2004

Constraint Programming for Scheduling

John J. Kanet
University of Dayton, jkanet1@udayton.edu

Sanjay L. Ahire
University of Dayton

Michael F. Gorman
University of Dayton, mgorman1@udayton.edu

47

Constraint Programming for Scheduling

John J. Kanet
University of Dayton

Sanjay L. Ahire
University of Dayton

Michael F. Gorman
University of Dayton

47.1 Introduction

Scheduling has been a focal point in operations research (OR) for over fifty years. Traditionally, either special purpose algorithms or integer programming (IP) models have been used. More recently, the computer science field, and in particular, logic programming from artificial intelligence has developed another approach using a declarative style of problem formulation and associated constraint resolution algorithms

1-58488-397-9/$0.00+$1.50

for solving such combinatorial optimization problems [1–3]. This approach, termed as constraint logic programming or CLP (or simply CP), has significant implications for the OR community in general, and for scheduling research in particular.

In this chapter, our goal is to introduce the constraint programming (CP) approach within the context of scheduling. We start with an introduction to CP and its distinct technical vocabulary. We then present and illustrate a general algorithm for solving a CP problem with a simple scheduling example. Next, we review several published studies where CP has been used in scheduling problems so as to provide a feel for its applicability. We discuss the advantages of CP in modeling and solving certain types of scheduling problems. We then provide an illustration of the use of a commercial CP tool (OPL Studio®[1]) in modeling and designing a solution procedure for a classic problem in scheduling. Finally, we conclude with our speculations about the future of scheduling research using this approach.

47.2 What is Constraint Programming (CP)?

We define CP as an approach for formulating and solving discrete variable constraint satisfaction or constrained optimization problems that systematically employs deductive reasoning to reduce the search space and allows for a wide variety of constraints. CP extends the power of logic programming through application of more powerful search strategies and the capability to control their design using problem-specific knowledge [1,3].

CP involves the use of a mathematical/logical modeling language for encoding the formulation, and allows the user to apply a wide range of search strategies, including customized search techniques for finding solutions. CP is very flexible in terms of formulation power and solution approach, but requires skill in declarative-style logic programming and in developing good search strategies.

We begin with a description of the structural features of CP problems and then present a general CP computational framework. In this process, we introduce several technical CP keywords/concepts (denoted in *italics*) that are either not encountered in typical OR research or are used differently in the CP context.

47.2.1 Constraint Satisfaction Problem

At the heart of CP is the constraint satisfaction problem (CSP). Brailsford, Potts, and Smith [4] define CSP as follows: Given a set of discrete variables, together with finite domains, and a set of constraints involving these variables, find a solution that satisfies all the constraints.

In constraint satisfaction problems, variables can assume different types including: Boolean, integer, symbolic, set elements, and subsets of sets. Similarly, a variety of constraints are possible, including:

- mathematical: $C = s + p$ (completion time = start time + processing time)
- *disjunctive*: tasks J and K must be done at different times
- relational: at most five jobs to be allocated at machine R50
- explicit: only jobs A, B, and E can be processed on machine Y50

Finally, a CSP can involve a wide variety of constraint operators, such as: $=, <, >, \geq, \leq, \neq$, subset, superset, union, member, $\vee$ (Boolean OR), $\bullet$ (Boolean AND), $\Rightarrow$ (implies), and $\Leftrightarrow$ (iff). In addition to these constraints, CP researchers have developed special purpose constraints that can implement combinations of the above types of constraints efficiently. For example, the *alldifferent* constraint for a set of variables implements their pairwise inequalities efficiently through logical filtering schemes [5].

A (feasible) *solution* of a CSP is an assignment of a value from its domain to every variable, in such a fashion that every constraint of the problem is satisfied. When tackling a CSP, we may want to identify just one or multiple solutions.

[1] OPL Studio® (ILOG, Inc., Mountain View, CA).

47.3 How Does Constraint Programming Solve a CSP?

47.3.1 A General Algorithm for CSP

Figure 47.1 summarizes a general algorithm for solving a CSP. It starts with CSP formulation including defining variables, their domains, and constraints (block 1). Constraints are stored in a *constraint store.*

In block 2 of the figure, the domains of individual variables are reduced using logic-based *filtering algorithms* for each constraint that systematically reduce the domains of the variables. As the domain of a variable is reduced, each constraint that uses the variable is then activated for application of its associated filtering algorithm. This systematic process is called *constraint propagation* and *domain reduction.*

After constraint propagation and domain reduction, two possibilities arise (block 3), i.e., either a solution is found or not. If a solution is found the algorithm terminates (End1). If all solutions are required, the basic process is repeated. If no solution is found, problem *inconsistency* (the state where the domain of at least one variable has become empty) at the current stage is examined (block 4).

If inconsistency is not proven, then a search is undertaken using some search strategy for *branching* (block 6). Branching divides the main problem into a set of mutually exclusive and collectively exhaustive subproblems by temporarily adding a constraint. Branching selects one of the branches and propagates all constraints again using the filtering algorithms (block 2).

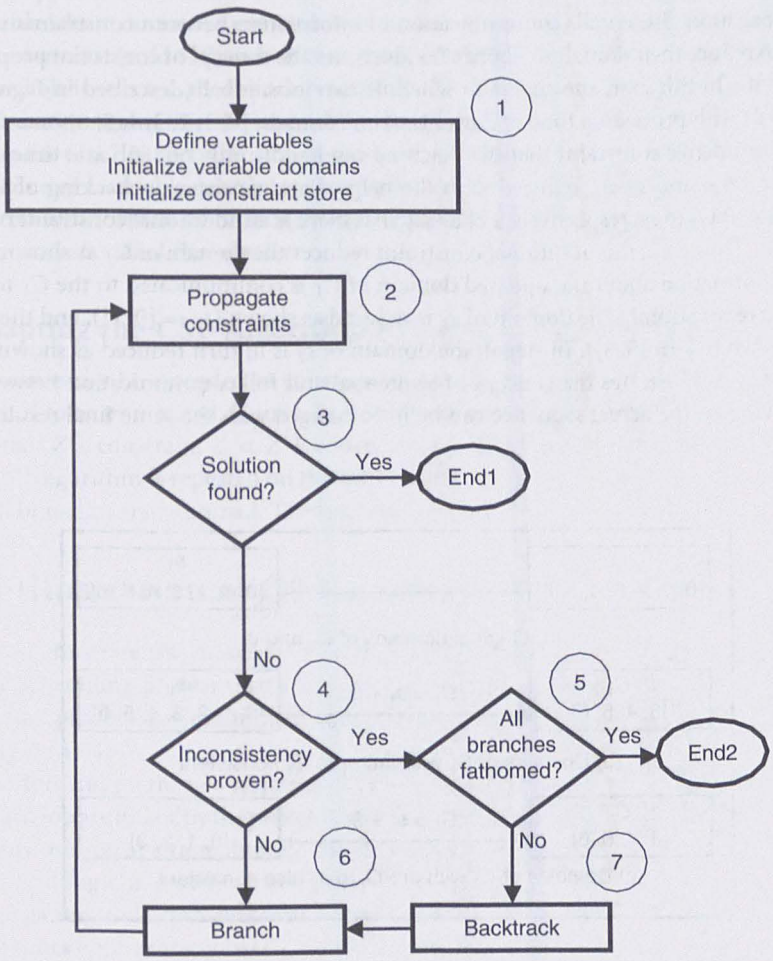

FIGURE 47.1 A general algorithm for constraint satisfaction problems (CSP).

If, in block 4, inconsistency is proven (a *failure*), the search tree is examined in block 5 to check if all subproblems have been explored (fathomed). If all branches have been fathomed, problem inconsistency is proven. If not, the algorithm *backtracks* (block 7) to the previous stage and branches to a different subproblem (block 6).

47.3.2 Constraint Propagation

47.3.2.1 Within-Constraint Domain Reduction

Constraint propagation (Figure 47.1-block 2) uses the concept of logical *arc consistency checking* to communicate information about variable domain reduction within and across constraints involving the specific variables. Figure 47.2 illustrates the concept. In this figure, part (a) shows that job 1 has a processing time of 3 (that may not be interrupted). We start with an initial domain of [0, 1, 2, 3, 4, 5, 6] for the variables C_1 (completion time of job 1) and s_1 (start time of job 1). In part (b), the constraint $C_1 = s_1 + 3$ reduces the domain of C_1 using the domain values of s_1, making arc $C_1 \leftarrow s_1$ consistent. In part (c), the constraint now reduces the domain of s_1 using the updated domain values of C_1, making arc $C_1 \rightarrow s_1$ consistent. This bi-directional arc consistency check ensures full reduction of domains of the two variables involved in this constraint.

