PATHOS: Object-Oriented Concurrent Constraint Timetabling for Real World Cases

Juan Francisco Díaz
Universidad Del Valle
Cali, Colombia
jdiaz@eisc.univalle.edu.co

and

Luis O. Quesada
University of Louvain
luque@info.ucl.ac.be

and

Camilo Rueda
Universidad Javeriana-Cali
crueda@atlas.puj.edu.co

and

Catherine García
Universidad Javeriana-Cali
artemisa@puj.edu.co

and

Sandra Cetina
Lince Tech
lince@parquesoft.com

Abstract

University timetabling is a fundamental periodic activity in academic planning. In its general setting this problem is $\mathcal{NP}$-complete. Devising effective strategies for solving it has been a challenge for several decades. Most approaches that work in real applications either find approximate solutions or only consider constraints of a very specific type. Our approach has been to tackle the full complexity of the problem using concurrent constraint programming techniques. We describe *PATHOS*, a concurrent constraint object oriented software written in *Mozart* that has been effectively used in real life situations. *PATHOS* improves the state of the art in automatic timetabling strategies since it handles problems of a much bigger size than has been considered so far using constraint programming techniques. The strategies and techniques used in *PATHOS* are also applicable to other types of planning and scheduling problems.

Keywords: Timetabling, Concurrent Constraint Programming

Automating scheduling of rooms and courses in an university has attracted attention since the 70's. Earlier attempts assigned courses incrementally based on local reasoning criteria. For example, scheduling courses sequentially in decreasing order of the number of constraints involving them ([4]). Later on in the 80's, methods based on linear programming and network flows were used ([5]). More recently, the problem has been tackled by means of genetic algorithms ([8]), simulated annealing ([1]), constraint satisfaction ([7])and concurrent constraint programming ([2, 6]). The existence of an international conference on the topic (PATAT) bears witness of the fact that the problem remains a very active research topic.

The challenge is to provide a coherent computational model taking into account a whole variety of requirements coming from very different sources. For example, lectures for certain courses are required to be given in rooms of some specific size, type or location, scheduled within given time slot ranges, required not to overlap with some other lectures and satisfy lecturer schedule constraints. Since the timetabling problem is NP hard, it is usually assumed that approximation methods (such as genetic algorithms) are unavoidable when handling real-life situations.

The main contribution of this paper is to question this belief by exhibiting a system based entirely on constraint programming techniques that has been used successfully to solve a problem instance of more than 2000 variables. We have been able to do so by a judicious choice of constraint propagators and a carefully chosen balance between search and propagation. To our knowledge, no solutions to problems bigger than 700 variables using these techniques have been reported before.

Solving problems of important size requires a through understanding of the concurrent constraint model and of its implementation. Our system is implemented in the concurrent constraint programming language *Mozart* ([11]). The experience we developed in *Mozart* allowed us to identify certain limitations of its current implementation that have a substantial impact on the efficiency of our implementation. We argue that removing these limitations should be a concern for *Mozart* developers. Another contribution of our work is the implementation of a new general search mechanism for *Mozart* that achieves significant improvements in efficiency for problems involving a great number of variables.

The paper is organized as follows: In section 2 the notion of constraint programming is introduced. In section 3 we specify the object-oriented architecture of the system and describe the problem representation in terms the finite domain constraints system of *Mozart*. Section 3.4 discusses the representation of *soft constraints*, section 4 shows time and space performance measures for several problem instances. Finally, in section 5 we give conclusions.

2 Concurrent constraint programming model

A great variety of combinatorial problems can be expressed as searching for one or several elements in a vast space of possibilities. In general, the search space is defined as all combinations of possible values for a predefined set of variables. Elements to be searched for are particular values of these variables. In most cases the desired values of the elements are implicitly specified by properties they should obey. These properties are known as *constraints*. Constraints are usually expressed as predicates over some set of variables. Concurrent constraint programming (CCP) languages ([9]) take these predicates as basic primitives. Most notably, the store-as-valuation concept of Von Neumann computing is replaced in CCP by the notion of store-as-constraint. Constraints provide partial information about variables (e.g. $x < y + 2$) that accumulates monotonically in the store as the computation proceeds.

