In Petrovic, S. & Vanden Berghe, G. (ed.), Annals of Operational Research, Special Issue on Personnel Scheduling and Planning, Vol.155, No.1, November 2007, 257-277 (Accepted for publication, January 2006)

Towards a Practical Engineering Tool for Rostering

Edward Tsang, John Ford, Patrick Mills, Richard Bradwell, Richard Williams, Paul Scott
Department of Computer Science, University of Essex, United Kingdom

Abstract

The profitability and morale of many organizations (such as factories, hospitals and airlines) are affected by their ability to schedule their personnel properly. Sophisticated and powerful constraint solvers such as ILOG, CHIP, ECLiPSe, etc. have been demonstrated to be extremely effective on scheduling. Unfortunately, they require non-trivial expertise to use. This paper describes ZDC-Rostering, a constraint-based tool for personnel scheduling that addresses the software crisis and fills a void in the space of solvers. ZDC-Rostering is easier to use than the above constraint-based solvers and more effective than Microsoft's Excel Solver. ZDC-Rostering is based on an open-source computer-aided constraint programming package called ZDC, which decouples problem formulation (or modelling) from solution generation in constraint satisfaction. ZDC is equipped with a set of constraint algorithms, including Extended Guided Local Search, whose efficiency and effectiveness have been demonstrated in a wide range of applications. Our experiments show that ZDC-Rostering is capable of solving realistic-sized and very tightly-constrained problems efficiently. ZDC-Rostering demonstrates the feasibility of applying constraint satisfaction techniques to solving rostering problems, without having to acquire deep knowledge in constraint technology.

1. Introduction

Personnel scheduling is a problem that affects the core operations of many organizations, such as factories, hospitals and airlines. The profitability of a factory or airline depends on its ability to generate efficient schedules to fully utilize its staff. Regulations, in the form of constraints, must be satisfied in hospital and airline crew scheduling.

A well known example is nurse rostering. This problem involves the assignment of personnel holding the correct professional qualifications to shifts, subject to satisfying regulatory and other types of hard and soft constraints. It is a particular category of rostering problem that has been studied for decades (see [Burke et al 2004] [Ernst et al 2004a,b] [Cheang et al 2003] for recent surveys). Such problems present difficulties for the constraints practitioner in two distinct ways: first, in devising an effective model of the problem and, second, in its efficient solution. These will be discussed below.

Typical nurse rostering problems can be formulated as Constraint Satisfaction Problems (CSP) or Constrained Optimization Problems (COP) [Tsang 1993; Freuder & Mackworth 1994; Dechter 2003]. Constraint satisfaction is a powerful technique which has been successfully applied to scheduling problems, e.g. see [Lever et al 1995; Rodosek & Wallace 1998; Hnich et al 2002; Bourdais et al 2003]. In general, these techniques use constraints in the problem to prune the search space (especially in complete search) or guide the search (especially in stochastic search).

Commercial products such as ILOG, Charme, Cosytec CHIP, ECLiPSe and many more have been demonstrated to be very effective in scheduling problems. One of the major obstacles that developers face is cost: these products are not cheap to purchase. Besides, developers without knowledge of constraint technology may find the learning curve rather steep. The number of expert users in these products is limited. Therefore, the cost of developing software could be very high. These all contribute to the software crisis in constraint programming: the power of the technology is limited by the complexity of using it.

Within the constraint satisfaction/optimization research community, a large amount of effort has been invested in engineering stronger algorithms, studying problem difficulty and (more recently) studying the implications of using different problem formulations. As a result, individual researchers, and the research community as a whole, have accumulated a large amount of implicit and explicit domain knowledge regarding how best to solve problems. However, it is difficult to transfer the technology to an application, such as nurse rostering, without requiring an expert in the field, when one bears in mind the knowledge required to apply constraint technology effectively. This effectively includes:

- Knowing how to formulate a given problem as a CSP/COP
- Knowing how to incorporate domain knowledge into the solver
- Knowing how to choose a good formulation for a given problem
- Knowing which solver to apply to a given problem
- Knowing how to engineer a solver.

Various industrial strength packages, such as OPL++ (which builds a layer on top of ILOG Solver), have been implemented with the explicit aim of making access to constraint technology easier (http://www.ilog.fr) [Michel & Van Hentenryck 2001]. Unfortunately, even these packages require a significant amount of expertise to use because they usually come in the form of a constraint library that can be linked to a standard application, written in the desired 3rd Generation Language. Knowledge of the target language and the constraint library is still required. Generally these packages concentrate only on the issue of solving. They are written to relieve the user of the burden of writing their own solvers, but deep knowledge in constraint solving is still required in order to use these packages effectively.

The Computer-aided Constraint Programming (CACP) project attempts to provide a system that encompasses the core process of applying constraint technology [60]. The ZDC system developed in the project is designed to bring end-users and solution providers together. The design philosophy is that problem solvers should not be required to gain deep knowledge of constraint technology before being able to use it, and algorithm designers should develop reusable software for helping end-users. The rostering system presented in this paper serves to demonstrate the usability of the ZDC system.

Within the research community, competition between algorithms drives researchers to produce ever-faster algorithms that yield better results, for a restricted set of problems. Generally this is accomplished by tailoring the algorithm using large amounts of domain knowledge. The goal of the CACP project was not to compete with these highly specialized algorithms. Rather, the goal was to provide a simple, easy-to-use system to enable researchers and users to produce solutions without having to invest much effort in learning the constraint language, constraint solving techniques or how to use the system. Thus, our research echoes the philosophy in hyper-heuristics, which is to produce "off the peg" systems as opposed to producing tailor-made systems [Burke et al 2003a, b] [Burke et al 2005] [Ross 2005]. The system implemented is primarily targeted for users who are not necessarily interested in implementing constraint programming techniques, but would nevertheless like to exploit constraint technology for their own benefit.

2. The ZDC-Rostering Architecture

ZDC-Rostering is built upon the ZDC constraint modelling system (version 1.83) [Bradwell et al 2000]. ZDC is an open-source system that supports the tasks of problem formulation and entry, in addition to supplying pre-written solvers and aiding the user in choosing which of the available algorithms to apply. Problems are modelled in the declarative language EaCL (standing for Easy abstract Constraint Language) [Mills et al 1998], which the user can enter via an intuitive user interface. The problem specification is decoupled from the solvers, so that users may experiment with different problem formulations easily. Careful attention has been directed to providing a user-friendly GUI-based system for easy entry of problem constraints. An extensive help system is also provided. Having formulated the problem in the EaCL language and entered it, the user can solve the problem by using one of the pre-written generic solvers. Choosing the correct solver for a problem is often a difficult task and therefore the CACP project makes an initial attempt to address this issue.

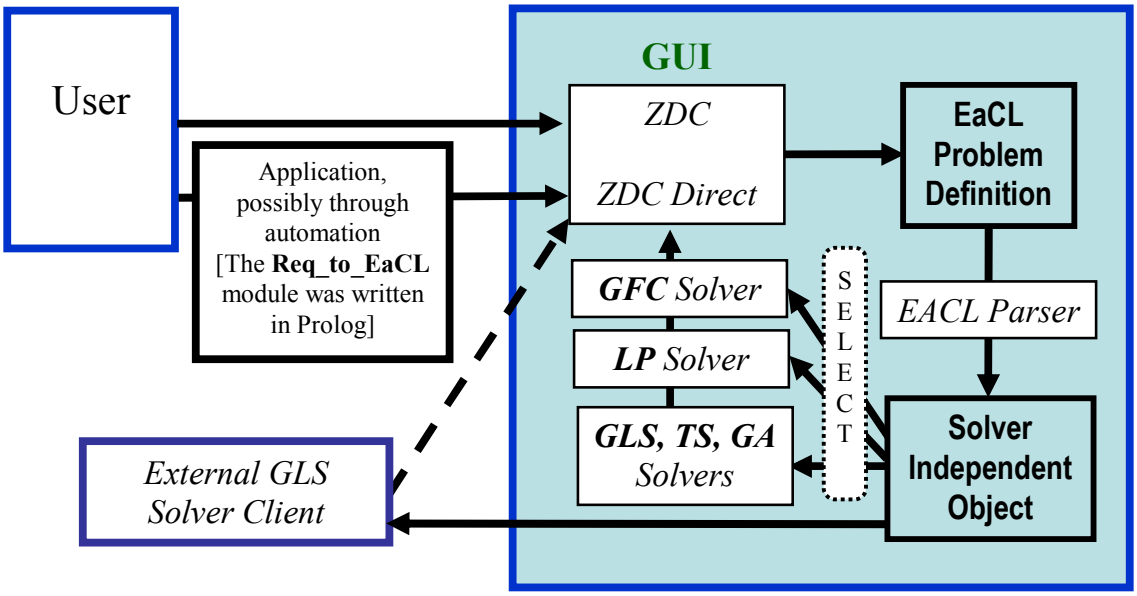

Figure 1: The ZDC Architecture, the foundation of ZDC-Rostering

The ZDC architecture is illustrated in Figure 1. The main flow of control starts with the entry of the problem definition in the EaCL language (developed within the group). Additional data, required for more demanding problems, can be read automatically from Excel by using it as an automation server. Problem description entry is performed using either the ZDC user interface or ZDC Direct. ZDC Direct, an interface for ZDC, allows the user to enter the problem formulation as text. It provides the user programs with a text-based interface to ZDC, and is used as such in ZDC-Rostering.