47.3.2.2 Between-Constraint Domain Reduction

Constraint propagation also entails communication of information between constraints involving common variables to reduce their domains. Figure 47.3 illustrates the concept of constraint propagation across related constraints. In this case, the goal is to schedule two jobs, job 1 (described in Figure 47.2) and a second job (job 2) with processing time of 2 over the time domain [0, 1, 2, 3, 4, 5, 6] on a single-machine (assuming the disjunctive constraint that the machine can handle only one job at a time). In step 1, the domain of C_1, s_1, C_2, and s_2 are reduced with the help of arc consistency checking of the constraints binding C_1 to s_1 and C_2 to s_2, respectively. Let us say that there is an additional constraint on C_1 due to its due date ($C_1 < 5$). In step 2, this additional constraint reduces the domain of C_1 as shown ($C_1 = [3, 4]$). In step 3, this information about the updated domain of C_1 is communicated to the C_1 to s_1 constraint and the disjunctive constraint. The domain of s_1 is reduced as shown ($s_1 = [0, 1]$), and the domain of C_2 is reduced as shown ($C_2 = [5, 6]$). In step 4, the domain of s_2 is in turn reduced as shown ($s_2 = [3, 4]$). The sequence of steps illustrates the concept of sequential and full communication between the related constraints. Obviously, the actual sequence can be interchanged with the same final result (for example, step 3 and 4).

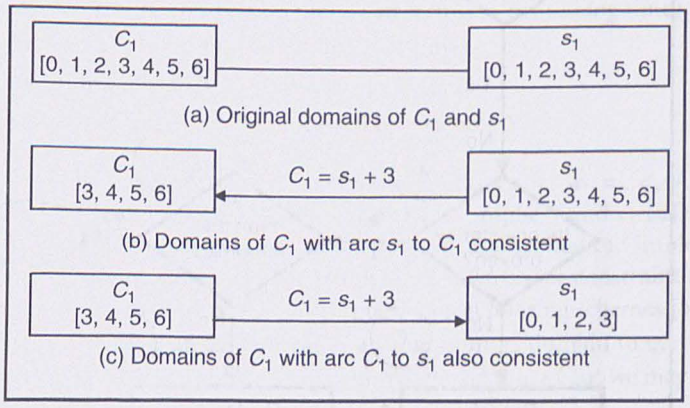

FIGURE 47.2 An illustration of domain reduction and are consistency checking logic.

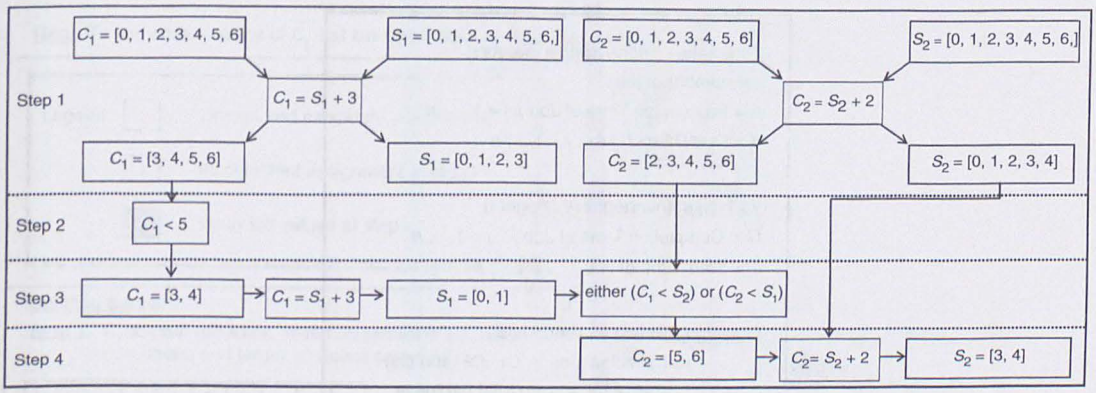

FIGURE 47.3 An illustration of constraint propagation logic.

47.3.3 Branching

Any branching strategy could be deployed in block 6 of Figure 47.1. Typically, a depth-first strategy is deployed but other more sophisticated look-ahead strategies could be used (see [4,6] for a more complete discussion). In any case, the decision of what to branch on is open as well. Often, we branch by first selecting a variable whose domain is not yet *bound* (reduced to a single value). Selection of the variable is often based on a *heuristic* (rule of thumb used to guide efficient search) such as "smallest current domain first." Once a variable is selected, the branch is established by *instantiating* the variable to one of the values in its current domain. Again, this choice could be made according to a heuristic like "smallest value in the current domain first." In fact, decision could be made to branch based on temporary constraint(s) involving one value of a variable, multiple values of the variable, multiple values of multiple variables. In CP jargon, the points at which the search strategy makes an advance along the search tree using one of these several choices are called *choice points*.

47.3.4 Adapting the CSP Algorithm to Constrained Optimization Problems

CSP problems can easily be extended to constrained optimization problems (COP). Suppose our COP has an objective Z to be minimized. If and when a first solution to the original CSP is found, its objective value is calculated (Z'), constraint $Z < Z'$ is added to the constraint store of the solved CSP to form a new CSP, and the CSP algorithm is repeated on this new problem. This process is repeated until inconsistency is found and all branches are fathomed. The last found solution is the optimum.

47.4 An Illustration of CP Formulation and Solution Logic

In this section we illustrate the concepts of CP formulation and solution logic with a simplified 3-job, single-machine scheduling problem with the constraint that all jobs are to be completed on or before their due dates and processed without interruption (time unit = day). The formulation of this problem is presented in Figure 47.4.

Note that the domains for the two variables (C_j and s_j) are finite sets. The start time (s_j) and completion time (C_j) for each job are linked by the processing time for the job (p_j). We express the fact that the machine can perform only one job at a time through the disjunctive constraints shown in the formulation.

The CP solution logic is presented in Figure 47.5. The figure maps the logic of domain reduction, constraint propagation, and depth-first search strategy in terms of the domain for the variables representing the completion times for the three jobs (C_1, C_2, and C_3). To track how CP evaluates and screens the solution space, at each step, the number of unexplored candidate values for C_1, C_2, and C_3 are provided. The values

Parameters: (nonnegative integers)
n = number of jobs
p_j = Processing Time of Job j; $j = 1,\ldots, n$
d_j = Due Date of Job j; $j = 1,\ldots, n$

Variables: (nonnegative integers)
C_j = Completion Time of Job j; $j = 1,\ldots, n$
s_j = Start Time of Job j; $j = 1,\ldots, n$

Objective: To find all schedules
(combinations of $C1$, $C2$, and $C3$)
that result in zero total tardiness

Variable Domains:
$p = p_1 + \cdots + p_n$
$C_j, s_j \in [0, 1,\ldots, p]$; $j = 1,\ldots, n$

Constraints:
$s_j = C_j - p_j$; $j = 1,\ldots, n$
$C_j \leq d_j$; $j = 1,\ldots, n$
$C_j \leq s_k \vee C_k \leq s_j$; $j, k = 1,\ldots, n; j \neq k$

FIGURE 47.4 CP formulation of a single-machine scheduling problem.

of the variables that are rendered inconsistent due to domain reduction induced through constraint propagation are shaded and labeled with the step at which they were rendered inconsistent. Finally, the bound values are shaded black and labeled with the step at which they were made. Note that, to start with, there are 343 ($7 \times 7 \times 7$) candidate assignments.

In step 1, domain reduction occurs on all the three variables individually with the constraints involving the completion time, processing time, start time, and the due date for each job. For example, because job 1 has a processing time of 3 days, it cannot be completed on days 0, 1, and 2. Also, job 1 is due on day 5. So, it cannot be completed on day 6. Note that the initial domain reduction results in 90 unexplored assignments (based on blank cells in Figure 47.5) after pruning 253 potential assignments.

