Constraint-based Scheduling[*]

Markus P.J. Fromherz
Xerox PARC, 3333 Coyote Hill Road, Palo Alto, CA 94304, USA
www.parc.com/fromherz

Abstract

Constraint-based scheduling has become the dominant form of modeling and solving scheduling problems. Recently, due to ever more powerful embedded processors, it has become possible to embed and run constraint-based schedulers on-line even for fast processes such as product assembly sequencing. This makes constraint-based scheduling interesting to the control community as a new tool for system control, distributed and reconfigurable control, and the integration of various planning, scheduling, and control tasks. This tutorial gives a brief introduction to constraint-based scheduling, generic constraint programming techniques for modeling and solving scheduling problems, and a concrete real-time application example.

1 Introduction

Scheduling is the process of allocating resources to activities over time [3]. In a typical scheduling problem, resources are scarce and constrained in various ways (e.g., in the capacity of resources and the order of activities), and one is looking for a schedule of the activities that both satisfies the constraints and is optimal according to some criterion (e.g., the length of the schedule).

Scheduling problems are ubiquitous and appear in many forms, from classic job-shop scheduling to manpower and service scheduling, from product assembly sequencing to logistics resource allocation and scheduling. Scheduling has become an important component of business processes such as supply chain management, and it has its own range of successful software tool vendors.

Over the last decade, constraint-based scheduling (CBS) has become the dominant form of modeling and solving scheduling problems [35,39,4,23]. CBS separates the model – the description of activities, resources, constraints, and objectives – from the algorithms that solve the problem. This allows one to deal with a wider variety of constraints, facilitates the changing of the model, even dynamically, without changing the algorithms, and enables the re-use of the model for other tasks, such as simulation, planning, and diagnosis. Constraint solving methods such as domain reduction, constraint propagation, and backtracking search have proved to be well suited for many industrial applications [23,17]. Today, these methods are increasingly combined with classic solving techniques from Operations Research (OR), such as linear, integer, and mixed integer programming [5,37], to yield powerful tools for constraint-based scheduling.

Recently, due to continuing increases in computational power and available memory of embedded processors, it has become possible to embed and run constraint-based schedulers on-line and in real time even for fast processes that demand solving times on the order of tens or hundreds of milliseconds [13,17]. This development makes CBS interesting to the control community as a new tool for system control, distributed and reconfigurable control, and the integration of various planning, scheduling, and control tasks. For the scheduling community, it highlights a set of concerns that are much less prominent or non-existing in stand-alone, off-line scheduling applications [14].

In this tutorial, I will first introduce scheduling in more detail and then concentrate on the presentation of constraint programming as the foundation of CBS. I will then discuss various scheduling-specific extensions and scheduling variations, including concerns in embedded, real-time scheduling. As a concrete example, I will present the use of constraint-based scheduling in print-engine control, which has led to a set of technologies developed at the Xerox Palo Alto Research Center and deployed in Xerox products. The primary purpose of this tutorial is to give control researchers and practitioners an overview of the workings and capabilities of current CBS techniques.

[*] Invited tutorial paper at the American Control Conference (ACC'01), Arlington, VA, June 2001.

2 Scheduling

The traditional positioning of scheduling is between planning and execution: for a set of desired products, a *planner* determines the sequence of tasks (the plan) that will deliver the products, the *scheduler* decides on the specific resources, operations, and their timing to perform the tasks (the schedule), and an *executor* (or controller) then directs the system's resources to perform the operations at the specified times such that the products are produced (Fig. 1). (Occasionally in the literature, the term "planning" is meant to include scheduling. Here, planning generally means the decision *what* to execute, and scheduling means the decision *when* and *how* to execute it.)

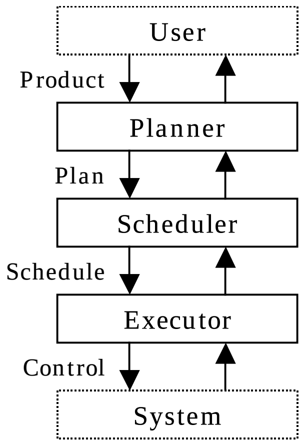


Fig. 1 Scheduling in context

The terms involved are meant to be generic. *Products* may be physical products, but also lectures in a university lecture timetabling problem, staff assignments in a staff scheduling problem, or object locations in a logistics problem. *Resources* may be machines or machine components, people, or any other object that is able to perform a useful operation. *Tasks* typically are individual steps available from unspecified resources in the system, such as moving or transforming a product. *Operations* refers to the specific capabilities of resources that perform particular tasks. The *system* is the collection of resources, from a single machine to a factory of machines to an organization of people, factories, vehicles, etc.

In the setup of Fig. 1, complexity increases from the bottom up, and consequently applications often are less automated (i.e., require more human intervention or at least preparation) at the higher levels. For example, in many applications (such as complex resource allocation or job-shop problems), even where scheduling is automated, planning is still being done by hand or relies on a library of plans prepared by human experts that allow a simple mapping from products to plans. Even for schedulers, readability and modifiability of schedules by humans is often an explicit requirement due to the limitations of existing technology.

In certain domains, however, both planning and scheduling can be completely automated and thus embedded in an integrated control system. One example is the planning and scheduling of paper transport and print operations on a modern, dynamically reconfigurable, multi-function print engine (copier, printer, fax machine, etc.). There, planning and scheduling are part of the machine's system control software, and scheduling may be thought of as just another level in the system's hierarchy of controllers. Scheduling tends to consider higher-level or aggregate operations and take a longer-term view, and thus scheduling problems are decidedly complex, combinatorial problems that are in general NP-hard [21,18]. At the same time, embedded schedulers are expected to have the usual properties of lower-level controllers, such as operating in parallel with the system's execution (on-line scheduling) and in a feedback loop (reactive scheduling), and providing results within hard real-time boundaries (real-time scheduling).

Classical scheduling problems are often categorized using the *machine shop* metaphor, such as flow shop, job shop, and open shop problems [7]. A machine shop consists of a set of machines (or resources), each of which can work on one task at a time. Jobs (e.g., the manufacturing of products) consist of tasks, each of which requires a resource during its execution. Particular machine shop problems differ in whether the order of tasks in a job is constrained (flow shop and job shop), and on whether the use of resources is fixed (flow shop). Deterministic algorithms are known only for problems with a few (2-3) machines and jobs.

Although the metaphor of resources, jobs and tasks can be generally applied to most scheduling problems, real-life problems typically have a much wider variety of constraints than just precedence and resource constraints [39]. Constraint-based methods have proved successful when problems are hard (solutions not obvious, many hard constraints, strong constraint interactions), when domain-specific and redundant constraints are available, and when problems change often [33].

