Extending an Algebraic Modeling Language to Support Constraint Programming

Robert Fourer

Department of Industrial Engineering and Management Sciences
Northwestern University
Evanston, Illinois 60208-3119

4er@iems.northwestern.edu
http://www.iems.northwestern.edu/~4er/

David M. Gay

Bell Laboratories, Lucent Technologies
Murray Hill, New Jersey 07974

dmg@research.bell-labs.com
http://www.cs.bell-labs.com/~dmg/

Algebraic modeling languages have become a standard tool in the development of linear and nonlinear programming applications [17], but they have had much less influence in the area of combinatorial or discrete optimization. Their one great contribution in that area has been to help analysts develop integer linear programming models for solution by general-purpose branch-and-bound procedures. Although this approach has been applied successfully in many cases, formulations as integer programs often lack the intuitive connection to the modeler's original concept of the problem that was the motivation for modeling languages to begin with. Branch-and-bound codes still have great difficulty solving many integer programs, moreover, particularly ones that derive from highly combinatorial problems whose formulations involve great numbers of zero-one variables.

The idea of a modeling language is not necessarily in conflict with the needs of discrete optimization. Indeed the concept of a modeling language for combinatorial problems appears as early as the 70s as the central idea of Lauriere's ALICE [18]:

> . . . the computer receives as data not only numerical values of some parameters but also and mainly the formal descriptive statement of distinct problems belonging to a rather large area.

Much of the underlying structure for expressing linear or integer programs, including simple and multidimensional sets, data indexed over sets, and basic mathematical operations on numbers and sets, is equally useful in describing combinatorial constraints. Integer-valued variables also have a natural role in various combinatorial problems. An extension to permit set-valued variables was shown by Bisschop and Fourer [1] to extend the naturalness of an algebraic modeling language to a variety of discrete problem types. Other kinds of extensions specifically motivated by constraint programming were proposed by Coullard and Fourer [3] and Fourer [4]. Hürlimann's LPL [12] augmented the standard features of an algebraic language by implementing a variety of logical and counting operators useful to combinatorial problems, with the option of automatic conversion to equivalent integer programming problems.

The principal barrier to extensions of these kinds did not lie in their lack of expressiveness, but in their lack of connections to solvers that could deal generally and directly with a variety of combinatorial objective and constraint types. Solvers that could satisfy this need were in fact under development, but independently, as the focus of what came to be known as *constraint programming* [19, 22]. Like branch-and-bound, constraint programming solvers were based on a tree search, but using domain reduction heuristics rather than LP subproblem bounds to prune the tree to manageable size. Their branching strategies and pruning procedures were tailored to a variety of general modeling operators and structures valuable in modeling discrete optimization problems.

Constraint programming solvers are used within modeling environments based on a variety of languages, notably Prolog and C++. Most recently, their suitability for supporting combinatorial extensions to the customary algebraic modeling language notation has been demonstrated by the OPL language [23], which, through the associated OPL Studio software [15], supports both ILOG's CPLEX [14] for linear and integer programming and its Solver [16] for constraint programming. In addition to language extensions for model formulation, OPL incorporates a syntax for providing nondeterministic directives to guide the tree search.

Given these developments, there is reason to be confident that algebraic modeling languages will make an increasingly valuable contribution to the formulation and testing of combinatorial models. Nevertheless, to fully realize their potential these languages will have to be extended not only in the variety of problems that they can express, but also in the variety of solvers that they can support. The ability to support a diverse range of competing solvers has been a strong force behind the popularity of modeling languages for linear and integer programming, and the user of these languages will expect no less as they continue to be extended. Yet the approaches currently employed to interface modeling languages to a range of linear and nonlinear solvers are inadequate to the needs of constraint programming solvers, because information about objective and constraint functions is not conveyed directly to the solvers, but rather indirectly through function evaluations.

The goal of this paper is thus to elucidate the mechanisms that algebraic modeling languages will require, if their current flexibility is to be extended for convenient connections to varied constraint programming solvers. We begin by discussing (in Section 1) the design and use of conventional modeler-solver interface libraries that provide connections to existing linear and nonlinear solvers. Our description emphasizes in particular how solver-specific drivers interact with a general-purpose interface library to accommodate the requirements of diverse algorithms and their data structures. We then introduce (in Section 2) the kinds of complications to be expected in writing drivers for constraint programming solvers, particularly the need to recursively "walk" an expression-tree representation within the driver code so as to provide solvers with function descriptions rather than evaluations. This material is applicable not only to constraint programming for combinatorial optimization but to other global search solvers, such as those that have recently been developed for global optimization of continuous nonlinear functions.

We subsequently present detailed discussions of the issues raised by particular extensions for "logical" constraints (Section 3), for constraints that employ "counting" operators (Section 4), for special-structure constraints such as "all different" (Section 5), and for expressions using variables in the "subscripts" of model parameters and variables (Section 6). Our account concludes (in Section 7) with a summary of other extensions being implemented and brief indications of the difficulties involved.

We illustrate our ideas by reference to a driver for the AMPL modeling language [5, 6] that we have implemented via ILOG's Concert Technology C++ interface [13] to solve constraint programs using ILOG Solver [16]. We use examples of AMPL declarations and Concert code to make our points explicit, but we have tried to choose and present these examples in a way that will permit the reader to appreciate their essential features even without prior knowledge of AMPL or C++.

Our discussion also addresses the general issues involved, in a way that we hope will encourage other applications of algebraic modeling languages in constraint programming. Although AMPL and ILOG Solver are proprietary, source code for the AMPL-to-solver interface libraries and for the driver routines described herein is publicly available from `netlib.bell-labs.com/netlib/AMPL/solvers` or `www.netlib.org/AMPL/solvers`. Over a dozen other AMPL drivers are also provided at these sites and have been used extensively over the past decade.

We use *CP* as an abbreviation for constraint program or constraint programming, and *IP* for integer program or integer programming, which we take to include the "mixed" case in which some variables are integer and others continuous. We follow the mathematical programming terminology that a *solution* is any assignment of values to variables, a *feasible* solution is one that satisfies all constraints, and an *optimal* solution is a feasible solution that minimizes or maximizes the objective.

1. Modeling language interfaces to solvers

Optimization systems based on algebraic modeling languages maintain a problem in the form of a symbolic model together with explicit data. Both the model and data can be entered and manipulated in various ways, but our main concern here is with what happens when the user asks to solve the current problem.

This section gives a top-down summary of current solver interfaces for modeling languages that are intended to work with a variety of solvers. We first review the solver invocation process in general terms, then describe the interface code — the solver "driver" — in more detail. Finally, we describe the forms in which individual constraints are represented.

Overall flow of control. Upon receiving a "solve" request, the modeling system generates from the current model and data a particular optimization problem, or problem *instance*, in a format that has been designed to be flexible and easy to generate. In contrast, the input format required by a solver is designed to concisely express the kinds of problems handled by that solver, in a form that is convenient for that solver to process. Thus the instance representation created by a general-purpose modeling system cannot be appropriate for direct input to any solver. Instead, the instance must be sent to a *driver*, an interface that converts between the modeling system's representation and the representation required by the solver. Typically each solver connected to the system has its own driver, which performs a series of conversions before it invokes the solver.

In the case of the AMPL modeling system that serves as our example, the commands leading up to an invocation of a solver named SuperSolv might be:

```
option solver supersolv;
option supersolv_options "maxiter=10000";
solve;
```

The `solve` command causes a problem instance to be generated from the current model and data and to be written to a temporary file in AMPL's general "`nl`" format. AMPL generates an arbitrary name or stub — say, `at13151` — and names the file `at13151.nl`. It then invokes SuperSolv's driver by executing the command

```
supersolv at13151 -AMPL
```

The driver in turn reads `at13151.nl` and any solver directives in `supersolv_options`, transforms the representation of the instance as necessary, passes the transformed instance to the routines of SuperSolv, and retrieves the solution that SuperSolv reports. Finally the driver writes a representation of the solution information to a second file, `at13151.sol`, and terminates. Upon the driver's termination, AMPL reads `at13151.sol`, prints a brief summary, and stores the rest of the information for retrieval by subsequent commands.

