Oz Scheduler:
A Workbench for Scheduling Problems

Jörg Würtz
Programming Systems Lab, DFKI GmbH,
University of the Saarland, Geb. 45, 66041 Saarbrücken, Germany,
Email: wuertz@ps.uni-sb.de

Abstract

This paper describes the Oz Scheduler, a workbench for scheduling problems. Through a graphical interface, the user can freely combine the elements that define a scheduling strategy. Such elements include constraints with different popagation behavior or distribution and search strategies. Exploring the possible combinations can lead to better solutions. Recent and successful techniques for scheduling are incorporated.

Resulting from the selections made, a constraint problem is generated dynamically. For this problem, the solution, statistics and the search can be inspected by several graphical tools. The functionality can be extended by sending messages to the Scheduler. The functionality and the implementation of the Oz Scheduler are discussed. The overall performance of the Scheduler for standard benchmarks is comparable to state-of-the-art special-purpose systems for scheduling. The implementation is based on the concurrent constraint language Oz.

1. Introduction

Scheduling problems are in the core of many real-world applications. They occur in areas as diverse as production planning, time tabling or personel planning. For certain well defined problem classes there exist efficient programs from Operations Research (OR). But these programs are often very problem specific and slight changes in the problem definition raise difficulties in the adaptation of the special purpose algorithms.

On the other hand, AI aims at developping general problem solving approaches. Unfortunately, these approaches are often much slower than special-purpose algorithms. One of the offsprings of AI is constraint programming [15, 24], which offers flexibility by the formulation of constraints in a high-level language. In the last years, constraint programming and especially Finite Domain Programming, has succeeded in solving real-world problems in various areas. Moreover, efficient OR techniques were successfully integrated into flexible constraint systems to solve scheduling problems, which are considered extremely hard [7, 3, 1].

Solving a scheduling problem is a difficult undertaking. To solve the problem one has to experiment with several subproblems or variations. Different techniques have to be tested like the kind of constraint propagation, ordering heuristics or search strategies. These experiments should be facilitated by an interface, which allows the combination of these parameters and provides tools to observe the experimentation.

This paper describes the Oz Scheduler, a workbench for scheduling problems. Through a graphical interface, the user can freely combine different

- problem specifications.
- constraints with different propagation behavior.
- distribution strategies for ordering the tasks to be scheduled on resources.
- search strategies like branch&bound or binary search (on lower and upper bounds) to find optimal solutions (including the proof of optimality) or to find good solutions quickly. Furthermore, search for iteratively improving lower and upper bounds is provided and can be customized. Search can be interrupted and resumed at any time.
- visualization tools for the solution(s), statistics (either text-based or graphically by so-called Gantt-charts) and visualization of the search tree.

The search is parameterized by the user's selections. The user can add further components like search strategies or constraints dynamically making use of the underlying object-oriented structure. The problem does not have to be stated statically as a constraint program in advance. Instead,

the selected parameters are used to dynamically create a procedure by a so-called scheduling compiler. While the first implementation of the Scheduler deals with one class of scheduling problems only (see Section 3), it can be extended to handle further classes by the dynamic extension to different scheduling compilers (see Section 4.2).

As an implementation language we use Oz [22, 23, 13]. Oz is a concurrent constraint language providing for functional, object-oriented, and constraint programming, which also features programmable search. Thus, Oz appears to be a promising candidate for implementing a workbench like this Scheduler.

Through the use of the constraint interface of Oz, we have incorporated recent OR techniques. This allows us to solve hard scheduling benchmark problems efficiently, comparable to state-of-the-art special-purpose tools for scheduling. This exemplifies the viability of the Oz Scheduler to be a flexible *and* efficient tool. As far as we know, there is no comparable graphical workbench available like the Oz Scheduler. The Scheduler will be freely available in the next release of Oz.

The paper is structured as follows. In Section 2, constraint programming in Oz is introduced. Section 3 explains the essentials of scheduling problems. In Section 4, the functionality and implementation applied in the Oz Scheduler are shown and its performance is evaluated.

2. Constraint Programming in Oz

This paper deals with constraints on finite sets of nonnegative integers, so-called *finite domains*, in the constraint programming language Oz. For a more thorough treatment see [21, 14, 17].

A *basic constraint* takes the form $x = n$, $x = y$ or $x \in D$, where n is a nonnegative integer and D is a finite domain. The basic constraints reside in the *constraint store*. Oz provides efficient algorithms to decide satisfiability and implication for basic constraints.

