WORKING PAPER WPS-MAS-08-01
Department of Management Science
School of Business Administration, University of Miami
Coral Gables, FL 33124-8237

Created: December 2005
Updated: August 21, 2008

An Integrated Solver for Optimization Problems

Tallys Yunes
Department of Management Science, School of Business Administration, University of Miami, Coral Gables, FL 33124-8237, tallys@miami.edu

Ionuţ D. Aron
ionut.aron@gmail.com

John Hooker
Tepper School of Business, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, john@hooker.tepper.cmu.edu

One of the central trends in the optimization community over the past several years has been the steady improvement of general-purpose solvers. Recent research in the area of integrated problem solving suggests that the right combination of different technologies can simplify modeling and speed up computation substantially. Nevertheless, integration often requires special purpose coding, which is time-consuming and error-prone. A logical next step in this evolution is to combine mixed integer linear programming, constraint programming, and global optimization in a single system. We present a general purpose solver, SIMPL, that allows its user to replicate (and sometimes improve on) the results of custom implementations with concise models written in a high-level language. We apply SIMPL to production planning, product configuration, and machine scheduling problems on which customized integrated methods have shown significant computational advantage. We obtain results that either match or surpass the original codes at a fraction of the implementation effort. We also solve pooling, distillation, and heat exchanger network design problems to demonstrate how global optimization fits into SIMPL's framework.

Subject classifications: Programming: linear, nonlinear, integer and constraint programming; modeling languages; global optimization; integrated optimization. Production: planning and product configuration. Scheduling: parallel machines.

1. Introduction

One of the central trends in the optimization community over the past several years has been the steady improvement of general-purpose solvers. Such mixed integer solvers as CPLEX and XPRESS-MP have become significantly more effective and robust, and similar advancements have occurred in continuous global optimization (BARON, LGO) and constraint programming (CHIP, ILOG CP Optimizer). These developments promote the use of optimization and constraint solving technology, since they spare practitioners the inconvenience of acquiring and learning different software for every application.

A logical next step in this evolution is to combine mixed integer linear programming (MILP), constraint programming (CP), and global optimization in a single system. This not only brings together a wide range of methods under one roof, but it allows users to reap the advantages of integrated problem solving. Recent research in this area shows that the right combination of different technologies can simplify modeling and speed up computation substantially. Commercial-grade solvers are already moving in this direction, as witnessed by the ECL^{i}PS^e solver, OPL Studio, and the Mosel language, all of which combine mathematical programming and constraint programming techniques to a greater or lesser extent.

Our objective, however, is *not* to demonstrate that integrated methods can sometimes be superior to the state-of-the-art standalone MILP or CP solvers. Firstly because no single paper can do this, and secondly because this has already been done (see Table 1 for a sample and Hooker (2007) for dozens of other examples). Our contribution is to show, both theoretically and empirically, that it is possible to unify most of what we know about integrated methods under one paradigm, without sacrificing performance.

Integration is most effective when techniques interleave at a micro level. To achieve this in current systems, however, one must often write special-purpose code, which slows research and discourages application. We have attempted to address this situation by designing an architecture that achieves low-level integration of solution techniques with a high-level modeling language. The ultimate goal is to build an integrated solver that can be used as conveniently as current mixed integer, global and constraint solvers. We have implemented our approach in a system called SIMPL, which can be read as a permuted acronym for Modeling Language for Integrated Problem Solving.

SIMPL is based on two principles: algorithmic unification and constraint-based control. *Algorithmic unification* begins with the premise that integration should occur at a fundamental and conceptual level, rather than postponed to the software design stage. Optimization methods and their hybrids should be viewed, to the extent possible, as special cases of a single solution method that can be adjusted to exploit the structure of a given problem. We address this goal with a *search-infer-and-relax* algorithmic framework, coupled with *constraint-based control* in the modeling language. The search-infer-and-relax scheme encompasses a wide variety of methods, including branch-and-cut methods for integer programming, branch-and-infer methods for constraint programming, popular methods for continuous global optimization, such nogood-based methods as Benders decomposition and dynamic backtracking, and even heuristic methods such as local search and greedy randomized adaptive search procedures (GRASPs).

Constraint-based control allows the design of the model itself to tell the solver how to combine techniques so as to exploit problem structure. Highly-structured subsets of constraints are written as metaconstraints, which are similar to "global constraints" in constraint programming. At the syntactic level, a metaconstraint is written like a linear or global constraint but it is followed by additional statements that describe the way that constraint should behave during the solution process (see Section 5 for details). For example, a metaconstraint may define the way it is going to be relaxed, the way it will do domain filtering, and/or the way to branch on a node in which infeasibility is caused by that constraint. When such definitions are omitted, some pre-specified default behavior will be used. The relaxation, inference and branching techniques are devised for each constraint's particular structure. For example, a metaconstraint may be associated with a tight polyhedral relaxation from the integer programming literature and/or an effective domain filter from constraint programming. Because constraints also control the search, if a branching method is explicitly indicated for a metaconstraint, the search will branch accordingly.

The selection of metaconstraints to formulate the problem determines how the solver combines algorithmic ideas to solve the problem. This means that SIMPL deliberately sacrifices independence of model and method: the model must be formulated with the solution method in mind. However, we believe that successful combinatorial optimization leaves no alternative. This is evident in both integer programming and constraint programming, since in either case one must carefully write the formulation to obtain tight relaxations, effective propagation, or intelligent branching. We attempt to make a virtue of necessity by explicitly providing the resources to shape the algorithm through a high-level modeling process.

We focus here on branch-and-cut, branch-and-infer, generalized Benders, and a sub-class of global optimization methods, since these have been implemented so far in SIMPL. The system architecture is designed, however, for extension to general global optimization, general nogood-based methods, and heuristic methods.

The main contribution of this paper is to demonstrate that a general-purpose solver and modeling system can achieve the computational advantages of integrated methods while preserving much of the convenience of existing commercial solvers. We use SIMPL to model and solve four classes of problems that have been successfully solved by custom implementations of integrated approaches. We find that a properly engineered high-level modeling language and solver can match and even exceed the performance of hand-crafted implementations. Although high-level descriptions of some of SIMPL's ideas and architecture have been published before (Aron, Hooker and Yunes 2004), this is the first time that they are empirically demonstrated with concrete examples.

After a brief survey of previous work, we review the advantages of integrated problem solving and present the search-infer-and-relax framework. We then summarize the syntax and semantics of SIMPL models. Following this we describe how to model and solve the four problem classes just mentioned in an integrative mode. A production planning problem with semi-continuous piecewise linear costs illustrates metaconstraints and the interaction of inference and relaxation. A product configuration problem illustrates variable indices and how further inference can be derived from the solution of a relaxation. Finally, a planning and scheduling problem and a few chemical engineering problems show how Benders decomposition and global optimization methods, respectively, fit into the framework. In each case we exhibit the SIMPL model and present computational results. We conclude with suggestions for further development.

2. Previous Work

Comprehensive surveys of hybrid methods that combine CP and MILP are provided by Hooker (2000, 2002, 2006, 2007), and tutorial articles may be found in Milano (2003).

Various elements of the search-infer-and-relax framework presented here were proposed by Hooker (1994, 1997, 2000, 2003), Bockmayr and Kasper (1998), Hooker and Osorio (1999), and Hooker et al. (2000). An extension to dynamic backtracking and heuristic methods is given in Hooker (2005). The present paper builds on this framework and introduces the idea of constraint-based control, which is key to SIMPL's architecture. A preliminary description of the architecture appears in a conference paper (Aron, Hooker and Yunes 2004), without computational results. The present paper develops these ideas further and demonstrates that SIMPL can reproduce the computational advantages of integrated methods with much less implementation effort.

Existing hybrid solvers include ECL^{i}PS^e, OPL Studio, Mosel, and SCIP. ECL^{i}PS^e is a Prolog-based constraint logic programming system that provides an interface with linear and MILP solvers (Rodošek, Wallace and Hajian 1999; Cheadle et al. 2003; Ajili and Wallace 2003). The CP solver in ECL^{i}PS^e communicates tightened bounds to the MILP solver, while the MILP solver detects infeasibility and provides a bound on the objective function that is used by the CP solver. The optimal solution of the linear constraints in the problem can be used as a search heuristic.

OPL Studio provides an integrated modeling language that expresses both MILP and CP constraints (Van Hentenryck et al. 1999). It sends the problem to a CP or MILP solver depending on the nature of constraints. A script language allows one to write algorithms that call the CP and MILP solvers repeatedly.

Mosel is both a modeling and programming language that interfaces with various solvers, including MILP and CP solvers (Colombani and Heipcke 2002, 2004). SCIP is a callable library that gives the user control of a solution process that can involve both CP and MILP solvers (Achterberg 2004).

3. Advantages of Integrated Problem Solving

One obvious advantage of integrated problem solving is its potential for reducing computation time. Table 1 presents a sampling of some of the better computational results reported in the literature, divided into four groups. (a) Early efforts at integration coupled solvers rather loosely but obtained some speedup nonetheless. (b) More recent hybrid approaches combine CP or logic processing with various types of relaxations used in MILP, and they yield more substantial speedups. (c) Several investigators have used CP for column generation in a branch-and-price MILP algorithm, as independently proposed by Junker et al. (1999) and Yunes, Moura and de Souza (1999). (d) Some of the largest computational improvements to date have been obtained by using generalizations of Benders decomposition to unite solution methods, as proposed by Hooker (2000) and implemented for CP/MILP by Jain and Grossmann (2001). MILP is most often applied to the master problem and CP to the subproblem.

We solved three of the problem classes in Table 1 with SIMPL: piecewise linear costs (Refalo 1999; Ottosson, Thorsteinsson and Hooker 1999, 2002), product configuration (Thorsteinsson and Ottosson 2001), and planning and scheduling (Jain and Grossmann 2001). We selected these problems because the reported results are some of the most impressive, and because they illustrate a variety of solution approaches.

Our experiments show that SIMPL reproduces or exceeds the reported advantage of integrated methods over the state of the art at that time. This level of performance can now be obtained with much less effort than invested by the original authors. All of the problems in Table 1 can in principle be implemented in a search-infer-and-relax framework, although SIMPL in its current form is not equipped for all of them.

Aside from computational advantages, an integrated approach provides a richer modeling environment that can result in simpler models and less debugging. The full repertory of global constraints used in CP are potentially available, as well as nonlinear expressions used in continuous global optimization. Frequent use of metaconstraints not only simplifies the model but allows the solver to exploit problem structure.

This implies a different style of modeling than is customary in mathematical programming, which writes all constraints using a few primitive terms (equations, inequalities, and some algebraic expressions). Effective integrated modeling draws from a large library of metaconstraints and presupposes that the user has some familiarity with this library. For instance, the library may contain a constraint that defines piecewise linear costs, a constraint that requires flow balance in a network, a constraint that prevents scheduled jobs from overlapping, and so forth. Each constraint is written with parameters that specify the shape of the function, the structure of the network, or the processing times of the jobs. When sitting down to formulate a model, the user would browse the library for constraints that appear to relate to the problem at hand.

Integrated modeling therefore places on the user the burden of identifying problem structure, but in so doing it takes full advantage of the human capacity for pattern recognition. Users identify highly structured subsets of constraints, which allows the solver to apply the best known analysis of these structures to solve the problem efficiently. In addition, only certain metaconstraints tend to occur in a given problem domain. This means that only a relevant portion of the library must be presented to a practitioner in that domain.

