Models for Global Constraint Applications

Helmut Simonis

September 2, 2006

Abstract

In this paper we give an overview of some industrial applications built using global constraints. We look at three systems from different application domains and show the core models used to express their constraints. We also consider different search strategies that have been applied and discuss some of the application aspects.

Contents

1 Introduction

Constraint programming [6] always has had a strong link to applications, with some of the early work driven by case studies and comparisons with other problem solving techniques [68, 27]. An overview of early CHIP [24] applications was already given in 1998 [28]. But it was some time before industrial applications of constraint programming did appear, with the earliest complete system probably described in [38]. This delay was not just caused by the on-going development of the solver technology, but also by the need to build application frameworks which allowed to put constraint applications into an end-user's hands [37]. Some of these early applications suffered from limitations of the constraint engines, so that very large scale or highly combinatorial problems were not solved satisfactorily, leading for some to disappointment with the technology [23].

An alternative was to look for ways to improve the reasoning power of the constraint systems, and of these approaches, the introduction of global constraints [2, 43, 13] was probably the most successful. Global constraints serve two purposes: on one hand, they allow more constraint propagation than a collection of primitive constraints, on the other hand they also simplify modelling of problems by providing high level constraint abstractions, which simplify the construction of large scale models.

But with this added functionality also comes added complexity. Which model of a problem is the best one, and which problems can be modelled with constraints in the first place? There are a number of papers which consider constraint applications [71, 51, 41, 67], but they are more descriptive than prescriptive.

In recent years, research on global constraints has gained much in popularity. There are many papers which deal with the algorithmic aspects of specific global constraints, improving complexity bounds or implementing a different form of consistency for a known constraint. There are also papers which introduce new global constraints, or extend old ones in novel ways. Finally there is an extensive study of generic methods for building global constraints from abstract properties. The global constraint catalog [12] contains 235 entries (as of May 2005). But there is relatively little work reported on what to do with them [64, 46, 66, 17, 61, 49, 16, 48, 58]. A series of tutorials at the PAP/PACT/PACLP series of conferences on modelling with global constraints is not readily accessible [53, 54, 55, 56, 57].

This paper tries to present a few case studies on how global constraints can be used in modelling and solving complex problems. It complements the overview [63], which describes some of the system aspects of other constraint applications.

The paper is structured in the following way: We are now at the end of the introduction, which provided some background on global constraints and the applications that use them. We will next look at three case studies from commercial constraint applications, and see how global constraints make it possible to build very concise, yet powerful models for them.

- The first (section 2) is the *APACHE* [25] system, an airport stand allocation application which was initially built (without global constraints) as a demonstrator for AirFrance, and which eventually was implemented at an airport in Korea [63].

- The second case study (section 3) is the *MOSES* feed mill scheduling system [55, 4], a complex, multi-resource production scheduling system with an interactive problem solver.
- The third study (section 4) is an instance of a nurse rostering system called *GYMNASTE* [74, 21, 22, 20] developed in France and in practical use in hospitals there. It is one instance of personnel planning and assignment systems, which have been very successfully implemented using constraint programming.

For each case study, we present the problem in general terms, then discuss the model by describing variables and constraints, mention possible additional constraints and look at possible search routines for this problem type.

In section 5 at the end of the paper we try to extract common characteristics of the problems, which should be similar for many other constraint applications as well.

2 APACHE - Airport Stand Allocation

This form of assignment problem arises in many transportation systems. We present here the model for airport stand allocation, but similar problems are also common in train operations (dynamic platform allocation for train stations) and for harbour operations (berth allocation).

2.1 Problem

The description of the problem is based on [25], the model presented here has been first discussed in [54]. For flights arriving at and departing from an airport, we have to allocate parking positions, which are needed from the time of arrival to the time of departure. Many of the parking positions (also called *stands*) are located at a terminal, with a walkway connecting the plane parked in this location to a gate in the terminal. There are often other parking positions off the terminal (on the *apron*), where planes can be parked for longer periods, but where passengers must be transported by mobile lounges or buses, causing cost, inconvenience and often delay. At most airports, the set of possible stands is a predetermined, finite set, making this problem an ideal candidate for finite domain constraint programming.

Not every plane can be parked at every location: There are physical constraints due to plane size and type, or the walkway may be incompatible with the particular aircraft type. Often, certain parts of the terminal are reserved for specific airlines, and usually national and international flights must be kept separate due to customs regulations. There are also typical security restrictions, where flights from/to certain countries must be handled via more secure areas of the airport. Given information about the aircraft, its type and flight numbers of arriving and departing flights it is easy to come up with the set of possible stands that can be used for one particular airplane. We often also have preferences for particular stands.

The main objective is to come up with a feasible solution, one which satisfies all hard constraints. We then try to find a solution which reduces operational

cost or passenger movements. A typical rule is to minimise the number of passengers that have to be transported with buses. But many of the preferences are perceived differently by different stake-holders in the operations: Competing airlines only consider their own flights, for which they expect preferential treatment by the airport. Ground operations may complain about resource requirements, even if passenger satisfaction increases.

Another aspect of the allocation problem is that planes only directly compete for resources if their stay at the airport overlaps in time. The stand allocation system is not allowed to shift flights in time, but must take arrival and departure times as given. When delays, cancellations or early arrivals change the set of planes to be handled, the problem must be resolved incrementally. Much of the current assignment can no longer be changed, since planes are already parked, or passengers have been told about a gate assignment. Some of the changes may create additional cost, which has to be balanced against smooth operations without causing delays to flights.

How difficult is this problem? If every plane can be placed on every stand, then it reduces to graph colouring in an interval graph, for which easy polynomial algorithms are sufficient. If the possible stands for the planes are more restricted, then the problem becomes hard. But problem sizes are often quite manageable with planes and stands in the low hundreds. Except for extreme situations, there are normally enough stands to place all aircraft somewhere on the apron. On the other hand operational costs and delays may increase dramatically if direct gate connections are not used well. Since the number of buses and drivers is limited, these resource constraints will create a bottleneck if many planes are placed on the apron.

An important consideration in solving this problem effectively is the question whether it can be decomposed into smaller, independent subproblems. Most large airports consist of multiple terminals, which are largely run as independent operations. If at all possible, the overall stand allocation problem should be decomposed into allocation problems for individual terminals, each dealing with its own set of constraints. Any interaction between the problems (for example "borrowing" a stand at a different terminal for a period of time), would then be treated as a temporary modification of the sub problems.

2.2 Model

We present three different models for this application type, starting with a naive binary constraint model, which is easily changed into a model using multiple *alldifferent* constraints and finally a model using a single *diffn* constraint. We begin with some problem definitions.

The set of all planes to be allocated is denoted by I. For a plane i, we are given fixed arrival and departure times s_i and e_i, defining the duration d_i of the stay at the airport. The possible return of a plane to the airport later on in the day is considered to create another, independent instance of a parking demand. The set P_i defines the set of all possible parking positions for plane i, which typically is derived by a complex rule set from flight information data and operational rules of the airport. For every possible stand assignment, we also can define operational costs $C_i : P_i \mapsto \mathbb{N}$ and assignment preferences $Q_i : P_i \mapsto \mathbb{N}$ which are given by extension. As cost we can for example count how many passengers would need to be transported by bus, if the plane is not

connected directly to the terminal. A way of calculating preferences is described in section 2.5.

The allocated stand for a plane is given by variable y_i whose domain is the set P_i. We also introduce domain variables for the cost c_i and the assignment preference q_i of a plane i.

We first define the domain of the variables and link assignment, cost and preference variables. The *element(X,L,C)* constraint states that C is equal to the X^{th} element in list L.

$$\forall_{i \in I} : y_i \in P_i \tag{1}$$

$$\forall_{i \in I} : \text{element}(y_i, C_i, c_i) \tag{2}$$

$$\forall_{i \in I} : \text{element}(y_i, Q_i, q_i) \tag{3}$$

The objective is to find a feasible solution which minimises operational cost

$$\min \sum_{i \in I} c_i \tag{4}$$

This leaves us with the most important constraint, stating that we never have two planes occupying the same stand. We present three alternatives.

2.2.1 Model 1

In our first model, the non-overlap condition is expressed by binary disequalities. For every pair of parking periods that overlap in time, we have to state that the planes may not be parked in the same location.

$$\forall_{i,j \in I} : \max(s_i, s_j) < \min(e_i, e_j) \Rightarrow y_i \neq y_j \tag{5}$$

This model does not allow much constraint propagation, as we can only use forward checking on the binary constraints.

2.2.2 Model 2

We can replace the binary constraints by a collection of *alldifferent* constraints. This is based on the observation that all binary constraints are covered if we consider the sets of planes which are at the airport at each time an airplane arrives.

$$\forall_{t \in \{s_i | i \in I\}} : \text{alldifferent}(\{y_i | s_i \leq t < e_i\}) \tag{6}$$

The *alldifferent(L)* constraint states that the elements of list L are pairwise different.

The set of constraints usually can be further reduced by checking for subsumption. If the set of variables in one constraint is a subset of another constraint's variables, we can remove the smaller set. By generating the constraints systematically in temporal sequence we can easily check for this subsumption.

This is exemplified in figure 1. It shows five planes y_1 to y_5 with their arrival and departure times and assumed sets of possible stands 1..4 for y_1, y_2 and y_3, $\{1, 2\}$ for plane y_4 and $\{1, 3\}$ for plane y_5. Of the five time points a to e we only have to consider d and e, since a, b and c are subsumed by d.