3 Constraint Programming

At its heart, scheduling is a *constraint satisfaction and optimization problem*. Thus, in constraint-based scheduling, tasks are represented by resource selection and timing *variables*, possibly *constrained* by precedence constraints, resource constraints, routing conditions, and other restrictions. The search space is the space of possible assignments of tasks to resources and the concrete timings of tasks. As tasks are assigned to resources, their timing variables are further constrained by *resource constraints*, such as limited capacities and setup delays. A solution – a schedule – is an assignment to the resource and timing variables that specifies when the tasks have to be executed on what resources. Sometimes, a constraint-satisfying solution is sufficient, but often we demand an optimal schedule. In that case, an *objective function* defines the optimality criterion, such as schedule length, resource usage, or average or maximum lateness with respect to due dates.

Modern constraint solvers have their roots in at least three research communities [5,29]: Artificial Intelligence (AI), which investigated constraint satisfaction problems (CSPs) with a focus on general search procedures; Logic Programming, which led to constraint (logic) programming (CP/CLP) with a focus on modeling and programmability; and Operations Research (OR) with a focus on efficient algorithms for restricted problem formulations. Thus, CSP research has focused on the basic means for representing and operating on constraints, in particular various forms of backtracking search algorithms [30]. CP added an explicit programming component that allows programmers to more easily and flexibly define how to combine the basic constraint operations [35]. Today, solver algorithms are increasingly augmented with techniques from OR for specific constraint systems, such as linear constraints or constraints over reals [37]. In the following, I will take primarily a constraint programming view and use "CP" to refer to the combination of techniques just mentioned.

In the following, I will first introduce generic definitions and techniques used for constraint satisfaction and optimization problems. Later, I will present some of the constraints and search algorithms that have been added to address the specific characteristics of scheduling problems.

3.1 Constraint problems

A *constraint satisfaction problem* (CSP) is defined by a set of n variables x_i ($i = 1,\ldots,n$), a corresponding set of domains D_i ($i = 1,\ldots,n$) that declare the allowable values for each variable x_i, and a set of m constraints $c_j(x_1,\ldots,x_n)$ ($j = 1,\ldots,m$) over the variables which restrict the allowable combinations of variable values. More formally, a *constraint* is a mathematical relation which defines a subset S of the domain space $D_1 \times \ldots \times D_n$ such that the constraint is said to be *satisfied* for a variable assignment $\langle x_1,\ldots,x_n\rangle = \langle v_1,\ldots,v_n\rangle$, $v_i \in D_i$ ($i = 1,\ldots,n$), if $\langle v_1,\ldots,v_n\rangle \in S$. A variable assignment is *inconsistent* if at least one constraint is not satisfied. The collection of constraints is often called the *constraint store*. A *solution* to the CSP is an assignment of domain values to the variables that satisfies the constraints. In short, a constraint satisfaction problem is defined as follows.

Given n variables x_i with domains D_i and m constraints c_j,

find a solution	$\langle x_1,\ldots,x_n\rangle = \langle v_1,\ldots,v_n\rangle$	
such that	$v_i \in D_i$	$i=1,\ldots,n$
	$c_j(v_1,\ldots,v_n)$	$j=1,\ldots,m$

Before we look at variables and constraints in more detail, we can extend this definition to a constrained optimization problem. A *constrained optimization problem* (COP) is defined as a CSP with an additional objective function $h(x_1,\ldots,x_n)$ that specifies a preference criterion between solutions. That is, if the goal is to minimize h, then solution $\langle v_1,\ldots,v_n\rangle$ is preferred to solution $\langle v_1',\ldots,v_n'\rangle$ if $h(v_1,\ldots,v_n) < \mathrm{h}(v_1',\ldots,v_n')$. Consequently, a solution $\langle v_1,\ldots,v_n\rangle$ is optimal if $h(v_1,\ldots,v_n) \leq \mathrm{h}(v_1',\ldots,v_n')$ for all other solutions $\langle v_1',\ldots,v_n'\rangle$. Without loss of generality, we assume that h is to be minimized. The constrained optimization problem is thus defined as follows.

Given n variables x_i with domains D_i, m constraints c_j, and objective function h,

find a solution	$\langle x_1,\ldots,x_n\rangle = \langle v_1,\ldots,v_n\rangle$	
with minimal	$h(v_1,\ldots,v_n)$	
subject to	$v_i \in D_i$	$i = 1,\ldots,n$
	$c_j(v_1,\ldots,v_n)$	$j = 1,\ldots,m$

3.2 Modeling constraint problems

Variables may be of various types, i.e., have different *types of domains*:

- integer domains [l,u] with a lower bound l and upper bound u, i.e., variables may take any integral value in that range;
- logical (or Boolean) domains [false,true], often represented by integer variables with domains [0,1];
- enumeration (or choice) domains representing a set of choices, often represented by integer variables with range [1,k], where k is the number of choices;
- real domains with a lower and upper bound which may take any real value in that range;
- set domains where each value is a set.

Correspondingly, each variable type has its own *types of constraints*, such as

- arithmetic constraints for integer and real variables (e.g., $3x+y \leq z$);
- Boolean relations for logical variables (e.g., $x \wedge (y \vee \neg z)$);
- set constraints (e.g., to enforce element and subset relations).

Constraints over integer and related variables are also called *finite-domain constraints*. They are often handled in similar ways and form the dominant part of many solvers from the AI and CP communities. Many scheduling problems can be represented as finite-domain COPs. Also, finite-domain techniques are less well known than continuous optimization techniques in the control community. I therefore focus on these in this paper.

A set of primitive constraints (e.g., equality and inequality relations over integer expressions) together with rules for how to combine and derive constraints is called a *constraint system*. Depending on the constraint system, constraints may be linear or nonlinear over discrete or continuous variables and include equality, inequality, disequality ($\neq$), or special-purpose relations.

More generically, however, variables, domains, and constraints may be of any arbitrary, application-specific type without necessarily stepping outside the CP framework. In fact, as the next section will demonstrate, many constraint programming tools are parametric with respect to their constraint systems and usually allow multiple constraint systems to be used in parallel. This generality and extensibility is part of what makes CP so attractive for scheduling applications with their diverse types of constraints.

An *objective function* may be as simple as a single variable (e.g., representing the last task in a schedule), or it may formalize more complex criteria, including combinations of objectives, such as a weighted sum of the individual objective functions.

3.3 Solving constraint problems

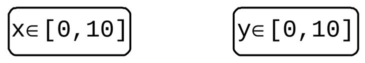
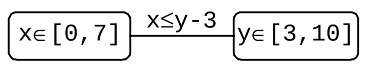
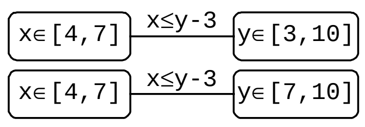
The usual way of solving finite-domain CSPs (and COPs) in CP is a combination of domain reduction, constraint propagation, and search. To understand these techniques and their various instances, it is best to think of a finite-domain CSP as a graph, where the nodes are variables (represented by their current domains) and the edges are constraints between the variables (cf. table below).

Domain reduction, constraint propagation

Domain reduction is the direct application of a unary constraint $c(x)$ to the variable x. For example, if x is an integer variable with current domain [0,10] and c is $x > 3$, then the domain of x becomes [4,10].