4. The Basic Algorithm

The search-infer-and-relax algorithm can be summarized as follows:

Search. The search proceeds by solving problem *restrictions*, each of which is obtained by adding constraints to the problem. The motivation is that restrictions may be easier to solve than the original problem. For example, a branch-and-bound method for MILP enumerates restrictions that correspond to nodes of the search tree. Branch-and-infer methods in CP do the same. Benders decomposition enumerates restrictions in the form of subproblems (slave problems). If the search is exhaustive, the best feasible solution of a restriction is optimal in the original problem. A search is exhaustive when the restrictions have feasible sets whose union is the feasible set of the original problem.

Infer. Very often the search can be accelerated by inferring valid constraints from the current problem restriction, which are added to the constraint set. MILP methods infer constraints in the form of cutting planes. CP methods infer smaller variable domains from individual constraints. (A variable's domain is the set of values it can take.) Classical Benders methods infer valid cuts from the subproblem by solving its dual.

One can often exploit problem structure by designing specialized inference methods for certain metaconstraints or highly structured subsets of constraints. Thus MILP generates specialized cutting planes for certain types of inequality sets (e.g. flow cuts). CP applies specialized domain reduction or "filtering" algorithms to such commonly used global constraints as *all-different*, *element* and *cumulative*. Benders cuts commonly exploit the structure of the subproblem.

Inference methods that are applied only to subsets of constraints often miss implications of the entire constraint set, but this can be partially remedied by *constraint propagation*, a fundamental technique of CP. For example, domains that are reduced by a filtering algorithm for one constraint can serve as the starting point for the domain reduction algorithm applied to the next constraint, and so forth. Thus the results of processing one constraint are "propagated" to the next constraint.

Relax. It is often useful to solve a relaxation of the current problem restriction, particularly when the restriction is too hard to solve. The relaxation can provide a bound on the optimal value, perhaps a solution that happens to be feasible in the original problem, and guidance for generating the next problem restriction.

In MILP, one typically solves linear programming or Lagrangian relaxations to obtain bounds or solutions that may happen to be integer. The solution of the relaxation also helps to direct the search, as for instance when one branches on a fractional variable. In Benders decomposition, the master problem is the relaxation. Its solution provides a bound on the optimum and determines the next restriction (subproblem) to be solved.

Like inference, relaxation is very useful for exploiting problem structure. For example, if the model identifies a highly structured subset of inequality constraints (by treating them as a single metaconstraint), the solver can generate a linear relaxation for them that contains specialized cutting planes. This allows one to exploit structure that is missed even by current MILP solvers. In addition, such global constraints as *all-different*, *element* and *cumulative* can be given specialized linear relaxations. See Williams and Yan (2001), Hooker (2000) and Hooker and Yan (2002), respectively.

5. The Syntax and Semantics of SIMPL

An optimization model in SIMPL is comprised of four main parts: declarations of constants and problem data, the objective function, declaration of metaconstraints and search specification. The first two parts are straightforward in the sense that they look very much like their counterparts in other modeling languages such as AMPL (Fourer, Gay, and Kernighan 2003), Mosel (Colombani and Heipcke 2002, 2004), and OPL (Van Hentenryck et al. 1999). Hence, because of space restrictions, this section concentrates on explaining the syntax of the metaconstraint and search parts, as well as the semantics of the model. Complete model examples can be found in Section 10.

5.1. Multiple Problem Relaxations

Each iteration in the solution of an optimization problem P examines a *restriction* N of P. In a tree search, for example, N is the problem restriction at the current node of the tree. Since solving N can be hard, we usually solve a *relaxation* N_R of N, or possibly several relaxations.

In an integrated CP-MILP modeling system, linear constraints are posted to a linear programming (LP) solver, normally along with linear relaxations of some of the other constraints. Constraints suitable for CP processing, perhaps including some linear constraints, are posted to a CP solver as well. Thus each solver deals with a relaxation of the original problem P. In this example, each problem restriction N has only an LP and a CP relaxation, but extending this idea other kinds of relaxations is straightforward.

5.2. Constraints and Constraint Relaxations

In SIMPL, the actual representation of a constraint of the problem inside any given relaxation is called a *constraint relaxation*. Every constraint can be associated with a list of constraint relaxation objects, which specify the relaxations of that constraint that will be used in the solution of the problem under consideration. To *post* a constraint means to add its constraint relaxations to all the appropriate problem relaxations. For example, both the LP and the CP relaxations of a linear constraint are equal to the constraint itself. The CP relaxation of the *element* constraint is clearly equal to itself, but its LP relaxation can be the convex hull formulation of its set of feasible solutions (Hooker 2000).

In branch-and-bound search, problem relaxations are solved at each node of the enumeration tree. In principle the relaxations could be generated from scratch at every node, since they are a function of the variable domains. Nonetheless it is more efficient to regenerate only relaxations that change significantly at each node. We therefore distinguish *static* constraint relaxations, which change very little (in structure) when the domains of its variables change (e.g. relaxations of linear constraints are equal to themselves, perhaps with some variables removed due to fixing); and *volatile* relaxations, which change radically when variable domains change (e.g. linear relaxations of global constraints). When updating the relaxations at a new node in the search tree, only the volatile constraint relaxations are regenerated. Regeneration is never necessary to create valid relaxations, but it strengthens the relaxation bounds.

A metaconstraint in SIMPL receives a declaration of the form

```
<name> means {
    <constraint-list>
    relaxation = { <relaxation-list> }
    inference = { <inference-list> }
}
```

where the underlined words are reserved words of the modeling language. `<name>` is an arbitrary name given to the metaconstraint, whose purpose will be explained in Section 5.3. `<constraint-list>` is a list of one or more constraints that constitute the metaconstraint. For example, this list could be a single global or linear constraint, a collection of linear constraints, or a collection of logical propositions. Finally, there are two other statements: `relaxation`, which is mandatory, and `inference`, which is optional. The `<relaxation-list>` contains a list of problem relaxations to which the constraint relaxations of `<constraint-list>` should be posted. An example of such list is {`lp, cp`}, which indicates a linear programming relaxation and a constraint programming relaxation. The `<inference-list>` contains a list of types of inference that `<constraint-list>` should perform. For example, if `<constraint-list>` is a global constraint such as *cumulative*, we could invoke a particular inference (or filtering) algorithm for that constraint by writing `inference =` {`edge-finding3`}, which indicates an $O(n^3)$ edge-finding filtering algorithm (Baptiste, Le Pape and Nuijten 2001). In the absence of the `inference` statement, either a default inference mechanism will be used or no inference will be performed. The actual behavior will depend on the types of constraints listed in `<constraint-list>`.

5.3. Search

The last part of the model, following the constraints, specifies the search method. We first explain how SIMPL internally implements search and then show how to control the search in a high-level language.

The main search loop inside SIMPL is implemented as shown below.

```
procedure Search($A$)
    If $A \neq \emptyset$ and stopping criteria not met
        $N$ := $A$.getNextNode()
        $N$.explore()
        $A$.addNodes($N$.generateRestrictions())
        Search($A$)
```

Here, N is again the current problem restriction, and A is the current list of restrictions waiting to be processed. Depending on how A, N and their subroutines are defined, we can have different types of search, as mentioned in Section 1. The routine N`.explore()` implements the infer-relax sequence. The routine N`.generateRestrictions()` creates new restrictions, and A`.addNodes()` adds them to A. Routine A`.getNextNode()` implements a mechanism for selecting the next restriction, such as depth-first, breadth-first or best bound.

In tree search, N is the problem restriction that corresponds to the current node, and A is the set of open nodes. In local search, N is the restriction that defines the current neighborhood, and A is the singleton containing the restriction that defines the next neighborhood to be searched. In Benders decomposition, N is the current subproblem and A is the singleton containing the next subproblem to be solved. In the case of Benders, the role of N`.explore()` is to infer Benders cuts from the current subproblem, add them to the master problem, and solve the master problem. N`.generateRestrictions()` uses the solution of the master problem to create the next subproblem. In the sequel, we restrict our attention to branch-and-bound search.

5.3.1. Node Exploration. The behavior of N`.explore()` for a branch-and-bound type of search is

```
1. Pre-relaxation inference
2. Repeat
3.     Solve relaxations
4.     Post-relaxation inference
5. Until (no changes) or (iteration limit)
```

The inference procedures in Steps 1 and 4 extract information from each relaxation in order to accelerate the search, as explained in Section 5.4 below. The loop continues to execute as desired until the domains reach a fixed point.

5.3.2. Branching. SIMPL implements a tree search by branching on constraints. This scheme is considerably more powerful and generic than branching on variables alone. If branching is needed, it is because some constraint of the problem is violated, and that constraint should “know” how to branch as a result. This knowledge is embedded in the *branching module* associated with the constraint. For example, if a variable $x \in \{0,1\}$ has a fractional value in the current LP, its indomain constraint I_x is violated. The branching module of I_x outputs the constraints $x \in \{0\}$ and $x \in \{1\}$, meaning that two subproblems should be created by the inclusion of those two new constraints. Traditional branching on a variable x can therefore be interpreted as a special case of branching on a constraint. In general, a branching module returns a sequence of constraint *sets* $C_1, \ldots, C_k$.

Each C_i defines a subproblem at a successor node when it is merged with the current problem. There is no restriction on the type of constraints appearing in C_i.

Clearly, there may be more than one constraint violated by the solution of the current set of problem relaxations. A *selection module* is the entity responsible for selecting, from a given set of constraints, the one on which to branch next. Some possible criteria for selection are picking the first constraint found to be violated or the one with the largest degree of violation.

5.3.3. Specifying the Search Syntactically, the search section of a SIMPL model is declared as follows.

```
SEARCH
    type = { <search-type> }
    branching = { <branching-list> }
    inference = { <general-inference-list> }
```

where the underlined words are reserved words of the modeling language. `<search-type>` indicates the type of search to be performed. For example, `bb` means branch-and-bound; `benders` means Benders decomposition; `bp` means branch-and-price; `ls` means local search, and so forth. Sometimes, an argument can be added to the search type to specify a particular node selection strategy. For instance, `bb:bestbound` means we want to do branch-and-bound with best-bound node selection.

The `<branching-list>` is a comma-separated list of terms where each term assumes the following form: `<name>:<selection-module>:<branching-module>`. `<name>` is the name of a metaconstraint, as described in Section 5.2. `<selection-module>` and `<branching-module>` represent, respectively, the selection module and branching module to be used when branching on the constraint named `<name>`. Constraints are checked for violation (for branching purposes) in the order they appear in `<branching-list>`, from left to right. When `<constraint-list>` in the metaconstraint `<name>` contains more than one constraint, `<selection-module>` will specify the order in which to check those constraints for violation. Current possibilities are `most` (most violated first), `least` (least violated first) and `first` (first violation in the list). Once a violated constraint c is selected, the optional argument `branching-module` specifies the way to branch on c. For example, let us say our model has two metaconstraints named `c1` and `c2`. The branching statement could look like `branching = { c1:most, c2:first:sos1 }`. This means we first check the constraints in `c1` for violation and branch on the most violated of those, if any, according to some default criterion. If the solution of the current relaxation satisfies `c1`, we scan the constraints in `c2` for violation in the order they are listed (because of `first`). If one is found violated, we use `sos1` branching. In case a variable name is used in the branching list instead of a metaconstraint name, this means we want to use the *indomain* constraints of that set of variables for branching. The indomain constraint of a variable is the constraint that specifies the possible values of that variable in the variable declaration section. Examples of such constraints are: `x[1..10] in {0, 1}`, for binary variables, and `y[1..5] in [0..10, 12..15]` for real-valued variables with holes in their domains.

