CONSTRAINT-BASED TIME-TABLING—A CASE STUDY

MARTIN HENZ and JÖRG WüRTZ
Programming Systems Lab, Saarland University, and German Research Center for Artificial Intelligence (DFKI), SaarbrÜcken, Germany

In this article we concentrate on a typical scheduling problem: the computation of a timetable for a German college. Like many other scheduling problems, this problem contains a variety of complex constraints and necessitates special-purpose search strategies. Techniques from operations research and traditional constraint logic programming are not able to express these constraints and search strategies on a sufficiently high level of abstraction. We show that the higher order concurrent constraint language Oz provides this high-level expressivity, and can serve as a useful programming tool for college time-tabling.

Constraint logic programming over finite domains is a rapidly growing research area aimed at the solution of large combinatorial problems. For many real-world problems the constraint logic programming approach (extended to the concurrent constraint approach) is competitive or better than traditional operations research (OR) algorithms. OR techniques lack flexibility, and the effort to achieve customized solutions is often unaffordable. Constraint logic programming combines the flexibility of the approaches in artificial intelligence with the efficiency known from special-purpose algorithms in OR.

The power of constraint logic programming has been proven by languages such as CHIP (Dincbas et al., 1988), Prolog III (Colmerauer, 1990), or CLP(R) (Jaffar & Michaylov, 1987). To solve real-world problems, several new constraints were added as primitives (like *atmost* or the *cumulative* constraint (Aggoun & Beldiceanu, 1993) for scheduling or placement problems importing experience from OR). This approach might be viable for well-known problems but it is not going to foster the exploration of new areas of applications. More clarity and flexibility for the programmer were achieved by clp(FD) (Diaz & Codognet, 1993) inspired by cc(FD) (Van Hentenryck et al., 1991). This approach is based on a single primitive constraint (called indexical) with which more complicated constraints may be defined. While

We thank the Prorektor of the Catholic College for Social Work at Saarbrücken, Peter Huberich, for explaining his timetabling problems to us. We thank Tobias Müller for implementing the constraint solver and Benjamin Lorenz for maintaining the timetable program. Martin Müller and Christian Schulte contributed comments on a draft of the article. The research reported in this article has been supported by the Bundesminister für Bildung, Wissenschaft, Forschung und Technologie (FTZ-ITW-9105), the Esprit Project ACCLAIM (PE 7195), and the Esprit Working Group CCL (EP 6028).

The DFKI Oz system and the documentation are available from the programming systems lab of DFKI through anonymous ftp from ps-ftp.dfki.uni-sb.de or through WWW from http://ps-www.dfki.uni-sb.de/oz/.

Address correspondence to Martin Henz, Programming Systems Lab, Saarland University, P.O. Box 15 11 50, 66041 Saarbrücken, Germany. E-mail: henz@ps.uni-sb.de

in terms of efficiency, clp(FD) is competitive with CHIP, for certain benchmarks, it adds some flexibility. However, still missing is one characteristic that we consider essential for solving certain constraint problems: the flexibility to exploratively invent new constraints and search strategies. While experimental languages such as cc(FD) (Van Hentenryck et al., 1995) and AKL(FD) (Carlson et al., 1994) provide flexibility in formulating constraints, their search strategies are still fixed.

Oz (Smolka, 1995b; Schulte & Smolka, 1994; Schulte et al., 1994; Smolka & Treinen, 1995) is a concurrent constraint language providing for functional, object-oriented, and constraint programming. It is based on a simple yet powerful computation model (Smolka, 1995a), which can be seen as an extension of the concurrent constraint model (Saraswat & Rinard, 1990).

In this article we describe how the unique features of Oz contribute to computing the timetable of a German college that we describe in the next section. The problem contains a combination of complex constraints preventing the application of more standard timetabling techniques.

What can Oz offer to solve this problem?

Constraint programming. The concurrent constraint language Oz allows us to restrict the possible values of variables to finite sets of integers. In the section, Constraints and Propagators in Oz, we introduce some basics about concurrent constraint programming. Crucial for constraint programming is the ability to add constraints on variables concurrently and incrementally. In Oz, this is done by propagators, which can express several constraints of our timetable problem.

Reified constraints. A general scheme, called reified constraints, allows us to express the remaining constraints of our timetable problem, as discussed in the section Reified Constraints. Reified constraints reflect the fact that a constraint holds into a 0/1-valued variable.

Constructive disjunction. Disjunctive constraints (like resource or capacity constraints) can be used constructively in Oz, i.e., information common to different branches can be lifted out for active pruning (Van Hentenryck et al., 1995). The section Constructive Disjunction describes this technique in detail and shows how it can be used for overlap constraints in our timetabling problem.