Constraint propagation is the propagation of changes in one variable's domain to the domains of other variables connected by constraints. For example, if x and y are integer variables with current domains [0,10], and a constraint $x \leq y–3$ is added, we can immediately propagate the lower bound of x to y, i.e., y's domain becomes [3,10], and we can propagate the upper bound of y to x, i.e., x's domain becomes [0,7]. Furthermore, if a constraint $x > 3$ is added next, reducing x's domain to [4,7], the new

Table 1 Propagation in a constraint graph

Operation	**Constraint graph**
Define x and y.	
Add $x \leq y–3$; propagate.	
Add $x > 3$; reduce x, propagate.	 x∈[4,7] x≤y-3 y∈[3,10] x∈[4,7] x≤y-3 y∈[7,10]

lower bound is propagated to y's domain, which becomes [7,10]. Table 1 gives a recap of this sequence of operations.

Notice that, with a constraint like $x \le y-3$, only the upper bound of x and the lower bound of y can be propagated. Propagation can often be attached to or "indexed" by components of a domain representation. This realization has led to the introduction of so-called *indexicals* (or projection constraints), efficient specialized edges in a constraint graph that replace the original constraints [10]. For example, the constraint $x \le y-3$ can be represented by the two indexicals $\mathrm{lb}(y) := \max(\mathrm{lb}(x)+3, \mathrm{lb}(y))$ and $\mathrm{ub}(x) := \min(\mathrm{ub}(y)-3, \mathrm{ub}(x))$, where lb and ub denote the lower and upper bounds of the domains, respectively. The first indexical, for instance, is "attached" to the lower bound of x's domain and triggers propagation whenever that value changes. More generically, a constraint $c(x_1,\ldots,x_n)$ is transformed to one or more indexicals "x_i in r_i" ($i = 1,\ldots,n$), where r_i specifies the feasible values for x_i in terms of the other variables in the constraint.

This form of constraint propagation leads to so-called *arc-consistency* [30], as it removes inconsistent values from the variables' domains along the edges (or arcs) of the graph. By itself, constraint propagation is an incomplete technique, since it does not remove all possible combinations of values that are inconsistent. For example, the solution $\langle x,y \rangle = \langle 7,7 \rangle$ would not be a consistent solution in the above scenario, even though 7 is still in both domains.

Because of this incompleteness, domain reduction and constraint propagation have to be complemented by search. Note that search is usually interleaved with constraint propagation: fixing a variable value during search may lead to further propagation from that variable. For example, given the constraint problem above, if the search procedure selects 6 as the value for x, the domain of y becomes [9,10] through propagation. From the search point of view, the goal of domain reduction and constraint propagation is to reduce the search space as much as possible before and during search.

Search

Search methods in constraint programming may be divided into refinement-based methods and repair-based methods [34].

Refinement-based methods are the most common form of search in constraint programming. Here, each of the variables is assigned a value incrementally until a complete solution is found or a constraint is violated ("labeling"). If a constraint is violated, the last assignment is undone and an alternative value is chosen ("enumeration"). If no value assignment is consistent, search backtracks to a previously assigned variable, and so on. The result is a depth-first tree search. The following is the corresponding pseudo-code.

```
set $X = \langle x_1,\ldots,x_n \rangle$;
while there are unassigned variables in $X$
  select an unassigned variable $x$ from $X$;
  select a value $v$ from the domain of $x$;
  assign $x = v$;
  backtrack if a constraint is violated;
end while
```

This algorithm contains primarily two choices, namely variable selection and value selection. The order in which variables and values are selected can have a significant impact on search efficiency, and various heuristics exist for these choices that try to minimize the need for backtracking. For example, heuristics that select variables with the smallest current domain and values that reduce other variables' domains the least through propagation are known to lead to good performance in many applications [2].

While such domain-independent heuristics can be effective, domain-specific heuristics are even better. This is of interest in the context of scheduling problems, where prior knowledge about the problem (e.g., the typical order of tasks) can be used as a heuristic (e.g., select variables and values in time order, constrain the maximum total schedule length) [19].

Another choice is in the backtracking procedure. Chronological backtracking – backtracking to and undoing previous variable and value selections in reverse order of assignment – is the easiest to implement, often quite effective, and thus the most common form used in constraint programming. However, various forms of "intelligent backtracking" have been explored in the CSP community. It has been shown, for example, that "easy" problems (with many solutions) sometimes lead algorithms to make a wrong choice early on that is not detected until much later during search [22]. In that case, instead of spending time on fruitless search at the bottom of the search tree, a "backjumping" algorithm will try to identify the earliest inconsistent assignment and directly backtrack to that variable.

Different degrees of arc-consistency embedded in search lead to different degrees of lookahead, from pure generate-and-test if no propagation is performed to full lookahead if complete network consistency is enforced [31,25]. Since full lookahead can be costly to compute and add more in overhead than it saves in backtracking, backtracking search with partial (local) arc-consistency has been shown to perform best for a variety of constraint problems.

If the constraint problem is an optimization problem, a refinement-based search can be augmented easily with a mechanism that adds a new constraint $h(x_1,\ldots,x_n) < h(v_1,\ldots,v_n)$ every time a new solution $\langle v_1,\ldots,v_n\rangle$ is found. This forces subsequent solutions to have increasingly better objective values and can be very effective in removing parts of the search tree. At the end, the last-found solution is returned as the optimal solution. (This kind of optimizing search has been called *branch-and-bound search*, not to be confused with the branch-and-bound optimization technique used to solve integer optimization problems with continuous techniques.) A variation of this technique, which is effective in scheduling problems, is *binary search*, which keeps progressively narrower lower and upper bounds l and u on $h(x_1,\ldots,x_n)$. The algorithm first adds the constraint $h(x_1,\ldots,x_n) \leq (l+u)/2$. If a solution $\langle v_1,\ldots,v_n\rangle$ is found, u is set to $h(v_1,\ldots,v_n)$; otherwise, l is set to $(l+u)/2+1$. The new value for l or u is substituted in the objective bound $h(x_1,\ldots,x_n) \leq (l+u)/2$ and search is restarted. This is repeated until $l \geq u$, when the optimal solution is found. Yet another effective and popular variant is *iterative deepening*, where the limit l on h in the constraint $h(x_1,\ldots,x_n) \leq l$ starts from a lower bound and is increased until a solution is found. This is effective if the initial l is close to the optimum. A more recent technique that has proved to be very effective for large problems is so-called *limited discrepancy search*, which assumes that the chosen value heuristic makes few mistakes, and which therefore initially limits the number of allowed deviations from the heuristic [20,38].

Repair-based methods start with a complete assignment to the variables. If this assignment is inconsistent, i.e., at least one constraint is violated, the assignment is "repaired" iteratively by assigning different values to one or more of the variables until a solution is found. The following is the corresponding pseudo-code.

```
set $V = \langle v_1,\ldots,v_n\rangle$ as initial solution for $\langle x_1,\ldots,x_n\rangle$;
while $V$ is inconsistent
  select an inconsistent assignment $x = v$ from $V$;
  select a new value $v'$ for $x$;
  assign $x = v'$ in $V$;
end while
```

This algorithm again contains primarily two choices, namely variable selection and value selection. Well-known heuristics include selecting variables that violate the largest number of constraints with values that decrease the number of violations. In the case of an optimization problem, repair-based methods can be combined easily with well-known optimization techniques such as hill climbing or simulated annealing. In these cases, variable and value selection not only consider the reduction in constraint violation, but also the decrease in the objective function when selecting variables and values to repair.