Since no further domain reduction is possible and we do not have a solution, the depth-first search strategy is initiated in step 2. Using the search heuristic of "smallest current domain first", C_1 is chosen for instantiation. As soon as $C_1 = 3$ is executed, constraint propagation occurs with the effects shown on the domains of C_1, C_2 and C_3. C_1 can no longer assume values of 4 and 5. C_2 can no longer assume values of 2, 3, and 4. Finally, $C3$ can no longer assume values of 1, 2, and 3. Step 2 results in reduction of unexplored assignments from 90 to 6.

In step 3, C_2 is chosen to be instantiated again using the "smallest current domain first" heuristic. The first value of $C_2 = 5$ renders $C_2 = 6$ inconsistent, and also renders $C_3 = 4$ and $C_3 = 5$ inconsistent. Thus, in step 3, the search actually encounters the first feasible solution to this problem ($C_1 = 3, C_2 = 5, C_3 = 6$).

If we desire to identify all solutions, the depth-first search strategy now backtracks to step 2 for the next C_2 values (note that for $C_1 = 3, C_2$ could be either 5 or 6), and bounds C_2 to the next value in its current domain, namely, $C_2 = 6$. As soon as it instantiates $C_1 = 3$ and $C_2 = 6$, the propagation process now updates the domain of C_3 to see what values of C_3 might be consistent with these instantiations. It encounters the single consistent value of $C_3 = 4$, and identifies the second solution: $C_1 = 3, C_2 = 6, C_3 = 4$.

Backtracking, step 5 now reverts back to the C_1 level and instantiates it to $C_1 = 4$, and repeating the logic mentioned in steps 1 through 3, it identifies the third solution: $C_1 = 4, C_2 = 6, C_3 = 1$. Finally, instantiating $C_1 = 5$ yields the fourth (and last) solution: $C_1 = 5, C_2 = 2, C_3 = 6$.

Goal: To find combinations of C_j that satisfy all disjunctive constraints and result in zero total tardiness.

Legend:
- Unexplored candidate assignment
- t — Inconsistent assignment at step t
- t — Bounded values at step t

Data:

Job	p_j	d_j
1	3	5
2	2	6
3	1	8

Solution Steps:

Step 1: C_j domain reduction based on processing times, due dates, and target tardiness constraints.

Day	0	1	2	3	4	5	6
Job 1	1	1	1				1
Job 2	1	1					
Job 3	1						

Step 2: C_1 bound to 3.

Day	0	1	2	3	4	5	6
Job 1	1	1	1	2	2	2	1
Job 2	1	1	2	2	2		
Job 3	1	2	2	2			

Step 3: C_1 bound to 3; C_2 bound to 5 (=> C_3 = 6).

Day	0	1	2	3	4	5	6
Job 1	1	1	1	2	2	2	1
Job 2	1	1	2	2	2	3	3
Job 3	1	2	2	2	3	3	3

C_1 = 3; C_2 = 5; C_3 = 6 is the first solution.

Step 4: C_1 bound to 3; C_2 bound to 6 (=> C_3 = 4).

Day	0	1	2	3	4	5	6
Job 1	1	1	1	2	2	2	1
Job 2	1	1	2	2	3	4	4
Job 3	1	2	2	2	4	4	4

C_1 = 3; C_2 = 6; C_3 = 4 is the second solution.

Step 5: C_1 bound to 4 (=> C_2 = 6; C_3 = 1).

Day	0	1	2	3	4	5	6
Job 1	1	1	1	5	5	5	1
Job 2	1	1	5	5	5	5	5
Job 3	1	5	5	5	5	5	5

C_1 = 4; C_2 = 6; C_3 = 1 is the third solution.

Step 6: C_1 bound to 5 (=> C_2 = 2; C_3 = 6).

Day	0	1	2	3	4	5	6
Job 1	1	1	1	6	6	6	1
Job 2	1	1	6	6	6	6	6
Job 3	1	6	6	6	6	6	6

C_1 = 5; C_2 = 2; C_3 = 6 is the fourth solution.

FIGURE 47.5 CP logic for a single-machine, three-job scheduling problem.

47.5 Selected CP Applications in the Scheduling Literature

In this section we describe selected applications of CP to scheduling-related problems from recent studies in the literature. We review applications from a number of specific scheduling subject areas: job shop scheduling, single-machine sequencing, parallel machine scheduling, vehicle routing, and timetabling[2]. Finally, we review recent developments in the integration of the CP and IP paradigms.

[2]Throughout this section, numerous specific software products are referenced as the specific CP or IP software used to implement solutions. These products are:
ILOG Solver®, OPL Studio®, and CPLEX®(ILOG, Inc., Mountain View, CA);
OSL®– Optimization Solutions and Library – (IBM, Inc., Armonk, NY);
CHARME is a retired predecessor language of ILOG Solver;
ECLiPSe, free to researchers through Imperial College, London. http://www.icparc.ic.ac.uk/eclipse;
Friar Tuck, free software available at http://www.friartuck.net/
CPLEX and OSL are MIP software; all others are CP-based software.

47.5.1 Job Shop Scheduling

Brailsford, Potts, and Smith [4] reviewed the application of CP in the job shop scheduling environment. They described a general job shop scheduling problem, in which there are n jobs each with a set of operations which must be conducted in a specific order. The jobs are scheduled on m machines which have a capacity of one operation at any one time and the objective is to minimize the makespan. Brailsford et al. [4] noted that the problem instances of this type with as few as 15 jobs and 10 machines have not been solved with current branch and bound algorithms because these methods suffer from weak lower bounds which inhibit effective pruning of the branch and bound tree. Further, Brailsford et al. [4] concluded that "CP compares favorably with OR techniques in terms of ease of implementation and flexibility to add new constraints."

Nuijten and Aarts [7] addressed the problem described by Brailsford et al. [4] above. They provided a CP-based approach for solving these classic job shop problems and found that their search performed favorably compared with branch and bound algorithms when measured by solution speed. Their study leveraged the flexibility of CP search by incorporating randomization and restart mechanisms. Lustig and Puget [8] have given an instructive example of a simple job shop scheduling formulation in OPL Studio.

Darby-Dowman, Little, Mitra, and Zaffalon [9] applied CP to a generalized job assignment machine scheduling problem, in which different products with unique processing requirements are assigned to machines in order to minimize makespan. The problem is similar to Brailsford et al. [4], but in Darby-Dowman et al. [9], "machine cells" have a capacity greater than one. They found CP to be intuitive and compact in its formulation. They solved their CP problem using ECLiPSe and the IP version in CPLEX. They reported that CP performed more predictably than IP (when measured by search speed). CP's domain reduction approach reduced the search space quickly, contributing to its superior performance.

47.5.2 Single-Machine Sequencing

It is noteworthy that relatively little application of CP has taken place in the job sequencing area. The majority of this literature uses special-purpose codes to take advantage of the special problem structure of the job sequencing problem. A notable exception is Jordan and Drexl [10], which addressed the single-machine batch sequencing problem with sequence-dependent setup times. They assumed every job must be completed before its deadline, and all jobs are available at time zero. They solved this problem under a number of different objectives, including minimize setup costs, minimize earliness penalties, both setup and earliness costs combined, and finally, with no objective function (as a constraint satisfaction problem). They used CHARME to solve their CP formulation and compared computational results of the same model solved using OSL, an IP optimization software. They found that CP worked best when the machine was at high capacity (tight constraints; smaller feasible solution space), and IP worked best when the shop was at low capacity. Solution speeds are comparable for the two methods, but CP showed smaller variability in solution times overall. Both methods performed disappointingly on larger problems.

47.5.3 Parallel Machine Scheduling

Jain and Grossman [11] examined an application for assigning customer orders to a set of nonidentical parallel machines. Their objective was to minimize the processing cost of assignment of jobs to machines which have different costs and processing times for each job. Each job had a specific release (earliest start) and due (latest completion) dates. They found that for their problem, the CP formulation required approximately one-half the number of constraints and two-thirds the number of the variables of IP. They solved the CP problems using ILOG Solver, and the IP problems using CPLEX. They found that CP outperforms IP for smaller problems, and its performance is comparable to IP for larger problems.

