A FRAMEWORK FOR INTEGRATING SOLUTION METHODS

J. N. Hooker
Graduate School of Industrial Administration
Carnegie Mellon University, Pittsburgh, PA 15213 USA
jh38@andrew.cmu.edu

Abstract

We describe a modeling framework that integrates mathematical programming (MP), constraint programming (CP) and heuristic methods. It is extendible to other solution methods as well. The problem structure is mirrored in the model structure, and the solver exploits this structure in a principled way to combine methods effectively. The approach generalizes and extends past research on the integration of MP and CP. Six modeling examples are given. In particular, it is shown that a recent integration scheme for CP and MP based on Benders decomposition is a special case of the framework described here.

Solution methods often have complimentary strengths that allow them to be profitably combined. Yet different problems require solvers to be combined in different ways, and it is often necessary to write special-purpose code for each occasion. It would be useful to have a multi-purpose solver that could recognize which portion of a problem should be attacked by each method, and that could somehow coordinate these methods in a principled way.

This aim of the present paper is to outline such a framework that unifies mathematical programming (MP), constraint programming (CP) and heuristic methods. Although it combines three specific methods, it uses an architecture that is generalizable to additional methods.

The proposal described here is based on about a decade of experience with hybrid methods that combine MP and CP, surveyed in [8, 9]. It specifically extends ideas in [2, 4, 6, 7, 8, 11, 20, 21, 22, 23, 24, 26]. The main contribution of this paper is to generalize the framework developed in these papers and add heuristic methods to the mix. This earlier framework obtained considerable computational success on selected problems, such as processing network design [4], combinatorial scheduling [11], fixed charge network flow [17], piecewise linear costs [21], and truss design [2]. There are as yet no computational tests, however, of the expanded framework that includes heuristic methods. The paper describes work-in-progress that is expected to result eventually in a software package that will be released for public use.

Section 1 shows how different solution methods have strategies in common, and how these can serve as a basis for a unified solver. Section 2 explains how this approach can exploit problem structure using the idea of a global constraint from CP. Sections 4, 5, 6, and 7 use the proposed framework to formulate traveling salesman, processing network design, lot sizing, and allocation problems. Section 8 shows how the framework can be extended to another problem solving context, in this case Benders decomposition, which combines a master problem solver with a subproblem solver. This allows Section 9 to demonstrate that a very promising recent scheme for unifying CP and MP, based on generalized Benders decomposition, is a special case of the framework described here.

1. Exploiting Common Solution Strategies

Different solution methods sometimes use the same general strategies. This can serve as a basis of unification, since the combined solver can consist of a module for each general strategy. Each module would adapt itself to the structure of a particular problem, or portion of a problem.

Three general solution strategies are found in MP, CP and/or heuristic methods: a search over problem restrictions, inference of new constraints, and solution of relaxations. For definiteness we will regard integer linear programming (IP) as representing MP methods, and local search (LS) as representing heuristic methods.

1.1 Searching over problem restrictions

All three solution methods can be seen as conducting a search over restrictions of the problem.

- IP branches on values of a variable that has a noninteger value in the continuous relaxation. Each branch defines a restriction of the problem by restricting one of the variables. The search tree as a whole represents an exhaustive search over problem restrictions.

- CP branches in a similar fashion by splitting the *domain* of a variable, which is the set of values the variable can take. Typically a domain is a finite set or an interval of real numbers. Since CP normally seeks a feasible solution, the search stops when a solution is found, or when an exhaustive enumeration proves infeasibility.

CP is easily adapted to optimization by placing a bound on the objective function that is tightened as one discovers feasible solutions.

- LS typically moves from one complete restriction of the problem to a “neighboring” restriction. That is, it deals with restrictions that specify the value of each variable, although one could equally well design LS to enumerate partial restrictions. The search heuristic may implicitly enforce constraints that are not explicit stated, opening the possibility of a model that is partly declarative and partly procedural.

It is evident that all three search regimes can be specified recursively. The recursive step in IP and CP generate restrictions by splitting a variable domain in the current restriction, and the recursive step in heuristics generates a restriction that neighbors the current one. Thus a unified solver would contain a single recursion that implements the desired traversal of the solution space.

1.2 Inferring new constraints

IP and CP infer new constraints from the original ones and add them to the constraint set.

- IP uses *cutting plane* methods to infer valid inequalities that are implied by the original constraints. They are chosen to result in a tighter continuous relaxation (discussed below). Modelers may also write redundant constraints when formulating the model, for the same reason.

- CP uses *domain reduction* methods to infer *in-domain* constraints, each of which restricts the domain of a variable. This is accomplished by processing each constraint with a domain reduction or *filtering* algorithm that is tailored to that type of constraint. The aim is to eliminate, from each variable domain, values that cannot be part of any feasible solution of that constraint. If all such values are eliminated, *hyperarc consistency* (also known as *generalized arc consistency*) is achieved with respect to the constraint in question.

  The reduced domains are passed on to the next constraint in a process called *constraint propagation.* There are various schemes for cycling through the constraints and updating the domains, perhaps until a fixed point is reached. (This process generally does not achieve hyperarc consistency with respect to the entire constraint set, even if it achieved with respect to each individual constraint.)

As in IP, modelers may also add redundant constraints by hand, but in this case the constraints are designed to result in more effective constraint propagation. Sometimes two models of the same problem are used in the formulation.

1.3 Creating and solving relaxations

Both IP and CP create a relaxation of the restricted problem at each node of the search tree. Its solution contains information that can help to guide the search and possibly create a stronger relaxation. The solution may also happen to be feasible for the original problem. The constraints in the relaxation are structured in such a way that the relaxation is easy to solve.

- IP generally builds a relaxation, at each node of the search tree, that consists of the linear inequalities in the current problem restriction. Its optimal value provides a bound on the value of the original problem that can be used to prune the search tree (*branch and bound*). If the relaxation is infeasible, the search backtracks. If a solution exists and all variables are integral, the incumbent solution is updated (if necessary) and the search backtracks. Otherwise the search branches on a variable with a nonintegral value and possibly generates *separating cuts* with respect to the solution of the relaxation.

- CP builds a relaxation that is called the *constraint store*, by generating in-domain (and possibly other) constraints. The relaxation can be trivially solved by selecting an arbitrary value from each domain. This solution may be infeasible in the original problem, even if hyperarc consistency is achieved with respect to the entire constraint set. This is because variable assignments that are individually feasible need not be feasible when assembled into a complete solution.

  CP methods normally do not actually solve the constraint store, but they extract useful information from the domains. If a domain is empty, the constraint store is infeasible and the search backtracks. If all domains are singletons, the search terminates with a feasible solution (in this case the constraint store is actually solved). Otherwise further branching is necessary, and the search typically branches by splitting the smallest domain in what is known as a *first-fail* branching strategy.