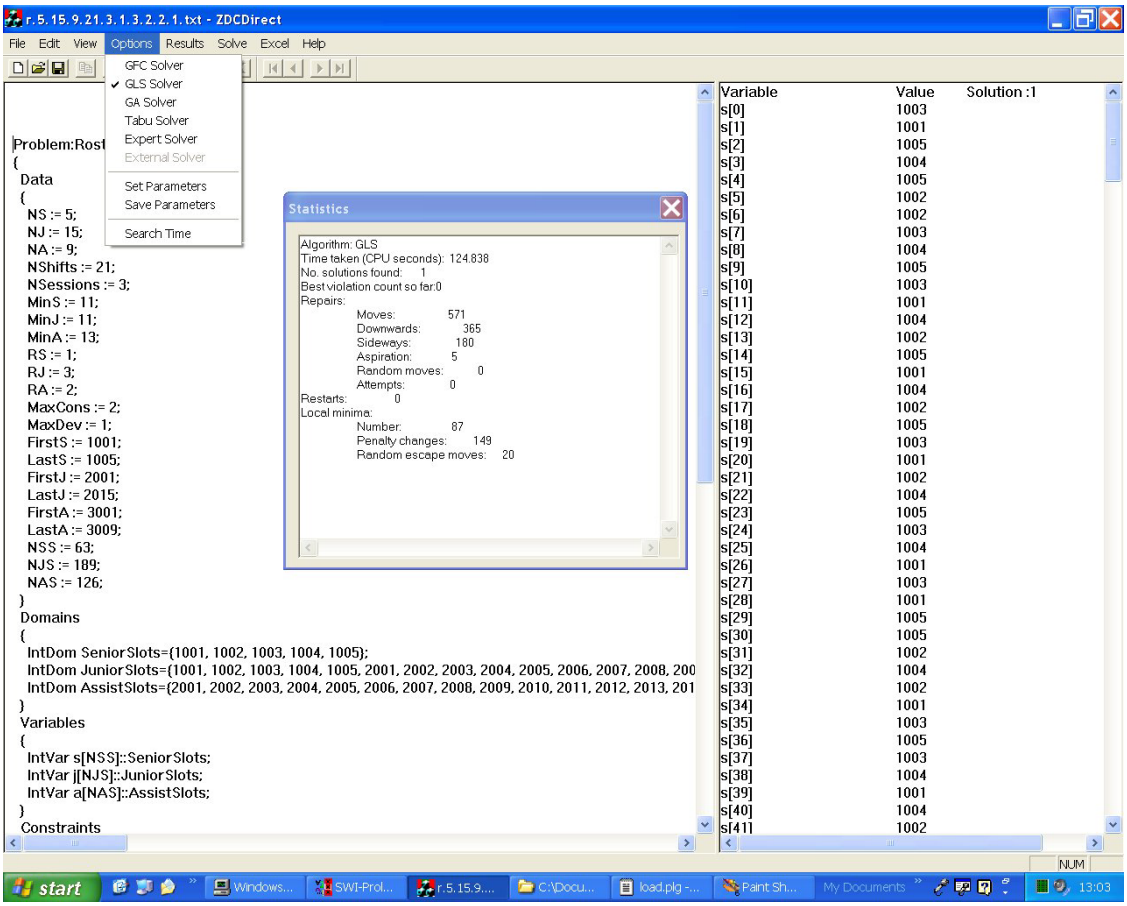

Figure 2: A snapshot of ZDC 1.8, showing the user interface with a problem loaded (left), a solution (right) and the statistics window (middle)

Figure 2 shows a screen shot of ZDC 1.8 "in action". A problem, written in EaCL (such as the one shown in Appendix B, which was generated from the Req_to_EaCL module of ZDC-Rostering), can be loaded into the ZDC system. ZDC can be used to find the first solution using the default solver (which can be user-defined), which is then shown in the right-hand frame, or to find some or (in case complete search is used) all of the solutions. A statistics window shows the statistics relevant to the algorithm selected. The user may modify the problem (e.g. by adding or relaxing constraints) and re-solve, if so desired. He/she can also choose the solver to use (under the Options menu – the simplex solver is not shown in the menu as it will be invoked automatically if the problem is detected to be a linear programming problem). Standard Windows conventions, such as on-line "Help", are adopted. Output can be displayed in Excel grids for easy export.

Once a problem description has been entered by the user, the EaCL parser parses the description. An invalid formulation is reported to the user via the user interface. A correctly parsed definition generates the raw, solver-independent constraint objects. These solver-independent objects are used by the selected solver as a guide to generate its own set of constraint objects. This architecture essentially separates the solver and its object library from the parser, which makes the architecture easily extensible. It is important to allow this separation because EaCL potentially supports a large number of solvers for different problem types. Some of these solvers specialize in solving a particular class of problem and therefore require a different set of constraint objects from solvers dealing with other problem types. Solvers supplied in ZDC include a generalized Forward Checking solver, a Linear Programming (LP) solver and local search solvers implementing Extended Guided Local Search (EGLS), Tabu Search (TS) and Genetic Algorithms (GA).

An algorithm selection expert module is responsible for matching the problem formulation to the available solver algorithms that can potentially be applied to that formulation. A solver, when invoked, runs in a separate thread, with only one thread executing at a time. The thread terminates once the solver has found a solution or the solver has been terminated prematurely. The results from the solver are passed back to the user interface, where they are relayed to the user. What we have described above is the normal usage of the system. ZDC can also perform the role of an automation server. This means that stand-alone applications with specific user interfaces can be written for a particular domain, yet as automation clients they can access some of the problem-solving capabilities of ZDC.

ZDC can also perform the role of a problem description server, using sockets. This means that ZDC can provide an external solver with a description of the variables, domains and constraints of the currently parsed problem. This facilitates a greater separation between the two phases of problem modelling and solving. The EaCL Parser generates a solver-independent description of the problem. The description instructs the external client solver how to generate a constraint object tree, using its own constraint object library.

The separation of modelling and solving in ZDC enables problem solvers to focus on modelling the problem (without worrying about how to solve it) and constraint algorithm designers to develop solvers (without having to worry about building user interfaces). In ZDC, it is very easy to add additional solvers to the system. To add a new solver (which may implement a new search algorithm) to ZDC, all the algorithm developer has to do is to build an interface to input solver-independent objects. External solver clients may register with the server so that the server becomes aware of their existence. An external solver can submit itself as a slave solver by sending an appropriate message to the server. As a slave, the external solver is under the direct control of the server. The user can interact with the ZDC interface and force an external solver to solve the current problem being modelled and then return the results, just as if it was an internal solver.

The application in ZDC-Rostering is written in Prolog. It generates (from the user's problem specification) a constraint model in EaCL syntax, which is then input to ZDC Direct for solving. By building on ZDC, we did not have to reproduce the solvers when we implemented ZDC-Rostering. Neither did we have to know which algorithm is most appropriate for solving the problem on hand. We were able to pay full attention to the implementation of an interpreter that converts a problem specification into the EaCL language.

3. Problem Modelling in ZDC-Rostering

To solve a rostering problem, ZDC-Rostering takes the following steps:

Step 1. Problem specification – this comes from domain knowledge;
Step 2. Constraint modelling, or problem formulation – this involves defining the variables, domains and constraints of the problem based on the problem specification;
Step 3. Expressing the problem in EaCL – this requires knowledge of the constraint language;
Step 4. Solving the problem – this is done by calling ZDC's library solvers.

