SIMPL: A System for Integrating Optimization Techniques

Ionuţ Aron, John N. Hooker, and Tallys H. Yunes

Graduate School of Industrial Administration, Carnegie Mellon University
Pittsburgh, PA 15213-3890, USA
{iaron,jh38,thy}@andrew.cmu.edu

Abstract In recent years, the Constraint Programming (CP) and Operations Research (OR) communities have explored the advantages of combining CP and OR techniques to formulate and solve combinatorial optimization problems. These advantages include a more versatile modeling framework and the ability to combine complementary strengths of the two solution technologies. This research has reached a stage at which further development would benefit from a general-purpose modeling and solution system. We introduce here a system for integrated modeling and solution called SIMPL. Our approach is to view CP and OR techniques as special cases of a single method rather than as separate methods to be combined. This overarching method consists of an infer-relax-restrict cycle in which CP and OR techniques may interact at any stage. We describe the main features of SIMPL and illustrate its usage with examples.

1 Introduction

In recent years, the Constraint Programming (CP) and Operations Research (OR) communities have explored the advantages of combining CP and OR techniques to formulate and solve combinatorial optimization problems. These advantages include a more versatile modeling framework and the ability to combine complementary strengths of the two solution technologies. Examples of existing programming languages that provide mechanisms for combining CP and OR techniques are ECL^{i}PS^e [32,35], OPL [34] and Mosel [8].

Hybrid methods tend to be most effective when CP and OR techniques interact closely at the micro level throughout the search process. To achieve this one must often write special-purpose code, which slows research and discourages broader application of integrated methods. We address this situation by introducing here a system for integrated modeling and solution called SIMPL (Programming Language for Solving Integrated Models). The SIMPL modeling language formulates problems in such a way as to reveal problem structure to the solver. The solver executes a search algorithm that invokes CP and OR techniques as needed, based on problem characteristics.

The design of such a system presents a significant research problem in itself, since it must be flexible enough to accommodate a wide range of integration

This work has been supported by the National Science Foundation under grant ACI-0121497 and by the William Larimer Mellon Fellowship.

methods and yet structured enough to allow high-level implementation of specific applications. Our approach, which is based partly on a proposal in [14,16], is to view CP and OR techniques as special cases of a single method rather than as separate methods to be combined. This overarching method consists of an infer-relax-restrict cycle in which CP and OR techniques may interact at any stage.

This paper is organized as follows. In Sect. 2, we briefly review some of the fundamental ideas related to the combination of CP and OR that are relevant to the development of SIMPL. We describe the main concepts behind SIMPL in Sect. 3 and talk about implementation details in Sect. 4. Section 5 presents a few examples of how to model optimization problems in SIMPL, explaining the syntax and semantics of the language. Finally, Sect. 6 outlines some additional features provided by SIMPL, and Sect. 7 discusses directions for future work.

2 Previous Work

A comprehensive survey of the literature on the cooperation of logic-based, Constraint Programming (CP) and Operations Research (OR) methods can be found in [15]. Some of the concepts that are most relevant to the work presented here are: decomposition approaches (e.g. Benders [3]) that solve parts of the problem with different techniques [10,14,19,21,24,33]; allowing different models/solvers to exchange information [32]; using linear programming to reduce the domains of variables or to fix them to certain values [4,11,32]; automatic reformulation of global constraints as systems of linear inequalities [30]; continuous relaxations of global constraints and disjunctions of linear systems [1,14,18,22,28,36,37,38]; understanding the generation of cutting planes as a form of logical inference [6,7]; strengthening the problem formulation by embedding the generation of valid cutting planes into CP constraints [12]; maintaining the continuous relaxation of a constraint updated when the domains of its variables change [29]; and using global constraints as a key component in the intersection of CP and OR [27].

Ideally, one would like to incorporate all of the above techniques into a single modeling and solving environment, in a clean and generic way. Additionally, this environment should be flexible enough to accommodate improvements and modifications with as little extra work as possible. In the next sections, we present the concepts behind SIMPL that aim at achieving those objectives.

3 SIMPL Concepts

We first review the underlying solution algorithm and then indicate how the problem formulation helps to determine how particular problems are solved.

3.1 The Solver

SIMPL solves problems by enumerating problem restrictions. (A restriction is the result of adding constraints to the problem.) Each node of a classical branch-and-bound tree, for example, can be viewed as a problem restriction defined by fixing certain variables or reducing their domains. Local search methods fit into the same scheme, since they examine a sequence of neighborhoods, each of which is the feasible set of a problem restriction. Thus SIMPL implements both exact and heuristic methods within the same architecture.

