EXTENDING A GENERAL-PURPOSE ALGEBRAIC MODELING LANGUAGE TO COMBINATORIAL OPTIMIZATION: A LOGIC PROGRAMMING APPROACH

Robert Fourer

Department of Industrial Engineering and Management Sciences
Northwestern University, Evanston, IL 60208-3119, USA
4er@iems.nwu.edu

General-purpose *algebraic modeling languages* are a central feature of popular computer systems for large-scale optimization. Languages such as AIMMS [2], AMPL [12, 13], GAMS [4, 5], LINGO [23] and MPL [18] allow people to develop and maintain diverse optimization models in their natural mathematical forms. The systems that process these languages convert automatically to and from the various data structures required by packages of optimizing algorithms ("solvers"), with only minimal assistance from users. Most phases of language translation remain independent of solver details, however, so that users can easily switch between many combinations of language and solver.

Algebraic modeling languages have been applied most successfully in linear and smooth nonlinear optimization. They have been notably less successful in combinatorial or discrete optimization, however, for two interconnected reasons.

First, modeling languages have lacked the kinds of expressions that are needed to describe combinatorial problems in natural and convenient ways. Indeed, only one feature of these languages has been of direct use for combinatorial optimization: the option to specify that certain variables must take integer values. Hence these languages have been useful mainly for combinatorial problems that have straightforward formulations as integer linear programs. While in principle any combinatorial optimization problem can be expressed as an integer program, in practice a variety of problem-specific formulation tricks are often required to carry out the transformation. Much of the advantage of a modeling language for formulation and maintenance of models is lost as a result.

Second, modeling languages have lacked access to suitable algorithms for combinatorial optimization. The only applicable *general-purpose* methods commonly interfaced to algebraic modeling languages have been integer programming branch-and-bound schemes. Although automatic transformations to integer programs have been worked out for significant classes of combinatorial problems [15, 21, 22], branch-and-bound codes do not necessarily perform well on the resulting formulations. At the same time, research in combinatorial optimization has concentrated on *special-purpose* algorithms for which explicit performance bounds can be derived. Because these algorithms are focused on particular problem types, they do not address many of the complex combinatorial problems that are of practical interest. Even where they do apply, modeling software lacks efficient and reliable ways of deducing their applicability based on formulations expressed in modeling languages.

General-purpose software suitable for combinatorial optimization has in fact a substantial history, going back to Lauriere's ALICE [19]. Considerable development effort has led to successful current implementations of several kinds:

- Descendants of Prolog [26] that handle objectives and constraints, including CHIP [28], ECLiPSe [8] and Prolog III [6].
- Specialized C-like languages, such as 2LP [20].
- Specialized class libraries for C++, notably ILOG SOLVER [16, 24].

The modeling approaches and algorithmic methods employed by such systems are commonly referred to as *logic programming*, or as *constraint logic programming* to distinguish them from earlier systems more narrowly focused on logical inference. All of these systems incorporate computer languages that are algebraic to some degree, particularly in their descriptions of specific constraint expressions. They tend to differ substantially from the systems familiar to users of algebraic modeling languages for linear or integer programming, however, especially in two respects:

- Reflecting their origins (in Prolog, C or C++), their descriptions of complete optimization models depart in significant ways from the formulations that people most often use.
- Their implementations rely on specialized solvers, allowing users limited flexibility to choose language and solver independently.

These systems have nevertheless been successful in introducing constraint forms and algorithmic strategies that are natural and desirable for practical combinatorial optimization. For some difficult applications in scheduling, sequencing, location and assignment, a constraint logic programming approach has been reported to be superior to an integer programming approach of the sort that would be available through current modeling languages [7, 17, 25].

In sum, there have been a range of obstacles to development of fully effective general-purpose modeling languages for combinatorial optimization, based on either integer programming or logic programming. These obstacles have tended moreover to reinforce each other. Developers of algebraic modeling languages, believing that there are no efficient and reliable general-purpose solvers for combinatorial problems beyond integer programming branch-and-bound, have made little attempt to extend their languages to better express such problems. Developers of constraint logic programming systems, observing that algebraic modeling languages are limited to integer programs, have made little attempt to connect their solvers to existing modeling languages. The purpose of this paper is to suggest, by analysis and by example, how we are now in a much improved position to overcome this deadlock.

The first part of this paper (sections 1–3) explores combinatorial extensions to the AMPL modeling language [13], with the goal of showing that such a language can provide the expressiveness desired by builders of combinatorial optimization models. AMPL's broad range of forms for set and arithmetic expressions are particularly well suited to a variety of extensions. In many cases, it suffices to define new operators by analogy with the syntactic forms of current ones, or to extend the range of contexts in which variables are permitted. Further advantage is gained by extending the concept of a variable to permit values that are arbitrary objects or sets of objects.

The second part of this paper (sections 4–5) considers how ILOG SOLVER [16] could provide the general-purpose solver required by the first part's language extensions. The major focus of SOLVER has been to serve as a C++ class library for embedding constraint logic programming techniques in users' applications. As its range of applications has expanded, it has accumulated a variety of constraint and expression forms, which have turned out to correspond nicely to the features that would be needed to provide a general-purpose algorithmic framework for the AMPL combinatorial extensions. Thus SOLVER provides an empirical confirmation of the proposed extensions' practical value, while also demonstrating that a general-purpose optimizer for combinatorial optimization is a reasonable possibility.

An implementation of a link from integer programs in AMPL to the optimization procedures in SOLVER is described in section 4, and specific SOLVER C++ classes that would correspond to the proposed AMPL combinatorial extensions are identified in section 5. Thus this paper lays the groundwork for the actual implementation of combinatorial features in an algebraic modeling language. Section 6 concludes by discussing some of the challenges that are likely to be faced in any initial implementation and the refinements that are likely to follow.

The numerous examples throughout this paper are presented with the intention of being understandable to all readers, including those who are not familiar with the AMPL language. Many of the examples come from complete models that are identified in the text by filenames (such as `sched1.mod`). The corresponding files are listed in an appendix, and can also be found online at `http://www.iems.nwu.edu/~4er/LOGLANG/`.

1 EXTENDING AN ALGEBRAIC MODELING LANGUAGE

Extending a computer language tends to make it more powerful, but also more complicated. Thus the first aim of this paper is not merely to show that an algebraic modeling language *can* be extended to better express combinatorial optimization problems. The goal is to show that combinatorial extensions can be made without giving up the established convenience and power of the language's existing features.

A closer consideration of this goal motivates design principles that are in fact common to most modeling language extensions. First, an extension does need to offer some value to the user, and the principle involved is fairly obvious:

- ***Applicability.*** An extension should provide a concise and natural way to express some aspect of a formulation that arises in applications of optimization modeling.

This principle is actually most notable for what it does not say. It does not specify that extensions should be useful only for entirely new kinds of constraints, or that extensions should provide a minimal collection of language changes sufficient to express models of some kind. To the contrary, successful modeling languages incorporate considerable redundancy of features, because they are intended to express models in ways that are natural to people — and different people frequently have different ideas as to what is natural. A discussion of redundancy in some previous extensions to the AMPL language can be found in [11].

The extensions to be proposed in sections 2 and 3 have thus been derived empirically, by considering how people would like to be able to express a variety of combinatorial optimization models. The principle of applicability accepts any extension derived in this way, while other principles narrow the possibilities by imposing additional requirements.

Two other principles apply specifically to the relationships between extensions and the existing forms of a language:

- ***Independence.*** New extensions should be clearly differentiated from existing language forms. Users should not be required to be aware of the extensions unless they are formulating the kinds of problems for which the extensions were designed.
- ***Consistency.*** New extensions should be based on the same principles as existing language forms. Current users of the language should find the extensions to be familiar.

These requirements are intended to prevent the language from becoming overly extensive, to the point where it is no longer attractive to its original users.

As an example, AMPL [13] provides an iterated summation operator that consists of a keyword (`sum`), an expression for an indexing set (delimited by `{` and `}`), and an expression to be summed over the set:

```
sum {i in ORIG, j in DEST: cost[i,j] < BIG} cost[i,j] * Trans[i,j]
```

Because this operator is repeatedly used in formulating linear programs, an extension cannot modify it without violating the independence principle. On the other hand, we would not want to define an entirely new syntax for the iterated operations needed in combinatorial optimization, as that would almost surely violate consistency. The preferred approach is to retain the form of the `sum` operator while altering a few of its specifics, such as by introducing a new keyword and changing the type of expression that follows the indexing.

These principles are not intended to be absolute, and indeed will have to be relaxed in some cases to achieve the desired expressiveness of the extended language. For instance, a further extension to add a parenthesized argument after an iterated operator keyword may also be desirable (as Section 2 will explain), although a degree of consistency is given up as a result.

Two universal principles of modeling language design are also particularly relevant to proposed extensions:

- ***Generality.*** An expression that would be meaningful to most users should not be ruled out by complex or arbitrary restrictions.
- ***Implementability.*** An expression must be translatable with reasonable efficiency into the forms required by suitable algorithms.

Since greater generality tends to complicate implementation, these requirements tend to trade off against each other.

To continue the preceding example, imagine that `Trans[i,j]` are variables representing shipments, and suppose that we want to extend the AMPL `sum` operator to support a distinction between fixed and varying shipment costs. By allowing variables to appear in a comparison *within* an indexing expression, we could provide a convenient way of expressing total cost as the sum of a fixed cost `fcost[i,j]` and a varying cost `vcost[i,j] * Trans[i,j]` for each pair of indices `i` and `j` such that shipment is positive:

```
sum {i in ORIG, j in DEST: Trans[i,j] >= fuzz}
   (fcost[i,j] + vcost[i,j] * Trans[i,j])
```

Generality would then suggest that the comparison within the indexing expression should be extended by allowing variables to appear at any place where constants (or parameters, in AMPL terminology) are currently allowed. The resulting design might not be implementable, however, due to insurmountable difficulties in translating some of the resulting comparisons to a form usable by a general-purpose solver. We could instead try to insure implementability by placing greater restrictions on the ways variables are allowed to appear inside an indexing expression; but then the design might tend to violate generality, since it is difficult to come up with any simple rule in this case to distinguish the permitted expressions from the prohibited ones.

A better alternative might thus be to use AMPL's `if` ... `then` ... `else` operator with variables in the expression after the `if`. Then fixed costs would still be easy to express:

```
sum {i in ORIG, j in DEST}
   (if Trans[i,j] >= fuzz
      then fcost[i,j] + vcost[i,j] * Trans[i,j] else 0.0)
```

Implementation would be more practical, because this expression specifies unambiguously, for given `i` and `j`, the value to be taken both when the condition following `if` is true and when it is false. Because this is a restriction inherent in the way that that the `if` operator is currently defined in AMPL, the principle of generality is maintained.

The next section covers extensions that involve adding operators or extending the applicability of operators. Section 3 then describes extensions that permit variables to take objects or sets as values.