Flexible enumeration. In Oz the programmer can invent customized search strategies for solving the timetabling problem and optimizing the solutions found. In the section, Enumeration, we develop a search strategy especially adapted to our problem, combining the first-fail heuristic for variable selection with a priority scheme for value selection. This strategy results in an efficiently computed first solution approximating an optimality criteria for the distribution of the college courses.

Optimization. This first solution can be further optimized using a branch-and-bound technique. Optimization can be achieved through an incremental process, allowing the user to inspect the current solution any time and to interrupt and resume the optimization at will, as described in the section Optimization.

Interoperability. The interoperability libraries of Oz allow convenient programming of graphical user interfaces, including the visualization of the computed time-tables, as presented in the section, Implementation Issues.

In the final section we compare the described techniques with other approaches in constraint logic programming. Owing to space limitations, we cannot further detail other aspects of Oz like conditionals and disjunctions, which have been shown to be useful for other constraint problems. The interested reader is referred to the documentation of the Oz system (see footnote number two on page 439).

PROBLEM

Our goal was to find a weekly timetable for the Catholic College for Social Work in Saarbrücken, Germany, in the spring semester 1995. The school offers a 4-year program for a degree in social work. Some courses are mandatory for students of a certain year, while others are optional and open to all students. There are 91 courses, 34 instructors, and 7 rooms of varying size. The assignment of instructors to courses is fixed. Each course needs to take place in a room of sufficient size. There are five school days, and the courses are held between 8:15 am and 5 pm. They may start every quarter of an hour. There are short courses of 3/4 of an hour and long courses of 1-1/2 hours. There must be a break of at least 1/4 hour after a short course and of at least 1/2 hour after a long course.

In the following we state the different constraints that a schedule must fulfill.

C1. Some courses are limited to certain time slots.
C2. Some instructors have times of unavailability.
C3. There are different lunch breaks for the different years in the program.
C4. Some courses must be held after others.
C5. There are sets of courses whose members must be held in parallel.
C6. To every course, a room of sufficient size must be assigned.
C7. An instructor can only teach one course at a time.
C8. Two instructors want to take turns in caring for an infant child and therefore cannot teach at the same time.
C9. The mandatory courses of each year must not overlap.
C10. Some optional courses must not overlap with courses of the first 2 years, and others not with courses of any year.
C11. Two mandatory courses a year may overlap if they are split into groups.
C12. The instructors do not want to teach on more than 3 days a week.
C13. All members of some sets of courses must be held on different days.

The schedule should obey the following criteria as closely as possible. The courses of the first 2 years should be grouped around Monday, Tuesday, and Wednesday. If this is not possible, they should be scheduled on Thursday or Friday morning. The third year courses should be placed on Wednesday, and the fourth preferably on Tuesday and Thursday. The number of courses after the lunch break and on Thursday and Friday should be minimized. (In Germany, several holidays in the spring semester fall on Thursdays.)

CONSTRAINTS AND PROPAGATORS IN OZ

Our goal is to assign to every course a starting time and a room. Note that the assignment of instructors to courses is fixed in our college. Here, we concentrate on the starting time and show in the following section how the room assignment is handled.

The courses are held between 8:15 am and 5 pm on five school days and may start every quarter of an hour. Thus, there are $36 \times 5 = 180$ possible starting times for each course. Since there are 91 courses, the overall search space contains 180^{91} elements. Instead of enumerating the whole search space and testing whether a valuation satisfies all the constraints (generate and test), the idea of constraint programming is to restrict the search space a priori through constraints. While the search space is being explored, more information on the start times becomes known, i.e., the search space can be further pruned, while it is being explored. A programming language for constraint programming needs to provide flexible means to express pruning operators. In this and the following section, we concentrate on pruning operators, while the exploration of the search space is described in the sections, Enumeration and Optimization.

We represent a course, say, Psychology 101, by grouping together its start time, duration, instructor, room size, and name in a record of the form

```
Psych101 = course(start:_ dur:6 instructor: 'Smith' roomsize:big name: 'Psychology
                  101')
```

The underscore _ indicates that no information on the start value is known. The start time of a course is represented by an integer denoting the corresponding quarter of the school week. Because initially it is known that the course must start between quarter 1 and quarter 180, we can add the constraint

```
Psych101.start :: 1#180
```

expressing that this course can take values between 1 and 180, i.e., Psych101.start $\in \{1, \ldots, 180\}$ in a more mathematical notation. We say that the start time is constrained to a finite domain.