In the basic model of concurrent constraint programming the store contains predicates that represent partial information over variables. Concurrent agents interact with the store adding information (*tell* agents) or asking questions (*ask* agents). *Tell* agents add constraints to the store whereas *ask* agents query the store to see if a given constraint can be deduced from it. For example, let the available agents be

$$tell(X > 10),\ ask(X < 50)\ then\ P, ask(X = 15)\ then\ Q,\ tell(X < 20).$$

Considering this agents from left to right they exhibit the following behavior: information $X < 10$ is first added to the store, then it is asked whether $X < 50$ can be inferred from the store. Since this is not the case, the *ask* agent blocks. Similarly, the *ask* agent $X = 15 \to Q$ is blocked. When the *tell* agent $X < 20$ adds this information, the *ask* agent $X < 50 \to P$ is resumed and can then continue with process P. The result of a process in this model is the set of variable values that are compatible with all constraints added by the process to the store.

All concurrent constraint programming language (CCP) guarantee efficient propagation processes. There are, however, some differences regarding the set of available constraints and the type of constraint system they operate upon. A common type is the *finite domain constraint* system. In this system each variable has

variables. The model discussed in this paper uses the powerful finite domain constraint system of *Mozart*.

We use an object oriented architecture for the system. We describe next its components.

3 Problem representation in Pathos

The structure of classes in *Pathos* mimics the usual organization structure of a university from the point of view of timetabling. There are seven main elements: university, faculty, program, curriculum, course, group and session. A curriculum is the set of courses that the students of a given program (i.e. academic unit, department, etc) should, in principle, take together in a given academic period. Therefore, their schedule cannot contain intersections. A course can be divided into groups. Each group is assigned a number of lecture sessions in the basic period (a week, for example).

The only relation between classes is containment. For example, a university contains faculties, instructors and buildings. There are constraints involving information about each one of these classes. For instance, constraints about the availability of instructors make reference to the instructor class, while constraints about the type of room needed for a session make reference to the building class.

The timetabling problem expressed in the setting of concurrent constraint programming must define variables and constraints between those variables. In the following we describe variable representation.

3.1 Finite domain variables

All variables in *Pathos* are *finite domain* variables. In *Mozart*, a finite domain is a set of integers. The notation $m\#n$ represents the set of all the integers in the interval $[m, n]$. Constraints over finite domain variables are called domain constraints. A domain constraint has the form $x :: D$, where D is a finite domain. Domain constraints can also be expressed in the form $x = n$ since $x = n$ is equivalent to $x :: n\#n$. Similarly for inequations and inequalities. Terms of the form $n\#m$ are data structures in *Mozart*. This allows to assign $n\#m$ to a variable and then use this variable to define complex domains.

In *Pathos*, each variable represents the beginning time of a particular session. Initially these variables are assigned the interval 1#168 corresponding to all possible time slots in the basic period, which has been defined as a week. For the particular application we describe in this paper we consider 6 day weeks, each day having 28 possible starting slots (since slots are 30 minutes long and courses are scheduled from 7:00AM to 1:00 PM and from 2:00 PM to 10:00 PM). For example, to establish that a session S start on Tuesday at 7:00am, constraint $S.start = 29$ must be asserted. Similarly, we could constrain S to be scheduled on Tuesday by asserting $S.start :: 29\#56$.

Since all variable in *Pathos* represent sessions this means that every constraint must be expressed as a constraint over sessions. For example, if a lecturer availability is the range R, then all sessions this lecturer teaches are constrained to be in the range R.