The `<general-inference-list>` is an optional list of inferences that should be performed in addition to the inferences specified inside each individual metaconstraint (see Section 5.2). If, for instance, we want to generate lift-and-project cuts (Balas, Ceria and Cornuéjols 1993) (recall that cutting planes are a form of inference) and also perform reduced-cost fixing on our `x` variables, the inference statement in the search section of the model would look like `inference = { lift-and-project, x:redcost }`.

This ends the syntactic description of a SIMPL model. The next section gives more semantic details related to different kinds of inference.

5.4. Inference

We now take a closer look at the inference steps of the node exploration loop in Section 5.3.1. In step 1 (pre-relaxation inference), one may have domain reductions or the generation of new implied constraints (Hooker and Osorio 1999), which may have been triggered by the latest branching decisions. If the model includes a set of propositional logic formulas, this step can also execute some form of resolution algorithm to infer new resolvents. In step 4 (post-relaxation inference), other types of inference may take place, such as fixing variables by reduced cost or the generation of cutting planes. After that, it is possible to implement some kind of primal heuristic or to try extending the current solution to a feasible solution in a more formal way, as advocated in Sect. 9.1.3 of Hooker (2000).

Since post-relaxation domain reductions are associated with particular relaxations, the reduced domains that result are likely to differ across relaxations. Therefore, at the end of the inference steps, a synchronization step must be executed to propagate domain reductions across different relaxations. This is done in the algorithm below.

```
1. $V := \emptyset$
2. For each problem relaxation r
3.     $V_r :=$ variables with changed domains in $r$
4.     $V := V \cup V_r$
5.     For each $v \in V_r$
6.         $D_v := D_v \cap D_v^r$
7. For each $v \in V$
8.     Post constraint $v \in D_v$
```

In step 6, D_v^r denotes the domain of v inside relaxation r, and D_v works as a temporary domain for variable v, where changes are centralized. The initial value of D_v is the current domain of variable v. By implementing the changes in the domains via the addition of indomain constraints (step 8), those changes will be transparently undone when the search moves to a different part of the enumeration tree. Similarly, those changes are guaranteed to be redone if the search returns to descendants of the current node at a later stage.

6. Example: Piecewise Linear Functions

A simple production planning example with piecewise linear functions illustrates integrated modeling as well as the search-infer-and-relax process. The approach taken here is similar to that of Refalo (1999) and Ottosson, Thorsteinsson and Hooker (1999, 2002).

The objective is to manufacture several products at a plant of limited capacity C so as to maximize net income. Each product must be manufactured in one of several production modes (small scale, medium scale, large scale, etc.), and only a certain range of production quantities are possible for each mode. Thus if x_i units of product i are manufactured in mode k, $x_i \in [L_{ik}, U_{ik}]$. The net income $f_i(x_i)$ derived from making product i is linear in each interval $[L_{ik}, U_{ik}]$, with $f_i(L_{ik}) = c_{ik}$ and $f_i(U_{ik}) = d_{ik}$. When the $[L_{ik}, U_{ik}]$ intervals are disjunct, this means that f_i is a semi-continuous piecewise linear function (Fig. 1):

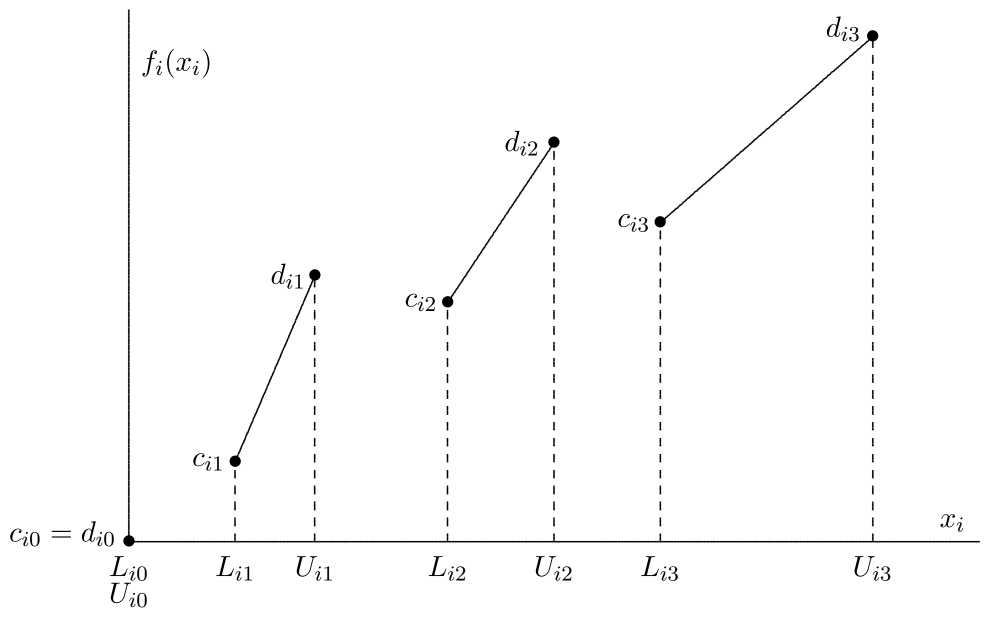
$$f_i(x_i) = \frac{U_{ik} - x_i}{U_{ik} - L_{ik}} c_{ik} + \frac{x_i - L_{ik}}{U_{ik} - L_{ik}} d_{ik}, \text{ if } x_i \in [L_{ik}, U_{ik}] \tag{1}$$

Making none of product i corresponds to mode $k = 0$, for which $[L_{i0}, U_{i0}] = [0, 0]$.

An integer programming model for this problem introduces 0-1 variables y_{ik} to indicate the production mode of each product. The functions f_i are modeled by assigning weights λ_{ij}, μ_{ik} to the endpoints of each interval k. The model is

$$\begin{aligned}
&\max \sum_{ik} \lambda_{ik} c_{ik} + \mu_{ik} d_{ik} \\
&\sum_i x_i \le C \\
&x_i = \sum_k \lambda_{ik} L_{ik} + \mu_{ik} U_{ik}, \text{ all } i \\
&\sum_k \lambda_{ik} + \mu_{ik} = 1, \text{ all } i \\
&0 \le \lambda_{ik} \le y_{ik}, \text{ all } i, k \\
&0 \le \mu_{ik} \le y_{ik}, \text{ all } i, k \\
&\sum_k y_{ik} = 1, \text{ all } i \\
&y_{ik} \in \{0, 1\}, \text{ all } i, k
\end{aligned} \tag{2}$$

Figure 1 A semi-continuous piecewise linear function $f_i(x_i)$.



If desired one can identify $\lambda_{i0}, \mu_{i0}, \lambda_{i1}, \mu_{i1}, \ldots$ as a specially ordered set of type 2 for each product i. However, specially ordered sets are not particularly relevant here, since the adjacent pair of variables $\mu_{ik}, \lambda_{i,k+1}$ are never both positive for any k.

The most direct way to write an integrated model for this problem is to use conditional constraints of the form $A \Rightarrow B$, which means that if the antecedent A is true, then the consequent B is enforced. The piecewise linear cost function $f_i(x_i)$ can be coded much as it appears in (1):

$$\begin{aligned}
&\max \sum_i u_i \\
&\sum_i x_i \le C && (a) \\
&(x_i \in [L_{ik}, U_{ik}]) \Rightarrow \left(u_i = \frac{U_{ik} - x_i}{U_{ik} - L_{ik}} c_{ik} + \frac{x_i - L_{ik}}{U_{ik} - L_{ik}} d_{ik} \right), \text{ all } i, k && (b) \\
&x_i \in \bigcup_k [L_{ik}, U_{ik}], \text{ all } i && (c)
\end{aligned} \tag{3}$$

Note that no discrete variables are required, and the model is quite simple.

The problem can be solved by branching on the continuous variables x_i in a branch-and-bound method. In this case the domain of x_i is an interval of real numbers. The search branches on an x_i by splitting its domain into two or more smaller intervals, much as is done in continuous global solvers, so that the domains become smaller as one descends into the search tree. The solver processes the conditional constraints (b) by adding the consequent to the constraint set whenever the current domain of x_i lies totally within $[L_{ik}, U_{ik}]$.

Although model (3) is a perfectly legitimate approach to solving the problem, it does not fully exploit the problem's structure. For each product i, the point (x_i, u_i) must lie on one of the line segments defined by consequents of (b). If the solver were aware of this fact, it could construct a tight linear relaxation by requiring (x_i, u_i) to lie in the convex hull of these line segments, thus resulting in substantially faster solution.

This is accomplished by equipping the modeling language with metaconstraint *piecewise* to model continuous or semi-continuous piecewise linear functions. A single piecewise constraint represents the constraints in (b) that correspond to a given i. The model (3) becomes

$$\begin{aligned}
&\max \sum_i u_i \\
&\sum_i x_i \le C && (a) \\
&\text{piecewise}(x_i, u_i, L_i, U_i, c_i, d_i),\ \text{all } i && (b)
\end{aligned} \tag{4}$$

Here L_i is an array containing $L_{i0}, L_{i1}, \ldots$, and similarly for U_i, c_i, and d_i. Each piecewise constraint enforces $u_i = f_i(x_i)$.

In general the solver has a library of metaconstraints that are appropriate to common modeling situations. Typically some constraints are written individually, as is (a) above, while others are collected under one or more metaconstraints in order simplify the model and allow the solver to exploit problem structure. It does so by applying inference methods to each metaconstraint, relaxing it, and branching in an intelligent way when it is not satisfied. In each case the solver exploits the peculiar structure of the constraint.

Let us suppose the solver is instructed to solve the production planning problem by branch and bound, which defines the *search* component of the algorithm. The search proceeds by enumerating restrictions of the problem, each one corresponding to a node of the search tree. At each node, the solver *infers* a domain $[a_i, b_i]$ for each variable x_i. Finally, the solver generates bounds for the branch-and-bound mechanism by solving a *relaxation* of the problem. It branches whenever a constraint is violated by the solution of the relaxation, and the nature of the branching is dictated by the constraint that is violated. If more than one constraint turns out to be violated, the solver will branch on the one with the highest priority, as specified by the user (see Section 10.1 for an example).

It is useful to examine these steps in more detail. At a given node of the search tree, the solver first applies inference methods to each constraint. Constraint (a) triggers a simple form of *interval propagation.* The upper bound b_i of each x_i's domain is adjusted to become $\min\left\{b_i, C - \sum_{j \neq i} a_j\right\}$. Constraint (b) can also reduce the domain of x_i, as will be seen shortly. Domains reduced by one constraint can be cycled back through the other constraint for possible further reduction. As branching and propagation reduce the domains, the problem relaxation becomes progressively tighter until it is infeasible or its solution is feasible in the original problem.