Such constraints are stored in a constraint store. For the constraints residing in the constraint store, Oz provides efficient algorithms to decide satisfiability. The largest set of integers satisfying all the constraints for a variable in the store is called the domain of that variable. To distinguish the constraints in the store from more complex constraints, we often call them basic constraints.

The idea of constraint programming is to install constraints that further limit the possible start value for every course. For example, a constraint of type C1 may say that the course Psychology 101 must be held on Monday morning or on Tuesday morning. This constraint is imposed on the constraint store by the expression Psych101.start :: [1#18 37#54] limiting the start value in the constraint store to satisfy Psych101.start $\in$ {1, . . ., 18, 37, . . ., 54}. (The term [1#18 37#54] denotes a list consisting of the two pairs 1#18 and 37#54.) Besides C1, the constraints C2 and C3 can be expressed by such constraints.

For more complex arithmetic constraints, it is known that deciding their satisfiability is not computationally tractable (there are several NP-complete problems on finite domains, e.g., graph coloring). Thus such constraints are not contained in the constraint store but are modeled by installing so-called propagators inspecting the constraint store, as depicted in Figure 1.

A propagator inspects the store, and when values are ruled out from the domain of a variable, it may add more information to the store, i.e., it may amplify the store by adding more basic constraints. Thus we are replacing global consistency, which is assured for the constraints in the store, by local consistency, where unsatisfiability may not be detected. It is important to allow propagators to act concurrently, since it is not statically known when they will be able to perform their computation. This is one major motivation behind concurrent constraint programming.

As an example, consider a constraint of type C4, which states that the course Sociology 101 must be held after our course Psychology 101. It can be expressed by installing the propagator

Psych101.start + Psych101.dur =<: Socio101.start

For example, if Psych101.start is constrained as above to Monday morning or Tuesday morning, and Psych101.dur to quarter 6 of the school week, then the propagator will add the basic constraint

Socio101.start :: 7#180

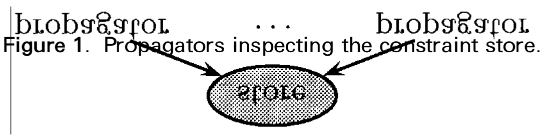
Figure 1. Propagators inspecting the constraint store.

to the constraint store, and vice versa. If later on it becomes known that Socio101 starts the latest at 10:30 am on Monday (Socio101.start $\in \{7, \ldots, 10\}$), Psych101.start will be constrained to start the latest at 9:00 am (Psych101.start $\in \{1, \ldots, 4\}$). Note that the propagator remains active, waiting for more information on either Psych101.start or Socio101.start.

Observe that in the implementation we have to add the necessary break after Psych101 (depending on Psych101.dur) but we omit the breaks in this presentation for simplicity.

The constraint C5 is modeled by using the propagator =: expressing equality. Note that S=:T is modeled by S=<:T S>=:T. Thus, holes in the domains of S and T as in Psych101.start are not considered for =:. It implements only partial arc-consistency instead of full arc-consistency, in the terminology of Macworth (1977). Interval consistency can be implemented efficiently, since the propagator only needs to watch the currently smallest and biggest possible values for the involved variables.

REIFIED CONSTRAINTS

We have seen in the previous section that propagators are crucial components in a constraint programming system, since they allow us to prune the search space. A constraint programming language therefore must strive to easily express many kinds of propagators. In this section, we introduce reified constraints as a generic tool to express new propagators and show how the remaining constraints C6–C13 of our college problem can be expressed with them.

Constraint C6 says that for every point in time the number of courses of a certain size must not exceed the number of rooms of that size. Assume that there are NumberOfRooms different rooms available of a given size. If we are able to compute for every quarter of an hour Q of the teaching week the number of courses CourseAtQ of the given size being held in this quarter, then we only would need to install the propagator

CoursesAtQ =<: NumberOfRooms

to express C6 for every quarter Q and every room size. To compute CoursesAtQ, it would be convenient to be able to constrain a boolean variable CAtQ to 1 if a given course overlaps Q and to 0 if it does not (and vice versa). Then CoursesAtQ can be obtained simply by computing the sum of all CAtQ over all courses of the given size. To compute CAtQ, we use reified constraints, i.e., propagators that reflect the validity of a constraint into a variable. Reified constraints are also known in the literature as nested constraints (Benhamou & Older, 1992; Sidebottom, 1993). The constraint whose validity we want to reflect has the form

Course.start :: Q-Course.dur+1 # Q

expressing that Course has started before or at quarter Q, but not finished at Q.

