CONSTRAINT PROGRAMMING AND UNIVERSITY TIMETABLING

G.W. Groves and W. van Wijck

Department of Industrial Engineering
University of Stellenbosch
wvw@ing.sun.ac.za

ABSTRACT

The technology of Constraint Programming is rapidly becoming a popular alternative for solving large-scale industry problems. This paper provides an introduction to Constraint Programming and to Constraint Logic Programming (CLP), an enabler of constraint programming. The use of Constraint Logic Programming is demonstrated by describing a system developed for scheduling university timetables. Timetabling problems have a high degree of algorithmic complexity (they are usually NP-Complete), and share features with scheduling problems encountered in industry. The system allows the declaration of both hard requirements, which must always be satisfied, and soft constraints which need not be satisfied, though this would be an advantage.

OPSOMMING

Hierdie artikel beskryf ’n familie van probleem-oplossingstegnieke bekend as “*Constraint Programming*”, wat al hoe meer gebruik word om groot-skaalse industriële probleme op te los. Die nut van hierdie tegnieke word gedemonstreer deur die beskrywing van ’n skeduleringsisteem om die roosters vir ’n universiteit te genereer. Rooster-skeduleringsprobleme is in praktiese gevalle NP-volledig en deel baie eienskappe met industriële skeduleringsprobleme. Die sisteem wat hier beskryf word maak gebruik van beide harde beperkings (wat altyd bevredig moet word) en sagte beperkings (bevrediging hiervan is wel voordelig maar dit is opsioneel.)

1. Introduction

In the last few years constraint programming has attracted much attention as an alternative to solving, particularly, combinatorial problems arising in industry. The importance of constraint programming is emphasised by the fact that the ACM (Association for Computing Machinery) has identified it as one of the strategic directions in computer research [3]. The goal of this paper is both to serve as an introduction to the field, and to present an application of constraint programming to a university lecture-timetabling problem. University Timetabling problems share common characteristics with many resource scheduling problems encountered in industry, and are known to be **NP-Complete** for most practical cases. **NP-Complete** problems have the property that no algorithm capable of solving them in a time bounded by a polynomial function of the input size of the problem is thought to exist. The execution time of algorithms solving these problems therefore grows extremely quickly as the size of the problem increases. Constraint programming has been successfully applied to many **NP-Complete** problems associated with the field of *Operations Research*, notably including scheduling problems [2,5]. Constraint Programming offers several advantages as a solution method, and is a good method to use on problems that need to be expressed in integer variables and that have many logical dependencies between the variables.

Constraints arise in many areas of human activity. They formalise the dependencies in the physical world and are used to guide common-sense reasoning. If I say "I will be busy from five to six o'clock", then I cannot be scheduled to attend a meeting during that time. In mathematical terms, a constraint can be defined as a logical relation among several variables, each taking a value in a given domain. The constraint restricts the possible values that variables can take. Usually, we do not solve one constraint only, but a collection of constraints that are seldom independent.

Constraint programming is the study of computational systems based on constraints. The fundamental principle behind constraint programming is that of stating the constraints about the problem and finding a solution satisfying all of the constraints. This highlights an important feature of constraint programming, namely its declarative nature, i.e. the constraints specify what relationship must hold without specifying a computational procedure to enforce the relationship. In this sense it is similar to using linear programming, where a solver operating in the background determines a solution to a set of constraints. While constraint programming is declarative, the human programmer must however still specify aspects of the solution procedure to be followed. The term `*programming*' in `*constraint programming*' therefore refers to the act of programming a computer, whereas in the field of *mathematical programming* the word has its origins in the United States of America's defence department, and was used to describe programs of activities (i.e. plans) for conflict situations. Constraint programming is done in a constraint programming system, which consists of a constraint and a programming component. The constraint component performs reasoning over the set of constraints, whereas the programming component guides the overall way in which the constraint component will operate. A *constraint program* is therefore not a statement of a problem, as it is for a mathematical program, but rather a computer program that specifies a method for solving a particular problem.