The search proceeds by looping through an infer-relax-restrict cycle: it infers new constraints from the current problem restriction, then formulates and solves relaxations of the augmented problem restriction, and finally moves to another problem restriction to be processed in the same way. The user specifies the overall search procedure from a number of options, such as depth-first branching, local search, or Benders decomposition. The stages in greater detail are as follows.

Infer. New constraints are deduced from the original ones and added to the current problem restriction. For instance, a filtering algorithm can be viewed as inferring indomain constraints that reduce the size of variable domains. A cutting plane algorithm can generate inequality constraints that tighten the continuous relaxation of the problem as well as enhance interval propagation.

Relax. One or more relaxations of the current problem restriction are formulated and solved by specialized solvers. For instance, continuous relaxations of some or all of the constraints can be collected to form a relaxation of the entire problem, which is solved by a linear or nonlinear programming subroutine. The role of relaxations is to help direct the search, as described in the next step.

Restrict. The relaxations provide information that dictates which new restrictions are generated before moving to the next restriction. In a tree search, for example, SIMPL creates new restrictions by branching on a constraint that is violated by the solution of the current relaxation. If several constraints are violated, one is selected according to user- or system-specified priorities (see Sect. 4.3). Relaxations can also influence which restriction is processed next, for instance by providing a bound that prunes a branch-and-bound tree.

If desired, an inner infer-relax loop can be executed repeatedly before moving to the next problem restriction, since the solution of the relaxation may indicate further useful inferences that can be drawn (post-relaxation inference). An example would be separating cuts, which are cutting planes that “cut off” the solution of the relaxation (see Sect. 4.3).

The best-known classical solution methods are special cases of the infer-relax-restrict procedure:

– In a typical *CP solver*, the inference stage consists primarily of domain reduction. The relaxation stage builds a (weak) relaxation simply by collecting the reduced domains into a constraint store. New problem restrictions are created by splitting a domain in the relaxation.
– In a *branch-and-bound solver* for integer programming, the inference stage can be viewed as “preprocessing” that takes place at the root node and possibly at subsequent nodes. The relaxation stage drops the integrality constraints and solves the resulting problem with a linear or perhaps nonlinear programming solver. New problem restrictions are created by branching on an integrality constraint; that is, by branching on a variable with a fractional value in the solution of the relaxation.

– A *local search* procedure typically chooses the next solution to be examined from a neighborhood of the current solution. Thus local search can be regarded as enumerating a sequence of problem restrictions, since each neighborhood is the feasible set of a problem restriction. The "relaxation" of the problem restriction is normally the problem restriction itself, but need not be. The restriction may be solved to optimality by an exhaustive search of the neighborhood, as in tabu search (where the tabu list is part of the restriction). Alternatively, a suboptimal solution may suffice, as in simulated annealing, which selects a random element of the neighborhood.
– In *Branch-and-Infer* [7], the relaxation stage is not present and branching corresponds to creating new problem restrictions.

An important advantage of SIMPL is that it can create new infer-relax-restrict procedures that suit the problem at hand. One example is a hybrid algorithm, introduced in [14,21], that is obtained through a generalization of Benders decomposition. It has provided some of the most impressive speedups achieved by hybrid methods [10,16,17,19,24]. A Benders algorithm distinguishes a set of primary variables that, when fixed, result in an easily-solved subproblem. Solution of an "inference dual" of the subproblem yields a Benders cut, which is added to a master problem containing only the primary variables. Solution of the master problem fixes the primary variables to another value, and the process continues until the optimal values of the master problem and subproblem converge. In typical applications, the master problem is an integer programming problem and the subproblem a CP problem. This method fits nicely into the infer-relax-restrict paradigm, since the subproblems are problem restrictions and master problems are relaxations. The solution of the relaxation guides the search by defining the next subproblem.

The choice of constraints in a SIMPL model can result in novel combinations of CP, OR and other techniques. This is accomplished as described in Sect. 3.2.

3.2 Modeling

SIMPL is designed so that the problem formulation itself determines to a large extent how CP, OR, and other techniques interact. The basic idea is to view each constraint as invoking specialized procedures that exploit the structure of that particular constraint. Since some of these procedures may be from CP and some from OR, the two approaches interact in a manner that is dictated by which constraints appear in the problem.