The task of *Pathos* is to determine the starting slot of each session involved in the problem so that at least the following conditions hold:

- Two session of the same group must be scheduled on different days (except when one of the sessions is defined as a laboratory).
- Groups of courses of the same curriculum must be scheduled in such a way that a student registered in a given group (or recitation) could also register in at least one of the groups of each of her courses.
- All sessions of a given instructor must be non overlapping.
- The number of sessions using a certain type of room at some specific hour cannot be greater than the number of rooms of that type available at that hour.

The above constraints are asserted by default. Any solution *Pathos* finds must maintain them. The user, however, can specify some of the following constraints according to the characteristics of his problem:

- The beginning of a session can be restricted to a set of specific hours.
- A session must end before another one starts or two given sessions should not overlap.
- At least one group of each one of two courses (not necessarily belonging to the same curriculum) must not overlap.
- The availability of an instructor can be restricted to some hours.

- A curriculum can be associated to a maximum number of sessions per day.

The user can assert these additional constraints either as mandatory or *weak*. Any solution *Pathos* finds must satisfy mandatory constraints. *Weak* constraints are discarded when no solution considering them exists. In addition, a visual programming interface allows defining new constraints and so tailoring *Pathos* to each particular case.

A brief description of the interface follows.

3.2 Interface

Pathos takes a XML file for input data. This file describes the structure of the university (i.e. faculties, departments, curricula, rooms and their types, etc.) from information that is usually taken directly from a data base using SQL. Data concerning constraints can also be included in the XML file or entered manually using a graphical interface. The main window in this interface is split in two parts, one showing a tree representing the university academic structure and the other containing a box for each course in a particular academic program. The boxes shown correspond to courses in the department selected in the tree. Courses are presented as a set of columns where each column contains all courses of a given term. The user selects any two courses (or groups, sessions from a group, etc) by clicking on them and then chooses from a menu the type of constraint to assert and whether it should be handled as a strong or weak constraint (see figure 1).

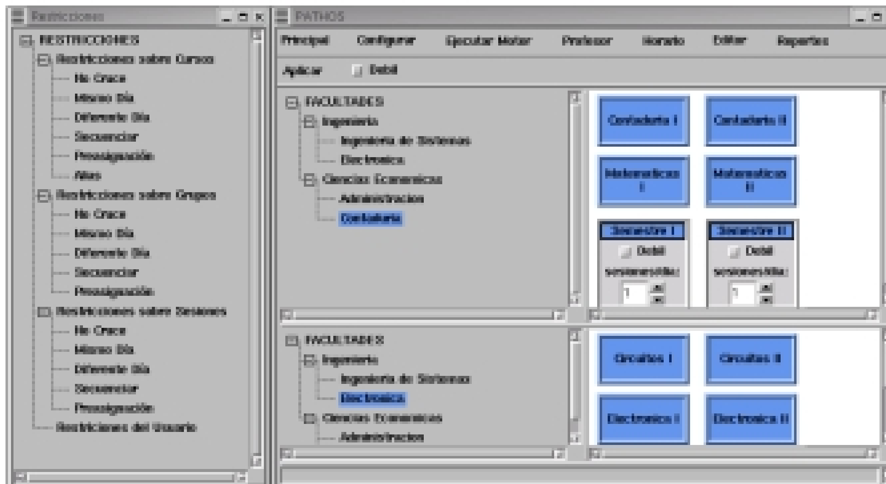

Figure 1: Selecting Constraints

The interface includes a visual language for programming user defined constraints (see figure 2). Boxes in this language represent predicates. Each predicate box contains a number of entries. Lines are used to connect entries from different predicate boxes. Semantically, a connection asserts that the corresponding entries are constrained to be equal. A variety of types of predicate boxes are supplied, such as the one asserting that the schedule of each pair of its entries must be such that no intersections occurs. These predicate boxes are the basic primitives of the visual language. Any entry left unconnected is supposed to refer to a course, group or session. For these entries the user only specifies the type (e.g. *group*), but not the particular instance of this type the constraint should be applied to. Constraints defined by the user in the visual language are compiled into the main graphical interface so that they appear in the menu showing all available constraints. In this way they can be selected and asserted exactly as the builtin constraints.