2. Constraint Programming and Constraint Logic Programming

Constraint programming has its origins in research done in the field of artificial intelligence in the 1960's and 1970's. Early applications such as Waltz's scene labelling application [14], as well as the Sketchpad [13] and Thinglab [4] graphics systems contributed toward techniques used in constraint programming systems today. An important step in the development of the field occurred with the integration of constraint techniques and *Logic Programming* languages [7, 8], giving birth to so-called Constraint Logic Programming (CLP) languages. Logic programming languages are declarative, and all program information is expressed as logic statements, on which the logic programming system operates. The execution of a logic program is not explicitly specified in the code, by using instructions for loops and conditions for example, but is instead controlled by the method that the logic programming system uses to evaluate the logic statements. Constraint Logic Programming languages are usually based on the Prolog programming language, a language in which all program information is expressed in *first order logic* form. Program execution in Prolog is controlled according to depth-first search (i.e. tree search). During execution, the system proceeds to evaluate the truth of logic statements until an expression evaluates to *false*. At this point the Prolog system will return to a previous choice point and try another option. This is referred to as backtracking, and this depth-first approach is an inefficient way of solving problems when used by itself. Constraint Logic Programming systems augment logic programming systems with the ability to deal with constraints and a look-ahead feature to improve the efficiency of depth-first search. Constraint Programming is however not limited to CLP in Prolog, and constraints have also been integrated into languages such as C++ and Java.

2.1. Branches of Constraint Programming

There are two branches of constraint programming, namely the branches of *constraint satisfaction* and *constraint solving*. The vast majority of industry applications belong to the branch of constraint satisfaction [3] and it is therefore the focus of this paper. The branch of constraint satisfaction deals with the so-called Constraint Satisfaction Problem (CSP). A CSP is defined as a problem consisting of:

- A set of variables $X = \{X_i, ..., X_n\}$
- For each variable X_i, a finite set D_{X_i} of possible values (its domain), and
- A set of constraints restricting the values that the variables can simultaneously take

The values of the variables need not be a set of consecutive integers (although they often are), and need not even be numeric. A solution to a CSP is an assignment of a value from its domain to every variable, in such a way that all constraints are satisfied at once. The goal of the problem may be to either find a single solution (with no preference as to which one), all solutions, or one which is optimal (or satisfactory) with respect to some objective function defined in terms of some or all of the variables.

The branch of constraint solving is similar to constraint satisfaction in that a set of constraints is stated, which are then solved. The constraints, however, are defined over infinite or more complex domains. The solution methods followed also differ. Instead of using combinatorial methods as in constraint satisfaction, here the algorithms are based on techniques such as automatic differentiation, Taylor series or the Newton method.

2.2. Consistency Techniques

Constraint Programming operates by reducing the domains of each of a problem's variables until each domain is reduced to a single element. This is done by using so-called consistency techniques, constraint propagation methods, and a search strategy. Consistency techniques have the function of deducing domain-related information from constraints, and will attempt to reduce the domains of a constraint's variables by evaluating the relationships imposed by the constraint and removing inconsistent values from the domains. For example, given the constraint

$$X_1 < X_2 + 3$$

with variable domains $D_{X_1} = \{x \in N \mid 1 \le x \le 10\}$ and $D_{X_2} = \{x \in N \mid 1 \le x \le 10\}$, a consistency technique operating on it deduces the new domain values $D_{X_1} = \{x \in N \mid 1 \le x \le 10\}$ and Dx2 = $D_{X_2} = \{x \in N \mid 1 \le x \le 8\}$. In this example, some of the remaining values clearly cannot exist simultaneously in the domains of both variables. Consistency techniques differ in the amount of information that they extract from constraints, and some techniques will attempt to deduce more information relating to the combinations of domain values that are compatible. The most effective techniques, however, do not deduce too much information and remain purposefully incomplete due to benefits in efficiency. Consistency techniques are applied automatically by the constraint programming system, just as the truth testing of logical expressions is applied automatically by a logic programming system. The encounter of an inconsistent constraint is equivalent to deducing a *false* result when evaluating a logical expression, and will result in a backtrack when using a depth-first search strategy.