Other modeling systems communicate with drivers through data structures in memory rather than through explicit files, but either way this is a design that favors flexibility. A driver is "installed" simply by placing its executable anywhere in the current search path. Once a problem instance has been written and a solver driver has been invoked, the entire solution-finding process runs independently of the modeling system. Usually the modeling system remains an active process while the driver and solver are running, but even that is not necessary. The driver is typically compiled together with its solver, but the driver can also be an independent program that sends problem instances to some remote location where the solver runs.

The alternative is to tightly integrate the solver with the modeling system, so that the driver is no longer a separate component. Although such an approach has a potential to be more efficient, the prospective gain in efficiency is often small, because solving time tends to dominate communication time. The main advantage of tight integration is to trade some flexibility for a greater degree of control. As a result this approach is most attractive for integrating a modeling language and solvers that belong to the same developer, as in the case of ILOG's OPL Studio [15].

Structure of the driver. Each driver must contain some parts written specifically for the solver being interfaced. A driver may also use code that is common to all drivers for a particular modeling system. This latter code, comprising interface routines and header files for an interface data structure, makes up an *interface library* typically supplied by the modeling system's developer. In the case of AMPL, an AMPL-solver interface library written in C is freely available from netlib; our examples will refer to this library as ASL and to its components as the ASL routines and ASL data structures.

In terms of the solver-specific code and the interface library routines, the essential steps of a driver can be described as follows:

▷ Call interface routines to load a problem instance from the modeling system into the interface data structure.

- ▷ Apply solver-specific processing to convert information from the interface data structure to the forms and structures that the solver requires as its input.
- ▷ Invoke the solver and wait for it to complete its run.
- ▷ Apply solver-specific processing to convert the solver's output back to the form of the interface data structure.
- ▷ Call interface routines to send the results back to the modeling system.

For linear, quadratic, and related optimization solvers that receive all of their information about the problem from data structures passed as input arrays, the interface routines are used only as indicated in these steps. In the case of more general nonlinear programming, however, the situation is more complex.

A traditional nonlinear solver generates a series of trial solutions, or iterates, and requires the values of nonlinear objective and constraint functions only at those iterates. Thus the solver typically requires each user to provide a routine, in a programming language such as C or Fortran, that can be called with the values of the variables at the current iterate, and that returns the objective and constraint function values and optionally their derivative values. The calling conventions for this routine differ from one solver to the next, while the body of the function is entirely specific to the problem to be solved.

When a traditional nonlinear solver is hooked to a driver, the developer of the driver supplies a generic function evaluation routine to take the place of the routine that would otherwise be provided by users. This generic routine adheres to the solver's particular calling conventions, but it performs the function evaluations by passing the variables' values to the interface library's evaluation routine, which computes the function values and derivatives by use of the interface data structure that the driver previously set up. Thus the interface routines and data structure are used not only by the driver program, but also by the solver as it evaluates iterates on the path toward a solution.

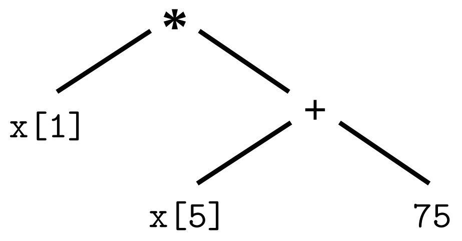
Representation of constraints. Any constraint written in an algebraic modeling language can be converted to a standard algebraic form,

$$\textit{lower-bound} \;\leq\; \textit{linear-expr} + \textit{nonlinear-expr} \;\leq\; \textit{upper-bound}.$$

Each modeling system performs a conversion to this or a very similar form as part of its translation from model and data to problem instance. Hence this form is explicit in the format of the interface library's data structure. (Objectives are also expressed as *linear-expr* + *nonlinear-expr*, but without the bounds. The extensions to the expression forms described in this paper are applicable to objectives as well as constraints, but overall the consequences for constraint forms and processing are much more significant.) The constraint lower-bound and upper-bound are numerical values that occupy arrays in the interface data structure. Single-inequality constraints have one of these bounds equal to an "infinite" value (defined in the interface library's headers), while equality constraints have the two bounds equal.

Each *linear-expr* represents the linear part of a constraint or objective expression. In the interface data structure, the *linear-expr*s for all constraints are gathered together into a sparse coefficient list. This information can be made available to the driver in the usual format for coefficients of a linear program, consisting of a column-wise pair of arrays that hold coefficient values and row indices, plus an array to indicate where the values for each column begin. The same information might alternatively be provided in a row-wise linked list of nonzero coefficients, with an array of pointers specifying the beginning of each linked list, an arrangement that is more convenient for generating input to some nonlinear solvers.

Each *nonlinear-expr* has a representation as an expression tree, with internal nodes representing operators or functions and leaf nodes standing for individual variables or constants:



The interface data structure must incorporate a concise representation of such a tree that will be convenient for evaluation of the *nonlinear-expr* given values of the variables. For traditional nonlinear solvers that rely on the differentiability of constraint and objective functions, the data structure must also facilitate concurrent computation of derivatives by efficient techniques of automatic differentiation [10, 11].

In the case of AMPL, the expression tree is represented in the `nl`-file by a Polish prefix notation, and in the ASL data structure as a directed acyclic graph that is conceptually identical to the tree but with one leaf node for each independent variable. Nodes are implemented as C structures connected by pointers. There are about 10 different node structures in all, corresponding to different kinds of operators and functions as well as variables and constants. A few fields are added to hold intermediate information required in automatic differentiation. Finally, one additional array contains a pointer to the root node of each constraint expression tree.

For efficiency in handling the common case of a sum of terms within a nonlinear expression, AMPL recognizes a special summation node. For example, in the expression tree for

```
(target - sum j in 1..n cost[j] * Buy[j]) ** 2
```

there is one summation node with `n` children representing the `n` summands, rather than a cascade of `n-1` binary `+` nodes. No special attempt is made to distinguish summations that are actually (like this one) linear subexpressions from those that have nonlinear terms.

AMPL's evaluation of a *nonlinear-expr* and its derivatives at given values of the variables can be accomplished efficiently by a straightforward recursive "walk" of its tree [8]. The developer of a driver can choose among several ASL evaluation routines that provide different amounts and formats of first and second derivative information [9]. For any particular problem, the ASL routines for function and derivative evaluation are likely to be slower than compiled C or Fortran routines that have been hand-coded or that have been automatically generated from an `nl`-file by use of the `nlc` utility [8], but the difference is only a moderate linear factor [7]. The convenience and reliability of expressing nonlinear functions by means of a modeling language and performing evaluations via expression trees are almost always sufficient to outweigh any drawbacks due to slower evaluation.

2. Extending modeling language interfaces for global search solvers

The representation of nonlinear expressions described above is sufficient for driving traditional nonlinear solvers, because the methods employed by those solvers are forms of *local search.* They basically progress through a series of trial points and need only the function (and derivative) values at those points.

The methods used by CP solvers are, by comparison, an example of *global search.* They progressively subdivide the feasible region into smaller subproblems until each subproblem is able to be solved. This approach cannot rely on function evaluations alone; it rather requires access to the actual forms of the functions, from which optimal solutions to nontrivial subproblems can be deduced efficiently.

(We distinguish global search from *global optimization,* which refers to any method for seeking the best objective value over all feasible solutions. Global search methods are necessarily a subset of global optimization methods, but global optimization methods may also involve local search — although usually with weaker optimality properties. Heuristic approaches such as simulated annealing, tabu search, evolutionary methods, and a variety of others [2, 20] are examples of global optimization methods that use local search.