Note that even in the reduced set some binary constraints between planes may be covered by more than one *alldifferent* constraint, in figure 1 the overlap

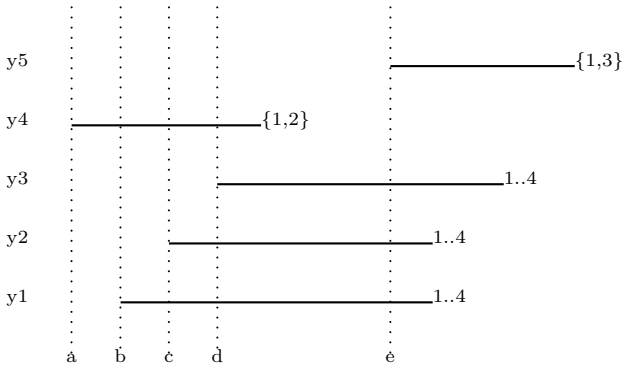

Figure 1: Example Problem

between y_1, y_2 and y_3 is handled both at time point d and e. This redundant expression is acceptable when compared to the increase of propagation this model allows.

For the *alldifferent* constraint we can choose between different propagation algorithms with varying reasoning power and computational complexity [69]. Which selection will be more successful will depend on the particular problem instance. If the domains are very sparse and vary significantly between planes, then a hyper-arc consistent version may provide more propagation, which offsets its higher complexity. If domains are large and quite similar, then a fast, bound-consistent implementation will be more effective.

But even if we use hyper-arc consistency for the *alldifferent* constraints, there are some domain reductions we can't perform since we do not consider the interaction of multiple constraints. We again use figure 1 as an example. Planes y_4 and y_5 must be assigned the same value, in this case 1, as it is the only common value in their domains. But enforcing hyper-arc consistency on sets d and e does not detect this. Figure 2 shows two bipartite matchings for the two constraints, but there are no compatible solutions unless both y_4 and y_5 are assigned to 1.

2.2.3 Model 3

Our third alternative model of the non-overlap constraint uses the *diffn* constraint of CHIP [13]. The constraint expresses that n-dimensional recti-linear objects do not overlap. We model each parking event as a two dimensional rectangle with origin at (s_i, y_i), length d_i and height 1. The non-overlap constraint then is written as

$$\text{diffn}(\{\langle s_i, y_i, d_i, 1\rangle | i \in I\}) \tag{7}$$

It is up to the constraint designer to decide whether he wants to use generic non-overlap reasoning [14], or whether the constraint internally is further specialised to handle this particular assignment structure of the constraint, where x-dimension, width and height are constant. It is then possible to transparently

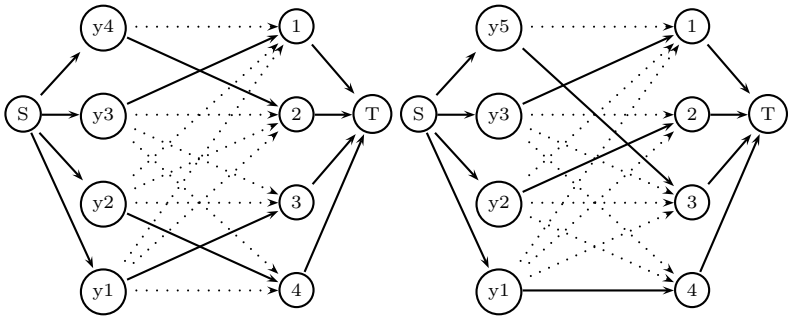

Figure 2: Example Matchings

include reasoning like [5] which looks at the interaction of multiple *alldifferent* constraints.

2.3 Additional Constraints

So far we have described only the core constraint of the stand allocation problem, in many practical situations we may encounter a variety of additional constraints.

2.3.1 Sharing Space

The space around an airport terminal can be used more effectively if the size of the planes is taken into account. A gate (say gate A12) that has enough room for one large plane might be used alternatively for two smaller ones (gates A12a and A12b). This can be easily modelled by using a single value v_1 for A12 in the domain for large planes and two additional values (v_2 and v_3 for A12a and A12b) in the domain of smaller planes. We now have to extend the non-overlap condition to sets of values by adding *atmost* constraints [27] for each relevant timepoint. We need to express that either A12 may be used, or A12a and/or A12b. If a y variable represents a large plane, we include it twice in our list of variables, otherwise we include it only once.

$$\text{atmost}(2, [y_1, y_2, \ldots, y_i, y_i, \ldots y_n], [v_1, v_2, v_3]) \tag{8}$$

The atmost constraint states that two of the given variables can take values in the set $\{v_1, v_2, v_3\}$.

2.3.2 Cumulative Resource Limits

If a plane is parked on the apron, then we will need buses to transport passengers from and to the plane. Their number will depend on the number of passengers on the plane. If too many planes require buses at the same time, we will encounter a resource bottleneck. It is tempting to introduce cumulative resource constraints [2] to model this restriction, but this may stop us from finding any feasible solution, if the resource limits are too tight.

2.3.3 Moving planes

Our model so far assumed that planes are parked in the same location for their entire stay at the airport. In certain situations a plane might be moved by a tractor to another stand, in order to free up space at the terminal or because the plane is required at a different position. The tractors are also required to "push-back" departing flights from the gate, since turbine engines can not be used to reverse the plane, and also to move planes on the ground more economically. If additional plane movements are allowed, we gain a degree of freedom in deciding when the plane should be moved. But this is best decided heuristically before the constraint model is set-up, since otherwise the non-overlap condition can no longer be expressed with the given models. We would also have to consider cumulative resource constraints for the tractors.

2.4 Incremental Resolving

We have assumed that the start and end times of the parking activities are constants when the model is set-up. In practice, these times are constantly changing, since planes arrive or depart later than predicted or planes arrive before their scheduled arrival time. This may invalidate the previous solution, and we have to re-run the solver. Many elements of the previous solution must be kept; all planes currently on a stand must continue to stay there in an updated solution. Some part of the future schedule may be fixed as well: if passengers already have been sent to a gate, it is quite difficult to reverse that decision.

We also can make our solution more robust by introducing safety margins in the overlap constraints. We extend each parking period by some margin before its arrival and/or after its departure and compute solutions using these extended time intervals. If flight times change only slightly during the day, then the previous solution will still be feasible. Choosing the right compromise between cost optimisation and safety margins is an important interactive choice, which also depends on the current state of operations.

2.5 Search

The choice of an appropriate search routine is essential for solving large scale problems, even if the constraint model is quite powerful. The problem size typically prohibits an exhaustive search of all possible assignments to find an optimal solution, especially since the solving times should be in seconds due to the rapid change of the underlying data. The search routine was refined in a number of steps, beginning with a static ordering.

In a first version, the planes are ordered statically by decreasing number of passengers, and the domains are enumerated such that stands at the terminals are tested first. This can be done by enumerating the q_i variable of a plane before fixing its y_i variable. A feasible solution (often of acceptable quality) will be found without backtracking, but search for the optimal solution leads to shallow backtracking.

The next improvement is to choose the stand more wisely. For this, we statically calculate an indicator $demand_j$ which describes how many planes want to use a given stand j at a terminal. For this calculation we restrict the domains

to stands connected to the terminal.

$$demand_j = \sum_{i \in I} \frac{j \in P_i}{|P_i|} \tag{9}$$

We order the domain of a plane by increasing demand, choosing first those stands which are not so useful for other planes. All stands on the apron are given a very low preference, ordered by increasing distance to the terminal. The assignment will now choose to label the preference variable first, and then the assignment variable. This typically will improve the quality of the first solution, but does not really improve the backtracking behaviour.

One reason for this shallow backtracking is the "near symmetry" of many of the possible values. In a terminal, many gates have similar characteristics, and on the apron, there is little to distinguish neighboring stands. Note that this is not symmetry in the proper sense, since an existing partial assignment will introduce small differences between the stands. It can still be worthwhile to remove these equivalence classes by committing to a single representative of each preference class. We still label by preference first, and then try to choose a stand with that preference. If we find one, we will not evaluate any other stand with the same preference on backtracking, but instead choose another, lower preference. This leads to a drastic reduction in the search space, at the cost of making the search incomplete, so we may miss the optimal solution.

If the search space is still too big for complete enumeration in the given time horizon, we can use a partial search routine like credit-based search [10] or limited discrepancy search [32]. This will allow us to explore more significant alternatives in the search space.

2.6 Background

The first industrial constraint application, developed for the HIT container harbour in Hong Kong [38], was quite similar to the problem described here. The objective was to allocate berths for ships in the harbour, so that loading and unloading times are minimised by providing the necessary resources and stacking space.

The *APACHE* system was a demonstrator developed by COSYTEC with AirFrance for terminal 2 at Charles de Gaulle airport in Paris. The initial model used arc-consistent *alldifferent* constraints, similar to model 2 in section 2.2.2. Model 3 using the *diffn* constraint and the different search strategies were described in [54].

Another instance of this application type has been developed at one of ISAB's Italian refineries. It deals with the unloading/loading of crude oil tankers and barges at an oil refinery. Due to the high running costs of such ships it is important that the operation starts as soon as the ship arrives in the harbour.