In contrast to refinement-based methods, repair-based methods typically are not complete, i.e., they are not guaranteed to find the global optimum or even a variable assignment that satisfies all the constraints. Therefore, repair-based methods typically require additional termination criteria, such as an upper limit on the number of repairs.

Both classes of search algorithms may further be augmented with randomization techniques. As already noted, refinement-based search in particular can be inefficient for "easy" optimization problems, where many solutions exist and no strong upper bound on the objective function helps during propagation. Approximate techniques using random restarts and limits on backtracking have been shown to be effective for a range of scheduling problems [4].

4 Constraint-based Scheduling

4.1 Modeling scheduling problems

Building on the CP representations and techniques introduced above, various variable and constraint types have been developed specifically for scheduling problems. Variable domains include

- interval domains where each value is an interval (e.g., start and duration);
- resource variables for various classes of resources.

In scheduling applications, integer variables might be used to represent timings, interval variables to represent tasks, logical variables to represent mutual dependencies or exclusions, resource domains to denote classes of resources, etc. Higher-level domains may also be defined in terms of lower-level domains; for example, interval variables are often represented as tuples of integer variables that denote start and duration of the interval.

Scheduling-specific constraints include

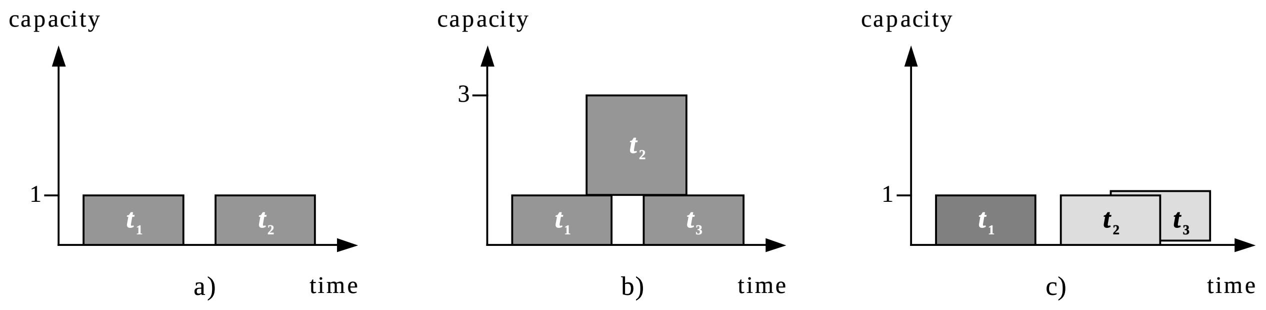
- interval constraints for interval variables (e.g., $t_1 \leq t_2$ to express that task 1 has to occur before task 2);
- resource constraints for timing (integer or interval) variables (e.g., allocate(r,t) for resource r and interval t to express that the task occupies resource r during interval t).

Again, higher-level constraints may be defined in terms of lower-level constraints. For example, for intervals t_1 and t_2 with start time variables t_i.s and duration variables t_i.d (i = 1,2; t_i.d > 0), the interval constraint $t_1 \leq t_2$ is equivalent to the integer constraint t_1.s+t_1.d $\leq t_2$.s.

Resource constraints define and constrain the available resources by restricting how multiple uses of a resource can be combined [1,5,27,17]. Resources are either renewable or consumable. Some *renewable resource* types are depicted in Fig. 2. Unary resources r can handle only one task at a time. Volumetric resources (also called multi-unit or n-ary resources) allow tasks to overlap so long as the total amount of resource use at any time does not exceed a given capacity limit. State resources allow tasks to overlap if they require the same state (e.g., the state of a switch). In contrast to renewable resources, *consumable resources* get depleted with each use and have to be renewed explicitly. Finally, tasks may or may not be interruptible, leading to the distinction between preemptive and non-preemptive scheduling [28] (not further discussed here).

As an example, we can assign a unary resource r to two tasks t_1 and t_2 by posting the constraints allocate(r,t_1) and allocate(r,t_2). These constraints are equivalent to the disjunctive constraint $t_1 \leq t_2 \vee t_2 \leq t_1$. Explicit use of resource constraints allows the solver to use and reason about more efficient representations than with such disjunctive constraints.

Instead of resource constraints with incremental allocations, some constraint programming systems provide *global constraints* [1,36]. A sample global constraint useful for scheduling is the cumulative constraint: given intervals t_i, resource uses u_i during t_i, and resource capacity k, cumulative([t_i,…], [u_i,…], k) is satisfied if, for every time point with overlapping tasks and corresponding resource uses U at this time point, $\Sigma_{u \in U}\, u \leq k$. In systems without interval constraints, this constraint would be defined as cumulative([s_i,…], [d_i,…], [u_i,…], k), where s_i and d_i correspond to the start and duration times



```
unary_resource(r),
allocate(r, t1),
allocate(r, t2)
⇔
cumulative([t1,t2],[1,1],1)
```

```
volumetric_resource(r, 3),
allocate(r, t1, 1),  % use=1
allocate(r, t2, 2),  % use=2
allocate(r, t3, 1)   % use=1
⇔
cumulative([t1,t2,t3],[1,2,1],3)
```

```
state_resource(r, [1,2]),
allocate(r, t1, 1), % state=1
allocate(r, t2, 2), % state=2
allocate(r, t3, 2)  % state=2
⇔
cumulative([t1,t2],[1,1],1),
cumulative([t1,t3],[1,1],1)
```

Fig. 2 Some types of resources: a) unary resource (tasks cannot overlap); b) volumetric resource (sum of task uses cannot exceed capacity (here 3) at any time; height indicates task use); c) state resource (tasks can only overlap if they require the same state; color indicates state). In all cases, length indicates task duration. The examples are shown with their definitions as resource and global constraints.

of the intervals, respectively. Other examples for global constraint are the all-different constraint (forcing all variables in a given set to have different values) and the cardinality constraint (limiting the number of variables that a given value can be assigned to) [36].

One difference between using allocation constraints and global constraints is that global constraints cannot be defined incrementally, i.e., all tasks have to be known when the constraint is posted, which makes them less useful for on-line scheduling applications.

The constraints presented so far are *hard* constraints, i.e., they represent restrictions that must be satisfied. Scheduling applications have led to increased interest in *soft* constraints, i.e., restrictions that can be relaxed if no feasible schedule can be found (in the allotted time). Examples are job due dates or resource use preferences. Some tools provide specific soft constraint systems [12,6]. Alternatively, soft constraints can often be added in appropriate form to the objective function (e.g., “minimize the difference between desired and actual due dates”).