For more expressive constraints, like $x + y = z$, deciding their satisfiability is not computationally tractable. Such nonbasic constraints are not contained in the constraint store but are imposed by *propagators*. A propagator is a computational agent that tries to narrow the domains of the variables occurring in the corresponding constraint. This narrowing is called *constraint propagation*.

As an example, assume a store containing $X, Y, Z \in \{1, \ldots, 10\}$. The propagator for $X + Y < Z$ narrows the domains to $X, Y \in \{1, \ldots, 8\}$ and $Z \in \{3, \ldots, 10\}$ (since the other values cannot satisfy the constraint). Adding the constraint $Z = 5$ causes the propagator to strengthen the store to $X, Y \in \{1, \ldots, 3\}$ and $Z = 5$. Imposing $X = 3$ lets the propagator narrow the domain of Y to one. A tool for displaying arbitrary data structures and the current domains of variables is the Oz Browser. Changes in the domains cause an update of the display [23].

A solution to a set of finite domain constraints is a mapping from variables to nonnegative integers. To obtain a solution for a set of constraints S we usually have to choose a (not necessarily basic) constraint C and solve both $S \cup \{C\}$ and $S \cup \{\neg C\}$; we *distribute S with C* at the current *choicepoint*. The second alternative $S \cup \{\neg C\}$ is solved if the first alternative leads to an inconsistent store (the recovering of S may be done by backtracking or by using previous copies of the constraint stores, as in Oz). We say that a *failure* has occurred, if constraint propagation leads to an inconsistent store. Note, that distribution takes place only if propagation has reached a fixed point. In the example above we have first distributed with $Z = 5$ and then with $X = 3$. Thus, solving a constraint problem consists in a sequence of interleaving *propagation and distribution steps*. In Oz, distribution strategies like first fail but also more elaborated ones can be programmed by the user [21].

Additionally, Oz offers programmable search [21]. Besides depth-first for one solution or branch&bound, one can program resource limited search (limited number of time or failures) or strategies from AI like limited discrepancy search [12]; both used in the Oz Scheduler. Furthermore, it is possible to encapsulate a constraint store and the connected propagators. Into this capsule, further constraints can be injected, allowing for local search techniques (parts of a former solution are injected into a base problem, while the other parts are subject of a new search run). The search for a solution can be visualized by the Oz Explorer [20] (see also Section 4).

A constraint problem can be solved by the combination of three orthogonal concepts: Propagation, distribution and search. Because in Oz these concepts are completely independent from each other, the implementation of a tool like the Scheduler is so convenient.

Oz comes with a fully developped finite domain system offering reified constraints (reflecting the validity of a constraint into a 0/1 valued variable), symbolic constraints like `atmost`, and generic propagators for arithmetic (like arbitrary scalar products or nonlinear constraints).

3. Scheduling

In this section, the characteristics of one class of *scheduling problems* are shown by means of an example. The problem is to build a bridge with a set of resources like a crane, a concrete mixer etc. [11]. Because only one crane etc. is available, the resources are called *unary*. Activities or *tasks* are, for example, positioning a bearer with the crane, erecting a pillar with the bricklayer and so on. A start time must be assigned to a task such that its resource is available through the whole duration of the task. There

exist *precedence constraints*, like that the pillars must be erected before the corresponding bearer is positioned, and *resource constraints*, stating that the execution of tasks on the same resource must not overlap in time (the tasks on the resource must be *serialized*). Furthermore, there may be *additional constraints* like that the time interval between the completion of two tasks is restricted. The *solution* of a scheduling problem (a *schedule*) consists in an assignment of start times to tasks that is consistent with all constraints. Often, one is interested in the schedule with the smallest length, i.e., an *optimal solution*. For the first implementation of the Oz Scheduler, we consider non-preemptive scheduling problems with unary resources and fixed durations (for extensions see Section 4.2).

Modeling a scheduling problem with finite domain constraints is simple, if the right abstractions are available. A finite domain models the start time of a task. A precedence constraint like that A with duration $d(A)$ must precede B is stated as

$$A + d(A) \leq B.$$

The resource constraints can be modeled by stating the following disjunction for all pairs of tasks A and B on the same resource.

$$A + d(A) \leq B \ \vee \ B + d(B) \leq A,$$

i.e., either A precedes B or vice versa. By *reified constraints* this can be expressed as

$$C_1 = (A + d(A) \leq B) \ \wedge \ C_2 = (B + d(B) \leq A) \\ \wedge \ C_1 + C_2 = 1.$$

Here, the validity of e.g. $A + d(A) \leq B$ is reflected into the 0/1 valued variable C_1. If the constraint (resp. its negation) is implied by the constraint store, C_1 is constrained to 1 (resp. 0). On the other hand, if C_1 is constrained to 1 (resp. 0), the constraint (resp. its negation) is imposed.

