > 0;                       # Scale objective to simulate ratio
param Bprop > 0;                       # Gprop/Bprop of goods to bads

param Gratio := (Gprop/(Gprop+Bprop)) / (card {Good}/card {people});
param Bratio := (Bprop/(Gprop+Bprop)) / (card {Bad}/card {people});

###  VARIABLES  ###

var Wt_const;                          # Constant term in computing all scores

var Wt {j in factors} >= if wttyp[j] = 'pos' then 0 else -Infinity
                      <= if wttyp[j] = 'neg' then 0 else +Infinity;
                                       # Weights on the factors

var Sc {i in people};                  # Scores for the individuals
```

```
### OBJECTIVE ###

minimize penalty:                     # Sum of penalties for all individuals
   Gratio * sum {i in Good} << {k in 1..Gpce-1} if Gbktyp[k] = 'A'
                                   then Gbkfac[k]*app_amt
                                   else Gbkfac[k]*bal_amt[i];
                               {k in 1..Gpce} Gslope[k] >> Sc[i] +
   Bratio * sum {i in Bad}  << {k in 1..Bpce-1} if Bbktyp[k] = 'A'
                                   then Bbkfac[k]*app_amt
                                   else Bbkfac[k]*bal_amt[i];
                               {k in 1..Bpce} Bslope[k] >> Sc[i];

### CONSTRAINTS ###

def_Sc {i in people}:
   Sc[i] = Wt_const + sum {j in factors} answer[i,j] * Wt[j];
                                      # Score = sum of answers times weights
```

Appendix C. STRUC, a piecewise-linear structural design model

Given a set of admissible joints and a set of admissible bars connecting the joints, this model determines bar widths that minimize the total weight of the structure while maintaining an equilibrium with external loads [4, 13]. Each term of the objective is the length of a bar times the absolute value of the force (positive for tension, negative for compression) on the bar.

```
###  DATA  ###

set joints;
set bars within {i in joints, j in joints: i <> j};
                                     # Definition of admissible structure:
                                     # each bar connects two joints

param fixed symbolic in joints;      # Designated position of fixed support
param rolling symbolic in joints;    # Designated position of roller support

param density > 0;                   # Density of bar material
param yield_stress > 0;              # Yield stress of bar material

param xpos {joints};                 # Horizontal positions of joints
param ypos {joints};                 # Vertical positions of joints
   check {(i,j) in bars}: xpos[i] <> xpos[j] or ypos[i] <> ypos[j];

param xload {joints};                # Horizontal external loads on joints
param yload {joints};                # Vertical external loads on joints

param length {(i,j) in bars} :=
                      sqrt ((xpos[j]-xpos[i])^2 + (ypos[j]-ypos[i])^2);
                                     # Bar lengths calculated from positions

param xcos {(i,j) in bars} := (xpos[j]-xpos[i]) / length[i,j];
param ycos {(i,j) in bars} := (ypos[j]-ypos[i]) / length[i,j];
                                     # Cosines of bar angles with
                                     # horizontal and vertical axes


###  VARIABLES  ###

var Force {bars};                    # Forces on bars:
                                     # positive in tension, negative in compression


###  OBJECTIVE  ###

minimize weight:  (density / yield_stress) *
   sum {(i,j) in bars} length[i,j] * <<0; -1,+1>> Force[i,j];
                                     # Weight is proportional to length
                                     # times absoluted value of force


###  CONSTRAINTS  ###

subject to xbal {k in joints: k <> fixed}:
     sum {(i,k) in bars} xcos[i,k] * Force[i,k]
   - sum {(k,j) in bars} xcos[k,j] * Force[k,j] = xload[k];

subject to ybal {k in joints: k <> fixed and k <> rolling}:
     sum {(i,k) in bars} ycos[i,k] * Force[i,k]
   - sum {(k,j) in bars} ycos[k,j] * Force[k,j] = yload[k];
                                     # Net sum of forces must balance external
                                     # load, horizontally and vertically
```

Appendix D. TRAIN, a network allocation model

Given a day's schedule, this network flow model allocates passenger cars to trains so as to minimize either the number of cars required or the number of car-miles run [9, 10]. The same model is formulated in [7] using algebraic constraints.

```
###  SCHEDULE SETS AND PARAMETERS  ###

set cities;
set links within {c1 in cities, c2 in cities: c1 <> c2};
                        # Set of cities, and set of intercity links

param last > 0 integer; # Number of time intervals in a day
set times := 1..last;   # Set of time intervals in a day

set schedule within
      {c1 in cities, t1 in times,
       c2 in cities, t2 in times: (c1,c2) in links};
                        # Member (c1,t1,c2,t2) of this set represents
                        # a train that leaves city c1 at time t1
                        # and arrives in city c2 at time t2

###  DEMAND PARAMETERS  ###

param section > 0 integer;
                        # Maximum number of cars in one section of a train

param demand {schedule} > 0;
                        # For each scheduled train:
                        # the smallest number of cars that
                        # can meet demand for the train

param low {(c1,t1,c2,t2) in schedule} := ceil(demand[c1,t1,c2,t2]);
                        # Minimum number of cars needed to meet demand

param high {(c1,t1,c2,t2) in schedule}
   := max (2, min (ceil(2*demand[c1,t1,c2,t2]),
                   section*ceil(demand[c1,t1,c2,t2]/section) ));
                        # Maximum number of cars allowed on a train:
                        # 2 if demand is for less than one car;
                        # otherwise, lesser of
                        # number of cars needed to hold twice the demand, and
                        # number of cars in minimum number of sections needed