This idea of associating constraints with procedures already serves as a powerful device for exploiting problem substructure in CP, where a constraint typically activates a specialized filtering algorithm. SIMPL extends the idea by associating each constraint with procedures in all three stages of the search. Each constraint can (a) activate inference procedures, (b) contribute constraints to one or more relaxations, and (c) generate further problem restrictions if the search branches on that particular constraint.

If a group of constraints exhibit a common structure—such as a set of linear inequalities, flow balance equations, or logical formulas in conjunctive normal form—they are identified as such so that the solver can take advantage of their structure. For instance, a resolution-based inference procedure might be applied to the logical formulas.

The existing CP literature typically provides inference procedures (filters) only for CP-style global constraints, and the OR literature provides relaxations (cutting planes) only for structured groups of linear inequalities. This poses the research problem of finding specialized relaxations for global constraints and specialized filters for structured linear systems. Some initial results along this line are surveyed in [15].

Some examples should clarify these ideas. The global constraint *element* is important for implementing variable indices. Conventional CP solvers associate *element* with a specialized filtering algorithm, but useful linear relaxations, based on OR-style polyhedral analysis, have recently been proposed as well [20]. Thus each *element* constraint can activate a domain reduction algorithm in the inference stage and generate linear inequalities, for addition to a continuous relaxation, in the relaxation stage. If the search branches on a violated *element* constraint, then new problem restrictions are generated in a way that makes sense when that particular constraint is violated.

The popular *all-different* and *cumulative* constraints are similar in that they also have well-known filters [31,2] and were recently provided with linear relaxations [36,22]. These relaxations are somewhat weak and may not be useful, but the user always has the option of turning off or on the available filters and relaxations, perhaps depending on the current depth in the search tree.

Extensive polyhedral analysis of the traveling salesman problem in the OR literature [13,25] provides an effective linear relaxation of the *cycle* constraint. In fact, SIMPL has the potential to make better use of the traditional OR literature than commercial OR solvers. Structured groups of inequalities can be represented by global constraints that trigger the generation of specialized cutting planes, many of which go unused in today's general-purpose solvers.

4 From Concepts to Implementation

SIMPL is implemented in C++ as a collection of object classes, as shown in Fig. 1. This makes it easy to add new components to the system by making only localized changes that are transparent to the other components. Examples of components that can be included are: new constraints, different relaxations for existing constraints, new solvers, improved inference algorithms, new branching modules and selection modules, alternative representations of domains of variables, etc. The next sections describe some of these components in detail.

4.1 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*[1] N_R of N, or possibly several relaxations.

[1] In general, we say that problem Q_R is a relaxation of problem Q if the feasible region of Q_R contains the feasible region of Q.