There are also suitable graphical representations to ease constraint parameterization. For example, all constraints referring to time ranges, such as instructor availability in the period, are provided a representation in the form of an actual time table where the appropriate time slots can be selected. Space restrictions preclude their inclusion in this paper.

3.3 Implementation of constraints

Constraints the user specifies in *Pathos* are implemented as procedures in *Mozart*. A simple example is the following, which asserts that two sessions (the arguments of the procedure) are scheduled in different days.

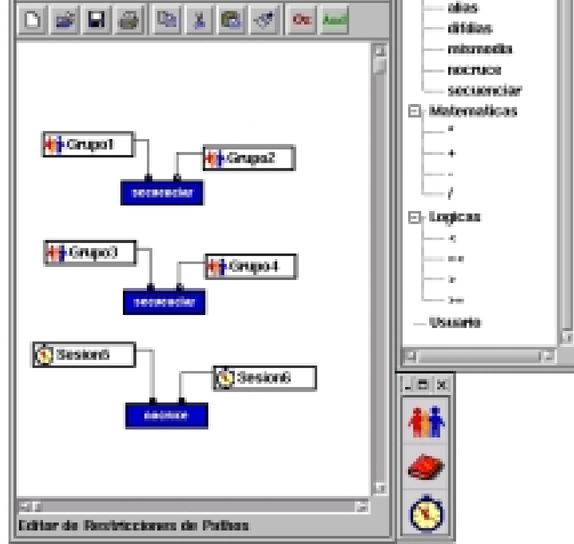

Figure 2: Visual constraint programming

```
1    proc    {Different_days St1 St2}
2            thread  cond
3                St1::[Monday] then
4                St2::[Tuesday Wednesday Thursday Friday Saturday]
5            []
...          ...
19               St1::[Saturday] then
20               St2::[Monday Tuesday Wednesday Thursday Friday]
21             else skip
22             end
23           end
24           thread  cond
25               St2::[Monday] then
...          ...
44             end
45           end
46   end
```

In the above procedure lines after 24 are symmetric to lines between 4 and 21. Once the range of possible schedules for any one of the two sessions is known to belong to a certain day, the procedure eliminates that same day from the range of the other. Each thread in the procedure can be regarded as a concurrent *propagation* agent, i.e. one that waits for any changes to occur in the domain of a variable ($St1$ or $St2$ in the example) in order to propagate its effects, that is, to reduce the domains of other variables.

When all propagation stabilizes the domain of the variables has possibly been greatly reduced but not necessarily determined in a single value. To find a solution some exploration process must be launched to choose one of the remaining values of the domain of some variable. This may cause further propagation. Finding a solution in a *CCP* languages entails thus a close interaction between propagation and exploration. When propagators are weak domain reduction is low but the propagation process can be efficiently implemented. When they are strong many values are eliminated from the domains but the propagation process can be slow. Since each choice of value in exploration awakes propagators a careful balance must be found between strength and efficiency. In *Pathos* propagators are usually strong. In procedure $\{Different_daysSt1St2\}$, for example, as soon as it is known that session **St1** is restricted to Monday (line 3), session **St2** is restricted to slots other than Monday (line 4). The same occurs for each one of the other days.