4.2 Scheduling-specific propagation techniques

Several scheduling-specific propagation techniques have been developed, mostly concerning the reasoning about resources and intervals. In one technique, *resource timetables* [26], a timetable with required and available capacity at any time is maintained for each resource. Propagation between resources and tasks works both ways: as a task time becomes fixed, the task resource usage is entered in the timetable; conversely, as available capacity in the timetable is reduced, the interval domains of associated tasks are updated.

Another technique, *edge finding* [9,32], reasons about the order in which tasks can execute on a given resource. Each task is evaluated with respect to a set of other tasks. If it is determined that the task must or cannot execute before (or after) these tasks, it may be possible to infer new precedence constraints and new bounds on the task’s interval domain.

As mentioned before, depending on prior knowledge about the scheduling problem, there may also be domain-specific redundant constraints and search heuristics that lead to increased propagation and smaller search trees.

4.3 Scheduling

As an example, a simple but complete job-shop scheduling problem can be defined as a COP as follows. Given are a set of jobs and a set of machines. Each job consists of a set of tasks with given durations to be processed in a given order on assigned machines. Each machine can process only one task at a time. The problem is to find a schedule, i.e., a start time for each task, that minimizes the makespan (the total length of the schedule, i.e., the time at which all jobs are finished).

We can represent a task by an interval variable t, where t.s and t.d represent the start and duration times of the task, respectively, and $t.\mathrm{e} = t.\mathrm{s}+t.\mathrm{d}$ is its end time. We can further represent a machine by a unary resource variable r. Assume that all times are integers. Finally, the makespan is represented by variable l. Then the following constraints apply.

For every task t: $0 \le t \le l$.

For every task t, given duration d: $t.\mathrm{d} = d$.

For every job with tasks $\langle t_1,\ldots,t_n\rangle$:
$t_i \le t_{i+1}$ $(i = 1,\ldots,n-1)$.

For every task t, given assigned machine r:
allocate(r, t).

The objective function is defined as $h = l$, i.e., the goal is to minimize l. Remember that, given the appropriate resource constraint system, a unary resource r with allocated tasks T_r automatically enforces the constraints $t_i \le t_j \vee t_j \le t_i$ for all t_i, t_j in T_r. Alternatively, using global constraints, the resource constraints could be defined as cumulative(T_r, [1,1,…], 1) for each resource r with allocated task list T_r.

In traditional, *off-line / predictive scheduling*, this problem is solved by posting the constraints to the constraint solver and then calling the search procedure, e.g., minimize(X, l), where X is the list of all task intervals t and the variable l. This will instantiate the variables in X, using the propagation and search techniques presented above.

In *on-line / reactive scheduling*, the tasks and their constraints may become known only incrementally, and the scheduler may run concurrently to the execution of a previous (partial) schedule. Furthermore, the constraints (e.g., the availability of resources) may change during scheduling and/or execution, in which case part or all of the problem has to be rescheduled. The simplest approach to on-line scheduling is to treat the scheduling problem as a sequence of constraint problems, and to transfer commitments (variable assignments being "executed") from one problem to the next. However, optimizations to this approach are possible and necessary for real-time applications [14,24,11].

More generally, when embedding scheduling into a control system, a set of unique requirements come into play that require to adapt the constraint solver to this environment. These issues include memory management, real-time issues in updating constraints and assigning variables, and the interface to the control application [14].

There are many published examples for industrial constraint-based scheduling applications. (Description can for example be found on the Web sites of major tool vendors such as Ilog and Cosytec.) Examples for deployed real-time schedulers embedded in an on-board control system include Xerox's print-engine paper-path scheduler [17], NASA's Remote Agent Planner and Scheduler (RAX-PS) [24], and JPL's Aspen planner and scheduler [11]. The Xerox scheduler is discussed in the next section. The RAX-PS system for NASA Ames' Remote Agent, part of the experimental spacecraft Deep Space One, was supported by a constraint solver with customized (depth-first) search strategies and fast operations for on-line constraint management. JPL's Aspen planner and scheduler for NASA spacecraft contains an on-line constraint management system and provides real-time response by using iterative repair techniques.

Complexity

Unfortunately, most real scheduling problems are NP-hard [21,18]. Scheduling problems can sometimes be made easier by relaxing certain assumptions, e.g., by allowing to preempt tasks. However, most research has emphasized finding generic heuristics and enabling programmers to guide algorithms with domain-specific knowledge. Hence the move to "glass-box" constraint solvers, which are parametric not only with respect to constraint systems but even with respect to propagation rules [10].

For specific real-time applications, it is useful to perform a dedicated complexity analysis. As an example, we analyzed the complexity of the paper-path scheduler presented below by considering both typical and worst-case system configurations. While the scheduling problem is still exponential in general, we found that, for a set of typical machines, we can always find a solution in polynomial time, and for a set of typical jobs, we can also find the best solution in polynomial time [17]. Also, mitigating strategies may be available for atypical cases, such as slowing down the machine. Thus, separating typical from average and worst-case scenarios can be fruitful for real applications.

4.4 On-line scheduling and model-predictive control

Model-predictive control (MPC) [8] has become a popular control approach that has a number of interesting similarities to CBS. In particular, MPC also takes a model-based approach in which the model, objectives, and constraints are stated explicitly as a COP, independently of the control policy. MPC also shares with on-line scheduling the incremental nature of processing incoming requests and the optimization of decisions with respect to a horizon of known or predicted future events.

While it is impossible to do justice to the various forms of MPC, it is instructive to highlight some of the differences and similarities between CBS and MPC. Recall that in scheduling we are given n tasks with times t_i, m constraints c_j, and objective function h, resulting in the COP

$$\begin{array}{lll} \text{find a solution} & \langle t_1,\ldots,t_n\rangle = \langle v_1,\ldots,v_n\rangle & \\ \text{with minimal} & h(v_1,\ldots,v_n) & \\ \text{subject to} & c_j(v_1,\ldots,v_n) & j = 1,\ldots,m \end{array}$$

In on-line scheduling, this is the COP at a particular time step k, and the tasks $\langle t_1,\ldots,t_n\rangle$ represent the horizon at that time (with $t_i > k$, $i = 1,\ldots,n$). More tasks and constraints may be added in subsequent time steps as they become known. At each time step k, the goal, in its simplest form, is to identify which task(s) to start next. Depending on the solution times $\langle v_1,\ldots,v_n\rangle$, a single, multiple, or no tasks t_i may be scheduled to execute next (concretely all those with times $v_i = k+1$ in this example). The variables of these tasks thus become committed and no longer appear as variables at subsequent time steps.