The solver creates a relaxation at each node of the search tree by pooling relaxations of the various constraints. It relaxes each constraint in (b) by generating linear inequalities to describe the convex hull of the graph of each f_i, as illustrated in Fig. 2. The fact that x_i is restricted to $[a_i, b_i]$ permits a tighter relaxation, as shown in the figure. Similar reasoning reduces the domain $[a_i, b_i]$ of x_i to $[L_{i1}, b_i]$. The linear constraint (a) also generates a linear relaxation, namely itself. These relaxations, along with the domains, combine to form a linear relaxation of the entire problem:

$$\begin{aligned}
&\max \sum_i u_i \\
&\sum_i x_i \le C \\
&\text{conv}(\text{piecewise}(x_i, u_i, L_i, U_i, c_i, d_i)), \text{ all } i \\
&a_i \le x_i \le b_i, \text{ all } i
\end{aligned} \tag{5}$$

where *conv* denotes the convex hull description just mentioned.

The solver next finds an optimal solution $(\bar{x}_i, \bar{u}_i)$ of the relaxation (5) by calling a linear programming plug-in. This solution will necessarily satisfy (a), but it may violate (b) for some product i, for instance if $\bar{x}_i$ is not a permissible value of x_i, or $\bar{u}_i$ is not the correct value of $f_i(\bar{x}_i)$. The latter case is illustrated in Fig. 2, where the search creates three branches by splitting the domain of x_i into three parts: $[L_{i2}, U_{i2}]$, everything below U_{i1}, and everything above L_{i3}. Note that in this instance the linear relaxation at all three branches will be exact, so that no further branching will be necessary.

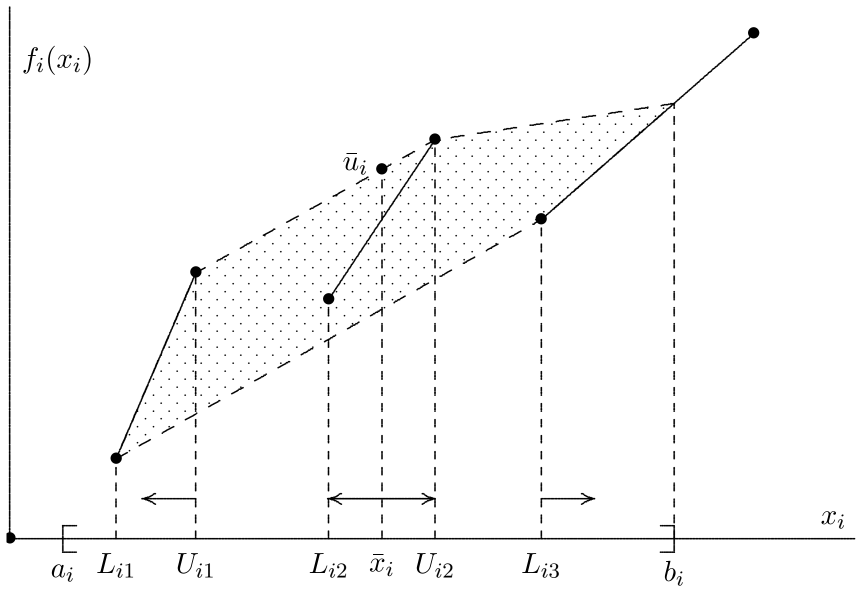
The problem is therefore solved by combining ideas from three technologies: search by splitting intervals, from continuous global optimization; domain reduction, from constraint programming; and polyhedral relaxation, from integer programming.

7. Example: Variable Indices

A variable index is a versatile modeling device that is readily accommodated by a search-infer-and-relax solver. If an expression has the form u_y, where y is a variable, then y is a *variable index* or variable subscript. A simple product configuration problem (Thorsteinsson and Ottosson, 2001) illustrates how variable indices can be used in a model and processed by a solver.

The problem is to choose an optimal configuration of components for a product, such as a personal computer. For each component i, perhaps a memory chip or power supply, one must decide

Figure 2 Convex hull relaxation (shaded area) of $f_i(x_i)$ when x_i has domain $[a_i, b_i]$.



how many q_i to install and what type t_i to install. Only one type of each component may be used. The types correspond to different technical specifications, and each type k of component i supplies a certain amount a_{ijk} of attribute j. For instance, a given type of memory chip might supply a certain amount of memory, generate a certain amount of heat, and consume a certain amount of power; in the last case, $a_{ijk} < 0$ to represent a negative supply. There are lower and upper bounds L_j, U_j on each attribute j. Thus there may be a lower bound on total memory, an upper bound on heat generation, a lower bound of zero on net power supply, and so forth. Each unit of attribute j produced incurs a (possibly negative) penalty c_j.

A straightforward integer programming model introduces 0-1 variables x_{ik} to indicate when type k of component i is chosen. The total penalty is $\sum_j c_j v_j$, where v_j is the amount of attribute j produced. The quantity v_j is equal to $\sum_{ik} a_{ijk} q_i x_{ik}$. Since this is a nonlinear expression, the variables q_i are disaggregated, so that q_{ik} becomes the number of units of type k of component i. The quantity v_j is now given by the linear expression $\sum_{ik} a_{ijk} q_{ik}$. A big-M constraint can be used to force q_{ij} to zero when $x_{ij} = 0$. The model becomes,

$$
\begin{aligned}
&\min \sum_j c_j v_j \\
&v_j = \sum_{ik} a_{ijk} q_{ik}, \text{ all } j \quad (a) \\
&L_j \le v_j \le U_j, \text{ all } j \quad (b) \\
&q_{ik} \le M_i x_{ik}, \text{ all } i,k \quad (c) \\
&\sum_k x_{ik} = 1, \text{ all } i \quad (d)
\end{aligned}
\tag{6}
$$

where each x_{ij} is a 0-1 variable, each q_{ij} is integer, and M_i is an upper bound on q_i. If the MILP modeling language accommodates specially ordered sets of non-binary variables, the variables x_{ij} can be eliminated and constraints (c) and (d) replaced by a stipulation that $\{q_{i1}, q_{i2}, \ldots\}$ is a specially ordered set of type 1 for each i.

An integrated model uses the original notation t_i for the type of component i, without the need for 0-1 variables or disaggregation. The key is to permit t_i to appear as a subscript:

$$\begin{aligned} &\min \sum_j c_j v_j \\ &v_j = \sum_i q_i a_{ijt_i}, \text{ all } j \end{aligned} \tag{7}$$

where the bounds L_j, U_j are reflected in the initial domain assigned to v_j.

The modeling system automatically decodes variably indexed expressions with the help of the *element* constraint, which is frequently used in CP modeling. In this case the variably indexed expressions occur in *indexed linear* expressions of the form

$$\sum_i q_i a_{ijt_i} \tag{8}$$

where each q_i is an integer variable and each t_i a discrete variable. Each term $q_i a_{ijt_i}$ is automatically replaced with a new variable z_{ij} and the constraint

$$\text{element}(t_i, (q_i a_{ij1}, \ldots, q_i a_{ijn}), z_{ij}) \tag{9}$$

This constraint in effect forces $z_{ij} = q_i a_{ijt_i}$. The solver can now apply a domain reduction or “filtering” algorithm to (9) and generate a relaxation for it.

For a given j, filtering for (9) is straightforward. If z_{ij}'s domain is $[\underline{z}_{ij}, \overline{z}_{ij}]$, t_i's domain is D_{t_i}, and q_i's domain is $\{\underline{q}_i, \underline{q}_i + 1, \ldots, \overline{q}_i\}$ at any point in the search, then the reduced domains $[\underline{z}'_{ij}, \overline{z}'_{ij}]$, D'_{t_i}, and $\{\underline{q}'_i, \ldots, \overline{q}'_i\}$ are given by

$$\begin{aligned} &\underline{z}'_{ij} = \max\left\{\underline{z}_{ij}, \min_k \left\{a_{ijk}\underline{q}_i, a_{ijk}\overline{q}_i\right\}\right\}, \ \overline{z}'_{ij} = \min\left\{\overline{z}_{ij}, \max_k \left\{a_{ijk}\underline{q}_i, a_{ijk}\overline{q}_i\right\}\right\}, \\ &D'_{t_i} = D_{t_i} \cap \left\{k \;\middle|\; [\underline{z}'_{ij}, \overline{z}'_{ij}] \cap \left[\min\{a_{ijk}\underline{q}_i, a_{ijk}\overline{q}_i\}, \max\{a_{ijk}\underline{q}_i, a_{ijk}\overline{q}_i\}\right] \neq \emptyset\right\} \\ &\underline{q}'_i = \max\left\{\underline{q}_i, \min_k \left\{\left\lceil \frac{\underline{z}'_{ij}}{a_{ijk}} \right\rceil, \left\lceil \frac{\overline{z}'_{ij}}{a_{ijk}} \right\rceil\right\}\right\}, \ \overline{q}'_i = \min\left\{\overline{q}_i, \max_k \left\{\left\lfloor \frac{\underline{z}'_{ij}}{a_{ijk}} \right\rfloor, \left\lfloor \frac{\overline{z}'_{ij}}{a_{ijk}} \right\rfloor\right\}\right\} \end{aligned}$$

Since (9) implies a disjunction $\bigvee_{k \in D_{t_i}} (z_{ij} = a_{ijk} q_i)$, it can be given the standard convex hull relaxation for a disjunction which in this case simplifies to

$$z_{ij} = \sum_{k \in D_{t_i}} a_{ijk} q_{ik}, \ \ q_i = \sum_{k \in D_{t_i}} q_{ik} \tag{10}$$

where $q_{ik} \geq 0$ are new variables.

If there is a lower bound L on the expression (8), the relaxation used by Thorsteinsson and Ottosson (2001) can be strengthened with integer knapsack cuts (and similarly if there is an upper bound). Since

$$\sum_i q_i a_{ijt_i} = \sum_i \sum_{k \in D_{t_i}} a_{ijk} q_{ik} \leq \sum_i \max_{k \in D_{t_i}} \{a_{ijk}\} \sum_{k \in D_{t_i}} q_{ik} = \sum_i \max_{k \in D_{t_i}} \{a_{ijk}\} q_i$$

the lower bound L on (8) yields the valid inequality

$$\sum_i \max_{k \in D_{t_i}} \{a_{ijk}\} q_i \geq L \tag{11}$$

Since the q_is are general integer variables, integer knapsack cuts can be generated for (11).

Based on these ideas, the automatically generated relaxation of (7) becomes

$$\begin{aligned}
&\min \sum_j c_j v_j \\
&v_j = \sum_i \sum_{k \in D_{t_i}} a_{ijk} q_{ik}, \text{ all } j \\
&q_i = \sum_{k \in D_{t_i}} q_{ik}, \text{ all } i \\
&L_j \leq v_j \leq U_j, \text{ all } j \\
&\underline{q}_i \leq q_i \leq \overline{q}_i, \text{ all } i \\
&\text{knapsack cuts for } \sum_i \max_{k \in D_{t_i}} \{a_{ijk}\} q_i \geq L_j, \text{ all } j \\
&\text{knapsack cuts for } \sum_i \min_{k \in D_{t_i}} \{a_{ijk}\} q_i \leq U_j, \text{ all } j \\
&q_{ik} \geq 0, \text{ all } i, k
\end{aligned} \tag{12}$$

There is also an opportunity for *post-relaxation inference*, which in this case takes the form of reduced cost variable fixing. Suppose the best feasible solution found so far has value z^*, and let $\hat{z}$ be the optimal value of (12). If $\hat{z} + r_{ik} \geq z^*$, where r_{ik} is the reduced cost of q_{ik} in the solution of (12), then k can be removed from the domain of t_i. In addition, if $r_{ik} > 0$, one can infer

$$\overline{q}_i \leq \max_{k \in D_{t_i}} \left\{ \left\lfloor \frac{z^* - \hat{z}}{r_{ik}} \right\rfloor \right\}, \text{ all } i$$

Post-relaxation inference can take other forms as well, such as the generation of separating cuts.