In many practical applications some constraints are intended to model preferences, predicates one would like it to hold but are not considered essential. These are called *weak* or *soft* constraints. They represent properties that are desirable but not mandatory. As far as we know, *CCP* languages do not have an explicit mechanism to define weak constraints (an extension of the formal model of *CCP* to handle *soft* constraints has been proposed recently in [3]). In *Mozart*, a simple type of weak constraint can be modeled using the notion of disjunction.

thread dis $\{P\}$ **then skip** [] **skip then skip end end**

Here, P represents any constraint that is to be treated as being *soft*. The *dis* construct creates local computation spaces for its alternatives, so that the effects of running them could easily be undone when a failure occurs. When P cannot be satisfied or, equivalently, when asserting P in the current computation space leads to failure (i.e. there is no solution), then the disjunct where P appears is simply discarded. In this case the only other enabled clause would then be the null clause (**skip then skip**). Since any propagator can take the role of P this is a way of 'testing' any constraint, i.e. of asserting it only if doing so does not lead to a failure.

The above scheme implements, of course, a very simple form of *soft* constraint. In particular, it does not handle preferences among constraints. Nevertheless it can be profitably used to model other forms of preferences. For example, *Pathos* lets the user associate a sequence of types of rooms to a session, the idea being that the order in the sequence corresponds to the order of preference for the user. The sequence is compiled into a *dis* construct of the above form, where P constraints the session to one of the types of rooms. Ordering the alternatives of this construct as in the sequence of types ends up ordering type of rooms by user preferences.

4 Efficiency analysis

As mentioned before, *Pathos* is implemented in *Mozart* as a collection of concurrent propagation agents, each running in its own thread. In general, it is not possible to predict the order of execution of threads for a given program. Since the execution order of propagators strongly influences total running time, it is very difficult to determine with some accuracy the runtime behavior of *Pathos*. Nevertheless, it is clear that the number of computation spaces generated in the path to a solution greatly affects running time.

Constraints conceptually work on a global computation space. As mentioned above, the presence of disjunctions makes it necessary to construct local computation spaces. The existence of theses spaces makes the language very powerful, but the price to pay is the amount of time and memory needed to maintain them. Each time a new computation space is needed it must be decided what to do with the current one. There are two possibilities(see [10]): either to save it so as to be able to recover it in case the new one fails or simply to discard it but keeping enough information around to be able to reconstruct the space in case of a failure.

To optimize running time there should be a careful balance between saving and recomputing spaces. One could decide, for instance, to save one out of d computation spaces. That is, spaces numbered $1, d, 2d, 3d, \ldots$ will be saved, whereas spaces $2, \ldots, d-1, d+1, \ldots, 2d-1, \ldots$ would have to be recomputed if needed. Number d is called the *recomputation distance*. Up to a point, increasing the recomputation distance means saving execution time since the time used saving the spaces could very well be greater than the time used recalculating them. This might happen when few shallow failures occur. On the other hand, increasing d beyond a certain point may impose very frequent recomputations and thus greatly increase running time, a situation one would expect to aggravate as d continues to increase.

In general, one would then expect the execution time in relation to recomputation distance to begin by decreasing for a short range of increasing values of d and then to start increasing smoothly for higher values of d.

For our real case, however,the actual observed behavior was the one in figure 3. Here running time does not increase smoothly after some point as d increases. Moreover, running time actually decreases drastically for some high values of d. The reason for this non intuitive result can be understood by considering that determining some variables takes more time than determining others. In our case, the first 30% of variables are determined rather quickly and there are no significant differences in running time for different values of d.

There is a direct relationship between the difficulty of finding a consistent value for a variable and the number of choice points associated to it. When there are no failed attempts to determine a variable there is only one choice point associated to that variable. Each failed attempt creates an additional choice point.

In table 1 we show the number of choice points for each set of 212 variables *Pathos* determines. The third column shows the relative increase in those choice points. Thus, for the first 849 variables, 3997 choice

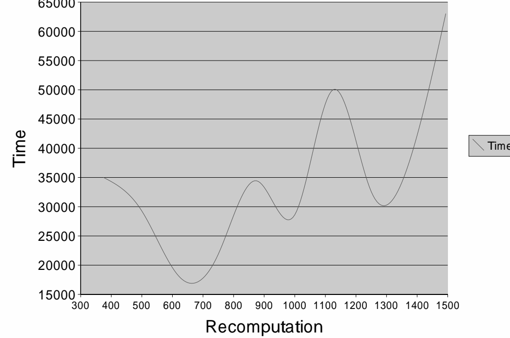

Figure 3: Real behavior

Variables	Choice Points	Relative difference
0	3190	-
212	3409	219
424	3647	238
637	3760	113
849	3997	237
1061	4199	202
1273	4424	225
1485	4757	333
1698	5193	436
1910	5653	460
2122	5944	291