2 EXTENSIONS TO OPERATORS

Many valuable modeling language extensions for combinatorial optimization can be achieved by expanding the uses and variety of operators. In some cases, the extensions merely allow existing operators to be applied more generally. Other extensions involve the definition of new iterated operators analogous to current ones.

Logical operators

Logical constraints may be built from simpler constraints using the standard boolean operators "and", "or" and "not". In the context of algebraic modeling languages, we can consider in particular the constraints that result when these operators are applied to conventional algebraic constraints.

The best known and most common constraints of this kind are *disjunctive* constraints, which say that at least one among several algebraic constraints must hold. As a simple example, in a flowshop problem we require that the start times of two jobs be sufficiently separated to prevent them from being scheduled for production at the same time. In algebraic terms, this reduces to saying that for any pair of jobs `i1` and `i2`, either

```
Start[i2] >= Start[i1] + t_offset[i1,i2]
```

or

```
Start[i1] >= Start[i2] + t_offset[i2,i1]
```

The standard approach to formulating this discrete restriction as an integer program (`flowshp0.mod`) involves defining a zero-one variable `Precedes[i1,i2]` and a sufficiently large parameter `M`:

```
subj to No12_Conflict {i1 in JOBS, i2 in JOBS: ord(i1) < ord(i2)}:
   Start[i2] >= Start[i1] +
      t_offset[i1,i2] - M * (1 - Precedes[i1,i2]);

subj to No21_Conflict {i1 in JOBS, i2 in JOBS: ord(i1) < ord(i2)}:
   Start[i2] >= Start[i1] +
      t_offset[i2,i1] - M * Precedes[i1,i2];
```

Yet the same thing could be said much more clearly and economically by representing the disjunctive form of the constraint directly (`flowshp1.mod`):

```
subj to No_Conflict {i1 in JOBS, i2 in JOBS: ord(i1) < ord(i2)}:
   Start[i2] >= Start[i1] + t_offset[i1,i2] or
   Start[i1] >= Start[i2] + t_offset[i2,i1];
```

The `or` operator is already a part of AMPL, but its use is restricted to conditions involving only sets and parameters (such as appear within indexing expressions). The extension here is simply to permit the use of `or` in constraint declarations.

The previously stated generality principle suggests that if we allow the `or` operator in constraints, then we should similarly allow AMPL's other boolean operators. To provide an example, here is how one real-world assignment model uses additional binary variables and constraints to enforce a requirement that prevents any person from being "isolated" in the solution. As specified in the initial part of the model (not shown here), there are `number[i1,i2]` people of rank `i1` from site `i2`, and there is a set `ROOM` of meeting rooms; each person must be assigned to exactly one room. The set `TYPE` contains every distinct "type" of person: every pair `(i1,i2)` that represents the rank and site of at least one person to be assigned. The main decision variables are declared by

```
var Assign {(i1,i2) in TYPE, j in ROOM} integer >= 0;
```

`Assign[i1,i2,j]` is the number of people of type `(i1,i2)` assigned to room `j`.

The constraints to rule out "isolation" employ some auxiliary zero-one variables:

```
var Any {(i1,i2) in TYPE, j in ROOM} binary;
```

The intent is that `Any[i1,i2,j]` will take the value one if there are any people of type `(i1,i2)` assigned to room `j`, and zero otherwise. Upper bounds on the values of the variables Assign[i1,i2,j] are also calculated, in terms of previously defined parameters:

```
param upperbnd {(i1,i2) in TYPE, j in ROOM} := min (
   ceil ((number[i1,i2]/card PEOPLE) * hiDine[j]) + give[i1,i2],
   hiTargetTitle[i1,j] + giveTitle[i1],
   hiTargetLoc[i2,j] + giveLoc[i2],
   number[i1,i2] );
```

Then three collections of constraints jointly specify the non-isolation requirement:

```
subj to Isolation0 {(i1,i2) in TYPE, j in ROOM}:
   Assign[i1,i2,j] <= upperbnd[i1,i2,j] * Any[i1,i2,j];

subj to Isolation1a {(i1,i2) in TYPE, j in ROOM}:
   Assign[i1,i2,j] >= Any[i1,i2,j];

subj to Isolation1b {(i1,i2) in TYPE, j in ROOM}:
   Assign[i1,i2,j] +
      sum {ii1 in ADJ[i1]: (ii1,i2) in TYPE} Assign[ii1,i2,j]
         >= 2 * Any[i1,i2,j];
```

When Any[i1,i2,j] is zero, constraint Isolation0[i1,i2,j] says no person of type (i1,i2) may be assigned to room j, while Isolation1a[i1,i2,j] and Isolation1b[i1,i2,j] are trivially satisfied. When Any[i1,i2,j] is one, Isolation0[i1,i2,j] is the trivial constraint; Isolation1a[i1,i2,j] insures that at least one person of type (i1,i2) is assigned to room j, and finally Isolation1b[i1,i2,j] — the actual non-isolation condition — says that room j must receive at least two people from site i2 who are of rank i1 or an “adjacent” rank (specified in the set ADJ[i1]).

In addition to the obvious drawbacks of complication and inefficiency, these constraints represent only one of many ways of writing the desired restriction in integer programming form; the modeler had to experiment with a number of possibilities before hitting on one that yielded good results in a branch-and-bound procedure. In contrast, the and and not operators can be used to describe these constraints in a much more concise and understandable form:

```
subj to Isolation {(i1,i2) in TYPE, j in ROOM}:
   not (Assign[i1,i2,j] = 1 and
      sum {ii1 in ADJ[i1]: (ii1,i2) in TYPE} Assign[ii1,i2,j] = 0);
```

This formulation requires no supplementary zero-one variables and is significantly closer to the modeler’s original conception.

The generality principle also argues for allowing constraints to use AMPL’s iterated analogues of and and or, which resemble the summation operator but with keywords `forall` and `exists`, respectively. These then become special cases of the iterated counting operators to be proposed later in this section.

The introduction of logical operators into constraints does raise some implementational complications. Since it extends the allowable forms for constraints, we must correspondingly extend the representation that AMPL uses in passing constraints to solvers. If AMPL were a modeling language for linear problems alone, such an extension would require fundamental changes to the language's translation process. AMPL already handles a variety of nonlinear expressions in the variables, however, by passing quite general expression trees to the solver. As a result, the changes required to accommodate new operators on variables — including `and`, `or`, `not`, and others to be proposed in this section — should be straightforward. The greater challenge will be to write AMPL's interfaces to different solvers, so as to convert the expression trees to whatever constraint representations the solvers require.

More seriously, with the introduction of logical operators it becomes possible to specify a feasible region that is not closed, with the result that a continuous-valued objective may have an infimum (or supremum) but may not achieve a minimum (or maximum). A simple case such as

```
not (sum {p in PROD} Trans[i,j,p] <= min_ship[i,j])
```

is easy to spot, but it is not computationally practical (or even theoretically possible) to detect every combination of constraints that gives rise to a non-closed region. The best one may be able to hope for is that solvers will deal sensibly with the non-closed case, such as by returning a solution that is optimal for the closure of the feasible region within specified tolerances.

Common subexpressions pose another challenge. For example, a simple "zero-or-range" restriction from a multicommodity transportation problem is naturally written as

```
subj to Multi_Min_Ship {i in ORIG, j in DEST}:
   sum {p in PROD} Trans[i,j,p] = 0 or
   min_load <= sum {p in PROD} Trans[i,j,p] <= limit[i,j];
```

For many solvers, it is worthwhile to recognize this as a special kind of restriction on the single expression `sum {p in PROD} Trans[i,j,p]`, rather than as a disjunction between two arbitrary constraint expressions. AMPL already detects common subexpressions, so the issues here are to decide which such expressions are significant in combinatorial optimization and how their presence should be communicated to solvers.

Conditional operators

Various if-then and if-then-else constructs provide another convenient way of specifying logical constraints. We review two of these *conditional* forms that are already available in AMPL, and suggest an extension that would be especially useful for formulating combinatorial models. (A third form of conditional — an if-then or if-then-else *statement* for use in programming with the AMPL command language — is not relevant to model formulation.)

AMPL provides a conditional operator,

```
if logical-expr then expr1
if logical-expr then expr1 else expr2
```

which takes the value *expr1* if *logical-expr* is true and the value *expr2* (or some default value) if *logical-expr* is false. When this operator appears in a constraint, the *logical-expr* can contain variables, in which case AMPL handles the constraint like other nonlinear constraints, passing an expression tree to the solver. In particular, the *logical-expr* may be any valid constraint expression. Thus in a location-transportation model it is entirely acceptable to the AMPL translator to write the objective in terms of variables `Build[i]` and `Ship[i,j]` as

```
minimize Total_Cost:
   sum {i in WHSE} (if Build[i] = 1 then build_cost[i]) +
   sum {i in WHSE, j in CUST} trans_cost[i,j] * Ship[i,j];
```

Solvers have the option of processing such an expression, although current AMPL interfaces to integer linear programming solvers reject it as a nonlinearity.

AMPL also has a form of conditional indexing expression, which is used in the context of constraints as follows:

```
subject to {if const-logical-expr}: constraint-expr;
```

Thus one indexing example in the AMPL book [13, §8.4] is:

```
subject to Time {if avail > 0}:
   sum {p in PROD} (1/rate[p]) * Make[p] <= avail;
```

It is arguably more natural, however, to make the `if` condition part of the constraint expression:

```
subject to Time: if avail > 0 then
   sum {p in PROD} (1/rate[p]) * Make[p] <= avail;
```

We then have a new conditional form of constraint:

```
if logical-expr then constraint-expr1
if logical-expr then constraint-expr1 else constraint-expr2
```

As in the case of the conditional operator, variables could be allowed in the *logical-expr*. AMPL could then conveniently express a considerably greater variety of conditional constraints. For example, the previously exhibited multicommodity transportation constraint could be written more naturally as follows (`multicom1.mod`):

```
subject to Multi_Min_Ship {i in ORIG, j in DEST}:
   if sum {p in PROD} Trans[i,j,p] > 0 then
      min_load <= sum {p in PROD} Trans[i,j,p] <= limit[i,j];
```

In the location-transportation example, capacity constraints could be expressed as:

```
subject to Capacity {i in WHSE}:
   if Build[i] = 1
      then sum {j in CUST} Ship[i,j] <= cap[i]
      else forall {j in CUST} Ship[i,j] = 0;
```

In general, the expression following `if` could be any valid constraint expression. The AMPL translator would convert these conditional constraints to expression trees, in much the same way that it currently handles conditional operators.