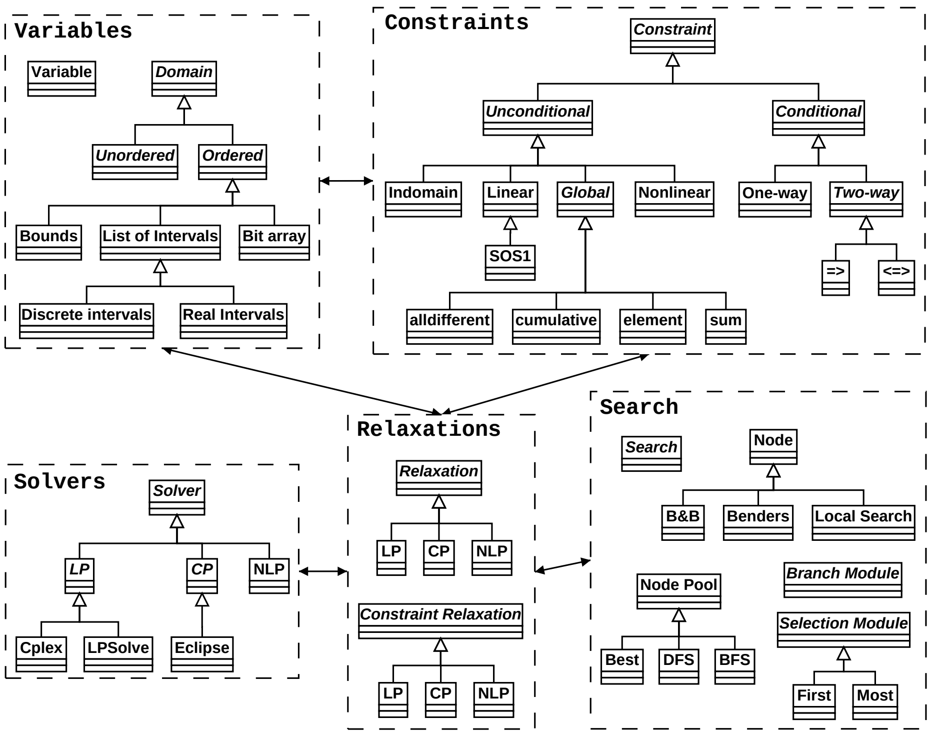


Figure1. Main components of SIMPL

In an integrated CP-IP modeling system, the linear constraints in the hybrid formulation are posted to a Linear Programming (LP) solver, and some (or all) of them may be posted to a CP solver as well. The CP solver also handles the constraints that cannot be directly posted to the LP solver (e.g. global constraints). Notice that each solver only deals with a relaxation of the original problem P (i.e. a subset of its constraints). In this example, each problem restriction N has two relaxations: an *LP relaxation* and a *CP relaxation*. Extending this idea to more than two kinds of relaxations is straightforward.

In principle, the LP relaxation of N could simply ignore the constraints that are not linear. Nevertheless, this relaxation can be strengthened by the addition of linear relaxations of those constraints, if available (see Sect. 4.2).

4.2 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 [14]. Besides the ones already mentioned in Sect. 3.2, other constraints for which linear relaxations are known include *cardinality rules* [37] and *sum* [38].

For a branch-and-bound type of search, the problem relaxations to be solved at a node of the enumeration tree depend on the state of the search at that node. In theory, at every node, the relaxations are to be created from scratch because constraint relaxations are a function of the domains of the variables of the original (non-relaxed) constraint. Nevertheless, this can be very inefficient because a significant part of the constraints in the relaxations will be the same from node to node. Hence, we divide constraint relaxations in two types:

Static: those that 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);
Volatile: those that radically change when variable domains change (e.g. some linear relaxations of global constraints).

To update the problem relaxations when we move from one node in the search tree to another, it suffices to recompute volatile constraint relaxations only. This kind of update is not necessary for the purpose of creating valid relaxations, but it is clearly beneficial from the viewpoint of obtaining stronger bounds.

4.3 Search

The main search loop in SIMPL is implemented as shown in Fig. 2. Here, N

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

Figure2. The main search loop in SIMPL

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 Sect. 3.1. 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 will restrict our attention to branch-and-bound search.

Node Exploration. Figure 3 describes the behavior of N`.explore()` for a branch-and-bound type of search. Steps 1 and 4 are inference steps where we

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

Figure3. The node exploration loop in branch-and-bound

try to use the information from each relaxation present in the model to the most profitable extent. Section 4.4 provides further details about the types of inference used in those steps. The whole loop can be repeated multiple times, as long as domains of variables keep changing because of step 4, and the maximum number of iterations has not been reached. This process of re-solving relaxations and looking for further inferences behaves similarly to a fix point calculation.

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” what to do. This knowledge is embedded in the so called *branching module* of that 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 will then output two constraints: $x \in \{0\}$ and $x \in \{1\}$, meaning that two subproblems should be created by the inclusion of those two new constraints. In this sense, branching on the variable x can be interpreted as branching on I_x. In general, a branching module returns a sequence of *sets* of constraints $C_1, \dots, C_k$. This sequence means that k subproblems should be created, and subproblem i can be constructed from the current problem by the inclusion of all constraints present in the set C_i. There is no restriction on the types of constraints that can be part of the sets 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.

4.4 Inference

We now take a closer look at the inference steps of the node exploration loop in Fig. 3. In step 1 (pre-relaxation inference), one may have domain reductions or the generation of new implied constraints (see [18]), 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 [14].

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 shown in Fig. 4. In step 6, D_v^r denotes the domain of v inside relaxation r, and

```
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 $v \in D_v$
```

Figure4. Synchronizing domains of variables across multiple relaxations

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 descendents of the current node at a later stage.

5 SIMPL Examples

SIMPL's syntax is inspired by OPL [34], but it includes many new features.

Apart from the resolution algorithm used in Sect. 5.3, SIMPL is currently able to run all the examples presented in this section. Problem descriptions and formulations were taken from Chapter 2 of [14].