2.3. Constraint Propagation and Search

Consistency techniques do not usually solve a problem when used alone, although it is theoretically possible that they can reduce the domains of each of a problem's variables to a single value, given an unlikely combination of constraints. In practice, some form of search is required to find variable assignments that are compatible with all constraints. The task of searching for values for variables once the constraints have been specified is referred to as *labelling*. It is possible to use ordinary depth-first search, done in combination with consistency checks on variables thus instantiated, to perform labelling. However, as mentioned earlier, this is an inefficient way to search. Although the efficiency of a program can be improved by using backward checking methods (see Haralick et al. [9]), CLP systems usually make use of *constraint propagation* techniques to achieve a degree of forward-looking ability. Constraint propagation techniques differ in the amount of forward looking that they achieve. The majority of techniques perform simple consistency checks on variables that have recently had their domains reduced, i.e. if the domain of a variable is reduced, constraint propagation triggers consistency checks on all constraints involving that variable, in turn reducing the domains of more variables and triggering further constraint propagation. More complicated methods can remove more inconsistencies [9,12], but these methods are computationally expensive and are not used by most applications. As is the case with consistency techniques, constraint propagation is automatically initiated by the constraint programming system. It is however possible to specify user-defined consistency and propagation behaviour with some systems (e.g. ILOG Solver, Oz, and Eclipse).

The order in which variables are instantiated (the process of assigning a value to a variable is termed *instantiation* in constraint programming) can influence solution time, and solution time can usually be reduced by labelling variables with smaller domains sooner. These variables are more likely to yield backtracking, and it is beneficial to limit backtracking to the earlier parts of the search. The example presented in the next section demonstrates the concepts of constraint satisfaction in constraint programming.

Simple Example

The principles of constraint satisfaction in Constraint Programming are demonstrated through an example problem defined by the code fragment in Figure 1. The code is written in the (Prolog based) Eclipse CLP language. The code of other constraint programming languages (e.g. ILOG Solver) is very similar to Eclipse code.

```
example_solved :-
        Vars = [X,Y,Z],
        Vars :: [1..3],
        X #> Y,
        Y #<> Z,
        forall(V, Vars) {
            indomain(V)
         }.
```

Figure 1. Example constraint satisfaction problem

The program makes the statement that the logic term *example_solved* is true if each of its lines is true (i.e. the comma after each line represents an *AND* statement). The first line of the procedure (note that the word *procedure* used here is not technically correct in Prolog, and is used here for convenience) defines the data structure *Vars*, which is a list containing the three variables *X*, *Y* and *Z*. The second line initialises the domain of each of the variables to all integer values between, and including, 1 and 3. The next two lines of the procedure state the constraints of the problem. The hash symbol instructs the system that the constraint is an active constraint (i.e. it will be stored in memory and will become awoken by constraint propagation). The *forall* command iteratively initiates *V* to contain each of the variables in *Vars*, and is actually a macro defined by the Eclipse system. In pure Prolog code, the functionality of loops such as these is achieved by using recursive