The one method of global search commonly supported by algebraic modeling languages is branch-and-bound. In their role as IP solvers, branch-and-bound codes get all of the information they need from coefficient-list representations of linear expressions as described in the previous section. Most of these codes also handle extensions, however, such as for variables that have arbitrary finite domains and for separable piecewise-linear expressions. (The mechanisms for doing so are known as special ordered sets of various types.) The presence of these extensions must also be communicated explicitly to the branch-and-bound solver, and indeed that is the motivation for the piecewise-linear notation in several modeling languages.

To support the more general methods of global search characteristic of constraint programming, a driver must also convert non-linear information in the expression trees to forms that the solver requires. This conversion has to be completed before the solver is invoked, so that the solver has all the information it needs at the outset. Thus in most cases a complete scan of the constraint expression trees must be performed by the solver-specific code in the driver, rather than being left to interface function-evaluation routines that are later called back from the solver as it iterates.

The remainder of this section introduces expression-tree processing in drivers for global search solvers, by considering applications to expressions already available in algebraic modeling languages. We first briefly introduce the example we will be using, and then describe the general code for constraint generation and the specific cases that make up the tree-walk routine. Subsequent sections will address complications that are introduced by CP extensions.

A driver example. To provide a concrete illustration, we describe a driver that uses ILOG's Concert Technology C++ interface [13]. The Concert interface provides tools for constructing a problem data structure specific to ILOG's optimization products. For the work described here, our interest is in using Concert to build an AMPL driver for ILOG Solver [16], which is capable of applying global search methods to a broad variety of nonlinear, integer, and constraint programming problems. The design of such a driver necessarily brings up many key challenges of constraint programming via an algebraic modeling language.

The Concert Technology interface uses C++ object classes and operator overloading to provide an expression and constraint syntax that has some resemblance to algebraic notation. Thus a C++ expression of the form `exSum += exCoef * exVar[k]` resembles the adding of a linear term to a partial sum. When the entities involved have certain Concert object types, however, this expression actually adds a Concert representation of a linear term to a Concert representation of an expression. Specifically, the `*` operator is overloaded so that when its left operand `exCoef` is an object of type `IloNum` and its right operand `exVar[k]` is an object of type `IloNumVar`, the result is an object of type `IloExpr`. When the overloaded operator `+=` has this `IloExpr` object on its right and another `IloExpr` object `exSum` on its left, the effect is to update the left `IloExpr` to reflect the addition of the term represented by the right `IloExpr`.

As another example, in the C++ statement `exIneq[i] = (exSum <= exBnd[i])`, the `<=` operator is overloaded so that when its left operand `exSum` is an object of type `IloExpr` and its right operand `exBnd[i]` is of type `IloNum`, the result is an `IloRange` object. `IloRange` is a subclass of `IloConstraint` that represents algebraic equations and inequalities in the same way that algebraic modeling languages do, as an expression subject to lower and upper bounds. The overloaded assignment operator `=` then gives the value of the `IloConstraint` expression on its right to the `IloConstraint`-valued variable `exIneq[i]` on its left.

Like any C++ class library, Concert represents objects through data structures whose internal details need not concern the writer of the driver. Objects are created and manipulated entirely by member functions and operators. Even the expression `exVar[k]` above, which would appear to be an element selected from an array, is in fact an overloading of the subscripting operator `[]` applied to the `IloNumVarArray` object `exVar` and the `int k`, returning an object of type `IloNumVar` as its result.

Processing constraints. Given the arrangement we have described, the driver's work is to convert the contents of the ASL data structure to C++ Concert code. The top-level statements for this purpose simply declare an `IloNumVarArray` containing the appropriate numbers of continuous and integer variables:

```
Var = IloNumVarArray (env,n_var);

for (j = 0; j < n_var - n_var_int; j++)
   Var[j] = IloNumVar(env, loVarBnd[j], upVarBnd[j], ILOFLOAT);
for (j = n_var - n_var_int; j < n_var; j++)
   Var[j] = IloNumVar(env, loVarBnd[j], upVarBnd[j], ILOINT);
```

Then the constraint processing loop is easily written like this:

```
IloRangeArray Con(env,n_con);

for (i = 0; i < n_con; i++) {

   IloExpr conExpr(env);

   for (cg = Cgrad[i]; cg; cg = cg->next)
      conExpr += (cg -> coef) * Var[cg -> varno];

   if (i < nlc)
      conExpr += build_expr (con_de[i].e);

   Con[i] = (loConBnd[i] <= conExpr <= upConBnd[i]);
}
```

Following initialization of an `IloRangeArray` object `Con` of appropriate length, each pass through the loop builds and saves an `IloRange` object `Con[i]` corresponding to one of the AMPL constraints. The loop's first statement creates an empty expression object, the second and third statements add the linear and nonlinear parts to the expression object, respectively, and the fourth uses the expression object to create and save the appropriate Concert constraint object.

More specifically, any linear part of the expression object `conExpr` is built term by term in the second loop statement, by stepping through the ASL linked list of coefficients starting at `Cgrad[i]`. If there is a nonlinear part, the driver's function `build_expr` is called in the third loop statement to construct an object of type `IloExpr` that corresponds to the ASL expression tree at `con_de[i].e`, and this object is added to `conExpr`. Finally, `conExpr` together with the lower and upper constraint bounds from the ASL data structure is used to create an `IloRange` constraint that is saved as `Con[i]`.

Walking the expression trees. The routine `build_expr` must be written by us as part of the driver. It takes as its argument an expression tree, and returns an `IloExpr` object that is equivalent to the expression represented by the tree:

```
IloExpr build_expr (expr *e)

{  ...

   opnum = (int) e->op;

   switch(opnum) {

      case PLUS_opno: ...

      case MINUS_opno: ...

      ...
   }
}
```

More specifically, this function's argument points to an instance `e` of an ASL structure, `expr`, that represents expression tree nodes. The `opnum`, extracted from a field of `e`, indicates which of the many AMPL operators and functions is represented. Hence each case of the `switch` on `opnum` handles a different operation.

Most cases are handled recursively, by calling `build_expr` to create `IloExpr` objects for each argument or operand, and then returning the appropriate Concert expression as the result. Because AMPL and Concert use many of the same standard algebraic operators and functions, many cases require only a single statement:

```
case PLUS_opno:  // x + y
   return build_expr (e->L.e) + build_expr (e->R.e);

case LOG_opno:  // log x
   return IloLog (build_expr (e->L.e));
```

Even where there is no exact match, an appropriate Concert expression is not hard to construct.

The special ASL node for an iterated `sum` specifies an array of nodes whose associated expressions are to be summed. Thus the code for this case resembles the accumulation of linear terms previously shown in our example from the main driver program. The difference is that the `IloExpr` object for each term is now created not by a multiplication operator, but by a recursive call to `build_expr`:

```
case SUMLIST_opno:  // iterated sum
   partSum = IloExpr(env);
   for (ep = e->L.ep; ep < e->R.ep; *ep++)
      partSum += build_expr (*ep);
   return partSum;
```

Similar code handles other iterated operators, though sometimes with additional complications due to differences between AMPL operators and Concert functions.

Two kinds of leaf nodes provide the base cases for the recursion. A node that corresponds to a numerical constant causes `build_expr` to return an expression defined to be fixed at the constant's value:

```
case NUM_opno:  // constant
   return IloExpr (env, ((expr_n*)e)->v);
```

A node that corresponds to a decision variable causes `build_expr` to return an `IloNumVar` from `Var`, our Concert `IloNumVarArray` of variables:

```
case VARVAL_opno:  // variable
   return Var[e->a];
```

Concert provides for an `IloNumVar` to be converted to the specified return type `IloExpr`. If returning an `IloExpr` object for each constant or variable proved to be a serious inefficiency, then the other cases could instead be modified to test for operands that are merely constants or variables and to handle those cases specially without a recursive call to `build_expr`.