In a discrete state-space formulation of MPC, we start from a process model

$$\begin{aligned} x_{k+1} &= Ax_k + Bu_k \\ y_{k+1} &= Cx_k \end{aligned}$$

that predicts the evolution of system state x and system output y over time, given control input u

and system, input, and output matrices A, B, and C, respectively. (x, y, and u are in general vectors.) In a simple control example, we are given a reference trajectory r_i, $i = k+1,\dots,k+n$, for a horizon of n steps at time step k, together with m constraints c_j, resulting in the COP

find a solution	$\langle u_{k+1},\dots,u_{k+n}\rangle = \langle v_{k+1},\dots,v_{k+n}\rangle$
with minimal	$h(v_{k+1},\dots,v_{k+n})$
subject to	$c_j(v_{k+1},\dots,v_{k+n}) \quad j = 1,\dots,m$

where h requires the system output to match the reference trajectory, e.g.,

$$h = \sum\nolimits_{i=k+1}^{k+n} (r_i - y_i)^2 \ ,$$

with y_i defined by the process model as a function of the current system state x_k and the control inputs $u_i = v_i$. The constraints c_j typically encode limits on control and output values.

In contrast to tasks t_i in scheduling, the order of reference points r_i and control inputs u_i is fixed and given. Also, the solution values v_i in MPC do not represent different tasks, but the values of the same control "task" at different times. Thus, at each time step k, the goal is to determine at what value to set the next control input, u_{k+1}. The resulting state x_{k+1} becomes the new "start" state for the next time step, and the horizon is extended accordingly.

From a constraint programming perspective, online scheduling and MPC have several common requirements. Both work incrementally on a stream of requests (tasks, reference points), and have to be reactive in the sense that tasks, states, and reference trajectories may change over time from their original or predicted values. Furthermore, both generally use a *full optimization with minimal commitment* approach [13]: while the variable assignment for the next step to be executed is part of an optimal solution for all variables, the other variables are to be kept open in case more information becomes available (in the form of new or changed tasks and reference points, respectively). In other words, the optimizer is to return a solution for the next step only, but guarantee that it is part of an optimal solution for all known future steps. Because of these similarities, it is to be expected that constraint-based scheduling and model-predictive control will benefit from each other's techniques as common needs and generic solutions are better understood.

5 Application Example: Constraint-based Scheduling for Paper-path Control

Modern digital reprographic systems come in many forms, from low-end printers to high-end multifunction devices. They typically consist of a source of paper and images, a paper path that brings these together at the right time, place and orientation, and finishing components that collate, sort, staple and bind the resulting, marked sheets. As a concrete example for how CBS can be used in a real-time control environment, this section presents its use in paper-path control at Xerox [17,15,16,14].

5.1 Architecture and control process

Large machines are typically split into *modules* such as feeder, mark engine, and finisher, which in turn consist of components such as feed trays, transport belts, sheet inverters, etc.[1] As a concrete example, consider a sheet inverter with its two modes of operation (Fig. 3): either the sheet is guided by the inversion gate from the input rollers down into the inversion rollers, where it is stopped and then moved in reverse direction up and through the output rollers (inverting it from face-up to face-down orientation or vice versa), or it is guided by the inversion gate directly to the output rollers (leaving its orientation unchanged).

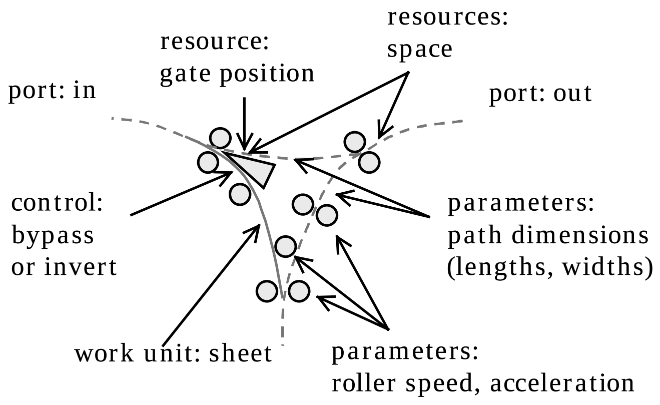


Fig. 3 Schematic view of a machine component (inverter) and its model elements

The hierarchical control software architecture mirrors the architecture of the machine, with low-level

[1] All examples of machine configurations and use scenarios are realistic but simplified, and none should be taken as describing an existing or future Xerox product.

controllers for modules and components and a system-level controller that plans and schedules module operations for the entire machine. For example, in order to deliver a desired output document (“five collated, stapled, double-sided copies of a given high-light color, 10-page document”), the system controller instructs the feeder, mark engine, and finisher modules to feed, print, and finish the sheets at certain times such that together a complete document is produced. The system controller works under tight real-time constraints, as machines may produce prints at the rate of 60 to 180 pages per minute. The controller typically gets a few tenths of a second of real time to plan and schedule each sheet’s operations.

5.2 Modeling component capabilities

In the past, paper-path schedulers were developed “by hand” for specific configurations based on an analysis of component interactions. A primary goal of our approach was to replace this heuristic approach by a constraint-based scheduler and model the capabilities of machine components separately, thus providing a solution that could be adapted automatically to new configurations by simply replacing the models.

Conceptually, reprographic machines may be thought of as *multi-pass assembly line machines*, where parts (e.g., sheets and images) are moved along the assembly line (e.g., paper path, photoreceptor belt), manipulated, and put together, until a desired output is produced. Consequently, we model a reprographic machine as a set of connected components with capabilities. A capability is a distinct component operation and defined by the transformation it performs, constraints on the features of sheets and images, its timing behavior, and any requirements on internal resources. A detailed description of the domain theory and our modeling language, CDL, is beyond the scope of this paper (cf. [17]). Instead, consider the inverter model in Fig. 4 as a concrete example.

Structurally, the inverter has two ports, `in` and `out`, through which sheets enter and leave, with rollers modeled as resources `r_in` and `r_out`. The inverter gate is a state resource `r_inv` that has to be in either the `"bypassing"` or the `"inverting"` position while the sheet is moving through. Finally, the model is parameterized by the length and speed of the transport. The complete CDL model of the inverter is defined in Fig. 4.

```
Component Inverter(int length, int speed) {
   EntryPort in;                // ports
   ExitPort out;
   UnaryResource r_in, r_out;   // decl.
   StateResource r_inv;
   IntVariable t_out, d, d_byp, d_inv;
   FeatureVariable s, s_in, s_out;

   Capability Bypass(IntVariable t_in) {
      in.Input(s, t_in);        // events
      out.Output(s, t_out);
      s.width <= 285;           // feature c.
      t_in + d_byp == t_out;    // time c.
      d_byp == length/speed;
      d == s.length/speed;
      r_in.Allocate(t_in, d);   // res. c.
      r_out.Allocate(t_out, d);
      r_inv.Allocate(t_in, d_byp,
                     "bypassing");
   } // Capability Bypass

   Capability Invert(IntVariable t_in) {
      in.Input(s_in, t_in);     // events
      out.Output(s_out, t_out);
      s_in.width <= 285;        // feature c.
      s_in.length <= 436;
      s_out == s_in             // trafo.
            except {orientation};
      s_in.orientation==1-s_out.orientation;
      t_in + d_inv == t_out;    // time c.
      d_inv == (length+s_in.length)/speed;
      d == s_in.length/speed;
      r_in.Allocate(t_in, d);   // res. c.
      r_out.Allocate(t_out, d);
      r_inv.Allocate(t_in, d_inv,
                     "inverting");
   } // Capability Invert
} // Component Inverter
```

Fig. 4 A model of the sheet inverter in the modeling language CDL (“decl.” = “declaration”, “c.” = “constraint”, “res.” = “resource”, “trafo.” = “transformation”)

In the inverter’s bypassing capability, a sheet `s` enters the component at time `t_in` and exits at time `t_out`, while, in the inverting capability, a sheet `s_in` is transformed to a sheet `s_out` that is identical to the input sheet except for its orientation. (Sheets and images are represented through their features, e.g., length, width, color, orientation, and images.) Notice how the sheet width, transport durations (`d` and `d_byp`/`d_inv`), and resources are constrained by feature, timing, and allocation constraints. The inverter’s controller will be instructed to perform these capabilities with the commands `Bypass(t_in)` and `Invert(t_in)`, respectively (name and reference time).