Counting operators

AMPL's `card` operator returns the number of members in a set. If we extend the language to let `card` be applied to a set defined in terms of variables, it can count the number of constraints of a given form that are satisfied. As an example, the integer programming form of the multicommodity transportation problem (`multicom0.mod`) has the following constraint on the number of destinations served by any origin:

```
subject to MaxServe {i in ORIG}:
   sum {j in DEST} Use[i,j] <= maxserve;
```

Using `card`, the same thing could be expressed in terms of the natural `Trans[i,j,p]` variables, without recourse to the auxiliary zero-one `Use[i,j]` variables:

```
subject to MaxServe {i in ORIG}:
   card {j in DEST: sum {p in PROD} Trans[i,j,p] > 0} <= maxserve;
```

This form might be too general, though. The operand of `card` could be any set expression. To implement the resulting constraint, the solver would have to receive enough information to enable it to evaluate that set expression given any values of the variables.

We could circumvent this difficulty by restricting the ways in which variables may appear in the argument to `card`, but the generality principle suggests that we should avoid complicating the language design with such restrictions. Instead, we could define a new operator that explicitly counts the number of times that a certain constraint is satisfied:

`count` {*indexing*} *constraint-expr*

The above constraint would then be written:

```
subject to MaxServe {i in ORIG}:
   count {j in DEST} (sum {p in PROD} Trans[i,j,p] > 0) <= maxserve;
```

The generality principle directs that any valid AMPL constraint should be allowed as the *constraint-expr* operand to `count`.

Additional iterated logical operators might be defined to simplify the descriptions of constraints in some common special cases. The following are self-explanatory:

```
atmost1 {indexing} constraint-expr
atleast1 {indexing} constraint-expr
exactly1 {indexing} constraint-expr
```

The `atleast1` operator is a synonym for `exists`, but the other two are not currently available in AMPL. A further generalization would replace the `1` in these operators by a parenthesized argument indicating the number of constraints to be satisfied:

```
atmost(k) {indexing} constraint-expr
atleast(k) {indexing} constraint-expr
exactly(k) {indexing} constraint-expr
```

The preceding example could then be further simplified (`multicom1.mod`) to:

```
subj to MaxServe {i in ORIG}:
   atmost(maxserve) {j in DEST} sum {p in PROD} Ship[i,j,p] > 0;
```

An argument like `(maxserve)` is an extension to current AMPL conventions for iterated operators. It has the same form as an argument to an AMPL function, however, so the violation of similarity is not great. Generality would suggest that the argument could be any AMPL expression, but implementability is likely to dictate that it be limited to a constant expression that evaluates to a positive (or perhaps nonnegative) value.

Another important special case of `count` occurs when constraining the *histogram* of a given expression: the array of the numbers of instances in which the expression takes different values. As a simple example, consider the scheduling problem that assigns a number of jobs to a smaller number of machines, so that at most `cap[k]` jobs are assigned to machine `k`. The conventional formulation (`sched0.mod`) defines a zero-one variable `Assign[j,k]` for each job-machine pair, such that `Assign[j,k]` will be 1 if and only if job `j` is assigned to machine `k`:

```
param n integer > 0;

set JOBS := 1..n;
set MACHINES := 1..n;

param cap {MACHINES} integer >= 0;
var Assign {JOBS,MACHINES} binary;
```

The requirements of the assignment can then be specified by one algebraic constraint for each job and for each machine:

```
subj to OneMachinePerJob {j in JOBS}:
   sum {k in MACHINES} Assign[j,k] = 1;

subj to CapacityOfMachine {k in MACHINES}:
   sum {j in JOBS} Assign[j,k] <= cap[k];
```

An alternative (`sched1.mod`), common in the logic programming literature, associates with each job only one variable, whose value is taken from the set of machines:

```
var MachineForJob {JOBS} integer >= 1, <= n;
```

For each `j` in `JOBS`, the value of `MachineForJob[j]` would be the number of the machine that is assigned to do job `j`. This approach requires fewer variables by an order of magnitude, and automatically enforces the requirement that one machine be assigned to each job. To specify that at most `cap[k]` jobs are assigned to machine `k`, we could use the proposed `count` operator:

```
subj to CapacityOfMachine {k in MACHINES}:
   count {j in JOBS} (MachineForJob[j] = k) <= cap[k];
```

This is not as readable a statement of the constraint as one might like, however, due to the necessity of writing `<= cap[k]` directly after `= k`. It is also likely to be inefficient. Because the `count` operator could be applied to any AMPL *constraint-expr*, its general implementation in the AMPL translator would scan through the entire set `JOBS` for each constraint, testing `MachineForJob[j] = k` for every combination of job `j` and machine `k` — even though only one pass through the jobs is necessary to accumulate the counts for all machines.

These circumstances suggest that AMPL should instead offer a more specialized iterated operator for counting individual values assumed by an expression:

`countof(`*k*`)` {*indexing*} *object-expr*

The scheduling constraint would then reduce (`sched2.mod`) to:

```
subj to CapacityOfMachine {k in MACHINES}:
   countof(k) {j in JOBS} MachineForJob[j] <= cap[k];
```

Although it would remain possible to implement this operator by scanning all of `MachineForJob` once for each machine `k`, the presence of `countof` could alert a solver interface to consider the possibilities of a more efficient treatment.

Pairwise operators

Various assignment and related combinatorial problems require that a collection of entities be pairwise different or disjoint. New iterated operators for these conditions would enable them to be stated more clearly and efficiently.

An example is given by an assignment problem that resembles the one defined above, but with equal numbers of jobs and machines. Each job is assigned to one machine, as before, but also each machine gets one job. The conventional integer programming formulation is much the same, except that all the parameters `cap[k]` are 1 (`assign0.mod`):

```
subj to OneMachinePerJob {j in JOBS}:
   sum {k in MACHINES} Assign[j,k] = 1;

subj to OneJobPerMachine {k in MACHINES}:
   sum {j in JOBS} Assign[j,k] = 1;
```

Using instead the variables `MachineForJob` as before, these constraints can be expressed more succinctly by saying that no two variables `MachineForJob[j1]` and `MachineForJob[j2]` may have the same value for different `j1` and `j2`. But how is such a restriction to be stated algebraically? The literal statement in terms of AMPL's inequality operator would be

```
subj to OneJobPerMachine {j1 in JOBS, j2 in JOBS: j1 < j2}:
   MachineForJob[j1] <> MachineForJob[j2];
```

This is a cumbersome way to express the simple idea of being pairwise different, however, and it requires a number of constraints that is on the order of the square of the number of variables. The difficulty is that we think of pairwise inequality as being a joint property of the collection of `MachineForJob` variables, rather than as a collection of binary relations between individual variables.

In AMPL, properties of indexed collections are defined by means of iterated operators such as `sum` and `exists`. Thus it would make sense to introduce an analogous operator for pairwise inequality in an indexed collection of variables:

`alldiff` {*indexing*} *object-expr*

In our example, the one-to-one relationship of jobs and machines would be written as a single constraint (`assign1.mod`):

```
subj to OneJobPerMachine: alldiff {j in JOBS} MachineForJob[j];
```

Using `alldiff` makes the constraint easier to read, and also conveys more useful information to a solver than a large collection of individual inequalities.

A similar operator could be introduced for indexed collections of set expressions, to express the property of being pairwise disjoint:

`alldisjoint` {*indexing*} *set-expr*

To express the same thing in current AMPL syntax, it is necessary to state for each pair of sets that their intersection has cardinality zero.

Variables in subscripts

When we use zero-one variables for the assignment model, it is easy to express the objective as a conventional linear programming expression (`assign0.mod`):

```
minimize TotalCost:
   sum {j in JOBS, k in MACHINES} cost[j,k] * Assign[j,k];
```

To use the `MachineForJob` variables, however, we are currently forced to resort to a more awkward formulation (`assign1.mod`):

```
minimize TotalCost:
   sum {j in JOBS, k in MACHINES}
      if MachineForJob[j] = k then cost[j,k];
```

If we could simply write `MachineForJob[j]` in place of `k`, then the objective could be expressed much more naturally and efficiently (`assign2.mod`):

```
minimize TotalCost:
   sum {j in JOBS, k in MACHINES} cost[j,MachineForJob[j]];
```

This represents an extension analogous to a number of the others proposed above. An AMPL operator — in this case, the "subscripting operator" — is extended by allowing it to apply to variables.

For reasons of both power and generality, it would be desirable to extend "variables in subscripts" to apply to other variables as well as parameters, and to apply in constraints as well as objectives. These requirements arise in sequencing models, for example, that assign a number of jobs to an equal number of "slots" so that each job is assigned to a different slot. The key difference between these slots and the machines of the preceding assignment model is that the slots have a significant ordering. Setup costs and times between jobs are a function of this slot ordering, in ways that can be hard to express using current AMPL features.

As an illustration, we consider a sequencing model introduced in [17] to exhibit a variety of requirements. For individual jobs, we have data,

- `procTime[j]`, the processing time required for job `j`
- `dueTime[j]`, the deadline for completion of job `j`
- `duePen[j]`, the cost per unit of time that job `j` is finished *early*
- `classOf[j]`, the setup class to which job `j` belongs

and corresponding variables:

- `ComplTime[j]`, the time when job `j` is assigned to finish

We have additional setup data for pairs of job classes:

- `setupTime[k1,k2]`, the setup time for a class `k2` job after a class `k1` job
- `setupCost[k1,k2]`, the setup cost for a class `k2` job after a class `k1` job

A representative integer programming formulation for this problem (`seq0.mod`) defines a zero-one variable for each pair of jobs:

- `Seq[j1,j2]`, equal to 1 if and only if `j2` immediately follows `j1`

Hence the number of variables is on the order of the number of jobs squared. An alternative (`seq1.mod`), following the approach introduced in the preceding assignment example, uses two arrays of variables whose values are slot or job numbers:

- `SlotForJob[j]`, the slot to which job `j` is assigned
- `JobForSlot[k]`, the job assigned to slot `k`

The number of variables thus remains proportional to the number of jobs, but again their benefit will only realized by allowing them to appear within subscripts.

Consider first the setup cost associated with the `k`th slot. It is the cost of going from `JobForSlot[k-1]` to `JobForSlot[k]`, or

```
setupCost[classOf[JobForSlot[k-1]],classOf[JobForSlot[k]]]
```

Thus the objective of minimizing total setup cost plus total earliness penalty can be written:

```
var DCost {j in 1..nJobs}
   = duePen[j] * (dueTime[j] - ComplTime[j]);
var SCost {k in 1..nSlots}
   = setupCost[classOf[JobForSlot[k-1]],classOf[JobForSlot[k]]];

minimize TotalCost:
   sum {j in 1..nJobs} DCost[j] + sum {k in 1..nSlots} SCost[k];
```

Similarly constructed terms are employed in the constraint that relates the completion times of two consecutive jobs:

```
subj to ComplTimeDefn {k in 0..nSlots-1}:
  ComplTime[JobForSlot[k]] =
    min (dueTime[JobForSlot[k]],
      ComplTime[JobForSlot[k+1]]
       - setupTime[classOf[JobForSlot[k]],classOf[JobForSlot[k+1]]]
       - procTime[JobForSlot[k+1]]);
```

The expression `ComplTime[JobForSlot[k]]` is an example of a variable in the subscript of a variable, representing in this case the finish time for the job assigned to slot `k`.