3. Extensions for logical constraints

Logical operators are already a standard feature of algebraic modeling languages, as they are necessary for specifying conditions that define the membership of indexing sets. In that context they apply only to the data, however. AMPL goes further by allowing constraints to contain a kind of `if-then-else` expression whose value depends on a condition involving variables:

```
subject to logRel {j in 1..N}:
   (if X[j] < -delta || X[j] > delta
      then log(1+X[j]) / X[j] else 1 - X[j] / 2) <= logLim;
```

The condition following `if` may use any of the arithmetic relational operators (like `<`) and logical operators (like `||` for "or"). Relational and logical operators may thus appear in the resulting expression tree, but for traditional nonlinear solvers they are involved only in conventional function evaluation. Given a particular iterate produced by the solver, the condition following `if` is tested to determine whether the expression following `then` or following `else` will be evaluated to determine the value of the entire `if-then-else` expression.

If variables are allowed more generally within expressions that use logical operators, then an algebraic modeling language can express the much broader variety of logical constraints that are to be found in CP formulations. An AMPL user formulating a scheduling problem might want to write two inequalities that are arguments to the `||` (`or`) operator, for instance:

```
subject to NoOverlap {j1 in 1..nJobs, j2 in j1+1..nJobs}:
   Start[j2] >= Start[j1] + setTime[j1,j2] ||
   Start[j1] >= Start[j2] + setTime[j2,j1];
```

A modeler developing an assignment problem could use a combination of `!` (`not`) and `&&` (`and`) to rule out solutions that unnecessarily "isolate" any individuals:

```
subject to NoIsolation {(i1,i2) in ISOPAIRS, j in GROUPS}
   ! (Assign[i1,i2,j] = 1 &&
      sum {ii1 in ADJ[i1], (ii1,i2) in TYPE} Assign[ii1,i2,j] = 0);
```

AMPL also provides an iterated operator, `forall`, that has the effect of connecting an indexed collection of operands by a series of `and` operations. Thus `forall` can be used in a constraint to represent the assertion that all equalities in some indexed collection must hold, as in this example from a transshipment model:

```
subject to BuildDefn {i in CENTERS}:
   (Build[i] = 1 && sum {j in CUST} Ship[i,j] <= cap[i]) ||
   (Build[i] = 0 && forall {j in CUST} Ship[i,j] = 0);
```

A similar `exists` operator connects equations or inequalities by a series of `||` operators. AMPL's `forall` and `exists` have the same syntax as `sum`, and bear the same relationship to the binary logical operators that `sum` bears to the `+` operator.

New logical operators of these kinds are readily added to a modeling language's constraint syntax. In our AMPL illustrations, their general forms are

> `!` *constraint-expr*
> *constraint-expr* `||` *constraint-expr*
> *constraint-expr* `&&` *constraint-expr*
> *exists* {*indexing*} *constraint-expr*
> *forall* {*indexing*} *constraint-expr*

where *constraint-expr* is any valid expression for a constraint, whether a simple equation or inequality or a more complex assertion built from arithmetic relations already connected by logical operators. (Parenthesization and operator precedence are handled in the usual ways.) It is convenient to think of the logical operators as "taking constraints as operands" even though the operands do not stand by themselves as constraints on the problem to be solved.

Because logical operators take constraints as operands, logical constraints do not fit the customary algebraic form of an expression between bounds. Hürlimann's LPL modeling language [12] was the first (to our knowledge) to attack this problem, by running each logical constraint through a series of transformations that convert it to a conjunctive normal form and then to an algebraic constraint in zero-one variables, which can be sent to any branch-and-bound solver for integer programming. Substantial transformations like this are less appealing for constraint programming, however, as they may hide from the solver some useful information about the modeler's choice of formulation. Moreover, any standard form adequate for the logical operators will again be insufficient as the language is expanded with further types of operations and constraints motivated by constraint programming.

We thus anticipate that algebraic modeling languages will follow a more flexible strategy, by generalizing their existing expression tree representations so as to record logical constraints in more-or-less the same form as the modeler writes them. This is a particularly straightforward extension in the case of AMPL, since nodes for logical operators (`&&`, `||`, `!`) and for binary relational operators (`<`, `<=`, `=`, `>=`, `>`, `!=`) have already been defined to meet the needs of the previously discussed `if-then-else` expressions. Instances of `forall` and `exists` are represented by new node types analogous to those for `sum`. A double inequality is converted to an `&&` between two single inequalities (`lo[j] <= X[j] <= hi[j]` becomes `lo[j] <= X[j] && X[j] <= hi[j]`, for instance), but it would not be hard to introduce more specialized nodes corresponding to double inequalities, if that would be helpful to some solvers.

By taking this approach, we introduce a kind of non-algebraic constraint that is represented in its entirety within an expression tree. Top-level processing for such a constraint is particularly simple, since all of the relevant information can be reached from its tree's root node. In our Concert driver, a pointer to the root of each constraint's expression tree is merely passed to our driver routine `build_constr` that returns an equivalent `IloConstraint`:

```
IloConstraintArray LCon(env,n_lcon);

for (i = 0; i < n_lcon; i++) {
   LCon[i] = build_constr (lcon_de[i].e);
}
```

The object returned from `build_constr` does not require any further processing, because logical constraints do not have components — like an algebraic constraint's linear coefficients and bounds — that are maintained in arrays or lists outside of the expression tree.

The `build_constr` routine for our example looks much like the previously described `build_expr`, though with an `IloConstraint` rather than `IloExpr` return type and with cases for operators that create non-algebraic constraints:

```
IloConstraint build_constr (expr *e)

{  ...

   opnum = (int) e->op;

   switch(opnum) {

      case OR_opno: ...

      case AND_opno: ...

      ...
   }
}
```

Cases for the comparison operators are handled straightforwardly by their counterparts in C++, which are overloaded by Concert to take `IloExpr` operands and return an `IloConstraint` result:

```
case GE_opno:  // >=
   return build_expr (e->L.e) >= build_expr (e->R.e);
```

The logical operators are handled similarly, but their arguments are what we have called *constraint-expr*s. Concert overloads their C++ counterparts so that they take `IloConstraint` operands, which we can also build by calls to `build_constr`:

```
case OR_opno:   // || (or)
   return build_constr (e->L.e) || build_constr (e->R.e);
```

Thus `build_constr` is used recursively to build up more complex logical constraints from simpler constraint expressions. Base cases for `build_constr` are via its eventual calls to `build_expr`, as there are no "true" or "false" literal values in the tree that AMPL sends to the driver.

Once a framework of this sort has been set up, it is a straightforward matter to add other logical operations, such as xor and implication, that might be convenient for modeling. The `if-then-else` operator that returns a numerical value, as described at the beginning of this section, can also be accommodated for CP purposes by adding an appropriate case to the `build_expr` routine.

4. Extensions for counting

Most algebraic modeling languages can define the set of all objects or numbers that satisfy given logical conditions. By counting the number of members in a set that has been so defined, a modeling language can in effect count the number of conditions in a given collection that are satisfied. If sets were allowed to be defined in terms of variables, then modeling languages could apply counting to conditions on the variables — that is, they could count the number of constraints in a given collection that were satisfied. Counting expressions of this kind are an important component of CP formulations. The incorporation of variables into set definitions is arguably too general an extension for this purpose, however; instead it makes sense to define "counting" operators specially.

As an illustration, in a transshipment problem the number of customers that have at least `min_ctn` cartons of demand could be represented for all products `p` by applying AMPL's `card` (cardinality) operator to the appropriate set expression:

```
param num_min_ctn {p in PROD} :=
   card {c in CUST: demand[c,p] > min_ctn};
```

Allowing variables in this sort of expression would amount to allowing the condition after the colon to be any *constraint-expr* as defined previously. Then to require, for instance, that the number of warehouses shipping a given product to a given customer may not exceed `max_store`, an AMPL model could state:

```
subj to Max_Whse_Used {p in PROD, c in CUST}:
   card {w in WHSE: Ship[w,c,p] > 0} <= max_store;
```

The use of `card` for this purpose is arguably unnatural, since we think of the constraint in terms of the number of conditions that are satisfied, without reference to any kind of set. Also, since set expressions are used in many contexts throughout the AMPL language, allowing variables to appear in them in only certain contexts in certain constraint expressions would entail new and potentially confusing rules.