Now we reflect the validity of this constraint into the variable CAtQ:

CAtQ = Course.start :: Q-Course.dur+1 # Q

First, every reified constraint always constrains the first variable to be either 0 or 1, i.e., CAtQ $\in \{0,1\}$. As in previous propagators, information flows either way. If the store logically implies Course.start $\in$ {Q-Course.dur+1, . . ., Q}, CAtQ is constrained to 1. If the store implies Course.start $\notin$ {Q-Course.dur+1, . . ., Q}, CAtQ is constrained to 0. If CAtQ is constrained to 1 or 0, the basic constraint Course.start :: Q-Course.dur+1 # Q or its negation Course.start \:: Q-Course.dur+1 # Q is imposed on the store. It is essential that while CAtQ is not determined to 0 or 1, the constraint on the right-hand side is used only for checking but not for pruning. If already NumberOfRooms courses are scheduled at Q, the remaining boolean variables CAtQ will be constrained to 0 by the propagator CoursesAtQ =<:NumberOfRooms.

Because we guarantee that at each time there are sufficiently many rooms available, it is straightforward to assign the possible rooms to courses. In particular, the room assignment can be performed after the timetable computation, considerably reducing the complexity of the problem.

Constraints C7–C11 express overlapping conditions on courses. Assume that Psych101 and Socio101 are mandatory courses for first-year students. Then C9 says that they may not overlap. This constraint can be expressed with a disjunction of the following form:

$$\begin{aligned} &\text{Psych101.start} + \text{Psych101.dur} \leq \text{Socio101.start} \\ \vee\ &\text{Socio101.start} + \text{Socio101.dur} \leq \text{Psych101.start} \end{aligned}$$

If we are able to install a propagator stating that at least one of a given set of complex constraints is valid, we can express this disjunction. Thus our problem is solved by reifying complex constraints:

```
   (Psych101.start + Psych101.dur =<: Socio101.start)
+  (Socio101.start + Socio101.dur =<: Psych101.start) >=: 1        (1)
```

which is an abbreviation for

```
B1 = Psych101.start + Psych101.dur =<: Socio101.start
B2 = Socio101.start + Socio101.dur =<: Psych101.start
B1 + B2 >=: 1
```

Let us consider the first reified constraint. If Psych101.start + Psych101.dur $\leq$ Socio101.start is logically implied by the constraint store, the basic constraint B1::1

is imposed on the store. If Psych101.start + Psych101.dur > Socio101.start is logically implied by the constraint store, the basic constraint B1::0 is imposed on the store. If B1 is constrained to 1, the propagator Psych101.start + Psych101.dur =<: Socio101.start is installed, and if B1 is constrained to 0, the propagator Psych101.start + Psych101.dur >: Socio101.start is installed.

As an example, consider now Psych101.start ∈ {8, . . ., 12}, Psych101.dur = 6, Socio101.start ∈ {10, . . ., 14} and Socio101.dur = 6. The second reified constraint constrains B2 to 0 because the constraint store implies Socio101.start + Socio101.dur > Psych101.start. The constraint B2 ::0 in the store allows the propagator B1 + B2 >=:1 to constrain B1 to 1. This allows the first reified propagator to install the propagator Psych101.start + 6 =<:Socio101.start, constraining Psych101.start to 8 and Socio101.start to 14, the only possible values, if the two courses do not overlap.

Encoding the constraints C7–C10 now becomes straightforward. For example, C7 says that no two courses of an instructor must overlap. Thus for every pair of courses of an instructor, a propagator of the above form must be installed.

The constraint C11 boils down to the constraint that a certain course, say, SplitCourse, may overlap with at most one of a list OtherSplitCourses of other courses. If we are able to install a propagator that constrains a variable Overlap to 1, if a course overlaps with the course SplitCourse, and to 0, if it does not, we can build the sum Sum of these variables over OtherSplitCourses. Then we only need to impose the constraint Sum =<:1, stating that at most one of OtherSplitCourses overlaps with SplitCourse.

So how can we constrain the Overlap variables? Two courses, SplitCourse and OtherSplit, overlap if OtherSplit starts before SplitCourse is finished and vice versa, i.e., if both the constraints hold: SplitCourse.start + SplitCourse.dur >: OtherSplit.start and OtherSplit.start + OtherSplit.dur >: SplitCourse.start. The variable Overlap must be constrained to 1 if these two constraints hold, and to 0 otherwise:

$$\begin{aligned}\text{Overlap} = {}&(\text{SplitCourse.start} + \text{SplitCourse.dur} >: \text{OtherSplit.start})\\ &+ (\text{OtherSplit.start} + \text{OtherSplit.dur} >: \text{SplitCourse.start}) =: 2\end{aligned} \tag{2}$$

which expands to

$$\begin{aligned}&\text{B1} = \text{SplitCourse.start} + \text{SplitCourse.dur} >: \text{OtherSplit.start}\\ &\text{B2} = \text{OtherSplit.start} + \text{OtherSplit.dur} >: \text{SplitCourse.start}\\ &\text{Overlap} = \text{B1} + \text{B2} =: 2\end{aligned}$$