The relaxation can be solved by any number of methods, such as linear programming or even heuristic methods. This provides a secondary mechanism for combining solution methods.

2. Exploiting Problem Structure

We now have the basic outline for an integrated solver. It contains

- a recursion that specifies the *search* by moving from one problem restriction to another,
- an *inference* engine that derives valid constraints for each problem restriction, and
- a mechanism for generating and solving a *relaxation* of each restriction.

Note that inference and relaxation can be used in the context of a heuristic search as well as an exhaustive search, even if this is not often done.

The basic question remains, however, as to how one can take advantage of an integrated solver to exploit the peculiar characteristics of a given problem. CP provides a valuable clue as to how this might be done. Very often subsets of constraints in a problem exhibit a structure that can be exploited by a specialized filtering algorithm. The solver must somehow recognize these subsets, however.

This might be done by equipping the solver with automatic pattern recognition, as is commonly done for network structure in MP solvers. Yet the modeler is generally already aware of a problem's special structure, because it is on this basis that the model is formulated. One generally writes a model by assembling some flow balance constraints, some hamiltonian path constraints, some capital budgeting constraints, and so forth. The modeler can inform the solver about these substructures, rather than putting them into an undifferentiated constraint set and expecting the solver to rediscover them.

2.1 The principle of global constraints

CP allows the modeler to indicate structure by writing a single *global* constraint in place of a structured subset of constraints. It is called a global constraint because it captures the global structure of the constraints it represents. The solver is equipped with filtering algorithms that are specialized to each type of global constraint.

An example is the global constraint `all-different`, which is very important in the formulation of scheduling problems. Let $y_1, \ldots, y_n$ be discrete variables, where each y_j has a finite domain D_j. The constraint all-different$(y_1, \ldots, y_n)$ imposes all the pairwise inequations $y_i \neq y_j$ for $i < j$. The variable y_j might be the machine assigned to job j, and the all-different constraint would say that each job is assigned to a different machine. To illustrate domain reduction suppose that $n = 4$. If $D_1 = D_2 = \{1, 2\}$ and $D_3 = D_4 = \{1, 2, 3, 4\}$, none of the pairwise inequations imply any domain reduction when considered individually. Yet the all-different constraint reduces D_3 and D_4 to $\{3, 4\}$, due to the fact that $\{y_1, y_2\} = \{1, 2\}$. A complete filtering algorithm for this particular constraint is Regin's [], which is based on maximum cardinality bipartite matching and a theorem of Berge.

Although CP uses global constraints to trigger specialized filtering algorithms, they can also be used to generate cutting planes and to create relaxations. For instance, a subset of constraints might consist of inequalities for which specialized cutting planes have been developed. The constraints could be represented by a global constraint that generates the appropriate cutting planes. Currently much cutting plane technology is underutilized because there is no convenient way to incorporate it in a general-purpose solver. Global constraints could overcome this obstacle.

Global constraints can also trigger the creation of a relaxation, perhaps simultaneously with filtering. A particularly useful constraint is element$(y, (x_1, \ldots, x_n), z)$, where y is an integer-valued variable, the x_i's are variables of any sort, and z is a variable of the same sort as the x_i's. The constraint imposes the equation $x_y = z$ whenever the value of y is determined. One can therefore implement an expression of the form x_y by replacing x_y with z and adding the element constraint to the model. There is a special-purpose filtering algorithm that can be applied to element, as well as the convex hull relaxation [8, 13]:

$$\sum_{i \in D_y} x_i - (|D_y| - 1)m \leq z \leq \sum_{i \in D_y} x_i$$

where D_y is the current domain of y and m is an upper bound on all of the x_i's. Both the filter and the relaxation would be invoked by the appearance of element.

2.2 The underlying data structure

So far it is proposed that modelers exploit problem structure by using global constraints, so that the solver knows where the structure is. Each global constraint invokes special purpose inference algorithms (for domain reduction, cutting plane generation, etc.) and/or a special-purpose relaxation.

At this point we know how to deal with portions of the problem, but it is unclear how to assemble the results to solve the problem as a whole. CP does this by propagation through the constraint store. Each filtering algorithm refines the constraint store by further reducing the domains. When all the constraints have been processed, branching is based on information in the current constraint store.

The constraint store is essentially an easily-solved relaxation. This suggests that, in a more general setting, the solver's various routines could be linked through one or more relaxations, and search could proceed on the basis of information in the relaxations. Each relaxation would be stored in an appropriate data structure. There would also be a data structure to hold the current problem restriction.

Two obvious relaxations are a constraint store and a set of linear inequalities. The former is updated by domain reduction, and the later by generating continuous relaxations for constraints. The search proceeds on the basis of information from the constraint store (whether all domains are singletons, and whether there is an empty domain), as well as information from the linear relaxations (the value of the relaxation, and which variables have noninteger values in the solution of the relaxation).

3. A General Modeling Framework

We propose to implement the above scheme in a very simple overall modeling framework. It consists of *modeling windows* that contain variable declarations, the objective function, constraints, relaxations, and search instructions. When creating a model, one might literally open a new window on the computer screen for each new modeling window. Each window is a modeling box within which one uses an appropriate modeling sublanguage.

3.1 Types of windows

The various types of windows may be described as follows.

Variable declaration window. This window lists the variables and the initial domains of each. There may be several types of domains (finite sets, intervals of real numbers , etc.) with their own appropriate data structures.

Constraint windows. The most basic component of a model is a constraint window, of which there are many types, depending on what type of constraint is specified in the window. Some windows specify linear inequalities, perhaps using a sublanguage resembling AMPL or GAMS. Others may contain specially structured sets of inequalities, such as network flow constraints, set covering constraints, or traveling salesman constraints (represented implicitly). Others may contain logical propositions or statements involving sets. Still others may implement a particular global constraint, such as *cumulative* or *all-different*.

A constraint may be associated with such inference procedures as a specialized filtering algorithm or cutting plane generation, and/or one or more relaxations.

Objective function window. This window specifies the objective function for the problem, if any.

Relaxation windows. Each relaxation that links the constraints is represented by a relaxation window that specifies the type of relaxation and the solver to be used. The window sets up a data structure for the relaxation and its solution, and it initializes the set of routines that generate relaxations of this sort for each constraint window. The window also specifies an objective function, because it could differ from the objective function of the original problem.

Relaxations have yet to be developed for a number of popular constraints, but this poses an interesting research program that polyhedral theory is well equipped to address. Linear relaxations have recently been put forward for `all-different` [8, 28], `element` [8, 13], cardinality constraints that count how many variables have a certain value [30], and the widely-used `cumulative` constraint for resource-constrained scheduling [15].