We thus begin by describing an extension that provides a more specialized integer-valued counting operator for use as an expression in constraints. Even this operator can be awkward in common circumstances, however, and thus we subsequently describe alternative operators that generate constraints directly by fixing or bounding specified counts.

Counting expressions. Like any modeling language operator that works with a collection or list, a counting operator is fundamentally an indexed operator. Indeed, it works much like summation, except that instead of summing the numerical values of its operands, it sums 1 for each operand that holds and 0 for each that does not. Thus an algebraic modeling language could provide this operator by adapting its summation syntax, but with constraint expressions replacing numerical ones.

We have basically followed this approach in designing a new AMPL operator, `count`, that expresses the constraint shown above by:

```
subj to Max_Whse_Used {s in STORE, p in PROD}:
   count {w in WHSE} (Ship[w,s,p] > 0) <= max_store;
```

Thus `count` is in fact written much like `sum`, except that the argument following the indexing expression must be parenthesized (to avoid certain grammatical ambiguities). To provide more flexibility in the list of constraints to be counted, however, we provide an unindexed as well as an indexed form:

`count (`*constraint-list*`)`
`count {`*indexing*`} (`*constraint-list*`)`

The *constraint-list* can be a single *constraint-expr* as in our example above, or more generally a list that may itself contain indexed sub-lists. This is consistent with the forms of other kinds of lists in AMPL, such as lists of arguments to user-defined functions and lists of items in `display` statements — as well as lists in other new operators motivated by constraint programming. We expect that the simple indexed `count` will be the most widely used, however.

Like other iterated operators in constraints, `count` is translated by AMPL to provide an explicit list of operands in the problem sent to the solver. Thus the ASL data structure accommodates `count` by defining a new kind of node that identifies an array of constraint nodes, each processed by a call to `build_constr`. The result of the `count` operation is an arithmetic value, however, and so it is processed in `build_expr` to return an `IloExpr` object. In fact the C++ code for the case of `count` is nearly the same as previously shown for iterated `sum`s:

```
case COUNT_opno:  // count
   partSum = IloExpr(env);
   for (ep = e->L.ep; ep < e->R.ep; *ep++)
      partSum += build_constr (*ep);
   return partSum;
```

The only difference is the substitution of `build_constr` for `build_expr` inside the loop. This works because the Concert interface reduces counting to the kind of summation that we previously described; specifically, it overloads arithmetic operators (like +=) so that they treat IloConstraint operands as 1 if they are true and 0 if they are false.

With the implementation of count, the build_expr and build_constr routines each call the other under certain circumstances. Thus they can be regarded as two pieces of a single recursive tree-walk procedure.

Counting constraints. Even with the use of count, our Max_Whse_Used constraint example ends awkwardly with the sequence

```
... > 0) <= max_store;
```

Thus, given that the purpose of modeling languages is to describe people's models in natural terms, there is reason to prefer a more direct way to say that there are at most max_store warehouses shipping any given product to a given store. This leads to the definition of the atmost operator, with which our AMPL constraint becomes

```
subj to Max_Whse_Used {s in STORE, p in PROD}:
   atmost max_store {w in WHSE} (Ship[w,s,p] > 0);
```

Similar operators atleast and exactly have the obvious analogous meanings. The syntax is the same as for count, except for the insertion of an arithmetic expression after the keyword.

These alternative operators return constraints that are merely bounds on the value of the corresponding count operator. Hence their realization in the LPL modeling language [12] simply reduces them to bounds on the kinds of sums that arise from counting operators. Since for a CP driver we prefer to preserve more of the original structure, our AMPL driver for Concert breaks each atmost, atleast, or exactly expression into an arithmetic expression and a count expression, which are processed within build_constr by the standard calls to build_expr. For example, the code for the case of atmost is:

```
case ATMOST_opno:  // at most
   return build_expr (e->L.e) >= build_expr (e->R.e);
```

This is in fact identical to the code for the AMPL >= operator previously shown.

5. Structure constraints

"All-different" constraints, stating that a specified list of expressions must all take different values, are a common component of CP model formulations. They are one example of *structure* constraints that jointly restrict the values of many variables in a highly regular way. (They are called *global* constraints in the CP literature, but we avoid that term here so as not to compound the confusion that already exists between the terms "global convergence" and "global optimization" in mathematical programming.)

From the modeler's point of view, a structure constraint is attractive when it concisely states a condition that would otherwise require a great number of other (usually algebraic) constraints. To also be attractive for CP solvers, a structure constraint must be associated with a fast algorithm for pruning large parts of the search tree. That is, given reduced domains for a structure constraint's variables at some node of the search, there must be efficient ways to deduce further domain reductions, to determine that no feasible solution is possible, or to remove domain values that cannot appear in any feasible solution constructed from the current domains. This process of *domain reduction* and *filtering* — akin to the *presolving* and *probing* employed by some branch-and-bound codes — is typically applied at each node in the course of the CP solver's tree search.

We describe here two structure constraints — all-different and its generalization, number-of — that are readily provided to modeling language users through the introduction of new operators. We then review the issues involved in deciding which structure constraints merit support in a general-purpose modeling language.

All-different. For the all-different constraint, the appeal to modelers is clear: a single list of the expressions that must all take different values replaces a much larger number of inequalities between all pairs of such expressions. The filtering process is simply a series of assignment (or matching) problems.

A modeling language extension for all-different is similarly straightforward. It requires only a new keyword and a way to specify a list of expressions. Lists of expressions are already familiar to AMPL users, through their use in `display` statements for viewing results. They are readily adapted to give a syntax for an all-different constraint:

```
alldiff (expression-list)
alldiff {indexing} (expression-list)
```

This is essentially the same as for `count`, `atmost`, and other new operators previously described, except that it specifies an *expression-list* rather than a *constraint-list.*

As an example, a simple production scheduling problem might be modeled in terms of variables that specify a machine for each waiting job. A corresponding AMPL model could define, for each job `j`, a variable `MachineForJob[j]` whose value would be the machine-number of that job's assigned machine:

```
param nJobs integer > 0;
param nMachines integer > 0;

var MachineForJob {1..nJobs} >= 1, <= nMachines;
```

To say that each machine is to be assigned to a different job, we could list all of the inequalities explicitly,

```
subject to AssignJobs0 {j1 in 1..nJobs, j2 in j1+1..nJobs}:
   MachineForJob[j1] != MachineForJob[j2];
```

Or, by converting the constraint to the equivalent statement that each machine is assigned at most one job, we could employ one of the previously described counting operators, either `count`,

```
subject to AssignJobs1 {i in 1..nMachines}:
   count {j in Jobs} (MachineForJob[j] = i) <= 1;
```

or `atmost`:

```
subject to AssignJobs2 {i in 1..nMachines}:
   atmost 1 {j in Jobs} (MachineForJob[j] = i);
```

Using the `alldiff` operator, however, we can convert the original statement of the constraint clearly and naturally into its statement in AMPL:

```
subject to AssignJobs:
   alldiff {j in Jobs} (MachineForJob[j]);
```

Simple all-different lists like this will be perhaps the most common, especially in introductory examples, but the AMPL *expression-list* syntax allows for much more general lists that might arise in complex applications.

Since `alldiff` is not an algebraic constraint, our AMPL driver for Concert must handle it as a case of `build_constr`. Its processing is much the same as what we previously showed for the `sum` operator, however. For each of a list of operands, we call `build_expr` to construct a corresponding `IloExpr` object. Then instead of the operands being summed, they are gathered together to be sent to `IloAllDiff`, Concert's constructor for all-different constraints. By explicitly constructing an all-different constraint at this point, we ensure that the efficient filtering mechanisms for such constraints will be applied; were we to instead use the `atmost` formulation, the CP solver would receive a separate constraint for each machine and would not recognize the presence of the all-different structure.