The problem can be solved by branch and bound. In this case, we can start by branching on the domain constraints $t_i \in D_{t_i}$. Since t_i does not appear in the linear relaxation, it does not have a determinate value until it is fixed by branching. The domain constraint $t_i \in D_{t_i}$ is viewed as unsatisfied as long as t_i is undetermined. The search branches on $t_i \in D_{t_i}$ by splitting D_{t_i} into two subsets. Branching continues until all the D_{t_i} are singletons, or until at most one q_{ik} (for $k \in D_{t_i}$) is positive for each i. At that point we check if all q_i variables are integer and branch on q if necessary.

8. Example: Logic-based Benders Decomposition

Nogood-based methods search the solution space by generating a *nogood* each time a candidate solution is examined. The nogood is a constraint that excludes the solution just examined, and perhaps other solutions that can be no better. The next solution enumerated must satisfy the nogoods generated so far. The search is exhaustive when the nogood set becomes infeasible.

Benders decomposition is a special type of nogood-based method in which the nogoods are Benders cuts and the master problem contains all the nogoods generated so far. In classical Benders, the subproblem (slave problem) is a linear or nonlinear programming problem, and Benders cuts are obtained by solving its dual—or in the nonlinear case by deriving Lagrange multipliers. *Logic-based* Benders generalizes this idea to an arbitrary subproblem by solving the *inference dual* of the subproblem (Hooker and Yan 1995; Hooker 1996, 1999).

A simple planning and scheduling problem illustrates the basic idea (Hooker 2000; Jain and Grossmann 2001; Bockmayr and Pisaruk 2003). A set of n jobs must be assigned to machines, and the jobs assigned to each machine must be scheduled subject to time windows. Job j has release time r_j, deadline d_j, and processing time p_{ij} on machine i. It costs c_{ij} to process job j on machine i. It generally costs more to run a job on a faster machine. The objective is to minimize processing cost.

We now describe the MILP model used by Jain and Grossmann (2001). Let the binary variable x_{ij} be one if job j is assigned to machine i, and let the binary variable $y_{jj'}$ be one if both j and j' are assigned to the same machine and j finishes before j' starts. In addition, let t_j be the time at which job j starts. The MILP model is as follows.

$$\begin{array}{lr}
\min \sum_{ij} c_{ij} x_{ij} & \\
r_j \le t_j \le d_j - \sum_i p_{ij} x_{ij}, \text{ all } j & \text{(a)} \\
\sum_i x_{ij} = 1, \text{ all } j & \text{(b)} \\
y_{jj'} + y_{j'j} \le 1, \text{ all } j' > j & \text{(c)} \\
y_{jj'} + y_{j'j} + x_{ij} + x_{i'j'} \le 2, \text{ all } j' > j,\ i' \ne i & \text{(d)} \\
y_{jj'} + y_{j'j} \ge x_{ij} + x_{ij'} - 1, \text{ all } j' > j,\ i & \text{(e)} \\
t_{j'} \ge t_j + \sum_i p_{ij} x_{ij} - \mathcal{U}(1 - y_{jj'}), \text{ all } j' \ne j & \text{(f)} \\
\sum_j p_{ij} x_{ij} \le \max_j\{d_j\} - \min_j\{r_j\}, \text{ all } i & \text{(g)} \\
x_{ij} \in \{0,1\},\ \ y_{jj'} \in \{0,1\} \text{ for all } j' \ne j &
\end{array} \tag{13}$$

Constraint (a) defines lower and upper bounds on the start time of job j, and (b) makes sure every job is assigned to some machine. Constraints (c) and (d) are logical cuts which significantly reduce solution time. Constraint (e) defines a logical relationship between the assignment (x) and sequencing (y) variables, and (f) ensures start times are consistent with the value of the sequencing variables y ($\mathcal{U} = \sum_j \max_i\{p_{ij}\}$). Finally, (g) are valid cuts that tighten the linear relaxation of the problem. There is also a continuous-time MILP model suggested by Türkay and Grossmann (1996), but computational testing indicates that it is much harder to solve than (13) (Hooker 2004).

A hybrid model can be written with the *cumulative* metaconstraint, which is widely used in constraint programming for "cumulative" scheduling, in which several jobs can run simultaneously but subject to a resource constraint and time windows. Let t_j be the time at which job j starts processing and u_{ij} the rate at which job j consumes resources when it is running on machine i. The constraint

$$\text{cumulative}(t, p_i, u_i, U_i)$$

requires that the total rate at which resources are consumed on machine i be always less than or equal to U_i. Here $t = (t_1, \ldots, t_n)$, $p_i = (p_{i1}, \ldots, p_{in})$, and similarly for u_i.

In the present instance, jobs must run sequentially on each machine. Thus each job j consumes resources at the rate $u_{ij} = 1$, and the resource limit is $U_i = 1$. Thus if y_j is the machine assigned to job j, the problem can be written

$$\begin{aligned}
&\min \sum_j c_{y_j j} \\
&r_j \le t_j \le d_j - p_{y_j j}, \text{ all } j \\
&\text{cumulative}((t_j | y_j = i), (p_{ij} | y_j = i), e, 1), \text{ all } i
\end{aligned} \tag{14}$$

where e is a vector of ones.

This model is adequate for small problems, but solution can be dramatically accelerated by decomposing the problem into an assignment portion to be solved by MILP and a subproblem to be solved by CP. The assignment portion becomes the Benders master problem, which allocates job j to machine i when $x_{ij} = 1$:

$$\begin{aligned}
&\min \sum_{ij} c_{ij} x_{ij} \\
&\sum_i x_{ij} = 1, \text{ all } j \\
&\text{relaxation of subproblem} \\
&\text{Benders cuts} \\
&x_{ij} \in \{0, 1\}
\end{aligned} \tag{15}$$

The solution $\bar{x}$ of the master problem determines the assignment of jobs to machines. Once these assignments are made, the problem (14) separates into a scheduling feasibility problem on each machine i:

$$\begin{array}{l} r_j \leq t_j \leq d_j - p_{\bar{y}_j j}, \text{ all } j \\ \text{cumulative}((t_j|\bar{y}_j = i), (p_{ij}|\bar{y}_j = i), e, 1) \end{array} \tag{16}$$

where $\bar{y}_j = i$ when $\bar{x}_{ij} = 1$. If there is a feasible schedule for every machine, the problem is solved. If, however, the scheduling subproblem (16) is infeasible on some machine i, a Benders cut is generated to rule out the solution $\bar{x}$, perhaps along with other solutions that are known to be infeasible. The Benders cuts are added to the master problem, which is re-solved to obtain another assignment $\bar{x}$.

The simplest sort of Benders cut for machine i rules out assigning the same set of jobs to that machine again:

$$\sum_{j \in J_i} (1 - x_{ij}) \geq 1 \tag{17}$$

where $J_i = \{j \mid \bar{x}_{ij} = 1\}$. A stronger cut can be obtained, however, by deriving a smaller set J_i of jobs that are actually responsible for the infeasibility. This can be done by removing elements from J_i one at a time, and re-solving the subproblem, until the scheduling problem becomes feasible (Hooker 2005a). Another approach is to examine the proof of infeasibility in the subproblem and note which jobs actually play a role in the proof (Hooker 2005b). In CP, an infeasibility proof generally takes the form of edge finding techniques for domain reduction, perhaps along with branching. Such a proof of infeasibility can be regarded as a solution of the subproblem's *inference dual.* (In linear programming, the inference dual is the classical linear programming dual.) Logic-based Benders cuts can also be developed for planning and scheduling problems in which the subproblem is an optimization rather than a feasibility problem. This occurs, for instance, in minimum makespan and minimum tardiness problems (Hooker 2004).

It is computationally useful to strengthen the master problem with a relaxation of the subproblem. The simplest relaxation requires that the processing times of jobs assigned to machine i fit between the earliest release time and latest deadline:

$$\sum_j p_{ij} x_{ij} \leq \max_j \{d_j\} - \min_j \{r_j\} \tag{18}$$

A Benders method (as well as any nogood-based method) fits easily into the search-infer-and-relax framework. It solves a series of problem restrictions in the form of subproblems. The search is directed by the solution of a relaxation, which in this case is the master problem. The inference stage generates Benders cuts.

The decomposition is communicated to the solver by writing the model

$$\begin{array}{ll} \min \sum_{ij} c_{ij} x_{ij} & \text{(a)} \\ \sum_{i} x_{ij} = 1, \text{ all } j & \text{(b)} \\ (x_{ij} = 1) \Leftrightarrow (y_j = i)\,, \text{ all } i, j & \text{(c)} \\ r_j \le t_j \le d_j - p_{y_j j}, \text{ all } j & \text{(d)} \\ \text{cumulative}((t_j | y_j = i), (p_{ij} | \bar{y}_j = i), e, 1), \text{ all } i & \text{(e)} \end{array} \tag{19}$$

where the domain of each x_{ij} is $\{0,1\}$. Each constraint is associated with a relaxation parameter and an inference parameter. The relaxation parameters for the constraints (b) and (d) will indicate that these constraints contribute to the MILP master relaxation of the problem. Note that (d) and (e) are part of the Benders subproblem. The relaxation parameter for (e) will add the inequalities (18) to the linear relaxation. The inference parameter for (e) will specify the type of Benders cuts to be generated. When the solver is instructed to use a Benders method, it automatically adds the Benders cuts to the relaxation. For more details on how these parameters are stated, see Section 10.3.

9. Example: Global Optimization

To illustrate the solution of global optimization problems with SIMPL, we will model and solve a few bilinear problems taken from chapter 5 of Floudas et al. (1999). Bilinear problems are an important subclass of non-convex quadratic programming problems whose applications include pooling and blending, separation sequencing, heat exchanger network design, and multicommodity network flow problems.

We consider the following general formulation of a bilinear problem, which is given in Floudas et al. (1999):

$$\begin{array}{l} \min\;\; x^T A_0 y + c_0^T x + d_0^T y \\ x^T A_i y + c_i^T x + d_i^T y \le b_i, \;\; i = 1, \ldots, p \\ x^T A_i y + c_i^T x + d_i^T y = b_i, \;\; i = p+1, \ldots, p+q \\ x \in R^n, \;\; y \in R^m \end{array} \tag{20}$$

where x and y are n- and m-dimensional variable vectors, respectively.