47.5.4 Timetabling

Henz [12] utilized a CP methodology to schedule a double round-robin college basketball schedule in a minimal number of dates subject to a number of constraints imposed by the league. Henz [12] was able to model unusual constraints such as "no more than two away games in a row" and "no two final away games" in a direct way with CP. Henz [12] found a dramatic improvement in performance over Nemhauser and Trick [13]. Nemhauser and Trick [13] solved the problem in three phases using OR methods such as pattern generation, set generation and timetable generation in 24 hours of computer time. Henz [12] reported solution time under 1 min using Friar Tuck, a CP software written specifically for addressing tournament scheduling problems.

More recently, Baker, Magazine and Polak [14] and Valouxis and Housos [15] have explored school timetabling using CP. Baker et al. [14] created a multiyear timetable for courses sections in order to maximize contact among student cohorts over a minimum time horizon. They found CP straightforwardly reduces the size of the solution space, thereby facilitating the search for an optimum. Valouxis and Housos [15] used a combination of CP and IP to address a daily high school timetabling problem in which the teachers move to several different class sections during the day and the students remain in their classrooms. Their objective function was to minimize the idle hours between the daily teaching responsibilities of all the teachers while also attempting to satisfy their requests for early or late shift assignments. They developed a hybrid search method for improving the upper bounding which helped reduce the search space and facilitated faster solution speed.

47.5.5 Vehicle Dispatching

The synchronized vehicle dispatching problem of Rousseau, Gendreau and Pesant [16] requires coordinating a number of vehicles such as ambulances with complementary resources such as medics to service demands such as patients. These resources are coordinated across time intervals in order to minimize the travel cost of the vehicles. Their model is a real-time model, with customer orders arriving sporadically even after vehicles have been dispatched (all orders are not known at the beginning of the time horizon). New customers are inserted as constraints into the model which is then resolved based on the new conditions. Rousseau et al. found modeling the complicated nature of the synchronization constraints is simplified in a CP environment. Further, because of the real-time nature of customer arrivals, insertion of additional customer constraints (orders) was found to be easy in the CP paradigm.

47.5.6 Integration of the CP and IP Paradigms

One of the most active and potentially highest-opportunity areas of recent research is in the exploration of hybrid methods for CP and IP search methodologies. It is widely held that CP and IP have complementary strengths (e.g., see [11,17]). CP's strengths arise from its flexibility in modeling, a rich set of operators, and domain reduction in combinatorial problems. IP offers specialized search techniques such as relaxation, cutting planes and duals for specific mathematical problem structures. Hooker [17], Milano, Ottoson, Refalo, and Thorsteinsson [18] and Brailsford et al. [4] have provided general discussions on developing a framework for integrating CP and IP.

There are recent studies which integrate CP and IP for solving specific problems. Darby-Dowman et al. [9] made an early attempt at integrating IP and CP by using CP as a preprocessor to limit the search space for IP, but had limited success. They suggested, but did not develop, using a more integrated combination of IP and CP: CP for generating cuts on the tree and IP for determining search direction. Jain and Grossman [11] presented a successful deeper integration of these methods for the machine scheduling problem based on a relaxation and decomposition approach. Finally, Valouxis and Housos [15] used local search techniques from OR to tighten the constraints for CP and improve the search time. If efforts at integrating these methods are successful, the end state for these methods may be that researchers view IP and CP as special case algorithms for a general problem type.

47.6 The Richness of CP for Modeling Scheduling Problems

As a general framework for developing search strategies, CP has a rich set of operators and variable types, which often allows for a succinct and intuitive formulation of scheduling problems. The following section describes the richness of the CP modeling environment with examples of variable indexing, and constraints such as strict inequality, logical constraints, and global constraints.

47.6.1 Variable Indexing

Just as in other computer programming languages, CP allows "variable indexing", in which one variable can be used as an index into another. This capability creates an economy in the formulation which reduces the number of required decision variables in the scheduling problem formulation. The result is often a more compact and intuitive expression of the problem.

To illustrate the value of variable indexing capability, an adaptation of single-machine batch sequencing problem with sequence-dependent setup costs is taken from Jordan and Drexl [10]. In this problem, the objective is to minimize setup costs and earliness penalties given varying setup costs for adjacent jobs in the sequence. In IP, a binary variable, $y[i, j]$, is used to indicate if job i immediately precedes job j. If n is the total number of jobs in the problem, with this formulation, there are $n(n-1)$ binary indicator variables to express the potential job sequences. If setup cost $[i, j]$ is the cost of setup for job i immediately preceding job j, then the total setup cost for the set of assignments is specified as the sum of setup cost $[i, j]y[i, j]$ for all i and $j, i \neq j$.

To formulate the problem in CP, we create the decision variable, job$[k]$ which takes the label of the job assigned to position k in the sequence. We then use job$[k]$ as an indexing variable to calculate the total setup cost of the assignments as the sum of setup cost[job$[k-1]$, job$[k]$] for all $k > 1$. With this indexing capability, the number of variables in the problem is reduced to the number of jobs, n. An additional benefit of variables indexing is that the model is more expressive of the original problem statement. The solution to the problem is more naturally thought of as "the sequence of jobs on the machine" as in CP, than, "the collection of immediately adjacent jobs on a machine", as is the case in the IP formulation.

Another example of the usefulness of variable indexing in the job shop literature comes from Darby-Dowman et al. [9]. They described a generalized job assignment problem that IP treats with mn decision variables (m machines, n jobs), but CP addresses with only n decision variables (jobs), which take on the value of the machine to which a job is assigned. In this setting as well, the flexibility of CP's variable indexing construct more intuitively and compactly expresses the problem as an assignment of jobs to machines, rather than IP's expression of a machine-job pair.

Generally stated, problems that can be expressed as a matching of one set to another can be more succinctly expressed using an index into one of the sets (as afforded in CP) than using a binary variable to represent every possible combination of the cross product of the two sets (as is the case in IP).

47.6.2 Constraints

CP allows for the use of a number of operators such as set operators and logical conditions which enable it to handle a variety of special constraints easily. In this section, we describe some of these special constraints that are common in scheduling: strict inequality, logical constraints, and global constraints, which, as Williams and Wilson [5] describe, are straight forward in CP, but difficult in IP.

47.6.2.1 Inequalities with Boolean Variables

We start with the simple example of strict inequality constraint for Boolean variables. Let us say in a scheduling application, the assignment of a job to a pair of machines can be expressed as Boolean variables (call them MachineA and MachineB) which take the value of 1 if a job is assigned, and the value of 0 if no job is assigned. Further, let us say that it is desirable for either one of two machines to have a job assigned, but not both, or equivalently that one Boolean variable does not equal another. We express, the inequality

directly in CP as

$$\text{MachineA} \neq \text{MachineB}$$

In IP, we specify the same constraint as:

$$\text{MachineA} + \text{MachineB} = 1$$

Both formulations accurately and efficiently impose the condition in a single constraint that either MachineA or MachineB receives a job (equals one), but not both. However, the CP statement is a more intuitive and direct expression of the naturally occurring constraint without the indirection necessary as in the IP formulation to express the constraint as the sum of two variables.

47.6.2.2 Inequalities with Integer Variables

Now we consider another situation in which MachineA and MachineB are integer variables representing the job number assigned to each machine. Further, we assume the same job cannot be assigned to both MachineA and MachineB. In CP, the constraint is again expressed using the simple constraint:

$$\text{MachineA} \neq \text{MachineB}$$

However, in IP, the $\neq$ condition on integer variables must be modeled as two inequalities ($>$, $<$) and an exclusive or (Xor) condition:

$$(\text{MachineA} > \text{MachineB}) \text{ Xor } (\text{MachineB} > \text{MachineA})$$

Further, in IP we are restricted to $\geq$ and $\leq$ constraints. So in order to capture the essence of the strict inequality, we add or subtract some small value, ε, to the integer values:

$$(\text{MachineA} \geq \text{MachineB} + \varepsilon) \text{ Xor } (\text{MachineB} \geq \text{MachineA} + \varepsilon)$$

Finally, in IP the logical Xor is modeled with a Boolean indicator variable, δ for each constraint, and a "BigM" multiplier is used to ensure that at least one but not both constraints hold:

$$\text{MachineA} - \text{MachineB} - \varepsilon + \delta\text{BigM} \geq 0$$
$$\text{MachineB} - \text{MachineA} - \varepsilon + (1 - \delta)\,\text{BigM} \geq 0$$

In this example, $\delta = 1$ implies MachineB $>$ MachineA; $\delta = 0$ implies MachineA $>$ MachineB. Familiar and widely-used constructs such as "BigM" and Boolean indicator variables have been borne out of necessity to create formulations which meet the linear problem structure requirements for IP.

