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

AMPL Optimization LLC
New Providence, New Jersey 07974

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

Abstract. Although algebraic modeling languages are widely used in linear and nonlinear programming applications, their use for combinatorial or discrete optimization has largely been limited to developing integer linear programming models for solution by general-purpose branch-and-bound procedures. Yet much of a modeling language's underlying structure for expressing integer programs is equally useful for describing more general combinatorial optimization constructs.

Constraint programming solvers offer an alternative approach to solving combinatorial optimization problems, in which natural combinatorial constructs are addressed directly within the solution procedure. Hence the growing popularity of constraint programming motivates a variety of extensions to algebraic modeling languages for the purpose of describing combinatorial problems and conveying them to solvers.

We examine some of these language extensions along with the significant changes in solver interface design that they require. In particular, we describe how several useful combinatorial features have been added to the AMPL modeling language and how AMPL's general-purpose solver interface has been adapted accordingly. As an illustration of a solver connection, we provide examples from an AMPL driver for ILOG Solver.

This work has been supported in part by Bell Laboratories and by grants DMI94-14487 and DMI98-00077 to Northwestern University.

1. Introduction

Algebraic modeling languages have become a standard tool in the development of linear and nonlinear programming applications [19], but they have had much less influence in the area of combinatorial or discrete optimization. Their one 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 direct correspondence to the modeler's original concept of the problem that was the motivation for modeling languages to begin with. Moreover, branch-and-bound codes still have great difficulty solving many integer programs, 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 appeared as early as the 1970s as the central idea of Lauriere's ALICE [20]:

> ... 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.

Many of the fundamental features of algebraic modeling languages, including simple and multidimensional sets, data indexed over sets, and basic mathematical operations on numbers and sets, are 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 a 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 by Fourer [4]. Hürlimann's LPL [13] augmented the standard algebraic language features by implementing a variety of logical and counting operators useful to combinatorial problems, with the option of conversion to equivalent integer programs provided as part of the model translation process.

Thus a lack of expressiveness has not been the principal barrier to extensions of algebraic modeling languages for discrete optimization. The major obstacle has rather been a lack of connections to solvers that can 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 [21, 26]. Like branch-and-bound, constraint programming solvers were based on a tree search, but using domain reduction procedures 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 operators and structures valuable in modeling discrete optimization problems.

Constraint programming solvers were initially integrated within modeling environments based on the Prolog language [21, 25]. Prolog could be viewed as a straightforward declarative means of expressing various problems of logical inference, much as the algebraic modeling languages were a straightforward declarative means for expressing integer programs. Prolog systems typically encompassed both modeling and solving, however, whereas the algebraic languages were intended to work with a variety of independently developed solvers. Moreover, as the ideas of

constraint programming were applied to an increasingly broad range of combinatorial problems, including many kinds of optimization problems, the naturalness of Prolog for modeling progressively diminished. Enhanced versions of Prolog* have thus come to be valued more for their power as specialized development languages; indeed one algebraic modeling language for global optimization, Helios, has been implemented by essentially a translation to Prolog [22]. At the same time, creators of commercial constraint programming solvers have devised procedural interfaces based on more general and object-oriented programming languages, notably C++ and Java.

Most recently, the compatibility of constraint programming solvers with algebraic modeling language expressions has been demonstrated by the OPL modeling language [27], which, through the associated OPL Studio software [17], supports both ILOG's CPLEX [15] for linear and integer programming and its Solver [18] 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. Yet 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 algebraic languages. The approaches currently employed to interface these languages to a range of linear and nonlinear solvers are inadequate to the needs of constraint programming solvers, however, due to the way that information about objective and constraint functions is currently conveyed: not 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. To place our presentation in context, we begin by reviewing (in Section 2) 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 3) 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 consider in detail the issues raised by particular extensions for "logical" constraints (Section 4), for constraints that employ "counting" operators (Section 5), for specially structured constraints such as "all different" (Section 6),

*Examples of enhanced versions of Prolog for constraint programming include CHIP (`www.cosytec.com`), ECLiPSe (`www-icparc.doc.ic.ac.uk/eclipse`), GNU Prolog (`pauillac.inria.fr/~diaz/gnu-prolog`), IF/Prolog (`www.ifcomputer.com/IFProlog`), Mozart (`www.mozart-oz.org`), and SICStus Prolog (`www.sics.se/sicstus.html`).

and for expressions using variables in the "subscripts" of parameters and variables (Section 7). Our account concludes (in Section 8) with brief discussions of the difficulties involved in other likely extensions.

We illustrate our ideas by reference to a simplified driver for the AMPL modeling language [5, 6] that we have implemented via ILOG's Concert Technology C++ interface [14] to solve constraint programs using ILOG Solver [18]. We employ 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 detailed knowledge of AMPL or C++.