Whenever SIMPL finds a nonlinear term of the form $x_i y_j$, it is replaced by a new variable z_{ij} and a metaconstraint bilinear(x_i,y_j,z_{ij}). This constraint will enforce the fact that $z_{ij} = x_i y_j$ and will also create, and automatically update, its relaxations. For instance, the bilinear constraint would be relaxed as itself into a non-linear solver and into a CP solver because both of these solvers can handle non-linear constraints directly. If posted to a linear relaxation, the bilinear constraint would be transformed into the following well known linear relaxation of the bilinear term x_iy_j:

$$\begin{aligned} L_jx_i + L_iy_j - L_iL_j \le z_{ij} \le L_jx_i + U_iy_j - U_iL_j \\ U_jx_i + U_iy_j - U_iU_j \le z_{ij} \le U_jx_i + L_iy_j - L_iU_j \end{aligned} \tag{21}$$

where $[L_i, U_i]$ and $[L_j, U_j]$ are the current bounds on x_i and y_j, respectively. The search branches on bilinear(x_i,y_j,z_{ij}) whenever the values of x_i and y_j in the current relaxation do not match the value of z_{ij} when multiplied together. Branching will split the domains of x_i and/or y_j, depending on a number of conditions (Section 10.4 has more details). The general idea behind solving global optimization problems with other types of non-linearities is similar to this one: capture the non-linearity into a metaconstraint, relax it into a convex set, and branch on it. The relaxation of the bilinear metaconstraint happens to be linear in this case. In situations when the relaxation is a convex non-linear function, it would be posted to and handled by a non-linear solver. The remaining steps of the solution algorithm stay the same. One of the most successful global optimization solvers available today is BARON (Tawarmalani and Sahinidis 2004), which uses a *branch-and-reduce* algorithm. One can see the branch step of BARON as the search step of SIMPL, and the reduce step of BARON as the infer step of SIMPL. Both systems create convex relaxations of the model and use them to calculate dual bounds on the objective function. Hence, when it comes to global optimization, BARON and SIMPL follow very similar strategies.

10. Computational Experiments

In the next four subsections, we formulate integrated models for the four examples described in Sections 6 through 9. We solve them with SIMPL and compare its performance to the reported performance of previously implemented integrated methods. We also formulate and solve the corresponding MILP and global optimization models with CPLEX 11 and BARON, respectively.

We report both the number of search nodes and the computation time. Since SIMPL is still a research code, the node count may be a better indication of performance at the present stage of development. The amount of time SIMPL spends per node can be reduced by optimizing the code, whereas the node count is more a function of the underlying algorithm. Even though the focus of this paper is not to show that integrated approaches can sometimes outperform traditional optimization approaches, we note that SIMPL requires less time, as well as fewer nodes, than current technology on two of the four problem classes studied here.

Unless indicated otherwise, all the experiments reported in this section have been run on a Pentium 4, 3.7 GHz with 4GB of RAM, running Ubuntu 8.04 with Linux kernel 2.6.24-19. We used

CPLEX 11 as the LP solver and ECL^{i}PS^e 5.10.41 as the CP solver in SIMPL. For simplicity, when showing the SIMPL code of each model we omit the data and variable declaration statements.

10.1. Production Planning

The SIMPL code that corresponds to the integrated model (4) of the production planning problem is shown below.

```
01. OBJECTIVE
02.   maximize sum i of u[i]
03. CONSTRAINTS
04.   capacity means {
05.     sum i of x[i] <= C
06.     relaxation = { lp, cp } }
07.   piecewisectr means {
08.     piecewise(x[i],u[i],L[i],U[i],c[i],d[i]) forall i
09.     relaxation = { lp, cp } }
10. SEARCH
11.   type = { bb:bestdive }
12.   branching = { piecewisectr:most }
```

Lines 06 and 09 of the above code tell SIMPL that those constraints should be posted to both the linear programming (`lp`) and constraint programming (`cp`) relaxations/solvers. Recall that the linear programming relaxation of the i^{th} piecewise constraint is the collection of inequalities on x_i and u_i that define the convex hull of their feasible values in the current state of the search. Line 11 indicates that we use branch-and-bound (`bb`), select the current active node with the best lower bound, and dive from it until we reach a leaf node (keyword `bestdive`). Finally, in line 12 we say that the branching strategy is to branch on the piecewise constraint with the largest degree of violation (keyword `most`). The amount of violation is calculated by measuring how far the LP relaxation values of x_i and u_i are from the closest linear piece of the function. That is, we measure the rectilinear distance between the point (x_i, u_i) and the current convex hull of piecewise.

We ran the integrated model over 20 randomly generated instances with the number of products n ranging from 5 to 100. Similar problems were solved by Ottosson, Thorsteinsson and Hooker (1999, 2002), but their models had a few additional constraints and their piecewise linear functions were continuous. We did not use their original instances because we wanted to experiment with a more challenging version of the problem (including discontinuous functions and more products).

The performance of their implementation was comparable to the best MILP results at that time, whereas our implementation is superior to the MILP approach. In all instances, products have the same cost structure with five production modes. For the purpose of reducing symmetry, the model also includes constraints of the form $x_i \le x_{i+1}$ for all $i \in \{1, \dots, n-1\}$.

The number of search nodes and CPU time (in seconds) required to solve the instances to optimality are shown in Table 2. For reference, we also include results obtained with the pure MILP model (2) with the above symmetry breaking constraints. As the number of products increases, number of search nodes required by the pure MILP approach can be over an order of magnitude larger than the number of nodes required by the integrated approach. This problem used to be considerably more challenging for previous versions of CPLEX, as shown by the columns under "MILP (CPLEX 9)". If we change the LP solver in SIMPL from CPLEX 11 to CPLEX 9, the numbers under the SIMPL column in Table 2 do not change significantly.

10.2. Product Configuration

The product configuration problem of Section 7 can be coded in SIMPL as follows.

```
OBJECTIVE
  minimize sum j of c[j]*v[j]
CONSTRAINTS
  usage means {
    v[j] = sum i of q[i]*a[i][j][t[i]] forall j
    relaxation = { lp, cp }
    inference = { knapsack } }
  quantities means {
    q[1] >= 1 => q[2] = 0
    relaxation = { lp, cp } }
  types means {
    t[1] = 1 => t[2] in {1,2}
    t[3] = 1 => (t[4] in {1,3} and t[5] in {1,3,4,6} and t[6] = 3 and
                 t[7] in {1,2})
    relaxation = { lp, cp } }
SEARCH
  type = { bb:bestdive }
  branching = { quantities, t:most, q:least:triple, types:most }
  inference = { q:redcost }
```

For our computational experiments, we used the ten hardest problem instances proposed by Thorsteinsson and Ottosson (2001), which have 26 components, up to 30 types per component, and 8 attributes. According to Thorsteinsson and Ottosson (2001), these instances were generated to "closely resemble" real-world instances. In addition, there are a few extra logical constraints on the q_i and t_i variables, which are implemented in lines 08 through 14 above. These constraints are also added to the MILP model (6).

For the `usage` constraints, note the variable subscript notation in line 05 and the statement that tells it to infer integer knapsack cuts (as described in Section 7) in line 07 (this would be the default behavior anyway). We define our branching strategy in line 17 as follows: we first try to branch on the q-implication (`quantities`), then on the indomain constraints of the t variables (most violated first), followed by the indomain constraints on the q variables (least violated first), and finally on the implications on the t variables (`types`, most violated first). The indomain constraint of a t_i variable is violated if its domain is not a singleton and two or more of the corresponding q_{ik} variables have a positive value in the LP relaxation (see Section 7 for the relationship between t_i and q_{ik}). When the element constraint (9) is relaxed, the q_{ik} variables are created and variable q_i is marked as a variable that decomposes into those q_{ik} variables (see the convex hull relaxation (10) in Section 7). In addition, variable t_i is marked as the indexing variable responsible for that decomposition and it saves a pointer to q_i and the list of q_{ik} variables. Hence, when the indomain constraint of t_i is checked for violation as described above, it knows which q_{ik} variables to look at. The keyword `triple` that appears after `q:least` in line 17 indicates that we branch on q_i as suggested by Thorsteinsson and Ottosson (2001): let $\bar{q}_i$ be the closest integer to the fractional value of q_i in the current solution of the LP relaxation; we create up to three descendants of the current node by adding each of the following constraints in turn (if possible): $q_i = \bar{q}_i$, $q_i \leq \bar{q}_i - 1$ and $q_i \geq \bar{q}_i + 1$. Finally, the post-relaxation inference using reduced costs is turned on for the q variables in line 18.

The number of search nodes and CPU time (in seconds) required to solve each of the ten instances to optimality are shown in Table 3. This problem has become easy for recent MILP solvers, but Thorsteinsson and Ottosson (2001) original implementation was orders of magnitude better than a pure MILP approach, both in terms of number of nodes and time. SIMPL's search tree is, on average, about 10 times smaller than Thorsteinsson and Ottosson (2001) search tree on the same problem instances, but SIMPL's current implementation does not include strong cutting plane algorithms, which makes a difference at the root node for this problem. Even then, SIMPL's performance is surprisingly comparable to CPLEX's.

10.3. Parallel Machine Scheduling

We now describe the SIMPL code to implement the Benders decomposition approach of Section 8 (model (19)) when the master problem is given by (15), the subproblem is (16), and the Benders cuts are of the form (17).

```
01. OBJECTIVE
02.   min sum i,j of c[i][j] * x[i][j];
03. CONSTRAINTS
04.   assign means {
05.     sum i of x[i][j] = 1 forall j;
06.     relaxation = { ip:master } }
07.   xy means {
08.     x[i][j] = 1 <=> y[j] = i forall i, j;
09.     relaxation = { cp } }
10.   tbounds means {
11.     r[j] <= t[j] forall j;
12.     t[j] <= d[j] - p[y[j]][j] forall j;
13.     relaxation = { ip:master, cp } }
14.   machinecap means {
15.     cumulative({ t[j], p[i][j], 1 } forall j | x[i][j] = 1, 1) forall i;
16.     relaxation = { cp:subproblem, ip:master }
17.     inference = { feasibility } }
18. SEARCH
19.   type = { benders }
```

The keywords `ip:master` that appear in the relaxation statements in lines 06, 13 and 16 indicate that those constraints are to be relaxed into an Integer Programming relaxation, which will constitute the master problem. Constraints that have the word `cp` in their relaxation statements will be posted to a CP relaxation and are common to all Benders subproblems. This is true for the `xy` and `tbounds` constraints. For the `machinecap` constraints, the keywords `cp:subproblem` in line 16, together with the `forall i` statement in line 15, indicate that, for each i, there will be a different CP subproblem containing the corresponding cumulative constraint, in addition to the common constraints mentioned above. Finally, we tell SIMPL that the cumulative constraints should generate the feasibility-type Benders cuts (17) in line 17. Hence, when a subproblem turns out to be infeasible, its cumulative constraint, which is aware of the jobs it was assigned to handle, has all the information it needs to infer a cut that looks like (17).

For our computational experiments, we used the instances proposed by Jain and Grossmann (2001) and, additionally, we followed their procedure to create three new instances with more than 20 jobs. These are the last instance in Table 4 and the last two instances in Table 5. As was the case in Section 10.2, Jain and Grossmann (2001)'s instances were also generated to resemble real-world instances.