These will be described in the following subsections.

3.1 Problem Specification

Specification of the problem comes directly from domain knowledge. In ZDC-Rostering, a problem is defined by the following facts:

- The number of senior staff, junior staff and assistants are fixed (NS, NJ and NA);
- The number of shifts (NShifts); for example, given that each week has 7 days, each of which has 3 shifts, one has to schedule 21 shifts in a week;
- The number of parallel sessions (NSessions) per shift; for example the team may be required to serve 3 wards;
- The number of senior staff, junior staff and assistants required for each session (RS, RJ and RA); senior staff may be used to fill junior slots, but not vice versa; junior staff may be used to fill assistant slots, but not vice versa;
- No one is allowed to work for more than a specified number (e.g. 2) of consecutive sessions (MaxCons);

- No one must work for fewer than a specified minimum number of sessions; this minimum is calculated as the average load minus a specified number MaxDev (e.g. 1).

An example of a problem specification is shown in Appendix A. The problem defined here is a generic one. More realistic constraints, such as the availability of individual staff, compatibility between leaders, varying number of sessions per shift, etc., can be specified with minimal impact on the system.

3.2 Constraint modelling, or problem formulation

Modelling is recognised as the frontier of constraint research [Freuder 99]. This is the step which required most of ZDC-Rostering's development time.

To define a constraint satisfaction problem in ZDC, one needs to define the variables (Z), the domains (D) and the constraints (C) (hence the name of the software). To model the rostering problem defined above, three arrays of variables are defined:

- s[0 .. NShifts*NSessions*RS – 1] for senior slots
- j[0 .. NShifts*NSessions*RJ – 1] for junior slots
- a[0 .. NShifts*NSessions*RA – 1] for assistant slots.
- The domains are defined as follows:
- The domain of s[i] for all i is the set of senior staff;
- The domain of j[i] for all i is the union set of senior and junior staff (because senior staff are allowed to serve junior sessions);
- The domain of a[i] for all i is the union set of junior staff and assistants (because junior staff are allowed to serve assistant sessions).

To formulate the constraint that no-one can work on two jobs at the same time, the global constraint AllDifferent has been used. It could equally be expressed as follows:

```
Forall t in [0 .. NShifts - 1]
{
 Forall ( i in [t*NSessions*RS .. (t+1)* NSessions*RS-1],
           j in [t* NSessions*RS .. (t+1)* NSessions*RS-1], i < j )
 {
            s[i] <> s[j];
 }
  … <similar constraints for junior and assistant slots> …
}
```

To formulate the constraint that no one can work for more than *k* consecutive sessions:

```
Forall t in [0 .. NShifts-k-1]
{
 Forall ( sf in [1 .. NS] )
  {
   Count( [ s[t* NSessions*RS], ..., s[(t+k+1)* NSessions*RS-1],
             j[t*NSessions*RJ], ..., j[(t+k+1)*NSessions*RJ-1]], [sf]) <= k;
  }
          … <similar constraints for junior and assistant slots> …
}
```

An appropriate index was needed to enumerate all the posts of all the sessions at the same time. Here t counts the time from 0 to NShifts-1, the number of shifts in the problem. Then it is a matter of counting the number of posts that take the same staff, and making sure that the sum does not exceed k. The above example deals with senior staff only. Here they are assumed to be numbered between 1 and NS, the number of senior staff available.

To formulate the constraint that no senior staff should work for k sessions fewer than a specified workload:

```
Forall ( sf in [1 .. NS] )
{
        Count( s, [sf] ) + Count( j, [sf] ) >= MinS;
}
```

Similar constraints for junior staff and assistants can be defined.

3.3 Coding in EaCL

The Easy abstract Constraint Programming Language (EaCL) is the language used to formulate problems prior to solving. The formulation above is close to, but not exactly in, the syntax of EaCL. A valid problem formulation in EaCL is split into data, domains, variables, constraints and optimization sub-sections. EaCL supports variables with integer, Boolean, real and sets as domains. The EaCL grammar supports a wide range of logical, integer, set and symbolic constraints. There are also various facilities supporting lists and sets as well as conditional branching. A complete description of the EaCL language is available on-line [Mills et al 1998]. Details of the rostering problem, as expressed in EaCL, will not be shown here. Interested readers may find an example in Appendix B.

The Req_to_EaCL module in ZDC-Rostering is responsible for translating the requirement, based on the problem specification, to EaCL. Once the specification is complete, implementing the Req_to_EaCL module is relatively straightforward. ZDC was designed for usability. The simplicity and expressive power of EaCL made Req_to_EaCL relatively easy to implement. For example, the quantification and nested quantification supported by EaCL helped to reduce the length of the problem file. Req_to_EaCL was written in roughly 200 lines of Prolog code (excluding comments), which is quite easy to re-implement.

Note that Req_to_EaCL is for demonstration purposes. Real problems typically have their specifications stored in a database, so Req_to_EaCL would have to be modified to read from a database (which should be relatively easy for Prolog). In fact, for a real application, one does not have to re-implement Req_to_EaCL. All one needs to do is to implement an interface which translates the specification from the database to EaCL. Being a small and simple language, EaCL should be much easier to learn than most other constraint languages. The decoupling of problem formulation and problem solving is designed to increase program reusability.

3.4 Solving the problem

Once the problem has been expressed in EaCL, it can be loaded into ZDC for solving. Solvers will be selected automatically. A simplex algorithm will be used to solve linear problems which contain variables with real domains and linear constraints only. A Generalized Forward Checking (GFC) algorithm has also been implemented, with a corresponding library of constraint objects, to represent the category of complete search algorithms. The forward checking algorithm, when applied to an optimization problem, maintains the best solution cost and uses it as a bound on the current solution cost. GFC has also been extended so that it can be applied to problems involving *n*-ary constraints, where *n* is greater than 2. Built into the GFC algorithm is a thrashing-detection mechanism that detects if the solver is an inefficient method for solving the current problem [Borrett et al 1996]. Upon detection of thrashing the GFC is automatically terminated.

Three local search techniques have been implemented, all sharing the same library of constraint objects. The solvers implemented are a Genetic Algorithm, Tabu Search (TS) and Extended Guided Local Search (EGLS). EGLS was found to be the most efficient solver for the rostering problem that we have implemented so far. The version of TS that was implemented shares most of its modules with GLS. It should be emphasized that only a simple TS, incorporating a taboo list with user-definable length, has been implemented in ZDC.

Guided Local Search (GLS) is a meta-heuristic, first invented by Voudouris [Voudouris & Tsang 2003]. When the hill climber is caught in a local minimum, GLS provides a means of escaping. Essentially, it escapes from a local minimum by adding extra penalty terms to the cost function. When the algorithm detects that it is in a local minimum, it chooses a feature of the current solution to penalize. A term is then added to the cost function to increase the cost of any solution containing the penalized feature. Penalizing

the feature results in an increase in the cost of the local minimum. Neighbouring solutions that do not exhibit the penalized features then become more desirable, and hill climbing re-commences.

The version of GLS implemented in ZDC is the Extended GLS (EGLS) developed by Mills, which incorporates aspiration moves and random moves [Mills 2002; Mills et al 2003]. EGLS searches on the augmented cost function, which is equal to the original cost function plus the penalty terms. An aspiration move is a move to a new solution which has a lower cost, in terms of the original cost function, than the best solution found so far. ZDC implements two types of random moves in EGLS. The first is a random move with a certain (normally low) probability. It was found to be useful in problems where the weight of the penalty is set too low. The second type of random move is invoked only at a local minimum: instead of using the standard moves after penalties are applied, random moves are used, with a (normally) low probability, to escape the local optimum.

Users without knowledge of constraint programming may choose to rely on the "Expert Solver" provided in the ZDC system, which will initially assign a random solver to the given problem, but learns (over time) the success rate of different solvers to the user. Based on the heuristic that the user is likely to solve similar problems over time, the Expert Solver selects solvers probabilistically according to their past success rate.

4. Problem-solving in ZDC-Rostering

4.1. Problems tackled and the size of their search space

In this paper, we use artificial problems to demonstrate the capability of ZDC-Rostering. These problems are generated by a problem generator, which is publicly available [60].