3 MOSES - Feed Mill Scheduling

Scheduling has long been the main application domain for constraint programming, with a significant number of industrial applications being developed by different teams around the world. We present here a rather complex production scheduling system for animal feed mills, which was developed by COSYTEC and

installed in a number of mills in the UK. This application was first described in [4], aspects of the model were discussed in [55].

3.1 Problem

Feed mills are large factories which produce hundreds of tons of products on a daily basis in a highly competitive market. Feed is produced locally, with farmers calling up the mill during the day and expecting a delivery on the next morning. They can select products from a large catalog, where each product type is highly specialised for a particular animal species. The product can either be a mixture of ground and treated raw materials (called *meals*) or is heated and put under pressure through a press, where the ingredients bond in forms of cylindrical *pellets* or *rolls* of various sizes. The bill of material for one product may contain ten or more elements, some of which can be highly dangerous to other animal species.

A typical layout of a feed mill is shown in figure 3, the flow of production is top down. Raw materials are stored in large bins from which they are drawn to a grinder/mixer, where a they are broken up and mixed with other ingredients. The resulting mixture can then either be used as a meal product or is temporarily stored in a number of pre-press bins. From there they are fed into the presses where, under pressure and temperature, the mix is formed into pellets and rolls.

The size of the product is determined by the die size inside the press, which is part of the press configuration. Changing dies is a long and personnel intensive process (the dies weigh over a hundred kilograms), while changing from one product to another which uses the same die requires much less time.

Not all products can be made on all presses, and some products have strong preferences for certain equipment, where particular pressure and temperature settings can be achieved. Machine throughput will depend on product and press combinations, and the length of production runs. The sequence of products on one press must be tightly controlled: We not only have to consider setup times, there are also forbidden or unwanted sequences. A forbidden step is when there is a contamination risk from the first to the second product. Some products (for example for pigs) are harmful for other species, to avoid a contamination risk we do not allow to follow them with feed for such species. For others, the temperature and press settings are so different that the quality of the product may suffer, creating more scrap material, and we try to avoid the sequence if possible.

From the presses the finished product is fed through a system of conveyors and lifts into finished product bins, each holding many tons of product, or to movable tote bins of much smaller capacity. Not all connections from presses to bins are possible and/or preferred. The number and capacity of bins is not large enough to make products for stock, products are made to order and only stay in the bins until picked up or delivered to the customer.

From the finished product bins there is another set of connections to the loading stations, where product is dumped directly into lorries, or to a bagging station, where smaller production runs are put into bags for delivery or distribution.

Typical problem sizes are several hundred products, 30-50 raw material bins, 2-3 mixers, 3-5 presses, 20-70 finished product bins, over 100 tote bins, and 2-5

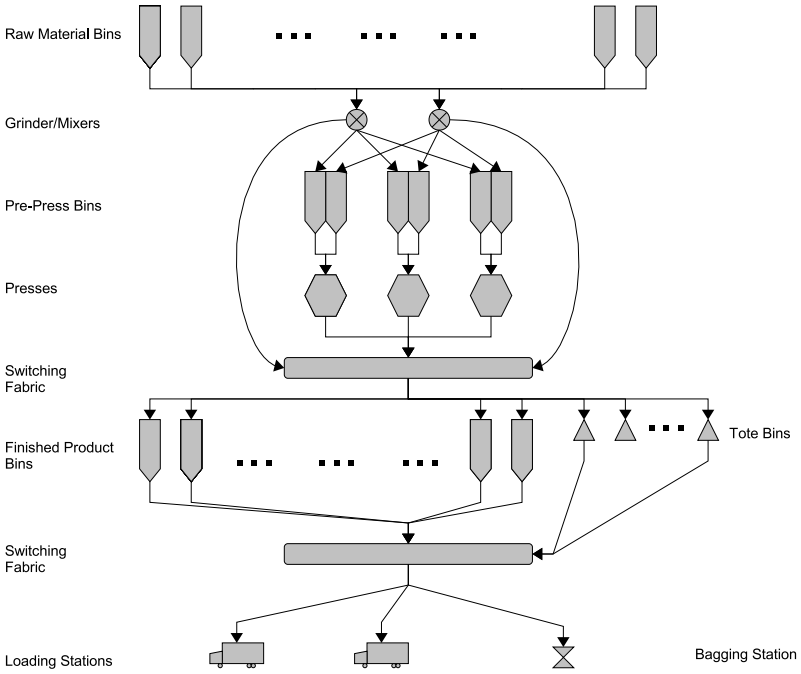

Figure 3: Feed Mill Layout

loading and bagging stations. The planning horizon is two days with a one minute time resolution, deciding in the afternoon what the production sequence should look like for the night and during the next day.

3.2 Model

The most important design decision is how to break this large problem into more manageable pieces. It may be possible to model the complete production process as one large constraint system, but it is unlikely that this would lead to a high quality solution. Observing the manual scheduling process in place and after a series of discussions with the plant managers, we split the overall production process into three components:

- Raw material, Mixing and Pre-Press bins
- Press Schedule
- Finished Product and loading schedule

The most important part centres on the schedule of the presses and a virtual resource, the meal line, for meal production. A schedule of these resources can then be used to determine the operations on the raw material side, and on the finished product part or the factory. The other elements of the schedule impose a number of constraints on the press schedule, but if these are satisfied, it is possible to construct their schedule from the press schedule.

3.2.1 Variables and Basic Constraints

The press schedule is a typical multi-machine production scheduling problem with machine and task dependent setup times, due dates and release dates for tasks, and multiple cumulative resource constraints.

As the schedule is customer driven and products are made "to order", we create a task for each order that falls within the planning horizon. That order is for a specific product and for a given quantity, and has a due date which is given by the promised delivery date to the customer. Via the bill of materials and the raw material stock information we can see if all raw materials are available; if not, this imposes a release date associated with the estimated arrival time of the materials. To describe the constraints we need to introduce some notation, it may be useful to refer to figure 4 to understand the link between the different variables.

The set of all n tasks is denoted by I, the set of all m machines by M. The planning horizon ranges from 0 to time end. The last assigned machine is given by the variable $maxm$ and the end of the schedule by the variables $maxe$. The product made in task i is called r_i.

Each task i is assigned to machine m_i and has a start time s_i, a duration d_i and an end of production e_i. After this end, there is the setup time $setup_i$, during which the machine is not available for production until time point l_i. We use a weight variable w_i to denote the period between s_i and l_i. We also introduce two variables $pred_i$ and $succ_i$ to describe the predecessor and successor tasks for task i on machine m_i. A machine preference variable p_i is used to express preferences of tasks to machines, based on product quality and machine throughput for the product.

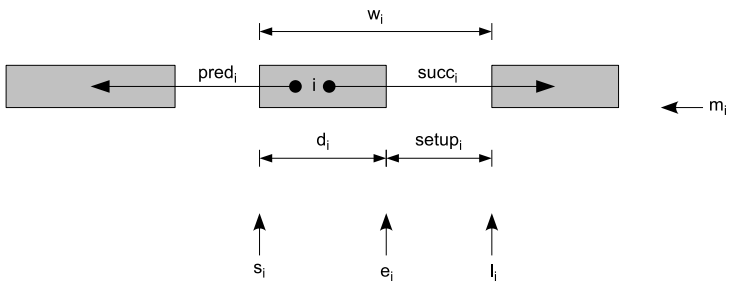

Figure 4: Task Structure

The first task on machine k is given by variable $msucc_k$, and the last task by $mpred_k$.

Machine dependent duration and machine preference are given extensionally by functions $D_i : M \mapsto \mathbb{N}$ and $P_i : M \mapsto \mathbb{N}$. The function $S_i : I \mapsto \mathbb{N}$ gives the setup time of task i as a function of the successor variable.

The release date rel_i is a constant which imposes a hard constraint, since it is imposed by the availability of raw materials. The due date value due_i does not create a hard constraint, but becomes part of the objective function. The variable del_i is used to indicate the delay of a task.

We first define the domains of the overall schedule ends:

$$maxm \in M \tag{10}$$
$$maxe \in 0..end \tag{11}$$

We then have a number of primitive constraints which link the variables of a task:

$$\forall_{i \in I} : [s_i, e_i, l_i] \in 0..end \tag{12}$$
$$\forall_{i \in I} : e_i = s_i + d_i \tag{13}$$
$$\forall_{i \in I} : l_i = s_i + w_i \tag{14}$$
$$\forall_{i \in I} : w_i = d_i + setup_i \tag{15}$$
$$\forall_{i \in I} : \text{element}(m_i, D_i, d_i) \tag{16}$$
$$\forall_{i \in I} : \text{element}(m_i, P_i, p_i) \tag{17}$$
$$\forall_{i \in I} : \text{element}(succ_i, S_i, setup_i) \tag{18}$$
$$\forall_{i \in I} : p_i \geq pref_{min} \tag{19}$$

Equations 19 are used to impose some global preference rule for all tasks. If the constant $pref_{min}$ is set to a high value, then tasks will be only considered on their preferred machines, if it is set low, then tasks can move to more machines. This gives a simple mechanism to switch from normal operations to "fire fighting" mode, when optimisation criteria are relaxed in order to deliver products on time.

The links between start, end, release and due dates are given in the next set of equations.

$$\forall_{i \in I} : s_i \geq rel_i \tag{20}$$
$$\forall_{i \in I} : del_i \geq e_i - due_i \tag{21}$$
$$\forall_{i \in I} : del_i \leq delay_{max} \tag{22}$$

Equations 22 allows to control the maximal delay allowed on any task. In particular, it allows to set total delay to zero, forcing a solution which delivers all products on time.