The `SlotForJob` variables make it easy to describe the precedence constraints:

```
subj to Precedence {j in 1..nJobs: classOf[j-1] = classOf[j]}:
   SlotForJob[j-1] < SlotForJob[j];
```

To ensure that the `JobForSlot` and `SlotForJob` variables represent the same valid sequencing, however, we need one more instance of a variable subscripted by a variable:

```
subj to JobSlotDefn {k in 0..nSlots}:
   SlotForJob[JobForSlot[k]] = k;
```

This constraint also indirectly ensures that each job is assigned a different slot, and each slot is assigned a different job.

Currently, AMPL allows a subscript to be specified by any arithmetic expression in sets and parameters of the model. Thus our generality principle would suggest that the language should be extended by allowing subscripts to contain valid arithmetic expressions that also use variables. Then rather than evaluating all subscripts while generating a problem (as at present), AMPL will need to send subscript expressions involving variables to the solver, for evaluation within the optimization algorithm. This extension to the implementation is not expected to give rise to any fundamental difficulties, since the expressions to be allowed within subscripts are no more general than the expressions currently allowed elsewhere in constraints.

3 EXTENSIONS TO THE RANGES OF VARIABLES

Algebraic modeling languages have traditionally been designed around the idea of the decision variable, and in particular the numerical-valued decision variable. This sort of variable has many uses even in formulations of combinatorial problems, as the preceding examples have suggested. Nevertheless, there remain a significant variety of combinatorial optimization problems that do not have natural descriptions solely in terms of traditional modeling language variables.

Since combinatorial optimization problems do involve specific decisions, the problem faced by modeling languages is not due to their fundamental reliance on decision variables, but rather to their restriction that variables take numbers as values. This section thus considers two useful extensions that are based on expanding the ranges of variables, first to arbitrary “objects” and then to sets. Since values of these kinds are already well supported in AMPL for other purposes, the principles of consistency and generality are easy to maintain in the extended language.

Object-valued variables

In the preceding assignment model (`assign2.mod`) we artificially defined `JOBS` and `MACHINES` to be sets of the first `n` integers. AMPL does not require the use of integers to represent objects that are not intrinsically numbered in any way, however. Instead it provides for sets of arbitrary character strings that stand for objects. A more natural formulation of the assignment problem would thus begin by declaring arbitrary job and machine sets, along with a check that they contain the same number of objects (`assign3.mod`):

```
set JOBS;
set MACHINES;
   check card (JOBS) = card (MACHINES);
```

For each `j` in `JOBS`, the variable `MachineForJob[j]` would then have as its value the member of set `MACHINES` that is assigned to do job `j`. Such a variable cannot be defined in AMPL as it currently exists, however, because all AMPL variables are limited to numerical values.

We are thus led to consider an extension that allows variables to take values from an arbitrary set. AMPL's variable declarations currently use the operators `>=` and `<=` to restrict a variable's domain to a particular numerical interval, as in `var FinishTime {JOBS} >= 0`. Thus it makes sense to allow the set-membership operator `in` to be used in the same context to denote restriction of a variable's domain to an arbitrary set:

`var` *var-name* `{`*indexing*`}` `in` *set-expr* `;`

In particular, the variable declaration for the assignment model would be written as:

```
var MachineForJob {JOBS} in MACHINES;
```

The remainder of the formulation would be the same as before. A similarly minor change to our sequencing example would permit it also to use a set of objects for the jobs.

This extension clearly satisfies the design principles of similarity and independence. It can satisfy the generality principle as well, by allowing the expression following `in` to be any AMPL expression (not involving variables) that evaluates to a set.

The usefulness of this extension occurs mainly in combination with the previously described extension to permit variables in subscripts. Without the latter, there is usually nowhere in a constraint expression that an object-valued variable can be used. A minor exception might occur in the case of a variable that is fundamentally numerical but that can take values only from an arbitrary set of numbers. For example, if we can decide to build a warehouse at any location in a set `LOC`, but only of a pre-determined size from the set `WSIZE`, then the variable `Build[i]` that represents the size of the warehouse built at `i` might be declared by

```
var Build {LOC} in {0} union WSIZE;
```

This situation is already handled efficiently by most branch-and-bound codes for integer programming, however, through the device known as special ordered sets of type 1 [1, 27].

Set-valued variables

Many kinds of combinatorial problems are described more naturally in terms of choosing an optimal subset, than in terms of choosing individual values of any kind. Thus another extension to the domain specification in AMPL's `var` statements would give rise to variables that take subsets as values.

As an example, consider first a simple knapsack problem that concerns a set of objects having given values and weights, and an overall capacity:

```
set OBJECTS;

param value {OBJECTS} > 0;
param weight {OBJECTS} > 0;

param capacity > 0;
```

The problem can be stated concisely and naturally as follows: Find a subset of the set `OBJECTS`, such that its total value is maximized, subject to its total weight being no more than the capacity. This statement can be converted to a conventional algebraic formulation by use of binary variables indexed over `OBJECTS`:

```
var In {OBJECTS} binary;

maximize TotalValue:
   sum {i in OBJECTS} value[i] * In[i];

subject to WeightLimit:
   sum {i in OBJECTS} weight[i] * In[i] <= capacity;
```

The variable `In[i]` is 1 if and only if object `i` is in the selected subset.

To represent the original problem description more directly, AMPL needs a better way of expressing the requirement to "find a subset of `OBJECTS` such that . . . ". Equivalently, in the terminology of algebraic model descriptions, AMPL needs a way to define variables that represent subsets. As in the previous case of object-valued variables, it suffices to expand the variety of operators that can be used in a `var` statement to specify a variable's domain. To express the fact that a variable represents some subset of a given set, we would use AMPL's set containment operator, `within`:

`var` *var-name* {*indexing*} `within` *set-expr* ;

For the knapsack example, this extension permits the following direct formulation:

```
var Knapsack within OBJECTS;

maximize TotalValue:
   sum {i in Knapsack} value[i];

subject to WeightLimit:
   sum {i in Knapsack} weight[i] <= capacity;
```

The alternative objective and constraint superficially resemble their counterparts that use zero-one variables, but they sum over the members of the set-valued variable `Knapsack` rather than over the given set `OBJECTS`. The result is a formulation that better preserves the brevity and naturalness of the original problem statement.

A somewhat different perspective is provided by the budgeted traveling salesman problem. It is stated in terms of a set of cities, one designated the home city; a set of city pairs on which travel is possible, and travel costs between those pairs; and an overall budget:

```
set CITIES;
param Home symbolic in CITIES;

set LINKS within {i in CITIES, j in CITIES: i <> j};
param cost {LINKS};

param budget > 0;
```

In one version of the problem, the goal is to plan a tour from the home city visiting as many cities as possible, using only available city-pair links, and subject to the total travel cost being within the budget. Thus a natural formulation is provided by using a set-valued variable to represent the subset of cities visited.

This example differs from the previous one, however, in that the order of the cities is also significant. We would thus want to further extend the `var` declaration to encompass ordered set-valued variables:

```
var var-name {indexing} ordered within set-expr ;
var var-name {indexing} circular within set-expr ;
```

The keywords `ordered` and `circular` would have the same meanings as they currently do in AMPL's `set` declarations. Both would indicate that the members of the subset are ordered, but `circular` would also say that the ordering is extended so that the first member of the set is regarded as following the last.

For the budgeted traveling salesman example, the natural set-valued variable would be the circularly ordered subset of cities visited:

```
var Tour circular within CITIES;
```

The objective and constraints are then easily specified:

```
maximize CitiesVisited: card {Tour};

subject to BudgetLimit:
   sum {c in Tour} cost[c,next(c)] <= budget;

subject to LeaveHome: first(Tour) = Home;

subject to LinkExists {c in Tour}: (c,next(c)) in LINKS;
```

Unlike the knapsack problem, this one has no clearly analogous equivalent as an integer program. The integer programming approach has had some success in solving this kind of problem, but only through substantial and somewhat problem-specific modifications to both the formulation and the branch-and-bound procedure. This state of affairs is characteristic of many combinatorial optimization problems of practical interest.

4 USING ILOG SOLVER FOR GENERAL-PURPOSE COMBINATORIAL OPTIMIZATION

ILOG SOLVER is a C++ class library for "problems in planning, allocation, optimization, management, mixing materials, assignment, layout" and many other areas [16]. Although these problems have a long association with linear and integer programming, SOLVER differs fundamentally from commercial integer programming codes, in that it does not carry out a branch-and-bound search based on solving continuous relaxations of integer programs. Instead it relies a constraint logic programming approach to problem-solving, which applies tree-search techniques developed through the study of logical inference in the field of artificial intelligence [28].

SOLVER addresses two distinct activities of large-scale optimization, which are described by its developers as follows:

- ***Problem representation.*** A problem representation consists of the declaration of the unknowns and the constraints of the problem. This representation is specific to the problem domain under consideration and requires a very expressive programming language to capture that specificity. SOLVER uses the object-orientation of C++ to make this activity easier: classes of objects are provided for representing unknowns. These objects are called *constrained variables.* With each of these constrained variables, we associate a set of possible values called the *domain* of the variable.
- ***Solution search.*** Solving the problem consists of selecting a value in the domain of each constrained variable, so that all the constraints are satisfied. Moreover, SOLVER can also be used to search for a solution that optimizes a given criterion.

SOLVER thus incorporates aspects of both a modeling language and an optimization algorithm. Used as intended, it would take the place of a modeling language such as AMPL and an integer programming code such as CPLEX, OSL or XA [10]. We have in mind two somewhat different uses, however, in the context of general-purpose combinatorial optimization.

First, SOLVER's problem representations can provide an independent confirmation of the usefulness of the AMPL extensions described in the two preceding sections. Like AMPL, SOLVER has been designed empirically, based on the developers' perceptions of users' ways of thinking about optimization. Thus if SOLVER can be found to incorporate C++ functions and classes that serve the same purposes as the proposed AMPL extensions, our confidence in the appropriateness of the extensions will be strengthened.

Second, SOLVER's problem representations have the potential to provide an excellent programming environment for implementing an interface from the proposed AMPL combinatorial extensions to SOLVER's solution search routines. Used in this way, SOLVER would play the role of a general-purpose combinatorial optimizer, the missing link (as explained in this paper's introduction) in the use of algebraic modeling languages for combinatorial optimization.

The remainder of this section introduces ILOG SOLVER's problem representation, and describes issues faced in using SOLVER's representation as a general-purpose interface to its solution search. The following section then considers specifically the SOLVER features that would correspond to each of the extensions proposed in sections 2 and 3.