Search window. This window directs the search recursively, whether it be an exhaustive branching search, a local search, or whatever. It does so by creating one or more new restrictions of the problem, using information in the current relaxations. It initiates processing of each of the new restrictions. The system could provide several templates for the search window, one for branching search, one for tabu search, and so forth, each allowing the user to specify parameters for the search. Each model has exactly one search window.

User-defined windows. These windows give the user direct access to search and constraint solving routines and supplies coded subroutines to be executed at specified points defined by the search window.

3.2 The search window

The search window can invoke any of several generic procedures:

- **search**(P, R, S). The search window directs the search recursively by invoking itself, passing along a restriction P of the problem, a vector $R = (R_1, \ldots, R_k)$ of the current relaxations, and a vector $S = (S_1, \ldots, S_k)$ of their solutions. The latter two are made available to enable efficient update of the current relaxation and solution.

- **infer**(P, R, S). This activates a constraint propagation algorithm that cycles through the constraint windows, inferring constraints for each. The constraints are added to the current problem restriction P, which is passed back to **search**. R and S may be provided for the generation of separating cuts. It may be desirable for **infer** to accept parameters specifying how it cycles through the windows, perhaps using a well-known CP procedure such as AC-1 or AC-3.

- **relax**(P, R_i, S_i, v_i). This replaces the previously used relaxation R_i with a relaxation of P of type i. It cycles through the constraint windows and for each generates constraints to add to R_i. It then solves R_i using the objective function in the corresponding relaxation window. The solutions is returned as S_i, and the optimal value as v_i.

Control is based on information in R and S.

4. Example: Traveling Salesman Problem

The traveling salesman problem has a particularly simple representation using the `cycle` constraint, which is related to `all-different`. Let y_j represent the city that comes after city j in a tour. Then `cycle`$(y_1, \ldots, y_n)$ states that $y_1, \ldots, y_n$ should describe a hamiltonian cycle. If c_{ij} is the distance from city i to city j, the traveling salesman problem can be written

$$\begin{array}{ll} \text{minimize} & \sum_{j=1}^{n} c_{jy_j} \\ \text{subject to} & \texttt{cycle}(y_1, \ldots, y_n) \end{array}$$

where the initial domain of each y_j is $\{1, \ldots, n\}$.

The model appears in Fig. 1. Modeling windows 1 and 2 define the variables and objective function. The variable z_j in the objection is defined by window 4 to be c_{jy_j}. Window 3 sets up the standard constraint

1. Variables and Initial Domains for Problem P
$y_j \in D_j = \{2, \ldots, n\}$ for $j = 2, \ldots, n$ (jth city in tour)
$D_1 = \{1\}$
$z_j \in \mathcal{R}$ for $j = 1, \ldots, n$. (cost of jth link in tour)

2. Objective Function
minimize $\sum_{j=1}^{n} z_j$

3. Relaxation R_1
Type: Constraint store, consisting of domains of $y_1, \ldots, y_n$.
Objective function: none.
Solver: select a value from each domain.

4. Relaxation R_2
Type: Linear programming
Objective function: minimize $\sum_{jk} c_{jk} x_{jk}$
Solver: linear programming.
Set incumbent value $\bar{z} = \infty$, where $\bar{z}$ is a global variable.

5. Constraint: element
`element`$(y_j, (c_{j1}, \ldots, c_{jn}), z_j)$ for $j = 1, \ldots, n$.
Inference: maintain hyperarc consistency.
Relaxation: add reduced domains to constraint store.
Relaxation: disjunctive relaxation.

6. Constraint: cycle
`cycle`$(y_1, \ldots, y_n)$
Inference: domain reduction algorithm to be developed.
Relaxation: add reduced domains to constraint store.
Relaxation: standard IP relaxation, with assignment inequalities, separating subtour elimination inequalities and various separating cuts with respect to S_2. Also fix $x_{jk} = 0$ if $k \notin D_j$ and $x_{jk} = 1$ if $D_j = \{k\}$

7. Search: branch and bound
BandBsearch$(P, R, S, v,$ **TSPbranch**$)$

8. User defined procedure
```
Procedure TSPbranch($P, R, S, i$). (Take the $i$th branch.)
    Let $x_{jk}$ be a variable with a nonintegral value in $S_2$.
    If $i = 1$ then create $P'$ from $P$ by letting $D_j = \{k\}$ and return $P'$.
    If $i = 2$ then create $P'$ from $P$ by letting $D_j = D_j \setminus \{k\}$ and return $P'$.
```

Figure 1. Model for the traveling salesman problem.

store R_1 containing the variable domains. Since the domains are also part of P, R_1 in this case is redundant. Yet formally speaking branching is based on the domains in R_1 rather than the identical domains in P.

```
Procedure **BandBsearch**$(P, R, S, \textbf{Branch})$.
    Perform **Infer**$(P)$. (Reduce domains.)
    Perform **Relax**$(P, R_1)$. (Check the domains.)
    If $R_1$ is infeasible then return.
    Perform **Relax**$(P, R_2, S_2, v_2)$. (Solve LP relaxation $R_2$.)
    If $R_2$ is feasible and $v_2 < \bar{z}$ then
        If $S_2$ is feasible then update the incumbent solution and let $\bar{z} = v_2$.
        Else
            Perform **BandBsearch**(**Branch**$(P, R, S_2, 1), R, S)$.
            Perform **BandBsearch**(**Branch**$(P, R, S_2, 2), R, S)$.
```

Figure 2. Search routine for standard branch and bound, where $R = (R_1, R_2)$ are the constraint store and a linear relaxation, and $S = (S_1, S_2)$ are their solutions. The function **Branch**(P, R, S, i) *returns the problem that results from taking the ith branch. The specific branching function is passed into* **BandBsearch** *as a parameter.*

Window 5 indicates that relaxation R_2 is a linear programming problem. Note that the objective function is the classical traveling salesman objective, which uses variables x_{jk}. Here $x_{jk} = 1$ if j immediately precedes k in the tour. In general it is possible to introduce new variables in order to formulate a relaxation. In this case variables x_{jk} that would traditionally be 0-1 variables in the traveling salesman model are used only as continuous variables in the relaxation and play no modeling role.

Window 6 imposes the `cycle` constraint. Curiously, there seems to be no filtering algorithm available for `cycle` in current CP technology, but it would be an interesting research project to develop one. The relaxation consists of assignment constraints, plus separating subtour elimination constraints and other separating cuts with respect to the previous solution S_2.

Window 7, the search window, calls a canned depth-first branching search procedure, which appears in Fig. 2. It passes to the search procedure a function **TSPbranch** that defines how the search should branch, based on the solution of the relaxation. The search differs slightly from standard branch and bound in that there is an initial check at each node for whether a domain is empty, in which case the search backtracks. The search could also check whether all domains are singletons, and if so whether resulting solution is feasible in the original problem, in which case the incumbent solution is updated. This step is omitted from Fig. 2 for simplicity.