3.2.2 Global Constraints

We now introduce the main constraint which deals with the machine assignment, and which ensures that tasks are not overlapping on machines. This again uses the *diffn* [13] constraint already presented in section 2.2.3.

$$\text{diffn}(\{\langle s_i, m_i, w_i, 1\rangle | i \in I\}) \tag{23}$$

Here three of the four parameters for each 2-D object are variables, the height of each object is set to 1.

We add projections of the machine assignment on both time and machine axis in order to improve propagation and add two *cumulative* [2] constraints. Recall that the *cumulative* constraint states that

$$\text{cumulative}(\{\langle s_i, d_i, r_i\rangle | i \in I\}, End, Limit) \tag{24}$$

is equaivalent to

$$End = \max_{i \in I}(s_i + d_i) \tag{25}$$

and

$$\max_{t \in [\min_{i \in I}(s_i), \max_{i \in I}(s_i + d_i) - 1]} (\sum_{i \in I, s_i \leq t, t < s_i + d_i} r_i) = Limit \tag{26}$$

We now can formulate the two redundant cumulative constraints. The first one states a cumulative resource constraint over all tasks

$$\text{cumulative}(\{\langle s_i, w_i, 1\rangle | i \in I\}, maxm, maxe) \tag{27}$$

The second *cumulative* constraint expresses a bin packing condition which can help to push the overall project end, if many tasks can only be scheduled on one particular machine.

$$\text{cumulative}(\{\langle m_i, 1, w_i\rangle | i \in I\}, maxe, maxm) \tag{28}$$

This redundant use of global constraints in multiple roles is discussed in more detail in [3] and [60].

We now look at the link between the successor variables and the machine assignment, which is handled by a *cycle* constraint [13, 17]. The constraint works on a directed graph which is shown in figure 5. There are two types of nodes, the task nodes (shown as circles) and the machine nodes (shown as rectangles). We have an edge from task i to task j, if task i can be followed by task j in the schedule. We have an edge from a machine k to a task i, if task i can be the first task on machine k, and an edge from task i to machine k if task i can be the last task on machine k. A solution consists in finding m cycles in that graph, each containing one machine node. Figure 5 shows two selected cycles (drawn with bold lines) on top of all edges of the graph. The *cycle* constraint has additional parameters which affect the start times s_i and the machine assignment variables m_i, so that the actual call for the constraint looks like this:

$$\begin{aligned} \text{cycle}(m, [succ_1, \ldots, succ_n, msucc_1, \ldots, msucc_m], \\ [n+1, \ldots, n+m], \\ [m_1, \ldots, m_n], \\ [w_1, \ldots, w_n], \\ [s_1, \ldots, s_n]) \end{aligned} \tag{29}$$

The link between successor and predecessor variables is done via an *inverse* constraint [17, 12]. The *inverse* constraint inverse($[s_1, \ldots, s_n], [p_1, \ldots, p_n]$) holds iff

$$\forall_{1 \leq i,j \leq n} : s_i = j \Leftrightarrow p_j = i \tag{30}$$

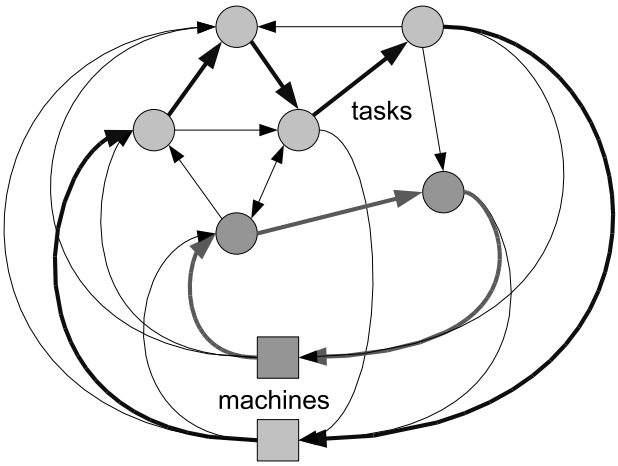

Figure 5: Graph Structure

We use it to connect the succ and msucc variables with the pred and mpred variables.

$$\begin{aligned}\text{inverse}([&succ_1, \ldots, succ_n, msucc_1, \ldots, msucc_m], \\ &[pred_1, \ldots, pred_n, mpred_1, \ldots, mpred_m])\end{aligned} \tag{31}$$

3.2.3 Objectives

In this problem, we do not just have a single objective function that needs to be optimised. Depending on the order structure, the work in progress and possible down times in equipment, the user may want to create schedules with very different objectives by selecting different options in the user interface. A first objective is to minimise the overall delay of tasks, which is a good objective in normal operations.

$$\min \sum_{i \in I} del_i \tag{32}$$

The results are similar but not identical, if we maximise the preferred assignment of tasks to machines. This will produce the highest quality and minimises scrap.

$$\max \sum_{i \in I} p_i \tag{33}$$

Energy costs are an important part of overall production costs. If we minimise the total production time, this cost is affected.

$$\min \sum_{i \in I} w_i \tag{34}$$

As energy costs vary during the day, this constraint can become much more complex, if we want to model this aspect accurately.

Another optimisation choice is to minimise overall production end, which avoids situations were many tasks are assigned to a preferred line, while other lines are empty.

$$\min maxe \tag{35}$$

For most of these objectives, we can also enable upper bounds so that in an interactive problem solving process we can derive a solution which is acceptable considering multiple objectives.

3.3 Additional Constraints

There are a number of additional constraints that we have ignored in the discussion so far.

3.3.1 Work in Progress

An important aspect of the schedule is the update from actual production data. Any task which has already started in production is no longer scheduled, but replaced with a "work in progress" activity. The timing of this activity can be manually changed, for example if a stoppage delays the completion of the task. The work in progress is modelled by dummy tasks which are added into the global constraints, and by re-evaluating the domain of the *msucc* variables.

3.3.2 Frozen Schedule

The scheduling system is not just used to create one master schedule, but is constantly re-run to update the schedule with the latest production data, so that the effects of delays on customer deliveries are immediately visible. When rescheduling we do not want the system to re-arrange the schedule completely. In the user interface we have the possibility to "freeze" the schedule of a line up to a given task. This fixes the order of the tasks, but not their start times, as these might be affected by the work in progress. This "freezing" can also be used to protect manual changes in the schedule.

3.3.3 Task Ordering

If we have multiple tasks producing the same product, then we can order them according to their due date. This reduces the amount of bin capacity we will need for this product and cuts some part of the search tree.

$$\forall_{i,j \in I} : r_i = r_j \wedge due_i < due_j \implies s_i < s_j \tag{36}$$

3.3.4 Throughput Learning

The accuracy of the schedule depends to a large extend on the accuracy of the throughput data which controls the duration of tasks and to some extend the machine assignment. The system contains a learning module which periodically checks actual production data in order to recalculate the throughput data for each product. The user can interact with this module by excluding certain measurements or by forcing a particular value.

3.4 Search

In a problem like this where there are many different types of variables with domains which even initially are very different in size it is not possible to rely on a standard labelling routine which selects the variables without an understanding of their purpose in the model.

We have developed a number of strategies which interleave the assignment of successor and setup variables, but which also use the start or end variables for task selection.

a This is a standard labelling routine, which was not able to find any solution for the sample problems below. It was therefore not included in the result tables.

b This strategy first assigns for each selected task the setup and then the successor variable. The selection is done by the most constrained successor variable. Doing this for all tasks fixes the machine assignment and the sequence of tasks on each machine. At the end it labels the start of all tasks to the earliest time point.

c Like **b**, but select the task with the smallest value in the successor variable.

d Like **b**, but select the task with the most constrained setup variable first.

e Like **b**, but select the task with the smallest value in the domain of the setup variable first.

f Select one task at a time in natural order, decide the machine assignment by setting the predecessor variable and then fixing the start time.

g Like **f**, but select the task with the smallest value in the domain of the start variable.

h Build schedule on one machine after the other, selecting tasks with smallest setup time first.

i Build the schedule by adding one task to each machine in a round-robin selection, always trying to find a task which minimises setup time.

j Select the task with the smallest value in the domain of the start variable, enumerate the setup variable and then the successor variable for this task, then fix the start variable to the smallest value in the domain.

k Like **j**, but select the task with the smallest value in the domain of the end variable first.

We have evaluated the strategies on some simplified test data sets which do not use due dates and assignment preference data. In table 1 we show the best cost obtained within 1 minute on a number of test data sets for our different strategies, using the objective function from equation 35. Entries with the best cost found are shown in bold. It is difficult to discern a clear favourite, seven

	Strategy									
Day	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**j**	**k**
11	**1251**	1334	**1251**	1638	1486	1546	1324	1282	1324	**1251**
12	**1473**	1629	1767	1783	1553	1807	**1473**	1488	**1473**	1516
13	1305	**1290**	**1290**	1464	1402	1492	1726	1726	1726	1535
16	1657	-	**1561**	1702	-	-	1896	1934	1896	**1561**
17	1433	1523	1509	1640	-	1741	**1430**	1509	**1430**	1450
18	**1850**	**1850**	**1850**	1940	-	-	**1850**	**1850**	**1850**	**1850**
19	1468	1352	1481	1523	1403	1505	1434	1371	1434	**1284**
20	1353	1559	1080	1150	1356	1047	1456	**960**	1456	1032

Table 1: Best solution: Overall End

of the strategies find the best solution for at least 2 data sets. Note that some strategies fail to find any solution within the given timeout.