Through this paper's combination of general discussion and specific illustration, we hope to encourage more widespread application of algebraic modeling languages in constraint programming. Although AMPL and ILOG Solver are proprietary, complete source code for the AMPL-to-solver interface libraries [8] and driver routines described herein is publicly available from `netlib`.* Over a dozen other AMPL drivers are provided at `netlib` 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 adopt 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.

2. Modeling language interfaces to solvers

Systems based on algebraic modeling languages maintain an optimization 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 currently represented problem.

This section presents a top-down summary of the interfaces that have enabled traditional algebraic languages to work with a variety of solvers. We first review the solver invocation process in general terms, then describe the structure of the interface code — the solver "driver" — in more detail. Finally, we address the forms in which individual model constraints and other model components are represented. Section 3 can then explain why a different approach must be taken to accommodate the combinatorial extensions that we want to consider.

Invocation of solvers. 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.

Hence the instance representation created by a general-purpose modeling system

*The `netlib` AMPL driver directories can be reached at `netlib.bell-labs.com/netlib/ampl/solvers` or `www.netlib.org/ampl/solvers`. Further information on AMPL and on ILOG Solver is available from `www.ampl.com` and `www.ilog.com/products/optimization`, respectively.

is not 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. Each solver for a modeling system has its own driver, tailored to its particular requirements. The driver handles a variety of solver-specific information, including algorithmic directives and solution reports, but for present purposes we are mainly interested in the driver's processing of problem instance representations.

In the case of the AMPL modeling system that serves as our example, the AMPL language translator converts the current problem instance to AMPL's general "`nl`" format. Each solver's AMPL driver transforms this representation as necessary, passes the transformed instance to the solver itself, and retrieves the reported solution. Finally the driver converts the solution information to a general "`sol`" format that the AMPL system is able to read.

The principal advantage of this arrangement lies in its flexibility. Writing a driver does not require access to proprietary information about either the modeling system or the solver. Thus the writing of drivers is encouraged. Some may be written by the modeling system's developers and others by individual solvers' developers. Driver source code can be made public, providing useful examples for writers of additional drivers. In the case of AMPL there are over 20 solver drivers in existence, with source code for many available through `netlib` or the solver developers as shown in the listings at `www.ampl.com/ampl/solvers.html`.

Such an arrangement also provides valuable flexibility in the way that a solver is executed. Once a modeling language system has translated a model and data to a problem instance and a solver driver has been invoked to read the instance, the 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 converted problem instances to some remote location where the solver runs.

To allow for an extension to the modeling language, however, the steps of this arrangement must be extended accordingly. In the case of the combinatorial extensions that we consider, there are straightforward ways to extend the instance and solution representations (specifically the AMPL `nl` and `sol` formats) to accommodate new kinds of operators and expressions. The principal difficulties lie in identifying useful combinatorial extensions to the modeling language and in establishing that a driver can convert the resulting instances to forms required by constraint programming solvers. Thus to establish the feasibility of the extensions in this paper, we must address not only general issues of modeling language design, but also specific details of driver construction.

Structure of drivers. Each driver contains parts written specifically for the solver being interfaced, and parts that are shared among many drivers for a particular modeling system. The latter, encompassing various interface routines as well as header files for interface data structures, constitutes an *interface library* that is typically supplied by the modeling system's developer. In the case of AMPL, an AMPL-solver interface library in C is freely available from `netlib.bell-labs.com/netlib/ampl/solvers` or `www.netlib.org/ampl/solvers`; our examples will refer to this library as the *ASL* and to its components as the ASL routines and ASL data

structures.

To be able to address the great variety of options and solvers that AMPL supports, the ASL data structures must incorporate numerous intricacies of design, many of which are not relevant to extensions for combinatorial optimization. Hence in this paper we employ a *simplified* ASL that retains only what is necessary for describing the extensions considered in this paper. Some structure fields are dropped and some C variables are renamed, for instance, with the aim of clarifying code examples for a variety of readers. For the programmers of AMPL drivers, a definitive description of the complete ASL is maintained in [8].

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

- ▷ Call interface library 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 library 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 library data structure.
- ▷ Call interface library routines to send the results back to the modeling system.

For linear, quadratic, and related solvers that receive all of their information about the problem from arrays passed as input, the interface routines are used only as indicated in these steps. In the case of more general nonlinear programming, however, the interface library may play an additional role.

A conventional nonlinear optimization method generates a series of trial solutions, or iterates, and requires the values of the nonlinear objective and constraint functions only at such iterates. Thus a nonlinear solver typically requires each user to provide a routine, in a programming language such as C or Fortran, that takes an iterate as input and that returns the corresponding objective and constraint function values (and possibly derivative values) as output. The calling conventions for this user-supplied routine differ from one solver to the next, while the body of the routine is entirely specific to the problem to be solved.

When a nonlinear solver of this kind is hooked to a modeling language driver, the developer of the driver supplies a generic function evaluation routine to take the place of the user-supplied routine. The driver's generic routine adheres to the solver's particular calling conventions, but performs the function evaluations by passing iterates back to the interface library. An interface library evaluation routine then actually computes the function and derivative values, by using the function representations previously stored in the interface library data structure.