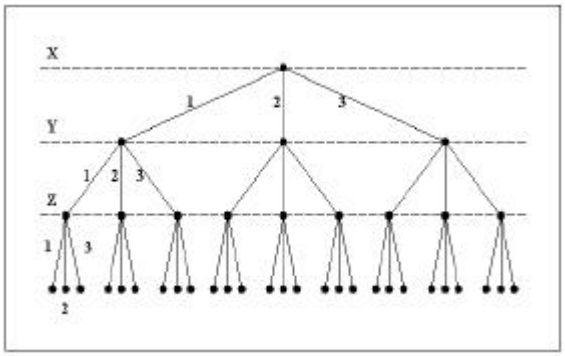

Figure 2. Full search tree for example problem

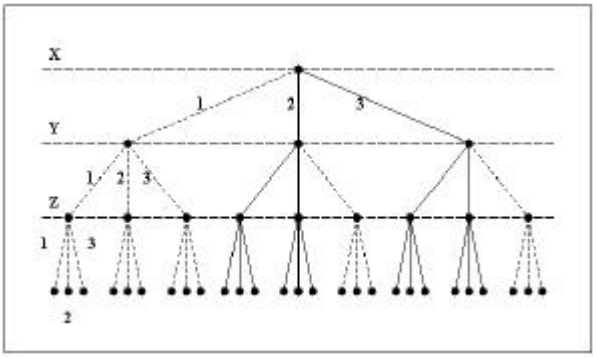

Figure 3. Search tree before the start of labelling

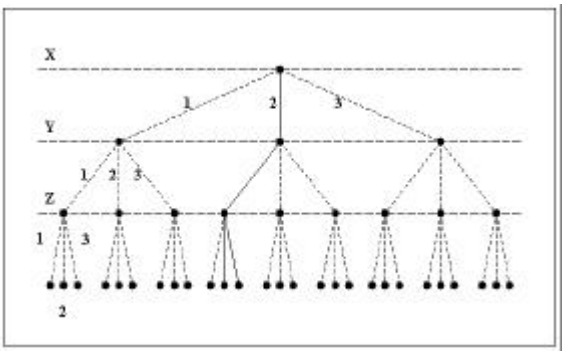

Figure 4. Search tree after iteration 1 and 2 of labelling

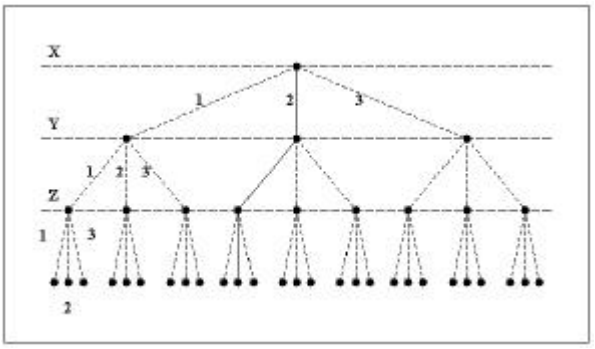

Figure 5. A solution to the example problem

procedures. The *indomain* command instantiates a variable to contain the smallest value of its domain. The *forall* loop and the *indomain* command contained in it represent the labelling function of the program. Figure 2 is a search tree for the problem. The important steps in the solution of the problem are now described in more detail.

Domain Reduction before labelling

Consistency checks take place as soon as constraints are stated, and constraint propagation will occur if any domains are reduced. When the first constraint is encountered, it is evaluated for consistency and the domains of variables *X* and *Y* are reduced, and the new variable domains become

$$D_X = \{2,3\}, \quad D_Y = \{1,2\}, \quad D_Z = \{1,2,3\}.$$

After evaluating the next constraint, no new domain reductions are made. Both of the constraints still have un-instantiated variables, and they therefore return to a dormant state in memory, until they are again awakened by constraint propagation. Figure 3 shows the reduced search tree for the problem

Domain Reduction during iteration 1 of labelling

The *indomain* command assigns the value of 2 to *X*, being the smallest value of D_x. The first constraint (*X #> Y*) awakens and the domain of Y is consequently reduced by removing the value 2. This domain reduction is propagated to the other constraint (*Y#<>Z*), resulting in the removal of the value 1 from its domain. The new domains are given by

$$D_X = \{2\}, \quad D_Y = \{1\}, \quad D_Z = \{2,3\}.$$

The domains of both *X* and *Y* have therefore been reduced to one value. The new search tree is shown in Figure 4.

Domain Reduction during iteration 2 of labelling

The value of the variable Y has already been deduced, and no new information is determined during this step. The search tree stays the same as that of Figure 4. Note that if the domain of

any of the variables becomes empty at any point during the procedure's execution, the system will backtrack to the nearest point where it can make an alternative choice and try another option. During labelling this implies that the *indomain* command will be executed again to assign a different value to the variable being labelled. If all values have been tried, the system will backtrack even further, to the previous variable, and try to label it again. During backtracking, all domain reduction that took place since the last time that the variable was labelled is reversed.

