J Intell Manuf (2010) 21:5–15
DOI 10.1007/s10845-008-0203-4

Constraint satisfaction techniques in planning and scheduling

Roman Barták · Miguel A. Salido · Francesca Rossi

Received: 5 August 2008 / Accepted: 30 October 2008 / Published online: 20 November 2008

Abstract Over the last few years constraint satisfaction, planning, and scheduling have received increased attention, and substantial effort has been invested in exploiting constraint satisfaction techniques when solving real life planning and scheduling problems. Constraint satisfaction is the process of finding a solution to a set of constraints. Planning is the process of finding a sequence of actions that transfer the world from some initial state to a desired state. Scheduling is the problem of assigning a set of tasks to a set of resources subject to a set of constraints. In this paper, we introduce the main definitions and techniques of constraint satisfaction, planning and scheduling from the Artificial Intelligence point of view.

Keywords Constraint satisfaction · Planning · Scheduling

Introduction

Planning and scheduling techniques have recently seen important advances thanks to the application of constraint satisfaction models and tools. Most real-world problems can be cast as highly coupled planning and scheduling problems, where resources must be allocated so as to optimize overall performance objectives. Therefore, solving these problems requires an adequate mixture of planning, scheduling and resource allocation to competing goal activities over time in the presence of complex state-dependent constraints. Solutions to these problems must integrate resource allocation and plan synthesis capabilities, which can be efficiently managed by using constraint satisfaction techniques. The aim of this paper is to give a general overview of three different but interrelated areas: constraint satisfaction, planning, and scheduling.

R. Barták
Charles University, Prague, Czech Republic
e-mail: bartak@ktiml.mff.cuni.cz

M. A. Salido (✉)
Universidad Politécnica de Valencia, Valencia, Spain
e-mail: msalido@dsic.upv.es

F. Rossi
University of Padova, Padua, Italy
e-mail: frossi@math.unipd.it

Constraint satisfaction problems

Constraint programming (CP) is a powerful paradigm for solving combinatorial problems. CP was born as a multidisciplinary research area that embeds techniques and notions coming from many other areas, among which artificial intelligence, computer science, databases, programming languages, and operations research play an important role. CP is currently applied with success to many domains such as scheduling, planning, vehicle routing, configuration, networks, and bioinformatics. More information about CP can be found in (Apt 2003; Dechter 2003; Tsang 1993; Rossi et al. 2006).

Definitions

The concept of a constraint satisfaction problem (CSP) is fundamental in CP.

Definition 1 (*CSP*). A *constraint satisfaction problem* consists of:

- a set of variables $X = \{x_1, x_2, \ldots, x_n\}$
- a set of domains $D = \{D_1, D_2, \ldots, D_n\}$ such that for each variable $x_i \in X$ there is a domain D_i;

– a set of constraints $C = \{c_1, c_2, \ldots, c_k\}$ such that the scope of each constraint is a subset of X.

Definition 2 (*Variable Domain*). The *domain* of a variable is a set of all considered values that can be assigned to the variable. Usually, we assume finite discrete domains.

Definition 3 (*Instantiation*). An *instantiation* is a set of pairs (x_i, a_i) such that x_i is a variable; a_i is a value from the domain of x_i; and each variable appears at most once in the instantiation.

We say that the instantiation is *complete* for a set of variables X if each variable from X appears in the instantiation and no other variable appears there. If the instantiation does not contain all variables from X, then the instantiation is incomplete (partial) for X. We say that instantiation I extends instantiation J if $I \supseteq J$.

Definition 4 (*Constraint*). A *constraint* is a pair (t, R), where t is a set of variables (called *scope*; the size of scope is called *arity*) and R is a set of complete instantiations for t (sometimes called a domain of the constraint). We can also see the constraint as a subset of the Cartesian product of domains of variables in t—the constraint restricts the values that the variables can simultaneously take. A CSP with constraints of arity one or two is called a *binary CSP*.

Instantiation I *satisfies constraint* (t, R) if there exists $J \in R$ such that I extends J. Constraint can be specified extensionally as a set of tuples (satisfying instantiations) or intentionally as a formula that defines the satisfying instantiations.

Definition 5 (*Solution*). A *solution* to a CSP is a complete instantiation of the variables in X satisfying all the constraints in C.

Definition 6 (*Consistency*). If a CSP has at least one solution, it is said that the CSP is *satisfiable* or *consistent*, otherwise we say that it is inconsistent.

Example: map/graph coloring problem

The map coloring problem is the problem to color the areas in a map using a predefined number of colors so that the neighboring areas have different colors. The map coloring problem can be represented as a graph coloring problem to color the vertices of a given graph using a predefined number of colors in such a way that connected vertices get different colors. It is very easy to model this problem as a CSP: there are as many variables as vertices, and the domain of each variable contains the colors to be used. If there is an edge between the vertices represented by variables x_i and x_j, then there is an inequality constraint referring to these two variables, namely: $x_i \neq x_j$.

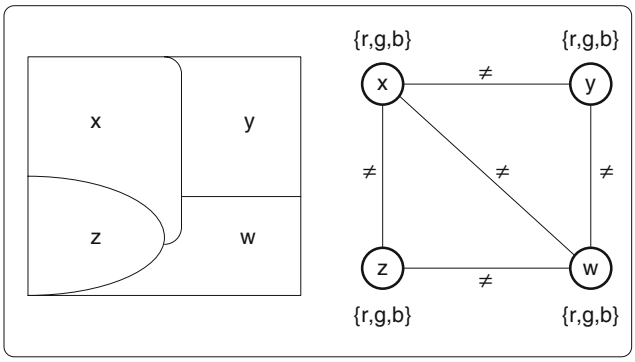

Fig. 1 Map/graph coloring problem

Graph coloring is known to be NP-complete, so one does not expect a polynomial-time algorithm for solving this problem. It is easy to generate a large number of test graphs with certain parameters, which are more or less difficult to be colored, so the family of graph coloring problems is appropriate to test algorithms thoroughly. Furthermore, many practical problems, like ones from the field of scheduling and planning, can be expressed as an appropriate graph coloring problem (Ruttkay 1998).