Number-of. A more flexible kind of production scheduling allows each machine to accept multiple jobs, up to some specified capacity. In AMPL this could entail defining a capacity parameter,

```
param cap {1..nMachines} integer > 0;
```

and then replacing the constant `1` by the relevant capacity in either of our previous constraint examples,

```
subject to AssignCapJobs1 {i in 1..nMachines}:
   count {j in Jobs} (MachineForJob[j] = i) <= cap[i];
```

or

```
subject to AssignCapJobs2 {i in 1..nMachines}:
   atmost cap[i] {j in Jobs} (MachineForJob[j] = i);
```

Again we have an indexed collection of constraints that might be expressed more cleanly and handled more efficiently if it were treated instead as a single structure constraint. In a CP solver, the filtering process would solve a kind of network-flow problem (which generalizes the assignment problem used for all-different).

It seems logical that an algebraic modeling language should extend to this case as well, but the situation is not nearly as straightforward as in the case of `alldiff`. A single constraint stating that "machine `i` can serve at most `cap[i]` jobs `j`" would have to incorporate the set of machines and their capacities, as well as the set of jobs and the list of job assignments. In the style of AMPL, the syntax might be

```
subject to AssignCapJobs2:
   atmost {i in 1..nMachines} cap[i]
      of i in {j in 1..nJobs} (MachineForJob[j]);
```

Alternatively, the syntax might have more of a functional form, with a keyword followed by listed arguments in parentheses as in the style of OPL [23, 15]. Whatever the syntax, however, the resulting statement falls short in both convenience and naturalness. It tries to cram all of the indexing within the single constraint, whereas in fact what the modeler has in mind is a restriction for each machine on the number of jobs assigned to it.

This analysis suggests that we really do want to define an indexed collection for the constraints we have in mind, only using a constraint syntax more natural and specific for the intended purpose. In fact the only new syntax we require is for an alternative counting operator, one that counts the number of times that a specified value appears in a specified list of values. For AMPL, we can use a syntax like that of `atmost`, but with an *expression-list* like that of `alldiff`:

> `numberof` *target-expr* `in` (*expression-list*)

In our example, we want to count the number of times that machine `i` appears in a the list of machines `{j in 1..nJobs} MachineForJob[j]`:

```
subject to AssignCapJobs {i in 1..nMachines}:
   numberof i in ({j in 1..nJobs} MachineForJob[j]) <= cap[i];
```

Here we only constrain the result of the `numberof` operator to be less than or equal to a constant, but it could be used just as well anywhere that a numerical value involving variables would be allowed.

A straightforward adaptation of our `build_expr` case for `count` suffices to handle `numberof` in any situation. A Concert `IloExpr` object corresponding to the ASL expression tree for the *target-expr* is built and assigned to `targetExpr`. Then the loop accumulates in `partSum` the number of right operands to `+=` that are true:

```
case NUMBEROF_opno:  // number of
   partSum = IloExpr(env);
   ep = e->L.ep;
   targetExpr = build_expr (*ep);
   for (*ep++; ep < e->R.ep; *ep++)
      partSum += (build_expr (*ep) == targetExpr);
   return partSum;
```

The main difference is that the right operand to `+=`, which was `build_constr (*ep)` for the case of `count`, is instead given by `build_expr (*ep) == targetExpr`, which contributes a `1` to the sum when and only when a value from the `numberof` operator's *expression-list* is equal to the value of the operator's *target-expr*.

We need a more intricate implementation to handle a `targetExpr` that is a constant, however, since this is the situation in which a structure constraint is relevant. Specifically, a structure constraint of the kind relevant to `numberof` is built by a call to the Concert interface's constructor `IloDistribute`, which takes three array arguments:

- ▷ `list`, an array of variables;
- ▷ `target`, an array of constants; and
- ▷ `count`, an array of variables of the same length as `target`.

These are interpreted to assert that each element `count[k]` equals the number of times that `target[k]` appears in `list`. A single call to `IloDistribute` thus represents all of an AMPL model's `numberof` operators that have the same *expression-list*, and permits more effective domain reduction and filtering than would be possible if each `numberof` occurrence were to be handled separately.

For our scheduling example, the `IloNumVar` objects in the `list` array correspond directly to the `MachineForJob[j]` variables in the AMPL model, and each constant target[k] equals k, so count[k] is simply the number of times that k appears in the schedule. Our implementation for the AMPL driver is more general, however. The members of the list array may be new IloNumVars set up to equal more general items in the numberof operator's *expression-list.* For each occurrence of numberof that has this *expression-list*, the constant *target-expr* is placed in the target array, and the corresponding IloNumVar object is added to the count array to be returned by build_expr as the value of some numberof operation.

All of the activity surrounding the processing of a numberof node is encapsulated in a driver function build_numberof, so that the case for this operator in build_expr is extended by only a few lines:

```
case NUMBEROF_opno:  // number of
   ep = e->L.ep;

   if ((int) *ep->op == CONST_opno)  // target is a constant
      return build_numberof (e);

   else {
      // same as shown previously
   }
```

In brief, the major work of build_numberof is to check whether the current node's *expression-list* has been seen before; this requires a recursive comparison function for determining whether the subtrees beneath two nodes are the same. Also build_numberof creates a data structure, specific to the driver, that keeps track of the *expression-list*s and *target-expr* constants encountered, and the IloNumVar objects created. Its return value is the IloNumVar that will hold the result value of the numberof operation; hence, as seen above, this value can be returned by build_expr without any further processing.

When the tree-walks for all of the AMPL model's constraints are complete, the contents of the list, target, and count arrays for each of the relevant structure constraints are completely represented in the driver data structure. They are readily extracted and passed via IloDistribute to complete the representation required by the constraint solver.

General issues. We have taken the trouble to describe the handling of number-of in some detail, because it illustrates several of the issues that algebraic modeling languages face in providing modelers with the benefits of structure constraints.

First, concepts most naturally handled by the modeler through specialized expressions may be best processed as structure constraints by a CP solver. We have seen that the numberof operator falls into this category. Piecewise-linear functions of individual variables provide another example. An objective function of piecewise-linear cost terms could be written in AMPL as

```
minimize Total_Cost: sum {i in ORIG, j in DEST}
   <<lim1[i,j], lim2[i,j]; r1[i,j], r2[i,j], r3[i,j]>> Trans[i,j];
```

If the terms are not convex then this is a hard problem, traditionally solved by use of a specialized mechanism (so-called special ordered sets of type 2) in branch-and-bound codes for integer programming. Recently the handling of these terms in the context of constraint programming has received increasing attention. For CP purposes, however, each term in the above sum is treated as if replaced by a variable y[i,j] subject to a structure constraint y[i,j] = <<lim1[i,j], lim2[i,j];

`r1[i,j], r2[i,j], r3[i,j]>> Trans[i,j]`. Constraints of this kind have been shown to admit highly effective domain reduction procedures [21].

Second, there may be a tradeoff between making a modeling concept easy to state and making the corresponding structure constraint easy to detect. The design philosophy of AMPL has been to favor naturalness of expression, as in the `numberof` example, so long as structure detection can be kept reasonably fast. This approach has left more work to the writer of each driver, however. An attractive compromise may be to move some of this work to specialized routines included with the driver interface library, if such work is likely to be the same from one driver to the next. But it is not so clear whether our `build_numberof` routine would be suitable for this treatment.

Finally, every new structure-constraint syntax added to a modeling language introduces some extra complexity, for which it must offer sufficient benefit in compensation. The extension may make the modeling language easier and more natural to use across a reasonably broad variety of applications, for instance. In the case of languages like AMPL, the extension should also offer the prospect of support through drivers for a range of solvers. For general and widely useful structure constraints such as all-different and number-of, the benefits of these kinds seem reasonably clear. But there are many more specialized structure constraints, and for each modeling language the line will have to be drawn somewhere among them. The current Concert interface alone provides functions to construct structure constraints of four additional distinct kinds:

- ▷ `IloAllMinDistance`: Any two among a specified collection of numerical values must be at least a certain distance apart.
- ▷ `IloInverse`: For two specified arrays `x` and `y`, `x[i]` equals `j` if and only if `y[j]` equals `i`.
- ▷ `IloPathLength`: For a specified path structure in a network, specified cumulative costs along the paths are consistent with a specified table of node-to-node transit costs.
- ▷ `IloSequence`: For argument arrays `list`, `target`, `count` as in `IloDistribute`, each sequence of a specified length within `list` must contain a number of different values that is within specified lower and upper bounds.