Thus for conventional nonlinear programming the interface library routines and data structure are used not only by the driver program, but also by the solver as it processes iterates on its path toward a solution. A different generic function evaluation routine is needed for each combination of modeling system and solver, but the same routine can serve for any problem instance. This arrangement permits

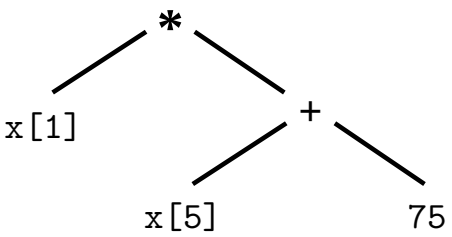

Figure 2–1: *Diagram of the expression tree for* `x[1] * (x[5] + 75)`.

a human modeler to deal with nonlinear optimization problems exclusively via their natural representations in the modeling language. We will subsequently describe a rather different arrangement that will let us make an analogous assertion for a range of combinatorial optimization problems.

Representation of model components. A model written in an algebraic language can involve numerous collections of variables indexed in various ways. By the time that the model has been instantiated with data and passed to a general-purpose solver, however, the variables have been mapped to a single numbered list. To avoid inessential complications, we assume that this mapping is done by the model translator before it passes any information to the driver, so that from the driver's standpoint the variables can be viewed as simply `x[0]`, `x[1]`, `x[2]`, and so forth.

A model can also involve numerous collections of variously indexed constraints, which are instantiated to a list of individual constraints. Any constraint arising from a representation in a conventional algebraic modeling language can moreover be converted to a standard algebraic form,

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

Hence this form is explicit in the format of the interface library's data structure.*

The constraint *lower-bound*s and *upper-bound*s are numerical values that occupy arrays in the 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 may 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. For the simplified ASL data structure, Table 2–1 shows how tables and arrays are organized for this purpose, and Table 2–2 presents the structure of the coefficient list.

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

*Objectives are also expressed as *linear-expr* + *nonlinear-expr*, but without the bounds. The new kinds of numerical-valued expressions described in this paper are thus applicable to objectives as well as constraints, although to streamline the presentation we often refer only to constraints.

`n_var`	number of variables
`n_var_int`	number of integer variables
`loVarBnd`	array of lower bounds on variables
`upVarBnd`	array of upper bounds on variables
`n_con`	number of constraints
`loConBnd`	array of lower bounds on constraint expressions
`upConBnd`	array of upper bounds on constraint expressions
`coefList`	array of pointers to constraint coefficient lists
`exprList`	array of pointers to constraint expression trees

Table 2–1: *Scalar and array components of the simplified ASL data structure,* for use by a driver in the processing of variables and constraints. The complete ASL data structure contains additional components for handling other aspects of models such as objectives, columnwise lists of coefficients, network constraints, and complementarity conditions.

```
struct conCoef {  /* linear coefficient list item */
   int varno;        /* index of variable that coefficient multiplies */
   real coef;        /* value of coefficient */
   conCoef *next;    /* pointer to next coefficient in same constraint */
};

struct expr {  /* expression-tree node */
   int op;        /* node type */
   expr *L;       /* left (or only) operand sub-tree */
   expr *R;       /* right operand sub-tree */
   expr **Lp;     /* beginning of operand sub-tree list */
   expr **Rp;     /* end of operand sub-tree list */
   int ind;       /* index (if leaf node for variable) */
   real val;      /* value (if leaf node for constant) */
};
```

Table 2–2: *Simplified C-structure components of the ASL data structure,* for use by an AMPL driver in processing linear and nonlinear expressions. The complete data structure uses a more complex arrangement together with a C `union` to represent a greater variety of node types while saving space; the above simplification shows only what is necessary for describing the extensions considered in this paper.

ables or constants. Thus for example `x[1] * (x[5] + 75)` is represented by a tree that can be depicted as shown in Figure 2–1. The interface data structure must incorporate a concise representation of such a tree that will be convenient for evaluation of the *nonlinear-expr* — and possibly its derivatives — given values of the variables.

In the case of AMPL, the variables are already mapped to a numbered list in the `nl` format (AMPL's output) that is read by the driver. Expression trees are represented in the `nl` format 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 representing multiple instances of the same 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 — unary, binary,

and so forth — as well as variables and constants. Fields within the node structure play various roles: indicating the operation or function to be applied, pointing to operands or arguments, indicating particular variables or constants (at leaf nodes), and holding intermediate information required for computation of derivatives by automatic differentiation [10, 11]. A separate array contains a pointer to the root node of each constraint expression tree. For our simplified ASL data structure, details are again provided in Tables 2–1 and 2–2.

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.* Similar `n`-ary nodes will be seen to be useful for several combinatorial extensions.