As usual, this constraint also works the other way around, e.g., if Overlap is known to be 1, then B1 + B2 =:2 is installed. Thus B1 and B2 are constrained to 1, and hence both propagators are installed. If Overlap is known to be 0, for example, because another Overlap variable in Sum is already 1, the propagator B1 + B2 \=: 2 is installed, stating that only one of B1 and B2 may be constrained to 1. Thus, if for example, B2 is constrained to 1, then B1 is constrained to 0 and the reified constraint for B1 installs the nonreified version of its negation: SplitCourse.start + SplitCourse.dur =<: OtherSplit.start.

In a similar way, the constraint C12 can be expressed. Assume that all courses taught by a given instructor are contained in the list Courses. For every instructor and every day, we compute the boolean value TeachesOnDay with

TeachesOnDay = 1 =<: SumOfCoursesOnDay

where SumOfCoursesOnDay is the sum of the boolean variables obtained by reifying for every element of Courses a constraint that states that the course is taught on that day. For every instructor, we can express that she only teaches on 3 days with the propagator

TeachesOnDays =<: 3

where TeachesOnDays is the sum of all TeachesOnDay variables over the week. For example, if three courses have already been placed on different days, say, Monday through Wednesday, then all the remaining courses are constrained to be held on Monday through Wednesday, thus reducing the search space considerably.

The same technique can be applied for our last constraint C13. Assume that all elements of the list DifferentDayCourses with length NumberOfDifferentDayCourses must be held on a different day. Then the propagator

TeachesOnDays =: NumberOfDifferentDayCourses

does the job, where TeachesOnDays is defined as for C12.

CONSTRUCTIVE DISJUNCTION

In this section we reconsider how to model the overlapping of two courses. Let us assume that the starting time of our two courses Psych101 and Socio101 is between (and including) 8:15 am (quarter 1) and 10:30 am (quarter 10), i.e., Psych101.start, Socio101.start $\in \{1, \ldots, 10\}$, and both durations are 6 quarters. Then the nonoverlapping constraint of the courses expresses the disjunction

$$\text{Psych101.start} + 6 \leq \text{Socio101.start} \vee \text{Socio101.start} + 6 \leq \text{Psych101.start}$$

The left alternative of the disjunction constrains Psych101.start to $\{1, \ldots, 4\}$, i.e., Psych101 must start before or at 9 am. Analogously, the right alternative constrains

Psych101.start to {7, . . ., 10}, i.e., it must start after or at 9:45 pm. Thus, independent of which alternative will succeed, we know that Psych101 cannot start at 9:15 am (quarter 5) or at 9:30 am (quarter 6), i.e., Psych101 $\in$ {1, . . ., 4, 7, . . ., 10}.

The propagators in the previous section, however, do not extract this valuable information on Psych101. To obtain more pruning, there is a more active form of disjunction available in Oz, called constructive disjunction (Van Hentenryck et al., 1995). It extracts the common information from the alternatives of a disjunction. We replace program Eq. (1) by constructive disjunction, supported in the following syntax:

```
dis Psych101.start + Psych101.dur =<: Socio101.start
[] Socio101.start + Socio101.dur =<: Psych101.start
end
```

As in Eq. (1), if one alternative is unsatisfiable, the propagator corresponding to the other alternative is installed. Additionally, common information is extracted as described above. While the pruning is enhanced by constructive disjunction, it is also potentially more expensive, since extraction of common information may be attempted relatively often. Thus it takes some experimentation to find out which form of disjunction is most appropriate for a given application. It is essential that propagators also take holes in the domains into account because in our problem, constructive disjunction cuts holes in domains of variables. This is the case of reified basic constraints like B = X::9#10. If, for example, the basic constraint X::[1#8 11#15] is added, B is constrained to 0. The use of constructive disjunction for all nonoverlap constraints (C7–C11) in our college problem resulted in a speedup of more than 1 order of magnitude.

For modeling constraint C12, we use a ternary constructive disjunction instead of program Eq. (2):

```
dis B =: 1 SplitCourse.start + SplitCourse.dur >: OtherSplit.start
       OtherSplit.start + OtherSplit.dur >: SplitCourse.start
[] B =: 0 SplitCourse.start + SplitCourse.dur =<: OtherSplit.start
[] B =: 0 OtherSplit.start + OtherSplit.dur =<: SplitCourse.start
end
```

The common information is in this case extracted from all three alternatives (or two if one alternative is known to be unsatisfiable). If all but one alternative is known to be unsatisfiable, the propagators of the remaining alternative are installed (since the disjunction must be true).