A problem is characterized by a vector:

((NS, NJ, NA), NShifts, NSessions, (RS, RJ, RA), MaxCons, MaxDev)

where NS, NJ and NA are, respectively, the number of senior staff, junior staff and assistants available. NShifts is the number of shifts to schedule and NSessions the number of sessions per shift. RS, RJ and RA senior staff, junior staff and assistants, respectively, are required per session. No one should work for more than MaxCons consecutive sessions. No one should work fewer than the average workload (in terms of sessions) minus MaxDev sessions.

We first asked ZDC-Rostering to produce weekly schedules, where there are three shifts per day (so NShifts = 7 x 3 = 21). We assume that there are three sessions per shift (Nsessions = 3); each session requires one senior staff, three junior staff and two assistants. The number of staff available is adjusted to vary the tightness of the problem. In other words, we have tackled a family of problems with the following settings:

((NS, NJ, NA), 21, 3, (1, 3, 2), 2, 1).

The size of the search space in a constraint satisfaction problem is $\prod_x |D_x|$, where D_x is the domain of variable x and $|D_x|$ is its size. For example, when there are 5 senior staff, 16 junior staff and 12 assistants in the above problem, the size of the search space is calculated as follows:

- Number of senior slots to fill: NSS = 21 (shifts) x 3 (sessions per shift) x 1 (staff per session) = 63
- Number of junior slots to fill: NJS = 21 (shifts) x 3 (sessions per shift) x 3 (staff per session) = 189
- Number of assistant slots to fill: NAS = 21 (shifts) x 3 (sessions per shift) x 2 (staff per session) = 126
- Domain size for each senior slot i: $D_{s[i]} = 5$
- Domain size for each junior slot i: $D_{j[i]} = 5 + 16 = 21$ (as senior staff can serve junior slots)
- Domain size for each senior slot i: $D_{a[i]} = 16 + 12 = 28$ (as junior staff can server assistant slots)

Therefore, the size of the search space is 5^{63} x 21^{189} x 28^{126}, which is roughly 10^{476}, and thus yields a problem of considerable size.

Next, we tested the scalability of ZDC-Rostering. We asked ZDC-Rostering to produce schedules for 2 and 3 weeks, which means NShifts was set to 42 and 63 respectively. A little reflection should convince the readers that the size of the search space grows exponentially as NShifts grows.

4.2. Solver Selection

Experiments were conducted on a PC with an Athlon 2500+ processor and 1GB of RAM, running under Windows XP.

The problem ((5, 16, 12), 21, 3, (1, 3, 2), 2, 1) was fed to the Generalized Forward Checking Solver (GFC, which conducts a complete search) with minimum domain size heuristic for variable ordering. No solutions were found in over 16 hours. Neither could the Tabu Search Solver and the Genetic Algorithm Solver manage to find solutions in 16 hours. Only the Extended Guided Local Search (EGLS) Solver was able to solve this problem, which it did within 1 minute. Therefore, in the rest of the experiments reported here, EGLS was used for all problems.

It is important to emphasize again here that the versions of Tabu Search and Genetic Algorithm implemented were fairly basic. For example, aspiration has not been made available in the Tabu Search. Therefore, the results reported here should not be regarded as a beauty-contest between the above-mentioned stochastic search algorithms.

The user specifies the time allowable to solve a problem. ZDC-Rostering will report failure when this time has elapsed. The user is given the option to increase the execution time. If the user chooses to use the algorithm selection expert in ZDC-Rostering, it will select any of the above-mentioned algorithms at random and switch to another algorithm should it fail to solve the problem within the time specified. ZDC-Rostering will detect that not all the constraints are linear inequalities, so that the Linear Programming Solver will not be selected. For the problem ((5, 16, 12), 21, 3, (1, 3, 2), 2, 1), had the time specified been under one hour, ZDC-Rostering will spend from 1 minute (had it chosen EGLS at the beginning) to just over 3 hours (had all the other three algorithms been picked before EGLS) before it settles down to the EGLS Solver.

In ZDC-Rostering, one can configure EGLS to incorporate different options, such as switching aspiration and random moves on or off [Mills 2002]. Once the configuration is fixed, there is only one major parameter to set in EGLS, which determines the weight of penalties. In our experiments, the default configuration (which is to include aspiration and random moves) and default penalty weight (which is 1) were used. The values of the parameter could affect the efficiency of the algorithms in certain applications. Experiments showed that the joint effect of aspiration and random moves reduces the sensitivity of the results to the value of the penalty weight. Details of this result can be found in [Mills 2002] and [Mills et al 2003].

5. Experimental Results

We have tested ZDC-Rostering extensively on problems in varying classes. The size of a constraint satisfaction problem is one of the factors that affect the difficulty of the problem, but it is not as important as the tightness of the problem [Cheeseman et al 1991]. We therefore concentrated on ZDC-Rostering's performance on problems of varying tightness. To help in focussing our analysis, we report our results on problems of the class ((5, 16, A), 21, 3, (1, 3, 2), 2, 1), with decreasing A (i.e. increasing tightness), starting with A=12. What we found was consistent with the experience of other constraint researchers regarding problem tightness.

For loose problems, the time taken by ZDC-Rostering increased gradually when A decreases. Problems with the number of assistants (A) between 12 and 9 took roughly one minute, with a standard deviation of below 7 seconds. However, when A was reduced to 8, the time taken by ZDC-Rostering ranged from 195 seconds to over one hour (in which case it was terminated). This is due to the stochastic nature of EGLS (as mentioned above, EGLS was the only solver that was able to solve these problems within 16 hours). When A was reduced to 7, no solution was found after one hour. So it is reasonable to assume that ((5, 16, 8), 21, 3, (1, 3, 2), 2, 1) is close to phase transition [Hogg & Williams 1994; Smith & Grant 1995]. As reported in the literature, for problems of similar size, computation cost increases sharply as one approaches phase transition. Figure 3 shows the minimum time taken by ZDC-Rostering to solve the problems under discussion.

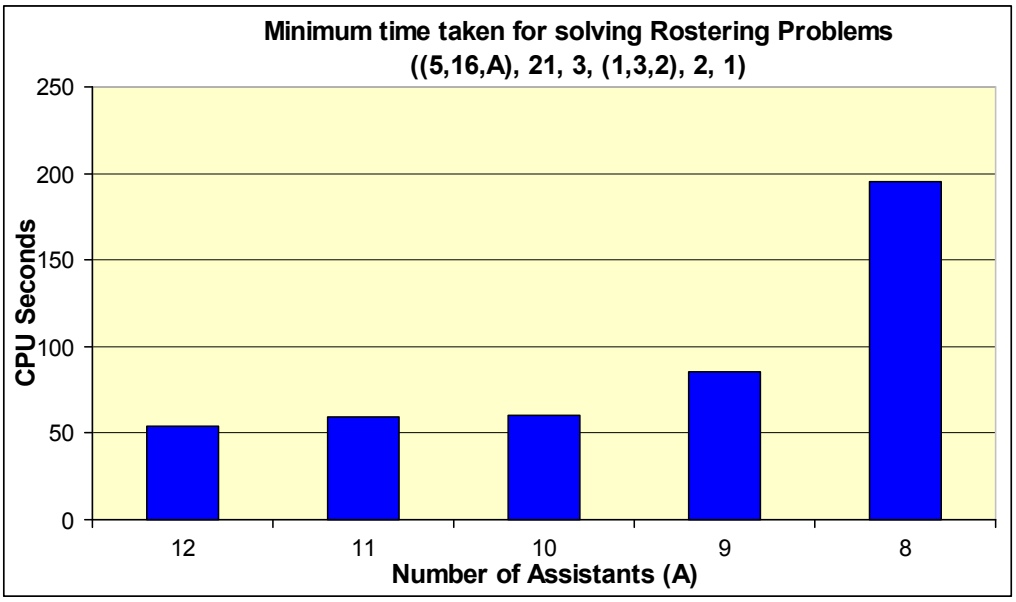

Figure 3: Minimum time taken for solving rostering problems of increasing tightness

Next, we investigate the impact of increasing the problem size. When one increases the number of shifts in the problem, the number of variables grows. Consequently, the size of the neighbourhood grows. The major impact on EGLS is that it takes more time to explore the neighbourhood in each step. Like other local search algorithms, EGLS examines $\sum_x |D_x|$ neighbours in each move. Therefore, as the number of variables grows, we expect EGLS to take more time per move.