The SOLVER problem representation

The user of ILOG SOLVER constructs a model by writing an executable C++ program. The object-orientation of C++ is exploited to make the model-defining parts of the program relatively declarative in nature, so that the program states the model in somewhat the same fashion that AMPL would. Nevertheless, SOLVER is unlikely to supplant AMPL as a modeling language, due to limitations inherent in the design of programming languages like C++.

As an example of SOLVER's strengths and limitations as a declarative modeling language, we compare the previous sequencing example as it could be expressed in extended AMPL (`seq1.mod`) to its equivalent as a SOLVER C++ program (`seq.cc`).

For one-dimensional data, there is a direct analogy between declarations in AMPL and in SOLVER. To concisely display analogous AMPL and SOLVER expressions, we'll show them separated by a line, with the AMPL one above:

```
param dueTime {0..nJobs} >= 0;
------------------------------
dueTime = IlcIntArray(nJobs+1);
```

There's a similar analogy for one-dimensional arrays of variables:

```
var SlotForJob {j in 0..nJobs} in 0..nSlots;
---------------------------------------------
IlcIntVarArray SlotForJob(nJobs+1,0,nSlots);
```

Arithmetic and relational operators and the "subscripting operator" `[]` are the same in AMPL and in C++. As a result, there is a pronounced similarity between AMPL and SOLVER constraint declarations:

```
subj to ComplTimeFinalDefn:
   ComplTime[JobForSlot[nSlots]] = dueTime[JobForSlot[nSlots]];
------------------------------------------------------------------
IlcPost(
   ComplTime[JobForSlot[nSlots]] == dueTime[JobForSlot[nSlots]] );
```

For indexed collections of constraints, the indexing expression delimited by `{}` in AMPL has as its analogue a C++ `for` loop:

```
subj to Precedence
  {j in 1..nJobs: classOf[j-1] = classOf[j]}:
      SlotForJob[j-1] < SlotForJob[j];
```

```
for (j = 1; j < nJobs+1; j++)
   if (classOf[j-1] == classOf[j])
      IlcPost( SlotForJob[j-1] < SlotForJob[j] );
```

The C++ function `IlcPost` directs SOLVER to set up a constraint, and hence serves much the same purpose as AMPL's `subj to`.

Two-dimensional arrays are more of a challenge to SOLVER. In our example, the analogue of AMPL's two-dimensional parameter `setupTime` is a one-dimensional C++ array of corresponding size:

```
param setupTime {0..nClasses,1..nClasses};
```

```
setupTime = IlcIntArray((nClasses+1)*(nClasses+1));
```

With the help of a simple auxiliary function,

```
IlcIntExp classesOf(IlcIntVar job1, IlcIntVar job2){
   return (classOf[job1] * (nClasses+1) + classOf[job2]);
}
```

the SOLVER definition of the `ComplTime` variables can be made remarkably similar to the corresponding AMPL statement:

```
subj to ComplTimeDefn {k in 0..nSlots-1}:
  ComplTime[JobForSlot[k]] =
    min (dueTime[JobForSlot[k]],
      ComplTime[JobForSlot[k+1]]
       - setupTime[classOf[JobForSlot[k]],classOf[JobForSlot[k+1]]]
       - procTime[JobForSlot[k+1]]);
```

```
for (k = 0; k < nSlots; k++)
  IlcPost( ComplTime[JobForSlot[k]] ==
    IlcMin( dueTime[JobForSlot[k]],
      ComplTime[JobForSlot[k+1]]
       - setupTime[classesOf(JobForSlot[k],JobForSlot[k+1])]
       - procTime[JobForSlot[k+1]] ) );
```

SOLVER's approach does involve a certain amount of low-level programming that has no counterpart in AMPL, however. As an alternative, a SOLVER user can achieve the effect of a two-dimensional data array by declaring a C++ array of objects of type `IlcIntArray`. Individual setup times can then be accessed by expressions of the form `setupTime[i1][i2]`, but SOLVER's overloading of the `[]` operator is not currently sufficient to permit general integer expressions in place of both `i1` and `i2`.

Both AMPL and SOLVER can express the objective function by defining arrays of variables representing earliness penalties,

```
var DCost {j in 1..nJobs}
   = duePen[j] * (dueTime[j] - ComplTime[j]);
```

```
IlcIntVarArray DCost(nJobs+1); DCost[0] = IlcIntVar(0,0);
for (j = 1; j < nJobs+1; j++)
   DCost[j] = duePen[j] * (dueTime[j] - ComplTime[j]);
```

and setup costs,

```
var SCost {k in 1..nSlots}
   = setupCost[classOf[JobForSlot[k-1]],classOf[JobForSlot[k]]];
```

```
IlcIntVarArray SCost(nSlots+1); SCost[0] = IlcIntVar(0,0);
for (k = 1; k < nSlots+1; k++)
   SCost[k] = setupCost[classesOf(JobForSlot[k-1],JobForSlot[k])];
```

The full objective is then written as summation + summation:

```
minimize TotalCost:
   sum {j in 1..nJobs} DCost[j] + sum {k in 1..nSlots} SCost[k];
```

```
IlcIntVar TotalCost = IlcSum(DCost) + IlcSum(SCost);
```

In both cases, the introduction of the auxiliary variables `DCost` and `SCost` need not increase the dimension of the problem that is ultimately sent to an optimizing algorithm. AMPL also offers the option of dispensing with these variables by substituting them explicitly into the summations:

```
minimize TotalCost:
   sum {j in 1..nJobs} duePen[j] * (dueTime[j] - ComplTime[j]) +
   sum {k in 1..nSlots}
      setupCost[classOf[JobForSlot[k-1]],classOf[JobForSlot[k]]];
```

SOLVER cannot avoid the auxiliary variables, however, because its `IlcSum` function currently requires an argument of type `IlcIntVarArray` in order to be applied to integer variables.

The SOLVER user clearly must work around a variety of C++ limitations. Most seriously, C++ only weakly supports many aspects of sets and indexing that enable AMPL to express large-scale optimization models naturally and clearly, such as:

- Indexing over arbitrary sets of numbers or objects.
- Multidimensional indexing.
- Indexing of "sparse" arrays over pairs, triples, and longer tuples.

SOLVER remedies these omissions to a limited extent, by supplying C++ class declarations that implement several set data types and functions. Even so, AMPL offers a much richer variety of facilities for computing sets, for indexing model components over sets, and for iterating operators over sets. (Indeed, set and indexing facilities account for many of the characteristics that most clearly distinguish algebraic modeling languages like AMPL from other computer languages.)

SOLVER's flexibility is also restricted by the fundamentally executable nature of C++, in contrast to the declarative nature of model formulations for optimization. In the illustration above, this difference has been played down by showing only the C++ statements that are most nearly analogous to AMPL statements. The C++ program does not actually define the array `dueTime`, for example, by simply executing

```
dueTime = IlcIntArray(nJobs+1);
```

Rather, it first defines `nJobs` and sets its value; then executes the above statement (which calls a constructor to allocate an array of the desired size); then reads and assigns values for the array elements `dueTime[j]`. Only after these steps may `dueTime` be used in SOLVER expressions for the constraints and objective. By contrast, an AMPL model need only declare

```
param nJobs integer > 0;
param dueTime {0..nJobs} >= 0;
```

References to `dueTime` may then appear in any objective and constraint declaration statements. The AMPL translator does not require that values for `nJobs` or each `dueTime[j]` be specified before it reads and processes the model; reading the data and instantiating the constraints are handled by a subsequent phase of translation, which also allocates storage automatically for the proper number of array elements.

As a result of AMPL's more declarative design, a clean separation of model and data can be more readily maintained, and the correctness of the model can be more easily verified. The inherent difficulties of maintaining an executable representation of a model have long been appreciated, as discussed in [9].

In addition to the difficulties cited so far, SOLVER users face the challenge of working in the C++ language. Although the class hierarchy hides many complications, eventually a SOLVER program must be submitted to a C++ compiler, and the program's author must be prepared to deal with relatively low-level error messages such as:

```
The operation "long*[IlcIntVar]" is illegal.
IlcIntVar::operator=(IlcIntExpI*) is not accessible from main().
Could not find a match for IlcConstraint::IlcConstraint(void).
Cannot assign long to IlcIntExp.
```

Successful use of SOLVER thus depends on a clear understanding of conversions, overloading, constructors, and other fundamental C++ concepts.

These comments suggest that C++'s lack of desirable optimization modeling features is not merely an oversight in design, but is rather due to fundamental differences between the design criteria of object-oriented programming languages and algebraic modeling languages. In particular, a modeling language is designed above all to allow people to work with models using the same terms in which they conceive of them. Thus modeling languages have highly developed model declaration features, such as AMPL's many set and indexing alternatives, that would be redundant for the purposes of a more generally applicable programming language.

Using SOLVER for general-purpose optimization

In light of the preceding observations, it makes sense to consider using a modeling language as a "front end" to the optimization techniques provided by ILOG SOLVER. The SOLVER class library would then serve as a tool for writing the AMPL/SOLVER interface routines.

When applied to the more general forms of models that would be received from AMPL, however, SOLVER's features for declaring particular models are stretched somewhat beyond their original purpose. As a test of the applicability of these features, we have constructed a SOLVER interface for AMPL integer programs. Since integer programs are already supported by AMPL, this experiment could be carried out without any of the modifications to the language that would be needed for the proposed combinatorial extensions.

Results have been favorable, in that a robust and efficient working interface has been successfully constructed. The following code suffices, for example, to generate a SOLVER constraint from the `i`th linear AMPL constraint, in the case where all coefficients are integral:

```
for (nonz = 0, cg = Cgrad[i]; cg; nonz++, cg = cg->next);

IlcIntArray a(nonz);
IlcIntVarArray Y(nonz);

for (k = 0, cg = Cgrad[i]; cg; k++, cg = cg->next) {
   a[k] = cg -> coef;
   Y[k] = X[cg -> varno];
   }

if (loConBnd[i] < upConBnd[i] && loConBnd[i] > negInfinity)
   IlcPost( IlcScalProd(a,Y) >= (IlcInt) loConBnd[i] );
if (upConBnd[i] < Infinity)
   if (loConBnd[i] < upConBnd[i])
      IlcPost( IlcScalProd(a,Y) <= (IlcInt) upConBnd[i] );
   else
      IlcPost( IlcScalProd(a,Y) == (IlcInt) upConBnd[i] );
```

The central `for` loop extracts nonzero constraint coefficients and corresponding variable indices from AMPL's sparse data structure `Cgrad`, and packs them into SOLVER objects `a` and `Y` of type `IlcIntArray` and `IlcIntVarArray`, respectively. The appropriate constraint on the scalar product of these two arrays is then posted by one of the `if` statements. Similar but separate statements are required for the case in which some of the coefficients might not be integer.

AMPL's standard mechanism for conveying options to optimization codes is readily hooked to the various predefined options for SOLVER's search procedure. Thus for example the user can give the AMPL command

```
option ilog_solver_options 'choose_var=2 gen_var=1';
```

to modify SOLVER's procedure for constructing the search tree. (In SOLVER's terms, these options say to use the enumeration function `IlcBestGenerate` with the variable choice function `IlcChooseMaxSizeInt` as an argument.)