Figure 1 shows an example of a map coloring problem and its graph representation. The map is composed of four regions/variables $\{x, y, z, w\}$ to be colored. Each region can be colored in three different colors: red (r), green (g), or blue (b) (its domain). Each edge represents the binary constraint which states that two adjacent regions must be colored with different colors.

Search techniques

A CSP can be solved by systematically exploring the solution space via an uninformed search. Such search algorithms instantiate variables one after the other in such a way that the current partial instantiation is always consistent. If this is not possible, that is, all the possible values for the next variable are in conflict with some earlier assignment, then backtracking takes place.

Following, we present some well-known search techniques. More information can be found in (Barták 1998; Rossi et al. 2006).

Complete search algorithms

Most algorithms for solving CSPs search systematically the space of all possible assignments of values to variables. Such algorithms are guaranteed to find a solution, if one exists, or to prove that the problem is insoluble. The disadvantage of these algorithms is that they may take a very long time. The

actions of many search algorithms can be described by a search tree.

Backtracking search (BT) A simple algorithm for solving a CSP is backtracking search (Bitner and Reingold 1975). Backtracking works with an initially empty set of consistently instantiated variables and tries to extend the set to a new variable and a value for that variable. If successful, the process is repeated until all variables are included. If unsuccessful, another value for the most recently added variable is considered. Returning to an earlier variable in this way is called a backtrack. If that variable doesn't have any further values, then the variable is removed from the set, and the algorithm backtracks again. The simplest backtracking algorithm is called *chronological backtracking* because at a dead-end the algorithm returns to the previous variable in the ordering.

In the BT method, as soon as all the variables relevant to a constraint are instantiated, the validity of the constraint is checked. If a partial solution violates any of the constraints, backtracking is performed to the most recently instantiated variable that still has alternatives available. Clearly, whenever a partial instantiation violates a constraint, backtracking is able to eliminate a subspace from the Cartesian product of the variable domains.

Look-back algorithms Chronological backtracking can suffer from thrashing; the same dead-end can be encountered many times. If X_i is a dead-end, the algorithm will backtrack to X_{i-1}. Suppose a new value for X_{i-1} exists, but there is no constraint between X_i and X_{i-1}. The same dead-end will be reached at X_i again and again until all values of X_{i-1} have been exhausted.

Look-back algorithms try to exploit information from the problem to behave more efficiently in dead-end situations. Like BT, look-back algorithms perform consistency checks *backwards* (between the current variable and past variables).

Backjumping (BJ) (Gaschnig 1979) is an algorithm similar to BT except that it behaves in a more intelligent manner when a dead-end (X_i) is found. Instead of backtracking to the previous variable (X_{i-1}), BJ backjumps to the deepest past variable X_j, with $j < i$, that is in conflict with the current variable X_i. It is said that variable X_j is in conflict with the current variable X_i if the instantiation of X_j precludes one of the values in X_i. Changing the instantiation of X_j may make it possible to find a consistent instantiation of the current variable. Thus, BJ avoids the redundant work that BT does by trying to reassign variables between X_j and the current variable X_i.

Conflict-directed backjumping (Prosser 1993), backmarking (Gaschnig 1977), and learning (Frost and Dechter 1994) are other examples of look-back algorithms.

Look-forward algorithms As we have explained, look-back algorithms try to enhance the performance of BT by a more intelligent behavior when a dead-end is found. Nevertheless, they still perform only backward consistency checks and they ignore the future variables.

Look-forward algorithms make *forward* checks at each step of the search. Let us assume that, when searching for a solution, variable X_i is given a value which excludes all the possible values for a later variable X_j. When using uninformed search, this will only turn out when X_j will be considered for instantiation. Moreover, in case of BT, thrashing will occur: the search tree will be expanded again and again till X_j, as long as the level of backtracking does not reach X_i. Both anomalies could be avoided by recognizing that the chosen value for X_i cannot be part of a solution, as there is no value for X_j which is compatible with it. Lookahead algorithms do this, by accepting a value for the current variable only if after having looked ahead, it could not be seen that the instantiation would lead to a dead-end. When checking this, problem reduction can also take place, by removing the values from the domain of the future variables which are not compatible with the current instantiation. The algorithms differ in how far and thorough they look ahead and how much reduction they perform.

Forward-checking (FC) (Haralick and Elliot 1980) is one of the most common look-forward algorithms. It checks the satisfiability of the constraints, and removes the values of the future variables which are not compatible with the current variable's instantiation. At each step, FC checks the current assignment against all the values of future variables that are constrained with the current variable. All values of future variables that are not consistent with the current assignment are removed from their domains. If a domain of a future variable becomes empty, the assignment of the current variable is undone and a new value is assigned. If no value is consistent then backtracking is carried out. Thus, FC guarantees that at each step the current partial solution is consistent with each value in each future variable. Thus, FC can identify dead-ends and prune the search space sooner.

Incomplete search algorithms

Although complete search techniques, such as those described in the previous sections, always return a solution if there is one, or prove that there is no solution, we may sometimes want to use other techniques that don't possess this desirable property if they are more convenient from other points of view.

Incomplete search methods (Michalewicz and Fogel 2000) do not explore the whole search space. They search the space either non-systematically or in a systematic manner, but with a limit on some resource. These approaches do not ensure to collect all the solutions, nor to find a solution if there is at least one, nor to detect inconsistency, but their computational time can be much shorter compared to systematic search techniques.

Moreover, they may be sufficient when just some solution, or a good enough solution, is needed. These methods, known as metaheuristics, covers a very large class of resolution paradigms, from evolutionary algorithms to local search techniques. The main approaches for incomplete search are based on constructive methods or on iterative repair methods. The first ones gradually extend a partial solution to a complete one, while the second ones start with an initial solution and incrementally modify the values to get a better one.

For instance, local search does not instantiate one variable at a time, but (in its simpler version) start with a complete assignment to all the variables, and then modifies it slightly to pass to a new complete assignment which is closer to be a solution, or closer to optimality.

Consistency techniques

Consistency techniques were introduced to simplify CSPs and to improve the efficiency of systematic search techniques. The number of possible solutions can be huge, while only very few may be consistent. By eliminating redundant values from the problem definition, the size of the solution space decreases. Reduction of the problem can be done once, as a pre-processing step, or it can be interleaved with the exploration of the solution space by a search algorithm.