Table 2 compares the results for the objective given by equation 34. The table again shows the best result obtained within 60 seconds. Here strategy **d** find the best solution in half the test cases, strategy **f** in two cases, and **g** and **k** each in one case.

For comparison, we also show the best results minimising total setup time in table 3. Here we have a clear winner, strategy **d** produces the best results for every data set, although strategies **h** and **j** also perform quite well.

3.5 Background

Scheduling has long been a main application area for constraint programming, starting with [26], and results compete very well against other solution methods

	Strategy									
Day	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**j**	**k**
11	3297	3326	3324	3342	**3260**	3291	3333	3328	3333	3324
12	4328	4295	**4252**	4311	4346	4292	4328	4328	4328	4325
13	3694	3712	3712	3719	**3686**	3731	3781	3781	3781	3751
16	4179	-	**4098**	4122	-	-	4217	4223	4217	4122
17	4118	4109	4135	4091	-	**4066**	4118	4135	4118	4116
18	3893	3855	3836	3836	-	-	3893	3817	3893	**3801**
19	3452	3374	**3322**	3360	3378	3361	3450	3432	3450	3413
20	2522	2552	**2401**	2453	2499	2449	2541	2425	2541	2441

Table 2: Best Solution: Total Production Time

	Strategy									
Day	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**j**	**k**
11	1005	1080	**970**	1410	1350	1455	**970**	985	**970**	**970**
12	925	**910**	**910**	1255	1090	1150	925	940	925	955
13	905	905	**900**	1285	1180	1180	**900**	**900**	**900**	915
16	1000	-	**955**	1495	-	-	970	**955**	970	970
17	950	985	**945**	1165	-	1225	**945**	**945**	**945**	960
18	**845**	**845**	**845**	1100	-	-	**845**	**845**	**845**	875
19	965	955	**945**	1240	1225	1285	**945**	960	**945**	975
20	**745**	**745**	**745**	910	850	880	**745**	760	**745**	760

Table 3: Best Solution: Total Setup Time

for very hard problems [2, 18, 19, 11, 8]. There is a large selection of techniques, some of which are packaged into global constraints of commercial suppliers [2, 13, 36, 4] or are described independently [15]. But there are relatively few industrial uses of constraints for scheduling described in the literature [64, 52, 63], although many more exist (some examples are listed in [63]).

The problem of scheduling with machine and sequence dependent setup times or costs that occurs as a sub-problem in the MOSES application is also described in [31, 70, 7].

4 GYMNASTE - Nurse Rostering

Personnel planning problems are another domain for constraint programming applications where global constraints can help to model complex problems. A wide range of problems has been covered, ranging from school or university timetabling, rostering for hospital or prison staff, to long-term personnel planning for Radio and TV stations. In this section we discuss nurse rostering, a planning problem for hospital wards. This has been an early application field for constraint programming [30, 74], and has led to a number of operational systems and practical case studies [34, 1, 29, 16]. Our presentation is based on the *GYMNASTE* system [21] and its model described in [54].

4.1 Problem

Hospitals are very large organisations, with hundreds or even thousands of medical personnel working in a large number of specialised departments and wards. Instead of solving the personnel assignment for the whole hospital in a single step, it is commonly broken into two separate components: Operational

(monthly) staff planning for nurses is typically done per ward, covering between 20 and 50 staff, while overall staff allocation levels are decided in a long-term, strategic planning process.

Wards usually operate on a shift basis, with a morning (**M**), afternoon (**A**) and a night (**N**) shift. For each day and each shift, there are staffing requirements which indicate how many people are required for this shift. These requirements can vary significantly during a month, as they are based on number of procedures, (projected) number of patients, administrative work and so on. We have to allocate enough nurses to each shift so that the demand is satisfied, while respecting the personnel work rules, individual contracts and satisfying personnel preferences. A basic rule states that a nurse can only work one shift on each day, and is either working or resting (**R**). This naturally leads to an assignment of nurses on each day to an activity, one of **M**, **A**, **N** or **R**. A simple example rule set is:

1. The demand of service required for each shift (**M**, **A**, **N**) must be met on every day. This can consist of a lower and an upper bound.
2. In a 4 week scheduling period, each person must have (4*2=8) rest days (**R**).
3. In any sequence of 6 days, there must be at least one rest day (**R**).
4. In any sequence of 5 days, there must be atmost 2 rest days (**R**).
5. In a 4 week scheduling period, a person is not allowed to work more than 8 night shifts (**N**).
6. Nobody is allowed to work more than 3 night shifts (**N**) in a row.
7. A day working night shift (**N**) may not be followed by a day working morning shift (**M**).
8. It is not possible to work night shifts (**N**), switch one day to a day shift (**M** or **A**) and then work night shifts (**N**) again.
9. There can be no "bridge" days, one day of work (**M**, **A**, **N**) in between two days off (**R**).

Many of the rules apply across multiple scheduling periods, it is for example not possible to assign 3 night shifts at the end of one scheduling period, and to start the next period with another two night shifts. These rules must be applied on a rolling horizon, while others, like rule (2), only apply from the beginning of a planning period to its end.

The exact form of the constraints will vary significantly between countries, hospitals or even wards within a hospital, and the rule set will undergo constant change as new regulations are imposed or new contracts negotiated with unions.

4.2 Model

It is possible to model the rules encountered in a nurse rostering problem with combinations of primitive constraints, in particular if combinators like reified constraints or cardinality constraints are available. A constraint model based on

such constraints will not perform much constraint propagation, but (combined with good heuristics) may well be sufficient to solve many problems were the constraints are not too tight. But a lot of of information can only be deduced by considering groups of constraints globally, as the example in figure 6 shows. We consider ten variables x_1 to x_{10}, each with domain 1..4. For each set of four consecutive variables, we have a constraint stating that two out of the four variables must have value 1. We also have a constraint stating that atmost four out of the ten variables may have value 1. From each constraint independently, we can not deduce any domain reduction. But considering the constraints together we find the reductions shown in the last line. The reasoning is a simple counting argument: variables x_1 to x_4 must contain two ones, variables x_5 to x_8 as well, using up all four available ones already. This means that x_9 and x_{10} can not be equal to one, therefore x_7 and x_8 must be equal to one, and so on. This shows that is can be worthwhile to express multiple constraints over a set

x_1	x_2	x_3	x_4							2 out of 4 have value 1
	x_2	x_3	x_4	x_5						2 out of 4 have value 1
		x_3	x_4	x_5	x_6					2 out of 4 have value 1
			x_4	x_5	x_6	x_7				2 out of 4 have value 1
				x_5	x_6	x_7	x_8			2 out of 4 have value 1
					x_6	x_7	x_8	x_9		2 out of 4 have value 1
						x_7	x_8	x_9	x_{10}	2 out of 4 have value 1
x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	atmost 4 have value 1
$\neq 1$	$\neq 1$	$=1$	$=1$	$\neq 1$	$\neq 1$	$=1$	$=1$	$\neq 1$	$\neq 1$	domain reduction

Figure 6: Example Constraint Interaction

of variables in one global constraint, as it gives us the opportunity to deduce more information.

4.2.1 Variables

We model the nurse rostering problem by defining a domain variable for each person and each day. This variable ranges over the four possible values **M**, **A**, **N**, and **R**. A typical problem size will be a schedule for 50 nurses over a four week horizon, so that we will need $50 * 28 = 1400$ variables. But in order to check for constraints with a rolling horizon, we may have to include the existing assignment of the previous time period as part of the problem, this requires another 1400, preassigned variables. We use the notation x_{ij} for the assignment of nurse i on day j. The set I of size n denotes the set of all nurses, the set J the set of m days in the planning horizon. We also use the set K to describe all shift types that need to be covered (**M**, **A** and **N**).

4.2.2 Constraints per Day

We next want to express the constraints for rule (1) satisfying the demands for each shift on each day. The constants l_{jk} and u_{jk} give the lower and upper bound for the demand on the shift type k on day j.

$$\forall_{j \in J, k \in K} : \text{among}(l_{jk}, u_{jk}, [x_{1j}, x_{2j}, \ldots, x_{nj}], k) \tag{37}$$

The *among* constraint states that the number of occurrences of the value k in the set of x variables must be between the lower l_{jk} and the upper u_{jk} bound.

4.2.3 Constraints per Person

We now look at the constraints which are expressed for each person. We use an extended *among* constraint to combine rules (2) and (3).

$$\forall_{i \in I} : \text{among}(1, 6, 6, 8, 8, [x_{i1}, x_{i2}, \ldots, x_{im}], \mathbf{R}) \tag{38}$$

The constraint states that at least 1 (argument 1) and atmost 6 (argument 2) variables in every sequence of 6 (argument 3) variables in the sequence of variables in argument 6 must have the value **R** (argument 7), and that in total there must be at least 8 (argument 4) and atmost 8 (argument 5) variables with value **R**. By combining rules (2) and (3) in this way, we are able to use counting arguments like those shown in figure 6.