Jain and Grossmann's original implementation only needed a small number of master iterations and a few seconds to solve these instances. The results we obtain with SIMPL match this performance. Although state-of-the-art MILP solvers have considerably improved since Jain and Grossmann's results were published, the largest instances are still intractable with the pure MILP model (13). In addition to being orders of magnitude faster in solving the smallest problems, the integrated Benders approach can easily tackle larger instances as well. As noted by Jain and Grossmann (2001), when processing times are shorter the problem tends to become easier, and we report the results for shorter processing times in Table 5. Even in this case, the MILP model is still much worse than the integrated Benders approach as the problem size grows. For instance, after more than 27 million search nodes and a time limit of 48 hours of CPU time, the MILP solver had found an integer solution with value 156 to the problem with 22 jobs and 5 machines, whereas the optimal solution has value 155. As for the largest problem (25 jobs and 5 machines), the MILP solver ran out of memory after 19 hours of CPU time and over 5 million search nodes, having found a solution of value 182 (the optimal value is 179). It is worth mentioning that the MILP formulations of these two problems only have 594 and 750 variables, respectively.

10.4. Pooling, Distillation and Heat Exchanger Networks

To demonstrate SIMPL's ability to solve the bilinear global optimization problems, we selected the 6 bilinear problems from chapter 5 of Floudas et al. (1999) that BARON was able to solve in less than 24 hours. Problem names correspond to section numbers in that chapter. Problems 5.2.2.1, 5.2.2.2, 5.2.2.3, and 5.2.4 are pooling problems; problem 5.3.2 is a distillation problem; and problem 5.4.2 is a heat exchanger network problem. For illustration purposes, we describe a SIMPL model that could be used to model problem 5.2.2.1 (case 1 of section 5.2.2 in Floudas et al. 1999). Models for the other problems would be very similar to this one.

```
OBJECTIVE
    max 9*x + 15*y - 6*A - 16*B - 10*(Cx + Cy);
```

```
03. CONSTRAINTS
04.   flow means {
05.     Px + Py - A - B = 0;
06.     x - Px - Cx = 0;
07.     y - Py - Cy = 0;
08.     relaxation = { lp, cp } }
09.   pooling means {
10.     p*Px + 2*Cx - 2.5*x <= 0;
11.     p*Py + 2*Cy - 1.5*y <= 0;
12.     p*Px + p*Py - 3*A - B = 0;
13.     relaxation = { lp, cp } }
14. SEARCH
15.   type = { bb:bestdive }
16.   branching = { pooling:most }
17.   inference = { redcost }
```

If a non-linear programming solver is linked to SIMPL, line 13 could be changed to `relaxation = { lp, cp, nlp }`. Because the CP solver can handle non-linear constraints directly, they are posted to the CP relaxation as such, and ECLiPSe takes care of doing the proper bounds propagation (using BARON's terminology, this type of inference would be called *feasibility-based range reduction*). Internally, the constraint in line 10 would be transformed into `ZpPx + 2*Cx - 2.5*x <= 0` and `bilinear(p,Px,ZpPx)`; the constraint in line 11 would be transformed into `ZpPy + 2*Cy - 1.5*y <= 0` and `bilinear(p,Py,ZpPy)`; and the constraint in line 12 would be transformed into the linear constraint `ZpPx + ZpPy - 3*A - B = 0`, because its bilinear terms have appeared before and there is no need for additional `bilinear` constraints. Branching is done on the most violated of the bilinear metaconstraints, where the violation is measured by $|x_i y_j - z_{ij}|$. Line 17 tells SIMPL to perform domain reduction based on reduced costs for all variables (using BARON's terminology, this type of inference would be called *optimality-based range reduction*).

Computational results appear in Table 6. For comparison purposes, we solved these problems with both SIMPL and BARON version 7.2.5 (Tawarmalani and Sahinidis 2004). Because we ran BARON on another machine (an IBM workstation with two 3.2 GHz Intel Xeon processors and 2.5 GB of RAM), the times reported in the BARON column of Table 6 cannot be directly compared with the times reported for SIMPL. Even though BARON is a much more mature, stable and advanced global optimization solver than SIMPL currently is, the results of Table 6 help to demonstrate that SIMPL's framework can also accommodate global optimization problems. As noted in Section 9, what BARON does when solving a global optimization problem, known as *Branch and Reduce*, can be interpreted as a special case of SIMPL's search-infer-relax paradigm.

SIMPL's implementation of global optimization still has a lot of room for improvement. In addition to better memory management, these improvements include the use of more powerful inference mechanisms (e.g. using lagrangian multipliers), support for other types of non-linear constraints and, most importantly, strong pre-processing techniques and a local search mechanism that help find good solutions early in the search, such as what BARON does. The next step is to link a standalone non-linear solver to SIMPL, such as KNITRO or MINOS, which would increase the range of global optimization problems that it can solve.

11. Final Comments and Conclusions

In this paper we describe a general-purpose integrated solver for optimization problems, to which we refer by the acronym SIMPL. It incorporates the philosophy that many traditional optimization techniques can be seen as special cases of a more general method, one that iterates a three-step procedure: solving relaxations, performing logical inferences, and intelligently enumerating problem restrictions. The main objective of SIMPL is to make the computational and modeling advantages of integrated problem-solving conveniently available.

We tested SIMPL's modeling and solution capabilities on four types of optimization problems. We found that SIMPL (a) reproduces or exceeds the computational advantage of custom-coded integrated algorithms on three of these problems; (b) solves two of the problem classes faster than the current state of the art, one of them by orders of magnitude, even though it is still an experimental code; and (c) provides these advantages with modest effort on the part of the user, since the integrated models are written in a concise and natural way; (d) demonstrates that its modeling and solution framework can accommodate a wide range of problem types, from linear programming to global optimization.

One may argue that it is unfair to compare SIMPL with an off-the-shelf commercial solver, since the latter does not contain facilities to exploit problem structure in the way that SIMPL does. Yet a major advantage of an integrated solver is precisely that it can exploit structure while remaining a general-purpose solver and providing the convenience of current commercial systems. SIMPL's constraint-based approach automatically performs the tedious job of integrating solution techniques while exploiting the complementary strengths of the various technologies it combines.

Our examples suggest that a SIMPL user must be more aware of the solution algorithm than an MILP user, but again this allows the solver to benefit from the user's understanding of problem structure. We anticipate that future development of SIMPL and related systems will allow them to presuppose less knowledge on the part of the average user to solve less difficult problems, while giving experts the power to solve harder problems within the same modeling framework. In addition, we plan to increase SIMPL's functionality by increasing its library of metaconstraints, solver types, constraint relaxations, and search strategies, with the goal of accommodating the full spectrum of problems described in Table 1.

Those interested in reproducing our results can download a demo version of SIMPL from http://moya.bus.miami.edu/∼tallys/simpl.php. The package includes all the problem instances used in our experiments.

References

Achterberg, T. 2004. SCIP: A framework to integrate constraint and mixed integer programming, Konrad-Zuse-Zentrum für Informationstechnik Berlin, ZIB-Report 04-19.

Ajili, F., M. Wallace. 2003. Hybrid problem solving in ECLiPSe, in M. Milano, ed., *Constraint and Integer Programming: Toward a Unified Methodology*, Kluwer, 169–201.

Aron, I., J. N. Hooker, T. H. Yunes. 2004. SIMPL: A system for integrating optimization techniques, in J.-C. Régin and M. Rueher, eds., *Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2004)*, *Lecture Notes in Computer Science* **3011**, 21–36.

Balas, E., S. Ceria, G. Cornuéjols. 1993. A lift-and-project cutting plane algorithm for mixed 0-1 programs, *Mathematical Programming* **58** 295–324.

Baptiste, P., C. Le Pape, W. Nuijten. 2001. *Constraint-Based Scheduling – Applying Constraint Programming to Scheduling Problems*. Kluwer.

Beck, C., P. Refalo. 2003. A hybrid approach to scheduling with earliness and tardiness costs, *Annals of Operations Research* **118** 49–71.

Benoist, T., E. Gaudin, B. Rottembourg. 2002. Constraint programming contribution to Benders decomposition: A case study, in P. van Hentenryck, ed., *Principles and Practice of Constraint Programming (CP 2002)*, *Lecture Notes in Computer Science* **2470** 603–617.

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.

Bockmayr, A., N. Pisaruk. 2003. Detecting Infeasibility and Generating Cuts for MIP Using CP, *5th International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2003)*, Montréal, Canada.

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

Cheadle, A. M., W. Harvey, A. J. Sadler, J. Schimpf, K. Shen, M. G. Wallace. 2003. ECL^{i}PS^e: An introduction, IC-Parc, Imperial College (London), Technical Report IC-Parc-03-1.

Colombani, Y., S. Heipcke. 2002. Mosel: An extensible environment for modeling and programming solutions, International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2002).

Colombani, Y., S. Heipcke. 2004. Mosel: An overview, Dash Optimization white paper.

Easton, K., G. Nemhauser, M. Trick. 2002. Solving the traveling tournament problem: A combined integer programming and constraint programming approach, *Proceedings of the International Conference on the Practice and Theory of Automated Timetabling (PATAT 2002).*

Floudas, C., P. Pardalos, C. Adjiman, W. Esposito, Z. Gümüs, S. Harding, J. Klepeis, C. Meyer, C. Schweiger. 1999. *Handbook of Test Problems in Local and Global Optimization.* Volume 33 of the series *Nonconvex Optimization and Its Applications.* Kluwer.

Focacci, F., A. Lodi, M. Milano. 1999. Cost-based domain filtering, in J. Jaffar, ed., *Principles and Practice of Constraint Programming (CP)*, *Lecture Notes in Computer Science* **1713** 189–203.

Fourer, R., D. M. Gay, B. W. Kernighan. 2003. *AMPL – A Modeling Language for Mathematical Programming.* Thomson Learning.

Hajian, M. T., H. El-Sakkout, M. Wallace, J. M. Lever, E. B. Richards. 1996. Toward a closer integration of finite domain propagation and simplex-based algorithms, *Proceedings, Fourth International Symposium on Artificial Intelligence and Mathematics.*

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.

Hooker, J. N. 1996. Inference duality as a basis for sensitivity analysis, in E. C. Freuder, ed., *Principles and Practice of Constraint Programming (CP)*, *Lecture Notes in Computer Science* **1118**, Springer, 224–236.

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

Hooker, J. N. 1999. Inference duality as a basis for sensitivity analysis, *Constraints* **4** 104–112.

Hooker, J. N. 2000. *Logic-based Methods for Optimization: Combining Optimization and Constraint Satisfaction*, John Wiley & Sons.

Hooker, J. N. 2002. Logic, optimization and constraint programming, *INFORMS Journal on Computing* **14** 295–321.

Hooker, J. N. 2003. A framework for integrating solution methods, in H. K. Bhargava and Mong Ye, eds., *Computational Modeling and Problem Solving in the Networked World* (Proceedings of ICS2003), Kluwer (2003) 3–30.

Hooker, J. N. 2004. A hybrid method for planning and scheduling, in M. Wallace, ed., *Principles and Practice of Constraint Programming (CP 2004)*, *Lecture Notes in Computer Science* **3258** 305–316.

Hooker, J. N. 2005a. A search-infer-and-relax framework for integrating solution methods, in R. Barták and M. Milano, eds., *Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2005)*, *Lecture Notes in Computer Science* **3709**, 314–327.

Hooker, J. N. 2005b. A hybrid method for planning and scheduling, *Constraints* **10**, 385–401.

Hooker, J. N. 2006. Operations research methods in constraint programming, in F. Rossi, P. van Beek and T. Walsh, eds., *Handbook of Constraint Programming*, Elsevier, 525–568.