Additional constraints may be added by imposing further propagators.

After imposing the constraints, a *distribution strategy* is needed to find a schedule. For scheduling problems, it is often of advantage to serialize the tasks first,i.e., to distribute with constraints $A + d(A) \leq B$, and then to distribute the start times with constraints $A = n$. (For problems where only constraints of the form $A + d(A) \leq B$ occur, the serialization is sufficient.) To obtain the optimal solution one can use a *branch&bound* search strategy, stating that the alternative solution must denote a schedule with smaller length than the best solution found so far. The idea is that imposing a bound allows for further propagation and can strongly prune the search space.

While modeling the scheduling problem in this way is simple, local reasoning on task pairs is insufficient for harder scheduling problems. As an example consider three tasks A, B and C each with duration 8, and possible start times between 1 and 10. Stating for the pairs (A, B), (A, C) and (B, C) that they must not overlap by reified constraints, will lead to no further propagation. On the other hand, the tasks must be scheduled between time point 1 and 18 (the latest completion time of either A, B or C). Because the overall duration is 24, this is impossible.

Hence, stronger propagators were suggested in [5] (in terms of Operations Research, of course), reasoning on the whole set of tasks on a resource and, thus, are called *global constraints*. The principal ideas behind it are simple but very powerful. For an arbitrary set of tasks S to be scheduled on the same resource, the available time must be sufficient (see the example above). Furthermore, one checks whether a task T in S must be scheduled as the first or last task of S (and analogously if T is not in S). Let S' be S without T. Then, T must be first, if it cannot be scheduled after all tasks in S' and not between two tasks in S'. Let $s(T)$, $c(T)$, and $d(T)$ be the earliest possible start time, the latest possible completion time, and the duration of T, respectively. Let $s(S')$, $c(S')$, and $d(S')$ be the earliest possible start time, the latest possible completion time and the sum of durations of tasks in S'. Then, if

$$c(S') - s(S') < d(S),$$

T cannot be between two tasks of S', and if

$$c(T) - s(S') < d(S),$$

T cannot be scheduled after all tasks in S'. Hence, T must be first and corresponding propagators can be imposed, narrowing the start times. Analogously, if

$$c(S') - s(S') < d(S) \ \wedge \ c(S') - s(T) < d(S),$$

T must be last. For this kind of reasoning, the term *edge-finding* was coined in [2]. There are several variations of this idea in [5, 2, 6, 16] for the OR community and in [19, 7] for the constraint community; they differ in the amount of propagation and which sets S are considered for edge-finding (in principle, there are exponentially many). The resulting propagators do a lot of propagation, but are also more expensive than e.g. reified constraints. Depending on the problem, one has to choose the appropriated propagator.

For distribution, a similar idea is used, i.e., the sets of tasks are computed, which can be scheduled first or last. From these sets, one task is chosen to be the first (or the last) and it is distributed with the corresponding constraints.

Unfortunately, scheduling problems are very hard and a simple branch&bound search started from scratch often converges too slowly to the optimal solution. Thus, the problem can be divided into two. One problem is to find successively increasing lower bounds LB such that it is proved that no

schedule with length LB or smaller exists (here, the used search strategy must be *complete*). The second problem is to find successively decreasing upper bounds UB of the optimal schedule length (and a corresponding schedule). Here, an *incomplete* strategy like local search is sufficient, because one is interested in finding rather good solutions quickly. The optimal solution is between LB and UB. If the search for an upper bound does not improve anymore, one may stop or switch to a complete branch&bound search to find the optimal solution and prove its optimality.

A famous example of a scheduling problem (a so-called job-shop problem[1]) is MT10 [18], which consists of 10 resources à 10 tasks. Stated 1963, it was not until 1989 that the optimal solution was found and optimality proved with the technique mentioned above [5]. The naive modeling cannot solve the problem in several days. In the mid 90's, it was also solved by the constraint community by adapting the ideas of [5] (see e.g. [7, 3]). The results on MT10 and other problems for the Oz Scheduler are given in Section 4.3.