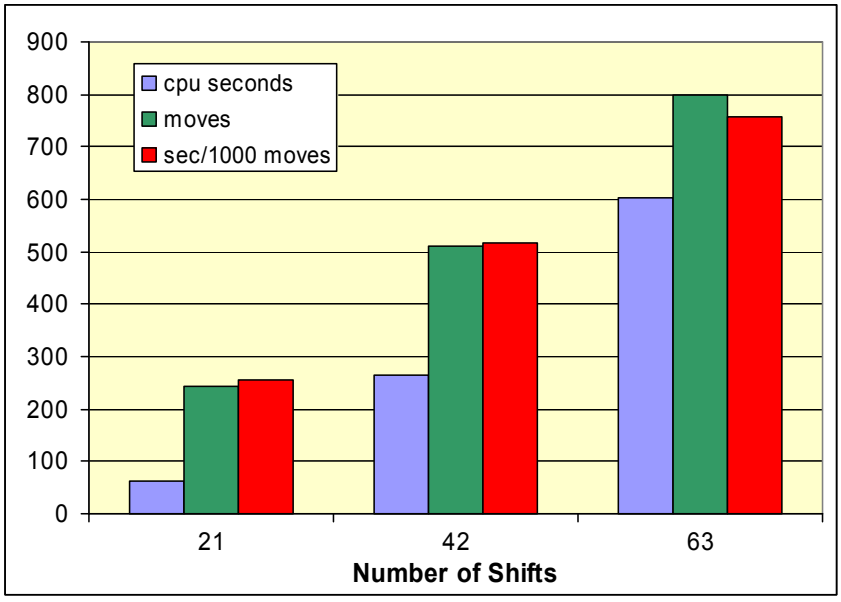

Figure 4: Increase in search time per move in EGLS as the number of shifts increases

Figure 4 shows the results of three runs with the number of shifts varied. In one of the runs in solving ((5, 16, 12), 21, 3, (1, 3, 2), 2, 1), EGLS took 62 seconds and 243 moves to solve the problem, which meant it took 0.25 seconds per move. When the number of shifts was doubled, EGLS took 264 seconds and 512 moves to solve the problem, which meant it took 0.52 seconds per move. When the number shifts was trebled, EGLS took 604 seconds and 799 moves to solve the problem, which meant it took 0.76 seconds per move. As expected, the number of moves required to find solutions increases as the number of shifts increases. The time required per move increased roughly linearly with the number of shifts.

Note that the size of the search space grows exponentially as the number of shifts grows (as discussed in Section 4.1). Results show that, by using EGLS, the time and number of moves taken by ZDC-Rostering only grew linearly. This suggests that ZDC-Rostering is capable of scaling up to solve large problems.

6. Discussion

6.1. Relating constraint solvers

We have no intention to claim that Extended Guided Local Search is the fastest heuristic search algorithm, despite its success in ZDC-Rostering. Neither do we consider ZDC-Rostering to be more complete than other constraint solvers. Readers interested in comparison of constraint solvers may refer to, for example, [Fernandez & Hill 2000] and [Wallace et al 2004]. Typically, each solver has its strengths and weaknesses. Comparison between solvers is often difficult and sometimes meaningless [Hooker 1995]. What the community should promote is effort to assess the strength and weakness of each solver, and position them within the space of solvers. In this section, we attempt to do this for ZDC-Rostering.

No general-purpose problem-solving system is perfect. In designing a constraint system, one has to consider, among other things:

- Expressive power of the language;
- Usability of the system (by which we mean how easy it is to learn and use it); and
- Efficiency.

Our focus is on the usability and cost-effectiveness of the system. Commercial systems such as ILOG Solver, CHIP and ECLiPSe are equipped with very efficient algorithms. They are empowered by rich languages and libraries, which implement specialized algorithms or heuristics for specialized constraints. ZDC is designed to be easy-to-learn and easy-to-use. ZDC-Rostering is a flexible system which allows the users to change the specification of the problem easily without needing a substantial amount of programming.

OPL++ [Michel & Van Hentenryck 2001] is built to make ILOG Solver easier to use, and therefore, shares a common research agenda with ZDC. However, to allow constraint experts to fully exploit the power of advanced constraint solving techniques, OPL++ needs to be a rich language (which is based on C++). The EaCL language is designed to be a small language compared with OPL++. With a smaller language, expressiveness of the language is compromised. This is a price that we pay for simplicity. ZDC allows software engineers with limited knowledge to gain access to constraint technology. Therefore, it occupies a different position in constraint programming.

The research agenda in the CHIC [Chamard et al 1995] and CHIC-2 [Gervet, 2001] projects is also related to ours. The main aim of these projects is to develop methodologies to make constraint technology more accessible to researchers.

To complete the spectrum, one should not forget that Microsoft Excel can be used to model constraint satisfaction problems. Problems can be solved by Excel's "Solver", which is an add-on. Not being restricted to constraint satisfaction, one should not expect Solver to solve constraint satisfaction problems as efficiently as state-of-the-art constraint systems. However, it does not require much expertise in constraint technology to use, and therefore should not be ignored as a useful tool for simple constraint satisfaction.

Figure 5 shows the relative position of ZDC compared to other systems. The relative positions are indicative only and not meant to be precise. State-of-the-art constraint solvers such as ILOG Solver are powerful but (apart from being expensive) are not easy to use, especially by users who have limited knowledge in

constraint satisfaction. Microsoft's Excel Solver is easy to use, but limited in problem solving ability. There is a huge gap between these two sets of solvers. ZDC is designed to fill this gap. There is certainly room for more solvers to be developed to fill the Pareto front of solvers. This will especially be true if we consider more dimensions in the space of solvers, such as product cost and whether a product is open-source or not.

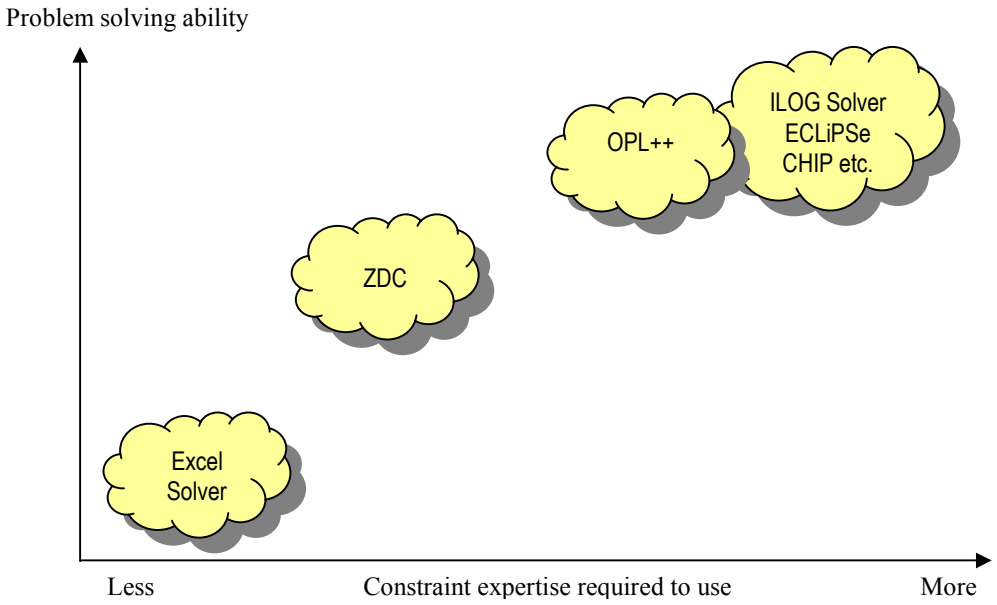

Figure 5 – Relative positions of solvers in constraint satisfaction problem solving (the positions are indicative and not meant to be precise, hence the clouded shapes); there is room for more solvers to fill the Pareto front.

6.2. Providing constraint-solving power to non-constraint experts

In this section, we examine where exactly in the problem solving process ZDC-Rostering can help users to exploit the power of constraint technology – the main objective of this research.

In order to exploit constraint technology, one must first be able to formulate the problem on hand (such as nurse rostering) as a constraint satisfaction problem. Some constraint models are easier to solve than others. Therefore, problem formulation is arguably the most important step in solving constraint satisfaction problems. Modelling is considered to be the final frontier of constraint satisfaction [Freuder 1999]. Tools for comparing models (e.g. [Borrett & Tsang 2001]) or guidelines on modelling (e.g. [Gervet 2001]) are still in their infancy. By providing a simple constraint language, ZDC aims to help non-experts to build constraint models [Bradwell et al 2000]. With a simple language and a modelling interface, ZDC helps users to experiment with different models more easily.