Domain Reduction during iteration 3 of labelling

During this iteration, the variable Z is assigned the value of 2. The problem is now solved because all of the variables have been assigned values. Figure 5 shows the final solution.

This section has described the principles of constraint programming and constraint logic programming. The next section presents a university timetabling application of constraint logic programming.

3. Timetabling System

The timetabling system was developed to determine timetables for the Industrial Engineering Department of Stellenbosch University. The task of finding timetables manually is a cumbersome duty that the system was expected to alleviate. The system attempts to schedule a fixed number of classes in a way that no constraints are violated. Each subject has a fixed duration (in periods), and this value is specified beforehand. The system also accepts "soft" constraints, which should preferably be satisfied, though this is not a requirement. The soft constraints are accommodated through the use of a branch-and-bound optimisation procedure.

3.1. Problem Model

The problem is modelled as a constraint satisfaction problem. The following integer constants are defined.

D : Number of days in a week
A : Number of periods in an afternoon
M : Number of periods in a morning
S : Number of subjects to timetable
R : Number of available rooms
S_k : First domain value for day k

The following variables are used

x_i : starting time of class i
r_i : room assigned to class i.

In addition, the following expressions are defined

$\text{dur}(x_i)$: duration of subject i, in time slots.
$\min(D_{x_i})$: minimum value in the domain of x_i

The variables x_i and r_i are the problem variables, and their domains are given by

$$D_{x_i} = \left\{ \; x \in N \mid \bigcup_{j=1}^{D} \left(\left(S_j \le x \le S_j + M - 1 \right) \cup \left(S_j + M \le x \le S_j + M + A - 1 \right) \right) \right\}$$

$$D_{r_i} = \left\{ x \in N \mid 1 \le x \le R \; \right\}$$

The values of S are chosen so that the number of integer values separating the starting value of any day and the ending value of its preceding day is a constant value. Let s denote this constant. Hence,

$$s = S_k - \left(S_{k-1} + M + A + 1 \right), \quad 2 \le k \le D$$

The reason for separating the domain values of consecutive days is to simplify the use of hard constraints that schedule subjects on either separate or the same days (see Section 3.1.1). For these constraints to function properly, the value of s must be greater than the total number of time slots in a day.

3.1.1. Hard Constraints

The system supports a large number of hard constraints. These can be divided into *unary* constraints that operate on a single variable, removing values directly from its domain, and more complicated constraints involving several variables. The latter group are specified as specialised constraint procedures, which behave as ordinary constraints (i.e. they are suspended in memory and awaken by constraint propagation through their variables). The hard constraints supported by the system are now described.

Unary Constraints

These types of constraints involve directly reducing the domain of a single variable before labelling occurs.

- x_i occurs in period P: $x_i \# = P$
- x_i does not occur in period P: $x_i \# <> P$
- r_i occurs in room R: $r_i \# = R$
- r_i does not occur in room R: $r_i \# <> P$
- x_i occurs in a morning: $D_{x_i} = D_{x_i} - \left\{ x \mid \bigcup_{j=1}^{D} \left(S_j + M \le x \le S_j + M + A - 1 \right) \; \right\}$
- x_i occurs in an afternoon: $D_{x_i} = D_{x_i} - \left\{ x \mid \bigcup_{j=1}^{D} \left(S_j \le x \le S_j + M - 1 \right) \; \right\}$

Subjects that may not overlap

The majority of constraints belong to this category. For example, all subjects that share at least one lecturer or that belong to the same year group may not overlap. The *no_clash* constraint procedure, described in the pseudo-code listing of Figure 6, enforces this constraint. The procedure accepts the starting time variables of two subjects, X and Y, as input parameters.

$$
\begin{aligned}
no\_clash(X,Y) \equiv\ & min(D_x) + dur(X) > max(D_y) \Longrightarrow Y + dur(Y) \#\leq X \\
& OR \\
& min(D_y) + dur(Y) > max(D_x) \Longrightarrow X + dur(X) \#\leq Y \\
& OR \\
& (SUSPEND)
\end{aligned}
$$

Figure 6. The *no_clash* constraint procedure.