ENUMERATION

To achieve maximal pruning of the search space, we allow the propagators to exhaustively amplify the constraint store. We call a store, together with all the propagators, *stable* if none of the propagators can add any more information to it. Typically, many variables still have more than one possible value after stability of the store. Thus we want to explore the remaining search space. We proceed in two steps. First, we compute a fairly good first solution as described in this section, and then we optimize starting from this solution as described in the following section.

To explore the remaining search space, one of the variables that has more than one possible value is selected and speculatively constrained to these values. In order to speculatively constrain a variable to a value, we impose this constraint on a copy of the current constraint store, including all the propagators. If, later on, the computation fails, another value can be tried on another copy of the store. (Instead of copying the store, as is done in Oz, one can also trail the previous domain of the variable and restore it on failure.) We call this process *enumeration* (in the literature it is also known as labeling). Once a variable is speculatively constrained to an integer, some propagators typically become able to amplify the constraint store again. When the constraint store becomes stable again, the next variable is selected for enumeration, and so on. Thus propagators allow us to prune the search space while it is being explored. This scenario makes clear why a sequential language is inappropriate for describing constraint problems. In a sequential language the complex interaction between enumeration and propagation needs to be made explicit by the programmer, while concurrent constraint languages allow us to conceptually separate propagation and enumeration. Sequential languages like ECLiPSc (European Computer Science Research Center, 1994) deal with that problem by introducing ad hoc concepts like freeze and demons.

The enumeration process has two degrees of freedom. First, the variable to be enumerated next needs to be selected, and second, the order in which the remaining possible values are tried needs to be fixed. It is an essential ingredient of Oz that both variable and value selection can be programmed in Oz, achieving a high degree of flexibility. For variable selection, we apply the first-fail strategy, in which a variable with the currently smallest domain is selected.

Using the first-fail strategy, with the usual value selection beginning with the smallest possible value, does not lead to a solution of our problem after 1 day of computation on a Sun Sparc 20. (The usual first-fail strategy performs poorly because this strategy tries to place courses in a compact timetable. Owing to the topology of our search space, this results in a behavior we call “thrashing” when constraints are violated because the enumeration strategy is not “clever” enough to find the responsible variables. An approach using intelligent backtracking could help here (Bruynooghe & Pereira, 1984).) The so-called first-fit strategy, which tries to place a course in the day with the fewest already placed courses as described by Boizumault et al., 1994), also does not lead to a solution in reasonable time. Instead, we enumerate the courses of each year starting from a different time of the week. The domain from 1 through 180 is divided in 10 blocks, representing mornings and afternoons of school days. These blocks can be individually ordered for each course (in the implementation, we use the same ordering for each year). By carefully choosing these blocks, we can come to a first solution very fast.

We can make use of this additional flexibility to optimize the timetable according to the criteria in the problem section above. We simply order the blocks such that the preferred times are tried earlier than others. The first solution is now more likely to be better with respect to the criteria.

OPTIMIZATION

In the previous section, we have seen how we approach the optimization criteria by choosing a suitable enumeration strategy. However, experimentation shows that enumeration alone cannot guarantee that the first solution will fulfill the criteria sufficiently well. A way to achieve optimal solutions in constraint logic programming is the use of branch-and-bound. Branch-and-bound starts out with one solution and imposes that every next solution must be better than the previous one using a suitably defined cost function.

Soft constraints, i.e., constraints that should hold but might be dropped, if necessary, can be modeled by including their reified version in the cost function. As an example, consider that Course should be scheduled on Monday. The cost function will use the result B of the reified constraint B = CourseNew::Monday >: CourseOld::Monday.

As for many constraint problems in the real world, in time tabling there rarely exists a unique cost function to optimize. For example, the goals to achieve compact timetables for students and instructors may contradict each other. Hence we have chosen general criteria that students and instructors can agree upon, such as minimizing the number of courses being held after the lunch break.

After the first (fairly good) solution is found with customized first-fail, we use branch-and-bound search to further optimize starting from this solution. Since, owing to the topology of our search space, going for the globally best solution is not feasible, the user can define a limit on the number of enumeration steps leading to a resource-limited branch-and-bound search. The user can interrupt the optimization at any time and request the currently best solution. This so-far best solution can be inspected, and the user can decide whether it is good enough or whether to search for a better solution. This process can be continued arbitrarily often. Thus we can say that we implement an anytime algorithm.