Table 1: Variables versus choice points

points are needed, representing an increment of 237. It can be observed that for up to 60% (1273) of the variables, choice points increase roughly as the number of variables. This means those are easily determined variables. The last 40% of the variables, however, are increasingly more difficult to determine (except the last 10%). From this it can be inferred that saving computation spaces in the neighborhood of the limit of 60% of the variables could result in important running time savings. If this is the only "difficult" region it suffices to take as d some approximate divisor of 4424 (choice points for 60% of the variables).

To summarize, running time strongly depends on the number computation spaces created and saved and these are dependent on the number of choice points. The distance of recomputation should be such that spaces are saved in the neighborhood of difficult regions. When several difficult regions exist it is not obvious how to choose a single d so that its multiples fall in the right places. Moreover, it is probably not very useful having to run the application once without any optimization just to identify the number of choice points and the difficult regions. An alternative is described further below.

4.1 Copying propagators

In real life timetabling problems the size of a given computation space can be large since it depends, among other things, on the number of undetermined variables. The current implementation of *Mozart* defines maximum RAM memory to be 512Mb. This means that there is not enough room to save many computation spaces and therefore some recomputation of spaces would be anyhow needed. It is thus important to avoid as much as possible commitment to choices leading to failure. In *Pathos* propagators have been designed with this aim in mind. However, a limitation of the current *Mozart* implementation makes it difficult to accomplish this purpose. Indeed, in *Mozart* propagators from a parent computation space are not copied to its children spaces. Tis is a major source of inefficiencies.

Whenever a new local computation space is created, all basic constraints applied in its father space are inherited. However, the same does not occur with the propagators (or agents) of the father. This may hinder the possibility of predicting possible failures, as is shown in the program below.

In the above program, assuming both X and Y having initial domain equal to 1#5, one would count on having enough information to infer that in the *dis* construct the option $X = 1$ is not viable. This is because

```
3        dis
4            X = 1 then {P}
5        []
6            X| =: 1 then {Q}
7        end
8    end
```

if $X = 1$ then the agent in line 1 would cause the computation space to fail (since $Y :: 1\#5$ and $Y :: 10\#20$ are inconsistent). However, this inference is not considered at the moment of analyzing the alternative $X = 1$ because each clause of the *dis* construct executes in a local computation space and the agent of line 1 does not belong to it. This agent belongs to the father of that local space but propagators are not copied from a father to its children. As s result, the explorer blindly commits to the *dis* alternative $X = 1$ (thus running P) and it is only after attempting to merge the results in the child space and in its father space that the explorer faces the failure of the computation space. Since each failed commitment causes a new cycle of space recomputations this strongly affects efficiency.

4.2 Improving the Search Engine of *Mozart*

The size of our problem demanded a search engine supporting recomputation distance. We considered for this the search engine in (http://www.mozart-oz.org/documentation/system/ node9.html#chapter.search). In this search engine the user defines the number of tree levels (choice points) before a space is saved. For instance, in a 10 level search tree with recomputation distance equal to 3, only 4 spaces are saved (those at levels 0, 3, 6 and 9). The problem with this approach is that it does not take into account the relation between the level reached and the time spent getting to it. In our case determination of variable begins beyond level 3190. Reaching this level takes one minute or less but 12 spaces are saved up to that point. This is a big waste of resources.