5.1 A Hybrid Knapsack Problem

Let us consider the following integer knapsack problem with a side constraint.

$$\begin{array}{rl} \min & 5x_1 + 8x_2 + 4x_3 \\ \text{subject to} & 3x_1 + 5x_2 + 2x_3 \geq 30 \\ & \textit{all-different}(x_1, x_2, x_3) \\ & x_j \in \{1, 2, 3, 4\}, \text{ for all } j \end{array}$$

To handle the *all-different* constraint, a pure MIP model would need auxiliary binary variables: $y_{ij} = 1$ if and only if $x_i = j$. A SIMPL model for the above problem is shown in Fig. 5. The model starts with a DECLARATIONS section in which constants and variables are defined. The objective function is defined in line 06. Notice that the range over which the index i takes its values need not be explicitly stated. In the CONSTRAINTS section, the two constraints of the problem are named totweight and distinct, and their definitions show up in lines 09 and 12, respectively. The RELAXATION statements in lines 10 and 13 indicate the relaxations to which those constraints should be posted. The linear constraint will be present in both the LP and the CP relaxations, whereas the alldiff constraint will only be present in the CP relaxation. In the SEARCH section, line 15 indicates we will do branch-and-bound (BB) with depth-first search (DEPTH). The BRANCHING statement in line 16 says that we will branch on the first of the x variables that is not integer (remember from Sect. 4.3 that branching on a variable means branching on its indomain constraint).

```
01. DECLARATIONS
02.     n = 3; cost[1..n] = [5,8,4]; weight[1..n] = [3,5,2]; limit = 30;
03.     DISCRETE RANGE xRange = 1 TO 4;
04.     x[1..n] IN xRange;
05. OBJECTIVE
06.     MIN SUM i OF cost[i]*x[i]
07. CONSTRAINTS
08.     totweight MEANS {
09.        SUM i OF weight[i]*x[i] >= limit
10.        RELAXATION = {LP, CS} }
11.     distinct MEANS {
12.        alldiff(x)
13.        RELAXATION = {CS} }
14. SEARCH
15.     TYPE = {BB:DEPTH}
16.     BRANCHING = {x:FIRST}
```

Figure5. SIMPL model for the Hybrid Knapsack Problem

Initially, bounds consistency maintenance in the CP solver removes value 1 from the domain of x_2 and the solution of the LP relaxation is $x = (2\frac{2}{3}, 4, 1)$. After branching on $x_1 \leq 2$, bounds consistency determines that $x_1 \geq 2$, $x_2 \geq 4$ and $x_3 \geq 2$. At this point, the alldiff constraint produces further domain reduction, yielding the feasible solution $(2, 4, 3)$. Notice that no LP relaxation had to be solved at this node. In a similar fashion, the CP solver may be able to detect infeasibility even before the linear relaxation has to be solved.

5.2 A Lot Sizing Problem

A total of P products must be manufactured over T days on a single machine of limited capacity C, at most one product each day. When manufacture of a given product i begins, it may proceed for several days, and there is a minimum run length r_i. Given a demand d_{it} for each product i on each day t, it is usually necessary to hold part of the production of a day for later use, at a unit cost of h_{it}. Changing from product i to product j implies a setup cost q_{ij}. Frequent changeovers allow for less holding cost but incur more setup cost. The objective is to minimize the sum of the two types of costs while satisfying demand.

Let $y_t = i$ if and only if product i is chosen to be produced on day t, and let x_{it} be the quantity of product i produced on day t. In addition, let u_t, v_t and s_{it} represent, respectively, for day t, the holding cost, the changeover cost and the ending stock of product i. Figure 6 exhibits a SIMPL model for this problem. We have omitted the data that initializes matrices d, h, q and r. We have also left out the statements that set $y_0 = 0$ and $s_{i0} = 0$ for $i \in \{1, \ldots, P\}$.

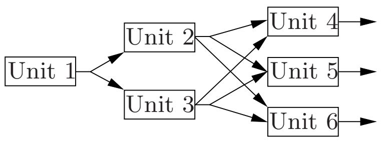
```
01. DECLARATIONS
02.     P = 5; T = 10; C = 50;
03.     d[1..P,1..T] = ; h[1..P,1..T] = ; q[0..P,1..P] = ; r[1..P] = ;
04.     CONTINUOUS RANGE xRange = 0 TO C;
05.     DISCRETE RANGE yRange = 0 TO P;
06.     x[1..P,1..T] IN xRange; y[0..T] IN yRange;
07.     u[1..T], v[1..T], s[1..P,0..T] IN nonegative;
08. OBJECTIVE
09.     MIN SUM t OF u[t] + v[t]
10. RELAXATIONS
11.     LP, CS
12. CONSTRAINTS
13.     holding MEANS { u[t] >= SUM i OF h[i,t]*s[i,t] FORALL t }
14.     setup MEANS { v[t] >= q[y[t-1],y[t]] FORALL t }
15.     stock MEANS { s[i,t-1] + x[i,t] = d[i,t] + s[i,t] FORALL i,t }
16.     linkyx MEANS { y[t] <> i -> x[i,t] = 0 FORALL i,t }
17.     minrun MEANS {
18.        y[t-1] <> i and y[t] = i ->
19.        (y[t+k] = i FORALL k IN 1 TO r[i]-1) FORALL i,t }
20. SEARCH
21.     TYPE = {BB:BEST}
22.     BRANCHING = {setup:MOST}
```