Local inconsistencies are single values or combination of values for variables that cannot participate in any solution because they do not satisfy some local consistency property. For instance, if a value a of variable x is not compatible with all the values in a variable y that is constrained with x, then a is inconsistent and this value can be removed from the domain of the variable x.

In the following paragraphs we introduce the most well-known and widely used algorithms for binary CSPs.

- A CSP is *node-consistent* if all the unary constraints are satisfied by all the elements of the variable domains. The straightforward node-consistency algorithm (NC), which removes the redundant elements by checking the domains one after the other, has $O(dn)$ time complexity, where d is the maximum size of the domains and n is the number of variables. Thus, enforcing this consistency property ensures that all values of a variable satisfy all the unary constraints on that variable.
- A CSP is *arc-consistent* if for any pair of constrained variables x_i, x_j, for every value a in D_i there is at least one value b in D_j such that the assignment (x_i, a) and (x_j, b) satisfies the constraint between x_i and x_j, and viceversa. Any value in the domain D_i of variable x_i that is not arc-consistent can be removed from D_i since it cannot be part of any solution.

  Arc-consistency has become very important in CSP solving and it is in the heart of many CP languages. The optimal algorithms to make the CSP arc-consistent require time $O(ed^2)$, where e is the number of constraints (arcs in the constraint network) and d is the size of domains. Arc-consistency can also be easily extended to non-binary constraints.
- A CSP is *path-consistent*, if for every pair of values a and b for two variables x_i and x_j, such that the assignments of a to x_i and b to x_j satisfies the constraint between x_i and x_j, there exist a value for each variable along any path between x_i and x_j such that all constraints along the path are satisfied.

  When a path-consistent problem is also node-consistent and arc-consistent, then the problem is said to be strongly path-consistent.

Consistency techniques can be exploited during the forward checking stage of search algorithms. Each time some search decision is taken (for example, a value is assigned to the variable), the problem is made arc consistent. If failure is detected (that is, any domain becomes empty) then it is not necessary to instantiate other variables and backtracking occurs immediately.

Planning problems

Planning is an important aspect of rational behavior and it is a fundamental topic of artificial intelligence since its beginning. Planning capabilities are necessary for autonomous controlling of vehicles of many types including space ships (Muscettola et al. 1998) and submarines (McGann et al. 2008), but we can also find planning problems in areas such as manufacturing, games or even printing machines (Ruml et al. 2005). In this section we formally describe the planning problem and its representation.

Classical planning deals with finding a sequence of actions that transfer the world from some initial state to a desired state. The state space is large but finite. It is also *fully observable* (we know precisely the state of the world), *deterministic* (the state after performing the action is known), and *static* (only the entity for which we plan changes the world). Moreover, we assume the actions to be instantaneous so we only deal with action sequencing. Naturally, there exist extensions of planning problems dealing with durative and parallel actions with uncertain effects, the state of the word may not be fully known or may be changed by other entities such as nature. However, these extensions are out of scope of this introduction, for a detailed survey see (Ghallab et al. 2004).

Typically, the world *state* is described as a set of predicates that hold in the state, such as $location(robot_1, city_{23})$ saying that $robot_1$ is located in $city_{23}$. In other words, for

each predicate and for each state we describe whether the predicate holds in the state or not. This is called a classical representation of planning problems. *Actions* are described using a triple $(Prec, Eff^+, Eff^-)$, where $Prec$ is a set of predicates that must hold for the action to be applicable (preconditions), Eff^+ is a set of predicates that will hold after performing the action (positive effects), and Eff^- is a set of predicates that will not hold after performing the action (negative effects). For example, action $move(robot_1, city_{12}, city_{23})$, describing that $robot_1$ moves from $city_{12}$ to $city_{23}$, is specified as a triple $(\{location(robot_1, city_{12})\}, \{location(robot_1, city_{23})\}, \{location(robot_1, city_{12})\})$. Formally, action a is applicable to state s if $Prec(a) \subseteq s$. The result of applying action a to state s is a new state $\gamma(s, a) = (s - Eff^-(a)) \cup Eff^+(a)$. Notice that this description assumes a *frame axiom*, that is, other predicates than those mentioned among the effects of the action are not changed by applying the action. The set of predicates together with the set of actions is called a *planning domain*. We assume both sets of predicates and actions to be finite. The *goal* is specified as a set of predicates that must hold in the goal state, that is, if g is a goal then any state s such that $g \subseteq s$ is a goal state. The *classical planning problem* is defined by the planning domain, the initial state s_0, and the goal g, and the task of planning is to find a sequence of actions $< a_1, a_2, \ldots, a_n >$, called a *plan*, such that a_1 is applicable to the initial state s_0, a_2 is applicable to state $\gamma(s_0, a_1)$ etc., and $g \subseteq \gamma(\gamma(\gamma(s_0, a_1), a_2) \ldots, a_n)$.

There exists an alternative to the above logical formalism that is based on so called *multi-valued state variables*. For each feature of the world, there is a variable describing this feature, for example $location(robot_1, S)$ describes the position of $robot_1$ at state S. Instead of specifying the validity of the predicate in some state S, say $location(robot_1, city_{23})$, we can specify the value of the state variable in a given state, in our example $location(robot_1, S) = city_{23}$. Hence the evolution of the world can be described as a set of state-variable functions where each function specifies the evolution of the values of certain state variable. The actions are described as entities changing the values of state variables. We can still use preconditions specifying required values of certain state variables, but the positive and negative effects are merged to effects of setting the values of certain state variables. Notice that this multi-valued formulation is more compact than the logical formulation, where, for example, one needs to express explicitly that if $robot_1$ is in $city_{23}$ then it is not present in another location. For example, the action of moving $robot_1$ from $city_{23}$ to $city_{24}$ needs to explicitly describe (in negative effects) that, after performing the action, the predicate $location(robot_1, city_{23})$ is no more valid. In the multi-valued representation assigning value $city_{24}$ to state variable $location(robot_1, S)$ implicitly means that $robot_1$ is not at a different location at state S.

Scheduling problems

Scheduling concerns with the allocation of resources to activities with the objective of optimizing some performance measures. Depending on the situation, resources could be machines, humans, runways, processors etc., activities could be manufacturing operations, duties, landings and take-offs, computer programs etc., and objectives could be minimization of the schedule length, maximization of resource utilization, minimization of delays, and others.