The constraint procedure tests if the earliest possible starting times of either of its two input subjects is later than the latest possible ending time of the other. If this is the case then an appropriate constraint is introduced to ensure that the subjects do not overlap. If these conditions cannot be deduced, then the procedure is instructed to suspend itself in memory, until it can again be awakened by constraint propagation. Note that this procedure introduces more constraints into the problem during labelling, because generally only during labelling will enough information become available for one of the two conditions to be satisfied. This procedure would be awkward to formulate in an integer program, and would require the introduction of auxiliary variables.

Subjects that must occur on the same day

The *same_day* constraint procedure, listed in the pseudo code of Figure 7, enforces this constraint.

$$
\begin{aligned}
same\_day(X,Y) \equiv\ & min(D_x) \geq max(D_y) \Longrightarrow X\text{-}Y \#> s \\
& OR \\
& min(D_y) > max(D_x) \Longrightarrow Y\text{-}X \#> s \\
& OR \\
& (SUSPEND)
\end{aligned}
$$

Figure 7. The *same_day* constraint

The procedure introduces a constraint forcing the number of domain values separating its two input variables to exceed the value of s (defined earlier), depending on the domain information available.

Subjects that must occur on different days

The *different_day* constraint procedure, listed in the pseudo-code of Figure 8, enforces this constraint.

$different\_day(X,Y) \equiv min(D_x) \geq max(D_y) \Longrightarrow X\text{-}Y \#> s$
OR
$min(D_y) > max(D_x) \Longrightarrow Y\text{-}X \#> s$
OR
(SUSPEND)

Figure 8. The *different_day* constraint.

Its operation is similar to the *same_day* constraint.

Room constraints

No two subjects may simultaneously share the same room. This requirement is enforced by the *room_no_clash* constraint procedure, listed in Figure 9. The procedure accepts two subject starting time variables, X and Y, as well as the respective room variables, A and B, of the subjects.

$room\_no\_clash\ (X,Y,A,B) \equiv (\ max(D_x) + dur(X) \leq min(D_y)$
$\quad OR\ max(D_y) + dur(Y) \leq min(D_x) \Longrightarrow true\ )$
OR
$(\ min(D_x) + dur(X) \geq max(D_y)$
$\quad AND\ min(Y) + dur(Y) \geq max(D_x) \Longrightarrow A \#<> B)$
OR
$A <> B \Longrightarrow true$
OR
(SUSPEND)

Figure 9. The *room_no_clash* procedure

The procedure operates as follows: If the two subjects cannot overlap no constraint is generated, and if they are known to overlap, then a constraint forcing the rooms to be unequal is introduced, and if the rooms are already known to be unequal then no new constraint is introduced.

3.1.2. Soft Constraints

Soft constraints are optimised using a branch-and-bound algorithm, as was mentioned earlier. The branch-and-bound method has a long history in the field of operations research, and is traditionally used with the simplex method to solve integer-programming problems. Modern integer programming systems, however, use more efficient algorithms than branch-and-bound. Most CLP systems provide a packaged branch-and-bound algorithm. Several types of soft constraints are supported by the timetabling system

- Subjects that should not overlap
- Preferred starting times for subjects
- Times at which a subject should preferably not be scheduled in
- Preferred number of days separating subjects

The objective function used by the branch-and-bound algorithm is expressed as a sum of the total number of soft constraint violations, weighted according to the type of constraint being violated. The weights, as well as termination criteria, are specified by the user.

3.1.3. Labelling

The system labels all *starting time* (x_i) variables before the room variables (r_i), the reason being that it is usually more difficult to find a feasible subject timetable than to allocate rooms. Note that although the room variables are labelled later, the *room_no_clash* procedure can affect the search beforehand, because of the fact that it has starting time variables as arguments. When choosing variables to label, the remaining variable with the smallest domain is selected during each iteration.