The branching function **TSPbranch**(P, R, S, i) appears in window 8, a user-defined window. The function returns the problem to which one branches in the ith branch. It branches on a variable x_{jk} that is nonintegral in the solution of the linear relaxation. This is accomplished

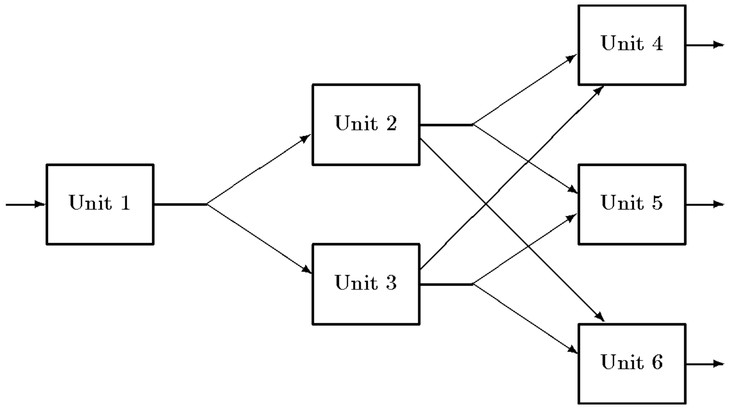

Figure 3. Superstructure for a processing network design problem.

by setting $D_j = \{k\}$ for the $x_{jk} = 1$ branch and setting $D_J = D_j \setminus \{k\}$ for the $x_{jk} = 0$ branch.

5. Example: Processing Network Design

An early application of integrated modeling was to processing network design problems in the chemical industry [4, 23].

Figure 3 displays a small instance of a processing network design problem. The object is to determine which units to include in the network so as to maximize net income (revenue minus cost). Each processing unit i incurs a fixed cost d_i and delivers revenue $d_i u_i$, where the variable u_i represents the flow volume entering the unit. The revenue is normally positive for the terminal units (units 4–6) because their output is sold, and it is normally negative for the remaining units.

The model appears in Fig. 4. The first modeling window defines the variables and domains; note that y_i is a logical proposition that is true when unit i is installed. The quantities c_i and c_{ij} are capacities. The objective function in window 2 subtracts total fixed costs from net variable income. Window 3 defines the constraint store, and window 4 sets up a linear relaxation with the same objective function as the original problem.

The linear constraints $u = Ax$ in window 5 define the flows u_i through the units in terms of the flows x_{ij} on the arcs. The constraints $bu = Bx$ compute the flows out of each intermediate unit. The constraints in this window can be added directly to the linear relaxation. They can also be processed with a filtering algorithm that reduces the interval domains

1. Variables and Initial Domains
$u_i \in [0, c_i]$ (flow through unit i)
$x_{ij} \in [0, c_{ij}]$ (flow on the arc from unit i to unit j)
$z_i \in [0, \infty]$ (fixed cost of unit i, if any)
$y_i \in D_i = \{T, F\}$ (true when unit i is installed)

2. Objective Function
maximize $\sum_i r_i u_i - \sum_i z_i$

3. Relaxation: R_1
Type: constraint store, consisting of variable domains.
Objective function: none.
Solver: Select a value from each domain.

4. Relaxation R_2**.**
Type: linear programming.
Objective function: maximize $\sum_i r_i u_i - \sum_i z_i$
Solver: linear programming.
Set incumbent value $\bar{z} = -\infty$, where $\bar{z}$ is a global variable.

5. Constraint: Linear Inequalities
$u = Ax$
$bu = Bx$
Inference: bounds consistency maintenance.
Relaxation: add reduced domains to constraint store.
Relaxation: add all these inequalities to LP relaxation.

6. Constraint: Disjunction of Linear Inequalities
$\begin{pmatrix} y_i \\ z_i \geq d_i \end{pmatrix} \vee \begin{pmatrix} \neg y_i \\ u_i \leq 0 \end{pmatrix}$ for each i
Inference: none.
Relaxation: Generate the projected big-M relaxation for the LP.

7. Constraint: Propositional Logic

$y_1 \rightarrow (y_2 \vee y_3)$	$y_3 \rightarrow y_4$
$y_2 \rightarrow y_1$	$y_3 \rightarrow (y_5 \vee y_6)$
$y_2 \rightarrow (y_4 \vee y_5)$	$y_4 \rightarrow (y_2 \vee y_3)$
$y_2 \rightarrow y_6$	$y_5 \rightarrow (y_2 \vee y_3)$
$y_3 \rightarrow y_1$	$y_6 \rightarrow (y_2 \vee y_3)$

Inference: Apply the resolution method.
Relaxation: Add reduced domains to constraint store.
(One could generate linear inequalities that relax the propositions.)

8. Search
Procedure **BandBsearch**($P, R, S,$ **NetBranch**)

9. User defined procedure
```
Procedure NetBranch($P, R, S, i$)
    Let $i$ be a unit for which $u_i > 0$ and $z_i < d_i$.
    Let $D_i$ be the domain of $y_i$.
    If $i = 1$ then create $P'$ from $P$ by letting $D_i = \{T\}$ and return $P'$.
    If $i = 2$ then create $P'$ from $P$ by letting $D_i = \{F\}$ and return $P'$.
```

Figure 4. *Model for the processing network design problem.*

as much as possible, using straightforward interval arithmetic. This is called *bounds consistency maintenance.*

Window 6 requires that one either pay for a unit or shut it down. That is, either $z_i \geq d_i$ or $u_i \leq 0$ for each unit i. The same constraint requires that if y_i is true, then $z_i \geq d_i$, and otherwise $u_i \leq 0$. It in effect defines y_i. This type of disjunctive constraint receives its own window because is quite common in modeling situations and can be given a succinct relaxation [11]. In the present case the relaxation simplifies to $c_i z_i \geq d_i u_i$ for each i.

One can accelerate the search by making some simple observations. It is clearly pointless to install unit 1 unless one installs unit 2 or unit 3. This can be written $y_1 \rightarrow (y_2 \vee y_3)$. Rules of this sort have been called "don't be stupid constraints" and appear in window 7. A large number of don't-be-stupid rules can be used when they are processed symbolically rather than added to the relaxation. Logical inference methods (such as the resolution method) can be applied to fix variables or detect infeasibility.

The search window specifies a traditional depth-first branching search. The branching is directed by which disjunctions in window 6 are violated by the solution of the linear relaxation. The search branches on y_i for some unit i whose disjunction is violated.

6. Example: Lot Sizing

A lot sizing problem discussed by Wolsey (1998) illustrates the role of conditional constraints. (We modify the example slightly.) Several products i must be shipped in specified quantities d_{it} on each day t. However, at most one product can be manufactured on a given day, so that inventory must be accumulated. The unit daily holding cost for product i is h_i, and q_{ij} is the cost of switching the manufacturing process from product i to product j ($q_{ii} = 0$). A product may be manufactured for a fraction of a day or for several days in a row.