Scheduling has been studied since 1950s when researchers were faced with problems of efficient management of operations in workshops (Leung 2004). The problems studied at that time were relatively simple and a number of efficient algorithms providing optimal solutions were proposed. In late 1960s, scheduling problems were encountered in the area of computer science where the problem of efficient utilization of scarce computational resources became very important. As scheduling problems became more complicated, researchers were unable to develop efficient algorithms for them and the proposed techniques were essentially exponential in time. This is not so surprising, since many scheduling problems have been shown to be NP-hard. Nowadays, an increased attention is paid to approximation and stochastic algorithms that can provide some solutions even to hard problems. The history of scheduling is reflected in the style of research in the area. The focus is almost always on solving a specific scheduling problem or showing its complexity rather than on providing a general scheduling approach. The result is that we have a huge number of scheduling algorithms for a large number of specific problems. This makes scheduling very different from the approach of planning where the focus is on solving general planning problems rather than developing ad-hoc techniques for particular planning problems.

Let us now introduce some basic scheduling terminology more formally. Typically, the scheduling problem consists of a set of n jobs that can run on m machines. Each job i requires some processing time p_{ij} on a particular machine j. This part of a job is usually called an operation. The schedule of each job is an allocation of one or more time intervals (operations) to one or more machines. The scheduling problem is to find a schedule for each job satisfying certain restrictions and optimizing given objectives. The start time of job j can be restricted by a *release date* r_j, which is the earliest time at which job j can start its processing. A release date models the time when the job arrives at the system. Similarly, a deadline d_j can be specified for job j, which is the latest time by which the job j must be completed. More frequently, a *due date* δ_j is specified for job j which is an expected time of job completion. The job can complete later (or earlier) the due date, but it will incur a cost. Let C_j be the completion time of job j. Then *lateness* of job j is defined as $L_j = C_j - \delta_j$ and the *tardiness* of job j is defined as $T_j = max(0, L_j)$.

Completion time, lateness, and tardiness are the typical participants in traditional objective functions expressing the quality of the schedule.

It is possible to specify precedence constraints between jobs, which express the fact that certain jobs must be completed before certain other jobs can start processing. In the most general case, the precedence constraints are represented as a directed acyclic graph, where each vertex represents a job and, if job i precedes job j, then there is a directed arc from i to j. If each job has at most one predecessor and at most one successor, then we are speaking about *chains*; they correspond to serial production. If each job has at most one successor then the constraints are referred to as an *intree*. This structure is typical for assembly production. Similarly, if each job has at most one predecessor, then the constraint structure is called an *outtree*. This structure is typical for food or chemical production where from the raw material we obtain several final products with different "flavors".

Each job (operation) requires certain resource(s) for its processing. Operations can be pre-assigned to particular resources and then we are looking for time of processing only. Or there may be several alternative resources to which the operation can be allocated, and then *resource allocation* is part of the scheduling task. Such alternative resources are either identical then we can union them into a so called *cumulative resource* that can process several operations in parallel until some given capacity, or the alternative resources may be different (for example processing time may depend on the resource). The resource, that can process at most one operation at any time, is called a *unary* or *disjunctive resource*. If processing of operation on a machine can be interrupted by another operation and then resumed possibly on a different machine, then the job is said to be *preemptable*.

There exists a special category of scheduling problems with a particular combination of precedence constraints, so called *shop problems*. If each job has its own predetermined route to follow, that is, the job consists of a chain of operations where each operation is assigned to a particular machine (the job may visit some machines more than once or never) then we are talking about job-shop scheduling (JSS). If the machines are linearly ordered and all the jobs follow the same route (from the first machine to the last machine) then the problem is called a *flow-shop problem*. Finally, if we remove the precedence constraints from the flow-shop problem, that is, the operations of each job can be processed at any order, then we obtain an *open-shop problem*.

As we mentioned above, the scheduling research focuses on solving specific scheduling problems, so it is crucial to have a good classification of scheduling problems. The most widely used notation to classify scheduling problems is the well known $\alpha|\beta|\gamma$ notation by Graham et al. (Graham et al. 1979). The α field describes the machine environment, for example whether there is a single machine (1), m parallel identical machines (Pm), job-shop (Jm), flow-shop (Fm) or open-shop (Om) environment with m machines. The β field characterizes jobs and scheduling constraints and it may contain none entry or multiple entries. The typical representatives are restriction of the release dates (r_j) or deadlines (d_j) for jobs, assuming precedence constraints ($prec$) or allowing pre-emption of jobs ($pmtn$). Finally, the γ field contains the objective function to optimize, usually a single entry is present but more entries are allowed. The widely used objectives are minimization of the maximal completion time—a so called $makespan (C_{max})$ or minimization of maximal lateness (L_{max}).

Constraint satisfaction in planning and scheduling problems

Constraint satisfaction is a general technology for solving combinatorial optimization problems and hence it is not surprising that this technology has been applied to planning and scheduling problems as well. Constraint-based planning is a discipline that studies how to solve planning problems by constraint satisfaction, while constraint-based scheduling deals with applying constraint satisfaction techniques to scheduling problems.

Constraint satisfaction techniques typically assume the problem to be fully specified in advance by fixed sets of variables and constraints. The scheduling problem with a known set of activities is static in the sense that the activities and hence the size of the solution are known in advance so the scheduling problem can be naturally mapped to a CSP. On contrary, the planning problem is dynamic in the sense that the activities and hence the size of the solution are unknown in advance. So it is more complicated to map a full planning problem to a CSP—usually a series of CSPs is necessary to solve the planning problem or constraints are used only to model some planning sub-problem such as temporal consistency.