###  DISTANCE PARAMETERS  ###

param dist_table {links} >= 0 default 0.0;
param distance {(c1,c2) in links} > 0
   := if dist_table[c1,c2] > 0 then dist_table[c1,c2] else dist_table[c2,c1];
                        # Inter-city distances: distance[c1,c2] is miles
                        # between city c1 and city c2

###  OBJECTIVES  ###

minimize cars;          # Number of cars in the system:
                        # sum of unused cars and cars in trains during
                        # the last time interval of the day

minimize miles;         # Total car-miles run by all scheduled trains in a day
```

```
###  NODES  ###

node N {cities,times};  # For every city and time:
                        # unused cars in present interval will equal
                        # unused cars in previous interval,
                        # plus cars just arriving in trains,
                        # minus cars just leaving in trains

###  ARCS  ###

arc U {c in cities, t in times} >= 0
      from N[c,t]  to N[c, if t < last then t+1 else 1]
      obj cars (if t = last then 1 else 0);
                        # U[c,t] is the number of unused cars stored
                        # at city c in the interval beginning at time t

arc X {(c1,t1,c2,t2) in schedule}
      >= low[c1,t1,c2,t2]  <= high[c1,t1,c2,t2]
      from N[c1,t1]  to N[c2,t2]
      obj cars (if t2 < t1 then 1 else 0)
      obj miles distance[c1,c2];
                        # X[c1,t1,c2,t2] is the number of cars assigned
                        # to the scheduled train that leaves c1 at t1
                        # and arrives in c2 at t2
                        # The bounds insure that the cars meet demand,
                        # but that they are not so far in excess of demand
                        # that unnecessary sections are required
```

References

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

[2] A. Brooke, D. Kendrick and A. Meeraus, 1988. *GAMS: A User's Guide.* Scientific Press, Redwood City, CA.

[3] V. Chvátal, 1983. *Linear Programming.* W.H. Freeman and Company, New York.

[4] W.S. Dorn, R.E. Gomory and H.J. Greenberg, 1964. Automatic Design of Optimal Structures. *Journal de Mécanique* **3,** 25–52.

[5] R. Fourer, 1983. Modeling Languages versus Matrix Generators for Linear Programming. *ACM Transactions on Mathematical Software* **9,** 143–183.

[6] R. Fourer, 1986. A Simplex Algorithm for Piecewise-Linear Programming III: Computational Analysis and Applications. Technical Report 86-03, Department of Industrial Engineering and Management Sciences, Northwestern University, Evanston, IL; also to appear in *Mathematical Programming.*

[7] R. Fourer, D.M. Gay and B.W. Kernighan, 1987. AMPL: A Mathematical Programming Language. Computing Science Technical Report 133, AT&T Bell Laboratories, Murray Hill, NJ; also Technical Report 87-03, Department of Industrial Engineering and Management Sciences, Northwestern University, Evanston, IL.

[8] R. Fourer, D.M. Gay and B.W. Kernighan, 1990. A Modeling Language for Mathematical Programming. *Management Science* **36,** 519–554.

[9] R. Fourer, J.B. Gertler and H.J. Simkowitz, 1977. Models of Railroad Passenger-Car Requirements in the Northeast Corridor. *Annals of Economic and Social Measurement* **6,** 367–398.

[10] R. Fourer, J.B. Gertler and H.J. Simkowitz, 1978. Optimal Fleet Sizing and Allocation for Improved Rail Service in the Northeast Corridor. *Transportation Research Record* **656,** 40–45.

[11] R. Fourer and R.E. Marsten, 1990. Solving Piecewise-Linear Programs: Experiments with a Simplex Approach. Technical Report 90-07, Department of Industrial Engineering and Management Sciences, Northwestern University, Evanston, IL; also to appear in *ORSA Journal on Computing.*

[12] D.M. Gay, 1985. Electronic Mail Distribution of Linear Programming Test Problems. *Committee on Algorithms Newsletter* **13,** 10–12. Also Numerical Analysis Manuscript 86-0, AT&T Bell Laboratories, Murray Hill, NJ.

[13] J.K. Ho, 1975. Optimal Design of Multi-Stage Structures: A Nested Decomposition Approach. *Computers & Structures* **5,** 249–255.

[14] G. Kahan, 1982. Walking through a Columnar Approach to Linear Programming of a Business. *Interfaces* **12:**3, 32–39.

[15] R.V. Simons, 1987. Mathematical Programming Modeling Using MGG. *IMA Journal of Mathematics in Management* **1,** 267–276.

[16] J.S. Welch, Jr., 1987. PAM—A Practitioner's Approach to Modeling. *Management Science* **33,** 610–625.

[17] S.A. Zenios, 1990. Integrating Network Optimization Capabilities into a High-Level Modeling Language. *ACM Transactions on Mathematical Software* **16,** 113–142.