47.6.2.3 Logical Constraints

Logical constraints are common in scheduling. The "precedes" condition is a ubiquitous example from sequencing. For example, let us say that the completion of JobA must precede the start of JobB or that the completion of JobB must precede the start of JobA, but not both. This is an example of the exclusive or (Xor) constraint. CP handles constraints of this form with relative ease with constraints of the form:

$$(\text{A.start} > \text{B.end}) \text{ Xor } (\text{B.start} > \text{A.end})$$

In IP, the relationship is stated as follows:

$$\text{A.start} - \text{B.end} - \varepsilon + \delta\text{BigM} \geq 0$$
$$\text{B.start} - \text{A.end} - \varepsilon + (1 - \delta)\,\text{BigM} \geq 0$$

where $\delta = 1$ implies job A is first and $\delta = 0$ implies job B is first.

CP expresses the Xor condition directly and naturally. It is interesting to note that because the $\neq$ condition and the Xor condition are logically equivalent, both are modeled in the same way in IP, even though the basic problem statements (precedes and $\neq$) are perceived differently.

47.6.2.4 Global Constraints

Global constraints apply across all, or a large subset of, the variables. For example, imagine we are assigning jobs to m machines, and it is desired to assign different jobs to each machine in an efficient way. We may want to extend the $\neq$ constraint discussed above to apply to all of the machines under consideration. In CP, the alldifferent constraint is used:

alldifferent(Machine).

This constraint assures that no two machines are assigned to the same job. It should be noted that the alldifferent constraint in CP is a single, global constraint, and that a constraint of this form has stronger propagation properties than n inequality constraints (see Hooker, 2002).

In IP, the alldifferent concept is implemented as an inequality constraint (as described above) for every machine pair. For m machines, this would require $m(m-1)$ constraints, and $m(m-1)/2$ indicator variables.

Other global constraint constructs exist in CP. For the preceding example, we may want to assure that each machine gets no more than one job in the final solution. In CP, we use the "distribute(Machine)" constraint — a global constraint restricting the count of jobs on each machine to one. In IP, we create an indicator variable and accompanying constraint for each machine which indicates if it has a job assigned or not, then, create a constraint on the sum of the indicator variables.

The need for the indicator variables and "BigM" constructs is obviated in the more general modeling framework of CP. Because the problem can be directly and succinctly stated, accurate formulations are easily created, maintained, modified and interpreted, and the potential for errant formulations are reduced.

Other special-purpose CP constructs have been developed to aid in model building that are useful to the scheduling community. For examples, ILOG's OPL includes the "cumulative" constraint, which defines the maximum resource availability for some time period, and "reservoir" constraints, which capture resources that can be stockpiled and replenished (such as raw materials, funds, or inventory). These constructs are special-purpose routines that take advantage of the known properties of the scheduling problem in order to represent resources accurately and leverage their properties in order to reduce domains more rapidly.

47.7 Some Insights into the CP Approach

We do not want to encourage discussion on "which is better", CP or IP. Which performs better depends on a specific problem, the data, the problem size, the researcher, the commercial software and the model of the problem (among other things!). In any case, the debate on which is better, CP or IP, may be moot because the two are so different. IP is a well-defined solution methodology based on particular mathematical structures and search algorithms. CP, on the other hand, is a method of modeling using a logic-based programming language (see [8]). Also, as covered previously, the two approaches have complementary strengths that can be combined in many ways (see [11,17]). Despite their orthogonality in approach, in this section we compare and contrast CP and IP as approaches to solving scheduling problems, and provide some insight into when CP may be an appropriate method to consider.

47.7.1 Contrasting the CP and IP Approaches to Problem Solving

When applied to scheduling-related problems, CP and IP both attempt to solve NP-hard combinatorial optimization problems; however, they have different approaches for addressing them. IP leverages the mathematical structure of the problem. In the CP approach, the freedom and responsibility for creating

constraints and a search strategy falls primarily to the researcher. This is similar to the requirement of developing a good IP model that fits the linear structure required of an IP solver. CP search methods rely less on particular mathematical structure of the objective function and constraints, but more on the domain knowledge of specific aspects of the problem. Consider the trivial example of the single-machine job sequencing problem to minimize total tardiness. An IP formulation would not likely contain an explicit constraint that states there need be no gaps between jobs on a machine; it is rather a logical outcome of the optimization process. Using the CP paradigm, the researcher who knows, logically, the optimal solution has no such gaps might utilize this knowledge by introducing an explicit constraint to that effect, possibly reducing the solution space and improving performance of CP search.

In general, IP is objective-centric while CP is constraint-centric. IP follows a "generate-and-test strategy" evaluating the objective function for various values of the decision variables. CP focuses more on the constraints in the problem, constantly reevaluating the logical conclusions resulting from the interactions of the constraints and, as a result, reducing the domain of feasible solutions.

Another difference between CP and IP falls in what might be called "modeling philosophy". In IP the modeler is encouraged to specify a minimal set of constraints sufficient to capture the essence of a given problem. Fewer constraints are generally viewed as "better" or more elegant (for examples, see [19, p. 194] and [20, p. 34]). In CP, the modeling philosophy is somewhat different. The modeler is encouraged to add constraints that more carefully capture the nuances of a particular problem. More constraints are better, because taken together they reduce the solution domain. Contrary to the IP approach, even redundant constraints are considered desirable in CP in some cases if they improve constraint propagation and domain reduction.

47.7.2 Appropriateness of CP for Scheduling Problems

There are certain attributes of problems that researchers can be aware of when deciding if CP is an appropriate methodology to employ for solving scheduling problems. First, CP is most appropriate for pure integer, or Boolean decision variable problems; CP methods are not effective for floating point decision variables. CP is more successful reducing domains for variables with finite domains at the outset.

Second, CP is well suited for combinatorial optimization problems that tend to have a large number of logical, global and disjunctive constraints that CP handles well with its rich set of operators and special-purpose constraints. CP more naturally expresses variable relationships in combinatorial optimization problems. It can be useful for quickly formulating these types of problems without significant formulation difficulties.

Third, CP operates more efficiently when there are a large number of interrelated constraints with relatively few variables in each constraint. This characteristic results in better constraint propagation and domain reduction, and better performance of CP algorithms. For example, if a constraint is stated as the sum($x1$ to $x100$) < 1000, and $x1$ is instantiated, not much can be said about $x2$ to $x100$ because there are so many variables in the constraint. On the other hand, if $x1 + x2 < 100$, and $x1$ is bound to 50, then it follows immediately that $x2 < 50$, and the domain of $x2$ is significantly reduced. Further, if $x2$ is in another constraint with only $x3$, then $x2$'s domain reduction propagates immediately to $x3$'s domain reduction. Thus, a problem with a large number of interrelated constraints with few variables in each constraint tends to be well suited for CP.

Finally, in a more general sense, CP applies well to any problem that can be viewed as an optimal mapping of one ordered set to another ordered set, where the relation between variables in each set can be expressed in mathematical terms that are significant to the objective. Scheduling problems fit into this description. An example may be a set of operations with precedence conditions being assigned to machines over time-indexed periods. Both sets are ordered in some way, and the solution is a mapping between them. The decision variables are integer or Boolean, constraints tend to be logical, global or disjunctive in nature, and there is a large number of constraints that often contain few variables. The strengths of CP seem well suited for application in scheduling research.

47.8 Formulating and Solving Scheduling Problems via CP

We now illustrate how CP might be useful in modeling scheduling problems. We will use the well-known single-machine weighted tardiness problem as a specific case in point. In the notation of Pinedo [21] this is described as $1||\Sigma w_j T_j$. The $1||\Sigma w_j T_j$ problem is simple to specify but well known to be NP-hard (see [22]). It well serves the purpose of showing the look and feel of a CP approach without excessive clutter. Typically $1||\Sigma w_j T_j$ is solved by special purpose branch-and-bound algorithms implemented in a high level general purpose programming language such as C++. See for example [23]. Our illustration will show several ways the problem can be formulated via CP. In doing this we will use the commercially available CP modeling language OPL as implemented in ILOG OPL Studio. For details regarding the OPL programming language the reader is directed to Van Hentenryck [24].