IMPLEMENTATION ISSUES

Figure 2 shows the top-level graphical user interface to our timetabling program. By using the object-oriented features of Oz and the interface to the window programming toolkit Tcl/Tk (Mehl, 1994), it was straightforward to implement the

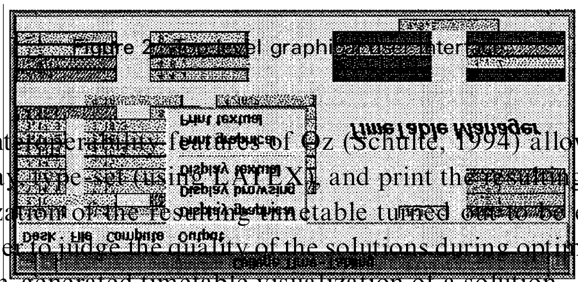

Figure 2. Top-level graphical user interface.

interface. The interoperability features of Oz (Schulte, 1994) allow the integration of tools to display, type-set (using LATEX), and print the resulting timetable.

The visualization of the resulting timetable turned out to be extremely useful for the human user to judge the quality of the solutions during optimization. Figure 3 shows a program-generated timetable visualization of a solution.

Other features of the language Oz, such as statically scoped higher order programming and concurrent object-oriented programming, which are not available in other constraint logic languages, vastly facilitate coding and maintenance of programs.

The program deals with more than 25,000 propagators. The first solution is found in less than 1 min on a Sun Sparc 20 with 60 MHz. A considerable optimization of 5 lectures less at afternoons is obtained after a further 10 min.

COMPARISON WITH CONSTRAINT LANGUAGES

In this section we briefly compare Oz with existing constraint systems for solving combinatorial problems.

CHIP (Dincbas et al., 1988) is the forerunner for the most commercial constraint systems. It lacks flexibility for search strategies and constructive disjunction. Only a small set of predefined reified constraints is supported by an if-then-else construct. Nevertheless, it is a successful commercial tool because several OR techniques have been incorporated in operators dealing with disjunctive information. Disjunctive constraints not appropriate for these operators must be modeled passively by choice-points.

ECLiPSe (ECRC, 1994) is the research successor of CHIP enriched with features like attributed variables. By using attributed variables, it is possible to program user-defined constraints on a rather low level.

The system clp(FD) (Diaz & Codognet, 1993) is a constraint language compiling into C-code. It is based on indexicals (Van Hentenryck et al., 1991) and is the fastest finite domain system freely available. Nevertheless it

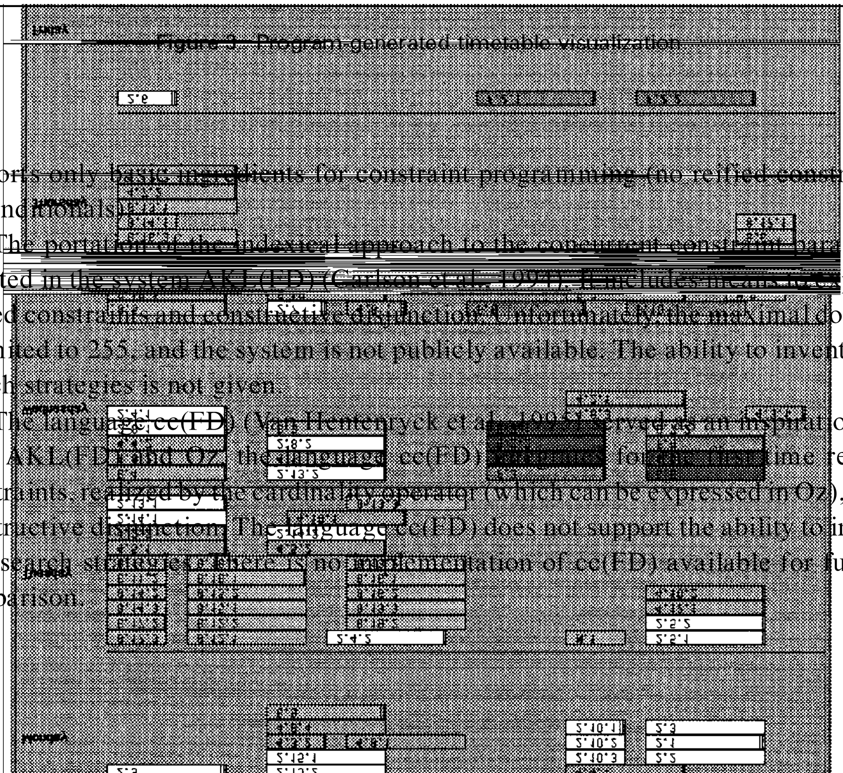

Figure 3 Program-generated timetable visualization

supports only basic ingredients for constraint programming (no reified constraints or conditionals).