4. The Oz Scheduler

This section explains the Oz Scheduler. Section 4.1 introduces the Scheduler by means of an example. In Section 4.2, the implementation of the Scheduler is described. The paper concludes with a performance evaluation.

4.1. An Example

Figure 1 shows the Oz Scheduler. From the main menu `Compiler Spec`, we select the bridge problem (see Section 3) from a submenu and use the default settings branch&bound search (others are available from the menu `Search Spec`), reified constraints for the resource constraints and a rather naive distribution strategy. The search is started by selecting a submenu from `Run`.

We inspect the computation statistics by chosing an appropriate submenu from the menu `Display`. By a pop-up window (see Figure 2) we are informed about the name of the problem and some of the used parameters. Furthermore, the computational resources like time (divided into subgroups) and the number of failures and solutions are displayed (for details see [23]). We observe that three solutions were found and, thus, the optimal solution was the third one. Hence, we choose another distribution strategy for experimentation. This time, we want to obtain more information about the search tree and select the Oz Explorer from the menu `Run`. The Explorer is invoked and the resulting search tree is shown in Figure 3. By clicking on the search nodes

[1] An nxm job-shop problem consists of n jobs and m resources. Each job consists of m tasks to be scheduled on different resources. The tasks in a job are linearly ordered by precedence constraints. The resources are unary and no preemption is allowed.

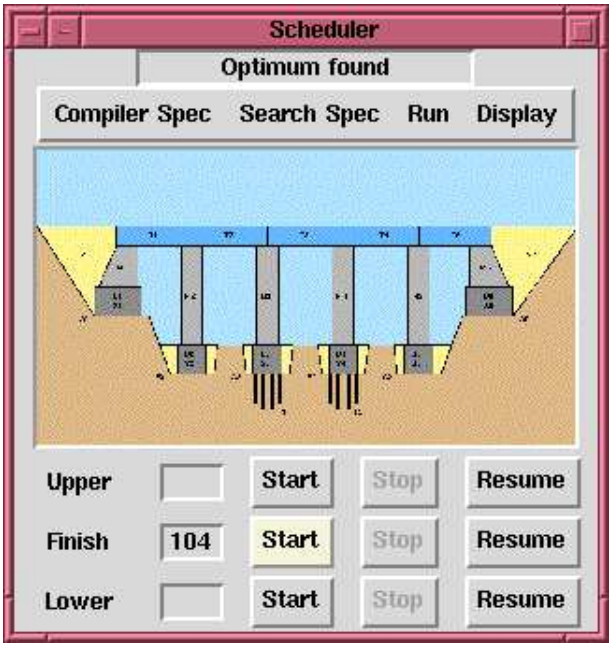

Figure 1. The Oz Scheduler

in the Explorer, the domains of the variables in the selected node are displayed in the Oz Browser. Other visualization tools may be added. While the first solution found is now the optimal one (the leftmost leaf), we observe that the proof of optimality is still large (including the whole right subtree at the root of the search tree). Hence, we choose edge-finding from a submenu under `Compiler Spec`. We solve the problem again in the Explorer and note that the tree now contains only 29 choice-points, the proof of optimality only 3 choice-points. The solution can be inspected graphically by a so-called Gantt-chart as shown in Figure 4. Tasks on the same resource share the same color and a vertical bar indicates the optimal length (104).

The Oz Scheduler provides also for means to divide the search for a schedule into three phases. In all phases, the search can be interrupted at any time and resumed later on to continue the search at the state of interruption. All parameters are considered in each phase except the search strategy. The provided search for an upper bound uses a local search strategy and displays succesively the length of the current found schedule in the corresponding label. Search for an optimal solution is done by a phase called `Finish`. The search is started at the last found upper bound or a naively computed upper bound, otherwise. This phase must be complete and proves also the optimality of a found solution. Succesively, the last found solution is displayed. The third phase consists in searching for a lower bound. It displays succesively the last found lower bound. A submenu allows to reset stored bounds.