An OPL program is commonly comprised of three major program blocks including:

- a Declarations/Initializations block for declaring data structures, initializing, and reading data
- an Optimize/Solve block for specifying, in a declarative mode, the problem specifications (either a constraint satisfaction or an optimization problem)
- an optional Search block for specifying how the search is to be conducted

The absence of any search specification invokes the OPL default search (to be described below).

47.8.1 A Basic Formulation

Figure 47.6 shows a fairly compact encoding of $1||\Sigma w_j T_j$ in OPL. We refer to it hereafter as Model 1.

Lines 2 to 5 in the Declarations/Initializations block define and initialize the number of jobs, processing times, due dates and job weights (n, $p[.], d[.], w[.]$). The use of "int+" and "float+" specify nonnegative integer and floating point data types, respectively. The notation "= ...;" signifies that data is to be read from an accompanying data file. Lines 6 and 7 define the two nonnegative integer variables $C[.]$, and $s[.]$. Since weighted tardiness is regular, we can ignore schedules with inserted idle time (see, e.g., Baker, 2000, p. 2.4). We take advantage of this fact by limiting the domains for $s[.]$ and $C[.]$ to $\{0, 1, \ldots, \Sigma_{j\in 1,\ldots,n} p[j]\}$.

Lines 9 to 15 of Figure 47.6 constitute the Optimize/Solve block. The presence of the keyword "minimize" signals that this is a minimization problem and that the associated objective follows. The keywords "subject to" introduce the so-called "constraint store." The constraint in line 13 assures the desired relation between $s[.]$ and $C[.]$. We could as well have specified $C[j] - s[j] >= p[j]$. However, the equality specification yields a smaller search space. Lines 13 and 14 in Figure 47.6 together define the disjunctive requirement that for all pairs of jobs j, k that either j precedes k or k precedes j but not both.

```
01 //SINGLE MACHINE WEIGHTED TARDINESS SCHEDULING MODEL 1
02 int + n= ... ;//n is the number of jobs
03 int + p[1 .. n] = ... ;//p[i] is the processing time for job i
04 int + d[1 .. n]= ... ;//d[i] is the due date for job i
05 float + w[1 .. n]= ... ;//w[i] is the weight for job i
06 var int + C[j in 1 .. n] in 0 .. sum(j in 1 .. n) p[j];//C[i] is the completion time for job i
07 var int + s[j in 1 .. n] in 0 .. sum(j in 1 .. n) p[j];//s[i] is the start time for job i
08 //END OF DECLARATIONS/INITIALIZATIONS BLOCK
09 minimize
10 sum(j in 1 .. n) w[j]*maxl(0,C[j]-d[j])
11 subject to{
12 forall(j in 1 .. n){
13 C[j]-s[j] = p[j];
14 forall(k in 1 .. n: k>j) (C[j] <= s[k]) V (C[k] <= s[j]);
15 };
16 };//END OF OPTIMIZE/SOLVE BLOCK
```

FIGURE 47.6 Example OPL program for $1||\Sigma w_j T_j$ (Model 1).

TABLE 47.1 Sample $1||\Sigma w_j T_j$ Problem (Elmaghraby, 1968)

Job	Processing Time	Due Date	Weight
1	3	2	1
2	3	5	3
3	2	6	4
4	1	8	1
5	5	10	2
6	4	15	3
7	4	17	1.5

TABLE 47.2 OPL Solution Using Model 1 on Elmaghraby Problem Data

Optimal Solution with Objective Value: 25.0000	
$s[1] = 19$	$C[1] = 22$
$s[2] = 0$	$C[2] = 3$
$s[3] = 3$	$C[3] = 5$
$s[4] = 5$	$C[4] = 6$
$s[5] = 6$	$C[5] = 11$
$s[6] = 11$	$C[6] = 15$
$s[7] = 15$	$C[7] = 19$

Since Model 1 provides no search specifications the default search in OPL is used. It works as follows. All possible domain reduction is first accomplished. If a solution does not result, then a depth-first search ensues. Variables are chosen for instantiation in the order of smallest domain size first. Values within the domain of a variable are chosen in order of smallest value first. Each instantiation represents a choice point.

As can be observed, Model 1 requires $2n$ variables and $n(n+1)/2$ constraints. We executed Model 1 on the 7-job instance of $1||\Sigma w_j T_j$ found in Elmaghraby [25] and depicted in Table 47.1.

Using OPL Studio we arrived at the solution in Table 47.2 after generating 254 choice points.

47.8.2 A Second Formulation

A second formulation (called Model 2) for $1||\Sigma w_j T_j$ is provided in Figure 47.7. Here the variable $s[.]$ is replaced with the variable position[.] to represent the position of job j in the sequence. The constraint in line 12 assures that for all combinations of job pairs the position numbers are different. Line 13 binds the two variables position[.] and $C[.]$ by specifying an equivalence relation for all permutations of job pairs. Line 14 is required to assure the job j in position 1 is completed at time $t = p[j]$. The formulation is relatively compact ($2n$ variables, $(3n^2 - n)/2$ constraints), but the performance is lackluster. Running this model for the sample Elmaghraby data causes 5391 choice points to be created. The constraint set as specified is sufficient to assure solution, but the filtering algorithms are not strong enough to enable much domain reduction as in Model 1. We begin to see with this example that model building in CP is, to a great degree, a craft. The goal is not to minimally express a problem specification but rather to express as much information in the constraint store so as to foster domain reduction.

47.8.3 Strengthening the Constraint Store

A first improvement to the Model 2 is accomplished by replacing line 12 with the so-called "all different" constraint; i.e., change line 12 to read "alldifferent(position);". This is a good example of a single global constraint as described earlier. The alldifferent constraint operates at once on the entire set of variables,

```
01 //SINGLE MACHINE WEIGHTED TARDINESS SCHEDULING MODEL 2
02 int + n = ... ;//n is the number of jobs
03 int + p[1 .. n] = ... ; //p[i] is the processing time for job i
04 int + d[1 .. n] =... ;//d[i] is the due date for job i
05 float + w[1 .. n] = ... ;//w[i] is the "weight" for job i
06 var int + C[j in 1 .. n] in 0 .. sum (i in 1 .. n)p[i]; //C[i] is the completion time for job i
07 var int + position[j in 1 .. n] in 1 .. n; //position[i] is job i's position in the sequence
08 //END OF DECLARATIONS/INITIALIZATIONS BLOCK
09 minimize
10 sum(j in 1 .. n) w[j]*maxl(0,C[j]-d[j])
11 subject to{
12 forall(j, k in 1 .. n: k>j) position[j] <> position[k];
13 forall (j, k in 1 .. n: j<>k) position[j] > position[k] <=> C[k] <= C[j]-p[j];
14 forall (j in 1 .. n) position[j] = 1 => C[j] = p[j];
15 };//END OF OPTIMIZE/SOLVE BLOCK
```

FIGURE 47.7 Example OPL program for $1||\Sigma w_j T_j$ (Model 2).

and is considered a single constraint. Making this replacement in the code reduces the number of choice points for the sample problem from 5391 to 3953.

We can improve the performance of Model 2 further by adding more detailed information regarding the relationship between position and completion time. The equivalence constraints of line 13 can be made stronger when jobs are adjacent in the schedule, for then the difference in their completion times is exactly the processing time of the first job in the pair. We implement this knowledge by adding the following adjacency constraints to the Model 2 formulation.

$$\text{forall}\,(j, k \text{ in } 1..n\text{: } j <> k)\ \text{position}[j] = \text{position}[k] + 1 <=> C[k] = C[j] - p[j]$$

We applied this additional constraint set to Model 2 and applied the new model to the Elmaghraby sample data. The result was a further reduction in the number of choice points from 3953 to 1969.