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 by use of AMPL's `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.

New operators for any purpose are readily added to the ASL data structure by generalizing or adding node types. Extensions for combinatorial optimization via constraint programming require a different approach to processing the nodes, however, to which we next turn.

3. Extending modeling language interfaces for global search solvers

We have shown that many solvers for nonlinear optimization can be supported by "function evaluation" modeling language drivers, 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

*No special distinction is made between summations that are actually (like this one) linear subexpressions and those that sum nonlinear terms.

†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, 23] are examples of global optimization methods that use local search.

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 objective and constraint functions, from which significant domain reductions or optimal solutions for subproblems can be efficiently deduced.

Algebraic modeling languages do support global search applied to linear integer programming, via drivers for branch-and-bound solvers that require only coefficient-list representations of problem instances. In fact many modeling languages support a few somewhat combinatorial extensions, such as

- ▷ variable domains that are nonconsecutive integers or that consist of zero and a positive continuous range,
- ▷ convex quadratic functions in objectives, and
- ▷ piecewise linear functions of individual variables,

specifically because branch-and-bound codes can recognize these extensions entirely through generalizations to the coefficient-list form of representation.

To support the more general methods of global search characteristic of constraint programming, however, there is no way to avoid having the driver convert essentially all of the information in the constraint expression trees into a form that the solver can recognize. This conversion has to be completed before the solver is invoked, so that the global search routines have all of the information that they need at the outset. Thus the driver's solver-specific code must perform a complete scan of the constraint expression trees. This work is accomplished by a recursive tree walk, but one that is quite different from the tree walk performed by the interface library's function evaluation routines for local search.

The remainder of this section introduces expression-tree processing in drivers for global search solvers. We consider in particular a kind of procedural solver interface that permits expressions to be communicated from driver to solver one operator or function at a time, as the tree is walked. 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. For clarity of presentation, we focus on expressions already common to algebraic modeling languages. Subsequent sections can then 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 [14]. 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 [18], 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 an existing 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-hand `IloExpr` to reflect the addition of the term represented by the right-hand `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.

In accordance with object-oriented design principles, Concert represents objects through data structures whose internal details need not concern the class library's user. A Concert-based driver creates and manipulates objects entirely via invocations of objects' 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 `IloInt 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 equivalent Concert objects. The top-level C++ statements for this purpose allocate an `IloNumVarArray` of a size equal to the number of variables, and then initialize each variable to have the proper bounds:

```
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);
```

This code is typical of the examples we will present. It combines references to the ASL data structures, such as the number `n_var` of variables and the arrays `loVarBnd` and `upVarBnd` of bounds on the variables, with references to Concert objects such as the variables `Var[j]`. Also, the outline of the code is more important than the details of the Concert function calls. In this example we observe that there are two loops, the first initializing the continuous variables and the second the integer variables.

The main constraint processing code is constructed analogously. The top-level C++ statements allocate an `IloRangeArray` of a size equal to the number of range constraints, and each pass through the subsequent loop builds one constraint:

```
IloRangeArray Con(env,n_con);
for (i = 0; i < n_con; i++) {
   IloExpr conExpr(env);
   for (cg = coefList[i]; cg; cg = cg->next)
      conExpr += (cg -> coef) * Var[cg -> varno];
   if (i < nlc)
      conExpr += build_expr (exprList[i]);
   Con[i] = (loConBnd[i] <= conExpr <= upConBnd[i]);
}
```

The loop's first statement allocates an empty `IloExpr` (or Concert expression object), the second and third statements add on the *linear-expr* and *nonlinear-expr* parts respectively, and the fourth constructs the appropriate Concert range constraint object from the completed `IloExpr` and the constraint's bounds.

Any linear part of the `IloExpr` is built directly from the ASL data structure in the outer loop's second statement, which steps through a linked list of coefficient records starting at `coefList[i]`. (Again, the details are inessential.) If there is a nonlinear part to the constraint expression, however, then a walk of the corresponding expression tree is necessary. This is the job of the recursive function `build_expr` in the loop's third statement. It takes as its argument a pointer to the root of the appropriate tree (given by the ASL expression `exprList[i]`) and returns an `IloExpr` that can be added in to the constraint expression. This function must be written as part of the driver, and we consider it next in more detail.

Walking expression trees. The driver's essential tree-walking function has just a few essential parts, which can be outlined as follows:

```
IloExpr build_expr (expr *e)
{  ...
   opnum = e->op;
   switch(opnum) {
      case PLUS_opno: ...
      case MINUS_opno: ...
      ...
   }
}
```

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 by the node. Hence each case of the `switch` on `opnum` deals with a different operation or function.