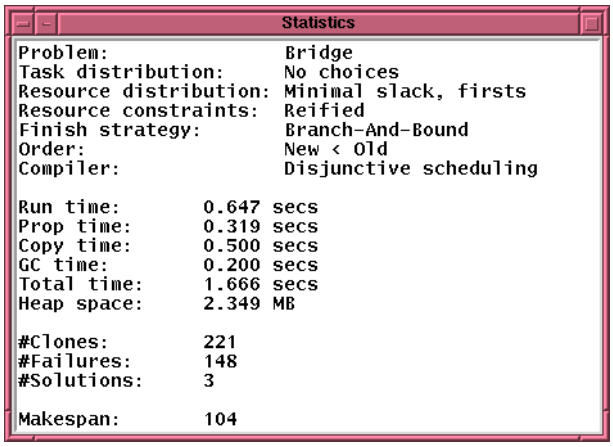

Figure 2. Statistics of the Oz Scheduler

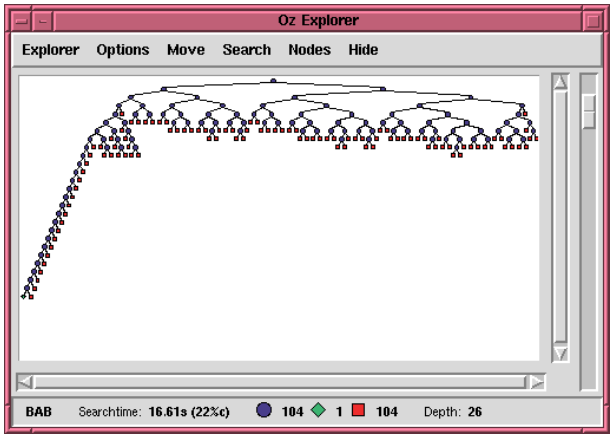

Figure 3. The Oz Explorer

4.2. Implementation

The Scheduler is provided as a concurrent object in Oz. Thus, several instances may be created and encapsulation and state are provided. The state is used to store the parameters, to resume search after interruption, and to coordinate the different possible phases of search. It is important that objects are concurrent, because search must be interruptable, i.e., search and the Scheduler must run in different threads of computation.

The overall structure of the Oz Scheduler is shown in Figure 5. The user specifies the way a problem should be solved by selecting the distribution strategy, the search strategy and the propagators for the resource constraints. These procedures, which correspond to the selected parameters (like the distribution strategy) are stored directly in the state of the Scheduler. In order to parameterize the scheduling

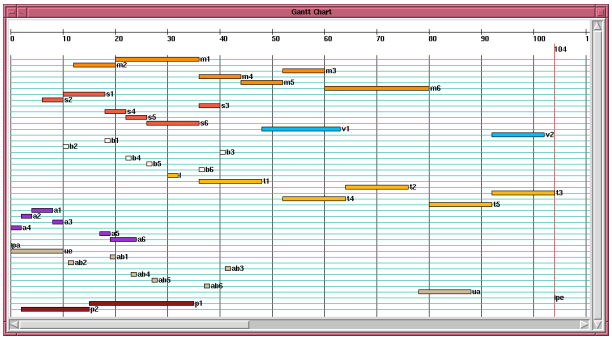

Figure 4. A Gantt-chart for the optimal solution of the bridge problem

problem with the selections made, a constraint problem in form of a procedure is generated dynamically by a scheduling compiler, which is itself an Oz procedure and knows the parameters. This is in contrast to an approach where the user writes statically different programs, which are just solved through an interface. The key to dynamic creation of constraint procedures is higher-order programming in Oz, i.e., functions and procedures can serve as values of data structures, arguments of procedures and return values.

Input of the search is a unary procedure, which serves as a query. If search or the Explorer are invoked, the scheduling compiler is applied with the selected parameters as arguments (the distribution strategy and resource constraints), which returns the query. The argument of the query may be bound to an arbitrary data structure containing e.g. the start times. The query hosts at least the resource constraints and calls to the distribution strategies (concluded by the selected submenus). The query is then passed to the search or Explorer, where it is applied.

All parameters including scheduling compiler and search phases can be extended by sending appropriated messages to the Oz Scheduler. The extensions are then available through new menu entries. The only data structure, which all scheduling compilers must make available (for statistics) is a record containing the start times of the tasks.

4.2.1 The Scheduling Compiler

For scheduling problems with unary capacity, a problem description is a record containing a datastructure and a procedure. The datastructure specifies the problem by stating duration, predecesors and used resources of the occurring tasks. The procedure hosts calls to the additional constraints. To impose the appropriated propagators, this procedure is parameterized by two records mapping the task names to start times and durations.