We need another set of *among* constraints to express rule (4) that in any sequence of 5 days we can have atmost 2 days off, using rule (3) again to provide an overall limit.

$$\forall_{i \in I} : \text{among}(0, 2, 5, 8, 8, [x_{i1}, x_{i2}, \ldots, x_{im}], \mathbf{R}) \tag{39}$$

We can also use the *among* constraint to express that we can have no more than 3 consecutive days of night shifts (rule 6). We state that in any sequence of four days, we can not have four times the value **N** and rule (5) that we can assign atmost 8 night shifts in the overall period.

$$\forall_{i \in I} : \text{among}(0, 3, 4, 0, 8, [x_{i1}, x_{i2}, \ldots, x_{im}], N) \tag{40}$$

A more complex rule like rule (7) is modelled using the *sequence* constraint of CHIP. It states that in any sequence of 2 (argument 3) variables from the sequence in argument 4 we have at least 0 (argument 1) and atmost 0 (argument 2) (i.e. never) instances of the pattern given in argument 5. The pattern is described in a complex nested structure, saying that the first variable has value **N** and the second variable has value **M**.

$$\forall_{i \in I} : \text{sequence}(0, 0, 2, [x_{i1}, x_{i2}, \ldots, x_{im}], \\ [[[\text{sum}, 1, =, [\mathbf{N}]], [\text{sum}, 1, =, [\mathbf{M}]]]]) \tag{41}$$

This pattern language may seem exceedingly complex at first, but provides the flexibility to express very complex conditions easily. In the next constraint we combine rules (8) and (9) to state that in no sequence of three variables we find a pattern consisting either of the sequence $\mathbf{N}[\mathbf{M}, \mathbf{A}]\mathbf{N}$ or the sequence $\mathbf{R}[\mathbf{M}, \mathbf{A}, \mathbf{N}]\mathbf{R}$. The pattern language of the sequence constraint is largely controlled by which pattern can be handled effectively in the propagation algorithm.

$$\forall_{i \in I} : \text{sequence}(0, 0, 3, [x_{i1}, x_{i2}, \ldots, x_{im}], \\ [[[\text{sum}, 1, =, [\mathbf{N}]], [\text{sum}, 1, =, [\mathbf{M}, \mathbf{A}]], [\text{sum}, 1, =, [\mathbf{N}]]], \\ [[\text{sum}, 1, =, [\mathbf{R}]], [\text{sum}, 1, =, [\mathbf{M}, \mathbf{A}, \mathbf{N}]], [\text{sum}, 1, =, [\mathbf{R}]]]]) \tag{42}$$

It is possible to express the constraints with other combinations of global constraints. The basic counting of occurrences per day and person can be handled with the global cardinality constraint (*gcc*) [44, 42]. In [16], a *stretch*

constraint [33] and a *pattern* constraint [39] are used instead of the *among* and *sequence* constraints of CHIP. They both have somewhat different expressive power and use, but can express typical rules occurring in rostering.

4.2.4 Objective

There is no clearly defined objective that should be optimised in rostering problems. In most cases, users look for feasible solutions, or even infeasible solutions which minimise constraint violations. There are two balancing rules which may play the role of objectives: one is to balance any excess capacity in an even way over time and between shifts, the other is to treat people evenly, so that all persons obtain the same workload, expressed in shift (types) to be worked, weekend assignment, or assignments over public holidays.

4.3 Additional Constraints

So far we have described the core constraints encountered in the nurse rostering problem, but there are many more constraints that can be added.

4.3.1 Preferences

The rostering system produces a plan for a period of four weeks into the future and in principle may assign any nurse to any of the shifts in that planning horizon. Often, persons have strong preferences for working particular shifts or for not working at certain times. It is trivial to take a small number of preferences into account by forcing an assignment or removing values from the domain of a variable. The problem is that taking all preferences together will nearly always lead to an infeasible problem.

One approach to solve this issue is to use hierarchical constraint programming as presented in [34], where the solver tries to minimise the number of constraint violations.

A disadvantage of this approach is that the global constraints have to be extended to handle constraint violations. In a limited way this has been attempted in the *sequence* constraint of CHIP, with additional indicator variables to indicate where violations occur. These variables can then be used in new constraints for example to balance the violations evenly through the schedule.

Another approach is to treat preferences as heuristic choices which are tried in the search process. This is very easy to program, but it is quite difficult to balance preference treatment for all nurses in the roster.

A third approach is an interactive constraint solving process, where a user enters choices via a user interface and sees what effect this choice has. This was proposed in [74], but relies on very strong constraint propagation to be workable. As soon as an infeasible constraint is added, this must be detected, giving the user the choice how to restore a feasible constraint set.

4.3.2 Qualifications

A rather common extension is to consider different qualifications of nurses and demands on each type of qualification for each shift. If the qualifications partition the set of nurses, then the problem can be trivially decomposed and each subproblem solved independently. If qualifications overlap, then the model may

become much more complicated, depending on whether a single person can satisfy the demand for two different qualifications at the same time.

4.3.3 Holidays

Another factor which might make the rule set a lot more complicated is any scheduled absence from work, either holidays (non-work) or for example training courses (work). It is easy to block out a person from the schedule for a predetermined period of time by introducing another value type and preassigning that value. It is much more complex to understand how the work rules will be affected and which rules should be applied for the working days of that person in the planning period.

Unscheduled absence from work (for example illness) pose another type of problem and are related to overall day-to-day operations and recording of actual work. This is related to work time recording and pay issues, with any number of exceptions and special cases arising. It can be very challenging to use the actual (rather than the planned) roster of this month as the basis for next month's schedule.

4.3.4 Row/Column Interaction

We have seen in figure 6 how overlapping constraints for a set of variables can force some domain reduction which can not be detected by looking at each individual constraint alone. The *among* constraints (equations 38, 39, 40) take that into account by considering both overlapping periods and the complete time horizon. But there is another possible interaction, which is not detected by our current constraint set. The constraints per day and per person (equations 37 and 38) may interact as well. If we only consider the total number of occurrences in each row and column of the model, then this can be expressed with the *cardinality matrix* constraint of [45]. This reasoning can detect propagation which is not achieved with *gcc* constraints on the rows and columns alone, see [58] for an example.

There is even more additional reasoning that can be performed on each overlapping subset, although no working system has been proposed yet.

4.3.5 Balancing

Balancing the workload between persons, but also within the schedule for a single person can have a big impact on the perceived quality of a solution. Two different approaches to this issue have been proposed. In [20], the personnel planning problem is decomposed into a hierarchical view of yearly and monthly planning problems. To achieve a balanced solution in the long term, imbalances in the short term have to influence future decisions.

The alternative proposal is the *spread* constraint of [40], which allows to express constraints over the distribution of values in a schedule. This can be used inside one person's assignment to balance periods of activity or to balance the schedule of several persons.

4.4 User Specified Constraints

We have seen that the example rule set from section 4.1 could be expressed very concisely with global constraints, but we can't expect the end-user of the system to write *sequence* constraints. As the rule set is quite variable between installations and changing over time, we have to allow the user to express the constraints as part of the data of the problem, rather than modifying the program every time the constraint set changes. We either need a modelling language in which the constraints can be expressed by non-programmers (a very challenging problem), or a user interface where new constraint instances can be generated in a structured way. *GYMNASTE* uses the second approach: There is a syntax-driven constraint editor in the graphical user interface where the end-user can specify rules in a format similar to the example rule set above. We use a syntax driven format as this guarantees the creation of legal rules, which can then be compiled into the underlying constraint implementation. This syntax editor was first used in the *OPTISERVICE* system, a personnel planning environment for RFO, the French overseas radio and TV network.

4.5 Search

The tutorial [54] contains a detailed comparison of different simple search strategies for small nurse rostering problems. It compares variable orderings, value selection methods and partial search methods on three data sets of varying demand complexity. Each problem set contains 100 problems for 10 persons and 14 days, where the demands are generated randomly between 7 and 8 demands per day in data set 1, between 6 and 9 demands per day in data set 2 and between 5 and 10 demands per day in data set 3.

The variable selection can either static or dynamic: *line* is a static order per line, we assign the complete schedule for one person, before trying to assign the next person. *column* is a static order by day, we assign all shifts for one day before trying the next day. *MC* is a dynamic most-constrained selection, where the variable with the smallest domain is selected first, and *MCPS* is a most-constrained selection with an initial static sorting of the variables based on decreasing demand on the day.

The value selection is either *indomain*, testing the values in order **R**, **M**, **A**, **N**, or *randomised* which selects the values randomly. We also use a domain splitting (*split*) method, which first decides whether to assign work or rest for a day.

The search routine is either complete (*C*) or a partial, credit based search (*P*) [11, 54]. In table 4 we show the number of problems solved after 1000 and 5000 backtracking steps.

The *combined* entry shows how many problems are solved by at least one of the methods.

The results indicate that some methods clearly don't work at all, in particular assigning one line at a time. But by combining dynamic variable selection, randomised value selection and partial search it is possible to achieve respectable success rates with a very limited number of backtracking steps. By testing several assignment methods before giving up the success rate can be further improved.

A more involved strategy is described in [16].