Most cases are handled recursively, by calling `build_expr` to create `IloExpr` objects for each operand or argument, and returning an appropriate C++ 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. For example, addition uses Concert's overloaded `+` operator, and logarithm uses a function `IloLog` from the Concert class library:

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

The simplified ASL node for an iterated AMPL `sum` specifies pointers to the beginning and end of an array of nodes whose associated expressions are to be summed. Thus the code for this case uses Concert's overloaded `+=` operator to accumulate terms much as in our previous 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
   sumExpr = IloExpr(env);
   for (ep = e->Lp; ep < e->Rp; *ep++)
      sumExpr += build_expr (*ep);
   return sumExpr;
```

The code for other cases (aside from extensions to be described) is much the same, 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 fixed at the constant's value:

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

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->ind];
```

Concert provides for an `IloNumVar` to be converted to the specified return type `IloExpr`.*

4. Extensions for logical constraints

Logical operators are already a standard feature of algebraic modeling languages, as they are necessary for specifying conditions on 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. For example:

```
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 relational operators like `<` as well as logical operators such as `||` (equivalently `or`). Relational and logical operator

*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`.

nodes may thus appear in the resulting expression tree. For conventional nonlinear solvers, however, these nodes are involved only in function evaluation. Given a particular iterate produced by the solver, the interface library evaluation routine tests the condition following `if`, and depending on the result it evaluates either the expression following `then` or the expression following `else` to determine the value of the entire `if-then-else` expression.

If logical operators are allowed more generally within constraint expressions, then an algebraic modeling language can express a much broader variety of logical conditions typical of combinatorial optimization models. An AMPL user formulating a scheduling problem might want to write two inequalities that are arguments to the `||` operator, for instance:

```
subject to NoOverlap {j1 in 1..nJobs, j2 in j1+1..nJobs}:

   Start[j2] >= Start[j1] + setupTime[j1,j2] ||
   Start[j1] >= Start[j2] + setupTime[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 `&&` operations. Thus `forall` can be used to assert that all equalities within 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 logical operators `&&` and `||` 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 applied in the usual ways.

Because the logical operators take *constraint-expr*s as operands, logical constraints cannot be processed as expressions between bounds like the customary range constraints. Hürlimann's LPL modeling language [13] was the first (to our knowledge) to attack this problem, by processing 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 in support of a solver for constraint programming, however, as they may hide from the solver some useful information about the modeler's choice of formulation.

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 (`<`, `<=`, `=`, `>=`, `>`, `!=`) are already defined to meet the needs of the previously discussed `if-then-else` expressions. Instances of `forall` and `exists` may be represented by new node types analogous to those for `sum`, while a double inequality may be converted to an `&&` between two single inequalities: `lo[j] <= X[j] <= hi[j]` could become `lo[j] <= X[j] && X[j] <= hi[j]`, for instance.*

By taking this approach, we introduce a kind of non-algebraic constraint that is represented in its entirety through 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 (lexprList[i]);
}
```

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 = e->op;
   switch(opnum) {
      case OR_opno: ...
      case AND_opno: ...
      ...
   }
}
```

Cases for the comparison operators are handled straightforwardly by their coun-

*More specialized nodes corresponding to double inequalities could be introduced instead, if that would be helpful for generating input to some solvers.

terparts in C++, which are overloaded by Concert to take `IloExpr` operands and return an `IloConstraint` result:

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

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 build by further calls to `build_constr`:

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

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 exclusive or and implication, that might be convenient for modeling. The algebraic `if-then-else` operator described at the beginning of this section can also be accommodated, for CP purposes, by adding an appropriate case to `build_expr`.

5. Extensions for counting

We have observed that 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 new counting operators specially.

As an illustration, in a transshipment problem the number of customers that have at least `min_ctn` shiploads 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 a language extension that provides a more specialized integer-valued counting operator for use 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. We show that the associated extensions to the driver involve no new complications.

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 points to an array of constraint nodes. 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
   sumExpr = IloExpr(env);
   for (ep = e->Lp; ep < e->Rp; *ep++)
      sumExpr += build_constr (*ep);
   return sumExpr;
```

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 introduction of `count`, our `Max_Whse_Used` constraint example ends awkwardly with the sequence

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

Thus, given that modeling languages are intended to describe models as people conceive of them, 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 the realization of their counterparts in LPL [13] simply reduces them to bounds on the sums that arise from counting operators. Similarly, 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) >= build_expr (e->R);
```

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

6. Structure constraints

The "all-different" constraint, stating that a specified list of expressions must all take different values, is a common component of CP model formulations. It is an example of a *structure* constraint that jointly restricts the values of many variables in a highly regular way.*

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 and effective algorithm for pruning the search tree. More precisely, given the currently reduced domains for a structure constraint's variables at some stage of the search, there must be efficient ways to

*These constraints 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.

- ▷ determine that no feasible solution is possible using values from the current domains, or
- ▷ effect further domain reductions by removing values that cannot possibly appear in any feasible solution taking values from the current domains.