Once a constraint model is formulated, ZDC-Rostering enables users to exploit constraint technology without (a) having to gain deep knowledge of the techniques, (b) needing to be able to pick the right algorithms for their problems, and (c) having to implement the constraint solving algorithms selected. The availability of tools like this often determines whether constraint technology can be used or not. OPL++ is a comparable tool. The differences between ZDC and OPL++ are: (i) the former uses a smaller language (EaCL), while the latter uses a much more sophisticated language (ILOG Solver), and (ii) the former is open source, while the latter is a commercial product and thus committed to using specific solvers marketed by the company.

6.3. Limitations and future research

ZDC-Rostering addresses the core issues in constraint problem solving, including problem formulation, algorithm selection and solution generation. It is a usable package but by no means a finished product. One of the major developments needed to deploy ZDC-Rostering in work places is to interface it with other programs, such as databases and graphic packages.

From a scientific point of view, the constraint language EaCL is simple, by design. Therefore, expressiveness is compromised. In many commercial packages, there are precise constraints to capture specific requirements in scheduling. Commercial software packages often have specialized procedures for handling different types of constraints. This is where their problem-solving power comes from. However, in order to exploit such power, users have to have good knowledge of the solvers, which contributes to the software crisis, as explained in Section 1.

We could have implemented a larger set of complete constraint methods. For example, we could have implemented MAC [Sabin & Freuder 1994] [Bessière & Régin 1996] or symmetry breaking methods (e.g. see [Gent & Smith 2000; Petrie & Smith 2003]). As explained in Section 2, ZDC is implemented as an open system which allows external solvers to be implemented and plugged in.

ZDC's algorithm selection mechanism is currently primitive. It can detect linear programming problems. However, it does not implement any complex mechanism to map problems to algorithms, or methods to switch from one algorithm to another (e.g. see [Borrett et al 1996]). Much more could be done in this area. Case-based reasoning has been demonstrated to be promising in timetabling (e.g. see [Burke et al 2003, 2006]); it could be useful for algorithm selection. Machine learning methods could also be used to map problems to algorithms [Kwan 1997].

Finally, debugging is an important aspect of programming that has not been addressed in CACP. Interested readers should consult the results of major projects such as [DiSCiPl 1999].

ZDC-Rostering is a prototype for demonstrating the feasibility of applying constraint satisfaction techniques to solve rostering problems, without having to acquire deep knowledge in constraint technology. There is full potential to extend it into an operational system.

Open-source is an effective way to promote constraint technology and help end-users to exploit the technology. ZDC is designed is to allow new interfaces or new solvers to be added to the system. For example, a more elaborate graphic user interface could be added for viewing the variables, domains, constraints and solutions. Alternatively, one could add a logic-based interface to ZDC to provide logicians with a more familiar language for describing their problems (e.g. see [Abbas & Tsang 2004]). Some recent developments that one could consider adding to ZDC include newer genetic algorithms [Aickelin and Dowsland, 2003], variable neighbourhood search [Burke et al 2003], tabu search [Burke et al 1999], fuzzy constraints [Meyer auf'm Hofe 2001a and 2001b], and hybrid methods [Li et al 2003; Burke et al 2001].

Constraint solving researchers may not be interested in writing interfaces for their solvers. By making sure that their solvers can handle the constraints in the language EaCL, they can make their solvers available to a wider audience by plugging their solvers into ZDC. Thus, ZDC can be used as a hub to bring constraint developers and end-users closer together.

7. Concluding Summary

Much research in constraint satisfaction focuses on producing algorithms. In recent years, more and more researchers have recognised the software crisis in constraint programming: the rapid increase in constraint-solving power has not been matched by developments that enable users to exploit the techniques. The software crisis has led to many research issues in software engineering, including modelling, usability and software reusability. These issues motivated the research reported in the CACP (Computer-Aided Constraint Programming) Project.

In the CACP project, a small constraint language EaCL is defined. Compared to commercial constraint software such as OPL++, expressive power is limited in EaCL. EaCL is deliberately kept small, but general features such as universal quantifiers are made available. Using commercial conventions, a user-interface is built for easy assess to constraint satisfaction problem formulation. A number of solvers have been implemented in ZDC. The inclusion of EGLS makes ZDC an efficient general-purpose solver. By balancing between usability and efficiency, ZDC occupies a unique place in the space of constraint solvers.

By using ZDC, a new rostering system (ZDC-Rostering) has been developed. By building on top of ZDC, ZDC-Rostering separates problem formulation from problem solving. It comprises a Req_to_EaCL module for problem formulation and ZDC for problem solving. Req_to_EaCL translates rostering requirements into constraint satisfaction problems, expressed in the EaCL grammar. After loading the problem into ZDC, the user may attempt to solve the problem using the solvers provided. We have demonstrated in this paper that the ZDC architecture allows system developers to focus on problem specification (formally defining the requirements), problem formulation (defining the variables, domains and constraints) and problem solving (finding solutions for CSPs) independently.

Equipped with Extended Guided Local Search (EGLS), ZDC-Rostering is capable of solving (possibly tightly constrained) problems of realistic size within minutes. Thanks to the modularity of ZDC, the constraints considered so far have been implemented easily in the Req_to_EaCL module. New constraints, such as the availability of individual staff, can be added easily, either through Req_to_EaCL, or directly in ZDC. This means that rescheduling in ZDC-Rostering is relatively easy. To summarize, being efficient, open-source, easy to use, implement and maintain, ZDC-Rostering has full potential to solve realistic rostering problems.

Acknowledgements

We wish to thank both Nathan Barnes and James Borrett for their contribution to the CACP project. This project was funded by the EPSRC grant GR/L20122 and a University of Essex Research Promotion Fund grant. The anonymous reviews have helped to improve the quality of this paper significantly.

References

[1] Abbas, A. & Tsang, E.P.K., Software Engineering aspects of constraint-based timetabling – a case study, Information & Software Technology Journal, Vol.46, 2004, 359-372

[2] Aickelin, U. & Dowsland, K.A., An indirect algorithm for a nurse scheduling problem, Computers and Operational Research, 31(5), 2003, 761-778

[3] Bartak, R., On-line guide to constraint programming, http://ktiml.mff.cuni.cz/~bartak/constraints/index.html

[4] Bessière, C. & Régin, J.C., MAC and combined heuristics: two reasons to forsake FC (and CBJ?) on Hard Problems, Proc., 2nd International Conference on Principle and Practice of Constraint Programming (CP'96), Cambridge, MA, USA, LNCS 1118, Springer-Verlag, August 1996, 61-75

[5] Borrett, J.E., Tsang, E.P.K. & Walsh, N.R., Adaptive constraint satisfaction: the quickest first principle, Proceedings, 12th European Conference on AI, Budapest, Hungary, 1996, 160-164

[6] Borrett, J.E. & Tsang, E.P.K., A context for constraint satisfaction problem formulation selection, Constraints, Vol.6, No.4, 2001, 299-327

[7] Bourdais, S., Galinier, P. & Pesant, G., HIBISCUS: a constraint programming application to staff scheduling in health care, in Rossi, F. (ed.), Proceedings, 9th Principles and Practice of Constraint Programming (CP 2003), 2003, 153-167

[8] Bradwell, R., Ford, J., Mills, P., Tsang, E.P.K. & Williams, R., An overview of the CACP project: modelling and solving constraint satisfaction/optimisation problems with minimal expert intervention, Workshop on Analysis and Visualization of Constraint Programs and Solvers, Constraint Programming, Singapore, 2000

[9] Burke, E.K., Causmaecker, P. De, & Vanden Berghe, G., A hybrid tabu search algorithm for the nurse rostering problem, B. McKay et al (Eds.): Simulated Evolution and Learning 1998, LNCS 1585, Springer Verlag, 1999, 187-194

[10] Burke, E.K., Causmaecker, P. De, Cowling, P., & Vanden Berghe, G., A memetic approach to the nurse rostering problem, Applied Intelligence special issue on Simulated Evolution and Learning, Springer Verlag, Vol 15, 2001, 199-214