Constraint satisfaction in planning

One of the difficulties of planning is that the length of the plan, that is, the set of used actions, is unknown in advance, so some dynamic technique which can produce plans of "unrestricted" length is required. Frequently, the shortest plan is being looked for, which is a form of *optimal planning*. As it has been shown in (Kautz and Selman 1992), the problem of shortest-plan planning can be translated into a series of logical satisfiability (SAT) problems, where each SAT instance encodes the problem of finding a plan of a given length. First, we start with finding a plan of length 1, and, if it does not exist, then we continue with a plan of length 2, etc., until the plan is found. There exist criteria to stop these extensions if

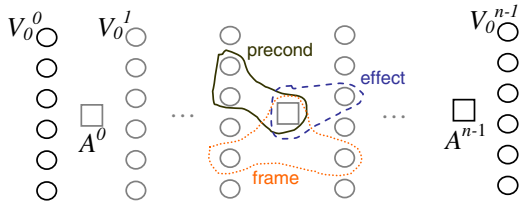

Fig. 2 Base decision variables and constraints modeling plans of length n

the plan does not exist (Ghallab et al. 2004), but for simplicity reasons in this paper we assume that a plan always exists.

Now, the problem of finding a plan of length n can be encoded as a CSP. The most important steps when designing a constraint model are the selections of variables, their domains, and finally constraints defining consistent tuples of the variables. We will present here the straightforward constraint model that has been described in (Ghallab et al. 2004) and a more advanced model called CSP-PLAN (Lopez and Bacchus 2003), both models for multi-valued state variables. For detailed comparison of these models and their further improvement, the interested reader is referred to (Barták and Toropila 2008).

We assume sequential planning where the world state is described using v multi-valued state variables, the instantiation of which exactly specifies a particular state. A CSP denoting the problem of finding a plan of length n consists of $n+1$ sets of above mentioned multi-valued variables, with the 1st set denoting the initial state and kth set denoting the state after performing $k-1$ actions, for $k \in \{2,\ldots,n+1\}$. We also need n action variables A^j, where j ranges from 0 to $n-1$, indicating the selected actions (Fig. 2).

The two above mentioned constraint models (straightforward and CSP-PLAN) differ in the set of constraints used to describe state transitions.

In the straightforward model we use logical constraints that connect two adjacent sets of state variables through the corresponding action variable between them, that is, for given s we connect state variable layers V_i^s and V_i^{s+1}, $i \in \{0,\ldots,v-1\}$, through the action variable A^s:

$$A^s = act \rightarrow Pre(act)^s, \forall act \in Dom(A^s), \quad (1)$$

$$A^s = act \rightarrow Eff(act)^{s+1}, \forall act \in Dom(A^s), \quad (2)$$

where $Pre(act)^s$ and $Eff(act)^{s+1}$ are conjunctions of equalities setting the values for required state variables corresponding to preconditions of action *act* in layer s, and its effects in layer $s+1$ respectively. We also need constraints representing the *frame axioms*, that is, constraints that would enforce equalities between those state variables V_i^s and V_i^{s+1}, which are not affected by the selected action A^s (the frame assumption is implicit in classical planning representations):

$$A^s \in NonAffAct(V_i) \rightarrow V_i^s = V_i^{s+1}, \forall i \in <0, v-1>, \quad (3)$$

where $NonAffAct(V_i)$ is the set of actions that do not have state variable V_i among its effects. Please note that this set depends purely on actions' definition and thus can be precomputed in advance.

The straightforward model uses separate constraints to describe preconditions and effects of actions and frame axioms specifying that the value of some state variable is not changed by the selected action. There is another more efficient way to describe how the state is changed by merging the constraints for effects and frame axioms into so called *successor state constraints* originally described in (Reiter 2001). This method has been used in CSP-PLAN (Lopez and Bacchus 2003) that was originally proposed as a constraint model for a so called planning graph (Blum and Furst 1997). In the planning graph, several parallel actions can be used in each layer provided that these actions do not influence each other. We will present here a simplified version of the constraint model with a single action per layer. The encoding of action preconditions is again using the constraints of type (1). However, in contrary to the straightforward model we use the successor state axioms instead of effects and frame axioms. In particular, for each possible assignment of state variable $V_i^s = val, val \in Dom(V_i^s)$, we have a constraint between it and the same state variable assignment $V_i^{s-1} = val$ in the previous layer. The constraint says that state variable V_i^s takes value *val* if and only if some action assigned this value to the variable V_i^s, or equation $V_i^{s-1} = val$ held in the previous layer and no action changed the assignment of variable V_i. Formally:

$$V_i^s = val \leftrightarrow A^{s-1} \in C(i, val) \vee \left(V_i^{s-1} = val \wedge A^{s-1} \in N(i)\right), \quad (4)$$

where $C(i, val)$ denotes the set of actions containing $V_i = val$ among their effects, and $N(i)$ denotes the set $NonAffAct(V_i)$ as described within the previous model.

Notice that the straightforward model uses constraints in the form of implication which is basically an abbreviation for disjunctive constraints. Disjunctive constraints are known for weak propagation so the constraint models with disjunctive constraints do not exploit constraint propagation a lot and hence search is the prevailing technique for solving such problems. The equivalence constraint in CSP-PLAN leads to a stronger domain filtering and hence this model is more efficient.

Designing a constraint model is just the first step to solve the problem using constraint satisfaction technology. The next step is defining the search strategy, that is, the order in which the variables are instantiated and the order in which the values are tried for the variables. One can use generic search techniques for CSPs, however, it is usually better to exploit the particular structure of the problem. First, one should realize that it is enough to instantiate just the action variables

A^s because when their values are known then the values of remaining variables, in particular the state variables, are set by means of constraint propagation. Of course, we assume that the values for state variables V_i^0 modeling the initial state were set and similarly the state variables V_i^n in the final layer were set according to the goal (the final state is just partially specified so some state variables in the final layer remain un-instantiated). The action variables can be instantiated in the increasing order A^0 to A^{n-1} to mimic the forward planning or in the decreasing order from A^{n-1} to A^0 to mimic regression (backward) planning. For value ordering (selection of action), one can use the planning heuristics designed for other planning algorithms (Ghallab et al. 2004).

In this section we gave two examples of constraint models for finding a plan of a given length. These models are generated automatically from the description of the planning domain and the planning problem. There also exist handcrafted constraint models, for example CPlan (van Beek and Chen 1999). All these models deal with sequential plans. Constraints are also used in partial order planners (the plan is a partially ordered structure of actions, it is a specific version of plan-space planning) such as CPT planner (Vidal and Geffner 2004). Last but not least, constraint satisfaction techniques are frequently applied to planning sub-problems such as temporal CSPs (Dechter et al. 1991) dealing with maintaining temporal relations between actions.