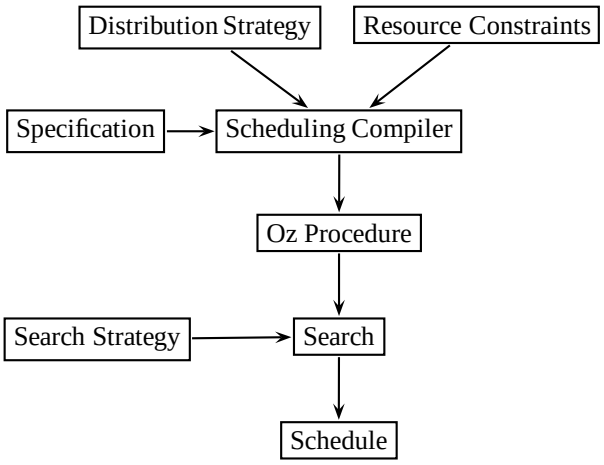

Figure 5. Structure of the Oz Scheduler

The scheduling compiler additionally does some preprocessing like extraction of the records mapping task names to start times and durations, or grouping of tasks scheduled on the same resource. It also hosts the applications of the additional constraints (stored in the problem description) and the precedence constraints.

4.2.2 Resource Constraints

Additionally to reified constraints and constructive disjunction [25], two global constraints are provided. The first compresses the set of reified constraints for tasks on the same resource into one propagator, saving memory. The second constraint uses the algorithmic ideas of [16] (basing on the principles illustrated in Section 3), to implement a propagator. It uses a (relatively) easy technique to construct the sets of tasks S, on which edge-finding is applied. The propagator is improved by incorporating the propagation available through reified constraints for each task pair. This propagator is implemented through a C++ interface, because the algorithm is implemented more efficiently in a language providing destructive datastructures.

4.2.3 Distribution Strategies

Currently, many distribution strategies are provided, among them the successful strategies of [10, 7, 4]. The best results for the benchmarks in Section 4.3 were obtained with a variant of the strategy proposed in [7]. While choosing the constraints to distribute with, this strategy also does some edge-finding. For all task pairs (t_1, t_2), scheduled on the same resource, the set S (called *task interval*) of tasks scheduled between the earliest start time of t_1 and the latest completion time of t_2 is computed. If there exists exactly one task to be scheduled first (resp. last), the corresponding constraints can be imposed deterministically without distribution. We improved the algorithm in that such information is imposed by dynamically added propagators. By implementing the distribution strategy also through the C++ interface, the provided atomicity helps to avoid the maintenance of data structures, which is necessary in [7]

4.2.4 Search

Some of the search procedures offered (like branch&bound or depth-first one solution search) are included in the libraries of Oz. Because search is programmable in Oz, strategies like limited discrepancy search[12] can be implemented in a straightforward way.

In the phase to find upper bounds, essentially the local search techniques used in [9] are applied. Firstly, a good initial solution is computed by a kind of greedy algorithm. The initial solution is then optimized locally. I.e., parts of the previous solution are kept (for example, the serialization of one resource), while the rest is rescheduled with the constraint to find a better solution than the previous one. For each iteration to find a better solution, only a limited number of failures is allowed. If the number of iterations exceeds a specified limit, the phase is stopped (if no user-interruption took place before). This technique and the injection of parts of the previous solution into a search problem, is provided by programmable search in Oz [21].

The phase to find lower bounds LB, is based on binary search. The initial interval ranges from the schedule length found by propagation only (no distribution) to the length of the last found solution in the upper bound phase or a naively computed bound, otherwise. The current LB is the left bound of the interval. The interval is split in the middle until LB+1 is the optimal solution.

The finishing phase is implemented with branch&bound search. An arbitrary cost function (selected in the Oz Scheduler) may be used to order different solutions. The phase stops, if the optimal solution is found and the optimality is proved.

Note that in all cases the search for a schedule is parameterized by the selected submenus. The user can also extend the Scheduler to deal with new kinds of search phases by sending it a message. This message must contain an Oz class, which provides at least the methods `start`, `stop`, and `resume` with appropriated arguments.

4.2.5 Extensions

The Scheduler can be extended dynamically to further problem classes like multiple capacitated resources, variable durations or periodic tasks. This is supported by the modular

structure of the Scheduler and the orthogonality of propagation, distribution and search in Oz. For multiple capacitated resources we already have generalized our edge-finding algorithm and plan to integrate it into the Oz Scheduler.