5 IMPLEMENTABILITY OF COMBINATORIAL AMPL EXTENSIONS THROUGH ILOG SOLVER

We now return to the AMPL extensions proposed in sections 2 and 3, to consider which ILOG SOLVER functions and classes might be useful in implementing them. The correspondences are mostly quite close, suggesting that the proposed extensions are of practical interest and that an AMPL/SOLVER interface is a sensible idea. Issues to be faced in going from this proposal to an implementation are discussed in section 6.

Names beginning in `Ilc` in the following discussion represent C++ types and functions defined in version 3.2 of SOLVER.

Logical operators

SOLVER provides analogues for AMPL's `and`, `or`, and `not` operators by overloading the C++ operators `&&`, `||`, and `!`, respectively. Thus for example this AMPL constraint expression using the `or` operator as proposed in Section 2,

```
Start[i2] >= Start[i1] + t_offset[i1,i2] or
Start[i1] >= Start[i2] + t_offset[i2,i1]
```

can be written in the same way in SOLVER:

```
Start[i2] >= Start[i1] + t_offset[i1][i2] ||
Start[i1] >= Start[i2] + t_offset[i2][i1]
```

The two overloaded `>=` operators return values of type `IlcConstraint`. Thus `||` is overloaded to accept two operands of type `IlcConstraint` and to return an `IlcConstraint`.

The effect of AMPL's iterated logical operators, `forall` and `exists`, can be achieved as special cases of SOLVER's `IlcCard` function; see the discussion of counting operators below.

Conditional operators

SOLVER can directly define if-then constraints by use of either a specialized constraint-posting function (`IlcIfThen`) or an implication operator (`<=`). There is no direct support for if-then-else constraints, however; they have to be built from two calls to `IlcIfThen` or an equivalent constraint expression using several logical operators.

AMPL's operator of the form `if` *logical-expr* `then` *expr1* `else` *expr2* has a direct analogue in C++'s ternary conditional operator *logical-expr* `?` *expr1* `:` *expr2*. The latter cannot be overloaded, however, to specialize its behavior when applied to SOLVER's constraint data types.

Counting operators

SOLVER's analogue to AMPL's cardinality operators is the `IlcCard` function. This function acts like AMPL's `card` when called with one argument representing a set of integers (type `IlcIntSetVar`) or objects (type `IlcAnySetVar`). It acts like the proposed AMPL `count` when called with an index (type `IlcIndex`) and a constraint using that index (type `IlcConstraint`). The special case of the proposed AMPL `countof(`*k*`)` operator is implemented efficiently through Solver's `IlcDistribute` function.

With two arguments, `IlcCard` acts like a highly restricted form of AMPL's proposed `count` operator. Its first argument defines only one index, which may be used in its second argument only to subscript objects of type `IlcIntVarArray` and `IlcAnyVarArray` that are all of the same size. AMPL's `count` would have no such restrictions, being defined to use any AMPL indexing expression and constraint expression as its two arguments. A SOLVER program could get the same effect, however, by summing a specially constructed array of auxiliary variables, each being 1 or 0 if its corresponding constraint is or isn't satisfied. Though such an array may be less convenient, it is easy to set up using SOLVER's "metaconstraint" feature that automatically associates a binary integer variable with any expression of type `IlcConstraint`.

Pairwise operators

SOLVER provides functions that correspond closely to the proposed AMPL pairwise operators. Function `IlcAllDiff` specifies that the components of an array of integers (type `IlcIntVarArray`) or an array of objects (type `IlcAnyVarArray`) must be pairwise different. Function `IlcAllNullIntersect` specifies that the components of an array of integer sets (type `IlcIntSetVarArray`) or an array of object sets (type `IlcAnySetVarArray`) must be pairwise disjoint. SOLVER's solution search procedure handles these relationships directly in an efficient way.

Variables in subscripts

SOLVER's arrays of integers (type `IlcIntArray`) and arrays of integer variables (type `IlcIntVarArray`) have a subscripting operator that admits any integer expression (type `IlcIntExp`) as its argument. Since an integer expression may in general contain variables, subscripts containing variables can implemented straightforwardly, as seen in section 4's sequencing example. This observation also extends to SOLVER's arrays of objects (types `IlcAnyArray` and `IlcAnyVarArray`).

Object-valued variables

SOLVER provides "constrained enumerated variables" (type `IlcAnyVar` and type `IlcAnyVarArray`) whose values are C++ generic (`void*`) pointers. Since pointers of this type may reference any type of C++ object, they can directly implement the proposed AMPL object-valued variables.

SOLVER's integer-valued variables can also be used in contexts where each integer stands for an object (rather than for a numerical value that participates in arithmetic expressions). The `IlcAllDiff` operator applies in the same way to object and to integer arrays.

Set-valued variables

SOLVER provides data types for sets of pointers (`IlcAnySet`) and sets of integers (`IlcIntSet`). Set-valued variables are provided by corresponding constrained set variables of pointers (types `IlcAnySetVar` and `IlcAnySetVarArray`) and integers (types `IlcIntSetVar` and `IlcIntSetVarArray`).

SOLVER incorporates functions corresponding to all of the AMPL set operators, including membership (`IlcMember`, `IlcNotMember`), containment (`IlcSubset`, `IlcSubsetEq`), union (`IlcUnion`), intersection (`IlcIntersection`), and cardinality (`IlcCard`). Specialized functions are also provided to define constraints restricting a given set to equal the union (`IlcEqUnion`) or intersection (`IlcEqIntersection`) of two other given sets.

None of these functions provides a convenient way of implementing AMPL sums indexed over set variables, as in the constraint `sum {i in Knapsack} weight[i] <= capacity` from the knapsack example in section 3. Where the knapsack members are represented by integers, however, new forms of the function `IlcSum` have been implemented to express summations of this kind in a particular client's application. These forms are likely to be included in future versions of the general ILOG SOLVER release.

6 DIRECTIONS FOR FURTHER DEVELOPMENT

This study has introduced a variety of alternatives for extending an algebraic modeling language to better support general-purpose combinatorial optimization. Evidence for the usefulness of these extensions has been presented through sample models (sections 2 and 3) and through examples from ILOG SOLVER's independently designed features (section 5). The comparison with SOLVER has also served to provide evidence for the implementability of these extensions.

The next step in this line of investigation will be to add the syntax of the proposed extensions to AMPL, and to create a general-purpose optimizer for models that use the extensions, working through an interface that exploits the corresponding SOLVER functions and data types. This work may face new challenges resulting from the application of SOLVER in a more general-purpose environment than originally intended. Experience with a SOLVER interface for integer programming (section 4) nevertheless suggests that the difficulties should be manageable.

Implementation will also require extensions to AMPL's data structures for conveying model instances to optimizers. AMPL currently uses two kinds of structures: lists of nonzero coefficients for linear expressions, and directed acyclic graphs (representing parse trees) for nonlinear expressions [14]. The latter are sufficiently general to represent most of the extensions proposed in this paper, but some refinements may be desirable to allow the interface to SOLVER to be faster or more effective. For example, many of the operands of the proposed operator extensions are in fact linear constraint expressions, for which a coefficient list would be more useful than a general subgraph. A more difficult problem is posed by constructs such as variables in subscripts and summations over set-valued variables, which require a new kind of expression to be included in the data structure. It would seem that subscripting in an expression like `procTime[JobForSlot[k]]` or `FinishTime[JobForSlot[k-1]]` could be handled as another kind of operator, for example, much as in C++; but unlike all current AMPL operators, it would not be operating on any one particular parameter or variable, but rather on the whole indexed collection `procTime` or `FinishTime`.

After an interface has been completed using SOLVER's predefined types and functions, performance might be improved by selectively defining new classes of objects. It would be possible, in particular, to define new constraint classes. A custom-defined constraint class might be able to implement certain AMPL extensions more effectively than a combination of predefined constraints. This advantage might be achieved by permitting a more efficient conversion from the original AMPL constraint representation, or by defining member functions that do a better job of propagating the constraint's effects through the search tree.

Performance can also often be improved by experimenting with SOLVER's search criteria. Access to predefined search options is readily provided from AMPL's environment, as noted in section 4. SOLVER also provides for user-defined, problem-specific search criteria through definition of certain new functions and object subclasses. At the same time, an algebraic modeling language is well-suited to expressing many search criteria, as shown by numerous examples in [3]. Thus a further extension to AMPL might permit the user to write algebraic expressions that define such choices as branching variable and direction.

Another possibility along these lines, also explored in [3], would be to supply algebraic expressions for bounds at the search nodes. These bounds would play the same role in pruning the search tree as the LP relaxation bounds computed by integer programming solvers. They could be tailored to specific combinatorial models, however, for which no useful analogue to the LP relaxation might be available. SOLVER does not currently incorporate any special mechanism for communicating bounds to its search procedures, but this is a feature that might not be hard to add.

Finally, it should be remarked that the ideas of this paper can be extended to models that have some real-valued decision variables. ILOG SOLVER supports such variables through appropriate C++ classes (`IlcFloatVar` and `IlcFloatVarArray`), which have been used to implement a mixed-integer generalization to the AMPL/SOLVER interface described in section 4. In fact this interface has been further extended to send mixed integer programs to ILOG PLANNER, a companion to SOLVER that incorporates a simplex method, computes LP relaxations, and carries out a standard mixed-integer branch-and-bound search. The search schemes of SOLVER and PLANNER can optionally be applied together, often more effectively than either one alone. Thus it is reasonable to expect that they could be used together to construct a general-purpose solver suitable for mixed-integer AMPL models that employ some of the combinatorial extensions described in this paper.