Constraint satisfaction in scheduling

Scheduling is a "killing application" for constraint satisfaction. The success of constraint-based scheduling in real-life applications is thanks to the fact that two research areas, namely operations research (OR) and artificial intelligence (AI), combined their complementary strengths in the constraint-based approach. The traditional OR approach to scheduling focuses on exploiting the combinatorial nature of a relatively simple mathematical model of the scheduling problem. This leads to a high level of efficiency when solving such problems. The drawback of this approach is that, when mapping the real problem to the mathematical model, usually some degrees of freedom need to be discarded and simplifying assumptions need to be taken. Discarding degrees of freedom may eliminate some interesting solutions while discarding side constraints may lead to unacceptable solutions. In contrast, the AI approach traditionally focuses on general problem-solving techniques so all degrees of freedom and side constraints are preserved which may have the disadvantage of poor performance when comparing to dedicated solving algorithms. Constraint satisfaction provides a very good framework for integrating OR techniques in more general AI solving algorithms. The key technology for this integration is based on the notion of *global constraints*. Global constraints encapsulate a certain part of the CSP and, rather than using a set of constraints to model this sub-problem, a dedicated "larger" global constraint is used that can exploit better the structure of the sub-problem. Global constraints together with sophisticated search techniques are the key power behind the success of constraint-based scheduling. Global constraints provide efficient algorithms to solve well-defined sub-problems while they can be still combined with other constraints modeling the side features of the problem.

Let us first describe the base constraint model of a scheduling problem. Recall that a scheduling problem is a decision problem where we are looking for when and where the jobs will be processed. Jobs typically consist of temporally connected operations that are the basic scheduling objects. For each operation, we can introduce three variables indicating its position in time, namely, the start time, the end time, and the processing time (duration). For operation A we denote these variables by $start(A)$, $end(A)$, and $p(A)$. We expect the domains for these variables to be discrete (for example natural numbers representing time) where the release date and the deadline of the operation make natural bounds for them. It is possible to further restrict the domain by assuming the time windows when the operation can be processed (time windows are a typical example of a real-life side constraint). Thought frequently the processing time of operation is a constant number and so the start time is enough to fully specify the allocation of operation to time, we prefer to use all three variables to simplify description of the constraints. Speaking about constraints, for operations without pre-emption, the following constraint connects the above three variables: $start(A) + p(A) = end(A)$. The preemptive case is slightly more complicated, the reader may look at (Baptiste et al. 2006, p. 765) for details. If resource allocation is a part of the scheduling problem, then one more variable is required to describe which resource will process the operation: *resource(A)*. Its domain equals the set of numbers, where the numbers are uniquely assigned to resources. The domain contains just the numbers indicating resources that can process a given operation.

Basically, there are two groups of constraints involved in the model: temporal and resource constraints. Temporal constraints describe the direct temporal relations between the operations such as the precedence relations. The relation that operation A must be processed before operation B can be modeled using the constraint: $end(A) \leq start(B)$. In the remaining text, we will denote this constraint as $A << B$. It is easy to generalize this constraint to model a more detailed temporal relation with minimal and maximal delays between the operations. Then the constraint has a form $min\_delay(A, B) \leq start(B) - end(A) \leq max\_delay(A, B)$. When the temporal constraint network is sparse, as is the usual case in scheduling, then standard arc-B-consistency (Lhomme 1993) is used to propagate these constraints. For more dense networks it is more appropriate to use path-consistency like algorithms.

Assume we have a unary (disjunctive) resource that can process at most one operation at any time. Two operations A and B processed on the unary resource cannot overlap in time, so either A precedes B or vice versa: $A << B \vee B << A$. The unary resource can be fully modeled by a set of such disjunctive constraints. Unfortunately, the disjunctive constraints do no propagate well (arc consistency does not prune a lot the domains of time variables). Nevertheless, as mentioned above, a set of disjunctive constraints can be encapsulated to a single global constraint where stronger domain pruning can be achieved by exploiting techniques for solving specific scheduling problems. Typically, the global constraints model resources, that is, all start times of all operations allocated to certain resource participate in the global constraint. Constraints modeling unary resources are frequently based on edge-finding technique (Baptiste and Le Pape 1996) deducing that certain operation must be processed first or last among the set of operations. Edge-finding algorithms with the time complexity as good as $O(n.log\ n)$ (Vilím et al. 2005), where n is the number of operations, exist. A complementary technique to edge-finding called "not-first/not-last" deduces that an operation cannot be processed first or last (Torres and Lopez 2000). (Vilím 2004) proposed a filtering algorithm with the time complexity $O(n.logn)$. Some of above mentioned techniques can be extended to cumulative resources, that is, discrete resources with capacity greater than one (more operations can be processed in parallel). For example, the paper (Baptiste et al. 2006) shows a cumulative version of the edge-finding technique. Other techniques have been proposed particularly for cumulative resources, for example the energy precedence propagation (Laborie 2003) combines information about precedence relations and limited capacity of the resource. As we already mentioned the big advantage of constraint models is their flexibility to combine constraints describing various aspects of the problem. So in addition to "standard" features of the problem modeled typically by global constraints, the user may define any other restriction on decision variables.

From the section on constraint satisfaction it should be clear that the constraint model is not enough to solve the problem; the constraint model needs to be accompanied by the search procedure that instantiates the variables. It is possible to use any search strategy developed for CSPs; however, the dedicated search strategies for a class of problems frequently give better results. For example, instead of instantiating the decision variables in the model, the scheduling search strategies are usually based on different branching schemes. As scheduling is basically about finding a sequence of operations, the branching is typically based on deciding which operation is processed before another operation. In particular, if operations A and B are processed on the same resource, we can decide which one will be processed first by exploring two alternatives $A << B$ or $B << A$. In (Baptiste et al. 1995) a different branching scheme for operation selection is studied. Rather than deciding about the order of two not-yet ordered operations, we can decide about the first operation in the resource—we are resolving the disjunction $A << \Omega \vee \neg A << \Omega$, where Ω is a set of operations allocated to the same resource as A. Heuristics for selection which disjunct in the above disjunctions should be tried first are frequently based on the notion of slack proposed in (Smith and Cheng 1993). Briefly speaking, slack describes the flexibility for allocating the operations and the disjunct with larger flexibility should be tried first.