As an example, Fig. 5 shows the allocations for a document that requires two bypass operations (allocations 1 and 2), two inversions (3 and 4), and two more bypass operations (5 and 6). (Observe the effect of the inversion constraint on the input gap between sheets 4 and 5.)

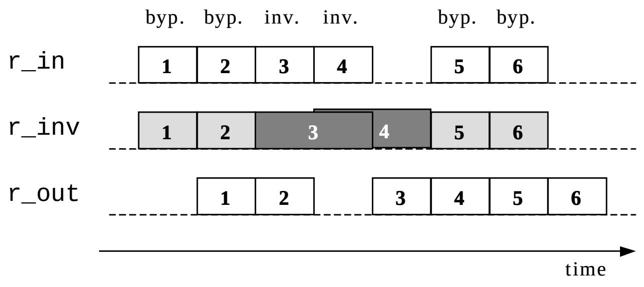


Fig. 5 Inverter resources with allocations for multiple selected capabilities from the inverter in Fig. 4 (two bypasses, two inversions, and two bypasses)

CDL also allows one to model the composition of components. When capabilities of two connected components are selected and composed for a sheet plan, the output event of the first component's selected capability becomes the input event of the second component's selected capability. In particular, feature constraints are propagated within a sequence of selected capabilities, providing a complete specification of the output sheet produced by the capabilities.

Fig. 6 shows representative resources and allocations for two typical machine capabilities, namely the printing of a simplex and a duplex sheet, put together from the resources and allocations of the machine's components. A simplex sheet is fed into the paper path, moved to the print station, and then moved to the stacker. A duplex sheet is also fed and moved to the print station, but then inverted and moved around the duplex loop for printing on the other side, before it is inverted again and move to the stacker.

We provide three constraint systems in CDL: an *integer constraint system* to represent times, lengths, etc., a *feature constraint system* to describe and reason about the attributes of sheets and images in a hierarchical, extensible fashion, and a *resource constraint system* with essentially the types of resources introduced above. CDL is parametric with respect to constraint systems, i.e., new constraint systems can be integrated seamlessly into the language as needed.

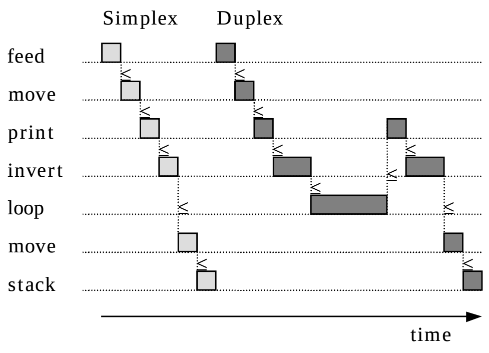


Fig. 6 Machine resources with allocations for simplex and duplex capabilities

5.3 Control

When composing machine modules to a complete machine, the modules pass up their module models to the system controller, where the models are composed automatically to a machine model. At run time, given this machine model and a stream of specifications for desired output sheets, the system controller's planner identifies the capabilities that need to be executed for each sheet and adds further constraints, such as precedence constraints between output times (to enforce the correct sheet output order). The scheduler receives this stream of sheet plans and incrementally and optimally solves the timing constraints in order to find feasible schedules, typically minimizing the output time of the last known sheet. In particular, a schedule instantiates the reference time variables of the control commands, which are then sent to the module controllers in order to execute the schedule. In the latest generation deployed in a product under development, CDL is translated directly into C++ data structures, and the scheduler is based on a generic finite-domain constraint solver using the techniques presented in Sections 3 and 4.

Fig. 7 shows an optimal schedule for printing ten sheets, namely six simplex sheets followed by two duplex sheets followed by two simplex sheets, given the individual capabilities presented in Fig. 6. Notice that the output order is correct even though the duplex sheets are fed out of order, since the interleaving simplex sheets pass the duplex sheets while those are in the duplex loop. Notice also that this interleaving leads to a shorter schedule than feeding sheets in output order (i.e., here we get the printing of four sheets “for free”).

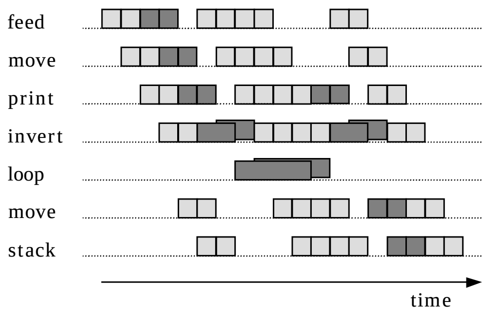


Fig. 7 Schedule for multiple capabilities (six simplex, two duplex, two simplex)

6 Conclusion

As the demand for providing more functionality for less cost increases, developing correct and effective control software for complex dynamic systems becomes both more important and more challenging. Constraint-based scheduling provides a set of technologies for constructing high-level control software that separates scheduling algorithms from system-specific information. As indicated in this tutorial, generic constraint solvers constitute powerful tools that facilitate control software development and enable optimal control, while still integrating well with a traditional procedural implementation environment.

The trend of more integration of CLP, AI, and OR techniques will continue to provide more versatile solvers. However, while generic constraint-based tools for traditional scheduling applications are widely available, the use of constraint-based techniques in a real-time application still requires a significant amount of adaptation of existing techniques to the particular environment. More work is also required in making constraint-based scheduling accessible to software engineers by developing usable modeling languages and associated analysis and transformation tools. As these libraries and tools become available, constraint-based scheduling is likely to become an important part of advanced control software.

References

[1] A. Aggoun & N. Beldiceanu, “Extending CHIP in order to Solve Complex Scheduling and Placement Problems.” In: *Journal of Mathematical and Computer Modelling*, vol. 17, no. 7, 1993, pp. 57-73.

[2] F. Bacchus and P. van Run, “Dynamic Variable Ordering in CSPs.” In: *Principles and Practice of Constraint Programming (CP’95)*, Springer-Verlag, LNCS 976, Sept. 1995, pp. 258-275.

[3] K. R. Baker, *Introduction to Sequencing and Scheduling*. Wiley & Sons, 1974.