47.8.4 A Final Improvement to Model 2

We can improve Model 2 further by adding problem-specific domain knowledge to the formulation. For example, note that when a job occupies position j we can place a lower bound on its completion time, namely the sum of its processing time plus the sum of the remaining $j-1$ jobs with smallest processing times. A revised Model 2 is provided in Figure 47.8 which includes this idea along with all the other aforementioned revisions to Model 2. The set of constraints bounding the jobs' minimum completion times is implemented in lines 19 to 24. Running this model on the sample problem data drastically reduces the number of choice points further from 1969 to 50. Note that in this case for simplicity we assume the jobs are numbered in order of nondecreasing processing time so the comparison in performance is not 100% fair. Nevertheless, it vividly illustrates the rather craft-like aspect of model building for scheduling using CP.

47.8.5 Utilizing the Special Scheduling Objects of OPL

Up to this point we have not taken advantage of the specialized scheduling objects embedded in the OPL modeling language. Objects like "activity" and "resource" can be used to relieve the modeler from some tedium as well as to take advantage of a number of built-in functions peculiar to scheduling applications. We illustrate with another formulation of $1||\Sigma w_j T_j$ as depicted in Figure 47.9 and refer to it as Model 3. Here line 6 defines the special object "scheduleHorizon" which is used to limit the domain of the search space. Line 7 declares a set of variable structures of type "Activity" with durations $p[.]$. Line 8 defines the variable

```
01 //SCHEDULING MODEL 2 (revised)
02 int + n= ... ;//n is the number of jobs
03 nt + p[1 .. n] = ... ;//p[i] is the processing time for job i
04 int + d[1 .. n]= ... ;//d[i] is the due date for job i
05 float + w[1 .. n]= ... ;//w[i] is the "weight" for job i
06 var int + C[j in 1 .. n] in 0 .. sum (i in 1 .. n)p[i]; //C[i] is the completion time for job i
07 var int + position[j in 1 .. n] in 1 .. n; //Job position number in sequence
08 //END OF DECLARATIONS/INITIALIZATIONS BLOCK
09 minimize
10 sum(j in 1 .. n) w[j]*maxl(0,C[j]-d[j])
11 subject to{
12 forall (j in 1 .. n) position[j] = 1 => C[j] = p[j];
13 alldifferent(position);
14 forall (j, k in 1 .. n: j<>k){
15 position[j] > position[k] <=> C[k] <= C[j]-p[j];
16 position[j] = position[k]+1 <=> C[k] = C[j]-p[j];
17 };
18 //NOTE: The following requires job numbers in non-decreasing processing time order
19 forall (j in 1 .. n){
20 forall (k in 1 .. n){
21 position[j]=k & j<=k => C[j] >= sum(l in 1 .. n: l<=k) p[l];
22 position[j]=k & j>k  => C[j] >= p[j]+sum(l in 1 .. n: l<k) p[l];
23 };
24 };
25 };//END OF OPTIMIZE/SOLVE BLOCK
```

FIGURE 47.8 Example OPL program for $1||\Sigma w_j T_j$ (Model 2 with all revisions).

```
01 //SINGLE MACHINE WEIGHTED TARDINESS SCHEDULING MODEL 3
02 int + n= ... ;//n is the number of jobs
03 int + p[1 .. n] = ... ;//p[i] is the processing time for job i
04 int + d[1 .. n]= ... ;//d[i] is the due date for job i
05 float + w[1 .. n]= ... ;//w[i] is the weight for job i
06 scheduleHorizon = sum(j in 1 .. n) p[j];
07 Activity job[j in 1 .. n](p[j]);
08 UnaryResource Machine;
09 //END OF DECLARATIONS/INITIALIZATIONS BLOCK
10 minimize sum(j in 1 .. n) w[j]*maxl(0,job[j].end-d[j])
11 subject to{
12 forall(j in 1 .. n) job[j] requires Machine;
13 };//END OF OPTIMIZE/SOLVE BLOCK
```

FIGURE 47.9 Example OPL program for $1||\Sigma w_j T_j$ (Model 3).

"Machine" as a "UnaryResource" (i.e., one that can service only one activity at a time). Line 12 specifies the constraint that each job (Activity) is to be processed on "Machine." Built in to the UnaryResource type along with the special constraint predicate "requires" is the assurance of the disjunctive constraint that only one job can occupy Machine at a given time. Note that, as with the other models presented earlier, no search block is provided so that OPL invokes the default search algorithm as needed. This rather compact formulation when run using the Elmaghraby sample data yields a solution after considering 117 choice points. We can attribute this relatively good performance to using OPL's special scheduling objects. Their use triggers special purpose filtering algorithms leading to more efficient domain reduction. Of course we could now commence as before with attempts to improve the performance by the addition of problem domain specific knowledge (more constraints).

```
14 search{
15   while(not isRanked(Machine)) do
16     select (j in 1.. n: isPossibleFirst(Machine,job[j])
17         ordered by increasing  maxl(d[j]-dmin(job[j].start), p[j])/w[j])
18          tryRankFirst(Machine,job[j]);
19 };
```

FIGURE 47.10 Example OPL search block for $1||\Sigma w_j T_j$ (Model 3).

47.8.6 Controlling the Search

OPL allows the modeler significant freedom in designing a search strategy. We provide a few simple examples to illustrate. In solving minimization problems with objective function Z, OPL has the aforementioned feature of adding the constraint $Z < z'$ to the constraint store where z' is always the objective value of the current best found solution. Suppose for $1||\Sigma w_j T_j$ we wish to steer the search to discover a good trial solution early on. We might be motivated then to make use of a good heuristic. A recent paper by Kanet and Li [26] reports on a heuristic dispatching rule "weighted modified due date" (WMDD) for $1||\Sigma w_j T_j$. At time $t = 0$, it computes for each job j the following value:

$$\text{WMDD}[j] = \max\{d[j] - t, p[j]\}/w[j]$$

It then selects the job k with smallest WMDD to occupy the machine. At time $t = t + p[k]$ the process is repeated on the remaining jobs so that a schedule is constructed from beginning to end. With OPL we can create a search strategy, which first constructs a schedule from beginning to end, plunging to a complete solution in conformance to WMDD, and then backtracks. To illustrate how we might implement this in OPL, we append a search block to Model 3; the OPL specification for which is depicted in Figure 47.10. The functionality of the various OPL keywords is almost self-explanatory. In OPL, a unary resource is said to be ranked when a permutation in which it services activities is completely specified. The keyword "dmin" is a "reflective" function that returns the current minimum value of the domain for its argument. In this case the argument is the variable job[j].start. This is what affords the building of the schedule from start to finish and the intended dynamic recalculation of WMDD. With each execution of line 18 another job is appended to the end of the schedule. Doing so clips the domains of the start times for the remaining unscheduled jobs accordingly so that on the next call to dmin the WMDD calculation in line 17 is dynamic. Running this version of the model on the sample Elmaghraby data produces the desired result; the number of choice points drops from 117 to 37.

From the previous examples we see that it is quite simple using OPL to organize the search either over job completion times or over jobs' positions. Alternatively, we might want to search over the jobs that occupy the different positions. To clarify, consider a variable job[j] to represent the job occupying position j in the sequence. After the appropriate declaration we need only the following two lines in the optimize/solve block.

```
alldifferent(job);
forall(j in 1..n) C[job[j]] = sum(k in 1..n : k <= j)p[job[k]];
```

The second constraint set assures that the completion time for a job occupying position number j is the sum of job completion times through j (and illustrates the use of indexing variables described earlier).

In CP, controlling the organization of the search space is even more flexible than suggested above. As described earlier, instead of branching on the different values of a variable within its domain, we might wish to branch on a specific condition (i.e., insert a constraint or set of constraints). Upon backtracking we introduce its negation. In the aforementioned Model 2, for example, we might wish to define a binary search where at each node in the search tree we choose, for some job pair (j, k), either to have position[j] < position[k] or position[k] < position[j]; i.e., at each choice point we first introduce the

constraint position[j] < position[k], then upon backtracking introduce position[j] > position[k]. A simple OPL implementation of this would look like the following.

```
search{
  forall (i in 1..n-1)
    try position[i] < position[i+1] } position[i] > position[i+1] endtry;
};
```

In addition to controlling the organization of the search space, OPL offers the modeler several choices for the basic branching strategy. Although depth-first search is the default search strategy, OPL offers a number of other choices including a best-first strategy (commonly used in branch-and-bound codes for scheduling).