Let y_t be the product manufactured on day t, with $y_t =$ null if nothing is manufactured. Note that y_t need not have a numerical value. Let x_{it} be the quantity of product i manufactured on day t. The stock level of product i on day t is s_{it}. These declarations appear in the first window of Fig. 5.

The objective (window 2) is to minimize the total cost of holding inventory and switching from one product to another, where the latter is indicated by v_t. The calculation of v_t incurs modeling difficulties in traditional integer programming, because a large number of doublyindexed 0-1 variables must normally be introduced for this purpose. Here the definition is given by window 7, to be discussed shortly.

Window 4 specifies a continuous relaxation. In addition, a solution of the relaxation is a candidate solution is it satisfies the conditional constraint in window 6. Window 5 contains the linear inventory balance constraints, which can be added directly to the relaxation.

The conditional constraint in window 6 is a versatile device. In general it has the form $A \rightarrow C$, and its function is to impose the consequent C as a constraint whenever, during the course of the search, the antecedent A becomes true. In the present case, the antecedent becomes true, and $x_{it} = 0$ is enforced, whenever the domain of y_t is reduced to the point that it excludes i.

The `element` constraint in window 6 defines $v_t = q_{y_{t-1},y_t}$; that is, the setup cost incurred on day t is the cost of switching from product y_{t-1} yesterday to product y_t today. The variable subscript of q is a pair (y_{t-1}, y_t), and we impose the constraint `element`$((y_{t-1}, y_t), Q, v_t)$, where Q is the matrix of q_{ij}'s.

The search window implements a standard first-fail branching algorithm.

7. Example: An Allocation Problem

An example proposed by Williams [27] illustrates a local search model. A firm wants each of its retail outlets to be supplied by one of the firm's two divisions. There are several products, and the division that supplies retailer j must provide it a_{ij} units of product i. The company wants division 1 to control a certain fraction of the market for each product. Let b_i be the corresponding quantity of product i that division 1 should supply. The problem is to approximate the desired allocation as closely as possible. It might be formulated as the following 0-1 programming problem.

$$\begin{array}{ll} \text{minimize} & \sum_i |s_i| \\ \text{subject to} & \sum_j a_{ij}x_j + s_i = b_i, \;\; \text{all } i \\ & x_j \in \{0,1\}, \;\; \text{all } j \\ & s_i \in \mathcal{R}, \;\; \text{all } i \end{array} \tag{1}$$

Where $x_j = 1$ when division 1 supplies retailer j, and $x_j = 0$ otherwise. The problem is known to be very hard for MP methods [3]. A local search method may therefore be appropriate.

The model of Fig. 6 illustrates a simulated annealing search. A random neighbor of the current solution is selected. If it is superior to the current solution, the search moves to the neighboring solution, and

1. Variables and Initial Domains
$s_{it} \in [0, \infty)$ (stock level of product i in period t)
$x_{ij} \in [0, C]$ (production level of product i in period t)
$y_t \in \{\text{prod } 1, \ldots, \text{prod } n, \text{null}\}$ (product manufactured in period t, if any)

2. Objective Function
maximize $\sum_t \left(\sum_i h_i s_{it} + v_t\right)$

3. Relaxation R_1
Type: Constraint store, consisting of variable domains.
Objective function: none.
Solver: select a value from each domain.

4. Relaxation R_2
Type: linear programming.
Objective function: maximize $\sum_t \left(\sum_i h_i s_{it} + v_t\right)$
Solver: linear programming.
Set incumbent value $\bar{z} = -\infty$, where $\bar{z}$ is a global variable.

5. Constraint: Linear Inequalities
$s_{i,t-1} + x_{it} = d_{it} + s_{it}$, all i, t
Inference: bounds consistency maintenance.
Relaxation: add reduced domains to the constraint store.
Relaxation: add all these inequalities to the LP relaxation.

6. Constraint: Conditional
$(y_t \neq i) \to (x_{it} = 0)$, all i, t
Inference: none.
Relaxation: add reduced domains to the constraint store.
Relaxation: add consequent to LP relaxation if antecedent is true.

7. Constraint: Element
Element$((y_{t-1}, y_t), Q, v_t)$, all t
Inference: maintain hyperarc consistency.
Relaxation: add reduced domains to the constraint store.
Relaxation: disjunctive relaxation.

8. Search
Procedure **BandBsearch**$(P, R, S,$ **FirstFailBranch**$)$

9. User defined procedure

```
Procedure FirstFailBranch(P, R, S)
    Let the discrete variable with the smallest domain have domain $D$.
    Split $D$ into $D_1$ and $D_2$.
    If $i = 1$ then create $P'$ from $P$ by letting $D = D_1$ and return $P'$.
    If $i = 2$ then create $P'$ from $P$ by letting $D = D_2$ and return $P'$.
```

Figure 5. Model for the lot sizing problem.

otherwise it moves to the neighboring solution with probability p. In general p depends on the "temperature," which falls over time, but for

1. Variables and Initial Domains
$x_j \in D_i = \{0\}$, $j = 1, \ldots, n$ (takes value 1 when division 1 is assigned retailer j)
$s_i \in (-\infty, \infty)$, $i = 1, \ldots, m$ (error in meeting goal for product i)

2. Objective Function
minimize $f(x) = \sum_i |s_i|$

3. Relaxation R_1
Type: linear.
Objective function: minimize $\sum_i |s_i|$
Solver: direct computation.

4. Constraint: Linear Inequalities
$s_i = b_i - \sum_j a_{ij} x_j$, all i
Inference: none.
Relaxation: $s_i = b_i - \sum_j a_{ij} x_j$, with each x_j is fixed to the single value in D_j

5. Search: Simulated Annealing

```
Procedure Search($P, R, S$).
    Return if search has run long enough.
    Let random($p$) be a random variable that has value 1 with probability $p$.
    To flip a domain $D_j = \{\alpha\}$ is to change it to $\{1 - \alpha\}$.
    Perform relax($P, R_1, S_1, v$).
    Do forever:
        Randomly select $j$ from $\{1, \ldots, n\}$ and change $P$ by flipping $D_j$.
        Perform relax($P, R_1, S_1, v'$).
        If $v' < v$ or random($p$) = 1 then perform Search($P, R, S$) and return.
        Change $P$ back by flipping $D_j$.
```

Figure 6. Model for the allocation problem.

simplicity temperature is assumed constant. The search begins with all $x_j = 0$.

The objective function is calculated in a linear relaxation R_1 of (1), in which the x_j's are fixed to the values in their current domains. This is the only relaxation used by the model.