[4] P. Baptiste, C. Le Pape, and W. Nuijten, “Constraint-based Optimization and Approximation for Job-shop Scheduling.” In: *AAAI-SIGMAN Workshop on Intelligent Manufacturing Systems, IJCAI’95*, Montreal, Canada, 1995.

[5] P. Baptiste, C. Le Pape, and W. Nuijten, “Incorporating Efficient Operations Research Algorithms in Constraint-based Scheduling.” In: *First Int. Joint Workshop on Artificial Intelligence and Operations Research*, Timberline Lodge, Oregon, 1995.

[6] S. Bistarelli, R. Gennari, and F. Rossi, “Constraint Propagation for Soft Constraint Satisfaction Problems: Generalization and Termination Conditions.” In: *Int. Conf. on Principles and Practice of Constraint Programming (CP’2000)*, Springer-Verlag, LNCS, 2000.

[7] J. Błazewicz, K. Ecker, G. Schmidt, and J. Weglarz, *Scheduling in Computer and Manufacturing Systems*. Springer-Verlag, 1993.

[8] E. F. Camacho and C. Bordons, *Model Predictive Control*. Springer-Verlag, 1999.

[9] J. Carlier and E. Pinson, “A Practical use of Jackson’s Pre-emptive Schedule for Solving the Job-shop Problem.” In: *Annals of Operations Research*, vol. 26, 1990, pp. 269-287.

[10] M. Carlsson, G. Ottosson, and B. Carlson, “An Open-Ended Finite Domain Constraint Solver.” In: *Int. Symp. on Programming Languages: Implementations, Logics, and Programming (PLILP’97)*, Springer-Verlag, LNCS 1292, Sept. 1997.

[11] S. Chien, G. Rabideau, R. Knight, R. Sherwood, B. Engelhardt, D. Mutz, T. Estlin, B. Smith, F. Fisher, T. Barrett, G. Stebbins, and D. Tran, “ASPEN – Automated Planning and Scheduling for Space Mission Operations.” In: *SpaceOps 2000*, Toulouse, France, June 2000.

[12] P. Codognet and F. Rossi, “Solving and Programming with Soft Constraints: Theory and Practice.” In: *ECAI’00, Tutorial on Soft Constraints*, May 2000. Also in: *AAAI’00, Tutorial on Soft Constraints*, July 2000.

[13] M. P. J. Fromherz and B. Carlson, “Optimal Incremental and Anytime Scheduling.” In: *Workshop on Constraint Languages/Systems and their Use in Problem Modeling, ILPS’94*, Nov. 1994, Syracuse, N.Y., USA; European Computer-Industry Research Centre, TR ECRC-94-38, pp. 45-59.

[14] M. P. J. Fromherz and J. H. Conley, “Issues in Reactive Constraint Solving.” In: *Workshop on Concurrent Constraint Programming for Time Critical Applications — COTIC’97, CP’97*, Linz, Austria, Oct. 1997.

[15] M. P. J. Fromherz and V. A. Saraswat, “Model-Based Computing: Using Concurrent Constraint Programming for Modeling and Model Compilation.” In: *Principles and Practice of Constraint Programming (CP’95)*, Springer-Verlag, LNCS 976, Sept. 1995, pp. 629-635.

[16] M. P. J. Fromherz and V. A. Saraswat, “Model-Based Computing: Constructing Constraint-Based Software for Electro-Mechanical Systems.” In: *Practical Applications of Constraint Technology*, Paris, France, April 1995, pp. 63-66.

[17] M. P. J. Fromherz, V. A. Saraswat, and D. G. Bobrow, “Model-based Computing: Developing Flexible Machine Control Software.” In: *AI Journal*, vol. 114, no. 1-2, Oct. 1999, pp. 157-202.

[18] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Co., New York, 1979.

[19] L. Getoor, G. Ottosson, M. P. J. Fromherz, and B. Carlson, “Effective Redundant Constraints for Online Scheduling.” In: *AAAI’97*, Providence, RI, July 1997, pp. 302-307.

[20] W. D. Harvey and M. L. Ginsberg, “Limited Discrepancy Search.” In: *IJCAI-95*, Montreal, Quebec, 1995, pp. 607-613.

[21] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnyoo Kan, “Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey.” In: *Discrete Optimization 1977*, Vancouver, BC, 1977.

[22] T. Hogg, B. Huberman, and C. Williams, “Phase Transition and the Search Problem.” In: *AI Journal*, vol. 81, 1996, pp. 1-15.

[23] *Optimization Technology White Paper – A Comparative Study of Optimization Techniques*. Ilog, 1997.

[24] A. Jónsson, P. Morris, N. Muscettola, K. Rajan, and B. Smith, “Planning in Interplanetary Space: Theory and Practice.” In: *Fifth Int. Conf. on Artificial Intelligence Planning Systems*, Breckenridge, CO, April 2000.

[25] V. Kumar, “Algorithms for Constraint Satisfaction Problems: A Survey.” In: *AI Magazine*, vol. 13, no. 1, 1992, pp. 32-44.

[26] C. Le Pape, “Implementation of Resource Constraints in ILOG SCHEDULE: A Library for the Development of Constraint-based Scheduling Systems.” In: *Intelligent Systems Engineering*, vol. 3, 1994, pp. 55-6.

[27] C. Le Pape and P. Baptiste, “A Constraint Programming Library for Preemptive and Non-Preemptive Scheduling.” In: *Third Int. Conf. and Exhibition on the Practical Application of Constraint Technology*, London, UK, 1997.

[28] C. Le Pape and P. Baptiste, “Resource Constraints for Preemptive Job-Shop Scheduling.” In: *Constraints*, vol. 3, 1998, pp. 263-287.

[29] I. J. Lustig and J.-F. Puget, *Program != Program: Constraint Programming and its Relationship to Mathematical Programming*. Ilog, 1999.

[30] A. K. Mackworth, “Consistency in Networks of Relations.” In: AI Journal, vol. 8, no. 1, 1977, pp. 99-118.

[31] B. Nadel, “Tree Search and Arc Consistency in Constraint Satisfaction.” In: *Search in Artificial Intelligence*, Springer-Verlag, 1988, pp. 287-342.

[32] W. Nuitjen and E. Aarts, “A Computational study of Constraint Satisfaction for Multiple Capacitated Job-shop Scheduling.” In: *European Journal of Operational Research*, vol. 90, 1996, pp. 269-284.

[33] H. Simonis, “Scheduling and Planning with Constraint Logic Programming.” In: *Practical Applications of Constraint Technology*, Tutorial, Paris, France, 1995.

[34] E. Tsang, *Foundations of Constraint Satisfaction*. Academic Press, San Diego, California, 1993.

[35] P. van Hentenryck, *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.

[36] P. van Hentenryck, “Scheduling and Packing in the Constraint Language cc(FD).” In: [39], pp. 137-167.

[37] P. van Hentenryck, *The OPL Optimization Programming Language*. MIT Press, 1999.

[38] T. Walsh, “The Constrainedness Knife-Edge.” In: *AAAI’98*, 1998, pp. 406-411.

[39] M. Zweben and M. Fox, *Intelligent Scheduling*. Morgan Kaufman, 1994.