Appendix: Examples

The following pages exhibit many of the models from which examples in this paper were taken. Models are arranged alphabetically according to the filenames by which they are referenced in the text. All are in AMPL except for one (`seq.cc`) that is an ILOG SOLVER C++ program.

assign0.mod

```
Assignment: Using n^2 zero-one variables
-------------------------------------------------------------

param n integer > 0;

set JOBS := 1..n;
set MACHINES := 1..n;

param cost {JOBS,MACHINES} > 0;
var Assign {JOBS,MACHINES} binary;

minimize TotalCost:
   sum {j in JOBS, k in MACHINES} cost[j,k] * Assign[j,k];

subj to OneMachinePerJob {j in JOBS}:
   sum {k in MACHINES} Assign[j,k] = 1;

subj to OneJobPerMachine {k in MACHINES}:
   sum {j in JOBS} Assign[j,k] = 1;
```

assign1.mod

```
Assignment: Using n integer variables,
with the "alldiff" operator
-------------------------------------------------------------

param n integer > 0;

set JOBS := 1..n;
set MACHINES := 1..n;

param cost {JOBS,MACHINES} > 0;

var MachineForJob {JOBS} integer >= 1, <= n;

minimize TotalCost:
   sum {j in JOBS, k in MACHINES}
      if MachineForJob[j] = k then cost[j,k];

subj to OneJobPerMachine:
   alldiff {j in JOBS} MachineForJob[j];
```

assign2.mod

```
Assignment: Using n integer variables,
with variable in subscript in the objective
----------------------------------------------------------

param n integer > 0;

set JOBS := 1..n;
set MACHINES := 1..n;

param cost {JOBS,MACHINES} > 0;

var MachineForJob {JOBS} integer >= 1, <= n;

minimize TotalCost:
   sum {j in JOBS, k in MACHINES} cost[j,MachineForJob[j]];

subj to OneJobPerMachine:
   alldiff {j in JOBS} MachineForJob[j];
```

assign3.mod

```
Assignment: Using n object-valued variables,
with variable in subscript in the objective
----------------------------------------------------------

set JOBS;
set MACHINES;
   check card (JOBS) = card (MACHINES);

param cost {JOBS,MACHINES} > 0;

var MachineForJob {JOBS} in MACHINES;

minimize TotalCost:
   sum {j in JOBS, k in MACHINES} cost[j,MachineForJob[j]];

subj to OneJobPerMachine:
   alldiff {j in JOBS} MachineForJob[j];
```

flowshp0.mod

Flow Shop: Integer programming formulation

```
set JOBS ordered;
set ALL_MACH ordered;

set MACH {JOBS} ordered within ALL_MACH;

param t_proc {i in JOBS, MACH[i]} > 0;

param t_cum {i in JOBS, j in MACH[i]} :=
   sum {jj in MACH[i]: ord(jj) <= ord(j)} t_proc[i,jj];

param t_offset {i1 in JOBS, i2 in JOBS: i1 <> i2} :=
   max {j in MACH[i1] inter MACH[i2]}
      (t_cum[i1,j] - t_cum[i2,j] + t_proc[i2,j]);

param M > 0;

var End >= 0;
var Start {JOBS} >= 0;
var Precedes {i1 in JOBS, i2 in JOBS: ord(i1) < ord(i2)} binary;

minimize Makespan: End;

subj to Makespan_Defn {i in JOBS}:
   End >= Start[i] + sum {j in MACH[i]} t_proc[i,j];

subj to No12_Conflict {i1 in JOBS, i2 in JOBS: ord(i1) < ord(i2)}:
   Start[i2] >= Start[i1] + t_offset[i1,i2] - M * (1 - Precedes[i1,i2]);

subj to No21_Conflict {i1 in JOBS, i2 in JOBS: ord(i1) < ord(i2)}:
   Start[i2] >= Start[i1] + t_offset[i2,i1] - M * Precedes[i1,i2];
```

```
flowshp1.mod
Flow Shop: Using disjunctive constraints
----------------------------------------------------------------

set JOBS ordered;
set ALL_MACH ordered;

set MACH {JOBS} ordered within ALL_MACH;

param t_proc {i in JOBS, MACH[i]} > 0;

param t_cum {i in JOBS, j in MACH[i]} :=
   sum {jj in MACH[i]: ord(jj) <= ord(j)} t_proc[i,jj];

param t_offset {i1 in JOBS, i2 in JOBS: i1 <> i2} :=
   max {j in MACH[i1] inter MACH[i2]}
      (t_cum[i1,j] - t_cum[i2,j] + t_proc[i2,j]);

var End >= 0;
var Start {JOBS} >= 0;

minimize Makespan: End;

subj to Makespan_Defn {i in JOBS}:
   End >= Start[i] + sum {j in MACH[i]} t_proc[i,j];

subj to No_Conflict {i1 in JOBS, i2 in JOBS: ord(i1) < ord(i2)}:
   Start[i2] >= Start[i1] + t_offset[i1,i2]  or
   Start[i1] >= Start[i2] + t_offset[i2,i1];
```

multicom0.mod

Multicommodity transportation:
Using auxiliary zero-one variables

```
set ORIG;
set DEST;
set PROD;

param supply {ORIG,PROD} >= 0;
param demand {DEST,PROD} >= 0;

   check {p in PROD}:
      sum {i in ORIG} supply[i,p] = sum {j in DEST} demand[j,p];

param limit {ORIG,DEST} >= 0;
param minload >= 0;
param maxserve integer > 0;

param vcost {ORIG,DEST,PROD} >= 0;
var Trans {ORIG,DEST,PROD} >= 0;

param fcost {ORIG,DEST} >= 0;
var Use {ORIG,DEST} binary;

minimize Total_Cost:
   sum {i in ORIG, j in DEST, p in PROD} vcost[i,j,p] * Trans[i,j,p]
 + sum {i in ORIG, j in DEST} fcost[i,j] * Use[i,j];

subject to Supply {i in ORIG, p in PROD}:
   sum {j in DEST} Trans[i,j,p] = supply[i,p];

subject to Max_Serve {i in ORIG}:
   sum {j in DEST} Use[i,j] <= maxserve;

subject to Demand {j in DEST, p in PROD}:
   sum {i in ORIG} Trans[i,j,p] = demand[j,p];

subject to Multi {i in ORIG, j in DEST}:
   sum {p in PROD} Trans[i,j,p] <= limit[i,j] * Use[i,j];

subject to Min_Ship {i in ORIG, j in DEST}:
   sum {p in PROD} Trans[i,j,p] >= minload * Use[i,j];
```

```
multicom1.mod
Multicommodity transportation:
Using conditional expressions and a counting operator
----------------------------------------------------------

set ORIG;
set DEST;
set PROD;

param supply {ORIG,PROD} >= 0;
param demand {DEST,PROD} >= 0;

   check {p in PROD}:
      sum {i in ORIG} supply[i,p] = sum {j in DEST} demand[j,p];

param limit {ORIG,DEST} >= 0;
param minload >= 0;
param maxserve integer > 0;

param fcost {ORIG,DEST} >= 0;
param vcost {ORIG,DEST,PROD} >= 0;
var Trans {ORIG,DEST,PROD} >= 0;

minimize Total_Cost:
   sum {i in ORIG, j in DEST}
      (if sum {p in PROD} Trans[i,j,p] > 0
         then fcost[i,j] + sum {p in PROD} vcost[i,j,p] * Trans[i,j,p]);

subject to Supply {i in ORIG, p in PROD}:
   sum {j in DEST} Trans[i,j,p] = supply[i,p];

subj to MaxServe {i in ORIG}:
   atmost(maxserve) {j in DEST} sum {p in PROD} Ship[i,j,p] > 0;

subject to Demand {j in DEST, p in PROD}:
   sum {i in ORIG} Trans[i,j,p] = demand[j,p];

subject to Multi_Min_Ship {i in ORIG, j in DEST}:
   if sum {p in PROD} Trans[i,j,p] > 0 then
      min_load <= sum {p in PROD} Trans[i,j,p] <= limit[i,j];
```

sched0.mod

Scheduling: Using n^2 zero-one variables

```
-----------------------------------------------------------------

param n integer > 0; set JOBS := 1..n;
set MACHINES := 1..n;

param cap {MACHINES} integer >= 0;

param cost {JOBS,MACHINES} > 0;
var Assign {JOBS,MACHINES} binary;

minimize TotalCost:
   sum {j in JOBS, k in MACHINES} cost[j,k] * Assign[j,k];

subj to OneMachinePerJob {j in JOBS}:
   sum {k in MACHINES} Assign[j,k] = 1;

subj to CapacityOfMachine {k in MACHINES}:
   sum {j in JOBS} Assign[j,k] <= cap[k];
```

sched1.mod

Scheduling: Using n integer variables,
with the "count" operator

```
-----------------------------------------------------------------

param n integer > 0;

set JOBS := 1..n;
set MACHINES := 1..n;

param cap {MACHINES} integer >= 0;
param cost {JOBS,MACHINES} > 0;

var MachineForJob {JOBS} integer >= 1, <= n;

minimize TotalCost:
   sum {j in JOBS, k in MACHINES}
      if MachineForJob[j] = k then cost[j,k];

subj to CapacityOfMachine {k in MACHINES}:
   count {j in JOBS} (MachineForJob[j] = k) <= cap[k];
```

```
sched2.mod
Scheduling: Using n integer variables,
with the "countof" operator
----------------------------------------------------------------

param n integer > 0;

set JOBS := 1..n;
set MACHINES := 1..n;

param cap {MACHINES} integer >= 0;
param cost {JOBS,MACHINES} > 0;

var MachineForJob {JOBS} integer >= 1, <= n;

minimize TotalCost:
   sum {j in JOBS, k in MACHINES}
      if MachineForJob[j] = k then cost[j,k];

subj to CapacityOfMachine {k in MACHINES}:
   countof(k) {j in JOBS} MachineForJob[j] <= cap[k];
```

seq0.mod

```
Sequencing: Using n^2 zero-one variables
---------------------------------------------------------------

param nJobs integer > 0;
param nClasses integer > 0;

param duePen {0..nJobs} >= 0;
param dueTime {0..nJobs} >= 0;
param procTime {0..nJobs} >= 0;
param classOf {0..nJobs} in 0..nClasses;

param setupTime {0..nClasses,1..nClasses};
param setupCost {0..nClasses,1..nClasses};

param BIG := max {j in 0..nJobs} dueTime[j];

var Seq {j1 in 0..nJobs, j2 in 1..nJobs+1: j1 <> j2} binary;
var ComplTime {j in 0..nJobs} >= 0, <= dueTime[j];

minimize TotalCost:
   sum {j in 1..nJobs} duePen[j] * (dueTime[j] - ComplTime[j]) +
   sum {j1 in 0..nJobs, j2 in 1..nJobs: j1 <> j2}
      setupCost[classOf[j1],classOf[j2]] * Seq[j1,j2];

subj to OneAfter {j1 in 0..nJobs}:
   sum {j2 in 1..nJobs+1: j1 <> j2} Seq[j1,j2] = 1;

subj to OneBefore {j2 in 1..nJobs+1}:
   sum {j1 in 0..nJobs: j1 <> j2} Seq[j1,j2] = 1;

subj to NoOverlap {j1 in 0..nJobs, j2 in 1..nJobs: j1 <> j2}:
   ComplTime[j1] + setupTime[classOf[j1],classOf[j2]] + procTime[j2]
      <= ComplTime[j2] + BIG * (1 - Seq[j1,j2]);

subj to Precedence
      {j1 in 0..nJobs-1, j2 in j1+1..nJobs: classOf[j1] = classOf[j2]}:
   Seq[j2,j1] = 0;

subj to Initialization: ComplTime[0] = 0;
```

```
seq1.mod
Sequencing: Using 2n integer variables,
with variables in subscripts
----------------------------------------------------------

param nJobs integer > 0;
param nSlots integer := nJobs;
param nClasses integer > 0;

param duePen {0..nJobs} >= 0;
param dueTime {0..nJobs} >= 0;
param procTime {0..nJobs} >= 0;
param classOf {0..nJobs} in 0..nClasses;

param setupTime {0..nClasses,1..nClasses};
param setupCost {0..nClasses,1..nClasses};

param BIG := max {j in 0..nJobs} dueTime[j];

var JobForSlot {k in 0..nSlots} in 0..nJobs;
var SlotForJob {j in 0..nJobs} in 0..nSlots;
var ComplTime {j in 0..nJobs};

var DCost {j in 1..nJobs}
   = duePen[j] * (dueTime[j] - ComplTime[j]);
var SCost {k in 1..nSlots}
   = setupCost[classOf[JobForSlot[k-1]],classOf[JobForSlot[k]]];

minimize TotalCost:
   sum {j in 1..nJobs} DCost[j] + sum {k in 1..nSlots} SCost[k];

subj to JobSlotInit: JobForSlot[0] = 0;
subj to JobSlotDefn {k in 0..nSlots}: SlotForJob[JobForSlot[k]] = k;

subj to ComplTimeInitDefn: ComplTime[0] = 0;

subj to ComplTimeDefn {k in 0..nSlots-1}:
   ComplTime[JobForSlot[k]] =
      min (dueTime[JobForSlot[k]],
           ComplTime[JobForSlot[k+1]]
            - setupTime[classOf[JobForSlot[k]],classOf[JobForSlot[k+1]]]
            - procTime[JobForSlot[k+1]]);

subj to ComplTimeFinalDefn:
   ComplTime[JobForSlot[nSlots]] = dueTime[JobForSlot[nSlots]];

subj to Precedence {j in 1..nJobs: classOf[j-1] = classOf[j]}:
   SlotForJob[j-1] < SlotForJob[j];
```