This section gave some examples and techniques of using constraint satisfaction for solving scheduling problems. A more detailed survey can be found in (Barták 2005) and (Baptiste et al. 2006), where planning techniques are also covered. Probably the most comprehensive coverage of constraint-based scheduling is (Baptiste et al. 2001) where the filtering algorithms behind the propagation rules for various types of resources are described.

Integration of planning and scheduling

By planning we refer generally to the process of deciding *what* to do, that is to say, the process of transforming strategic objective into executable activity networks. On the other hand, the scheduling process decides *when* and *how*, that is, which resources to use to execute various activities and over what time frames. Thus, planning deals with goal, states and actions meanwhile scheduling deals with temporal and resource constraints.

In the traditional approach to managing complex systems, planning and scheduling are two very distinct phases: first, the planner generates a plan and then the scheduler ensures the feasibility and optimality on the plan. This gap has several drawbacks and, in a wide variety of real applications, this strict separation is not possible or beneficial. In many real cases, planning and scheduling are intertwined: time constraints affect which actions may be chosen, and also how they should be combined (Planken et al. 2008); during scheduling it is often necessary to make planning decisions (plan the setup of a machine); moreover planning decisions can benefit from scheduling information (choose a process plan depending on resource loads), etc.

The Crikey planner (Halsey et al. 2004) circumvents this problem by identifying which parts are separable, and which are not. In the parts that are non-separable the casual and temporal problems are solved together; the separable parts are treated as separate problems. Whereas some researchers try to decouple planning and scheduling as much as possible (Halsey et al. 2004; Srivastava and Kambhampati 1999b), other works are based on the integration of them.

From the planning perspective, planning and scheduling models can be classified into two categories: (a) temporal planning and (b) integration of planning and scheduling models. The first group includes planners which can deal with time and resources, and we can distinguish two different approaches: planning systems with embedded scheduling sub-systems like LPG 1.2 (Gerevini and Serina 2000), MIPS-XXL (Edelkamp et al. 2006), LPG-TD (Gerevini et al. 2006); and scheduling systems with embedded planning sub-systems such as CPT (Vidal and Geffner 2006). The second group aims to exploit the advantages of both processes by sharing heuristics, space search reduction and improvements of the overall performance. However this integration is neither easy nor intuitive and there are some trends to combine these two processes which try to overcome the gap between them (Smith et al. 1996). First attempt tend to include a temporal reasoning process that carries out the scheduling process inside the own planning process (Garrido et al. 2000). Nevertheless, the temporal reasoning needed in real problems is too complex to be carried out in the same planning process so that it remains difficult to deal with the existing constraints and optimize the plan. Furthermore, it is difficult to determine when the system is planning or it is scheduling (Barták 2000). Many other attempts redefine both processes, but some of them are only useful when no complex availability constraints exist on share resources (Srivastava and Kambhampati 1999a). Many other interesting attempts can be found mainly from the works of Muscettola (Muscettola 1993), Smith (Smith et al. 2000), etc.

Acknowledgements Roman Barták is supported by the Czech Science Foundation under the contract 201/07/0205. Miguel A. Salido is supported by the research project TIN2004-06354-C02-01 (Min. de Educacin y Ciencia, Spain-FEDER), GV/2007/274 (Generalidad Valenciana, Spain).

References

Apt, K. R. (2003). *Principles of constraint programming*. Cambridge: Cambridge University Press.

Baptiste, P., Laborie, P., Le Pape, C., & Nuijten, W. (2006). Constraint-based scheduling and planning. In *Handbook of Constraint Programming* (pp. 761–799). Amsterdam: Elsevier.

Baptiste, P., & Le Pape, C. (1996). Edge-finding constraint propagation algorithms for disjunctive and cumulative scheduling. In *Proceedings of the 15th Workshop of the UK Planning and Scheduling Special Interest Group*, Liverpool, UK.

Baptiste, P., Le Pape, C., & Nuijten, W. (1995). Constraint-based optimization and approximation for job-shop scheduling. In *Proceedings of the AAAI-SIGMAN Workshop on Intelligent Manufacturing Systems, IJCAI-95*, Montreal, Canada.

Baptiste, P., Le Pape, C., & Nuijten, W. (2001). *Constraint-based scheduling*. Springer.

Barták, R. (1998). On-line guide to constraint programming. http://kti.mff.cuni.cz/~bartak/constraints/index.html.

Barták, R. (2000). Toward mixed planning and scheduling. In *Proceedings of CPDC'00 Workshop (Invited Talk)*, Stenungsund, Sweden.

Barták, R. (2005). Constraint satisfaction for planning and scheduling. In I. Vlahavas & D. Vrakas (Eds.), *Intelligent techniques for planning* (pp. 320–353), Hershey, PA: Idea Group.

Barták, R., & Toropila, D. (2008). Reformulating constraint models for classical planning. In *Proceedings of the 21st International Florida AI Research Society Conference (FLAIRS 2008)*, Florida, USA, pp. 525–530.

Bitner, J. R., & Reingold, E. M. (1975). Backtracking programming techniques. *Communications of the ACM 18*, 651–655.

Blum, A., & Furst, M. (1997). Fast planning through planning graph analysis. *Artificial Intelligence, 90*, 281–300.

Dechter, R. (2003). *Constraint processing*. San Mateo, CA: Morgan Kaufmann.

Dechter, R., Meiri, I., & Pearl, J. (1991). Temporal constraint network. *Artificial Intelligence, 49*, 61–95.

Edelkamp, S., Jabar, S., & Nazih, M. (2006). Large-scale optimal PDDL3 planning with MIPS-XXL. In *5th International Planning Competition Booklet (IPC-2006)*, Lake District, England, pp. 28–30.

Frost, D., & Dechter, R. (1994). Dead-end driven learning. In *Proceedings of the National Conference on Artificial Intelligence*, Seattle, USA, pp. 294–300.

Garrido, A., Salido, M. A., & Barber, F. (2000). Scheduling in a planning environment. In *Proceedings of ECAI-2000 Workshop on New Results in Planning, Scheduling and Design (PUK-2000)*, Berlin, Germany, pp. 36–43.