Variable Selection	Value Selection	Search	1000 backtrack Set 1	1000 backtrack Set 2	1000 backtrack Set 3	5000 backtrack Set 1	5000 backtrack Set 2	5000 backtrack Set 3
line	indomain	C	0	0	0	0	0	0
line	random	C	0	0	0	1	2	0
column	indomain	C	9	16	11	9	18	13
column	random	C	44	38	34	41	38	39
MC	indomain	C	86	67	39	86	71	43
MC	random	C	57	46	26	45	39	42
MCPS	indomain	C	57	46	39	63	53	41
MCPS	random	C	60	52	32	63	50	34
line	indomain	P	0	0	0	0	0	0
line	random	P	4	2	1	20	12	5
column	indomain	P	13	13	12	21	18	13
column	random	P	93	87	71	99	96	86
MC	indomain	P	98	90	69	99	97	83
MC	random	P	97	94	70	99	97	85
MCPS	indomain	P	98	90	63	99	96	77
MCPS	random	P	98	94	69	99	99	85
MC	split	P	98	92	72	99	97	86
MCPS	split	P	99	92	68	99	96	80
combined						99	99	91

Table 4: Evaluation Results

4.6 Background

The nurse rostering problem is just one instance of a personnel planning problem that can be solved with constraint programming. The PhD thesis of P. Chan [20] gives an overview of different problems that can be modelled and compares CP with alternative approaches. Here we will only briefly mention some large scale personnel planning systems that have been developed using constraint programming.

EPPER The EPPER system, developed for the catering company SERVAIR by the software house GSI performs personnel planning and assignment for the French TGV train bar/restaurant personnel. It creates a four week planning, assigning agents of the correct qualification to different catering jobs on the trains. The assignments considers travel times, rest periods and other, hard constraints, and tries to balance the workload for all agents.

TAP-AI [9] is a reactive planning system for crew assignment for the airline SAS. It is intended for day-to-day management of operations.

OPTISERVICE is a package for personnel assignment for all overseas TV and radio stations of the French RFO network. It assigns teams of qualified journalists and technicians to events that need to be covered, taking working time regulations and pay rules into account.

MOSAR The MOSAR system was developed for the French ministry of justice by Cisi and COSYTEC and assigns prison guards for 200 prisons in France in a rolling shift plan very similar, but significantly larger, to the problem

discussed here. It looks at a one year planning horizon in order to create balanced schedules.

5 Conclusion

We have now seen three applications from three very different domains, each modelled with a combination of global constraints of different types. What are common points between them, and can we expect other applications to be similar?

Global constraints allow a very concise model of the applications to be built. We typically require only a few constraint types to model most aspects of the application. This simplicity of modelling is an important advantage when building large scale systems [3].

All three systems are decision support applications, with a human operator very firmly in control. The objective is not to build a push-button problem solver, but a tool where the user can interactively build a solution. This requires control over the constraint model, enabling and disabling constraints depending on the situation, and changing the objective function to find a solution which fits the current problem state. All three systems also allow to re-run the solver multiple times, keeping parts of a solution while rescheduling others.

In order to be able to re-use the program for multiple installations, it becomes important that constraints are expressed as part of the data model, and can be defined by the end-user, rather than being hard-coded into the application. This requires another abstraction level, which is familiar to the end-user and which has enough flexibility to handle different instances of the problem. In the *GYMNASTE* program we use a syntax driven rule editor, in the *MOSES* system a drawing tool where a schematic plan of the factory is drawn from which we infer constraints about product flows, storage capacities, etc.

Optimisation is not a very strong requirement, as typically there is no consensus about an objective quality measure. Different stakeholders in the problem have very different views on what makes one solution better than another, and the human operator often tries to find a compromise between conflicting objectives. This matches the experience with the ATLAS scheduling system [63], where the system itself is used to exchange possible plans between users. It must be noted that this lack of optimisation requirements may be due to the fact that all cases shown are *operational* (and not *strategic*) tools, dealing with day-to-day running of a service. Resource levels are typically decided at another level of the business process.

Visualisation plays a major role in developing the applications. This clearly concerns the way data and results are presented, but extends to the dynamic visualisation [59, 60, 62] of the search routine, which was initially built for the developers only. In the *MOSES* system the end-users demanded the inclusion of the visualisation tool in the final delivery, after seeing it during development. The *GYMNASTE* visual display shows the remaining domains after each interactive problem solving step, and end-users become very skilled in interpreting the results.

The end users may not be familiar with the constraint solving paradigm and the algorithms used, but they normally understand the application domain in much more detail and sometimes can point out problems and missing constraints

after just a glance at the display. They also often understand the impact of constraints on a solution, and are happy to control the model by enabling/disabling constraints or switching search strategies.

Although all models seem to contain a lot of structural symmetry, this does not hold for the full-scale application. In the *APACHE* system for example, although many stands on the terminal may look identical, they become very different as soon as the current parking situation is added to the constraint set. Rows and columns in the nurse rostering system may seem to belong to a matrix model with full symmetries, but last month's schedule imposes enough constraints to remove these symmetries. In the scheduling case, many tasks may look exchangeable, but it would require much work to find if all preference and safety rules concerning them really are the same.

And finally, the constraints form only part of the overall system (see [63] for an evaluation of the percentage). It is clearly important that the constraint problem can be modelled and solved effectively, but there are many other components that require much work in order to deliver an industrial application.

References

[1] S. Abdennadher and H. Schlenker. INTERDIP an interactive constraint based nurse scheduler. In *Practial Application of Constraint Technologies and Logic Programming (PACLP) 1999*, London, UK, 1999.

[2] A. Aggoun and N. Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. *Mathematical and Computer Modelling*, pages 57–73, 1993.

[3] A. Aggoun, N. Beldiceanu, E. Bourreau, and H. Simonis. L' apport des contraintes globales pour la modelisation et la resolution d'applications industrielles. In *FRANCORO II, Deuxieme Journees Francophones de Recherche Operationelle*, Sousse, Tunisie, April 1998.

[4] A. Aggoun, Y. Gloner, and H. Simonis. Global constraints for scheduling in CHIP. Invited Industrial Presentation, JFPLC 99, 1999.

[5] Gautam Appa, Dimitris Magos, and Ioannis Mourtos. LP relaxations of multiple all_different predicates. In Régin and Rueher [47], pages 364–369.

[6] K. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

[7] C. Artigues, S. Selmokhtar, and D. Feillet. A new exact solution algorithm for the job-shop problem with sequence-dependent setup times. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems CP-AI-OR 2004*, pages 37–49, Nice, France, April 2004.

[8] Philippe Baptiste and Claude Le Pape. Constraint propagation and decomposition techniques for highly disjunctive and highly cumulative project scheduling problems. In Smolka [65], pages 375–389.

[9] G. Baues, P. Kay, and P. Charlier. Constraint based resource allocation for airline crew management. In *ATTIS94*, Paris, France, April 1994.

[10] N. Beldiceanu, E. Bourreau, P. Chan, and D. Rivreau. Partial search strategy in CHIP. In *2nd Int. Conf. on Meta-Heuristics*, 1997.

[11] N. Beldiceanu, E. Bourreau, D. Rivreau, and H. Simonis. Solving resource-constrained project scheduling problems with CHIP. In *Fifth International Workshop on Project Management and Scheduling*, Poznan, Poland, April 1996.

[12] N. Beldiceanu, M. Carlsson, and J.X. Rampon. Global constraint catalog. Technical Report T2005:08, SICS, May 2005.

[13] N. Beldiceanu and E. Contjean. Introducing global constraints in CHIP. *Mathematical and Computer Modelling*, 12:97–123, 1994.

[14] Nicolas Beldiceanu and Mats Carlsson. Sweep as a generic pruning technique applied to the non-overlapping rectangles constraint. In Walsh [73], pages 377–391.

[15] Nicolas Beldiceanu and Mats Carlsson. A new multi-resource cumulatives constraint with negative heights. In Pascal Van Hentenryck, editor, *CP*, volume 2470 of *Lecture Notes in Computer Science*, pages 63–79. Springer, 2002.

[16] Stéphane Bourdais, Philippe Galinier, and Gilles Pesant. Hibiscus: A constraint programming application to staff scheduling in health care. In Rossi [50], pages 153–167.

[17] E. Bourreau. *Traitement de Contraintes sur les Graphes en Programmation par Contraintes*. PhD thesis, L'Universite de Paris 13 - Institut Galilee Laboratoire d'Informatique de Paris Nord (L.I.P.N.), March 1999.

[18] Yves Caseau and François Laburthe. Improved CLP scheduling with task intervals. In *International Conference on Logic Programming (ICLP)*, pages 369–383, 1994.

[19] Yves Caseau and François Laburthe. Cumulative scheduling with task intervals. In M. Maher, editor, *Joint International Conference and Symposium on Logic Programming (JICSLP)*, pages 363–377, 1996.

[20] P. Chan. *La planification du personnel: acteurs, actions et termes multiples pour une planification operationelle des personnes*. PhD thesis, Universite Joseph Fourier, Grenoble 1. TIMC-IMAG, October 2002.

[21] P. Chan, K. Heus, and G. Weil. Nurse scheduling with global constraints in CHIP: Gymnaste. In *Practical Applications of Constraint Technology (PACT) 1998*, London, UK, March 1998.

[22] P. Chan and G. Weil. Cyclical timetabling using constraint logic programming. In *Practial Application of Constraint Technologies and Logic Programming (PACLP) 2000*, Manchester, UK, 2000.

[23] Yannick Cras. Using constraint logic programming in services: A few short tales. In M. Bruynooghe, editor, *International Logic Programming Symposium (ILPS)*, pages 3–16, 1994.

[24] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Fifth Generation Computer Systems (FGCS)*, pages 693–702, 1988.