Hooker, J. N. 2007. *Integrated Methods for Optimization*, Springer.

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

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

Hooker, J. N., H. Yan. 1995. Logic circuit verification by Benders decomposition, in V. Saraswat and P. Van Hentenryck, eds., *Principles and Practice of Constraint Programming: The Newport Papers*, MIT Press, 267–288.

Hooker, J. N., H. Yan. 2002. A relaxation for the cumulative constraint. In P. Van Hentenryck, editor, *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming (CP)*, *Lecture Notes in Computer Science* **2470**, 686–690. Springer-Verlag.

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

Junker, U., S. E. Karisch, N. Kohl, B. Vaaben, T. Fahle, M. Sellmann. 1999. A framework for constraint programming based column generation, in J. Jaffar, ed., *Principles and Practice of Constraint Programming (CP)*, *Lecture Notes in Computer Science* **1713** 261–274.

Maravelias, C. T., I. E. Grossmann. 2004. A hybrid MILP/CP decomposition approach for the continuous time scheduling of multipurpose batch plants, *Computers and Chemical Engineering* **28** 1921–1949.

Milano, M. 2003. *Constraint and Integer Programming: Toward a Unified Methodology*, Kluwer.

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

Ottosson, G., E. S. Thorsteinsson, J. N. Hooker. 2002. Mixed global constraints and inference in hybrid CLP-IP solvers, *Annals of Mathematics and Artificial Intelligence* **34** 271–290.

Rasmussen, R., M. Trick. 2005. A Benders approach for the minimum break scheduling problem, presented at INFORMS 2005, San Francisco, CA.

Refalo, P. 1999. Tight cooperation and its application in piecewise linear optimization, in J. Jaffar, ed., *Principles and Practice of Constraint Programming (CP 1999)*, *Lecture Notes in Computer Science* **1713** 375–389.

Rodošek, R., M. Wallace. 1998. A generic model and hybrid algorithm for hoist scheduling problems, in M. Maher and J.-F. Puget, eds., *Principles and Practice of Constraint Programming (CP 1998)*, *Lecture Notes in Computer Science* **1520** 385–399.

Rodošek, R., M. Wallace, M. T. Hajian. 1999. A new approach to integrating mixed integer programming and constraint logic programming, *Annals of Operations Research* **86** 63–87.

Sellmann, M., T. Fahle. 2001. Constraint programming based Lagrangian relaxation for a multimedia application, *Third International Workshop on Integration of AI and OR Techniques (CPAIOR 2001)*.

Tawarmalani, M., N. V. Sahinidis. 2004. Global optimization of mixed-integer nonlinear programs: A theoretical and computational study. *Mathematical Programming* **99(3)** 563–591.

Thorsteinsson, E. S. 2001. Branch-and-Check: A hybrid framework integrating mixed integer programming and constraint logic programming, in T. Walsh, ed., *Principles and Practice of Constraint Programming (CP 2001)*, *Lecture Notes in Computer Science* **2239** 16–30.

Thorsteinsson, E. S., G. Ottosson. 2001. Linear relaxations and reduced-cost based propagation of continuous variable subscripts, *Annals of Operations Research* **115** 15–29.

Timpe, C. 2002. Solving planning and scheduling problems with combined integer and constraint programming, *OR Spectrum* **24** 431–448.

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

Van Hentenryck, P., I. Lustig, L. Michel, J. F. Puget. 1999. *The OPL Optimization Programming Language*, MIT Press.

Van Hoeve, W. J. 2003. A hybrid constraint programming and semidefinite programming approach for the stable set problem, in Francesca Rossi, ed., *Principles and Practice of Constraint Programming (CP 2003)*, *Lecture Notes in Computer Science* **2833** 407–421.

Williams, H. P., H. Yan. 2001. Representations of the all_different predicate of constraint satisfaction in integer programming. *INFORMS Journal on Computing* **13(2)** 96–103.

Yunes, T. H., A. V. Moura, C. C. de Souza. 1999. Exact solutions for real world crew scheduling problems. *INFORMS Annual Meeting*, Philadelphia, PA, USA, November 7-10.

Yunes, T. H., A. V. Moura, C. C. de Souza. 2005. Hybrid column generation approaches for urban transit crew management problems, *Transportation Science* **39(2)** 273–288.

Table 1 Sampling of computational results for integrated methods.

Source	Type of problem/method	Speedup
Loose integration of CP and MILP		
Hajian et al. (1996)	British Airways fleet assignment. CP solver provides starting feasible solution for MILP.	Twice as fast as MILP, 4 times faster than CP.
CP plus relaxations similar to those used in MILP		
Focacci et al. (1999)	Lesson timetabling. Reduced-cost variable fixing using an assignment problem relaxation.	2 to 50 times faster than CP.
Refalo (1999)	Piecewise linear costs. Method similar to that described in Section 6.	2 to 200 times faster than MILP. Solved two instances that MILP could not solve.
Hooker & Osorio (1999)	Boat party scheduling, flow shop scheduling. Logic processing plus linear relaxation.	Solved 10-boat instance in 5 min that MILP could not solve in 12 hours. Solved flow shop instances 4 times faster than MILP.
Thorsteinsson & Ottosson (2001)	Product configuration. Method similar to that described in Section 7.	30 to 40 times faster than MILP (which was faster than CP).
Sellmann & Fahle (2001)	Automatic digital recording. CP plus Lagrangean relaxation.	1 to 10 times faster than MILP (which was faster than CP).
Van Hoeve (2001)	Stable set problems. CP plus semi-definite programming relaxation.	Significantly better suboptimal solutions than CP in fraction of the time.
Bollapragada et al. (2001)	Nonlinear structural design. Logic processing plus convex relaxation.	Up to 600 times faster than MILP. Solved 2 problems in < 6 min that MILP could not solve in 20 hours.
Beck & Refalo (2003)	Scheduling with earliness & tardiness costs.	Solved 67 of 90 instances, while CP solved only 12.
CP-based branch and price		
Easton et al. (2002)	Traveling tournament scheduling.	First to solve 8-team instance.
Yunes et al. (2005)	Urban transit crew management.	Solved problems with 210 trips, while traditional branch and price could accommodate only 120 trips.
Benders-based integration of CP and MILP		
Jain & Grossmann (2001)	Min-cost planning and disjunctive scheduling. MILP master problem, CP subproblem (Section 8).	20 to 1000 times faster than CP, MILP.
Thorsteinsson (2001)	Jain & Grossmann problems. Branch and check.	Additional factor of 10 over Jain & Grossmann (2001).
Timpe (2002)	Polypropylene batch scheduling at BASF. MILP master, CP subproblem.	Solved previously insoluble problem in 10 min.
Benoist et al. (2002)	Call center scheduling. CP master, LP subproblem.	Solved twice as many instances as traditional Benders.
Hooker (2004)	Min-cost and min-makespan planning and cumulative scheduling. MILP master, CP subproblem.	100 to 1000 times faster than CP, MILP. Solved significantly larger instances.
Hooker (2005)	Min-tardiness planning & cumulative scheduling. MILP master, CP subproblem.	10 to >1000 times faster than CP, MILP when minimizing # late jobs; ~ 10 times faster when minimizing total tardiness, much better solutions when suboptimal.
Rasmussen & Trick (2005)	Sports scheduling to minimize # of consecutive home or away games.	Speedup of several orders of magnitude compared to previous state of the art.

Table 2 Production planning: search nodes and CPU time.

Number of	MILP (CPLEX 9)		MILP (CPLEX 11)		SIMPL (CPLEX 11)	
Products	Nodes	Time (s)	Nodes	Time (s)	Nodes	Time (s)
5	93	0.03	45	0.03	20	0.02
10	423	0.12	198	0.08	41	0.06
15	1,321	0.34	265	0.19	51	0.10
20	4,573	1.14	652	0.64	38	0.14
25	5,105	2.43	660	0.78	47	0.20
30	4,504	2.10	375	0.44	61	0.34
35	6,089	3.30	550	0.79	34	0.29
40	7,973	4.06	592	1.29	69	0.44
45	23,414	14.72	746	1.44	34	0.41
50	18,795	9.45	677	2.39	45	0.43
55	46,349	34.60	679	2.14	40	0.50
60	99,606	55.07	767	1.78	99	1.14
65	43,759	39.85	784	2.86	101	1.42
70	103,470	98.77	899	2.56	54	0.82
75	76,540	84.54	698	3.61	84	1.79
80	78,479	60.82	865	4.55	59	1.25
85	143,457	185.11	1043	6.07	146	2.91
90	315,456	421.50	1043	6.22	69	1.69
95	345,724	503.95	1174	6.69	104	1.98
100	2,247,538	4,458.65	1229	4.40	137	2.75

Table 3 Product configuration: search nodes and CPU time.

	MILP (CPLEX 9)		SIMPL (CPLEX 9)		MILP (CPLEX 11)		SIMPL (CPLEX 11)	
Instance	Nodes	Time (s)	Nodes	Time (s)	Nodes	Time (s)	Nodes	Time (s)
1	1	0.06	94	0.81	1	0.07	56	0.49
2	1	0.08	32	0.27	1	0.10	32	0.25
3	184	0.79	152	1.43	31	0.68	186	1.67
4	1	0.04	22	0.19	1	0.02	28	0.24
5	723	4.21	24	0.24	1	0.12	32	0.33
6	1	0.05	18	0.19	1	0.07	9	0.09
7	111	0.59	32	0.25	10	0.18	35	0.30
8	20	0.19	39	0.34	1	0.10	32	0.25
9	20	0.17	22	0.17	1	0.05	28	0.22
10	1	0.03	17	0.16	1	0.12	14	0.13

Table 4 Parallel machine scheduling: long processing times.

		MILP (CPLEX 11)		SIMPL Benders		
Jobs	Machines	Nodes	Time (s)	Iterations	Cuts	Time (s)
3	2	1	0.00	2	1	0.00
7	3	1	0.05	13	16	0.12
12	3	3,351	6.56	26	35	0.73
15	5	2,779	8.84	20	29	0.83
20	5	33,321	882.31	43	82	5.44
22	5	352,309	10,563.15	69	98	9.60

Table 5 Parallel machine scheduling: short processing times. An asterisk * means "out of memory".

		MILP (CPLEX 11)		SIMPL Benders		
Jobs	Machines	Nodes	Time (s)	Iterations	Cuts	Time (s)
3	2	1	0.01	1	0	0.00
7	3	1	0.02	1	0	0.00
12	3	499	0.98	1	0	0.01
15	5	529	2.63	2	1	0.06
20	5	250,047	396.19	6	5	0.28
22	5	> 27.5M	> 48h	9	12	0.42
25	5	> 5.4M	> 19h*	17	21	1.09

Table 6 Bilinear global optimization: search nodes and CPU time. BARON and SIMPL were run on different machines.

			BARON		SIMPL	
Problem	Variables	Constraints	Nodes	Time (s)	Nodes	Time (s)
5.2.2.1	9	6	1	0.03	23	0.14
5.2.2.2	9	6	26	0.05	19	0.05
5.2.2.3	9	6	7	0.03	6	0.02
5.2.4	7	6	19	0.06	179	0.64
5.3.2	24	18	19	0.13	53	0.78
5.4.2	8	6	39	0.23	313	2.96