seq.cc

Sequencing: ILOG Solver C++ representation for seq1.mod

```
#include <ilsolver/ilcint.h>

IlcInt dummy = IlcInit();

IlcInt nJobs;
IlcInt nSlots;
IlcInt nClasses;

IlcIntArray duePen;    IlcIntArray dueTime;
IlcIntArray procTime;  IlcIntArray classOf;

IlcIntArray setupTime; IlcIntArray setupCost;

IlcIntExp classesOf(IlcIntVar job1, IlcIntVar job2){
   return (classOf[job1] * (nClasses+1) + classOf[job2]);
}

void readData(char* filename){
   FILE* data = fopen(filename,"r");
   if (!data) {
       cerr << "No such file: " << filename << endl; exit(1);
   }

   IlcInt i, j;

   fscanf(data,"%d%d",&nJobs,&nClasses); nSlots = nJobs;

   duePen   = IlcIntArray(nJobs+1);  duePen[0]   = 0;
   dueTime  = IlcIntArray(nJobs+1);  dueTime[0]  = 0;
   procTime = IlcIntArray(nJobs+1);  procTime[0] = 0;
   classOf  = IlcIntArray(nJobs+1);  classOf[0]  = 0;

   for (j = 1; j < nJobs+1; j++)
       fscanf(data,"%d%d%d%d",
           &duePen[j],&dueTime[j],&procTime[j],&classOf[j]);

   setupTime = IlcIntArray((nClasses+1)*(nClasses+1));
   setupCost = IlcIntArray((nClasses+1)*(nClasses+1));

   for (i = 0; i < nClasses+1; i++)
       for (IlcInt j = 1; j < nClasses+1; j++)
           fscanf(data,"%d",&setupTime[i*(nClasses+1)+j]);

   for (i = 0; i < nClasses+1; i++)
       for (IlcInt j = 1; j < nClasses+1; j++)
           fscanf(data,"%d",&setupCost[i*(nClasses+1)+j]);

   fclose(data);
}
```

```
int main(int argc, char **argv){
   IlcInt j, k;
   if (argc >= 2)
       readData(argv[1]);
   else{
       IlcOut << "No filename specified!" << endl; return 2;
   }

   IlcIntVarArray JobForSlot(nSlots+1,0,nJobs);
   IlcIntVarArray SlotForJob(nJobs+1,0,nSlots);
   IlcIntVarArray ComplTime(nJobs+1,0,IlcMax(dueTime));

   JobForSlot[0].setValue(0);

   for (k = 0; k < nSlots+1; k++)
       IlcPost( SlotForJob[JobForSlot[k]] == k );

   for (k = 0; k < nSlots; k++)
       IlcPost( ComplTime[JobForSlot[k]] ==
          IlcMin( dueTime[JobForSlot[k]],
             ComplTime[JobForSlot[k+1]]
                - setupTime[classesOf(JobForSlot[k],JobForSlot[k+1])]
                - procTime[JobForSlot[k+1]] ) );

   IlcPost(
       ComplTime[JobForSlot[nSlots]] == dueTime[JobForSlot[nSlots]] );

   for (j = 1; j < nJobs+1; j++)
       if (classOf[j-1] == classOf[j])
           IlcPost( SlotForJob[j-1] < SlotForJob[j] );

   // These constraints are redundant but do speed the search
   IlcPost( IlcAllDiff(JobForSlot) );
   IlcPost( IlcAllDiff(SlotForJob) );

   IlcIntVarArray DCost(nJobs+1); DCost[0] = IlcIntVar(0,0);
   for (j = 1; j < nJobs+1; j++)
       DCost[j] = duePen[j] * (dueTime[j] - ComplTime[j]);

   IlcIntVarArray SCost(nSlots+1); SCost[0] = IlcIntVar(0,0);
   for (k = 1; k < nSlots+1; k++)
       SCost[k] = setupCost[classesOf(JobForSlot[k-1],JobForSlot[k])];

   IlcIntVar TotalCost = IlcSum(DCost) + IlcSum(SCost);

   IlcGoal goal = IlcGenerate(JobForSlot);

   if (IlcMinimize(goal, TotalCost))
       IlcOut << "Optimal Solution: " << TotalCost.getValue() << endl;
   else
       IlcOut << "No Solution" << endl;

   IlcEnd(); return 0;
}
```

References

[1] E.M.L. Beale and J.A. Tomlin, Special Facilities in a General Mathematical Programming System for Non-Convex Problems Using Ordered Sets of Variables. In J. Lawrence, ed., *OR 69: Proceedings of the Fifth International Conference on Operational Research,* Tavistock Publications, London (1970) 447–454.

[2] J.J. Bisschop and R. Entriken, *AIMMS: The Modeling System.* Paragon Decision Technology, Haarlem, The Netherlands (1993). See also `http://www.paragon.nl/`.

[3] J.J. Bisschop and R. Fourer, New Constructs for the Description of Combinatorial Optimization Problems in Algebraic Modeling Languages. *Computational Optimization and Applications* **6** (1996) 83–116.

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

[5] A. Brooke, D. Kendrick and A. Meeraus, *GAMS: A User's Guide, Release 2.25.* Duxbury Press/Wadsworth Publishing Company, Belmont, CA (1992). See also `http://www.gams.com/`.

[6] A. Colmerauer, An Introduction to Prolog III. *Communications of the ACM* **33** (1990) 69–90.

[7] K. Darby-Dowman, J. Little, G. Mitra and M. Zaffalon, Constraint Logic Programming and Integer Programming Approaches and Their Collaboration in Solving an Assignment Scheduling Problem. *Constraints* **1** (1997) 245–264.

[8] ECRC GmbH, *ECL^iPS^e 3.5: ECRC Common Logic Programming System: User Manual.* European Computer-Industry Research Centre, München (1995). See also `http://www.ecrc.de/research/projects/eclipse/`.

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

[10] R. Fourer, Software Survey: Linear Programming. *OR/MS Today* **24:**2 (April 1997) 54–63.

[11] R. Fourer and D.M. Gay, Expressing Special Structures in an Algebraic Modeling Language for Mathematical Programming. *ORSA Journal on Computing* **7** (1995) 166–190.

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

[13] R. Fourer, D.M. Gay and B.W. Kernighan, *AMPL: A Modeling Language for Mathematical Programming.* Duxbury Press/Wadsworth Publishing Company, Belmont, CA (1992). See also `http://www.ampl.com/ampl/`.

[14] D.M. Gay, Hooking Your Solver to AMPL. Technical report, Bell Laboratories, Murray Hill, NJ (1993; revised 1994, 1997).

[15] E. Hadjiconstantinou and G. Mitra, A Linear and Discrete Programming Framework for Representing Qualitative Knowledge. *Journal of Economic Dynamics and Control* **18** (1994) 273–297.

[16] ILOG S.A., *ILOG Solver User Manual,* Version 3.2; *ILOG Solver Reference Manual,* Version 3.2. ILOG, Inc., Mountain View, CA (1996). See also `http://www.ilog.com/`.

[17] C. Jordan and A. Drexl, A Comparison of Constraint and Mixed-Integer Programming Solvers for Batch Sequencing with Sequence-Dependent Setups. *ORSA Journal on Computing* **7** (1995) 160-165.

[18] B. Kristjansson, *MPL Modelling System User Manual,* Version 2.8. Maximal Software Inc., Arlington, VA (1993). See also `http://www.maximal-usa.com/`.

[19] J.-L. Lauriere, A Language and a Program for Stating and Solving Combinatorial Problems. *Artificial Intelligence* **10** (1978) 29–127.

[20] K. McAloon and C. Tretkoff, 2LP: Linear Programming and Logic Programming. In V. Saraswat and P. Van Hentenryck, eds., *Principles and Practice of Constraint Programming,* MIT Press, Cambridge, MA (1995) 101–116.

[21] K.I.M. McKinnon and H.P. Williams, Constructing Integer Programming Models by the Predicate Calculus. *Annals of Operations Research* **21** (1989) 227–246.

[22] G. Mitra, C. Lucas, S. Moody and E. Hadjiconstantinou, Tools for Reformulating Logical Forms into Zero-One Mixed Integer Programs. *European Journal of Operational Research* **72** (1994) 262–276.

[23] J.P. Paul, LINGO/PC: Modeling Language for Linear and Integer Programming. *OR/MS Today* **16:**2 (1988) 19–22. See also `http://www.lindo.com/`.

[24] J.-F. Puget, A C++ implementation of CLP. *Proceedings of SPICIS 94,* Singapore (1994).

[25] B.M. Smith, S.C. Brailsford, P.M. Hubbard and H.P. Williams, The Progressive Party Problem: Integer Linear Programming and Constraint Programming Compared. *Constraints* **1** (1996) 119–138.

[26] L. Sterling and E. Shapiro, *The Art of Prolog: Advanced Programming Techniques,* 2nd ed. MIT Press, Cambridge, MA (1994).

[27] J.A. Tomlin, Branch and Bound Methods for Integer and Non-Convex Programming. In J. Abadie, ed., *Integer and Nonlinear Programming,* American Elsevier Publishing Company, New York (1970) 437–450.

[28] P. Van Hentenryck, *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, MA (1989).