4.3. Performance Evaluation and Related Work

To evaluate the performance of the Oz Scheduler, we choose ten instances of 10x10 job-shop problems used by Applegate and Cook in [2]. For all problems the optimal solution (starting with no information) has to be found and the optimality has to be proved. First, the upper bound phase is started, which terminates if a fixed number of iterations is reached. Then, the finishing phase is started, which terminates if the last solution found is proved to be optimal. Table 1 and Table 2 contain the results. *Problem* denotes the problem instance in [2], *Fails* the number of failures for the overall search (including the proof of optimality), *CPU* the corresponding runtime in seconds on a Sparc20/70 MHz workstation, and *Fails(pr)* and *CPU(pr)* the number of failures and the time needed for the proof of optimality only. For both phases, the distribution strategy described in Section 4.2 is used. For Table 1, reified constraints were used for the resource constraints[2], while for Table 2, edge-finding was used.

	Reified			
Problem	Fails	CPU	Fails(pr)	CPU(pr)
MT10	5838	169	3983	94
ABZ5	4295	130	2160	52
ABZ6	1737	64	239	5
La19	3798	112	1756	40
La20	4793	129	3247	78
ORB1	20164	554	16252	399
ORB2	2813	86	766	17
ORB3	42327	1071	39405	952
ORB4	6180	172	1939	45
ORB5	3987	114	1499	40

Table 1. Results for the Oz Scheduler

One surprising observation is that reified constraints for the resource constraints in combination with the used distribution strategy are sufficient to obtain good results. While the distribution strategy itself does some edge-finding, this is applied only once per choice-point and its implementation is rather simple. This observation contradicts the conventional wisdom on solving 10x10 job-shop problems, because no really elaborated edge-finding techniques are necessary to solve (at least) these problems.

[2]Except for finding the upper bound for problem ORB1 and ORB3, where edge-finding was used because in this phase, reified constraints produced a too bad schedule length.

	Edge-Finding			
Problem	Fails	CPU	Fails(pr)	CPU(pr)
MT10	4117	157	2564	81
ABZ5	3455	138	1597	52
ABZ6	1508	71	200	6
La19	3331	138	1371	45
La20	6496	228	1943	57
ORB1	14242	521	11775	388
ORB2	2421	99	596	19
ORB3	34422	1121	28232	850
ORB4	3722	140	1340	38
ORB5	3468	138	1155	40

Table 2. Results for the Oz Scheduler

For comparison, ILOG SCHEDULE and Claire are chosen, because they also rely on constraint technology and are comparable to special-purpose OR-algorithms in their efficiency. For these systems the number of backtracks is indicated. Note that for a completely failed binary tree with f failure leafs, the number of backtracks is $2f - 2$.

ILOG SCHEDULE [3] is a commercial C++ library dedicated to scheduling applications. By the combination with ILOG SOLVER, the user can write flexible constraint programs. In principle, a tool like Oz Scheduler can be written in ILOG too, but it would be far more inconvenient due to the rather low level of C++. The number of failures/backtracks for the Oz Scheduler is for the most problems smaller compared to ILOG SCHEDULE (see Table 3, *BT* denotes the number of backtracks). The runtimes are comparable (in [3] a IBM RS6000 workstation was used; further information on this machine is not available from ILOG).

	ILOG SCHEDULE				Claire	
Problem	BT	CPU	BT (pr)	CPU (pr)	BT (pr)	CPU (pr)
MT10	13 684	236	4 735	67	1 575	80
ABZ5	19 303	282	4 519	61	1 350	61
ABZ6	6 227	101	312	5	217	?
La19	18 102	270	6 561	91	1 361	48
La20	40 597	497	20 626	227	2 120	67
ORB1	22 725	407	6 261	108	7 265	315
ORB2	31 490	507	14 123	229	487	23
ORB3	36 729	606	22 138	343	7 500	320
ORB4	13 751	214	1 916	24	1 215	53
ORB5	12 648	211	2 658	37	904	43

Table 3. Results for ILOG and Claire

The programming language Claire [8] is a high-level language to be used in C++ environments. By rules, constraint programming techniques can be modeled. Claire lacks a rich

programming environment. We only compare the proof of optimality, because in [9] the time for the upper bound phase does not include the computation of the initial solution. For the proof of optimality, the number of failures/backtracks and the runtime (they used a Sparc10/40 MHz) are better for Claire (see Table 3). This is due to a more elaborated edge-finding and distribution strategy.

In [2], the proof of optimality for all problems took more than 650 000 nodes in the search tree. Thus, the Oz Scheduler outperforms this approach by more than one order of magnitude.