This process of "domain reduction" or "filtering" — akin to the "presolving" and "probing" employed by some integer programming branch-and-bound codes — is typically applied at each subproblem in the course of the CP solver's search.*

We describe here two structure constraints — all-different and its generalization, number-of — that are readily provided to modeling language users through new operators and correspondingly new expression-tree nodes and driver cases. We then address the more general 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 disequalities between all pairs of such expressions. The filtering process is a series of simple 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. In the case of AMPL, lists of expressions are already familiar to modelers through their use in `display` statements for viewing results. They are readily adapted to provide 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`,

*CP solvers can be described as searching through the "nodes" of a "tree" of subproblems, but we avoid these terms here to forestall confusion with the expression nodes and trees processed by the driver.

```
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 list forms that can arise in complex applications.

Since `alldiff` is not an algebraic constraint, our AMPL/Concert driver must handle it as a case of `build_constr`. Its processing is much the same as previously shown for `sum`, looping through a list of operands and calling `build_expr` to construct a corresponding `IloExpr` object for each one. But instead of the operands being summed, they are collected in an array to be sent to `IloAllDiff`, Concert's constructor for all-different constraints. A streamlined version of the driver case might look like this:

```
case ALLDIFF_opno:  // all-different
   alldiffArray = IloExprArray(env);
   for (ep = e->Lp; ep < e->Rp; *ep++)
      alldiffArray.add (build_expr (*ep));
   return IloAllDiff (env, alldiffArray);
```

A working version is a little more complicated, because Concert requires `IloAllDiff` to be called not with an array of expressions but rather with an array of variables, to whose domains the filtering can be applied. When an `alldiff` constraint is applied a list of expressions, the driver associates an auxiliary variable with each expression and the solver subsequently infers initial domains for the auxiliary variables from the domains of the variables that they contain.

By explicitly sending an all-different constraint to the CP solver in this way, we ensure that the efficient filtering mechanisms for such constraints will be applied. Were we to instead use the `atmost` formulation, the solver would receive a separate constraint for each machine and could not be guaranteed to 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 constraint might be

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

Alternatively, the constraint might have more of a functional form, with a keyword followed by listed arguments in parentheses as in the style of OPL [17, 27]. Whatever the syntax, however, the resulting statement falls short in convenience and in naturalness. It tries to cram all of the indexing within the single constraint, whereas in fact what the modeler has in mind is a limit *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, but using a constraint syntax more natural and specific for the intended purpose. In fact the only new syntactic element we require is 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 `numberof` could be used just as well anywhere that an algebraic expression involving variables would be allowed.

A straightforward adaptation of our driver case for `count` suffices to handle `numberof` in general. 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 `sumExpr` the number of assertions of a certain form that are true:

```
case NUMBEROF_opno:  // number of
   sumExpr = IloExpr(env);
   ep = e->Lp;
   targetExpr = build_expr (*ep);
   for (*ep++; ep < e->Rp; *ep++)
      sumExpr += (build_expr (*ep) == targetExpr);
   return sumExpr;
```

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 process a *target-expr* that is a constant, however, since this is the situation to be handled by a structure constraint. Concert communicates such a constraint to its solver through a call to `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`.

The constraint is interpreted to say that, for each index `k`, the variable `count[k]` equals the number of times that `target[k]` appears in `list`. Thus a single call to `IloDistribute` can represent all of an AMPL model's `numberof` operators that have the same *expression-list*, and can permit more effective filtering than would be possible if each `numberof` occurrence were to be treated as a separate constraint.

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 auxiliary `IloNumVar`s set up to equal whatever expressions appear in a `numberof` operator's *expression-list*. For each collection of `numberof` operators that have the same *expression-list*, moreover, the *target-expr*s are simply collected to form the `target` array; the content of this array is not limited to consecutive integers.

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->Lp;
   if ((*ep)->op == CONST_opno)  // target is a constant
      return build_numberof (e);
   else {
      // same as shown previously
   }
```

In brief, `build_numberof` maintains a data structure, specific to the driver, that keeps track of the *expression-list*s and the *target-expr* constants encountered. When a new *expression-list* is seen, a corresponding `list` array is created. When a new *target-expr* is seen for some *expression-list*, the target value — which must be a

constant in this case — is added to the `target` array, and a corresponding new variable is defined in the `count` array. In any case, `build_numberof` returns the `count` variable corresponding to the *target-expr* value. This variable represents the value of the `numberof` operation and hence, as seen above, can be returned by the `build_expr` case without any further processing.

The most costly part of `build_numberof` is to check whether a list of expressions in the ASL data structure is identical to any previously seen list. This relies on a recursive comparison function for determining whether the subtrees beneath two nodes are the same. There is also a cost in recursively calling `build_expr` to process the expression trees for elements of the list, but this needs to be done only the first time that the list is seen.