Figure6. SIMPL model for the Lot Sizing Problem

In line 07, we use the predefined continuous range `nonegative`. Notice the presence of a new section called RELAXATIONS, whose role in this example is to define the default relaxations to be used. As a consequence, the absence of RELAXATION statements in the declaration of constraints means that all constraints will be posted to both the LP and CS relaxations. The `holding` and `stock` constraints define, respectively, holding costs and stock levels in the usual way. The `setup` constrains make use of variable indexing to obtain the desired meaning for the v_t variables. The CS relaxation of these constraints uses *element* constraints, and the LP relaxation uses the corresponding linear relaxation of *element*. The symbol `->` in lines 16 and 18 implements a one-way link constraint of the form $A \rightarrow B$ (see [18]). This means that whenever condition A is true, B is imposed as a constraint of the model, but we do not worry about the contrapositive. Condition A may be a more complicated logical statement and B can be any collection of arbitrary constraints. There are also two-way link constraints such as "implies" (`=>`) and "if and only if" (`<=>`) available in SIMPL. Here, the `linkyx` constraints ensure that x_{it} can only be positive if $y_t = i$, and the `minrun` constraints make production last the required minimum length. The statements in lines 21 and 22 define a branch-and-bound search with best-bound node selection, and branching on the most violated of the `setup` constraints, respectively.

5.3 Processing Network Design

This problem consists of designing a chemical processing network. In practice one usually starts with a network that contains all the processing units and connecting links that could eventually be built (i.e. a superstructure). The goal is to select a subset of units that deliver the required outputs while minimizing installation and processing costs. The discrete element of the problem is the choice of units, and the continuous element comes in determining the volume of flow between units. Let us consider the simplified superstructure in Fig. 7(a). Unit 1 receives raw material, and units 4, 5 and 6 generate finished products. The



```
build MEANS {
   y1 => (y2 or y3), y3 => y4, y2 => y1,
   y3 => (y5 or y6), y2 => (y4 or y5),
   y4 => (y2 or y3), y2 => y6, y3 => y1,
   y5 => (y2 or y3), y6 => (y2 or y3)
   RELAXATION = {LP, CS}
   INFERENCE = {RESOLUTION} }
```

Figure7. (a) Network superstructure (b) The INFERENCE statement in SIMPL

output of unit 1 is then processed by unit 2 and/or 3, and their outputs undergo further processing. For the purposes of this example, we will concentrate on the selection of units, which is amenable to the following type of logical reasoning. Let the propositional variable y_i be true when unit i is installed and false otherwise. From Fig. 7(a), it is clearly useless to install unit 1 unless one installs unit 2 or unit 3. This condition can be written as $y_1 \Rightarrow (y_2 \vee y_3)$. Other rules of this kind can be derived in a similar way. SIMPL can take advantage of the presence of such rules in three ways: it can relax logical propositions into linear constraints; it can use the propositions *individually* as two-way link constraints (see Sect. 5.2); and it can use the propositions *collectively* with an inference algorithm to deduce stronger facts. The piece of code in Fig. 7(b) shows how one would group this collection of logical propositions as a constraint in SIMPL. In addition to the known RELAXATION statement, this example introduces an INFERENCE statement whose role is to attach an inference algorithm (resolution) to the given group of constraints. This algorithm will be invoked in the pre-relaxation inference step, as described in Sect. 4.4. Newly inferred resolvents can be added to the problem relaxations and may help the solution process.

5.4 Benders Decomposition

Recall from Sect. 4.3 that Benders decomposition is a special case of SIMPL's search mechanism. Syntatically, to implement Benders decomposition the user only needs to include the keyword MASTER in the RELAXATION statement of each constraint that is meant to be part of the master problem (remaining constraints go to the subproblem), and declare TYPE = {BENDERS} in the SEARCH section. As is done for linear relaxations of global constraints, Benders cuts are generated by an algorithm that resides inside each individual constraint. At present, we are in the process of implementing the class *Benders* in the diagram of Fig. 1.