Language designers might well disagree as to the generality of the structures treated by these constraints, but at least it seems unlikely that anyone will consider them all to be equally general.

6. Variables in subscripts

In many situations where an IP formulation would define zero-one variables, the corresponding CP formulation uses fewer variables having larger domains. This is perhaps the most characteristic modeling difference between the two approaches. To make CP formulations of this kind work, however, an extension of the usual algebraic notation is almost always necessary.

As an example, consider a location-distribution problem that involves `mCLI` clients and `nLOC` possible warehouse locations. An IP formulation in AMPL would typically define zero-one variables for each location and for each client-location pair,

```
var Open {1..nLOC} integer >= 0, <= 1;
var Serve {1..mCLI, 1..nLOC} integer >= 0, <= 1;
```

with the convention that `Open[j]` is `1` if and only if a warehouse is opened at location `j`, and `Serve[i,j]` is `1` if and only if client `i` is served from location `j`. Using these variables and correspondingly indexed costs, a linear objective function for this problem would be written as follows:

```
minimize TotalCost:
   sum {j in 1..nLOC} opnCost[j] * Open[j] +
   sum {i in 1..mCLI, j in 1..nLOC} srvCost[i,j] * Serve[i,j];
```

If we require that each client be served by one open warehouse, then the constraints can be expressed as

```
subject to OneEach {i in 1..mCLI}:
   sum {j in 1..nLOC} Serve[i,j] = 1;

subject to OpenEach {i in 1..mCLI, j in 1..nLOC}:
   Serve[i,j] <= Open[j];
```

where the first specifies that each client be served from one location, and the second insures that a warehouse is opened at any location serving a client.

In the analogous CP formulation, the `Serve` variables are indexed only over clients, but take values from the set of location numbers,

```
var Open {1..nLOC} integer >= 0, <= 1;
var Serve {1..mCLI} integer >= 1, <= nLOC;
```

so that `Serve[i]` is the location assigned to serve client `i`. The service cost in the objective function is then the total, over all clients, of the cost of serving each client from its assigned location:

```
minimize TotalCost:
   sum {j in 1..nLOC} opnCost[j] * Open[j] +
   sum {i in 1..mCLI} srvCost[i,Serve[i]];
```

The definition of the variables implicitly allows for only one location serving each client, and the requirement that the warehouse serving each client be open is very directly written as

```
subject to OpenEach {i in 1..mCLI}:
   Open[Serve[i]] = 1;
```

This is a formulation that could not be written in AMPL (or other conventional algebraic modeling languages), because it relies on an extension to allow a variable, `Serve[i]`, to serve as an index to a parameter (`srvcost`, in the objective) or to another variable (`Open`, in the constraints).

From a mathematical standpoint, such an extension involves using a variable as a “subscript” to a constant or another variable. A notation of this kind can be found, for example, in the use of terms such as $H_{\sigma(i)}$, where $\sigma(i)$ is a variable for each i, in the scheduling formulations employed by Woodruff and Spearman [24]. The usefulness of variable-in-subscript formulations is not limited to this simple case, moreover. The objective function in [24] includes a term $C(K_{\sigma(i-1)}, K_{\sigma(i)})$ that indexes the setup-cost table C by the job families of successive jobs, $K_{\sigma(i-1)}$ and $K_{\sigma(i)}$, which are themselves parameters having variables for subscripts. In the style of AMPL this term might be written as

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

where the two levels of variables in indexing can be clearly seen.

This and other examples strongly suggest that, once the concept of variables in subscripts has been accepted as a useful extension for modeling languages, it is difficult to restrict subscripts to contain variables only in certain contexts or in certain kinds of expressions. Any rules for this purpose are likely to be awkward for users to understand and for the modeling language software to apply. Instead, a general and convenient design would incorporate any subscript expression involving variables into the expression tree for the containing objective or constraint. The presence of such an indexing expression would be signaled by a new type of node that would take the place of the standard node for a variable or constant.

To handle indexing expressions that contain variables, CP solvers proceed once again by way of a structure constraint for which effective domain reduction and filtering procedures can be devised. In this case the new constraint is described by an *element* function that takes three arguments:

- ▷ `selector`, a nonnegative integer variable;
- ▷ `array`, an array of constants or of variables; and
- ▷ `result`, a variable.

The general idea is that the element constraint should be satisfied precisely when `array[selector]` equals `result`. Formally, the element function returns true if and only if the value of `selector` is a valid index into `array`, and the element of `array` indexed by the value of `selector` has the value of `result`. A reduction in the domain of `selector` or of `result` can be deduced from a reduction in the domain of the other, and if `array` contains variables then any reduction in their domains can be taken into account as well. These relationships can serve as a foundation for effective filtering procedures.

Given this framework, the job of a solver driver is to process variable-in-subscript nodes from the model translator into the element constraints that the solver requires. This is straightforward in principle, but specific cases pose problems not encountered in other modeling language extensions.

The case of a variable subscripted by a variable is the most straightforward. As a concrete example, suppose that our AMPL location-distribution problem is to be solved for data having `mCLI` = 40 clients and `nLOC` = 15 potential warehouse locations. The model translator will read the symbolic model and the data and will generate a problem instance, which the driver will then read and set up in an instance of the ASL data structure as described previously. At that point, all variables will have been mapped into one long array of variables, known to the Concert interface through the `IloNumVarArray` object `Var` seen in earlier examples; suppose that the `Serve` variables will be `Var[0]` through `Var[39]` and the `Open` variables will be `Var[40]` through `Var[54]`.

Consider how a variable-in-subscript expression like `Open[Serve[7]]` should be handled by way of an element constraint. The value of this expression can be expressed equivalently, in terms of the `Var` array alone, as `Var[39+Var[6]]`. Thus the model translator should create a variable-in-subscript node that points to the expression tree for 39 + Var[6]. In the associated element constraint, the array argument should be Var, and the selector argument should be an IloIntVar object defined to have the numbers 40 through 54 as its domain and constrained to equal 39 + Var[6]. The result argument should be a new IloNumVar object, which will be passed back from build_expr as the Concert counterpart of Open[Serve[7]].

In the Concert interface, the C++ code for an element constraint has the form result == array(selector), where the () operator for IloNumVarArrays has been overloaded to take an IloIntVar operand. Thus our build_expr case for a variable with variables in its subscript can be written as follows:

```
case VARSUBVAR_opno:  // variables in subscript of a variable
   esub = (expr_sub*)e;

   selectVar = IloIntVar (env, esub->LO.en->v, esub->UP.en->v);
   mod.add (selectVar == build_expr (esub->SUB.e));

   resultVar = IloNumVar (env, -IloInfinity, IloInfinity);

   mod.add (resultVar == Var(selectVar));
   return (resultVar);
```

What we have called the select and result arguments are set up in lines 3-4 and line 5, respectively, and our element constraint is added in line 6. The recursive call to build_expr in line 4 processes whatever subscripting expression the AMPL translator has provided. Hence this code can handle subscript expressions of arbitrary complexity, giving us the degree of generality that we seek. The only limitation is that AMPL be able to provide the appropriate selector expression for line 4, which will be the case so long as the variable being subscripted back in the model is indexed over an integer interval (and remains so after presolving). A generalization of the relevant concepts to indexing by two or more subscripts is also straightforward, but we will skip the details here.