8. Example: Benders Decomposition

Benders decomposition integrates two solution methods: one that solves the master problem, and one that solves the subproblem. We will consider the case of classical Benders applied to integer programming, which combines integer programming (master problem solver) and linear programming (subproblem solver). This will show how the modeling framework proposed here can be adapted to a suite of solution methods other than CP, MP and heuristics.

Classical Benders decomposition can be applied to problems of the form

$$\begin{array}{ll} \text{minimize} & cx + dy \\ \text{subject to} & Ax + By \geq a \\ & x \in \mathcal{R}^n,\ y \in \mathcal{Z}^m \end{array}$$

The basic strategy of the method is to try different values of y and find an optimal x for each one in a *subproblem.* Thus if y is fixed to $\bar{y}$ the subproblem is the linear programming problem

$$\begin{array}{ll} \text{minimize} & cx + d\bar{y} \\ \text{subject to} & Ax \geq a - B\bar{y} \end{array} \tag{2}$$

The dual of (2) is

$$\begin{array}{ll} \text{minimize} & u(a - B\bar{y}) + d\bar{y} \\ \text{subject to} & uA = c \\ & u \geq 0 \end{array}$$

If the subproblem (2) is feasible and bounded, and if $\bar{u}$ solves the dual, one obtains a Benders cut

$$z \geq \bar{u}(a - By) + dy \tag{3}$$

If the subproblem is infeasible, and $\bar{u}$ is an extreme ray solution of the dual, the Benders cut becomes

$$\bar{u}(a - By) \leq 0 \tag{4}$$

(We assume that the subproblem and its dual are not both infeasible.) If the subproblem is unbounded, the original problem is also unbounded, and the procedure stops. Otherwise one solves a *master problem* of the form

$$\begin{array}{ll} \text{minimize} & z \\ \text{subject to} & \text{cuts of the form (3) and (4) generated so far} \end{array}$$

The solution $\bar{y}$ of the master problem defines the next subproblem. The process continues until the master problem and subproblem have the same optimal value.

This process can be viewed as searching over restrictions of the original problem, namely over subproblems defined by the fixed value $\bar{y}$. Each time a subproblem is formulated, an inference method (solution of the linear programming dual) is used to infer a Benders cut. The master problem is a relaxation of the original problem, and its solution guides the search strategy by defining the next subproblem. The Benders algorithm is therefore easily formulated in the framework proposed here,

1. Variables and Initial Domains
$x_i \in \mathcal{R}$, all i
$y_j \in D_j = \{0\}$, all j (integer variables initially set to 0)

2. Objective Function
minimize $cx + dy$

3. Relaxation R_1
Type: Integer programming problem (master problem).
Objective function: minimize z.
Solver: integer programming solver.
Global variable $\bar{z}$ is optimal value of master problem.

4. Constraint: Linear Inequalities
$Ax + By \geq a$.
Inference: generation of Benders cuts by solving dual.
Relaxation: none.

5. Constraint: Benders cuts generated by inference
Benders cuts of the form $z \geq \bar{u}(a - By) + dy$ when the subproblem is bounded, or of the form $\bar{u}(a - By) \leq 0$ when the subproblem is unbounded, where the subproblem is P minus Benders cuts, and $\bar{u}$ is the solution of the subproblem dual. Initially there are no Benders cuts.
Inference: none.
Relaxation: add all Benders cuts to master problem.

6. Search
```
Procedure Search($P, R, S$)
    Perform Infer($P, R, S$). (Generate Benders cuts.)
    If dual is infeasible, stop with unbounded solution.
    Perform Relax($P, R, S, v$). (Solve master problem.)
    If $v$ is equal to the value of the dual, stop.
    Obtain $P'$ from $P$ by setting each $D_j$ to $\{\bar{y}_j\}$, where $\bar{y}_j$ is value of $y_j$ in $S$
        (solution of master).
    Perform Search($P', R, S$).
```

Figure 7. Model for classical Benders decomposition.

using one relaxation. The details appear in Fig. 7. Note that one of the constraint windows is initially empty and simply serves as a collection bin for Benders cuts.

9. Example: Machine Scheduling

In the above treatment of Benders decomposition, the use of duality to derive a Benders cut from the restricted problem (subproblem) is regarded as an inference method. This suggests a generalization of Benders in which any dual-based method of inferring valid constraints from the restricted problem is regarded as generation of a Benders cut. Interestingly, the linear programming dual can be interpreted as a special case of a general "inference dual," described in [], whose solution is the inference of a valid bound on the objective function. This is exactly what a Benders cut is.

CP provides a natural context in which to infer Benders cuts, due to the importance of inference methods in the field. In fact, Benders decomposition has been explored recently as an alternate method of combining CP with other methods, particularly MP [8, 12]. A CP method is applied to the subproblem (i.e., the current problem restriction) to obtain cuts, and an MP method solves the master problem (i.e., the relaxation). It follows that this Benders-based approach to combining methods is a special case of the framework proposed here.

Jain and Grossman's solution of a machine scheduling problem [16] nicely illustrates how CP can generate Benders cuts, and it also achieved dramatic computational success. The results are impressive because this particular problem happens to be especially well suited to a Benders approach, but it is likely that other problems sith similar structure may also benefit from it, such as a vehicle routing problem with time windows.

The problem may be stated as follows. Each job j is assigned to one of several machines i that operate at different speeds. Each assignment results in a processing time T_{ij} and incurs a fixed processing cost C_{ij}. There is a release date R_j and a due date S_j for each job j. The objective is to minimize processing cost while observing release and due dates.

To formulate the problem we need the `cumulative` constraint, which formulates a resource-constrained scheduling problem. It is written

$$\texttt{cumulative}(t, d, r, L)$$

where $t = (t_1, \ldots, t_n)$ is a vector of start times, d is a vector of corresponding job durations, and r a vector of corresponding resource consumption rates. The constraint requires that the total rate of resource consumption at any one time not exceed L:

$$\sum_{\substack{j \\ t_j \le t \le t_j + d_j}} r_j \le L, \quad \text{all } t$$

Let y_j be the machine to which job j is assigned and t_j the start time for job j. It also convenient to let $(t_j \mid y_j = i)$ denote the tuple of start times of jobs assigned to machine i, arranged in increasing order of the job number. The problem can be written

$$\begin{array}{lll} \text{minimize} & \sum_j C_{y_j j} & (a) \\ \text{subject to} & t_j \geq R_j, \ \ \text{all } j & (b) \\ & t_j + T_{y_j j} \leq S_j, \ \ \text{all } j & (c) \\ & \texttt{cumulative}((t_j \mid y_j = i), (T_{ij} \mid y_j = i), e, 1), \ \ \text{all } i & (d) \end{array}$$

The objective function (a) measures the total processing cost. Constraints (b) and (c) observe release times and deadlines. The cumulative constraint (d) gives each job a resource consumption rate of 1 and sets the maximum total rate to 1 (e is a vector of ones). It ensures that jobs assigned to each machine are scheduled so that they do not overlap.