3.2 Results

The system solves the timetables of the Industrial Engineering Department of Stellenbosch University near instantaneously, when using an Intel Pentium 600MHz personal computer. If soft constraints are included, the execution time increases to between 30 seconds and 5 minutes, depending on termination choices. It solves the whole engineering faculty's timetables, consisting of 150 lessons, in under a minute when using only hard constraints. These times compare favourably to other implementations [1,5] of similar size. It is however dangerous to make straightforward comparisons of execution time based on the size of the problems being solved. Different implementations not only make use of other hardware and software, but the solution time of timetabling problems do not grow according to a polynomial function of the size of the input problem. Without taking these factors into account, the nature of the constraints can also be an important factor governing execution time. Certain scenarios may cause a lot of backtracking, causing a seemingly small problem to take a long time to execute.

4. Conclusions

This paper has attempted to convey the principles of constraint satisfaction techniques in constraint programming through the description of a constraint logic programming application for timetabling problems. An important benefit of problem solving in constraint programming is its flexibility and expressiveness: constraints with complicated behaviour are natural to express, and programs are easy to maintain because new constraints can be easily incorporated without changing much code. The timetabling application presented greatly facilitates the task of generating timetables and seems to have a good execution speed. The system differs from existing timetabling applications due to the large number of constraint types that it supports. The technical characteristics and behaviour of the application are equally relevant to the broader field of constraint programming as to constraint logic programming.

5. References

[1] Abdennadher et al. *University Timetabling using Constraint Handling Rules*, Proceedings of the French speaking seminar on Logic Programming and Constraint Programming, Nantes, France (1998)

[2] P. Baptiste & C. Le Pape. *Disjunctive Constraints for Manufacturing Scheduling: Principles and Extensions*, Proceedings of the Third International Conference on Computer Integrated Manufacturing, Singapore (1995)

[3] R. Bartak. *Constraint Programming: In Pursuit of the Holy Grail*, in: Proceedings of WDS99, Prague (1999)

[4] A. Borning. *The Programming Language Aspects of ThingLab, A Constraint-Oriented Simulation Laboratory*, in: ACM Transactions on Programming Languages and Systems, 3 (4), pp. 252-387 (1981)

[5] T. Chase et al. *Centralized Vehicle Scheduler: An application of Constraint Technology*, in: Proceedings of INFORMS conference, Montreal (1998)

[6] Frangouli et al. *UTSE: Construction of Optimum Timetables for University Courses - A CLP Based Approach*, Proceedings of the 3rd International Conference on the Practical Applications of Prolog, Paris (1995)

[7] H. Gallair. *Logic Programming: Further Developments*, in: IEEE Symposium on Logic Programming, Boston (1985)

[8] J. Jaffar & M.J. Lassez. *Constraint Logic Programming*, in Proceedings of the ACM Symposium on Principles of Programming Languages (1997)

[9] R.M. Haralick & G.L. Elliot. *Increasing tree search efficiency for constraint satisfaction problems* Artificial Intelligence, 14, pp. 263-314 (1980)

[10] C. Le Pape. *Constraint-Based Programming for Scheduling: An Historical Perspective*, Working Notes of the Operations Research Society Seminar on Constraint Handling Techniques, London (1994)

[11] I.J. Lustig & J. Puget. *Program Does Not Equal Program: Constraint Programming and its Relationship to Mathematical Programming*, Interfaces, 31 (6), pp 29-53 (2001)

[12] B. Nadel. *Tree Search and Arc Consistency in Constraint Satisfaction Algorithms*, in: Search in Artificial Intelligence, Springer-Verlag, New York (1998)

[13] I. Sutherland. *Sketchpad: a man-machine graphical communication system*, in: Proceedings of the IFIP Spring Joint Computer Conference (1963)

[14] D.L. Waltz. *Understanding line drawings of scenes with shadows*, in: Psychology of Computer Vision, McGraw-Hill, New York (1975)