When the tree-walks for all of the AMPL model's constraints are complete, the driver's data structure contains a `list`, `target`, and `count` array for each different structure constraint implied by `numberof` operators in the problem instance. These triples of arrays are then easily passed as arguments to `IloDistribute` to complete the representation required by the CP 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 number-of falls into this category, as expressions using the `numberof` operator in an AMPL model are translated to constraints specified by the `IloDistribute` function in Concert. Piecewise-linear functions of individual variables provide another example. An objective function of piecewise-linear cost terms might 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 produces 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. CP solvers may instead replace each piecewise-linear term 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 [24].

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 solver 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 sufficient benefit must be gained in

compensation. A syntax extension may be justified as making the modeling language easier and more natural to use across a broad variety of applications, for instance. An extension may also offer the prospect of support, through correspondingly extended drivers, for a range of solvers. For general and widely useful structure constraints such as all-different and number-of, the benefits seem clear. But there are other more specialized structure constraints for which the decision is not so easily made. Consider that, in just the current Concert interface, there are functions to construct structure constraints of four more 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`: Specified arrays `list`, `target`, and `count` are related as in `IloDistribute`, and also each subsequence within `list` of a specified length must contain a number of *different* values that is within specified 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.

7. 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. After giving some examples, we describe in detail how a modeling language translator and driver handle the simple case of a variable that is indexed (subscripted) by an expression involving variables, and conclude by indicating how more general cases can be handled.

Examples. Consider a location-distribution problem that involves `mCLI` clients and `nLOC` possible warehouse locations. An IP formulation in AMPL typically defines binary (zero-one) variables for each location and for each client-location pair,

```
var Open {1..nLOC} binary;
var Serve {1..mCLI, 1..nLOC} binary;
```

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 can 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 each client must 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 [18], the `Serve` variables are indexed only over clients, but take values from the set of location numbers,

```
var Open {1..nLOC} binary;
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 [28]. The usefulness of variable-in-subscript formulations is not limited to this simple case, moreover. The objective function in [28] 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.

Implementation fundamentals. These 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 allow any subscript within an objective or constraint to be an expression involving variables. An expression tree for a subscript of this kind would be incorporated into the tree for the containing objective or constraint, by means of a new node type that would take the place of the standard node for a variable or constant.

To handle indexing expressions that contain variables, CP solvers proceed 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 an *element* constraint defined in terms of three items:

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

The general idea is that the element constraint is satisfied by the variables precisely when `array[selectVar]` equals `resultVar`. Formally, the constraint holds if and only if the value of `selectVar` is a valid index into `array`, and the element in the `selectVar` position within `array` has the same value as `resultVar`. A reduction in the domain of `selectVar` or of `resultVar` 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 the variable-in-subscript expression-tree nodes created by 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 `x` of variables, as explained in Section 2; suppose that the `Serve` variables will be `x[0]` through `x[39]` and the `Open` variables will be `x[40]` through `x[54]`.

Consider first how a variable-in-subscript expression such as `Open[Serve[7]]` should be handled in the ASL data structure. This expression can be written equivalently, in terms of the `x` array alone, as `x[39+x[6]]`. Thus in the ASL tree this expression should be represented by a variable-in-subscript node pointing to a subtree for the expression `39 + x[6]`.

Now we can describe the conversion to a form that uses an element constraint. There are three steps:

- ▷ Create a new variable `resultVar` that will represent the value of the variable-in-subscript expression.
- ▷ Create a new variable `selectVar` having the numbers 40 through 54 as its domain, along with a new constraint `selectVar = 39 + x[6]`.
- ▷ Add the element constraint defined by `selectVar`, `x`, `resultVar`.

The variable `resultVar` can then be used in place of the variable-in-subscript expression in the remaining processing of the tree. This conversion is repeated for each variable-in-subscript node, with of course different `resultVar` and `selectVar` variables created in each case.

In our example using the Concert interface, the role of `x` above is played by the `IloNumVarArray` that we call `Var`. The C++ code for an element constraint has the form `resultVar == Var(selectVar)`, where the `()` operator for the `IloNumVarArray` class has been overloaded to take an `IloNumVar` operand. Thus our `build_expr` case for a variable with variables in its one subscript can be written as follows:

```
case VARSUBVAR_opno:  // variables in subscript of a variable
   selectVar = IloIntVar (env, loSubBnd[e->ind], upSubBnd[e->ind]);
   mod.add (selectVar == build_expr (e->L));
   resultVar = IloNumVar (env, -IloInfinity, IloInfinity);
   mod.add (resultVar == Var(selectVar));
   return (resultVar);
```

The `selectVar` argument is set up in the first and second lines of the case. The AMPL translator is relied upon to communicate lower and upper bounds on this variable, kept in auxiliary arrays `loSubBnd` and `upSubBnd` of the ASL data structure, because these bounds are likely to be the same at many different nodes.* The recursive call to `build_expr` processes whatever subscripting expression the AMPL translator has provided, assuring that the driver can handle subscript expressions of arbitrary complexity. The third line sets up the `resultVar`, whose bounds are left to the CP solver to deduce. Then the remaining lines add the element constraint to the problem and return `resultVar` as the Concert `IloExpr` whose value is equal to that of the whole variable-in-subscript expression.

Since both `selectVar` and `resultVar` are defined by equality constraints, it would seem that this case could be more concisely written in the following way:

```
case VARSUBVAR_opno:  // variables in subscript alternative
   return (Var(build_expr(e->L)));
```

This would not necessarily result in the additional efficiency that the elimination of two variables might suggest, however. Like other structure constraints, the element constraint is useful to the solution search because a reduction in the domain of one of its arguments may be used to deduce reductions in the domains of other arguments. If we substitute away some of the argument variables, then we are basically counting on the CP solver to supply them implicitly in its search procedure. In fact the above code, in which the overloaded `()` operator is passed an `IloExpr` operand, is rejected by the Concert class library; an `IloNumVar` argument is required. Thus at most `resultVar` can be eliminated, by writing `return (Var(selectVar))`.

Implementation generalizations. The case of a model variable indexed by multiple subscripts, one or more containing variables, is not substantially more difficult. Basically it requires the model translator to generalize the expression to which the `selectVar` is equated. The driver processing is unchanged.

For variables in a parameter's subscripts the situation is also much the same,

*In our example with 40 clients and 15 locations, for every client `i` the model expression `Open[Serve[i]]` will give rise to an integer-valued `selectVar` that has bounds 40 and 54.

except that the `IloNumVarArray` object `Var` must be replaced by an `IloNumArray` object listing the values the parameter may take. To permit the construction of such an object, the model translator must provide a listing of possible parameter values to the driver. This is a new form of information, but it does not necessarily increase the total information conveyed to the driver. In our location-distribution example, for 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 `selectVar` 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 any number of subscripts and to variables' subscripts, but it must be recognized by both language translator and driver as an entirely separate case.

8. Other extensions

We conclude by summarizing further constraint and expression types that would facilitate natural formulations of combinatorial optimization problems and that could be implemented as extensions to algebraic modeling languages. All of these features are supported by a number of existing 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} binary;
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.

Membership constraints state that the value of a certain expression must be a member of a specified set. In our production scheduling model from Section 6, for instance, practical considerations might limit the assignment of consecutive jobs to certain pairs of machines. This restriction is naturally stated in terms of a set of allowable pairs:

```
set ALLOWABLE within {1..nMachines, 1..nMachines};

subject to AllowablePairs {i in 1..nJobs-1}:
   (MachineForJob[i],MachineForJob[i+1]) in ALLOWABLE;
```

This is another example of an operator (here called `in`) that is already present in many algebraic modeling languages but that could be generalized for combinatorial optimization by allowing its application to variables. Here the variables would be allowed anywhere within the left operand.

Set-valued variables are natural to a range of combinatorial optimization problems, as suggested in [1]. Within an algebraic model description, these variables can appear as indexing sets for iterated operators such as `sum` and in set-valued expressions for the right operand of `in`. Again, the result is a variety of cases that give rise to a variety of design and implementation challenges.

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 [17, 27] 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. Thus there is a complex tradeoff between the power and the "safety" of searches and the solver-

independence of the search directives; this remains a challenging area for language and driver design.

Finally, *hybrid methods* combining features of integer and constraint programming algorithms will pose further challenges to the construction of algebraic modeling language drivers. The Concert interface used as an example for this paper offers several options for integration [16], while a detailed account of the possibilities for merging techniques of optimization and constraint programming is provided by Hooker [12].

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 (Dordrecht, The Netherlands, 1989) 83–108.

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

[12] J.N. Hooker, Logic, Optimization and Constraint Programming. *INFORMS Journal on Computing* 14 (2002) — *elsewhere in this issue.*

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

[14] ILOG, Inc., *ILOG Concert Technology 1.1 User's Manual* and *ILOG Concert Technology 1.1 Reference Manual* (Gentilly, France, 2001).

[15] ILOG, Inc., *ILOG CPLEX 7.5 User's Manual* and *ILOG CPLEX 7.5 Reference Manual* (Gentilly, France, 2001).

[16] ILOG, Inc., *ILOG Hybrid Optimizers 1.1 User's Guide and Reference* (Gentilly, France, 2001).

[17] ILOG, Inc., *ILOG OPL Studio 3.5 User's Manual* and *OPL Studio 3.5 Language Manual* (Gentilly, France, 2001).

[18] ILOG, Inc., *ILOG Solver 5.1 User's Manual* and *ILOG Solver 5.1 Reference Manual* (Gentilly, France, 2001).

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

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

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

[22] Laurent Michel and Pascal Van Hentenryck, Helios: A Modeling Language for Global Optimization and Its Implementation in Newton. *Theoretical Computer Science* 173 (1997) 3–48.

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

[24] 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 (Berlin, 1999).

[25] Leon Sterling and Ehud Shapiro, *The Art of Prolog*, second edition, MIT Press (Cambridge, MA, 1994).

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

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

[28] 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.