The problem has two parts: the assignment of jobs to machines, and the scheduling of jobs on each machine. The assignment problem is treated as the master problem and solved with mixed integer programming methods. Once the assignments are made, the subproblems are dispatched to a constraint programming solver to find a feasible schedule. If there is no feasible schedule, a Benders cut is generated.

If y is fixed to $\bar{y}$, the subproblem is

$$\begin{array}{l} \texttt{cumulative}((t_j \mid \bar{y}_j = i), (T_{ij} \mid \bar{y}_j = i), e, 1), \ \ \text{all } i \\ R_j \leq t_j \leq S_j - D_{\bar{y}_j j}, \ \ \text{all } j \end{array}$$

The bounds on t_j reduce the domain of t_j before one applies `cumulative`. The subproblem can be decomposed into smaller problems, one for each machine. If CP methods determine that the smaller problem is infeasible for some i, then the jobs assigned to machine i cannot all be scheduled on that machine. In fact, going beyond Jain and Grossmann, there may be a subset J of these jobs that cannot be scheduled on machine i. This gives rise to a Benders cut stating that at least one of the jobs in J must be assigned to another machine.

$$\bigvee_{j \in J} (y_j \neq i)$$

where the symbol $\bigvee$ denotes a disjunction. Let y^k be the solution of the kth master problem, I^k the set of machines i in the resulting subproblem for which the schedule is infeasible, and J_{ki} the infeasible subset. The master problem can now be written,

$$\begin{array}{ll} \text{minimize} & \sum_j C_{y_j j} \\ \text{subject to} & \bigvee_{j \in J_{ki}} (y_j \neq i), \ \ i \in I^k, \ k = 1, \ldots, K \end{array}$$

The master problem can be reformulated for solution with conventional integer programming technology. Let y_{ij} be a 0-1 variable that is 1 when job j is assigned to machine i. The master problem (**??**) can be written

$$\begin{array}{ll}\text{minimize} & \sum\limits_{i,j} C_{ij} y_{ij} \\ \text{subject to} & \sum\limits_{j \in J_{ki}} (1 - y_{ij}) \geq 1, \;\; i \in I^k, \;\; k = 1, \ldots, K \\ & y_{ij} \in \{0,1\}, \;\; \text{all } i, j \end{array}$$

As noted in the previous section, the master problem can be regarded as a relaxation of the restricted problem. The relaxation can be tightened as follows:

$$\begin{array}{lll}\text{minimize} & \sum\limits_{i,j} C_{ij} y_{ij} & (a) \\ \text{subject to} & u_j \geq R_j, \;\; \text{all } j & (b) \\ & u_j + \sum\limits_{i} T_{ij} y_{ij} \leq S_j \;\; \text{all } j & (c) \\ & \sum\limits_{j \in J_{ki}} (1 - y_{ij}) \geq 1, \;\; i \in I^k, \;\; k = 1, \ldots, K & (d) \\ & \sum\limits_{j} T_{ij} y_{ij} \leq \max\limits_{j} \{S_j\} - \min\limits_{j} \{R_j\}, \;\; \text{all } i & (e) \\ & y_{ij} \in \{0,1\}, \;\; \text{all } i, j & (f) \\ & \underline{u} \leq u \leq \overline{u} & (g) \end{array}$$

The variables u_j have the same meaning as the start time variables t_j but are formally distinguished for purposes of solving the relaxation. The bounds in (g) are taken from the current domains of t. Constraints (b) and (c) simply observe the release times and due dates. Constraints (e) say that the total processing time on each machine must fit between the earliest release time and the latest deadline. It is reported in [25] that simply dropping (e) from the relaxation makes the solution method much less effective.

The model appears in Fig. 8. Note that the element constraint in Window 4 defines the objective function. In practice one might install a feature to create element constraints automatically to implement variable indices.

10. Conclusion

We presented a simple framework for combining solution methods that is based on their common solution strategies: search over problem restrictions, inference of new constraints, and solution of relaxations. The

1. Variables and Initial Domains
$t_j \in [0, \infty)$, all j (start times)
$z_j \in [0, \infty)$, all j (cost of processing job j)
$y_j \in T_j = \{1\}$, all i (start with arbitrary assignment of jobs to machines)

2. Objective Function
minimize $\sum_j w_j$

3. Relaxation R_1
Type: Integer programming problem (master problem).
Objective function: minimize $\sum_j w_j$.
Solver: integer programming solver.

4. Constraint: element
`element`$(y_j, (C_{1j}, \ldots, C_{mj}), w_j)$, all j.
Inference: domain reduction.
Relaxation: disjunctive relaxation.

5. Constraint: Linear Inequalities
$R_j \le t_j \le S_j - T_{y_j j}$, all j.
Inference: bounds consistency maintenance.
Relaxation: $R_j \le u_j \le S_j - T_{y_j j}$, all j
    $\sum_j T_{ij} y_{ij} \le \max_j\{S_j\} - \min_j\{R_j\}$, all j
    plus bounds on u_j's derived from current domains of t_j's.
    Fix y_j's to values in current master problem solution.

6. Constraint: Benders cuts generated by inference
Benders cuts of the form $\sum_{j\in J}(1 - y_{ij}) \ge 1$, where
    the subproblem is P minus Benders cuts. Initially there are no Benders cuts.
Inference: none.
Relaxation: add all Benders cuts to master problem.

7. Search
```
Procedure **Search**($P, R, S$)
    Perform **Infer**($P, R, S$). (Generate Benders cuts using CP.)
    If subproblem is feasible, stop with optimal solution.
    Perform **Relax**($P, R, S, v$). (Solve master problem.)
    Obtain $P'$ from $P$ by setting each $D_j$ to $\{\bar{y}_j\}$, where $\bar{y}_j$ is value of $y_j$ in $S$
        (solution of master).
    Perform **Search**($P', R, S$).
```

Figure 8. Model for a machine scheduling problem.

problem is modeled in a set of windows that define variables, constraints, and the objective function. Additional windows initialize relaxations and specify the search method.

Subsets of constraints with special structure are placed in separate constraint windows that have their own syntax for specifying constraints.

Each constraint window uses specialized inference methods and relaxations that exploit the structure of its constraints. The inference methods may include domain reduction (filtering) and cutting planes. The relaxations might consist of in-domain constraints (for a constraint store) or linear inequalities.

This a modeling style quite different from that of traditional operations research, which tends to write everything in terms of a few elementary constructs such as inequalities. The approach suggested here requires that the modeler be familiar with a library of global constraints. Yet it results in a simpler and more readable model and allows the solver to exploit the structure of the model. It can take advantage of the large literature on specialized cutting planes and other specialized methods.