[11] Burke, E.K., Causmaecker, P. De, Petrovic, S., & Vanden Berghe, G., Variable neighbourhood search for nurse rostering problems, M.C.G. Resende & J. Pinho de Sousa (Eds.), Metaheuristics: Computer Decision-making, Chapter 7, Kluwer, 2003, 153-172

[12] Burke, E.K., Kendall, G., Newall, J., Hart, E., Ross P. & Schulenburg, S., Hyper-heuristics: an emerging direction in modern search technology, Chapter 16, in Handbook of Meta-Heuristics, in Glover, F. & Kochenberger, G. (eds.), Kluwer, 2003, 457–474

[13] Burke, E.K., Kendall. G., Soubeiga. E., A tabu-Search hyperheuristic for timetabling and Rostering. Journal of Heuristics Vol 9, 2003, 451-470

[14] Burke, E., MacCarthy, B., Petrovic S. & Qu, R., Knowledge discovery in a hyper-heuristic for course timetabling using case-based reasoning, in E.K.Burke and P.Decausmaecker (edited), Practice and Theory of Automated Timetabling IV, Springer Lecture Notes in Computer Science Vol 2740, Springer 2003, 276-287

[15] Burke, E.K., De Causmaecker, P., Vanden Berghe, G. and Van Landeghem,H. (2004), The State of the Art of Nurse Rostering, The Journal of Scheduling, Volume 7 issue 6, November/December 2004, 441-499

[16] Burke, E.K., McCollum, B., Meisels, A., Petrovic, S. and Qu, R., A graph-based hyper heuristic for timetabling problems, accepted for publication in the European Journal of Operational Research, to appear 2005

[17] Burke, E., Petrovic, S. & Qu, R., Case based heuristic selection for timetabling problems, Journal of Scheduling, Vol 9 issue 2, 2006 (to appear)

[18] Chamard A., Fischler A., Guinaudeau, D-B. & Guillard A., CHIC Lessons on CLP Methodology, ECRC report, 1995

[19] Cheang, B., Li, H., Lim, A. and Rodrigues, B., Nurse rostering problems – a bibliographic survey, European Journal of Operational Research, Vol.151, No.3, 2003, 447-460

[20] Cheeseman, P., Kanefsky, B. & Taylor, W.M., Where the really hard problems are, Proc., 12th International Joint Conference on AI, 1991, 331-337

[21] Cosytec CHIP, http://www.cosytec.com/, accessed March 2005

[22] Dechter, R., Constraint processing, Morgan Kaufmann, 2003

[23] DiSCiPl debugging systems for constraint programming, http://discipl.inria.fr/, 1999

[24] Ernst, A., Jiang, H., Krishnamoorthy, M. and Sier, D., Staff scheduling and rostering : a review of applications, methods and models, European Journal of Operational Research, Volume 153, Issue 1, 2004, pp 3-27

[25] Ernst, A., Jiang, H., Krishnamoorthy, M., Owens, B. and Sier, D., Annotated bibliography of personnel scheduling and rostering, Annals of Operations Research, Vol.127, 2004, 21-144

[26] ECLiPSe, http://www.icparc.ic.ac.uk/eclipse/, accessed March 2005

[27] Fernandez, A. & Hill, P.M., A comparative study of eight constraint programming languages over the Boolean and finite domains, Constraints, Kluwer academic publishers, Vol.5(3), 2000, 279-305

[28] Freuder, E.C. & Mackworth, A., (ed.), Constraint-based reasoning, MIT Press, 1994

[29] Freuder, E.C., Modeling: the final frontier, The First International Conference on The Practical Application of Constraint Technologies and Logic Programming (PACLP), London, April 1999, 15-21

[30] Gent, I.P. & Smith, B., Symmetry breaking in constraint programming, Proceedings, European Conference on Artificial Intelligence, Berlin, Germany, August 2000, 599-603

[31] Gervet C., Large Scale Combinatorial Optimization: A Methodological Viewpoint, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol.57, 2001, 151-175

[32] Glover, F., Tabu search Part I, Operations Research Society of America (ORSA) Journal on Computing 1, 1989, 109-206

[33] Glover, F., TABU search and adaptive memory programming -- advances, applications and challenges, in Barr, Helgason & Kennington, (ed.), Interfaces in Computer Science and Operations Research, Kluwer Academic Publishers, 1996

[34] Goldberg, D.E., Genetic algorithms in search, optimization, and machine learning, Reading, MA, Addison-Wesley Pub. Co., Inc., 1989

[35] Haralick, R.M. & Elliott, G.L., Increasing tree search efficiency for constraint satisfaction problems, Artificial Intelligence, Vol.14, 1980, 263-313

[36] Hnich, B., Kiziltan, Z. & Walsh, T., Modelling a balanced academic curriculum problem, Proceedings, Fourth International Workshop on Integration of AI and OR Techniques in Constraint programming for combinatorial optimisation Problems, CP-AI-OR'02, 2002, 121-131

[37] Holland, J.H., Adaptation in natural and artificial systems, University of Michigan press, Ann Arbor, MI, 1975

[38] Hogg, T. & Williams, C.P., The hardest constraint problems: a double phase transition, Research Note, Artificial Intelligence, Vol.69, 1994, 359-377

[39] Hooker, J.N., Testing heuristics: we have it all wrong, Journal of Heuristics, Vol.1, No.1, 1995, 33-42

[40] ILOG, http://www.ilog.com, accessed March 2005

[41] Kwan, A., A framework for mapping constraint satisfaction problems to solution methods, PhD Thesis, Department of Computer Science, University of Essex, Colchester, UK, May 1997

[42] Lever, J., Wallace, M. & Richards, B., Constraint logic programming for scheduling and planning, British Telecom Technology Journal, Vol.13, No.1., Martlesham Heath, Ipswich, UK, 1995, 73-80

[43] Li, H., Lim, A. & Rodrigues, B., A hybrid AI approach for nurse rostering problem, Proceedings of the 2003 ACM Symposium on Applied Computing, ACM Press, 2003, 730-735

[44] Meyer auf'm Hofe, H., Solving rostering tasks as constraint optimization, E. Burke & W. Erben, Practice and Theory of Automated Timetabling 2000, LNCS 2079, Springer Verlag, 2001, 191-212

[45] Meyer auf'm Hofe, H., Nurse rostering as constraint satisfaction with fuzzy constraints and inferred control strategies, E.C. Freuder and R.J. Wallace (Eds.), Constraint Programming and Large Scale Optimisation Problems, DIMACS Series, Vol 57, AMS, 2001, 67-99.

[46] Michel, L. & Van Hentenryck, P., OPL++: a modeling layer for constraint programming libraries, Proceedings, Third International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR-01), Wye, UK, 8-10 April 2001, 205-219

[47] Mills, P., Tsang, E.P.K., Williams, R., Ford, J. & Borrett, J., EaCL 1.0: an easy abstract constraint programming language, Technical Report CSM-321, University of Essex, Colchester, UK, December, 1998

[48] Mills, P., Extensions to guided local search, PhD Thesis, Department of Computer Science, University of Essex, Colchester, UK, 2002

[49] Mills, P., Tsang, E.P.K. & J.Ford, J., Applying an extended guided local search to the quadratic assignment problem, Annals of Operations Research, Kluwer Academic Publishers, Vol.118, 2003, 121-135

[50] Petrie, K.E. & Smith, B.M., Symmetry Breaking in Graceful Graphs, in Francesca Rossi (ed), Proc., Principles and Practice of Constraint Programming (CP 2003), Springer, LNCS 2833, 2003, 930-934

[51] Rodosek, R. & Wallace, M., A generic model and hybrid algorithm for hoist scheduling problems, in Maher, M. & Puget, J-F. (ed.), Proceedings, 4th International Conference on Principles and Practice of Constraint Programming -- CP98, Pisa, Italy, October 1998, Springer Verlag, Lecture Notes in Computer Science, 1520, 385-399

[52] Ross, P., Hyper-heuristics, in Burke, E.K. & Kendall, G. (ed.), Search methodologies: introductory tutorials in optimization and decision support techniques, Springer 2005, Chapter 17, 2005, 529-526

[53] Sabin, D. & Freuder, E.C., Contradicting conventional wisdom in constraint satisfaction, Proc., 11th European Conference on Artificial Intelligence, 1994, 125-129