We decided to modify the source code of the search engine, which was kindly provided by Christian Schulte. Our search engine defines recomputation distance in terms of time rather than tree levels. The user specifies the number of seconds that must elapse before a new computation space is saved. Certainly this leads to a better memory administration since finding consistent values for difficult variables usually takes more time and therefore more memory is invested saving spaces in their neighborhood.

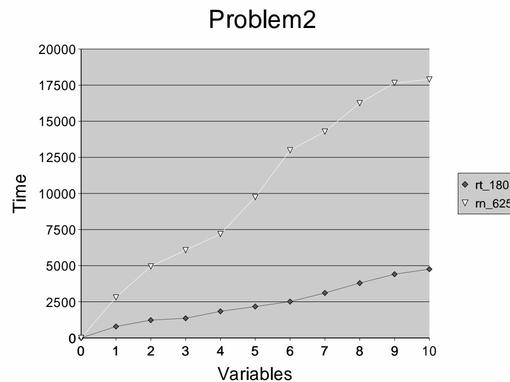

Figure 4: Old Engine Versus New Engine in Problem 2 (all variables)

There was a remarkable improvement when when using our search engine. Figure 4 shows the relation between the two engines in the real case taking into account the whole university timetabling problem. Time is shown in seconds and the *Variables* coordinate refers to the percentage of variables that are already determined.

In this paper we described *Pathos*, a concurrent constraint programming implementation of a university timetabling system. We showed its general architecture and interface. We discussed the implementation of problem constraints as strong propagators and developed the idea of using the *Mozart dis* construct to model *soft* constraints. We analyzed the efficiency of our system in terms of the number of computation spaces generated and saved. In this context we raised the issue of a particular limitation of the current implementation of the *Mozart* language that significantly affects efficiency. A new search engine controlling recomputation of spaces by time instead of exploration tree depth level was described. We showed the remarkable improvement in efficiency achieved by using this engine instead of the one supplied with *Mozart*. Finally, we argued that real cases of timetabling can indeed be handled using *CCP* by carefully controlling propagation and exploration.

References

[1] D. Abramson. Constructing school timetables using simulated annealing: Sequential and parallel algorithms. *Managemente Science*, 37(1):98–113, 1991.

[2] F. Azevedo and P. Barahona. Timetabling in constraint logic programming. In *Proceedings of 2nd World Congress on Expert Systems*, Estoril, Portugal, January 1994.

[3] S. Bistarelli, U. Montanari, and F. Rossi. Soft concurrent constraint programming. In *Proceedings SOFT'01, CP2001, Cyprus*, pages 1–15, 2001.

[4] J.A. Breslaw. A linear programming solution to the faculty assignment problem. *Socio-Economic Planning Science*, 10(1):227–230, 1976.

[5] M.W. Carter. A survey of practical applications of examination timetabling algorithms. *Operations Research*, 34(2):193–202, 1986.

[6] M. Henz and J. Wurtz. Using oz for college timetabling. In *International Conference on the Practice and Theory of Automated Timetabling (PATAT95, Edinbourgh, Scotland, August 1995)*, 1995.

[7] L. Kang and G.M. White. A logic approach to the resolution in constraints in timetabling. *European Journal of Operational Research*, 61:306–317, 1992.

[8] Lars Vestergaard Kragelund. Solving a timetabling problem using hybrid genetic algorithms. *Software Practice and Experience*, 27(10):1121–1134, October 1997.

[9] V. A. Saraswat. *Concurrent Constraint Programming.* The MIT Press, Cambridge, MA, 1993.

[10] Christian Schulte. Comparing trailing and copying for constraint programming. In Danny De Schreye, editor, *Proceedings of the Sixteenth International Conference on Logic Programming*, pages 275–289, Las Cruces, NM, USA, November 1999. The MIT Press.

[11] Peter Van Roy and Seif Haridi. Mozart: A programming system for agent applications. In *International Workshop on Distributed and Internet Programming with Logic and Constraint Languages*, November 1999. Part of International Conference on Logic Programming (ICLP 99).