6 Other SIMPL Features

Supported Solvers. Currently, SIMPL can interface with CPLEX [23] and LP_SOLVE [5] as LP solvers, and with ECL^{i}PS^e [35] as a CP solver. Adding a new solver to SIMPL is an easy task and amounts to implementing an interface to that solver's callable library, as usual. The rest of the system does not need to be changed or recompiled. One of the next steps in the development of SIMPL is the inclusion of a solver to handle non-linear constraints.

Application Programming Interface. Although SIMPL is currently a purely declarative language, it will eventually include more powerful (imperative) search constructs, such as loops and conditional statements. Meanwhile, it is possible to implement more elaborate algorithms that take advantage of SIMPL's paradigm via its Application Programming Interface (API). This API can be compiled into any customized C++ code and works similarly to other callable libraries available for commercial solvers like CPLEX or XPRESS [9].

Search Tree Visualization. Once a SIMPL model finishes running, it is possible to visualize the search tree by using Leipert's VBC Tool package [26]. Nodes in the tree are colored red, green, black and blue to mean, respectively, pruned by infeasibility, pruned by local optimality, pruned by bound and branched on.

7 Conclusions and Future Work

In this paper we introduce a system for dealing with integrated models called SIMPL. The main contribution of SIMPL is to provide a user-friendly framework that generalizes many of the ways of combining Constraint Programming (CP) and Operations Research (OR) techniques when solving optimization problems. Although there exist other general-purpose systems that offer some form of hybrid modeling and solver cooperation, they do not incorporate various important features available in SIMPL.

The implementation of specialized hybrid algorithms can be a very cumbersome task. It often involves getting acquainted with the specifics of more than one type of solver (e.g. LP, CP, NLP), as well as a significant amount of computer programming, which includes coordinating the exchange of information among solvers. Clearly, a general purpose code is built at the expense of performance. Rather than defeating state-of-the-art implementations of cooperative approaches that are tailored to specific problems, SIMPL's objective is to be a generic and easy-to-use platform for the development and empirical evaluation of new ideas in the field of hybrid CP-OR algorithms.

As SIMPL is still under development, many new features and improvements to its functionality are the subject of ongoing efforts. Examples of such enhancements are: increasing the vocabulary of the language with new types of constraints; augmenting the inference capabilities of the system with the generation of cutting planes; broadening the application areas of the system by supporting other types of solvers; and providing a more powerful control over search. Finally, SIMPL is currently being used to test integrated models for a few practical optimization problems such as the lot-sizing problem of Sect. 5.2.

References

1. E. Balas. Disjunctive programming: Properties of the convex hull of feasible points. *Discrete Applied Mathematics*, 89:3–44, 1998.

2. P. Baptiste, C. Le Pape, and W. Nuijten. *Constraint-Based Scheduling.* Kluwer, 2001.
3. J. F. Benders. Partitioning procedures for solving mixed-variables programming problems. *Numerische Mathematik*, 4:238–252, 1962.
4. H. Beringer and B. de Backer. Combinatorial problem solving in constraint logic programming with cooperating solvers. In C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications.* Elsevier Science, 1995.
5. M. Berkelaar. LP_SOLVE. Available from ftp://ftp.ics.ele.tue.nl/pub/lp_solve/.
6. A. Bockmayr and F. Eisenbrand. Combining logic and optimization in cutting plane theory. In H. Kirchner and C. Ringeissen, editors, *Proceedings of the Third International Workshop on Frontiers of Combining Systems (FroCos)*, *LNAI* 1794, 1–17. Springer-Verlag, 2000.
7. A. Bockmayr and T. Kasper. Branch and infer: A unifying framework for integer and finite domain constraint programming. *INFORMS Journal on Computing*, 10(3):287–300, 1998.
8. Y. Colombani and S. Heipcke. *Mosel: An Overview.* Dash Optimization, 2002.
9. Dash Optimization. XPRESS-MP. http://www.dashoptimization.com.
10. A. Eremin and M. Wallace. Hybrid Benders decomposition algorithms in constraint logic programming. In Toby Walsh, editor, *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming (CP)*, *LNCS* 2239, 1–15. Springer-Verlag, 2001.
11. F. Focacci, A. Lodi, and M. Milano. Cost-based domain filtering. In J. Jaffar, editor, *Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming (CP)*, *LNCS* 1713, 189–203. Springer-Verlag, 1999.
12. F. Focacci, A. Lodi, and M. Milano. Cutting planes in constraint programming: A hybrid approach. In R. Dechter, editor, *Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming (CP)*, *LNCS* 1894, 187–201. Springer-Verlag, 2000.
13. G. Gutin and A. P. Punnen, editors. *Traveling Salesman Problem and Its Variations.* Kluwer, 2002.
14. J. N. Hooker. *Logic-Based Methods for Optimization.* Wiley-Interscience Series in Discrete Mathematics and Optimization, 2000.
15. J. N. Hooker. Logic, optimization and constraint programming. *INFORMS Journal on Computing*, 14(4):295–321, 2002.
16. J. N. Hooker. A framework for integrating solution methods. In H. K. Bhargava and M. Ye, editors, *Computational Modeling and Problem Solving in the Networked World*, pages 3–30. Kluwer, 2003. Plenary talk at the Eighth INFORMS Computing Society Conference (ICS).
17. J. N. Hooker. Logic-based benders decomposition for planning and scheduling. Manuscript, GSIA, Carnegie Mellon University, 2003.
18. J. N. Hooker and M. A. Osorio. Mixed logical/linear programming. *Discrete Applied Mathematics*, 96–97(1–3):395–442, 1999.
19. J. N. Hooker and G. Ottosson. Logic-based benders decomposition. *Mathematical Programming*, 96:33–60, 2003.
20. J. N. Hooker, G. Ottosson, E. Thorsteinsson, and H.-J. Kim. On integrating constraint propagation and linear programming for combinatorial optimization. In *Proceedings of the 16th National Conference on Artificial Intelligence*, pages 136–141. MIT Press, 1999.
21. J. N. Hooker and H. Yan. 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 (Cambridge, MA, 1995) 267-288.