Remark and acknowledgements I would like to thank Gert Smolka for fruitful discussions on the scheduling compiler and functionality of the Oz Scheduler, and Joachim Walser for valuable comments on a draft version of this paper. The research reported in this paper has been supported by the Bundesminister für Bildung, Wissenschaft, Forschung und Technologie (FTZ-ITW-9105).

References

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

[2] D. Applegate and W. Cook. A computational study of the job-shop scheduling problem. *Operations Research Society of America, Journal on Computing*, 3(2):149–156, 1991.

[3] P. Baptiste and C. L. Pape. A theoretical and experimental comparison of constraint propagation techniques for disjunctive scheduling. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, Montreal, Quebec*, pages 600–606, 1995.

[4] P. Baptiste, C. L. Pape, and W. Nuijten. Incorporating efficient operations research algorithms in constraint-based scheduling. In *First International Joint Workshop on Artificial Intelligence and Operations Research*, 1995.

[5] J. Carlier and E. Pinson. An algorithm for solving the job-shop problem. *Management Science*, 35(2):164–176, 1989.

[6] J. Carlier and E. Pinson. Adjustment of heads and tails for the job-shop problem. *European Journal of Operational Research*, 78:146–161, 1994.

[7] Y. Caseau and F. Laburthe. Improved CLP scheduling with task intervals. In *Proceedings of the International Conference on Logic Programming*, pages 369–383, 1994.

[8] Y. Caseau and F. Laburthe. *Introduction to the CLAIRE programming language*. Laboratoire Mathematiques et Informatique de l'Ecole Normale Superieure, 1994.

[9] Y. Caseau and F. Laburthe. Disjunctive scheduling with task intervals. LIENS Technical Report 95-25, Laboratoire d'Informatique de l'Ecole Normale Superieure, 1995.

[10] C.-C. Cheng and S. Smith. Generating feasible schedules under complex metric constraints. In *Proceedings of the AAAI National Conference on Artificial Intelligence*, pages 1086–1091, 1994.

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

[12] W. Harvey and M. Ginsberg. Limited discrepancy search. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 607–613, 1995.

[13] M. Henz, G. Smolka, and J. Würtz. Object-oriented concurrent constraint programming in oz. In V. Saraswat and P. V. Hentenryck, editors, *Principles and Practice of Constraint Programming*, chapter 2, pages 27–48. The MIT Press, Cambridge, MA, 1995.

[14] M. Henz and J. Würtz. Using oz for college timetabling. In E. Burke and P. Ross, editors, *The Practice and Theory of Automated Timetabling: The Selected Proceedings of the 1st International Conference on the Practice and Theory of Automated Timetabling, Edinburgh 1995*, pages 162–178. Springer Verlag, 1996.

[15] J. Jaffar and M. Maher. Constraint logic programming - a survey. *Journal of Logic Programming*, 19/20:503–582, 1994.

[16] P. Martin and D. Shmoys. A new approach to computing optimal schedules for the job shop scheduling problem. To appear in IPCO V, 1996.

[17] T. Müller and J. Würtz. A survey on finite domain programming in Oz. In *Notes on the DFKI-Workshop: Constraint-Based Problem Solving, To appear as Technical report D-96-02*, 1996.

[18] J. Muth and G. Thompson. *Industrial Scheduling*. Prentice Hall, 1963.

[19] W. Nuijten. *Time and resource constrained scheduling*. PhD thesis, Technical University Eindhoven, 1994.

[20] C. Schulte. Oz Explorer: A visual constraint programming tool. Available from `http://www.ps.uni-sb.de/~schulte/papers.html`, 1996.

[21] C. Schulte, G. Smolka, and J. Würtz. Encapsulated search and constraint programming in Oz. In A. Borning, editor, *Second Workshop on Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, vol. 874, pages 134–150, Orcas Island, Washington, USA, 2-4 May 1994. Springer Verlag.

[22] G. Smolka. The Oz programming model. In J. van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin, 1995.

[23] G. Smolka and R. Treinen, editors. *DFKI Oz Documentation Series*. Deutsches Forschungszentrum für Künstliche Intelligenz GmbH, Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany, 1995.

[24] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. Programming Logic Series. The MIT Press, Cambridge, MA, 1989.

[25] P. Van Hentenryck, V. Saraswat, and Y. Deville. Design, implementation and evaluation of the constraint language cc(FD). In A. Podelski, editor, *Constraints: Basics and Trends*, Lecture Notes in Computer Science, vol. 910. Springer Verlag, 1995.