47.9 Concluding Remarks

47.9.1 CP and Scheduling Problems

We have argued that CP has good application to problems rife with logical constraints, particularly problems with many constraints involving few variables, because this affords numerous interactions inducing an abundance of domain reduction. Is this an inherent property of scheduling problems? We would so argue. Consider the definition of scheduling offered by Baker ([27], p. 2): "*Scheduling is the allocation of resources over time to perform a collection of tasks.*" Were it not for the little phrase "over time" then scheduling problems might not present the challenge that they do. It is this little phrase that begets the logical connections between the allocations. For example, consider the case of unary resources. Here the implication of the phrase "over time" means that no two jobs can be allocated in the same time interval — a set of two-variable (binary) constraints. For the case of sequence-dependent setup times the phrase "over time" affects pairs of chronologically adjacent allocations (another set of binary constraints). As another example consider the problem of assigning basketball referee crews to a league schedule of games. One constraint that makes this assignment problem a scheduling problem is the obvious requirement that a given crew may not be allocated to two different games that are scheduled to be played at the same time. It is the phrase "over time" which makes scheduling problems rife with logical conditions and constraints with few variables and thus amenable to the CP paradigm.

47.9.2 The Art of Constraint Programming

We have defined CP as a method for formulating and solving discrete variable constraint satisfaction/constrained optimization problems and have highlighted its reliance on logic-based computer programming. As such, CP involves choosing variables and data structures, representing the relations between these entities in a constraint store, and designing the search strategy that may ensue.

Employing CP for scheduling problems is a craft involving several interrelated skills perhaps not so customary to operations researchers. CP involves modeling skills for capturing relations (and including them in the constraint store) about the nature of the problem so as to enhance constraint propagation and domain reduction. Operations researchers have traditionally been trained to believe that, when formulating integer programs, expressing problems with as few variables and constraints as possible is generally better since it often leads to smaller memory requirements, faster solutions, and means for a cleaner, more elegant — more efficient formulation.[3] In CP, a compact formulation is not necessarily a good one. The goal is not to minimally represent the problem in terms of variable and constraint definitions but to use knowledge about the nature of the solution to fortify the constraint store. The discussion in the previous section

[3]However, as Williams (1999) points out with the prevalence of "presolve" algorithms in commercial solvers today, the modeler can afford to be more verbose in his modeling, without losing computational efficiency.

where the adjacency constraints were added to Model 2 illustrates this point. Although these constraints were unnecessary in terms of expressing a correct model specification, their introduction served to boost the deductive power of the constraint store and induce more domain reduction.

A second related skill in the craft of CP for scheduling is the ability to embed scheduling knowledge into the constraint set or into the design of the search strategy. For almost a half century there has been a steady stream of advances in the science of scheduling resulting in a great base of knowledge in the form of theorems and algorithms for specific scheduling problems. For scheduling problems we often know many pieces of information regarding the nature of optimum solutions, i.e., sufficient (but not necessary) conditions for optimality. For example, there are a number of precedence theorems now collected in the scheduling literature for the $1||\Sigma w_j T_j$ problem of the previous section [25,28–30]. Such theorems (scheduling domain knowledge) take the form "*If <condition> then there exists an optimum schedule in which job a precedes job b.*" Similarly there is a wealth of knowledge in scheduling about heuristic rules with empirical evidence to show they provide good results. The WMDD dispatching heuristic for $1||\Sigma w_j T_j$ described and illustrated in the previous section is a good example. Our experience in working with CP is that such scheduling-specific domain knowledge is relatively easy to implement within the CP framework. So we can look to CP as a tool that complements scheduling algorithmic knowledge by serving as a vehicle for its easy implementation. We see this as a trend, a trend that will undoubtedly be nurtured by further development and more widespread availability of CP tools and familiarity of operations researchers to the CP paradigm.

References

[1] Baptiste, P., Le Pape, C., and Nuijten, W., *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems*, Kluwer Academic Publishers, Boston, MA, 2001.

[2] Tsang, E., *Foundations of Constraint Satisfaction*, Wiley, Chichester, England, 1994.

[3] Van Hentenryck, P., *Constraint Satisfaction in Logic Programming*, MIT Press, Cambridge, MA, 1989.

[4] Brailsford, S.C., Potts, C.N., and Smith, B.M., Constraint satisfaction problems: algorithms and applications, *European Journal of Operational Research*, 119, 557, 1999.

[5] Williams, H. and Wilson, J., Connections between integer linear programming and constraint logic programming — an overview and introduction to the cluster of articles, *INFORMS Journal on Computing*, 10, 261, 1998.

[6] Freuder, E.C. and Wallace, M., Constraint satisfaction, in: Glover, F. and Kochenberger, G.A. (eds.) *Handbook of Metaheuristics*, Kluwer Academic Publishers, Boston, MA, 2003, chap. 14.

[7] Nuijten, W. and Aarts, E., A computational study of constraint satisfaction for multiple capacitated job shop scheduling, *European Journal of Operational Research*, 90, 269, 1996.

[8] Lustig, I. and Puget, J., Program does not equal program: constraint programming and its relation to mathematical programming, *Interfaces*, 31, 29, 2001.

[9] Darby-Dowman, K., Little, J., Mitra, G., and Zaffalon, M., Constraint logic programming and integer programming approaches and their collaboration in solving an assignment scheduling problem, *Constraints*, 1, 245, 1997.

[10] Jordan, C. and Drexl, A., A comparison of constraint and mixed integer programming solvers for batch sequencing with sequence dependent setups, *ORSA Journal on Computing*, 7, 160, 1995.

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

[12] Henz, M., Scheduling a major college basketball conference — revisited, *Operations Research*, 49, 163, 2001.

[13] Nemhauser, G. and Trick, M., Scheduling a major college basketball conference, *Operations Research*, 46, 1, 1998.

[14] Baker, K., Magazine, M., and Polak, G., Optimal block design models for course timetabling, *Operations Research Letters*, 30, 1, 2002.

[15] Valouxis, C. and Housos, E., Constraint programming approach for school timetabling, *Computers & Operations Research*, 30, 1555, 2003.

[16] Rousseau, L., Gendreau, M., and Pesant, G., The synchronized vehicle dispatching problem, *Working Paper*, Center for Transportation Research, University of Montreal, 2002.

[17] Hooker, J. N., Logic, optimization, and constraint programming, *INFORMS Journal on Computing*, 14, 295, 2002.

[18] Milano, M., Ottoson, G., Refalo, P., and Thorsteinsson, E., The role of integer programming techniques in constraint programming's global constraints, *INFORMS Journal on Computing*, 4, 387, 2002.

[19] Hillier, F.S. and Lieberman, G.J., *Introduction to Operations Research*, Holden-Day, Inc., San Francisco, 1980.

[20] Williams, H.P., *Model Building in Mathematical Programming*, Wiley, New York, 1999.

[21] Pinedo, M., *Scheduling: Theory Algorithms, and Systems*, Prentice Hall, Englewood Cliffs, NJ, 1995.

[22] Du, J. and Leung, J.Y.-T., Minimizing total tardiness on one machine is NP-hard, *Mathematics of Operations Research*, 15, 483, 1990.

[23] Potts, C.N. and Van Wassenhove, L.N., A branch and bound algorithm for the total weighted tardiness problem, *Operations Research*, 33, 363, 1985.

[24] Van Hentenryck, P., *The OPL Optimization Programming Language*, MIT Press, Cambridge, MA, 1999.

[25] Elmaghraby, S.E., The one machine sequencing problem with delay costs, *Journal of Industrial Engineering*, 19, 105, 1968.

[26] Kanet, J.J. and Li, X., A weighted modified due date rule for sequencing to minimize weighted tardiness, *Journal of Scheduling* (forthcoming).

[27] Baker, K. R., *Introduction to Sequencing and Scheduling*, Wiley, New York, 1974.

[28] Akturk, M. S. and Yildirim, M. B., A new dominance rule for the total weighted tardiness problem, *Production Planning & Control*, 10, 138, 1999.

[29] Rachamadugu, R.M.V., A note on the weighted tardiness problem, *Operations Research*, 35, 450, 1987.

[30] Rinnooy Kan, A.H.G., Lageweg, B.J., and Lenstra, J.K., Minimizing total costs in one-machine scheduling, *Operations Research*, 23, 908, 1975.

[31] Baker, K. R., *Elements of Sequencing and Scheduling*, K. R. Baker (ed.), Hanover, NH, 2000.