22. J. N. Hooker and H. Yan. 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)*, *LNCS* 2470, 686–690. Springer-Verlag, 2002.
23. ILOG S. A. The CPLEX mixed integer linear programming and barrier optimizer. http://www.ilog.com/products/cplex/.
24. V. Jain and I. E. Grossmann. Algorithms for hybrid MILP/CP models for a class of optimization problems. *INFORMS Journal on Computing*, 13(4):258–276, 2001.
25. E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. John Wiley & Sons, 1985.
26. S. Leipert. The tree interface version 1.0: A tool for drawing trees. Available at http://www.informatik.uni-koeln.de/old-ls_juenger/projects/vbctool.html.
27. M. Milano, G. Ottosson, P. Refalo, and E. S. Thorsteinsson. The role of integer programming techniques in constraint programming's global constraints. *INFORMS Journal on Computing*, 14(4):387–402, 2002.
28. G. Ottosson, E. S. Thorsteinsson, and J. N. Hooker. Mixed global constraints and inference in hybrid CLP-IP solvers. In *CP'99 Post Conference Workshop on Large Scale Combinatorial Optimization and Constraints*, pages 57–78, 1999.
29. P. Refalo. Tight cooperation and its application in piecewise linear optimization. In J. Jaffar, editor, *Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming (CP)*, *LNCS* 1713, 375–389. Springer-Verlag, 1999.
30. P. Refalo. Linear formulation of constraint programming models and hybrid solvers. In R. Dechter, editor, *Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming (CP)*, *LNCS* 1894, 369–383. Springer-Verlag, 2000.
31. J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the National Conference on Artificial Intelligence*, pages 362–367, 1994.
32. R. Rodošek, M. Wallace, and M. T. Hajian. A new approach to integrating mixed integer programming and constraint logic programming. *Annals of Operations Research*, 86:63–87, 1999.
33. E. S. Thorsteinsson. Branch-and-Check: A hybrid framework integrating mixed integer programming and constraint logic programming. In Toby Walsh, editor, *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming (CP)*, *LNCS* 2239, 16–30. Springer-Verlag, 2001.
34. P. Van Hentenryck. *The OPL Optimization Programming Language*. MIT Press, 1999.
35. M. Wallace, S. Novello, and J. Schimpf. ECL^{i}PS^e: A platform for constraint logic programming. *ICL Systems Journal*, 12:159–200, 1997.
36. H. P. Williams and H. Yan. Representations of the all_different predicate of constraint satisfaction in integer programming. *INFORMS Journal on Computing*, 13(2):96–103, 2001.
37. H. Yan and J. N. Hooker. Tight representations of logical constraints as cardinality rules. *Mathematical Programming*, 85:363–377, 1999.
38. T. H. Yunes. On the sum constraint: Relaxation and applications. In P. Van Hentenryck, editor, *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming (CP)*, *LNCS* 2470, 80–92. Springer-Verlag, 2002.