The situation is much the same for a parameter subscripted by an expression involving variables, except that the IloNumVarArray object Var must be replaced by an IloNumArray object listing the values the parameter may take. This is a new kind of information that must be conveyed to the driver, as otherwise parameter values are reflected in constants scattered throughout a problem instance's linear coefficients and nonlinear expression trees. It does not necessarily increase the amount of information to be conveyed to the driver, however. In our location-distribution example, all of the parameter values srvCost[i,j] will need to be conveyed to the solver to permit handling of the term srvCost[i,Serve[i]]; but if the IP formulation were used instead, the same values would be conveyed in the form of linear coefficients for the objective terms srvCost[i,j] * Serve[i,j].

Throughout this discussion we have been assuming that if an expression involving variables is employed to index a variable or parameter, then that expression is implicitly constrained to take only valid index values. In fact this assumption is built into the definition that we have given for the element constraint, and is enforced by explicit bounds placed on the selector variable by our driver code. As a result we do not have to be concerned that the CP solver will encounter a "subscript out of range" error, in the way that a nonlinear solver might detect a division by zero. For our simple example the model itself ensures that any computed subscript will be valid, by putting bounds 1 and nLOC on the variables Serve[i], even though these bounds would have been imposed anyway, because Serve[i] is used to represent a subscript that can only take the values `1` through `nLOC`. In the presence of more complicated indexing expressions, these implicit validity constraints may not be so trivial.

Although the framework we have described captures many useful modeling situations, it stops short of handling cases in which the indexing sets are not integer intervals or products of integer intervals. Suppose for example that we wish to index the service costs over only a set `ABLE` of client-location pairs `(i,j)` such that location `j` is able to serve client `i`:

```
set ABLE within {1..mCLI, 1..nLOC};
param srvCost {ABLE} > 0;
```

Then the rest of the model can be written as before, and the presence of the term `srvCost[i,Serve[i]]` in the objective implicitly constrains the pair `(i,Serve[i])` to lie in the set `ABLE` for each `i`. But there is no longer a simple function that maps `i` and `Serve[i]` to a location within an array of `srvCost` values. In this situation the only reliably efficient implementation, at least when `ABLE` contains a small fraction of all possible client-location pairs, is to provide the solver with a list of all valid (client, location, service cost) combinations. The Concert interface provides a function `IloTableConstraint` for the purpose of building constraints based on tuple lists of this kind. This approach extends to larger numbers of subscripts and to similarly subscripted variables, but it must be recognized by both language translator and driver as an entirely separate case.

7. Other extensions

We conclude by summarizing further constraint programming extensions currently under study and likely to be implemented in publicly available modeling language drivers for constraint programming and other global search solvers.

Object-valued variables are a natural extension once variables are allowed in subscripts. Rather than artificially numbering the locations and clients in our location-transportation model, for example, we can declare:

```
set LOC;   # set of possible warehouse locations
set CLI;   # set of clients
```

Then the `Open` variables are indexed over `LOC`, and the `Serve` variables over `CLI`. Moreover, since `Serve[i]` is the location assigned to serve client `i`, the `Serve` variables must be declared to take values from `LOC`:

```
var Open {LOC} integer >= 0, <= 1;
var Serve {CLI} in LOC;
```

Since the "object" values of such a variable are not meaningful in numerical expressions, its main use is to appear is subscripts within the objective and constraints:

```
minimize TotalCost:
   sum {j in LOC} opnCost[j] * Open[j] +
   sum {i in CLI} srvCost[i,Serve[i]];

subject to OpenEach {i in CLI}:
   Open[Serve[i]] = 1;
```

Object values are typically represented by character strings in a modeling language, but they could be converted to numbers by the language translator so that object-valued variables might be handled by drivers in much the same way as integer-valued variables.

Set-valued variables are natural to a range of combinatorial optimization problems, as suggested in [1]. Again, the variety of set and indexing forms in algebraic modeling languages gives rise to a variety of design and implementation challenges.

Finally, *tree-search directives* are critical in many cases to efficient performance of constraint programming solvers, at least at their current stage of development. ILOG's OPL modeling language [15, 23] incorporates a collection of search statements capable of describing quite sophisticated search strategies. With this power comes the ability to specify incomplete searches, however, possibly unintentionally. It remains to be seen how solver-independent search directives should be designed to offer the best tradeoff between power and "safety" of searches.

References

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

[2] David Corne, Marco Dorigo and Fred Glover, eds., *New Ideas in Optimization,* McGraw-Hill (London, 1999).

[3] Collette Coullard and Robert Fourer, Algebraic, Logical and Network Representations in the Design of Software for Combinatorial Optimization. *Proceedings of the 29th Hawaii International Conference on System Sciences,* Volume II: Decision Support and Knowledge-Based Systems, IEEE Computer Society Press (1996) 407–417.

[4] Robert Fourer, Extending a General-Purpose Algebraic Modeling Language to Combinatorial Optimization: A Logic Programming Approach. In *Advances in Computational and Stochastic Optimization, Logic Programming, and Heuristic Search: Interfaces in Computer Science and Operations Research,* D.L. Woodruff, ed., Kluwer Academic Publishers (Dordrecht, The Netherlands, 1998) 31–74.

[5] Robert Fourer, David M. Gay and Brian W. Kernighan, A Modeling Language for Mathematical Programming. *Management Science* 36 (1990) 519–554.

[6] Robert Fourer, David M. Gay and Brian W. Kernighan, *AMPL: A Modeling Language for Mathematical Programming,* Duxbury Press (Pacific Grove, CA, 1993).

[7] David M. Gay, Automatic Differentiation of Nonlinear AMPL Models. In *Automatic Differentiation of Algorithms: Theory, Implementation, and Application,* A. Griewank and G. Corliss, eds., SIAM (Philadelphia, 1991) 61–73.

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

[9] David M. Gay, More AD of Nonlinear AMPL Models: Computing Hessian Information and Exploiting Partial Separability. In *Computational Differentiation: Techniques, Applications, and Tools,* M. Berz, C. Bischof, G. Corliss and A. Griewank, eds., SIAM (Philadelphia, 1996) 173–184.

[10] Andreas Griewank, On Automatic Differentiation. In *Mathematical Programming: Recent Developments and Applications,* M. Iri and K. Tanabe, eds., Kluwer Academic Publishers (1989) 83–108.

[11] Andreas Griewank, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation,* SIAM (Philadelphia, 2000).

[12] Tony Hürlimann, *Mathematical Modeling and Optimization: An Essay for the Design of Computer-Based Modeling Tools.* Kluwer Academic Publishers (Dordrecht, The Netherlands, 1999).

[13] ILOG, Inc., *ILOG Concert Technology 1.0 User's Manual* and *ILOG Concert Technology 1.0 Reference Manual* (Gentilly, France, 2000).

[14] ILOG, Inc., *ILOG CPLEX 7.0 User's Manual* and *ILOG CPLEX 7.0 Reference Manual* (Gentilly, France, 2000).

[15] ILOG, Inc., *ILOG OPL Studio 3.0 User's Manual* and *OPL Studio 3.0 Optimization Programming Language Reference Manual* (Gentilly, France, 2000).

[16] ILOG, Inc., *ILOG Solver 5.0 User's Manual* and *ILOG Solver 5.0 Reference Manual* (Gentilly, France, 2000).

[17] C.A.C. Kuip, Algebraic Languages for Mathematical Programming. *European Journal of Operational Research* 67 (1993) 25–51.

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

[19] Kim Marriott and Peter J. Stuckey, *Programming with Constraints: An Introduction.* MIT Press (Cambridge, MA, 1998).

[20] Colin R. Reeves, ed., *Modern Heuristic Techniques for Combinatorial Problems,* McGraw-Hill (New York, 1995).

[21] P. Refalo, Tight Cooperation and its Application in Piecewise Linear Optimization. In *Principles and Practice of Constraint Programming — CP'99,* Joxan Jaffar, ed., Springer Verlag (1999).

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

[23] Pascal Van Hentenryck, *The OPL Optimization Programming Language,* MIT Press (Cambridge, MA, 1999).

[24] David L. Woodruff and Mark L. Spearman, Sequencing and Batching for Two Classes of Jobs with Deadlines and Setup Times. *Production and Operations Management* 1 (1992) 87–102.