[54] Smith, B.M. & Grant, S.A., Where the exceptionally hard problems are, Proceedings, Workshop on Studying and Solving Really Hard Problems, First International Conference on Principles and Practice of Constraint Programming, September, 1995, 172-182

[55] Tsang, E.P.K., Foundations of constraint satisfaction, Academic Press, London and San Diego, 1993

[56] Tsang, E.P.K., Mills, P., Williams, R., Ford, J. & Borrett, J., A computer aided constraint programming system, The First International Conference on The Practical Application of Constraint Technologies and Logic Programming (PACLP), London, April 1999, 81-93

[57] Voudouris, C. & Tsang, E.P.K., Guided local search, Chapter 7, in Glover, F. (ed.), Handbook of metaheuristics, Kluwer, 2003, 185-218

[58] Wallace, M., Schimpf, J., Shen, K. & Harvey, W., On benchmarking constraint logic programming platforms. response to Fernandez and Hill's "A comparative study of eight constraint programming languages over the Boolean and finite domains", Constraints, Kluwer academic publishers, Vol.9(1), 2004, 5-34

[59] Wallace, R.J. & Freuder, E.C., Supporting dispatchability in schedules that involve consumable resources, Journal of Scheduling, accepted for publication

[60] The Computer-aided Constraint Programming (CACP) Project (including links to downloads) http://cswww.essex.ac.uk/Research/CSP/cacp/, funded by EPSRC, March 1997 to August 2000, led by Edward Tsang, John Ford and Paul Scott.

Appendix A – Specification of a Sample Rostering Problem

The specification below shows the simplicity of ZDC-Rostering's user interface. When new constraints are needed, the program must be modified. The simplicity and expressiveness of EaCL [Mills et al 1998] makes it relatively easy to translate the specification to EaCL, as it is illustrated in the program in http://cswww.essex.ac.uk/CSP/cacp/cacpdemo.html.

```
/*
    The following clauses specify the senior staff, junior staff and assistants available (5, 16 and 8 in the following
    example):
*/
number_of_senior_staff( 5 ).
number_of_junior_staff( 16 ).
number_of_assistants( 8 ).


/*
    The following clauses specify the rostering demand:
        1. the number of time slots to be filled TS (=21 in this example);
        2. the number of sessions per time slot, SS (=3 in this example);
        3. the staff requirement, senior RS, junior, RJ and assistants RA (RS=1, RJ=3, RA=2 in this example).
*/
number_of_shifts( 21 ).
number_of_sessions( 3 ).
staff_requirements_per_slot( [senior(1), junior(3), assistant(2)] ).


/* The following clauses define the constraints */

/* Constraint 1.
    No one can work on two jobs at the same time; this applies to all rostering problems.
*/


/* Constraint 2.
    No staff can work for more than k consecutive sessions (k=2 in this example)
*/
max_consecutive_sessions( 2 ).


/* Constraint 3.
    Each staff member must work for a minimum number of sessions which is no more than m sessions less than the
    average load (m=1 in this example)
*/
max_deviation_from_avg_load( 1 ).
```

Appendix B – A Sample Rostering Problem in EaCL

This program was generated by the **Req_to_EaCL** module of **ZDC-Rostering** system. It has been edited for improved readability. The EaCL systax and the source code of Req_to_EaCL can be found in http://cswww.essex.ac.uk/CSP/cacp/cacpdemo.html.

```
Problem:Rostering
{
 Data
 {
  NS := 4;
  NJ := 6;
  NA := 5;
  NShifts := 6;
  NSessions := 2;
  MinS := 2;
  MinJ := 3;
  MinA := 3;
  RS := 1;
  RJ := 2;
  RA := 2;
  MaxCons := 3;
  MaxDev := 1;
  FirstS := 1001;
  LastS := 1004;
  FirstJ := 2001;
  LastJ := 2006;
  FirstA := 3001;
  LastA := 3005;
  NSS := 12;
  NJS := 24;
  NAS := 24;
 }
 Domains
 {
  IntDom SeniorSlots={1001, 1002, 1003, 1004};
  IntDom JuniorSlots={1001, 1002, 1003, 1004, 2001, 2002, 2003, 2004, 2005, 2006};
  IntDom AssistSlots={2001, 2002, 2003, 2004, 2005, 2006, 3001, 3002, 3003, 3004, 3005};
 }
 Variables
 {
  IntVar s[NSS]::SeniorSlots;
  IntVar j[NJS]::JuniorSlots;
  IntVar a[NAS]::AssistSlots;
 }
 Constraints
 {
  Forall (t in [0 .. NShifts - 1] )
  {
   AllDifferent([s[t*NSessions*RS+1], s[t*NSessions*RS], j[t*NSessions*RJ+3], j[t*NSessions*RJ+2],
       j[t*NSessions*RJ+1], j[t*NSessions*RJ], a[t*NSessions*RA+3], a[t*NSessions*RA+2], a[t*NSessions*RA+1],
       a[t*NSessions*RA]]);
  }
  Forall (t in [0 .. NShifts - MaxCons - 1] )
  {
```

```
    Forall (sf in [FirstS .. LastS] )
    {
      Count([s[t*NSessions*RS+7], s[t*NSessions*RS+6], s[t*NSessions*RS+5], s[t*NSessions*RS+4],
          s[t*NSessions*RS+3], s[t*NSessions*RS+2], s[t*NSessions*RS+1], s[t*NSessions*RS],
          j[t*NSessions*RJ+15], j[t*NSessions*RJ+14], j[t*NSessions*RJ+13], j[t*NSessions*RJ+12],
          j[t*NSessions*RJ+11], j[t*NSessions*RJ+10], j[t*NSessions*RJ+9], j[t*NSessions*RJ+8], j[t*NSessions*RJ+7],
          j[t*NSessions*RJ+6], j[t*NSessions*RJ+5], j[t*NSessions*RJ+4], j[t*NSessions*RJ+3], j[t*NSessions*RJ+2],
          j[t*NSessions*RJ+1], j[t*NSessions*RJ]], [sf] ) <= MaxCons;
    }
    Forall (jf in [FirstJ .. LastJ] )
    {
      Count([j[t*NSessions*RJ+15], j[t*NSessions*RJ+14], j[t*NSessions*RJ+13], j[t*NSessions*RJ+12],
          j[t*NSessions*RJ+11], j[t*NSessions*RJ+10], j[t*NSessions*RJ+9], j[t*NSessions*RJ+8], j[t*NSessions*RJ+7],
          j[t*NSessions*RJ+6], j[t*NSessions*RJ+5], j[t*NSessions*RJ+4], j[t*NSessions*RJ+3], j[t*NSessions*RJ+2],
          j[t*NSessions*RJ+1], j[t*NSessions*RJ],
          a[t*NSessions*RA+15], a[t*NSessions*RA+14], a[t*NSessions*RA+13], a[t*NSessions*RA+12],
          a[t*NSessions*RA+11], a[t*NSessions*RA+10], a[t*NSessions*RA+9], a[t*NSessions*RA+8],
          a[t*NSessions*RA+7], a[t*NSessions*RA+6], a[t*NSessions*RA+5], a[t*NSessions*RA+4],
          a[t*NSessions*RA+3], a[t*NSessions*RA+2], a[t*NSessions*RA+1], a[t*NSessions*RA]], [jf] ) <= MaxCons;
    }
    Forall (af in [FirstA .. LastA] )
    {
      Count([a[t*NSessions*RA+15], a[t*NSessions*RA+14], a[t*NSessions*RA+13], a[t*NSessions*RA+12],
          a[t*NSessions*RA+11], a[t*NSessions*RA+10], a[t*NSessions*RA+9], a[t*NSessions*RA+8],
          a[t*NSessions*RA+7], a[t*NSessions*RA+6], a[t*NSessions*RA+5], a[t*NSessions*RA+4],
          a[t*NSessions*RA+3], a[t*NSessions*RA+2], a[t*NSessions*RA+1], a[t*NSessions*RA]], [af] ) <= MaxCons;
    }
  }
  Forall (sf in [FirstS .. LastS] )
  {
    Count( s, [sf] ) + Count( j, [sf] ) >= MinS;
  }
  Forall (jf in [FirstJ .. LastJ] )
  {
    Count( j, [jf] ) + Count( a, [jf] ) >= MinJ;
  }
  Forall (af in [FirstA .. LastA] )
  {
    Count( a, [af] ) >= MinA;
  }
 }
}
```