[25] M. Dincbas and H. Simonis. Apache - a constraint based, automated stand allocation system. In *Proc. of Advanced Software Technology in Air Transport (ASTAIR'91)*, pages 267–282, London, UK, October 1991. Royal Aeronautical Society.

[26] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving large combinatorial problems in logic programming. *Journal of Logic Programming*, 8(1):75–93, 1990.

[27] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving the car sequencing problem in constraint logic programming. In *European Conference on Artificial Intelligence (ECAI-88)*, Munich, W. Germany, August 1988.

[28] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, and T. Graf. Applications of CHIP to industrial and engineering problems. In *First International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, Tullahoma, Tennessee, USA, June 1988.

[29] A. Chun et al. Nurse rostering at the hospital authority of Hong-Kong. In *AAAI Innovative Applications of Artificial Intelligence*, 2000.

[30] S. Darmoni et al. Horoplan: computer-assisted nurse scheduling using constraint-based programming. In *Twelfth International Congress of the European Federation for Medical Informatics*, Lisboa, Portugal, 1994.

[31] F. Focacci and W. Nuijten. A constraint propagation algorithm for scheduling with sequence dependent setup times. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems CP-AI-OR 2000*, 2000.

[32] William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In *IJCAI (1)*, pages 607–615, 1995.

[33] Lars Hellsten, Gilles Pesant, and Peter van Beek. A domain consistency algorithm for the stretch constraint. In Wallace [72], pages 290–304.

[34] H. Meyer aufm Hofe. ConPlan/SIEDAplan: Personnel assignment as a problem of hierarchical constraint satisfaction. In *Practical Applications of Constraint Technology (PACT) 1997*, London, UK, 1997.

[35] Ugo Montanari and Francesca Rossi, editors. *Principles and Practice of Constraint Programming - CP'95, First International Conference, CP'95, Cassis, France, September 19-22, 1995, Proceedings*, volume 976 of *Lecture Notes in Computer Science*. Springer, 1995.

[36] Wim Nuijten and Claude Le Pape. Constraint-based job shop scheduling with Ilog Scheduler. *J. Heuristics*, 3(4):271–286, 1998.

[37] H. Simonis P. Kay. Building industrial CHIP applications from reusable software components. In *Practical Applications of Prolog (PAP) 1995*, Paris, April 1995.

[38] M. Perrett. Using constraint logic programming techniques in container port planning. *ICL Technical Journal*, pages 537–545, May 1991.

[39] Gilles Pesant. A regular language membership constraint for finite sequences of variables. In Wallace [72], pages 482–495.

[40] Gilles Pesant and Jean-Charles Régin. Spread: A balancing constraint based on statistics. In Peter van Beek, editor, *CP*, volume 3709 of *Lecture Notes in Computer Science*, pages 460–474. Springer, 2005.

[41] Jean-Francois Puget. Applications of constraint programming. In Montanari and Rossi [35], pages 647–650.

[42] Claude-Guy Quimper, Alejandro López-Ortiz, Peter van Beek, and Alexander Golynski. Improved algorithms for the global cardinality constraint. In Wallace [72], pages 542–556.

[43] J.-C. Regin. A filtering algorithm for constraints of difference in CSPs. In *AAAI*, pages 362–367, 1994.

[44] Jean-Charles Régin. Generalized arc consistency for global cardinality constraint. In *AAAI/IAAI, Vol. 1*, pages 209–215, 1996.

[45] Jean-Charles Régin and Carla P. Gomes. The cardinality matrix constraint. In Wallace [72], pages 572–587.

[46] Jean-Charles Régin and Jean-Francois Puget. A filtering algorithm for global sequencing constraints. In Smolka [65], pages 32–46.

[47] Jean-Charles Régin and Michel Rueher, editors. *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, First International Conference, CPAIOR 2004, Nice, France, April 20-22, 2004, Proceedings*, volume 3011 of *Lecture Notes in Computer Science*. Springer, 2004.

[48] Cristina Ribeiro and Maria Antónia Carravilla. A global constraint for nesting problems. In Régin and Rueher [47], pages 256–270.

[49] L. Ros, T. Creemers, E. Tourouta, and J. Riera. A global constraint model for integrated routeing and scheduling on a transmission network. In *7th International Conference on Information Networks, Systems and Technologies*, Minsk, October 2001.

[50] Francesca Rossi, editor. *Principles and Practice of Constraint Programming - CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 - October 3, 2003, Proceedings*, volume 2833 of *Lecture Notes in Computer Science*. Springer, 2003.

[51] H. Simonis. Application development with the CHIP system. In G. Kuper and M. Wallace, editors, *Constraint Databases and Applications. Proc Contessa Workshop*, number 1034 in LNCS, Friedrichshafen, Germany, September 1995. Springer Verlag.

[52] H. Simonis. Scheduling and planning with constraint logic programming. Tutorial at Practical Applications of Prolog (PAP) 1995. London, UK, April 1995.

[53] H. Simonis. A problem classification scheme - when to use CLP. Tutorial at Practical Applications of Constraint Technologies (PACT) 1996. Paris, France, April 1996.

[54] H. Simonis. Standard models for finite domain constraint solving. Tutorial at Practical Applications of Constraint Technologies (PACT) 1997. London, UK, April 1997.

[55] H. Simonis. Standard models 2 for finite domain constraint solving. Tutorial at Practical Applications of Constraint Technologies (PACT) 1998. London, UK, March 1998.

[56] H. Simonis. Visualization in constraint logic programming. Tutorial at Practial Application of Constraint Technologies and Logic Programming (PACLP) 1999. London, UK, April 1999.

[57] H. Simonis. Finite domain constraint programming methodology. Tutorial at Practial Application of Constraint Technologies and Logic Programming (PACLP) 2000. Manchester, UK, April 2000.

[58] H. Simonis. Sudoku as a constraint problem. In *Fourth International Workshop on Modelling and Reformulating Constraint Satisfaction Problems (CP2005)*, October 2005.

[59] H. Simonis and A. Aggoun. Search-tree visualization. In P. Deransart, J. Maluszynski, and M. Hermenegildo, editors, *Analysis and Visualisation Tools for Constraint Programming*, volume 1870 of *LNCS*. Springer Verlag, 2000.

[60] H. Simonis, A. Aggoun, N. Beldiceanu, and E. Bourreau. Global constraint visualization. In P. Deransart, J. Maluszynski, and M. Hermenegildo, editors, *Analysis and Visualisation Tools for Constraint Programming*, volume 1870 of *LNCS*. Springer Verlag, 2000.

[61] H. Simonis, P. Charlier, and P. Kay. Constraint handling in an integrated transportation problem. *IEEE Intelligent Systems and their applications*, 15(1):26–32, Jan/Feb 2000.

[62] H. Simonis, T. Cornelissens, V. Dumortier, G. Fabris, F. Nanni, and A. Tirabosco. Using constraint visualization tools. In P. Deransart, J. Maluszynski, and M. Hermenegildo, editors, *Analysis and Visualisation Tools for Constraint Programming*, volume 1870 of *LNCS*. Springer Verlag, 2000.

[63] Helmut Simonis. Building industrial applications with constraint programming. In Hubert Comon, Claude Marché, and Ralf Treinen, editors, *Constraints in Computational Logics (CCL)*, volume 2002 of *Lecture Notes in Computer Science*, pages 271–309. Springer, 1999.

[64] Helmut Simonis and Trijntje Cornelissens. Modelling producer/consumer constraints. In Montanari and Rossi [35], pages 449–462.

[65] Gert Smolka, editor. *Principles and Practice of Constraint Programming - CP97, Third International Conference, Linz, Austria, October 29 - November 1, 1997, Proceedings*, volume 1330 of *Lecture Notes in Computer Science*. Springer, 1997.

[66] R. Szymanek, F. Gruian, and K. Kuchcinski. Application of constraint programming to digital systems design. In *Workshop on Constraint Programming for Decision and Control*, Institute for Automation, Silesian University of Technology, Gliwice, Poland, June 1999.

[67] Touraïvane. Constraint programming and industrial applications. In Montanari and Rossi [35], pages 640–642.

[68] P. Van Hentenryck and J.P. Carillon. Generality versus specificity: An experience with AI and OR techniques. In *AAAI*, pages 660–664, 1988.

[69] W. van Hoeve. The alldifferent constraint: A survey. In *6th Annual Workshop of the ERCIM Working Group on Constraints*, Prague, Czech Republic, June 2001.

[70] P. Vilim. Batch processing with sequence dependent setup times. In P. Van Hentenryck, editor, *Principles and Practice of Constraint Programing - CP 2002*, Cornell University, Ithaca, N.Y., September 2002.

[71] Mark Wallace. Practical applications of constraint programming. *Constraints*, 1(1/2):139–168, 1996.

[72] Mark Wallace, editor. *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, volume 3258 of *Lecture Notes in Computer Science*. Springer, 2004.

[73] Toby Walsh, editor. *Principles and Practice of Constraint Programming - CP 2001, 7th International Conference, CP 2001, Paphos, Cyprus, November 26 - December 1, 2001, Proceedings*, volume 2239 of *Lecture Notes in Computer Science*. Springer, 2001.

[74] G. Weil, K. Heus, F. Puget, and M. Poujade. Solving the nurse scheduling problem using constraint programming. *IEEE Engineering in Medicine and Biology*, July-August 1995.