The portation of the indexical approach to the concurrent constraint paradigm resulted in the system AKL(FD) (Carlson et al. 1994). It includes means to express reified constraints and constructive disjunction. Unfortunately, the maximal domain is limited to 255, and the system is not publicly available. The ability to invent new search strategies is not given.

The language cc(FD) (Van Hentenryck et al. 1993) served as an inspiration for both AKL(FD) and Oz; the language cc(FD) introduces for the first time reified constraints, realized by the cardinality operator (which can be expressed in Oz), with constructive disjunction. The language cc(FD) does not support the ability to invent new search strategies. There is no implementation of cc(FD) available for further comparison.

REFERENCES

Aggoun, A., and N. Beldiceanu. 1993. Extending CHIP in order to solve complex scheduling and placement problems. *Mathematical and Computer Modelling* 17(7):57–73.

Boizumault, P., C. Gueret, and N. Jussien. 1994. Efficient labeling and constraint relaxation for solving time tabling problems. Technical Report European Computer Science Research Center (ECRC)-94-38, Munich, Germany. ECRC.

Benhamou, F., and W. J. Older. 1992. Applying interval arithmetic to integer and boolean constraints. Technical report, Bell Northern Research, June 1992.

Bruynooghe, M., and L. M. Pereira. 1984. Deduction revision by intelligent backtracking. In J. A. Campbell (ed.), *Implementations of PROLOG.* Chichester, England: Ellis Horwood Limited.

Carlson, B., M. Carlsson, and D. Diaz. 1994. Entailment of finite domain constraints. In P. van Hentenryck (ed.), *Proceedings of the International Conference on Logic Programming,* 339–353. Cambridge, Mass.: MIT Press.

Colmerauer, A. 1990. An introduction to PROLOG III. *Communications of the ACM* July:70–90.

Diaz, D., and P. Codognet. 1993. A minimal extension of the WAM for clp(FD). In *Proceedings of the International Conference on Logic Programming,* 774–790. Cambridge, Mass.: MIT Press.

Dincbas, M., P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. 1988. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems FGCS-88,* 693–702. Tokyo, Japan.

ECRC. 1994. *ECL PS, User Manual Version 3.4.1,* July 1994.

Jaffar, J., and S. Michaylov. 1987. Methodology and implementation of a CLP system. In *Proceedings of the International Conference on Logic Programming,* 196–218.

Mackworth, A. K. 1977. Consistency in networks of relations. *Artificial Intelligence* 8:99–118.

Mehl, M. 1994. Window programming in DFKI Oz. DFKI Oz documentation series, German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany.

Schulte, C. 1994. Open programming in DFKI Oz. DFKI Oz documentation series, German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany.

Sidebottom, G. A. 1993. A language for optimizing constraint propagation. Ph.D. thesis, Simon Fraser University, Burnaby, B.C., Canada.

Smolka, G. 1995a. The definition of Kernel Oz. In A. Podelski (ed.), *Constraints: Basics and trends.* Lecture Notes in Computer Science, vol. 910, 251–292. New York: Springer-Verlag.

Smolka, G. 1995b. The Oz programming model. In Jan van Leeuwen (ed.), *Computer science today.* Lecture Notes in Computer Science, vol. 1000, 324–343. New York: Springer-Verlag, in press.

Saraswat, V. A., and M. Rinard. 1990. Concurrent constraint programming. In *Proceedings of the 7th Annual ACM Symposium on Principles of Programming Languages,* 232–245. San Francisco, Calif.

Schulte, C., and G. Smolka. 1994. Encapsulated search in higher-order concurrent constraint programming. In Maurice Bruynooghe (ed.), *Logic programming: Proceedings of the 1994 International Symposium,* 505–520. Cambridge, Mass.: MIT Press.

Schulte, C., G. Smolka, and J. Würtz. 1994. Encapsulated search and constraint programming in Oz. In A. H. Borning (ed.), *Second workshop on principles and practice of constraint programming.* Lecture Notes in Computer Science, vol. 874, 134–150. New York: Springer-Verlag.

Smolka, G., and R. Treinen (eds.), 1995. DFKI Oz Documentation Series. German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany.

Van Hentenryck, P., V. Saraswat, and Y. Deville. 1991. Constraint processing in cc(FD). Technical report, Brown University, Providence, R.I.

Van Hentenryck, P., V. Saraswat, and Y. Deville. 1995. Design, implementation and evaluation of the constraint language cc(FD). In A. Podelski (ed.), *Constraints: Basics and trends.* Lecture Notes in Computer Science, vol. 910, 293–316. New York: Springer-Verlag.