Gaschnig, J. (1977). A general backtrack algorithm that eliminates most redundant tests. In *Procceedings of IJCAI*, Cambridge, MA, USA, pp. 457.

Gaschnig, J. (1979). *Performance measurement and analysis of certain search algorithms*. Technical Report CMU-CS-79-124, Carnegie-Mellon University.

Gerevini, A., Saetti, A., & Serina, I. (2006). An approach to temporal planning and scheduling in domains with predictable exogenous events. *Journal of Artificial Intelligence Research, 25*, 187–231.

Gerevini, A., & Serina, I. (2000). Fast plan adaptation through planning graphs: Local and systematic search techniques. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems*, Breckenridge, CO, USA, pp. 112–121.

Ghallab, M., Nau, D., & Traverso, P. (2004). *Automated planning: Theory and practice*. San Francisco, CA: Morgan Kaufmann.

Graham, R. L., Lawler, E. L., Lenstra, J. K., & Rinnooy-Kan, A. H. G. (1979). Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics, 5*, 287–326.

Halsey, K., Long, D., & Foz, M. (2004). CRIKEY—A temporal planner looking at the integration of planning and scheduling. In *Proceedings on the ICAPS 2004 Workshop on Integrating Planning and Scheduling*, Whistler, Canada, pp. 46–52.

Haralick, R., & Elliot, G. (1980). Increasing tree efficiency for constraint satisfaction problems. *Artificial Intelligence, 14*, 263–314.

Kautz, H., & Selman, B. (1992). Planning as satisfiability. In *Proceedings of ECAI*, Vienna, Austria, pp. 359–363.

Laborie, P. (2003). Algorithms for propagating resource constraints in AI planning and scheduling: Existing approaches and new results. *Artificial Intelligence, 143*, 151–188.

Leung, J. Y. T. (2004). *Handbook of scheduling: Algorithms, models, and performance analysis*. Boca Raton, FL: Chapman & Hall.

Lhomme, O. (1993). Consistency techniques for numeric CSPs. In *Proceedings of 13th International Joint Conference on Artificial Intelligence*, Chambéry, France, pp. 232–238.

Lopez, A., & Bacchus, F. (2003). Generalizing GraphPlan by formulating planning as a CSP. In *Proceedings of IJCAI*, Acapulco, Mexico, pp. 954–960.

McGann, C., Py, F., Rajan, K., Ryan, J., & Henthorn, R. (2008). Adaptive control for autonomous underwater vehicles. In *Proceedings of AAAI'08*, Chicago, USA, pp. 1319–1324.

Michalewicz, Z., & Fogel, D. B. (2000). *How to solve it: Modern heuristics.* Berlin: Springer.

Muscettola, N. (1993). *HSTS: Integrating planning and scheduling*. Technical Report CMU-RI-TR-93-05, Robotics Institute, Carnegie Mellon University.

Muscettola, N., Nayak, P., Pell, B., & Williams, B. (1998). Remote agent: To boldly go where no AI system has gone before. *Artificial Intelligence, 103*, 5–47.

Planken, L., de Weerdt, M., & Van der Krogt, R. (2008). P^3C: A new algorithm for the simple temporal problem. In *Proceedings of ICAPS-2008*, Sydney, Australia, pp. 256–263.

Prosser, P. (1993). Hybrid algorithm for the constraint satisfaction problem. *Computational Intelligence, 9*, 268–299.

Reiter, R. (2001). *Knowledge in action: Logical foundations for specifying and implementing dynamic systems*. Cambridge, MA: MIT Press.

Rossi, F., Van Beek, P., & Walsh, T. (2006). *Handbook of constraint programming*, Amsterdam: Elsevier.

Ruml, W., Do, M. B., & Fromherz, M. (2005). On-line planning and scheduling for high-speed manufacturing. In *Proceedings of ICAPS'05*, Monterey, USA, pp. 30–39.

Ruttkay, Z. (1998). Constraint satisfaction—A survey. *CWI Quarterly, 11*(2&3), 123–162.

Smith, D. E., Frank, J., & Jonsson, A. K. (2000). Bridging the gap between planning and scheduling. *Knowledge Enginering Review, 15*, 47–83.

Smith, S. F., & Cheng, Ch.-Ch. (1993). Slack-based heuristics for constraint satisfaction scheduling. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, Washington, USA, pp. 139–144.

Smith, S. F., Lassila, O., & Becker, M. (1996). Configurable, mixed-initiative systems for planning and scheduling. In A. Tate (Ed.), *Advanced planning* (pp. 235–241). AAAI Press.

Srivastava, B., & Kambhampati, S. (1999a). Efficient planning through separate resource scheduling. *AAAI Spring Symposium on Search Techniques for Problem Solving under Uncertainty and Incomplete Information*, Orlando, USA.

Srivastava, B., & Kambhampati, S. (1999b). Scaling up planning by teasing out resource scheduling. In *Proceedings of the 5th European Conference on Planning: Recent Advances in AI Planning*, Durham, UK, pp. 172–186.

Torres, P., & Lopez, P. (2000). On Not-First/Not-Last conditions in disjunctive scheduling. *European Journal of Operational Research, 127*, 332–343.

Tsang, E. (1993). *Foundation of constraint satisfaction*. London: Academic Press.

van Beek, P., & Chen, X. (1999). CPlan: A constraint programming approach to planning. In *Proceedings of AAAI-99*, Orlando, USA, pp. 585–590.

Vidal, V., & Geffner, H. (2004). Branching and pruning: An optimal temporal POCL planner based on constraint programming. In *Proceedings of AAAI-04*, San Jose, USA, pp. 570–577.

Vidal, V., & Geffner, H. (2006). Branching and prunning: An optimal temporal pocl planner based on constraint programming. *Artificial Intelligence, 170*, 298–335.

Vilím, P. (2004). O(n log n) Filtering algorithms for unary resource constraint. In *Proceedings of CPAIOR*, Nice, France, pp. 335–347.

Vilím, P., Barták, R., & Cepek, O. (2005). Extension of O(n log n) filtering algorithms for the unary resource constraint to optional activities. *Constraints, 10*(4), 403–425.