The windows are linked by underlying data structures that hold the current problem restriction and the relaxations. The relaxations and their solutions provide information for directing the search. The relaxation windows initialize the relaxation data structures.

The search proceeds by enumerating problem restrictions, either exhaustively or heuristically. A search window directs the search recursively by invoking an a generic inference procedure, a generic relaxation procedure, and itself. The inference procedure cycles through the constraints in a specified way and triggers the generation of new constraints for addition to the current problem restriction (in-domain constraints, cutting planes, etc.). The relaxation procedure assembles the relaxations generated by the constraint windows.

The search procedure may be a canned procedure that accepts parameters, such as a branch-and-bound or first-fail branching search, or it may written by the user.

The framework was designed for combining CP, MP and heuristic methods, but it can represent any solution method that involves search over problem restrictions, inference and relaxation. This fact was illustrated by Benders decomposition, which fits readily into the framework. It can be viewed as an enumeration of problem restrictions (subproblems) that infers constraints (Benders cuts) and is directed by the solution of a relaxation (master problem). Since a recent and promising method for integrating CP and MP uses generalized Benders decomposition, it is a special case of the framework proposed here. Benders is generalized by using domain reduction, rather than linear programming duality, to infer the Benders cuts.

This design proposed here should allow a software developer to build the basic framework and add to it as desired.

References

[1] Bockmayr, A., T. Kasper. 1998. Branch and infer: A unifying framework for integer and finite domain constraint programming, *INFORMS Journal on Computing* **10** 287–300.

[2] Bollapragada, S., O. Ghattas, J. N. Hooker. 2001. Optimal design of truss structures by mixed logical and linear programming, *Operations Research* **49** (2001) 42–51.

[3] Cornuéjols, G. and M. Dawande. 1999. A class of hard small 0-1 programs, *INFORMS Journal on Computing* **11** 205-210.

[4] Grossmann, I. E., J. N. Hooker, R. Raman, H. Yan. 1994. Logic cuts for processing networks with fixed charges, *Computers and Operations Research* **21** 265–279.

[5] Hooker, J. N. 1992. Generalized resolution for 0-1 linear inequalities, *Annals of Mathematics and Artificial Intelligence* **6** 271–286.

[6] Hooker, J. N. 1994. Logic-based methods for optimization, in A. Borning, ed., *Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science **874** 336–349.

[7] Hooker, J. N. 1997. Constraint satisfaction methods for generating valid cuts, in D. L. Woodruff, ed., *Advances in Computational and Stochasic Optimization, Logic Programming and Heuristic Search*, Kluwer (Dordrecht) 1–30.

[8] Hooker, J. N. 2000. *Logic-Based Methods for Optimization: Combining Optimization and Constraint Satisfaction*, Wiley (New York).

[9] Hooker, J. N. 2001. Logic, optimization and constraint programming, to appear in *INFORMS Journal on Computing.*

[10] Hooker, J. N., Hak-Jin Kim, G. Ottosson. 2001. A declarative modeling framework that combines solution methods, *Annals of Operations Research*, *Annals of Operations Research* **104** 141–161.

[11] Hooker, J. N., M. A. Osorio. 1999. Mixed logical/linear programming, *Discrete Applied Mathematics* **96-97** 395–442.

[12] Hooker, J. N. G. Ottosson. 1998. Logic-based Benders decomposition, to appear in *Mathematical Programming.*

[13] Hooker, J. N., G. Ottosson, E. Thorsteinsson, Hak-Jin Kim. 1999. On integrating constrain propagation and linear programming for combinatorial optimization, *Proceeedings, 16th National Conference on Artificial Intelligence*, MIT Press (Cambridge) 136–141.

[14] Hooker, J. N., G. Ottosson, E. Thorsteinsson, Hak-Jin Kim. 2000. A scheme for unifying optimization and constraint satisfaction methods, *Knowledge Engineering Review* **15** 11–30.

[15] Hooker, J. N. and Hong Yan. 2001. A continuous relaxation of the cumulative constraint, manuscript, Hong Kong Polytechnic University.

[16] Jain, V., and I. E. Grossmann. 2001. Algorithms for hybrid MILP/CP models for a class of optimization problems, *INFORMS Journal on Computing* **13** 258-276.

[17] Kim, Hak-Jin, and J. N. Hooker. 2002. Solving fixed-charge network flow problems with a hybrid optimization and constraint programming approach. To appear in *Annals of Operations Research.*

[18] Marriott, K., P. J. Stuckey. 1998. *Programming with Constraints: An Introduction*, MIT Press (Cambridge).

[19] Nemhauser, G. L., and L. A. Wolsey. 1999. *Integer and Combinatorial Optimization*, Wiley (New York).

[20] Ottosson, G., E. Thorsteinsson. 2000. Linear relaxations and reduced-cost based propagation of continuous variable subscripts, presented at Second International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, University of Paderborn.

[21] Ottosson, G., E. Thorsteinsson, J. N. Hooker. 1999. Mixed global constraints and inference in hybrid CLP-IP solvers, *CP99 Post-Conference Workshop on Large Scale Combinatorial Optimization and Constraints*, http://www.dash.co.uk/wscp99, 57–78.

[22] Raman, R., I. Grossmann. 1991. Symbolic integration of logic in mixed-integer linear programming techniques for process synthesis, *Computers and Chemical Engineering* **17** 909–927.

[23] Raman, R., I. Grossmann. 1993. Relation between MILP modeling and logical inference for chemical process synthesis, *Computers and Chemical Engineering* **15** 73–84.

[24] Raman, R., I. Grossmann. 1994. Modeling and computational techniques for logic based integer programming, *Computers and Chemical Engineering* **18** 563–578.

[25] Thorsteinsson, E. S. 2001. Branch-and-check: A hybrid framework integrating mixed integer programming and constraint logic programming, *Seventh International Conference on Principles and Practice of Constraint Programming (CP2001), Lecture Notes in Computer Science* **2239**.

[26] Türkay, M., I. E. Grossmann. 1996. Logic-based MINLP algorithms for the optimal synthesis of process networks, *Computers and Chemical Engineering* **20** 959–978.

[27] Williams, H. P. 1999. *Model Building in Mathematical Programming, 4th ed.*, Wiley (New York).

[28] Williams, H. P., and Hong Yan. 2001. Representations of the all-different predicate, *INFORMS Journal on Computing*, to appear.

[29] Wolsey, L. A. 1998. *Integer Programming*, Wiley (New York).

[30] Yan, H., and J. N. Hooker. 1999. Tight representation of logical constraints as cardinality rules, *Mathematical Programming* **85** 363–377.

[31] Yan, H., and H. P. Williams. 2001. Convex hull representations of the at-least predicate of constraint satisfaction, manuscript, Hong Kong Polytechnic University.